

Aprendizaje automático de maniobras autónomas de combate aéreo aplicadas a videojuegos

Sebastián Uribe
suribe@dc.uba.ar

Tesis de Licenciatura en
Ciencias de la Computación

DEPARTAMENTO DE COMPUTACIÓN
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
UNIVERSIDAD DE BUENOS AIRES

Director:
Dr. Diego C. Martínez
Universidad Nacional del Sur
dcm@cs.uns.edu.ar

6 de julio de 2011

*Dedicada a mi viejo,
y a mi vieja.
Finalmente terminé.*

Resumen

Las maniobras aéreas de combate (Aerial Combat Maneuvering – ACM) se definen como *el arte de maniobrar un avión de combate para conseguir una posición desde la que se pueda atacar a otro avión*. Requiere muchos años de entrenamiento, práctica y estudios tecnológicos el lograr un buen nivel de experticia en el área.

La importancia suprema de minimizar la exposición humana a riesgos severos ha llevado al desarrollo de varios proyectos para la creación de dispositivos autónomos de navegación, principalmente para usos militares. Los juegos interactivos digitales (videojuegos) hacen uso también, aunque en forma simplificada, de agentes capaces de navegar en forma autónoma, principalmente como contrincantes del jugador.

El objetivo de esta Tesis es el estudio y desarrollo de técnicas de Inteligencia Artificial para el aprendizaje automatizado de navegación en dispositivos aéreos en un entorno simulado simplificado, así como el desarrollo de un entorno de simulación para poder evaluar estas técnicas.

Abstract

Aerial Combat Maneuvering (ACM) is defined as *the art of manoeuvring a combat aircraft in order to attain a position from which an attack can be made on another aircraft*. Lots of years of training, experience and technological studies are required to gain a good expertise.

In order to minimize human exposure to severe risks and death, many autonomous flight devices have been developed, mainly for military use. Interactive digital games (videogames) can also make use, although in simplified form, of agents capable of autonomous navigation, mainly as opponents to the player.

The objective of this work is the study and development of Artificial Intelligence techniques for autonomous training of aerial devices capable of navigating in a simplified environment, and the evaluation of these techniques.

Índice general

1. Introducción	4
1.1. Deficiones previas	5
1.2. El rol de la Inteligencia Artificial	6
1.3. Juegos, videojuegos e Inteligencia Artificial	8
1.3.1. Juegos con oponentes	9
1.3.2. Juegos en tiempo real	10
1.3.3. Juegos en espacios no discretos	10
1.3.4. Inteligencia Artificial y trampa	11
1.4. Disponibilidad de recursos para la IA dentro del juego	13
1.5. Aprendizaje versus programación manual del agente	14
1.6. Organización de la Tesis	15
1.7. Resumen	15
2. Aprendizaje por Refuerzo	17
2.1. Orígenes	17
2.2. Exploración vs. Explotación	17
2.3. Problemas de decisión de Markov	18
2.4. Políticas y funciones de valor	19
2.5. Algoritmos para Aprendizaje por refuerzo	19
2.5.1. Solución por Programación Dinámica	20
2.5.2. Solución por métodos de Monte Carlo	20
2.5.3. Solución por métodos de Diferencias Temporales	21
2.6. Resumen	23
3. Antecedentes	24
3.1. Juegos abstractos	24
3.2. Juegos en tiempo real	25
3.3. Resumen	29
4. Entorno y herramientas de simulación	30
4.1. Entorno de la simulación	30
4.2. Desarrollo de la herramienta	31
4.3. Visualización	32

4.3.1.	Visualización de la Simulación	33
4.3.2.	Visualización de la Memoria	33
4.4.	Resumen	36
5.	Desarrollo del agente autónomo	38
5.1.	Aprendizaje con Q-learning	38
5.2.	Estimación de la función Q con suavizado por núcleos	39
5.3.	Implementación	40
5.3.1.	Características del entorno	40
5.3.2.	Sensado del agente	40
5.3.3.	Simulación	42
5.3.4.	Comportamiento del agente	43
5.3.5.	Actualización de la memoria	44
5.3.6.	Actualización de los vecinos	44
5.3.7.	Elección de la acción óptima	45
5.3.8.	Cálculo de acción por combinación convexa	46
5.3.9.	Estimación de Q con suavizado por núcleos	47
5.4.	Resumen	47
6.	Experimentos y resultados	48
6.1.	Diseño general del entrenamiento	48
6.2.	Resultados	49
6.2.1.	Experimento preliminar	49
6.2.2.	Resultados en un entorno acotado	53
6.2.3.	Evolución del aprendizaje en un entrenamiento	56
6.2.4.	Resultados en dimensiones mayores	64
6.3.	Resumen	66
7.	Conclusiones, observaciones y trabajos futuros	67
7.1.	Resultado final del trabajo	67
7.2.	Conclusiones	68
7.3.	Implementación dentro de un juego	69
7.4.	Trabajos futuros	69
7.4.1.	Implementar las relaciones de Homotecia y Simetría	69
7.4.2.	Determinar automáticamente el valor óptimo de vecindad	70
7.4.3.	Optimizar la representación de la memoria para consultas únicamente	70
7.4.4.	Determinar automáticamente las transformaciones de variables al espacio de memoria	70
7.4.5.	Entrenar para distintos niveles de dificultad	71
7.4.6.	Aprender tanto del agente como del oponente	71
7.4.7.	Meta-Aprendizaje	71
7.4.8.	Detección temprana de proficiencia al entrenar	72

7.4.9. Paralelización del entrenamiento	72
7.4.10. Optimizaciones	72
7.4.11. Tiempo Variable de ejecución	73
Bibliografía	73
Índice de Figuras	77
Índice de Cuadros	78

Capítulo 1

Introducción

Las maniobras aéreas de combate (Aerial Combat Maneuvering – ACM) se definen como *el arte de maniobrar un avión de combate para conseguir una posición desde la que se pueda atacar a otro avión*. Requiere muchos años de entrenamiento, práctica y estudios tecnológicos el lograr un buen nivel de experticia en el área.

La importancia suprema de minimizar la exposición humana a riesgos severos ha llevado al desarrollo de varios proyectos para la creación de dispositivos autónomos de navegación, principalmente para usos militares. Los juegos interactivos digitales (videojuegos) también hacen uso, aunque en forma simplificada, de agentes capaces de navegar en forma autónoma, principalmente como contrincantes del jugador.

Dentro de los recursos con que cuenta un piloto de combate se encuentran las llamadas *maniobras de combate aéreo*, consistentes en secuencias de acciones que le permiten obtener una situación ventajosa cuando se encuentra en combate cercano con uno o más contrincantes. El objetivo de las maniobras puede ser defensivo u ofensivo, y lo que se busca generalmente es conseguir una posición ventajosa (*solución de tiro*) que permita disparar al oponente, evitando que el oponente haga lo propio.

Existen decenas de maniobras básicas y avanzadas, y se espera que un piloto experimentado sepa aplicar la maniobra adecuada teniendo en cuenta características del contexto, como capacidad de los aviones, altura, posición y velocidad actual, etc. En la Figura 1.1 pueden verse tres ejemplos de maniobras: a) El *barril* es una maniobra básica que funciona como base de otras maniobras, y consiste en generar una elevación y giro completo del avión, finalmente volviendo al rumbo original. Puede usarse para reducir la velocidad del avión o para evitar que un oponente obtenga una solución de tiro, entre otros usos. b) El *Giro de Immelmann* (cuya creación se atribuye al piloto Alemán de la 1^{ra} Guerra Mundial, Max Immelmann [Imm34]) consiste en una elevación, al igual que el barril, pero al llegar al punto superior del giro (cuando el avión se encuentra cabeza abajo) se mantiene el rumbo y se

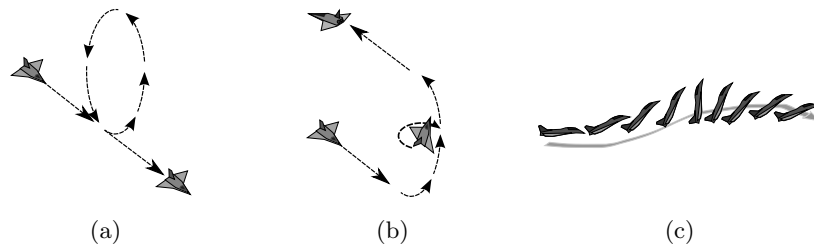


Figura 1.1: Ejemplos de maniobras aereas: Barril (loop), giro de Immelmann, Cobra de Pugachev

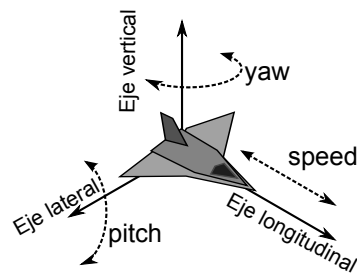


Figura 1.2: Ejes del avión y posibles dimensiones del movimiento

realiza un giro sobre el eje longitudinal del avión (Figura 1.2) para volver a una posición cabeza arriba pero en la dirección contraria a la que tenía al iniciar la maniobra. c) La *Cobra de Pugachev* (bautizada en honor al piloto Soviético Viktor Pugachev [Abs11]) consiste en elevar la trompa del avión hasta ubicarlo en una posición vertical mientras vuela a baja velocidad, y volver rápidamente a la posición inicial, perdiendo en el proceso poca altura. Esta maniobra permite limitar momentáneamente el avance del avión, aunque su utilidad en combate es acotada.

1.1. Deficiones previas

A lo largo de esta Tesis se usarán las siguientes definiciones y abreviaciones:

IA: Inteligencia Artificial. Puede referirse al campo en su totalidad, o a la parte del software o videojuego que toma las decisiones de agentes que el jugador no controla.

Agente: El individuo que es de interés y es controlado por la IA, en particular en este trabajo se trata del objeto del entrenamiento.

NPC: Non-Playable Characters (Personaje no controlado por el jugador). Puede tratarse tanto de individuos que acompañan y/o ayudan al jugador

como de contrincantes o individuos neutrales, con los que interactúa brevemente. Por su naturaleza misma son el objeto de control de la IA en el juego.

AR: Aprendizaje por Refuerzo [Sut98]. Algoritmo de aprendizaje no supervisado basada en la observación del entorno y una función de feedback sobre cada acción ejecutada.

GPU: Graphics Processing Unit (Unidad de procesamiento gráfico). El procesador que se encarga de manipular y mostrar los elementos de interfaz visual, sean en dos o tres dimensiones. Los procesadores gráficos modernos tienen una complejidad similar al procesador central, y una capacidad de cómputo en paralelo superior para varios tipos de tareas.

GPGPU: General Purpose computation on Graphics Processing Unit (Computación de propósito general en la unidad de procesamiento gráfico). El uso del GPU para realizar operaciones no relacionadas al procesamiento de gráficos, como las que pueden realizarse en un procesador de propósito general.

FPS: First Person Shooter (Juego de disparos en primera persona). Género de juegos consistente en controlar un avatar con una visión subjetiva (desde los ojos del personaje / en primera persona) y generalmente disparar contra gran cantidad de enemigos en un entorno tridimensional.

FSM: Finite State Machine (máquina de estados finitos). Dada su simpleza conceptual y de implementación, son usadas frecuentemente para el desarrollo de IA en videojuegos aplicada al comportamiento de individuos.

HFSM: Hierarchical Finite State Machine (máquina de estados finita jerárquica) [Har87]. FSMs basadas en una estructura jerárquica, donde los estados de más alto nivel contienen sub-FSMs que comparten alguna característica que justifica su agrupamiento. Son muy usadas en FPSs y generalmente la jerarquía de estados.

RTS: Real Time Strategy (Estrategia en tiempo real). Género de juego que consisten en hacer uso de una cantidad generalmente finita de recursos e individuos en un espacio geográfico limitado con el objetivo de vencer al contrincante en una batalla. Funcionan en tiempo real, es decir, el jugador y la IA operan en simultáneo dando instrucciones a los individuos, quienes las ejecutan continuamente. En estos juegos son importantes los elementos de administración de recursos además del control de los individuos. Ejemplos: *Starcraft*, *Warcraft*, *Command & Conquer*.

1.2. El rol de la Inteligencia Artificial

La investigación académica en Inteligencia Artificial ha comenzado a influir fuertemente en el desarrollo de videojuegos en la última década, aunque el área de los simuladores de vuelo ha recibido muy poca atención. Un atractivo importante es el poder automatizar la generación de personajes no con-

trolados por el jugador (Non-Playable Characters, NPC), disminuyendo el costo de autoría de los juegos. Un NPC es el equivalente en juegos a un dispositivo autónomo en una situación del mundo real.

La investigación vinculada a la industria se concentra mayormente en géneros como los juegos de acción en primera persona ([Gra04], [Ben10], [Mor10], [Pat10]) o los juegos de estrategia en tiempo real ([Con10], [Cun10], [But10], [Smi10]), por ser los géneros más populares de videojuegos actualmente, e inclusive cuentan con competencias cuyo objetivo es avanzar el estado del arte (AIIDE 2011 Starcraft Competition ¹, 2K BotPrize ²). Otros géneros, como los simuladores de vuelo, no han recibido tanta atención, a pesar de que pueden también verse beneficiados con trabajos académicos.

Diversas áreas de Inteligencia Artificial son de interés aplicativo a los videojuegos, como la representación de conocimiento y razonamiento, búsqueda heurística, planificación, razonamiento espacial y temporal, comportamiento colaborativo entre otros. Recientemente se han usado con éxito aprendizaje por refuerzo (AR) y redes neuronales evolutivas para lograr aprendizaje autónomo en NPCs, permitiendo a los agentes aprender el comportamiento correcto en un escenario virtual.

El problema principal al aplicar aprendizaje automático en videojuegos es la *maldición de la dimensionalidad*, definición de Bellman [Bel57] para referirse a aquellos problemas cuya complejidad computacional crece exponencialmente con el número de variables de estado (dimensiones) a tratar. Esta dimensionalidad se ve generalmente en dos aspectos: espacio de estados y espacio de acciones [Rus95]. Por ejemplo, en los juegos de estrategia hay que tener en cuenta en cada momento la posición de cientos de individuos, además de las características del terreno de juego y factores como recursos disponibles y campo de visión de cada unidad. Al tomar una decisión, la cantidad de posibles órdenes a cada unidad es muy grande dado el espacio de estados y acciones.

Esta dimensionalidad puede ser alterada dependiendo del nivel de abstracción de la información que se envíe al agente. Por ejemplo, si se usa la visión para percibir objetos y obtener sus características, será necesario procesar una imagen, detectar los objetos en ella, y luego actuar en función de esos objetos detectados. Este procesamiento de la imagen puede hacerse dentro del agente, o puede considerarse que está dado por el entorno, y al agente le llegan únicamente las características detectadas del objeto. En el primer caso, el agente deberá aprender a partir de una cantidad de información mucho mayor (supongamos, los píxeles de la imagen) mientras que en el segundo podrá trabajar con una cantidad más acotada de variables (asumiendo que la cantidad de características del objeto es menor a la cantidad de píxeles en la imagen). En definitiva, los datos que se procesan en

¹<http://skatgame.net/mburo/sc2011/>

²<http://botprize.org>

total, en ambos casos, son los mismos; la diferencia está en dónde se traza la división entre agente y entorno, lo que influirá en la cantidad de información que usará para su aprendizaje.

Algunos géneros de juegos, como el combate aéreo entre dos oponentes, pueden ser simplificados a un conjunto de variables tratables por las técnicas de aprendizaje automático, sin necesidad de reducir la calidad de la simulación o quitar elementos importantes del juego.

El aprendizaje automático se ha aplicado en casos “de juguete” o estrictamente académicos (como NERO - Neuro Evolving Robotic Operatives, de la Universidad de Texas) o en forma experimental en videojuegos simples (como Tao Feng). En el primero se usan redes neuronales evolutivas, y en el segundo aprendizaje por refuerzo. La escasa existencia de investigación aplicada a juegos de combate aéreo puede explicarse debido a que es un género cuya popularidad decreció considerablemente antes de que comenzaran las investigaciones de Inteligencia Artificial aplicadas a videojuegos. A pesar de esto, creemos que es un género útil para aplicar aprendizaje automático e intentar extender lo desarrollado a otros géneros.

El objetivo de esta Tesis es el estudio y desarrollo de técnicas de Inteligencia Artificial para el aprendizaje automatizado de navegación en dispositivos aéreos en un entorno simulado simplificado, así como el desarrollo de un entorno de simulación para poder evaluar estas técnicas. El agente a entrenar tendrá como objetivo derribar a un contrincante en este entorno simulado y evitar que el contrincante lo derribe.

1.3. Juegos, videojuegos e Inteligencia Artificial

Existe una larga historia de uso de Inteligencia Artificial en juegos de computadora. Los juegos clásicos de tablero como el ajedrez o damas, fueron de los primeros problemas que los investigadores de IA atacaron, dado que presentan algunas características que los hacen especialmente atractivos:

- Poseen un espacio de estados acotado, fácil de definir, accesible (completamente visible), y a la vez difícil de abarcar en forma completa (en todos sus estados posibles). Esto obliga a encontrar estrategias que usen conocimiento parcial del estado y hace difícil encontrar una solución por fuerza bruta, salvo en casos relativamente simples.
- Poseen un conjunto de acciones también acotado y fácil de definir, y una transición entre un estado y otro a partir de la acción del jugador (o jugadores) clara y definida. En algunos casos se ve involucrada una dosis de azar (generalmente un dado, o un oponente), pero no es complicada de representar como una acción o entrada más al sistema.

- Los juegos de por sí son muchas veces abstracciones. Por ejemplo, puede verse al ajedrez como una simplificación y abstracción de un campo de batalla. Pero también son *cosas* en si mismas, dignas de ser representadas en la computadora y analizadas. Dado que esta representación se puede hacer en forma perfecta o casi perfecta en la computadora, se está trabajando con un problema del mundo real, palpable.
- Finalmente, la capacidad para jugar juegos abstractos puede ser considerada como propia de un ser inteligente, y la IA necesitó a lo largo de casi toda su historia problemas de este tipo para lograr credibilidad en la comunidad científica en general.

El primer caso de desarrollo de una inteligencia artificial para jugar un juego es probablemente el programa de damas de Christopher Strachey desarrollado en 1951 en la Universidad de Manchester para la Ferranti Mark I. También para esa computadora y en el mismo año se desarrolló el primer programa de ajedrez, de Dietrich Prinz que a pesar de no poder jugar una partida completa, podía resolver el problema del mate en dos (dar jaque mate en una configuración de tablero si es posible darlo en dos movidas).

El ajedrez ha sido probablemente el juego que más investigación ha atraído, llegando quizás a su punto más importante en 1997 con la victoria de Deep Blue³ sobre el entonces campeón mundial de Ajedrez, Garry Kasparov. Aunque han existido sospechas de que Deep Blue hacía uso de “fuerza bruta” más que de inteligencia, no deja de ser un hito importante y una reafirmación de la utilidad de las técnicas empleadas.

El juego de damas no representa ya un desafío, dado que en el año 2007 se resolvió completamente el juego al computar todas las posibles movidas, y determinando que si ambos jugadores no cometen errores, se termina en empate. [Sch07]

El backgammon es otro juego que ha recibido mucha atención, sobre todo usando el método de *Temporal-Difference Learning* ([Tes95]), al punto que su uso generó cambios en la forma de analizar el juego en jugadores profesionales.

Quizás el juego de tablero que aún es un desafío y que obliga a desarrollar y probar nuevas técnicas es el Go. Los mejores algoritmos conocidos consiguen vencer a jugadores profesionales en tableros pequeños o teniendo ventaja inicial [Lee09].

1.3.1. Juegos con oponentes

Existen juegos que pueden ser jugados por una persona en forma individual (Solitarios de cartas, Puzzles) y otros que requieren de dos o más

³<http://www.research.ibm.com/deepblue/>

jugadores (Damas, Ajedrez, Go). En estos últimos el objetivo del algoritmo es encontrar las acciones óptimas que le permitan a un jugador ganar a pesar de las acciones que elija su oponente.

Para los juegos por turnos a veces es posible plantear el problema uno de búsqueda en un grafo, donde cada estado del juego es un nodo y cada acción de un jugador, un eje (alternando acciones entre jugadores). Es posible entonces evaluar (cuantificar numéricamente) el estado actual del juego para cada jugador, usando algoritmos de tipo *minimax* para encontrar la mejor acción a tomar en cada momento. Habitualmente hay dos problemas: Definir una función de valuación para el estado, y representar los posibles estados y transiciones entre ellos en memoria. El primero puede ser resuelto en forma aceptable con mayor o menor esfuerzo, pero el segundo en algunos casos (como el Go) resulta imposible.

1.3.2. Juegos en tiempo real

Los juegos de computadora abarcan mucho más que las representaciones digitales de juegos de tablero ya existentes. Existen infinidad de juegos abstractos (aquellos cuyos elementos constitutivos son abstracciones de cosas reales, como los de tipo *Tower Defense*⁴, o aquellos donde se opera sobre objetos abstractos que no representan nada del mundo real, como el Tetris⁵) creados específicamente para el medio, además de juegos consistentes en simulaciones mucho más complejas, que distan mucho de los juegos de tablero. La investigación de IA en juegos se ha concentrado clásicamente en los juegos abstractos y de tablero, pero hoy día la mayor parte de los videojuegos existentes son complejos y en tiempo real, en los que el jugador debe continuamente tomar decisiones que afectan el curso del juego, operando en un entorno dinámico que evoluciona inclusive sin su intervención.

En estos juegos resulta importante que el agente pueda tomar decisiones en forma rápida aún con información parcial del entorno, dado que generalmente está compitiendo contra otros agentes o contra el jugador para llegar a un estado deseado (ganar el juego).

1.3.3. Juegos en espacios no discretos

Otra característica de gran parte de los videojuegos es que simulan entornos y situaciones no discretas, sea en espacio o en tiempo. A pesar de que las computadoras no pueden trabajar sino con representaciones digitales (y por lo tanto discretas) de información, lo que los juegos intentan es dar la impresión al jugador de un espacio o tiempo continuo.

Existen juegos en espacios discretos completamente observables que de todas formas resultan difíciles para las computadoras. El caso más clásico

⁴http://en.wikipedia.org/wiki/Tower_defense

⁵<http://www.tetris.com/>

es el Go, ya mencionado, donde las posibles combinaciones de piezas válidas sobre el tablero son demasiadas como para tratarlo por fuerza bruta (probando todas las combinaciones posibles), y solo recientemente se han encontrado buenas heurísticas. Queda claro que en un espacio continuo y aunque se haga uso de una cantidad menor de variables, la representación completa de todas las combinaciones de estados es imposible de tratar.

1.3.4. Inteligencia Artificial y trampa

Los agentes de IA deben usar su percepción del entorno para la toma de decisiones en general. Muchas veces, dado que la capacidad del agente es acotada, el juego hace “trampa” y le permite percibir partes del entorno que no deberían poder dado su estado actual. Esto ocurría en muchos juegos de estrategia por computadora en los principios del género, cuando la capacidad de procesamiento de las computadoras era mucho menor y no era posible implementar algoritmos que dieran buenos resultados y consumieran pocos recursos. Algunos ejemplos clásicos de trampa de la IA:

- Muchos juegos de estrategia en tiempo real implementan un mecanismo llamado *neblina de guerra* (*fog of war*) que impide a cada jugador ver lo que ocurre en lugares donde no tiene unidades presentes. Una forma de trampa es que la IA pueda ver a pesar de la neblina, obteniendo información de las unidades del jugador que le permitirían planificar sus ataques en forma efectiva. Es una queja frecuente de los jugadores del *Command & Conquer* ⁶ el hecho de que la IA ignora la neblina de guerra.
- En juegos de carreras de autos, es frecuente el uso de *IA elástica* (*rubberband AI*), que impide que el jugador se aleje demasiado de los autos que controla la computadora. Cuando el jugador se adelanta mucho, los autos controlados por la IA aceleran a velocidades que no serían posibles para el jugador; cuando éste se atrasa, desaceleran hasta una velocidad que permita al jugador alcanzarlo. El juego *Mario Kart* es conocido por implementar este mecanismo ⁷.
- En juegos de pelea, como el *Mortal Kombat* ⁸, es importante conocer y ejecutar con precisión combinaciones de botones para lograr los golpes y defensas adecuadas, y para casi todo ataque existe una forma específica y puntual de contrarrestarlo. Es común que a medida que aumenta el nivel de dificultad en el juego, algunos oponentes controlados por la IA adquieren una capacidad casi infalible para interceptar

⁶<http://www.commandandconquer.com>

⁷<http://www.mariokart.com>

⁸<http://www.themortalkombat.com/>

los golpes del jugador humano y contrarrestarlos con la maniobra adecuada. En un jugador humano, esto requeriría de precognición, pero para el programa resulta simple emplear la información en forma inmediata. Estos oponentes resultan casi imposibles de vencer y generan frustración en los jugadores.

Generalmente se considera una mala práctica permitir que la IA haga uso de información que no correspondería a un oponente humano en las mismas condiciones de juego. No hacerlo produce un resultado que no es justo para el jugador, ni equilibrado en términos de posibilidades para cada parte. Por lo tanto, es imperativo desarrollar IA que, a partir de las mismas limitaciones y posibilidades que tendría un jugador humano en sus condiciones, pueda jugar una partida de forma adecuada y representar un desafío interesante y entretenido para el jugador humano, sin hacer trampa.

Soren Johnson, desarrollador de *Civilization III*⁹ y *Civilization IV*¹⁰ dice, sobre las expectativas de los jugadores con respecto a la IA¹¹:

The worst feeling for a player is when they perceive – or just suspect – that a game is breaking its own rules and treating the human unfairly. This situation is especially challenging for designers of symmetrical games, in which the AI is trying to solve the same problems as the human is. For asymmetrical games, cheating is simply bad game design. ... However, under symmetrical conditions, artificial intelligence often needs to cheat just to be able to compete with the player.

Lo peor que le puede pasar a un jugador es cuando percibe - o simplemente sospecha - que el juego está rompiendo sus propias reglas y tratándolo injustamente. Esta situación es especialmente desafiante para los diseñadores de juegos simétricos, en los que la IA intenta resolver el mismo problema que el humano. Para juegos asimétricos, hacer trampa es simplemente mal diseño. ... Sin embargo, bajo circunstancias simétricas, la inteligencia artificial necesita hacer trampa frecuentemente para poder competir contra el jugador.

Al momento de diseñar un mecanismo de entrenamiento, es importante entonces que aprenda a jugar sin más información que la que tendría un jugador humano en sus mismas condiciones, incluyendo el tipo de estrategia de su oponente, o las acciones que tomará a continuación.

⁹<http://www.civ3.com/>

¹⁰<http://www.2kgames.com/civ4/home.htm>

¹¹http://www.gamasutra.com/view/news/25127/Analysis_Game_AI__Our_Cheatin_Hearts.php

1.4. Disponibilidad de recursos para la IA dentro del juego

Tradicionalmente, en los videojuegos, el mayor tiempo de cómputo está dedicado a procesar datos visuales y de simulación (física, interacción entre elementos, etc.), y se le dedica a la IA un porcentaje muy bajo del tiempo de procesador (alrededor de un 10% ¹²). En el pasado esto resultaba en una capacidad muy limitada para la IA, especialmente en juegos donde el aspecto visual era el más importante.

Dos factores han modificado el panorama desde entonces. Primero, la mayor capacidad de procesamiento de las placas gráficas y la posibilidad de derivar en ellas gran parte del trabajo que antes se realizaba en el procesador central, permitió que se use más tiempo de éste para el procesamiento de IA. Inclusive es posible usar al procesador gráfico para realizar cálculos para la IA (GPGPU).

El segundo factor importante es la mayor capacidad de procesamiento de los procesadores centrales. El aumento exponencial en capacidad de cómputo significa que aunque se dedique el mismo porcentaje de su uso para la IA que hace 20 años, este porcentaje permite realizar una cantidad de cálculos órdenes de magnitud mayor. A esto se debe sumar la capacidad de memoria de las computadoras, que también aumentó en órdenes de magnitud a lo largo de las décadas.

Esta nueva capacidad aumentada de cómputo dentro del juego permite a la IA, entre otras cosas:

- El uso de algoritmos más sofisticados, que requieren más capacidad de procesamiento y memoria para llegar a buenos resultados, y que no requieren conocer partes del mundo que deberían ser en realidad desconocidas para el individuo simulado.
- La posibilidad de que la IA encuentre sola y bajo demanda la solución a algún problema, en vez de que los desarrolladores del juego deban invertir tiempo en resolverla previamente. Por ejemplo, para *pathfinding* (búsqueda de caminos), es posible pre-programar rutas entre distintos puntos de antemano, pero también es posible dejar que la IA encuentre sola el mejor camino cuando necesite llegar de un punto a otro. La ventaja en este caso es que requiere menos tiempo de desarrollo, y que es menos probable que se presenten errores durante el juego debido a problemas de diseño en el entorno o falta de coordinación entre los puntos de navegación y el diseño del entorno.
- La posibilidad de que la IA aprenda durante el juego. Este aprendizaje puede ser para adaptarse al estilo de juego del jugador, o como una

¹²http://www.gamasutra.com/view/feature/3371/game_ai_the_state_of_the_industry.php

característica intrínseca del juego. Ejemplo de esto son juegos como *Creatures*¹³ y *Black & White*¹⁴, en los que el entrenamiento de una criatura virtual es parte del objetivo del jugador.

1.5. Aprendizaje versus programación manual del agente

En la mayoría de los videojuegos la IA es desarrollada por un programador, y se usan herramientas como Máquinas de estados finitos, Grillas de navegación, Árboles de comportamiento, entre otras, dependiendo del problema que estén resolviendo. Podríamos calificar a esto como *programación manual* de la inteligencia artificial.

Una ventaja de desarrollar manualmente la IA es que se puede definir el comportamiento del agente para que abarque todas las situaciones particulares que el desarrollador pueda considerar necesarias, incluyendo tanto detalle de su accionar como sea requerido. También vuelve posible analizar el comportamiento del agente y entender por qué se comporta como lo hace en cada momento.

Pero este desarrollo manual presenta algunos inconvenientes también, los cuales pueden separarse en dos grupos: problemas de complejidad, y problemas por errores de diseño o coordinación. Los problemas de complejidad tienen que ver con que cuanto más complejo es el juego, más se notarán las falencias de la IA. Esto lleva normalmente a aumentar el trabajo de codificación de su comportamiento, tornando más complejo y costoso su desarrollo. Los problemas de diseño o coordinación surgen cuando el desarrollo de la IA no está bien coordinado con otros aspectos del desarrollo del juego, como el diseño del entorno o características de los NPC, o cuando la IA no contempla posibles acciones del jugador.

Un ejemplo de estos problemas se puede ver en el caso del *Tao Feng* (que se detallará en el Capítulo 3). En [Gra04] emplearon aprendizaje por refuerzo para entrenar a un agente para el juego, y encontraron una estrategia que aprovecha las falencias de la IA desarrollada manualmente y que le permite ganarle siempre. La industria es generalmente consciente de la existencia de problemas que pueden resultar embarazosos en los juegos¹⁵. Un tipo de problema muy frecuente es el de situaciones en las que la IA repite acciones a pesar de que no tienen ningún efecto real en el juego. (Por ejemplo, disparar contra una pared que separa al jugador del agente) Este tipo de problemas aparecen por usar un sistema basado en máquinas de estados finitos y falta de feedback del entorno cuando el agente realiza una acción. Si se entrenara a los agentes con un sistema basado en refuerzos, requiriendo que optimizaran

¹³http://www.gamewaredevelopment.co.uk/creatures_index.php

¹⁴<http://lionhead.com/Games/BW/>

¹⁵`\url{http://aigamedev.com/open/articles/bugs-caught-on-tape/}`

su comportamiento para maximizar los refuerzos futuros, probablemente evitarían acciones que no aporten resultados al repetirse indefinidamente. En otros casos se trata de falta de previsión de los desarrolladores de que podían ocurrir situaciones específicas, dejando la posibilidad al jugador de que las explote. En estos casos, la impresión del jugador es que la IA no funciona bien, porque no reacciona de una forma *aparentemente razonable* a sus acciones.

Finalmente, y en relación a la complejidad, un aspecto importante es el costo del desarrollo de la IA. A medida que la complejidad de los juegos y el entorno que simulan aumenta, y para solucionar los problemas antes mencionados, aumenta la complejidad de la IA misma. Esto requiere más trabajo e introduce nuevos problemas. Conceptos como el de *Director de IA* se introducen para estructurar esta complejidad, se emplean HFSSM para reemplazar FSM, y se prueban nuevos algoritmos y estructuras de control, pero en definitiva todo lleva a lo mismo: aumentar el trabajo manual de desarrollo de la IA.

1.6. Organización de la Tesis

El trabajo tendrá la siguiente estructura: el Capítulo 2 presentará Aprendizaje por Refuerzo, una forma de modelar y resolver problemas de aprendizaje, que será empleada como solución en esta Tesis. En el Capítulo 3 se presentarán antecedentes del uso de Inteligencia Artificial en juegos. En el Capítulo 4 se explicará el entorno creado para la evaluación del agente y las condiciones en las que se deberá desenvolver. El Capítulo 5 expondrá la técnica usada para entrenar al agente, y en el Capítulo 6 los resultados experimentales. Finalmente el Capítulo 7 cerrará el trabajo con conclusiones, planteo de posibles extensiones y trabajos a futuro.

1.7. Resumen

Ante los problemas planteados y las características actuales en el desarrollo de IA para videojuegos, se vuelve cada vez más interesante el uso de entrenamiento automático de los agentes. Si bien no es una solución para todo género de juego, hay lugares donde podría resultar un aporte útil. En el siguiente capítulo se verán casos en que se han aplicado técnicas de aprendizaje automático en videojuegos, y posteriormente el caso particular de este trabajo.

En definitiva, si entendemos la IA como parte del contenido del juego, tiene sentido buscar formas de automatizar su generación, de la misma forma que ocurre con otros tipos de contenidos. En palabras de los autores del juego *Galactic Arms Race - GAR* (que se mencionará con mayor detalle en el capítulo de antecedentes):

The goal of automated content generation is to both lower the burden of content generation on developers and to make games more compelling for players by maintaining the feeling of novelty and the drive for exploration by continually evolving new content. Such a capability, if proven viable, can have far-reaching implications for the game industry.¹⁶

El objetivo de la generación automática de contenido es tanto disminuir el costo de su generación para los desarrolladores, como el producir juegos más atractivos para los jugadores, al mantener la sensación de novedad y la motivación para la exploración mediante la evolución continua de nuevo contenido. Esta capacidad, si se prueba viable, puede tener implicaciones importantes para la industria de los videojuegos.

¹⁶<http://gar.eecs.ucf.edu/index.php>

Capítulo 2

Aprendizaje por Refuerzo

En esta Tesis se empleará *Aprendizaje por Refuerzo* (AR), una técnica que enfatiza el aprendizaje del individuo a través de la exploración del entorno, para entrenar al agente. En este capítulo se presentará brevemente AR, las principales técnicas empleadas para solucionar los problemas de AR, y al algoritmo que se usará para la resolución del presente trabajo.

2.1. Orígenes

El aprendizaje por refuerzo tiene sus raíces no sólo en las Ciencias de la Computación, sino también en la psicología, estadística y neurociencias [Kae96]. Pueden identificarse dos grandes ramas: por un lado, las relacionadas con el aprendizaje por prueba y error, sustentadas en estudios de animales y psicología humana; por el otro, los problemas de *control óptimo* y su solución empleando funciones de valor y programación dinámica [Sut98]. El área de control óptimo estudia soluciones a problemas de optimización, buscando leyes que permitan llegar a una situación definida como óptima, sujetos a restricciones del entorno. Una técnica usada en control óptimo y que el AR incorpora, es la *programación dinámica* definida por Bellman[Bel57]. El concepto de prueba y error surge de la observación de animales y personas probando distintas acciones ante un problema o situación, hasta llegar a la solución. Esto implica dos cosas: primero, que es posible distinguir acciones a partir de sus consecuencias en el entorno, y segundo que es posible relacionar una acción con su consecuencia.

2.2. Exploración vs. Explotación

A diferencia de los métodos supervisados, en AR no se indica al agente explícitamente cuál es la mejor acción para cada estado, sino cuán bueno es un estado al que llegó luego de tomar la acción. De esta forma, el agente aprende por prueba y error. Esto implica que mientras el agente no tenga

suficiente información de su entorno, deberá necesariamente probar acciones que nunca haya realizado para obtenerla. A su vez, es importante que aprende, *explote* el conocimiento ya adquirido, es decir, que elija con mayor frecuencia las acciones que le reporten mayor beneficio. Un agente que sólo explora no aprenderá suficiente sobre cuáles son los caminos óptimos para llegar a un buen resultado, y un agente que solo explota su conocimiento no conocerá nuevos caminos, potencialmente mejores.

2.3. Problemas de decisión de Markov

En AR, el agente recibe información del entorno sobre su estado, y a partir de ésta decide qué acción tomar. El entorno también entrega al agente una recompensa o *refuerzo* por el estado en el que se encuentra luego de tomada una acción. A partir de esto podrá estimar la suma de recompensas que recibirá a lo largo de sucesivas acciones, e intentar maximizarla. Se denominará S al espacio de estados y A el de acciones.

El estado del agente puede cambiar luego de tomar una acción. Este cambio se puede expresar como una función de probabilidad $P(s, a, s') : S \times A \rightarrow S$ que devuelve la probabilidad de que a partir del estado s , y ejecutando la acción a , se llegue a s' . Si se trata de un entorno *determinístico*, entonces para cada par (s, a) , existe un único estado s' tal que $P(s, a, s') = 1$, y para cualquier otro estado $s'' \neq s'$, se cumplirá que $P(s, a, s'') = 0$. Si el entorno no es determinístico esto no se cumple, pero si se cumplirá que $\forall s, a, \sum P(s, a, s') = 1$. En ambos casos, se espera que la función P no varíe con el tiempo.

Un Problema de Decisión de Markov (*Markov Decision Problem*, MDP) es un formalismo compuesto por una 4-upla (S, T, P, R) , dada por:

- Un espacio de estados S
- Una función $T(s) : S \rightarrow \mathcal{P}(A)$ que indica el conjunto de acciones posibles dado un estado
- Una función de probabilidad $P(s, a, s') : S \times A \rightarrow S$ que indica, dado un estado, acción y otro estado, la probabilidad de pasar del primer al segundo estado ejecutando la acción dada
- Una función $R(s, s') : S \times S \rightarrow \mathbb{R}$ que recibe dos estados e indica la recompensa obtenida si del primero de ellos se transita al otro

Una condición necesaria en un MDP es que la transición de un estado a otro sea dependiente únicamente del estado actual, y no de los estados anteriores. Los MDPs se usan para modelar los problemas de AR.

2.4. Políticas y funciones de valor

El agente decide qué acción tomar a partir del estado en que se encuentra. Esta decisión puede modelarse con una *política* $\pi(s, a) : S \times A \rightarrow [0, 1]$, que es la probabilidad de que estando en el estado s , tome la acción a . Las políticas pueden ser o no determinísticas. El caso en que el agente toma siempre la misma decisión ante un estado s es el de una política determinística, que se define como $\pi(s) : S \rightarrow A$, y es el de interés en esta Tesis.

Esta política debería, en el caso ideal, indicar la acción óptima para cada estado con el fin de que el agente llegue a la solución del problema realizando una secuencia apropiada de acciones. A esta política óptima se la denota π^* .

Para encontrar una política óptima es necesario poder comparar dos estados y determinar cuál es mejor en función de qué tanto acerca al agente a la solución del problema. Para eso se define una *función de valor* $V(s) : S \rightarrow \mathbb{R}$

$$V(s) \leftarrow \sum_a \pi(s, a) \sum_{s'} P(s, a, s') [R(s, s') + \gamma V^\pi(s')] \quad (2.1)$$

Donde P es la función de probabilidad de transición del MDP, R es su función de recompensa, y γ es un *factor de descuento* que se usa para limitar la suma de futuros refuerzos, tal que $0 \leq \gamma \leq 1$. Cuando $\gamma < 1$, se garantiza que la suma, aunque sea de una cantidad infinita de términos, tendrá siempre un resultado finito (siempre y cuando los valores de los refuerzos se encuentren acotados).

La valuación de un estado se define en función de las futuras recompensas que recibirá el agente de encontrarse en él, y si sigue una determinada política. Se denota $V^\pi(s)$ a la función de valor relacionada con la política π . Una política π es mejor que otra π' , si $\forall s, V^\pi(s) > V^{\pi'}(s)$.

El método *Q-learning* ([Wat89]) define una función $Q(s, a) : S \times A \rightarrow \mathbb{R}$ para evaluar un estado. $Q^\pi(s, a)$ devuelve la recompensa esperada a partir del estado s y tomando la acción a sumada a las recompensas acumuladas siguiendo la política π desde el estado al que se llega luego de ejecutar a . Tanto el valor de V^π como de Q^π pueden ser determinados en función de la experiencia del agente. La relación entre V^π y Q^π queda definida por

$$V^\pi(s) = \max_{a \in A(s)} Q^\pi(s, a) \quad (2.2)$$

2.5. Algoritmos para Aprendizaje por refuerzo

Existen distintas formas de resolver problemas de aprendizaje por refuerzo, separados en tres grandes ramas: Programación Dinámica, métodos

de Monte Carlo, y Diferencias Temporales. A continuación se explicarán brevemente las tres¹.

2.5.1. Solución por Programación Dinámica

Los algoritmos de Programación Dinámica (PD) permiten obtener políticas óptimas para un MDP. Estos algoritmos se basan en aprender π y V^π a partir de la exploración de todos los posibles estados y acciones. Requieren conocer el estado completo del entorno, lo que puede resultar impracticable por su complejidad en tiempo, o simplemente imposible para entornos en los que no hay visibilidad completa del estado.

Se comienza con una política inicial y su función de valuación correspondiente π_0, V^{π_0} , y mejorarla mediante un proceso iterativo hasta encontrar la política óptima. En la práctica esto requeriría una cantidad infinita de iteraciones, por lo que se define un criterio de detención, que puede estar dado por la evaluación de la máxima diferencia entre un estado en dos sucesivos pasos de la iteración. Es decir, se detiene si $\max_{s \in S} |V_{k+1}(s) - V_k(s)| < \delta$ para un δ determinado.

Teniendo una forma de calcular la función de valor, y con ella evaluar una política, sólo resta ver cómo mejorar la política. Teniendo en cuenta la ecuación 2.2 que establece la relación entre V^π y Q^π , y a partir de una política π , podríamos definir una nueva política π' como:

$$\forall s, \pi'(s) \leftarrow \operatorname{argmax}_a Q^\pi(s, a) \quad (2.3)$$

O sea, definimos la nueva política como la que toma para cada estado la acción que maximiza la función de valor $Q(s, a)$. A esto se lo llama una política *codiciosa* (greedy) a partir de Q . A continuación se obtiene la función de valor $V^{\pi'}, Q^{\pi'}$, y se vuelve a mejorar la política. Esto nos da una secuencia de iteraciones de mejora de política y generación de función de valor que finaliza con política y función de valor óptimas.

$$\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow \dots \rightarrow \pi^* \rightarrow V^* \quad (2.4)$$

Nuevamente hace falta un criterio de detención, que en este caso podrá ser cuando la política no sufra modificaciones, es decir, cuando $\forall s, \pi_k(s) = \pi_{k+1}(s)$.

2.5.2. Solución por métodos de Monte Carlo

Los métodos de Monte Carlo (MC), a diferencia de los de Programación Dinámica, no requieren un conocimiento completo del entorno. Se basan en el promedio de recompensas obtenidos a lo largo de la experimentación, y requieren que la tarea a realizar tenga forma episódica. Esto significa que el

¹Una demostración más formal y completa puede encontrarse en [Sut98]

agente realice la misma tarea varias veces (episodios) y que estos episodios terminen eventualmente sin importar las acciones del agente. Emplean también la función de valor y política, y a medida que se evalúan más episodios, el promedio estimado para estas funciones converge al valor esperado.

Un ejemplo de método de Monte Carlo es el de *primera visita*, que consiste en asignar a $V(s)$ el promedio de las recompensas obtenidas en un episodio, luego de la primera visita a s .

Algorithm 1 Algoritmo Monte Carlo *primera visita*, tomado de [Sut98]

```

 $\pi \leftarrow$  política a evaluar
 $recompensas(s) \leftarrow$  lista vacía  $\forall s \in S$ 
 $V \leftarrow$  función de valuación arbitraria
loop
  Generar un episodio usando la política  $\pi$ 
  for all estado  $s$  en el episodio do
     $R \leftarrow$  la recompensa obtenido luego de la primera visita a  $s$ 
    agregar  $R$  a  $recompensas(s)$ 
     $V(s) \leftarrow promedio(recompensas(s))$ 
  end for
end loop

```

Al igual que en Programación Dinámica, luego de obtener la función de valor se puede modificar la política en forma *codiciosa* (seleccionando la acción a que maximiza $Q(s, a)$), y repetir los pasos en forma iterativa hasta llegar a una política óptima.

Una característica interesante de los métodos Monte Carlo es que para calcular el valor de un estado, no requieren calcular el del resto ni tener un modelo del entorno, por lo que si lo que interesa es solo un subconjunto del espacio de estados, pueden resultar mucho más eficientes que la Programación Dinámica.

2.5.3. Solución por métodos de Diferencias Temporales

Los métodos de Diferencias Temporales (TD) toman elementos de Programación Dinámica y de los métodos Monte Carlo. Usan también una función de valor para estimar la política, y es similar a MC en que usan la experiencia del agente para actualizar su estimación. Es similar a PD en que actualiza esta estimación sin necesidad de que el problema sea episódico.

La idea central de los métodos de TD es la de actualizar la función de valor a medida que se obtienen refuerzos de parte del entorno. El método más simple, llamado $TD(0)$, actualiza la función de valor de la siguiente manera:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \quad (2.5)$$

Donde s_{t+1} es el estado al que se llega luego de ejecutar la acción que indica la política estando en el estado s_t , y r_{t+1} la recompensa correspondiente.

Para obtener la función de valuación V_π para una política π , puede usarse el siguiente algoritmo:

Algorithm 2 Algoritmo TD(0)

```

 $\pi \leftarrow$  política a evaluar
 $s \leftarrow$  estado inicial
while no termina el episodio do
   $t \leftarrow$  paso actual del episodio
   $a \leftarrow \pi(s)$ 
   $(r, s') \leftarrow$  recompensa y nuevo estado luego de ejecutar  $a$ 
   $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$ 
   $s \leftarrow s'$ 
end while

```

Una forma particular del algoritmo TD es Q-learning (mencionado en 2.4), que se define como:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (2.6)$$

y puede ser reescrito como:

$$Q(s_t, a_t) \leftarrow (1 - \alpha)Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a)] \quad (2.7)$$

Q-Learning posee algunas características que lo hacen atractivo a los fines del problema planteado en esta Tesis:

- Resuelve problemas episódicos. Los juegos en general (incluida una simulación de combate aéreo) pueden verse como episódicos por naturaleza, donde cada partida que se juega es un episodio.
- Permite aprender sin conocimiento previo de la política π o la función de valor Q , y, a medida que se juega, adquiere conocimiento con cada acción que toma el agente.

Es por estas características que ésta técnica será la que se use en la solución del problema planteado en la Tesis. Adicionalmente, hay antecedentes del uso de algoritmos TD en juegos en general [Tes95], pero no dentro del contexto de un videojuego en tiempo real, lo que vuelve atractivo investigar posibles formas de aplicarlo.

2.6. Resumen

Se presentaron en este capítulo los fundamentos de los problemas de aprendizaje por refuerzo. Se presentó el concepto de Procesos de Decisión de Markov y cómo se modela a través de ellos un problema de AR. Se mostraron tres familias de algoritmos: Programación Dinámica, métodos de Monte Carlo y métodos de Diferencias Temporales. Derivado de esta última se presentó Q-learning, que se usará para la solución del problema planteado en esta Tesis. En el siguiente capítulo se presentarán antecedentes del uso de aprendizaje automático en juegos, en su mayoría usando AR.

Capítulo 3

Antecedentes

En esta sección se presentarán algunos trabajos relacionados a esta Tesis, por usar técnicas similares o por abordar el aprendizaje automático en contextos de videojuegos. La mayor parte de la investigación actual en IA en videojuegos está concentrada en el desarrollo manual y no en el aprendizaje automático. Debido a esto, los casos que se mencionarán son mayormente sobre juegos abstractos o juegos simples desarrollados por investigadores para probar sus técnicas.

3.1. Juegos abstractos

Probablemente el juego donde más se ha aprovechado AR es en el backgammon. En *Temporal Difference Learning and TD-Gammon* [Tes95], se emplea AR para entrenar al agente usando el método de diferencias temporales, y redes neuronales como estimador de función. TD-Gammon alcanzó un nivel de competencia en el juego similar al de los mejores jugadores profesionales de Backgammon.

El autor menciona dos inconvenientes en el uso de AR para resolver problemas que puedan resultar interesantes en el mundo real:

- El problema de la asignación de crédito a una acción cuando se trabaja con refuerzos con *delay*, como en el presente trabajo. Dado que se ejecuta una secuencia generalmente larga de acciones, es difícil saber si alguna fue particularmente buena o mala, y por ende cómo distribuir la recompensa obtenida entre ellas.
- Que las herramientas existentes para almacenar la información de aprendizaje son o bien tablas (*look up tables*) o estimadores lineales de funciones, y ninguno parece suficientemente adecuado para situaciones del mundo real.

Las dos observaciones aplican a este trabajo y no han resultado impedimentos para el aprendizaje del agente (como se verá en el capítulo de

resultados). En el primer caso, Q-learning soluciona el problema de los refuerzos con *delay*, y en el segundo, el uso de suavizado por núcleos permite almacenar la información de aprendizaje y estimar la función Q en forma adecuada para el problema.

En *Learning to play chess using reinforcement learning with database games* [Man03] el autor usó redes neuronales entrenadas por refuerzo para que el agente aprenda a jugar al ajedrez, empleando una base de datos de partidas como fuente de datos para el aprendizaje. Plantea dos diferencias importantes entre aprender a partir de una base de datos y aprender jugando contra sí mismo:

1. Jugar contra sí mismo consume mucho más tiempo, dado que primero debe generar partidas de las cuales obtener conocimiento, mientras que teniendo una base de datos, ya se tienen esas partidas.
2. Es más fácil detectar malas decisiones con una base de datos, dado que es probable que en un partido real el oponente se aproveche del error y gane la partida, mientras que si aprende solo, todavía no sabrá cómo aprovecharse de su propio conocimiento limitado.

Estos problemas son ciertos, pero en el caso del combate aéreo que se trabaja en esta Tesis no existe una base de datos de jugadas previas, por un lado, y por el otro el uso de un estimador de función reduce enormemente la cantidad de datos computados y almacenados, y por consiguiente el tiempo de entrenamiento.

3.2. Juegos en tiempo real

En *Evolving Neural Network Agents in the NERO Video Game*¹ [Sta05] (Figura 3.1) definen un método que llaman *real-time NeuroEvolution of Augmenting Topologies* (rtNEAT) como mecanismo de evolución para las redes. rtNEAT está basado en NEAT, que usa algoritmos genéticos para evolucionar redes neuronales modificando tanto los valores dentro de cada neurona como la topología de la red misma. El objetivo del juego en NERO es evolucionar un conjunto de agentes (soldados) que deben combatir contra un enemigo. rtNEAT usa feedback del juego en tiempo real para alimentar esta evolución. De esta forma, los agentes aprenden durante el juego, y no previamente durante una etapa de entrenamiento. Los autores de NERO explican que no usan AR debido a los siguientes problemas:

- El espacio estado/acción grande que se ve reflejado en una gran necesidad de memoria para almacenar el conocimiento adquirido y en un tiempo prolongado para evaluar qué acción tomar en cada estado, lo

¹<http://nerogame.org/>



Figura 3.1: Captura de pantalla del juego *Neuro-Evolving Robotic Operatives (NERO)*

que lo volvería inusable en juegos en tiempo real. En esta Tesis se atacarán estos problemas.

- AR tiene como premisa la convergencia hacia una solución óptima, pero en un entorno con múltiples agentes esto implica también un único comportamiento para todos los agentes, y eso puede ser aburrido en un juego. Sin embargo, en *Sensing capability discovering in Multi Agent Systems* [Par10], los autores muestran que esto no es cierto, entrenando agentes con AR en un sistema multi-agente, y obteniendo distintos comportamientos para cada uno.
- Para que un agente aprenda es necesaria exploración aleatoria de vez en cuando, pero eso no es deseable durante el juego, cuando el jugador espera un comportamiento “razonable” de parte del agente. Esto es cierto, pero en el caso de realizar el entrenamiento off-line no es un problema.
- Las técnicas de AR requieren de mucho tiempo para adaptar el comportamiento del agente. Es trabajo de esta Tesis encontrar métodos que permitan atacar también este problema.

En NERO el aprendizaje es parcialmente supervisado: el usuario indica parámetros deseables en forma interactiva, modificando el comportamiento de los agentes a medida que transcurre el juego.

Una variante del método NEAT llamado NEAT Particles [Has07] fue usado en el juego *Galactic Arms Race (GAR)* ² [Has10], donde es usado

²<http://gar.eecs.ucf.edu/>

para evolucionar redes composicionales para evolución de patrones (CPPN - Compositional Pattern-Producing Networks [Sta07]) que conforman las armas del jugador. La IA analiza el comportamiento del jugador y decide la forma en que evolucionan determinados parámetros de las armas que usa. El juego trascendió el ambiente científico y se distribuye comercialmente.

Tanto la evolución del comportamiento de los agentes, en NERO, y de las armas, en GAR, dependen del comportamiento del usuario, y no se trata de entrenamiento *offline* de un agente, como lo que se busca en este trabajo.

En *EvoTanks II Co-evolutionary Development of Game Playing Agents* [Tho06] se usan nuevamente redes neuronales y algoritmos genéticos para evolucionar un agente. El objetivo del agente en EvoTanks II CT (Co-evolutionary Tournament) es vencer a un contrincante en un combate de tanques. Las acciones de los agentes son simples: moverse, girar la torreta y disparar. El entorno es en ciertos aspectos similar al que tratamos en esta Tesis. Los autores basan su trabajo en versiones anteriores (EvoTanks, EvoTanks II) en los que se encontraban con el problema de que el agente aprendía hasta un máximo local, y la estrategia generada al entrenarse contra un tipo de oponente no funciona contra otros tipos. En EvoTanks II CT se hacen evolucionar distintos agentes en simultáneo y competir en un torneo, enfrentando cada uno contra los otros, de forma tal de que el que resulte vencedor logre un buen desempeño contra contrincantes en general.

En los tres trabajos (EvoTanks, EvoTanks II, EvoTanks II CT) los agentes evolucionan mediante algoritmos genéticos, usando mutación y cruza para modificar una serie de genomas que codifican el comportamiento de cada tanque.

En esta Tesis también se entrenará al agente contra si mismo, además de hacerlo contra oponentes preprogramados.

En *Learning to Fight* [Gra04] se muestra el uso de AR para entrenar a un individuo en un juego de peleas, usando SARSA [Rum94] como mecanismo de aprendizaje de la función Q con el fin de entrenar a un oponente en un videojuego de pelea. Trabajaron con un juego real (*Tao Feng*³, Figura 3.2) al que modificaron para incorporar el aprendizaje. Algunos aspectos interesantes del trabajo son:

- Para reducir el espacio de estados dado por el sentido y las posibles acciones, realizaron una reducción de las características sensadas y una abstracción de más alto nivel de las acciones. Para las características, codifican la distancia al oponente y algunos datos adicionales, como que se encuentra bloqueando golpes (postura defensiva) o cuál fue su acción previa. Para las acciones se tomó un subconjunto pequeño de todas las posibles y se crearon *meta acciones* compuestas de una secuencia de acciones atómicas.

³http://en.wikipedia.org/wiki/Tao_Feng:_Fist_of_the_Lotus



Figura 3.2: Captura de pantalla del juego *Tao Feng: Fist of the Lotus*

- Se probaron distintos aproximadores para la función Q , y distintas funciones de recompensa. Entre los estimadores usaron uno lineal y una red neuronal *feed-forward*. Las funciones de recompensa probadas realizaban un balance entre la salud del agente y la del oponente para determinar la recompensa.
- En algunos experimentos, notaron que el agente vencía el 100% de las veces luego del entrenamiento. Esto demostró que existen falencias en la IA preprogramada del oponente, que permiten encontrar una estrategia óptima para vencerla. Esto puede verse como un éxito del entrenamiento, pero también como una llamada de atención sobre los problemas presentes al desarrollar en forma manual el comportamiento de los oponentes en un videojuego.

En *An Object-Oriented Representation for Efficient Reinforcement Learning* [Diu08] definen el algoritmo *Deterministic Object-Oriented Rmax* (DOOR-MAX), basado en el algoritmo R-max [Bra02], pero empleando la orientación a objetos para definir el entorno. Usan MDPs Orientados a Objetos (OO-MDP) para representar a los elementos del entorno y sus acciones, lo que permite definir una característica de un objeto, y que cuando éste se repite en el entorno, si el agente ya interactuó con él, sepa cómo se comportará esta nueva instancia. Por ejemplo, si el agente es un vehículo, prueba avanzar contra una pared y no puede, la siguiente vez que se encuentre con una pared sabrá que tampoco podrá avanzar.

Usan dos casos de ejemplo: el problema del Taxi [Die00] y el juego Pitfall⁴. En ambos logran resolver el problema planteado con éxito, acotando los posibles estados del agente en ordenes de magnitud. En ambos casos se

⁴<http://en.wikipedia.org/wiki/Pitfall!>

trata de dominios discretos, donde el conjunto de acciones es acotado y bien definido (En el caso del taxi, moverse en cuatro direcciones y levantar/dejar al pasajero; en el caso del Pitfall moverse a derecha o izquierda, saltar y subir/bajar escaleras) por lo que la solución no se aplica directamente al presente trabajo.

Otra diferencia importante es que los problemas planteados (Taxi y Pitfall) se benefician de la orientación a objetos dado que se repiten elementos del entorno, algo bastante habitual en los videojuegos. En este trabajo eso no sucede, pero sí podría suceder en posibles extensiones (ej: entrenar al agente contra dos oponentes en simultáneo).

3.3. Resumen

En este capítulo se vieron algunos antecedentes del uso de aprendizaje automático para entrenar agentes en videojuegos. Los resultados de la mayoría de estos trabajos son alentadores, dado que muestran la posibilidad de entrenar agentes en forma autónoma, aunque en otros contextos. Como se explicó en la introducción, el objetivo de este trabajo es entrenar a un agente para el contexto específico de maniobras de combate aéreo en un espacio continuo. Esta tarea no se ha encontrado abordada en la bibliografía consultada, y por lo tanto resulta de interés explorar, sobre todo luego de comprobar que en otros contextos es posible realizar entrenamiento de agentes para videojuegos.

En el siguiente capítulo se presentarán las características del entorno de entrenamiento y del agente, además de las herramientas de simulación desarrolladas para el entrenamiento y la visualización del agente.

Capítulo 4

Entorno y herramientas de simulación

En este capítulo se verán las características del agente y su entorno, además de la herramienta de simulación desarrollada para entrenar y probar al agente. Como se explicó antes, el objetivo de esta Tesis es el aprendizaje automático de maniobras de combate aéreo en un entorno de videojuego. En este contexto, el agente a entrenar será un avión que debe combatir contra un oponente en un entorno tridimensional sin elementos adicionales con que interactuar.

La herramienta desarrollada modela el espacio tridimensional en el que opera el agente y su contrincante y simula las consecuencias de las acciones tomadas por los aviones (movimiento y cambios de rumbo). Permite también visualizar el estado de la simulación y el aprendizaje del agente.

4.1. Entorno de la simulación

El entorno de entrenamiento es un espacio tridimensional donde están ubicados tanto el agente como su contrincante. Se simula el paso de tiempo a intervalos regulares, y en cada paso ambos aviones avanzan hacia la dirección a la que apuntan en ese momento. El agente tiene la posibilidad de cambiar su dirección horizontal (yaw), vertical (pitch) y velocidad. El oponente tiene las mismas capacidades, dependiendo del comportamiento pre-programado si hace uso de ellas o no.

Cada avión puede hacer *lock* sobre el contrincante, es decir, ubicarse en una posición ventajosa adecuada para disparar. Este *lock*, también llamado *solución de tiro*, ocurre cuando el oponente queda ubicado en el espacio dado por la intersección entre una esfera de radio d y centro en el avión, y una pirámide proyectada a través del eje longitudinal, acotada por ángulos definidos sobre los ejes lateral y vertical del avión (Figura 4.1). Para vencer, un avión debe posicionar al contrario en *lock*, o sea, maniobrar de forma tal

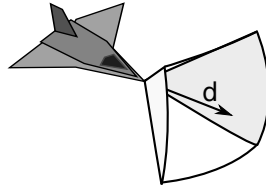


Figura 4.1: Proyección de la pirámide de lock

de obtener una solución de tiro.

Se considera a los aviones como objetos puntuales en el espacio tridimensional a los fines de definir su posición y detectar situaciones de lock. Si alguno de los aviones choca contra el piso (terreno completamente llano) o se eleva más allá de la máxima altura permitida, se considera que perdió.

En cada turno de la simulación, el agente percibe el estado propio y el del oponente, y debe decidir una acción en función de ello. El estado que percibe está compuesto por su posición (vector de 3 variables con las coordenadas x,y,z), dirección (vector de 3 variables, normalizado, representando la variación de x,y,z por unidad de tiempo) y velocidad (1 variable). El oponente recibe el mismo sentido del entorno.

4.2. Desarrollo de la herramienta

La herramienta de simulación y entrenamiento fue desarrollada en el lenguaje y entorno Java¹, y usa la librería *RL-Glue* (²), un entorno general para AR. RL-Glue define interfaces para el entorno y el agente, y establece un protocolo para comunicar ambos a través de observaciones (entorno \rightarrow agente) y acciones (agente \rightarrow entorno). Esto reduce el acoplamiento y permite implementar cada parte en forma clara e independiente. También permite implementar en distintos lenguajes al agente y al entorno, que corran en distintas computadoras (C/C++, Java, Lisp, Matlab y Python).

RL-Glue permite además intercambiar agentes y entornos con otros desarrolladores e investigadores, dado que se usa un protocolo común para comunicarlos. Eso significa que se puede implementar un entorno distinto al que se usó en esta Tesis, y probar allí la capacidad de aprendizaje del agente. Por ejemplo, podría implementarse un entorno similar a un FPS y probar la capacidad del agente en el mismo. En el caso particular del agente desarrollado, por razones de performance y para simplificar el código, la

¹<http://java.oracle.com>

²<http://glue.rl-community.org/>

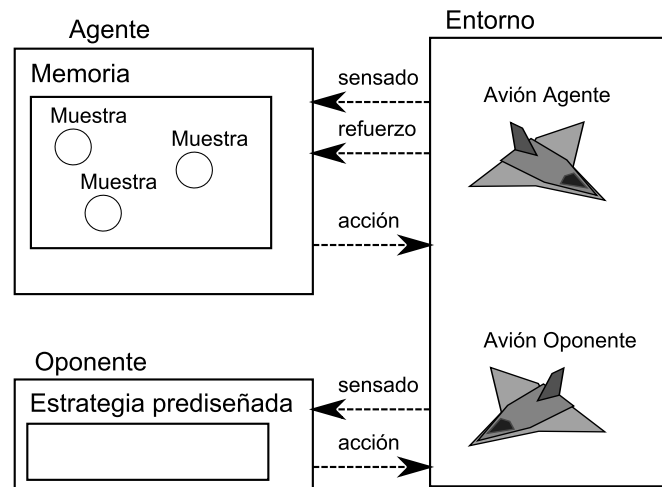


Figura 4.2: Estructura de comunicación entre agente, oponente y entorno

implementación es ligeramente dependiente del entorno.

Puede verse en la Figura 4.2 la forma en que se comunica la información entre agente, entorno y oponente. El entorno contiene los aviones, tanto el agente como el oponente obtienen la información de él. El agente también obtiene una señal de refuerzo, mientras que el oponente no. Ambos se comunican a su vez nuevamente con el entorno en forma de acciones, indicaciones de cómo modificar el comportamiento del avión (cambio de yaw, pitch y velocidad). El entorno realiza la simulación de las acciones, y al siguiente paso envía la nueva información de sensado al agente y oponente. El oponente contiene una estrategia prediseñada, no aprende durante la simulación (salvo indirectamente en el caso del oponente *Replicante*, que copia el comportamiento del agente y se explicará con mayor detalle en siguientes secciones).

4.3. Visualización

Como parte de la herramienta de simulación, se desarrollaron dos visualizaciones. La primera permite ver al agente y su oponente representados sobre el espacio, y la segunda permite ver el contenido de la memoria del agente.

4.3.1. Visualización de la Simulación

Para poder ver en el comportamiento del agente durante el entrenamiento, se usa la visualización de simulación. En su ventana (Figura 4.3) muestra el estado de los aviones con una proyección ortogonal desde arriba. Ambos se ven como círculos con una línea indicando su dirección y proyectando el cono de *lock*. El agente es verde y el oponente rojo, y el radio del círculo indica la altura como si fueran vistos desde arriba: cuanto más alto, más grande (Figura 4.4c). Además se muestran datos relevantes como el número de episodio actual en la corrida, el número de pasos en el episodio, la acción seleccionada, velocidad y recompensa, entre otros. Cuando un avión hace *lock* sobre el otro, su cono se tiñe del color del avión (En la Figura 4.4a se ve al agente - verde - haciendo lock sobre el oponente - rojo).

La ventana también permite pausar la simulación, ejecutarla paso por paso, ver el movimiento de los aviones en tiempo real (u ocultarlo y acelerar la simulación) y salir.

En la parte inferior muestra un *log* de los datos de cada entrenamiento seguidos de las pruebas, a razón de una corrida por línea. Al final del entrenamiento mostrará también el resultado de evaluarlo contra distintos tipos de agentes (ver capítulo de resultados).

En la secuencia de imágenes de la Figura 4.5 puede verse el comportamiento del agente a lo largo de un episodio. La secuencia muestra, de izquierda a derecha, y arriba hacia abajo, 6 pasos de un episodio de prueba. Pueden verse los datos de acción seleccionada, Q estimada para el estado actual, distancia y otros. En este episodio el oponente avanza sin desviarse de su dirección original, mientras que el agente maniobra hasta obtener la solución de tiro.

Puede verse en la Figura 4.4c un ejemplo de los dos aviones volando a distintas alturas: el agente (verde) se ve más grande por estar volando más alto. En los datos mostrados a izquierda de la pantalla se lee la diferencia de altura en metros.

4.3.2. Visualización de la Memoria

En la ventana de la Figura 4.6 se muestra un corte parcial de la memoria del agente. Se representa cada punto como un círculo cuyo radio depende (en forma logarítmica) del valor de Q almacenado, y su color depende de si es un valor positivo (amarillo) o negativo (gris). La posición x e y del punto representan parámetros configurables. Por ejemplo, si se relaciona el valor x con la diferencia de *yaw* entre los aviones, entonces en el extremo izquierdo estarán dibujados los puntos con valor de $-\pi$ y en el derecho π . También se puede usar la transparencia del círculo para representar una variable. Esto sirve para simular la proyección de una variable en una tercera dimensión (cuando más transparente un círculo más lejano), y resulta útil por ejemplo

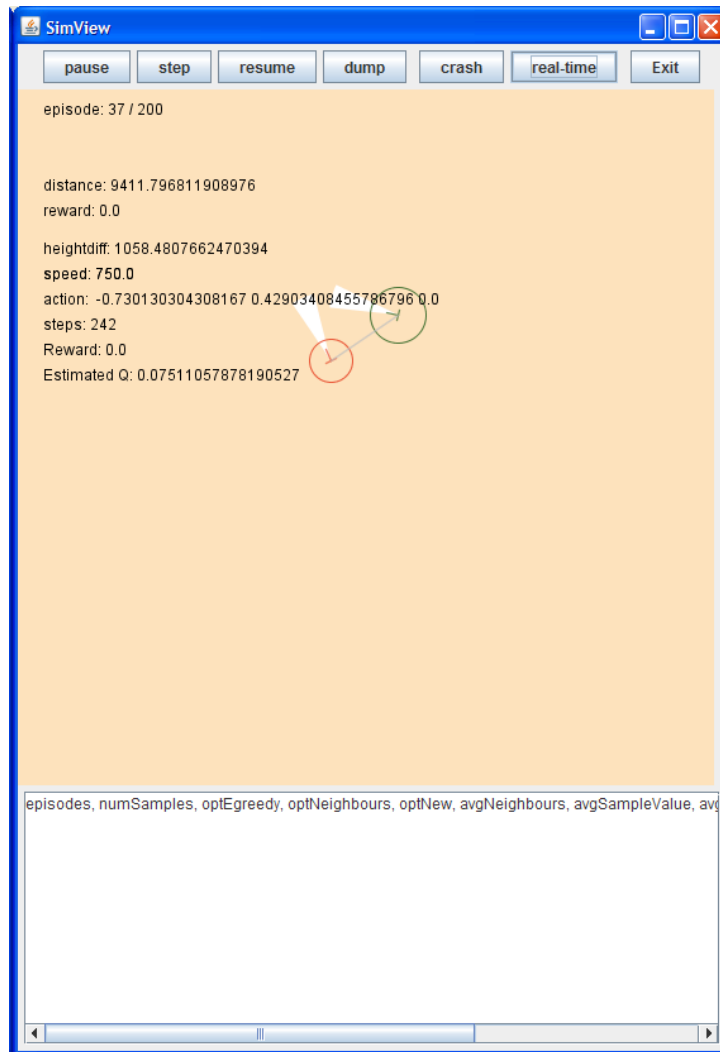


Figura 4.3: Captura de pantalla de la vista de simulación

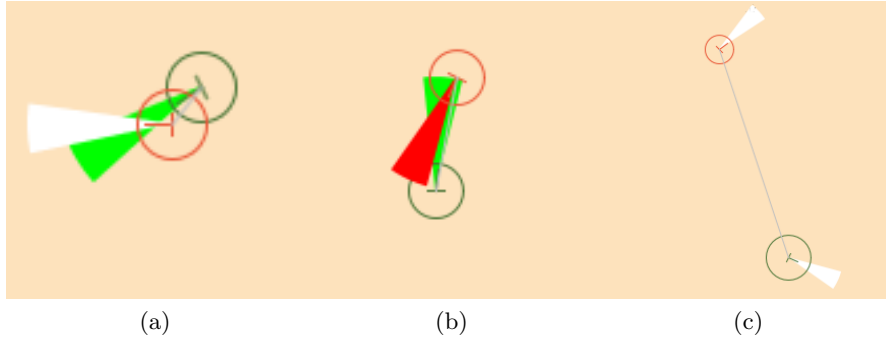


Figura 4.4: Detalle de la visualización, con el agente haciendo lock sobre el oponente, lock mutuo, y diferencia de altura entre los aviones

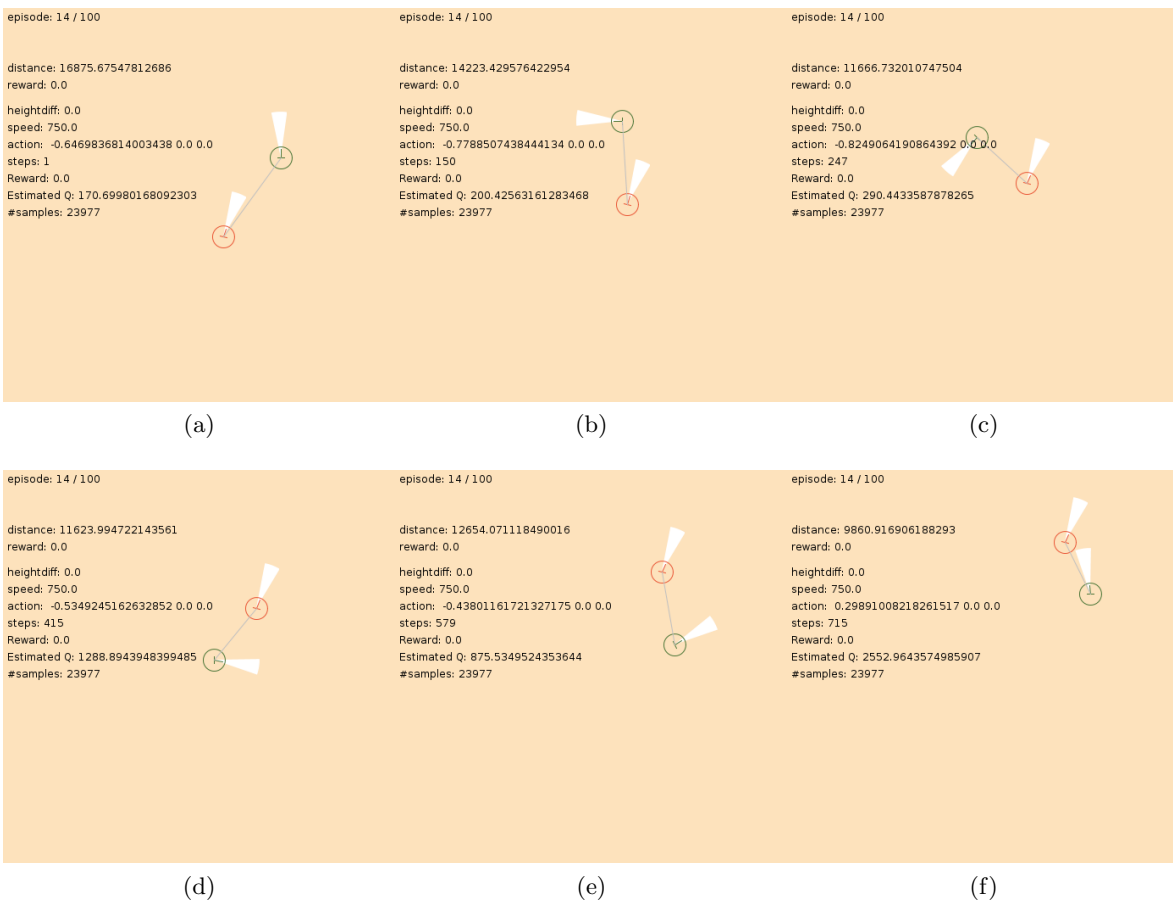


Figura 4.5: Secuencia de maniobra de ataque

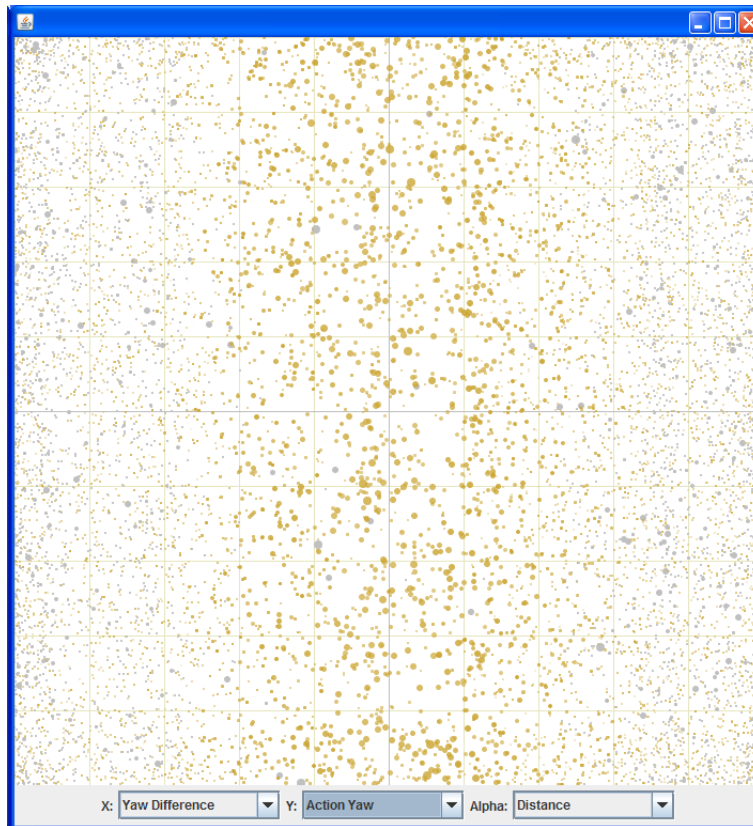
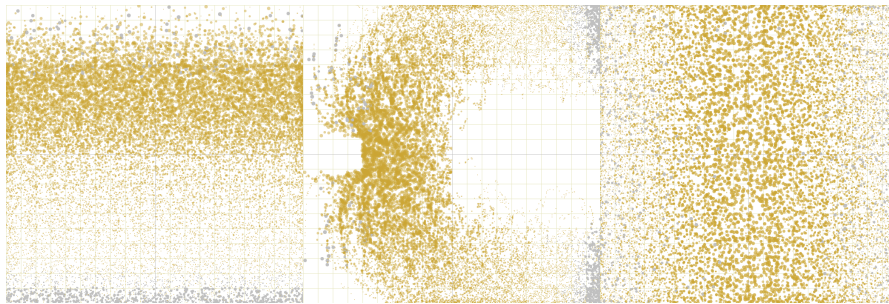


Figura 4.6: Captura de pantalla de la vista de memoria, con los controles para ajustar cada dimensión y botón para refrescar

para la diferencia en distancia.

4.4. Resumen

Se presentaron en este capítulo las características del entorno y herramientas de simulación empleados para el entrenamiento del agente. La aplicación permite ajustar características del entrenamiento para probar diversas estrategias en el aprendizaje, y ver el resultado en tiempo real. También provee una visualización de la memoria del agente, algo útil para entender las decisiones que toma, y para encontrar falencias en las opciones de entrenamiento. En el siguiente capítulo se extenderá la descripción del agente y se verá de qué forma toma las decisiones para interactuar con el entorno aquí descrito.



(a) x: Acción de yaw toma-da, y: Distancia
 (b) x: Distancia, y: Diferencia de yaw
 (c) x: Diferencia de yaw, y: Acción de yaw, transparencia: Distancia

Figura 4.7: Vistas de la misma memoria, usando distintos parámetros en cada dimensión (x,y)

Capítulo 5

Desarrollo del agente autónomo

En los capítulos anteriores se presentaron los problemas de aprendizaje por refuerzo y distintas formas de resolverlo. También se comentaron trabajos previos de entrenamiento autónomo para agentes en videojuegos, y los problemas inherentes a ese tipo de entorno. A continuación se mostrará el uso de AR para el entrenamiento del agente en la realización de maniobras de combate aéreo. Primero se explicará el algoritmo a usar, su extensión a estados y acciones continuos y el concepto de estimación de función Q por suavizado de núcleos. Luego se verán los detalles de implementación del agente, incluyendo pseudo-código que explica su funcionamiento.

5.1. Aprendizaje con Q-learning

El algoritmo de Q-learning [Wat89], presentado en el capítulo de Aprendizaje por Refuerzo, emplea la función de valor $Q(s, a)$ para evaluar cada estado y acción y a partir de esa valuación encontrar una política óptima para el agente en el entorno. La forma vista en la Ecuación 2.7 tiene la desventaja de que requiere una cantidad finita de estados y acciones, y no resulta aplicable, en forma directa, en un entorno con variables en el dominio de los reales, como el que se desea resolver en esta Tesis.

Una forma de tratar estos problemas es mediante una cuantización o discretización de las variables, generalmente dividiendo el espacio de cada dimensión en segmentos, convirtiendo un rango de valores a un único valor común. De esta forma la cantidad de estados totales estará dada por la multiplicación de la cantidad de segmentos definidos para cada dimensión.

Otra forma de encarar el problema es viendo que si $s, a \in \mathbb{R}$, entonces $V(s) : \mathbb{R} \rightarrow \mathbb{R}$, $Q(s, a) : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, y por lo tanto se podría usar un *estimador* (o *aproximador*) de función para V y Q en vez de intentar calcularlas directamente. Un estimador posible es una Red Neuronal [McC43], como las

usadas en algunos de los antecedentes antes citados. Otro tipo posible es un *estimador por suavizado por núcleos*, como el usado en este trabajo, que se ha empleado con éxito en problemas similares, aunque no en videojuegos.

5.2. Estimación de la función Q con suavizado por núcleos

Un estimador con suavizado por núcleos [Orm99] se basa en calcular el valor de la función en un punto dado, a partir del valor de los puntos cercanos, usando una función de núcleo o *kernel* K para sopesarlos. Este concepto se usa para estimar tanto Q como π . Si la función Q es continua a lo largo del espacio, puede estimarse tomando muestras en distintos puntos y realizando una aproximación para los puntos donde no hay muestras pero que contienen muestras cercanas. Algunas características deseables en K son: ser positiva, simétrica alrededor del 0, continua, y que su integral tenga valor 1.

En particular para esta Tesis se implementó el trabajo planteado en [Vil10]. Para realizar la estimación se registran las experiencias previas del agente en una memoria multi-dimensional (con dimensiones para cada variable del estado y de la acción), se toma un criterio de vecindad (distancia de Chebyshev en este caso), y al momento de indagar el valor de Q se toman las experiencias vecinas si no se cuenta con el valor para esa posición exacta. Esta técnica se usó con éxito en entornos similares al que presentamos, en los que un robot debía navegar por un entorno continuo y definir acciones también en un espacio continuo. Se usó $K(x) = e^{-(x/td)^2}$ como función de núcleo, donde td es la máxima distancia de vecindad, junto con una estimación por coeficientes de Nadaraya Watson [Nad64]:

$$w_i \leftarrow \sum \frac{K(x_i)}{\sum_{j=1}^n K(x_j)} \quad (5.1)$$

El algoritmo citado ([Vil10]), se ha aplicado con éxito en un contexto similar al de este trabajo, aunque con algunas diferencias:

- En [Vil10] el objetivo del agente (un robot) es llegar hasta un punto determinado, fijo, mientras que acá el objetivo (el avión oponente) se encuentra en movimiento y existen variaciones en la diferencia de velocidad de ambos aviones.
- En el presente trabajo el final de un episodio puede darse por varias causas (ganar, perder, fin de los pasos, distancia excesiva) por lo que debe diseñarse una función de refuerzo que incentive ganar y desincentive perder y que de alguna forma evite que el agente se aleje.

- No se implementaron las relaciones de Homotecia y Simetría. Sin embargo, la forma de representación del problema, al convertir los datos sensados de absolutos a relativos al agente, evita el almacenamiento de información redundante. Por ejemplo, al operar sobre la diferencia de altura, no importa si los aviones están a 1000 y 2000 metros o a 4000 y 5000, siempre que la diferencia de altura entre ambos sea la misma.

5.3. Implementación

5.3.1. Características del entorno

El entorno en que se mueven los aviones es una simulación de espacio aéreo sin características de terreno, es decir, con el suelo siempre a la misma altura, y sin estar acotado horizontalmente. Si el entorno tuviera características adicionales, podría ser útil emplear alguna técnica que permitiera simplificarlo o representarlo mediante objetos, como las descritas en [Diu08], ya mencionado en el capítulo de antecedentes, pero dado que el entorno no contiene características, lo único relevante es la información de los aviones.

5.3.2. Sensado del agente

El agente obtiene del entorno los datos de posición (3 variables), dirección (3 variables) y velocidad (1 variable) propia y del oponente. En total son 14 variables de punto flotante (7 por avión), que se reducen mediante un conjunto de transformaciones lineales a 5:

- **Diferencia de yaw:** la diferencia entre la dirección actual del agente y la dirección en que se encuentra el oponente, en forma de ángulo (Figura 5.1a).
- **Yaw relativo del oponente:** la diferencia, en ángulos, entre el yaw propio y el del oponente (Figura 5.1b).
- **Diferencia de altura:** la diferencia de altura entre el agente y el oponente, normalizada en función de la máxima diferencia posible (la distancia entre el piso y la máxima altura de vuelo).
- **Diferencia de velocidad:** normalizada en función de la diferencia entre las velocidades máximas y mínimas.
- **Distancia entre los aviones:** normalizada en función de la máxima separación que puede haber antes de considerar que el oponente (o el agente) se escapó y, por lo tanto, debe finalizar el episodio.

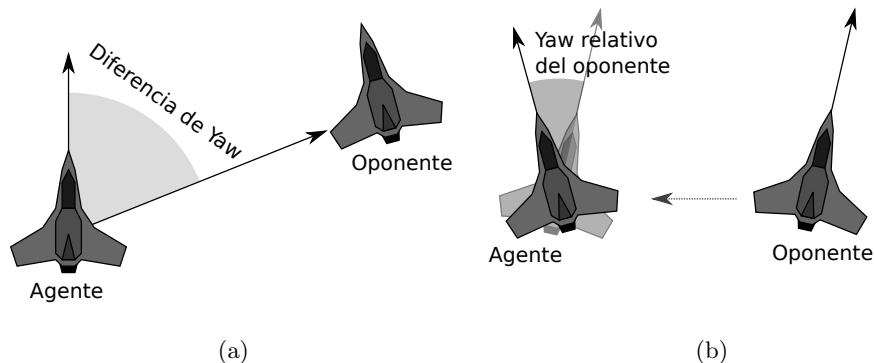


Figura 5.1: Diferencia de yaw y Yaw relativo del oponente

Puede notarse que no se registra la altura del agente, aparentemente necesaria para evitar chocar contra el piso. Esto obedece a que dado que lo relevante para el agente es obtener una posición de tiro sobre el oponente, no tiene sentido que tome la decisión de distanciarse del mismo. Por lo tanto, si el oponente se encuentra más arriba, el agente no necesitará bajar y por lo tanto nunca chocará. Si el oponente se encuentra debajo, solo necesitará bajar hasta una altura tal que pueda obtener la solución de tiro, y ésta altura puede ser superior, o inclusive la misma que la del oponente, pero no será necesario tampoco encontrarse más abajo. Dado que el oponente no se ha estrellado, el agente tampoco lo hará porque no bajará más de lo necesario. Por el mismo motivo no se elevará más allá de la altura máxima permitida una vez que haya sido entrenado.

La acción que ejecuta el agente está compuesta por 3 variables o dimensiones: cambio de *yaw*, cambio de *pitch*, y cambio de velocidad (ver Figura 1.2).

Al inicio de este trabajo se evaluaron ventajas y desventajas de distintos mecanismos de sensado en función de su nivel de abstracción. También se consideró la forma de representar las acciones. Entre otras, se consideraron las siguientes opciones:

1. **Sensado de bajo nivel:** Usar *Ray Casting* [Rot82] para simular la visión del agente. Esta opción hubiera requerido resolver varios problemas adicionales al foco de este trabajo, además de aumentar considerablemente las necesidades de procesamiento.
2. **Sensado de nivel medio/alto:** Recibir la posición y orientación de cada avión en forma precisa. Es la opción elegida por ofrecer una cantidad de información suficiente y a la vez fácil de procesar. En un avión real es posible obtener esta información por medio del radar.

3. **Acciones de bajo nivel:** Describir las acciones a nivel de *aviónica*, operando sobre instrumental del avión como alerones, potencia de motor, timón, etc. Este tipo de acciones son las que realizaría un jugador en un simulador de vuelo, pero se consideraron excesivas para el presente trabajo dado que obligaban a implementar una simulación física realista del modelo de vuelo.
4. **Acciones de nivel medio:** Describir las acciones en función del efecto que tendrán en el avión sobre sus tres ejes, es decir *roll* (giro sobre el eje longitudinal que lo recorre de punta a cola), *pitch* (control de la elevación) y *yaw* (giro alrededor del eje vertical, de izquierda a derecha). Este nivel de abstracción resultó atractivo dado que acotando a sólo 3 ejes dimensionales se pueden representar todas las acciones del avión.
5. **Acciones de alto nivel:** Describir las acciones en función de maniobras de alto nivel, como *tambor*, *Immelmann*, etc. (ver Figura 1.1). Esta opción resultaba atractiva, pero implicaba el desarrollo de una librería de maniobras predefinidas especialmente para el entorno de simulación desarrollado. Se descartó la opción, pero podría ser implementado en futuros trabajos.

5.3.3. Simulación

La simulación (Algoritmo 3) se ejecuta por pasos, avanzando el tiempo a intervalos fijos. A partir de la posición al comienzo del paso, el agente y oponente seleccionan sus acciones (Algoritmo 7), que son ejecutadas para obtener el nuevo estado. A partir del refuerzo obtenido se actualiza el agente (Algoritmo 4) Si no hay *lock* y mientras queden pasos en el episodio, la simulación continúa, alimentando el nuevo estado del agente y del oponente.

Algorithm 3 Simulación

```
end ← false
oldstate ← nil
while Quedan episodios do
  while Quedan pasos por ejecutar en el episodio and not end do
    if oldstate ≠ nil then
      action ← agente.accionOptima(oldstate) // Definir la acción que
      el agente tomará, Algoritmo 7
    end if
    opAction ← oponente.accionOptima(oldstate) // Definir la acción
    que el oponente tomará
    state ← simular(oldstate, action, opAction) // Tomar los datos
    nuevos de posición, dirección y velocidad de los aviones
    if agente.apuntando(oponente) then
      // Verifica si el avión oponente se encuentra en el área de lock
      reward ← recompensa por ganar
      end ← true
    else if oponente.apuntando(agente) then
      reward ← recompensa por perder
      end ← true
    end if
    agente.actualizar(state, reward) // Algoritmo 4
    oldstate ← state
  end while
end while
```

5.3.4. Comportamiento del agente

El agente tiene dos funciones básicas: determinar la acción óptima en cada caso (Algoritmo 7) y actualizar su memoria a partir del estado, acción y refuerzo recibido (Algoritmo 5).

La actualización de su memoria consiste en buscar puntos vecinos al correspondiente al estado actual, y alterar sus valores de Q (Algoritmo 6) a partir del refuerzo recibido.

Algorithm 4 Actualización del agente (*agente.actualizar()*)

```
stprev ← Estado observado del entorno en el paso anterior
actionprev ← Acción ejecutada en el paso anterior
stact ← Estado del entorno por ejecutar actionprev en stprev
ract ← Recompensa del entorno por ejecutar actionprev en stprev
if el agente está aprendiendo then
  memoria.actualizar(stprev, actionprev, stact, ract)
end if
```

5.3.5. Actualización de la memoria

La memoria almacena puntos (o *samples*) compuestos por estado, acción, estimación de Q y número de visitas ($s, a, Q, nvis$). Las componentes s y a se usan para ubicar al punto en el espacio de memoria y una vez almacenadas no se alteran, mientras que Q y $nvis$ son datos que pueden variar cada vez que se lo accede. En el Algoritmo 5 figurará el *sample* como la combinación de (s, a) y tanto el valor de Q como de $nvis$ figurarán como funciones a las que se asigna un nuevo valor para su parámetro s o (s, a).

Es importante destacar que se incorporan puntos nuevos únicamente en el caso de que no haya otros puntos vecinos en la memoria; en el caso de que los hubiera se actualizan sus valores, pero no se incorpora el nuevo punto. Esto significa que en la memoria se almacena una cantidad finita de puntos, definida por la cantidad de dimensiones y la distancia de vecindad.

Los puntos se almacenan en un KD-tree [Ben75], que permite encontrar puntos vecinos de manera eficiente. El árbol se reconstruye completamente cada 4000 puntos agregados, con el fin de mantenerlo balanceado.

Algorithm 5 Actualización de la memoria (*memoria.actualizar()*)

```

 $st_{prev} \leftarrow$  Estado observado del entorno en el paso anterior
 $action_{prev} \leftarrow$  Acción ejecutada en el paso anterior
 $st_{act} \leftarrow$  Estado del entorno por ejecutar  $action_{prev}$  en  $st_{prev}$ 
 $r_{act} \leftarrow$  Recompensa del entorno por ejecutar  $action_{prev}$  en  $st_{prev}$ 
 $sample \leftarrow (st_{prev}, action_{prev})$ 
 $\Delta_t \leftarrow r_{act} + \gamma \max_a Q(st_{act}, a)$  // Estimación de  $Q$  para  $st_{act}$ 
if  $sample$  no tiene vecinos en la memoria then
     $agregar(sample, \alpha \Delta_t)$ 
else // tiene vecinos
     $actualizarVecinos(sample, \alpha \Delta_t)$ 
end if

```

5.3.6. Actualización de los vecinos

La actualización de los vecinos se calcula con una fórmula distinta a la de [Vil10]. Para la estimación de Q de cada vecino ns_i , en vez de asignar $Q^{actual}(ns_i) \leftarrow Q^{anterior}(ns_i) + \alpha \Delta_t$, se asigna $Q^{actual}(ns_i) \leftarrow (1 - \alpha)Q^{anterior}(ns_i) + \alpha \Delta_t$ (Algoritmo 6). Esta modificación fue sugerida por los autores de [Vil10] en comunicaciones privadas, dado que la formula original tenía el problema de no converger para Q .

Algorithm 6 Actualización de los vecinos (*actualizarVecinos()*)

- 1: input: $sample \leftarrow$ Posición en la memoria cuyos vecinos se actualizarán
 - 2: input: $\alpha\Delta_t \leftarrow$ Estimación de Q para el estado al que se llegó desde $sample$ aplicando la última acción del agente
 - 3: $\vec{ns} \leftarrow vecinos(sample)$
 - 4: **for all** $ns_i \in \vec{ns}$ **do**
 - 5: $Q(ns_i) \leftarrow (1 - \alpha) \times Q(ns_i) + \alpha\Delta_t$
 - 6: $nvis(ns_i) \leftarrow nvis(ns_i) + 1$
 - 7: **end for**
-

5.3.7. Elección de la acción óptima

El agente puede hacer uso del conocimiento adquirido (explotar) o probar acciones aleatorias (explorar). Usa una política ϵ -greedy para determinarlo, eligiendo explorar si al tomar un valor aleatorio en el rango $[0, 1]$, resulta menor a un ϵ determinado (líneas 3, 5). El valor que toma este ϵ a lo largo del experimento se explicará en la Sección 6.1.

Para elegir una acción óptima se realiza una combinación entre los vecinos del estado actual. Cabe recordar que un punto en memoria está compuesto por una parte de estado (st) y una de acción (a). Al recibir la observación del entorno lo que el agente obtiene es sólo la primera parte ($state$). Se buscan entonces los vecinos \vec{st} s en el subespacio correspondiente al estado (línea 4), cada uno notado como (st_i, a_i) , tales que $distancia(state, st_i) < td$, donde td es la distancia máxima de vecindad entre puntos en la memoria.

Para cada vecino se combina su acción (a_i) con el estado obtenido inicialmente del entorno ($state$), obteniendo un conjunto de nuevos puntos con las componentes $(state, a_i)$. Para cada uno de ellos se realiza una combinación convexa de sus vecinos (Algoritmo 8) obteniendo una acción promedio a_i^* . El objetivo es determinar cuál de estas acciones es la mejor, pero dado que no existen como puntos en la memoria, debe buscarse su valor de Q usando suavizado por núcleos (Algoritmo 9). Se toma el vecino compuesto $(state, a_i)$ con la mayor Q estimada, y se devuelve su acción promedio a_i^* .

Algorithm 7 Elección de la acción óptima (*accionOptima()*)

```
1: input: state  $\leftarrow$  Estado observado del entorno
2: input: entrenando  $\leftarrow$  true si está entrenando, false si está evaluando
3: input:  $\epsilon$   $\leftarrow$  valor actual para la política  $\epsilon$ -greedy
4:  $\vec{s}ts \leftarrow vecinosDeEstado(state)$  // Vecinos en su estado, sin tener en
   cuenta las distancia entre sus acciones
5: if entrenando and ( $random(0, 1) < \epsilon$  or  $size(\vec{s}ts) = 0$ ) then
6:   return randomAction()
7: end if
8:  $\vec{top} \leftarrow$  los N vecinos ( $st, a$ ) en  $\vec{s}ts$  con mayor valor de  $Q$ 
9: optima  $\leftarrow$  Accion default (no hacer nada)
10: Qoptima  $\leftarrow -\infty$ 
11: for all  $top_i = (st_i, a_i) \in \vec{top}$  do
12:   sample  $\leftarrow (state, a_i)$ 
13:    $\vec{ns} \leftarrow vecinos(sample)$  // Vecinos tanto en estado como en acción
14:   cvxAction  $\leftarrow combinacionConvexa(sample, ns)$ 
15:   sample'  $\leftarrow (state, cvxAction)$ 
16:    $\vec{ns}' \leftarrow vecinos(sample')$ 
17:    $Q(sample') \leftarrow estimacionPorSuavizado(sample', \vec{ns}')$ 
18:   if  $Q(sample') > Qoptima$  then
19:     Qoptima  $\leftarrow Q(sample')$ 
20:     optima  $\leftarrow cvxAction$ 
21:   end if
22: end for
23: return optima
```

5.3.8. Cálculo de acción por combinación convexa

El cálculo de combinación convexa del conjunto de puntos usa sus valores de Q y $nvis$ para sopesarlos: el valor de Q , porque es lo que define qué tan buena fue la acción, y el de $nvis$ porque representa una idea de la confiabilidad del punto, expresada como la cantidad de veces que fue actualizado. Se usa la suposición de que cuantas más actualizaciones haya recibido el punto, más confiable será su valor.

Algorithm 8 Cálculo de acción por combinación convexa

```
1: input: sample  $\leftarrow$  El punto en la memoria para el que se calculará la
   acción
2: input:  $\vec{ns}$   $\leftarrow$  Los vecinos del punto que se usarán para el cálculo
3: input: td  $\leftarrow$  distancia máxima de vecindad entre puntos
4:  $d_i \leftarrow distancia(sample, ns_i)$ 
5: return  $\sum a_i \left[ \frac{nvis(ns_i)Q(ns_i)(td-d_i)}{\sum nvis(ns_i)Q(ns_i)(td-d_i)} \right]$ 
```

5.3.9. Estimación de Q con suavizado por núcleos

Se estima el valor de la función Q en el punto a partir de sus vecinos, usando un promedio sopesado de Nadarawya-Watson y la función de *kernel* mencionados en 5.2.

Algorithm 9 Estimación de Q con suavizado por núcleos

- 1: input: $sample \leftarrow$ El punto en la memoria para el que se calculará la acción
 - 2: input: $\vec{ns} \leftarrow$ Los vecinos del punto que se usarán para el cálculo
 - 3: input: $td \leftarrow$ distancia máxima de vecindad entre puntos
 - 4: $d_i \leftarrow distancia(sample, ns_i)$
 - 5: $w_i \leftarrow e^{-(d_i/td)^2}$ // w es el vector de pesos de Nadarawya-Watson
 - 6: **return** $\sum \frac{w_i * Q(ns_i)}{\sum_{j=1}^n w_j}$
-

5.4. Resumen

En este capítulo se mostró cómo funciona el agente, incluyendo la forma en que procesa la información que recibe del entorno para actualizar su conocimiento, y la forma en que aprovecha este conocimiento para tomar mejores decisiones. Se mostró también cómo se implementó el algoritmo descrito en [Vil10] adaptado para este entorno, y cómo los datos de la memoria se almacenan en una estructura eficiente para datos multidimensionales (KD-Tree). En el siguiente capítulo se presentarán los experimentos realizados que mostrarán que el agente, implementado de la forma recién vista, es capaz de aprender en el entorno simulado de combate aéreo.

Capítulo 6

Experimentos y resultados

En el capítulo anterior se mostró el comportamiento del agente y su interacción con el entorno, la forma en que adquiere conocimiento y cómo lo usa para decidir acciones. En este capítulo se mostrarán los experimentos diseñados para probar el aprendizaje del agente, y los resultados obtenidos. Primero se describirá la forma en que se diseñaron los experimentos, luego se mostrarán los resultados para distintas condiciones del entorno, agregando complejidad, hasta llegar a abarcar el problema completo (con todas las dimensiones de estado y acción).

Para comprobar la utilidad del entrenamiento, se diseñaron experimentos que permiten evaluar el aprendizaje del agente a lo largo de varios episodios, con el fin de determinar cuántos son necesarios para que se comporte con eficacia. Es difícil establecer formalmente qué es un comportamiento eficaz para el caso de un videojuego, dado que lo que se busca no es un agente capaz de ganar siempre, sino que presente un desafío interesante ante el jugador. A la vez se busca diseñar un agente que pueda servir tan bien como uno programado manualmente (contra los que se probará en los experimentos). Finalmente, el comportamiento del agente no es determinístico, debido a que en algunas circunstancias debe elegir una acción entre varias con el mismo peso, y puede hacerlo aleatoriamente.

Dado todo esto, se optó por definir como eficaz a un agente que pueda triunfar contra sus oponentes programados manualmente al menos un 75 % de las veces que se lo pruebe.

6.1. Diseño general del entrenamiento

El entrenamiento está separado en dos partes: en la primera, decrece el valor del parámetro ϵ de la política ϵ -greedy, desde un valor inicial hasta 0. Esto incentiva la exploración y aporta información (puntos) a la memoria. En la segunda, decrece el valor del parámetro α de actualización de los valores de la memoria hasta llegar a 0, de forma tal de que se estabilicen los valores

almacenados. Esta última es condición necesaria para la convergencia de los métodos basados en *Q-Learning*, demostrada en [Wat89].

Cada etapa consta de *corridas*. Una corrida contiene episodios de entrenamiento (durante los que el agente aprende) y episodios de evaluación. Esto permite seguir el progreso del agente a medida que aprende. Cada episodio puede terminar como *ganado* (el agente hace lock al oponente), *perdido* (el oponente hace lock al agente) o *empatado* (finalizan los pasos sin que ninguno haga lock, o se separan demasiado).

6.2. Resultados

En esta sección se mostrarán los resultados de diversos experimentos. Primero, los resultados generales comparados entre sí que surgen de entrenar el agente contra diversos tipos de oponentes y luego medir su performance. Luego se verán algunos entrenamientos en detalle, para entender la forma en que el agente aprende a lo largo del tiempo.

6.2.1. Experimento preliminar

Una característica importante del entrenamiento es que se realiza desde posiciones iniciales aleatorias, para que el agente pueda aprender a comportarse en distintas situaciones. En el caso de que el agente no aprenda, resulta difícil inicialmente determinar si es por un error algorítmico, de implementación, o porque requiere más tiempo de entrenamiento.

Para ayudar a determinar si el agente aprende, se diseñó un primer experimento en el que las posiciones iniciales de ambos aviones son fijas (Figura 6.1), y el oponente no puede moverse ni girar. Este entorno resulta muy similar al diseñado por Villar y Santos [Vil10]. El comportamiento esperado del agente en este experimento era que luego de una exploración inicial encontrara al oponente, y a partir de ese momento reforzara el camino óptimo hacia él, disminuyendo la cantidad de iteraciones necesarios para obtener una posición de *lock*.

El aprendizaje del agente debería reflejarse en la cantidad de iteraciones que necesita para llegar a hacer *lock* sobre el oponente: cuanto más aprende, debería requerir menos iteraciones. Se espera que en los primeros episodios no lo encuentre, o requiera muchas iteraciones para hacerlo, y que a medida que decrece el valor de ϵ de la política ϵ -greedy, y favorezca la explotación por sobre la exploración, aumente el valor estimado de Q para las acciones que lo llevan al oponente y por lo tanto las tome con mayor frecuencia. Esto a su vez se debería generar una disminución en la cantidad de iteraciones.

Se realizaron corridas de 50 episodios de entrenamiento y 100 de evaluación. Se puede apreciar (Figura 6.2) en los resultados del experimento que el agente aprende rápidamente a encontrar al oponente (300 iteraciones), y hacia el final del entrenamiento tiene una eficacia del 100%. Puede verse

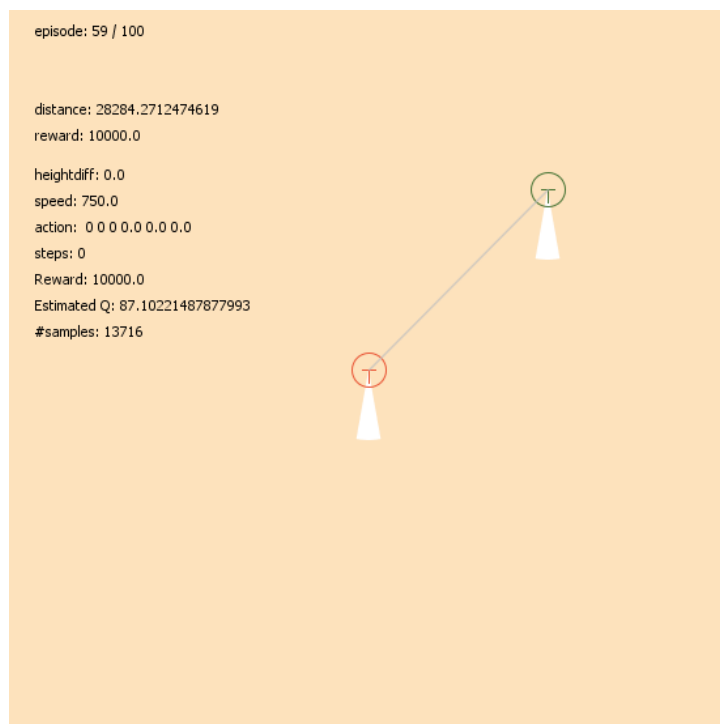


Figura 6.1: Posición inicial de los agentes en el experimento preliminar

también que el promedio de iteraciones necesarios para ganar se reduce hasta aproximadamente 280, donde permanece prácticamente inmóvil, lo que puede interpretarse como haber encontrado el camino óptimo hasta el oponente.

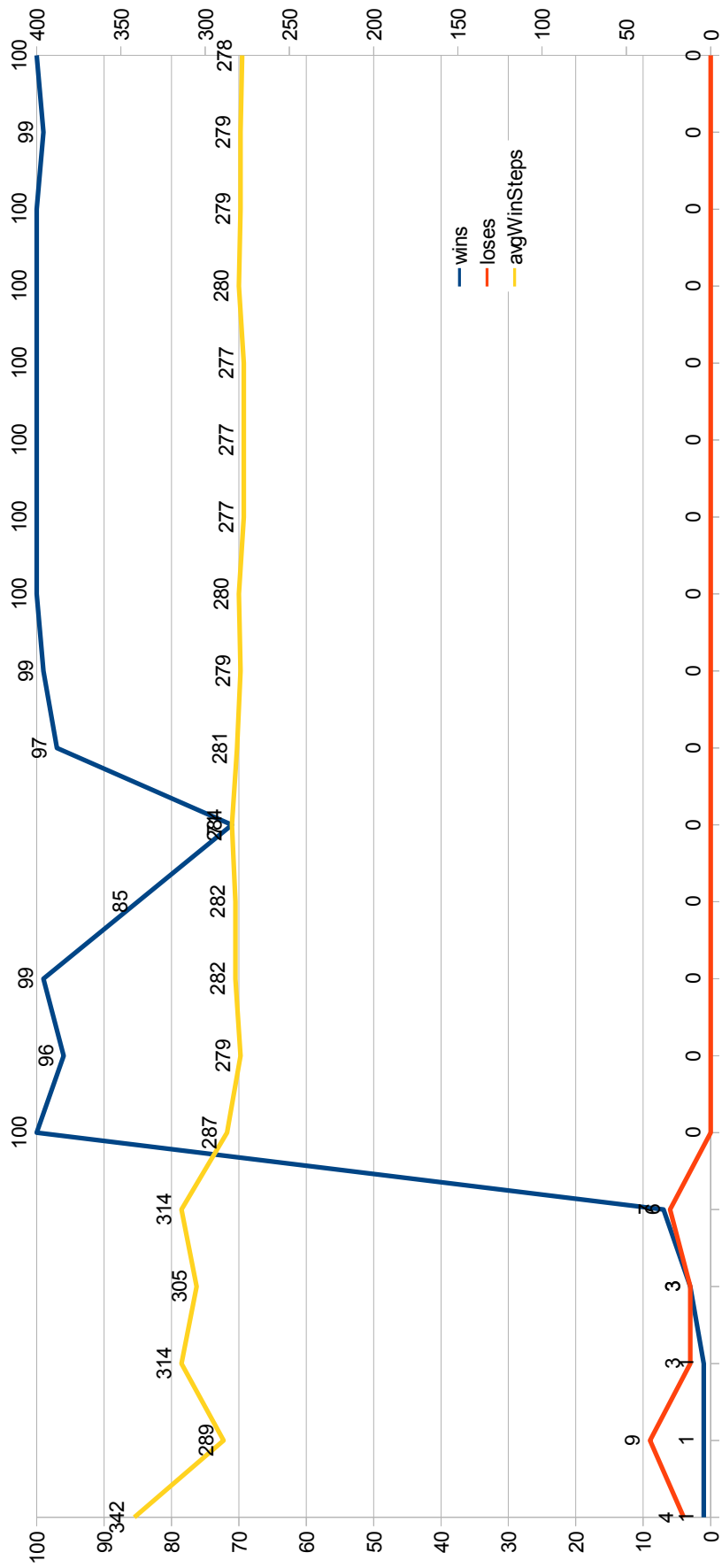


Figura 6.2: Detalle experimento preliminar

6.2.2. Resultados en un entorno acotado

Luego de comprobar que el agente aprende al entrenarse con episodios con inicio idéntico y oponente inmóvil, se probó el entrenamiento en una situación más cercana al objetivo del trabajo. En estos nuevos experimentos, y tanto en el entrenamiento como en la evaluación, se ubica al agente y su contrincante en una configuración inicial de posiciones y orientaciones aleatorias, tales que ninguno de los dos tenga al otro inmediatamente en la mira, aunque están suficientemente cerca como para que alguno pueda hacer *lock* durante el episodio. También se permite el movimiento del oponente, que obedecerá a alguno de los siguientes comportamientos pre-programados:

- *Dummy*: Se mueve en línea recta sin cambiar jamás su dirección.
- *Escape*: Se aleja siempre del agente, en la dirección opuesta.
- *Homing*: Avanza siempre en dirección al agente.
- *Replicant*: Usa la misma memoria del agente para elegir la acción óptima.

Los tres primeros comportamientos son simples; el objetivo es probar si un agente que aprendió solo su comportamiento es capaz de vencer a un oponente con una estrategia simple. En el caso del *Homing*, se trata de un comportamiento agresivo que a pesar de poder ser efectivo, no resulta práctico en una situación real, pero se lo incluye porque resulta difícil encontrar una estrategia contra él. El comportamiento *Replicant* permite probar al agente contra su propio conocimiento.

Sin embargo, los aviones no podían variar su velocidad, y comenzaban a la misma altura sin poder luego variarla (modificar su *pitch*). Esta restricción elimina no sólo espacio de acciones sino también de estados, dado que si no pueden variar su altura no tiene sentido sensarla, y lo mismo para la velocidad. Se eliminan por lo tanto 2 variables del estado, y 2 de las acciones, de las 5 y 3 definidas en la Sección 5.3.2. Quedan en el estado las variables para distancia, *yaw* del oponente y diferencia de *yaw*, y el cambio de *yaw* como única variable para la acción.

Parámetros usados en el experimento:

- Distancia de vecindad: 0,075.
- Valor iniciales para ϵ : 1. Para α : 0,1.
- Tasa de descuento γ : 0,9.
- Recompensa por ganar: 10000; perder: -10000; apartarse demasiado: -1000; finalización del tiempo sin *lock*: 0; recompensa a cada paso: 0.

- Cada episodio de entrenamiento o evaluación consta de 4000 iteraciones, cada corrida de 100 episodios.
- Se realizan 50 corridas (5000 episodios) decreciendo ϵ de la política ϵ -greedy, y luego 20 corridas (2000 episodios) decreciendo el parámetro α de aprendizaje.

Cada experimento consistió en entrenar al agente contra algún oponente, y al finalizar el entrenamiento evaluarlo contra todos. Los resultados se pueden ver en el Cuadro 6.1.

La primera columna indica contra qué oponente fue entrenado. En el caso *Random*, se elegía un tipo de oponente aleatoriamente en cada episodio de entrenamiento. La columna *Lock P.* indica la prioridad de lock (A = Agente, O = Oponente). Las columnas de evaluación corresponden a los distintos tipos de oponente y llevan su inicial: *Dummy*, *Escaping*, *Homing* y *Replicant*. Muestran el resultado de evaluar al agente contra cada uno, y los números indican, del total de 100 episodios de evaluación, en cuántos venció el agente y en cuántos el oponente. La última columna (*Prom*) contiene el promedio de resultados (victorias/derrotas) de las columnas anteriores, mostrando la efectividad de cada entrenamiento. La última fila contiene el promedio en función del oponente contra el que se lo evaluó, dando una idea aproximada de la dificultad de cada uno de ellos.

El tiempo de entrenamiento incluye tanto los episodios de entrenamiento como los episodios de prueba cada 100 entrenamientos, y fueron tomados de una PC rango medio-bajo para los estándares actuales (AMD Athlon 64 X2 3800+). Estos tiempos son relevantes principalmente para ser comparados entre si, aunque sirven también para ver que con una computadora moderna se puede entrenar un agente en pocos minutos. El entrenamiento hace uso de un único *core* del procesador.

En estos experimentos tanto el agente como su oponente mantienen una velocidad constante, pero la del agente es ligeramente superior. Si se hubiera usado la misma velocidad, sería imposible que ganara ante el oponente *Escaping* (conseguiría escaparse siempre), e imposible para la mayoría de los casos con *Dummy* (en caso de comenzar en una dirección que lo aleja del agente) o *Homing* (sería imposible para el agente tomar distancia para voltearse y apuntarle).

Del cuadro pueden desprenderse algunas observaciones:

- Entrenar contra *Dummy*, *Replicant* o aleatoriamente parece dar los mejores resultados al evaluar al agente contra todos los tipos de oponente. La principal diferencia que se desprende del cuadro es que tienen más del doble de puntos que los agentes entrenados con *Escaping* o *Homing*. Parece razonable esto si tenemos en cuenta que en el caso de *Escaping*, el agente sólo percibe al oponente alejándose, y en el caso de *Homing*, avanzando hacia él. En los dos casos la mitad del espacio de

Cuadro 6.1: Resultados de la evaluación cruzada de oponentes

Entrenamiento		Evaluación (ganados/perdidos)						
Oponente	Lock P.	puntos	Tiempo	D	E	H	R	Prom.
Dummy	A	59867	32:58	94/01	100/00	82/18	86/12	91/08
Escaping	A	26941	36:33	35/30	94/00	9/50	31/46	42/32
Homing	A	25473	15:56	62/10	78/00	62/35	36/11	60/14
Replicant	A	62373	40:23	96/03	100/00	76/23	72/28	86/14
Replicant	O	62776	44:06	87/11	94/00	05/95	46/54	58/40
Random	A	60468	35:40	97/00	100/00	82/18	80/17	90/09
Random	O	57900	45:58	90/09	100/00	09/91	41/59	60/40
Promedio				80/09	95/00	46/47	56/32	

estados queda oculto, dado que la variable correspondiente a diferencia de *yaw* tiene valores casi exclusivamente en $[-0,5, 0,5]$ para *Escaping* y en $[-1, -0,5] \cup [0,5, 1]$ para *Homing*, lo que podría justificar que tengan aproximadamente la mitad de puntos en memoria.

- El tiempo que lleva entrenar los agentes no es directamente proporcional a la cantidad de puntos almacenados (comparar *Homing* con *Dummy*, p.ej.).
- El oponente *Escaping* es, como era de esperar, el más fácil de derrotar, y *Homing* el más difícil. En el primer caso se debe a que jamás el oponente tendrá en la mira al agente. En el segundo, a que la política agresiva (y a veces suicida) del oponente hace que el resultado dependa de la habilidad del agente para maniobrar más rápido, y esencialmente de quién tiene la prioridad de lock. Esto puede verse en las filas donde la prioridad de lock es del oponente; en esos casos el agente logra vencer en menos del 10% de los casos. También puede verse cuando el entrenamiento fue con *Escaping*, y es comprensible porque en esos casos el agente jamás experimentó al oponente de frente.
- En los casos en que el oponente es *Replicant*, se obtiene un aprendizaje similar al de *Dummy*.
- Cuando se entrena contra oponente aleatorio se obtiene un resultado ligeramente más robusto que *Replicant*, pero nuevamente de un nivel similar al de *Dummy*.

Como conclusión final de estos experimentos, puede verse que para este contexto es suficiente entrenar contra el oponente más simple posible, el que no realiza ninguna acción. Esto podría deberse a que su comportamiento permite al agente experimentar una gran variedad de estados distintos.

6.2.3. Evolución del aprendizaje en un entrenamiento

A continuación se presenta el detalle de los entrenamientos con los cuatro tipos de oponentes y con oponente aleatorio, en todos casos con prioridad de lock para el agente (Figuras 6.3, 6.4, 6.5, 6.6 y 6.7), mostrando cómo evoluciona el agente a lo largo del tiempo. La línea negra vertical indica el punto en que el parámetro ϵ de la política ϵ -greedy llega a cero y por lo tanto no realiza más exploración.

Cada dato representa el resultado de evaluar al agente contra el mismo oponente con el que se está entrenando, cada 100 episodios de entrenamiento. La línea azul representa la cantidad de veces que ganó y la roja las que perdió, ambas mostrando valores según la escala del eje izquierdo. La línea amarilla indica la cantidad de iteraciones promedio que llevó ganar (en los episodios en que ganó), y su escala es la del eje derecho.

Observaciones a partir de los gráficos:

- El agente mejora con el tiempo, o sea, efectivamente aprende a lo largo del entrenamiento.
- Hacia el final del entrenamiento, el tiempo que lleva ganar es de entre 100 y 200 iteraciones en promedio, salvo para *Escaping*, que es superior (esperable, dado que se escapa y por lo tanto el agente tarda más en llegar). En *Dummy*, *Replicant* y *Random* el tiempo aumenta al principio y luego decrece. Esto puede interpretarse como efecto de la exploración y luego la explotación. En *Homing* esto no sucede porque la agresividad de su política lleva a que el episodio se resuelva rápidamente.

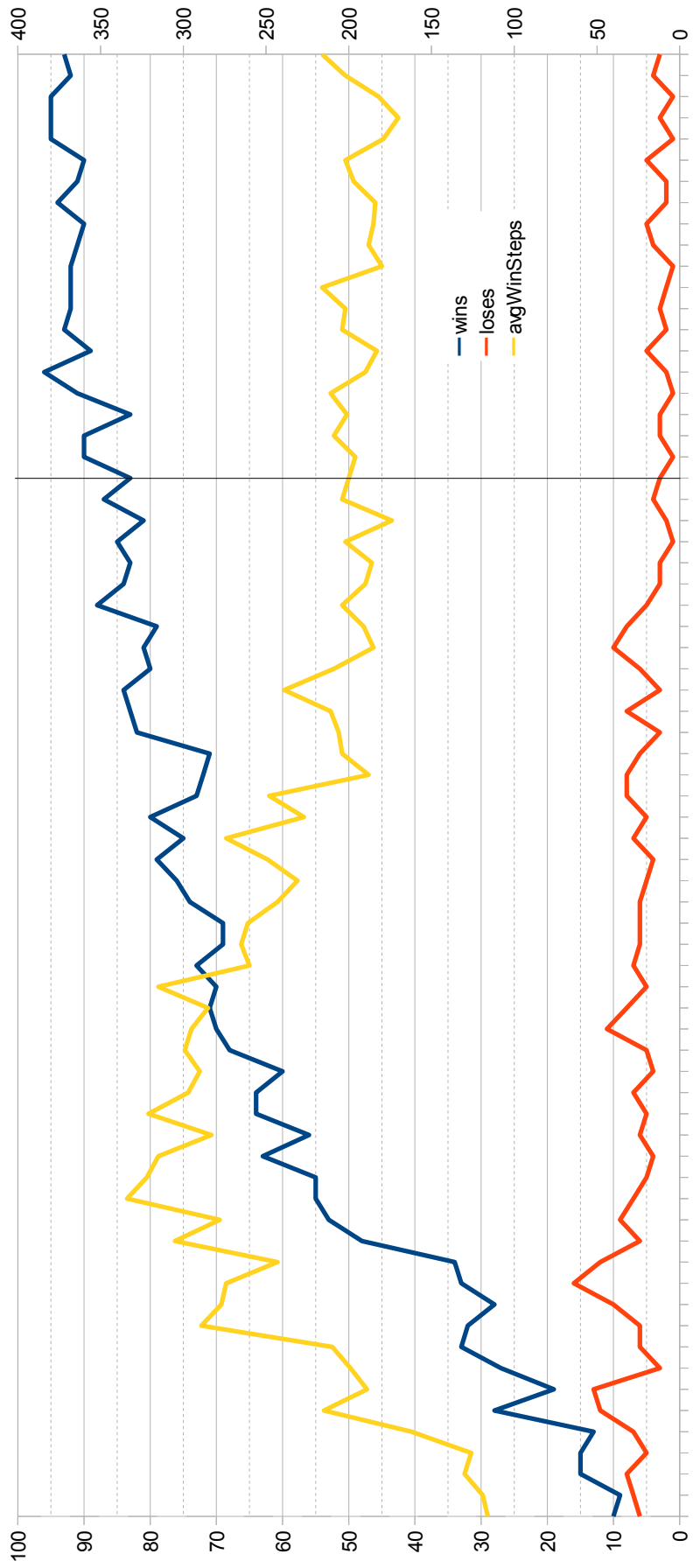


Figura 6.3: Detalle entrenamiento con oponente *Dummy*

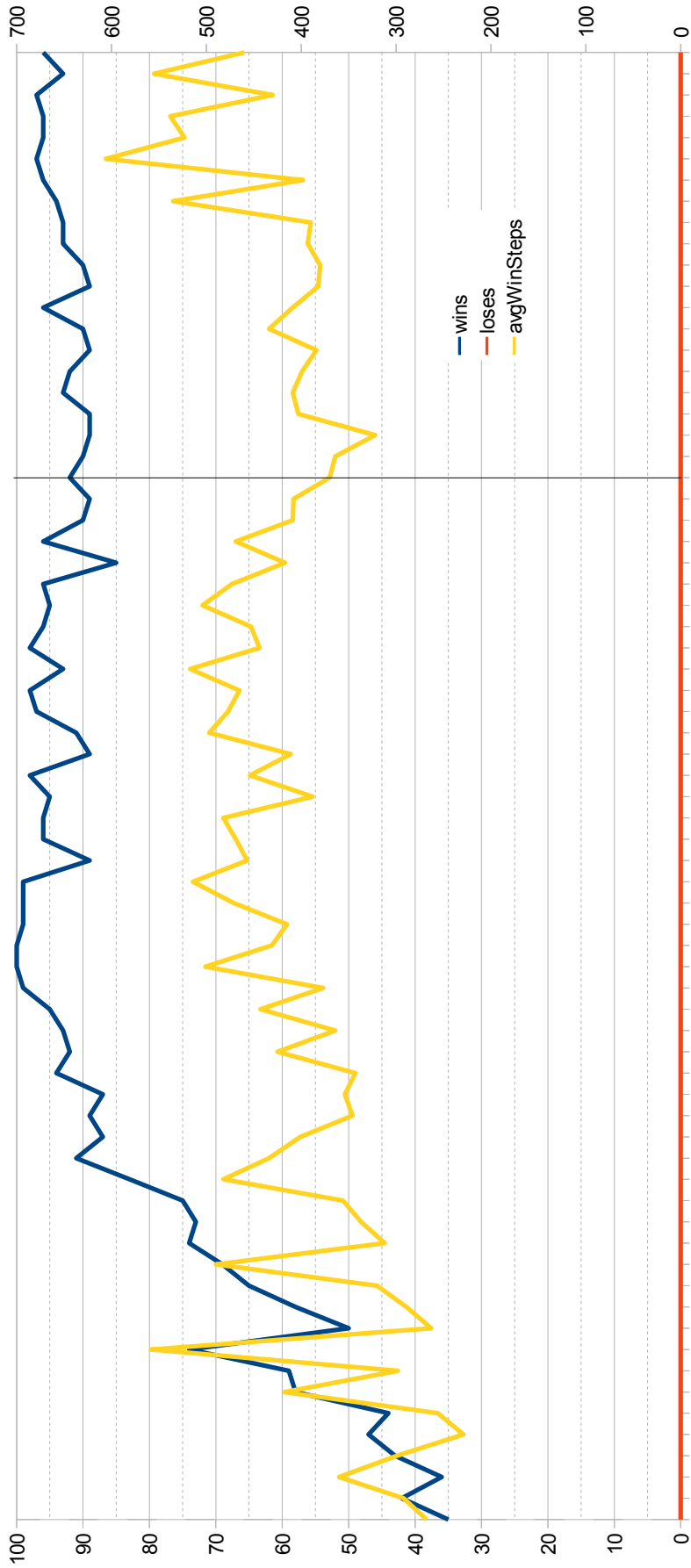


Figura 6.4: Detalle entrenamiento con oponente *Escaping*

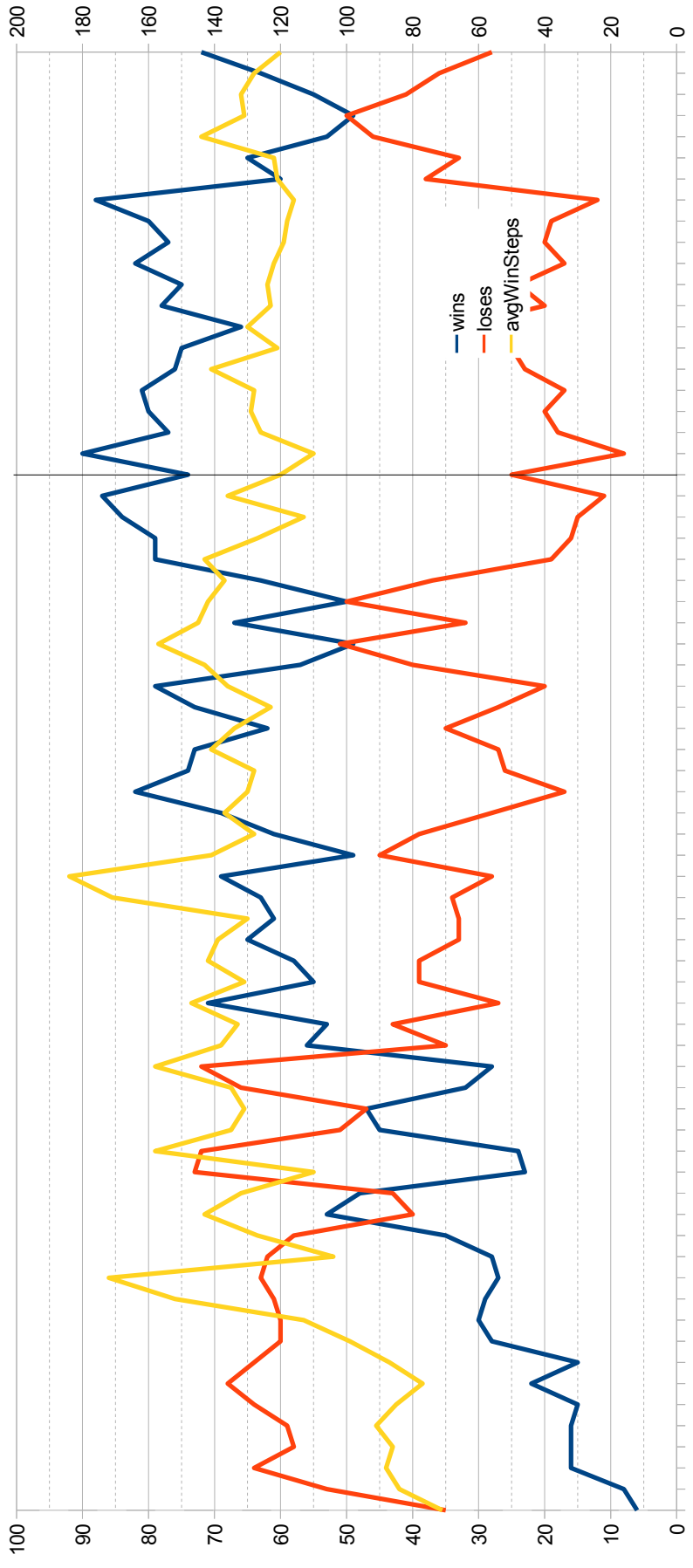


Figura 6.5: Detalle entrenamiento con oponente *Homing*

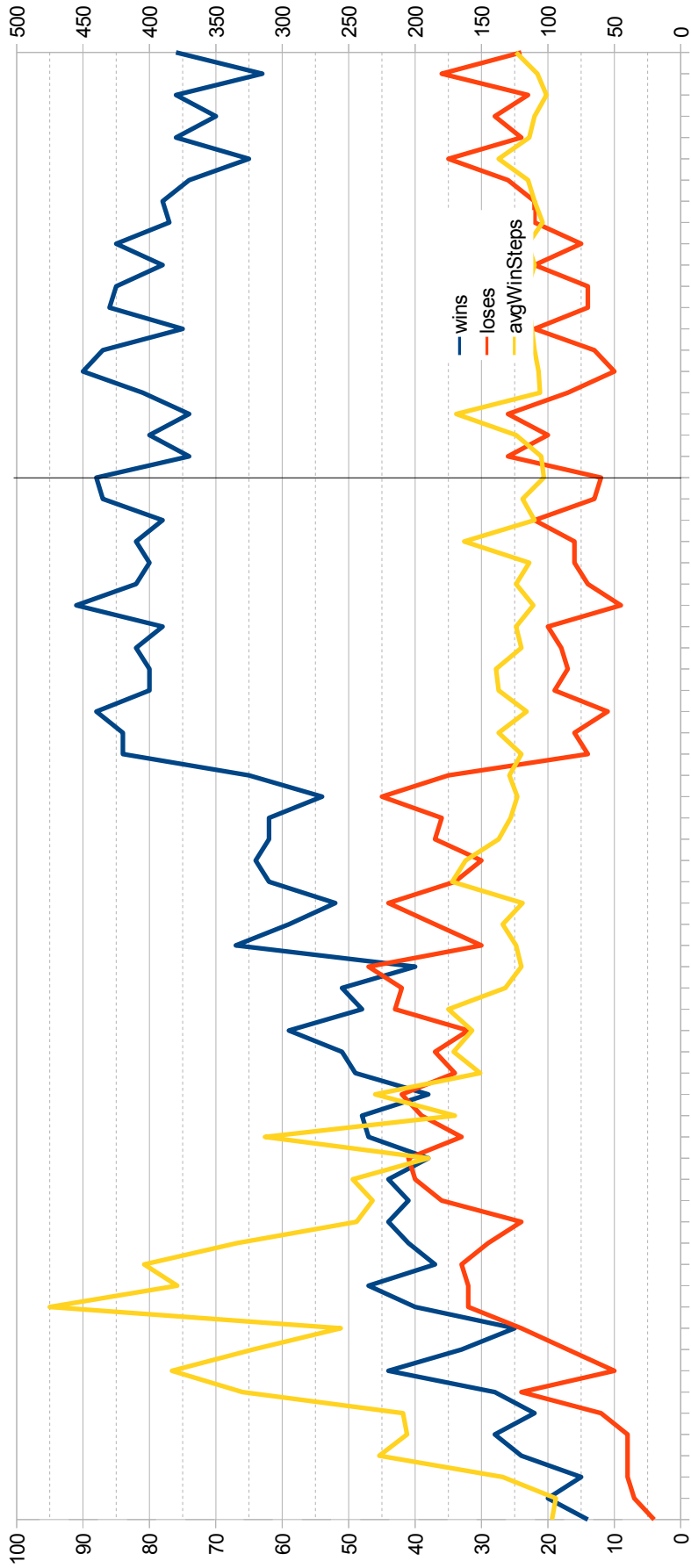


Figura 6.6: Detalle entrenamiento con oponente *Replicant*

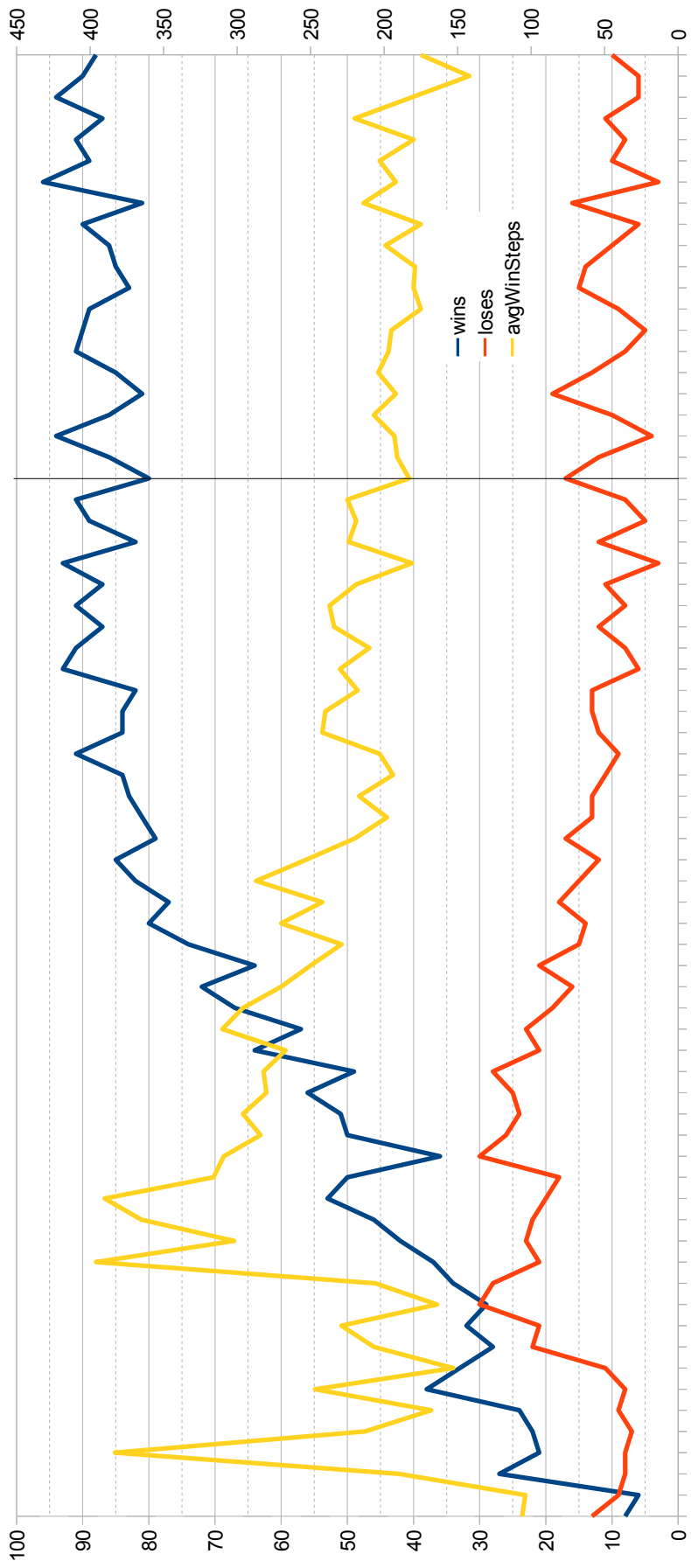


Figura 6.7: Detalle entrenamiento con oponente aleatorio

Evolución de puntos en la memoria

En el gráfico de la Figura 6.8 se puede ver cómo evoluciona la cantidad de puntos en memoria para cada uno de los tipos de entrenamiento. Se puede ver que al llegar a los 5000 episodios la curva se estabiliza hasta llegar a su punto máximo.

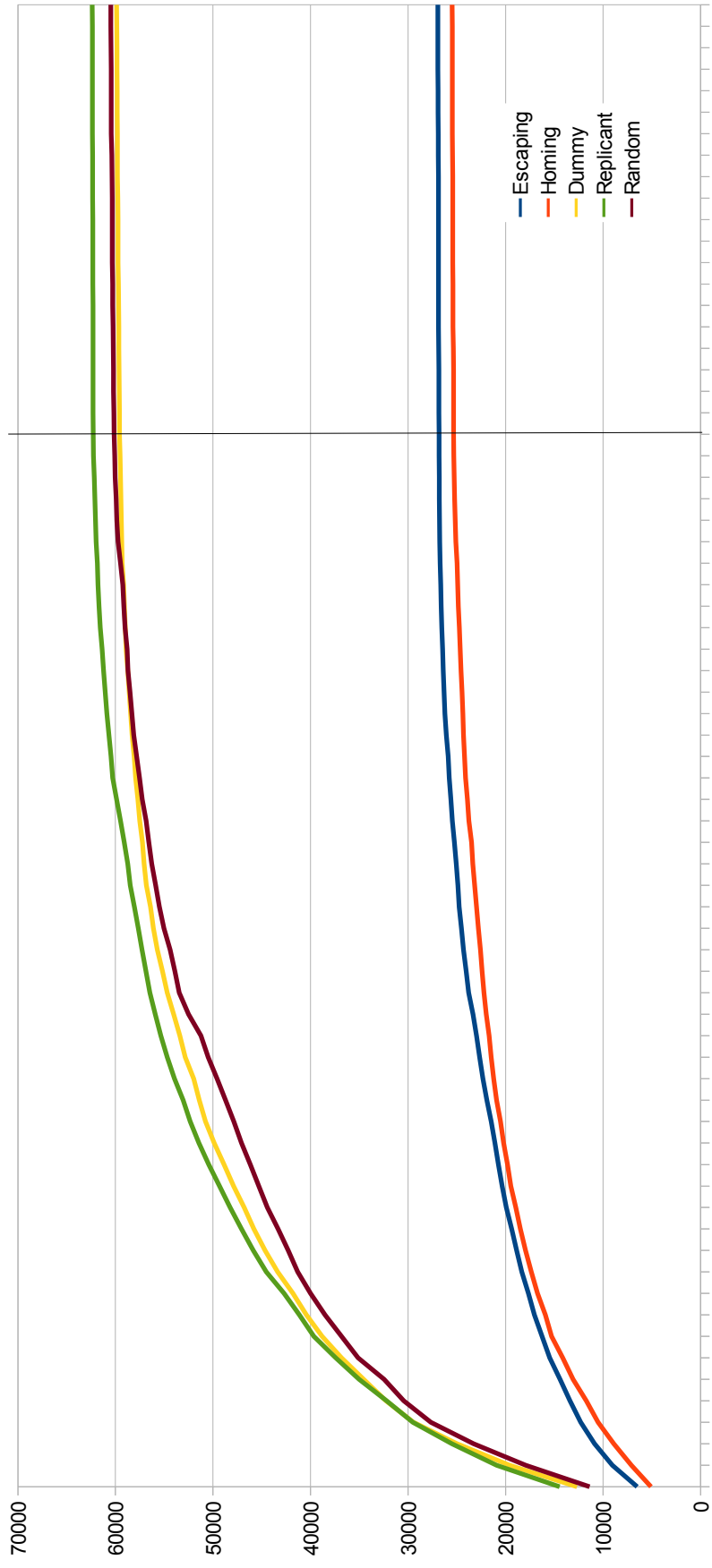


Figura 6.8: Puntos en memoria a lo largo del aprendizaje

6.2.4. Resultados en dimensiones mayores

Se replicó el experimento con el oponente Dummy pero empleando la cantidad completa de dimensiones que el entorno permite sensar, y actuar, es decir: sensado de diferencia de altura y velocidad, y acciones de cambio de *pitch* y velocidad. Esto llevó el espacio de memoria a \mathbb{R}^8 . Puede verse la evolución del aprendizaje en la Figura 6.9. Se eligió el oponente Dummy dado que en el Cuadro 6.1 se ve que el agente tiene buen desempeño contra otros oponentes luego de su entrenamiento, y el tiempo de entrenamiento es menor que con otros oponentes con los que logra también buenos resultados. Se modificaron algunos parámetros del experimento para, dado el aumento de dimensionalidad, poder abarcar mejor el espacio en memoria:

- Distancia de vecindad: 0,2. Permite contrarrestar el aumento en la cantidad de puntos en memoria (por el aumento dimensional)
- Episodios con decrecimiento de ϵ : 7000 (70 corridas). Se incentiva la exploración por más tiempo.
- Episodios con decrecimiento de α : 3000 (30 corridas). Al haber muchos más puntos, son necesarios más episodios para hacer converger los valores estimados de Q .

El entrenamiento finalizó con 342828 puntos en memoria, luego de 24 horas, 23 minutos y 53 segundos (1 día completo). La evaluación contra otro tipo de oponentes puede verse en el Cuadro 6.2. Se puede apreciar que no llegó a adquirir suficiente experiencia como para enfrentar a otros oponentes en forma eficaz.

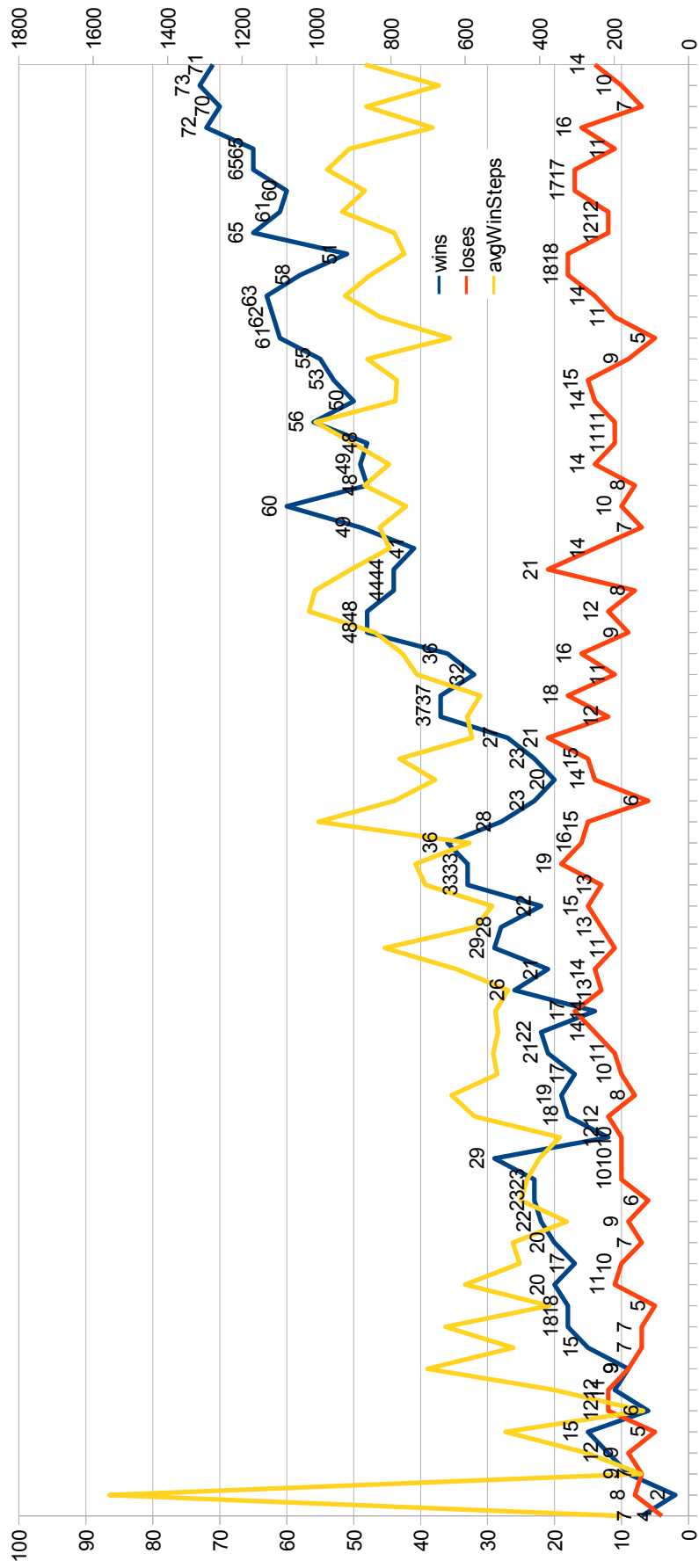


Figura 6.9: Evolución del aprendizaje con oponente Dummy y sensado y acciones completos

Cuadro 6.2: Resultados del entrenamiento con sensado y acciones completas

Entrenamiento				Evaluación (ganados/perdidos)			
Oponente	Lock P.	puntos	Tiempo	D	E	H	R
Dummy	A	342828	24:23:53	76/12	50/0	12/88	53/17

6.3. Resumen

Se vio a lo largo de los experimentos el comportamiento del agente en diversas circunstancias, que incluyen distintos tipos de oponentes, características de la memoria, sensado y acciones, entre otros. La cantidad de posibles combinaciones de éstos parámetros es inmensa e imposible de probar, pero con las evaluadas se obtuvieron resultados que alcanzaron los objetivos de este trabajo. En el siguiente capítulo se expondrán las conclusiones obtenidas a partir de los experimentos, así como posibles extensiones del trabajo y nuevas líneas de experimentación a seguir en el futuro.

Capítulo 7

Conclusiones, observaciones y trabajos futuros

En este capítulo se presentarán las conclusiones del trabajo, posibles modificaciones y trabajos futuros, y las posibilidades de uso de los resultados en juegos reales.

7.1. Resultado final del trabajo

A lo largo de esta Tesis se desarrolló una técnica para entrenar un agente en forma autónoma para realizar maniobras de combate aéreo en el contexto de un videojuego. El núcleo de lo buscado es una *política*, que determine para cada situación (estado) en la que el agente se encuentra, cuál es la mejor acción a realizar.

Para esto se investigó en la literatura (Capítulo 3) usos anteriores de aprendizaje automático aplicado al dominio de los videojuegos en general, dado que en el contexto específico de los simuladores de vuelo no se encontraron trabajos previos. Algunos de estos antecedentes [Gra04] [Diu08] [Man03] usan aprendizaje por refuerzo (Capítulo 2), pero no sobre un espacio de estados continuo, como el tratado aquí.

Para ello se debió recurrir a trabajos con orígenes en la robótica [Vil10], que hacen uso de estimadores de función para aproximar una función de valor y la política del agente.

Luego de determinar la técnica a usar, se desarrolló una herramienta (Capítulo 4) para simular el entorno del videojuego, visualizar el estado del agente, y realizar los experimentos. Esta herramienta proporcionó una capacidad de introspección, muy útil para estudiar el comportamiento del agente. Como el tipo de juego elegido requiere de un oponente, se diseñaron distintos tipos para ayudar en el entrenamiento del agente. Se mostró que esto no es estrictamente necesario dado que se puede entrenar al agente contra si mismo, lo que hace más atractiva la solución dado que no requiere

del desarrollo adicional de oponentes programados manualmente (con los problemas que esto trae y que se discuten en la Sección 1.5).

Se diseñó el agente (Capítulo 5) de forma tal que aprendiera a partir de su experiencia con el entorno simulado usando una memoria multi-dimensional en la que almacena sus experiencias y que actualiza continuamente mientras aprende.

Se realizaron experimentos (Capítulo 6) y encontraron parámetros adecuados para lograr un comportamiento eficaz del agente.

7.2. Conclusiones

El objetivo central de esta Tesis es encontrar formas de aprendizaje automático de maniobras de combate aéreo en el contexto de un videojuego. Este objetivo forma parte de una búsqueda más amplia de mecanismos de aprendizaje automático en juegos en general, con el fin último de proveer herramientas que faciliten el desarrollo de los mismos con menor trabajo y mayor calidad. En particular, es de interés contar con herramientas que permitan tratar problemas con una cantidad grande de dimensiones en su espacio de estados, en el contexto de los videojuegos.

Aunque el entorno elegido para el entrenamiento no es el de un videojuego complejo, se debió trabajar con características que sí lo son, principalmente un espacio de estados, acciones y tiempo continuos. Para abordar estas características se emplearon estimadores de núcleo para la función Q [Vil10] aplicados en forma adecuada al problema. También se encontró una reducción de las características del entorno que permitiera reducir inicialmente el espacio de 14 a 8 variables sin perder información.

Entre los resultados pudo verse que en situaciones con menos cantidad de dimensiones, es posible encontrar una estrategia de entrenamiento que produzca un agente eficaz contra varios tipos de oponentes. Pudo apreciarse también que el tiempo necesario de entrenamiento en este caso es relativamente breve (en el orden de una hora). Al aumentar la cantidad de dimensiones trabajadas (sumando variables al sensado y acciones) se vio que el entrenamiento resulta más costoso en cómputo, tiempo y memoria (cantidad de puntos). En el último experimento, debido al costo computacional del entrenamiento, fue difícil realizar suficientes pruebas como para ajustar los parámetros de forma tal que el agente se acercara al 100 % de efectividad, aunque el grado alcanzado de todas formas es adecuado a los fines buscados por este trabajo. Más adelante se mencionan posibles extensiones que podrían ayudar a disminuir el tiempo de procesamiento.

Pudo verse que el tipo de oponente contra el que se entrena influye fuertemente en la eficacia futura del agente, cuando se lo prueba contra otros tipos de agentes. Se obtuvieron buenos resultados al entrenar al agente contra sí mismo y, sorprendentemente, al entrenar contra oponentes con

comportamiento tan simple como volar en línea recta.

Si el tiempo de cómputo aumenta considerablemente con la complejidad del problema, y el problema planteado en esta Tesis abarca solo dos aviones, ¿Qué pasaría en caso de querer analizar más aviones, o de agregar al entorno otros elementos, como obstáculos? Una posibilidad es que las técnicas aquí planteadas no resultaran adecuadas y que, más allá de posibles optimizaciones, fuera necesario dividir al problema en otros más pequeños, usar *planning* [Rus95], o alguna otra forma de descomposición. De esta forma el agente trabajaría con menos variables, descartando las que resultarían innecesarias o agrupándolas en una estructura jerárquica.

7.3. Implementación dentro de un juego

La simulación desarrollada es independiente de cualquier juego preexistente, y fue creada ad-hoc para este trabajo. Sin embargo la aplicación práctica de esta técnica requeriría realizar el entrenamiento en el entorno mismo del juego.

Las herramientas modernas empleadas en el desarrollo de videojuegos incluyen no sólo las librerías o frameworks en los que se basan para funcionar, sino un entorno complejo de trabajo que permite definir los universos virtuales, manipular datos del juego, definir comportamientos y acciones de NPCs, etc.. Estos componentes adicionales funcionan sobre el mismo *motor* del juego, lo que permite ver inmediatamente el resultado del trabajo de la misma forma que lo percibirá el jugador. Por otro lado, existen herramientas que asisten en la generación de contenidos en forma automática o semi automática para terrenos, árboles e inclusive ciudades. No resulta entonces descabellado pensar que en algún momento las herramientas incluirán la posibilidad de simular y entrenar la IA del juego en el mismo entorno.

Pero mientras esto no suceda, serán necesarias herramientas como la simulación realizada en este trabajo para entrenar agentes, y luego probarlos dentro del entorno del juego.

7.4. Trabajos futuros

Durante el transcurso del trabajo se encontraron varios aspectos que podrían ser mejorados, pero que no aportaban significativamente al resultado final y por lo tanto no se analizaron en mayor profundidad. Se enumeran varios de ellos a continuación, en forma de posibles extensiones de esta Tesis:

7.4.1. Implementar las relaciones de Homotecia y Simetría

Como se mencionó en la Sección 5.2, no se implementaron las relaciones de Homotecia y Simetría en este trabajo. Si se modificara el algoritmo para

incluir esas relaciones entre los puntos de la memoria, se podría ahorrar no solo el uso de memoria sino también potencialmente de tiempo de procesamiento, al necesitar aprender menos puntos. La implementación de esas relaciones requeriría de un análisis más detallado de la relación entre los puntos de la memoria del agente que el realizado para esta Tesis.

7.4.2. Determinar automáticamente el valor óptimo de vecindad

La distancia de vecindad entre puntos es el principal parámetro que define la precisión de la estimación de Q , y uno de los dos parámetros (junto con la cantidad de dimensiones) que definen la cantidad de puntos en la memoria. La precisión en la estimación de la función Q y la cantidad de puntos están íntimamente relacionados: a menor cantidad de puntos para un mismo espacio, peor será la estimación, y a mayor cantidad de puntos mejor será. Sin embargo debería ser posible disminuir la cantidad de puntos de forma tal que el error en la estimación no supere un margen dado; de esta forma se podría disminuir el requerimiento de memoria del agente, manteniendo su eficacia en valores aceptables. Dado que el sensado del agente no puede disminuirse arbitrariamente, el único parámetro modificable es la distancia de vecindad, y sería útil encontrar una forma automática de definirlo.

7.4.3. Optimizar la representación de la memoria para consultas únicamente

En el trabajo se usó una representación de la memoria que contiene la n -upla estado, acción, Q estimada, número de visitas ($s, a, Q, nvis$). Estos valores son usados para el aprendizaje y para determinar la acción óptima en cada estado (su *política*). Sin embargo, cuando el agente deja de aprender y únicamente busca la acción óptima, lo único que importa es su política. En la implementación se encuentra atada también a todas las variables, pero quizás es posible una representación que, sin perder precisión, logre reducir la cantidad de memoria necesaria para determinar la acción óptima en cualquier estado.

7.4.4. Determinar automáticamente las transformaciones de variables al espacio de memoria

La memoria del agente contiene puntos en el espacio $[-1, 1]^n \in \mathbb{R}^n$. Las variables que se toman del entorno se encuentran en un rango generalmente fuera del $[-1, 1]$ y son por lo tanto proyectadas a ese subespacio mediante una función de *mapeo*. En este trabajo esa función se definió manualmente para cada variable. Sería útil determinar automáticamente la función de *mapeo* a partir de datos percibidos del entorno u otro mecanismo, sin necesidad de

definirla manualmente. Adicionalmente, la función actual es lineal, y no se ha probado con funciones que no lo fueran. Otro posible trabajo es evaluar la conveniencia de un tipo u otro en función de cada variable.

7.4.5. Entrenar para distintos niveles de dificultad

En muchos videojuegos se usan *niveles de dificultad*, opciones que varían la capacidad del oponente de forma tal que jugadores de distinta proficiencia puedan disfrutar por igual del juego, sin que resulte demasiado fácil o difícil. Se podría modificar el entrenamiento de forma tal que el agente finalice con un nivel de dificultad determinado. Una alternativa sería terminar el entrenamiento antes de que alcance su máxima proficiencia; otra podría ser entrenarlo contra distintos oponentes.

7.4.6. Aprender tanto del agente como del oponente

La herramienta de entrenamiento se diseñó para que el agente aprenda de sus propias acciones, pero si se modificara para que aprendiera también de las acciones del oponente, podría (potencialmente) mejorar la velocidad de aprendizaje, dado que tendría el doble de refuerzos por cada estado de la simulación.

7.4.7. Meta-Aprendizaje

Hay decenas de parámetros que se pueden ajustar al entrenar al agente, y gran parte del presente trabajo consistió en encontrar los adecuados. Estos parámetros son numéricos o booleanos, y si se los toma como dimensiones de un nuevo espacio, se materializa la posibilidad de realizar un aprendizaje sobre ese *metaespacio de entrenamiento*. La forma de evaluar la idoneidad de un elemento en ese metaespacio sería entrenar al agente y luego ver su proficiencia. El proceso total sería órdenes de magnitud más lento que el actual, por lo que un requisito previo sería encontrar formas de estimar antes de finalizar el entrenamiento si será bueno o no. Para explorar el metaespacio es probable que sea más conveniente un algoritmo como *Simulated Annealing* [Kir83] o algoritmos genéticos, en vez de usar nuevamente AR.

Otra perspectiva desde la que es posible abordar un meta-aprendizaje es en la elección de las características del entorno que se considera relevante sensor. En esta Tesis se seleccionaron 5 variables (que para algunos experimentos se redujeron a 3). Esta elección y reducción de características fue determinada e implementada manualmente, pero para cumplir con el fin de lograr un entrenamiento completamente automático, sería ideal contar con mecanismos de extracción de características relevantes en forma automática.

7.4.8. Detección temprana de proficiencia al entrenar

Como se vió en los resultados, algunos entrenamientos del agente pueden llevar varias horas, y a veces un día o más. Es improbable que se puedan mejorar órdenes de magnitud este tiempo mediante optimización, dado que ya se emplean estructuras eficientes (KD-Tree) para almacenar los puntos en memoria. Una alternativa interesante sería la posibilidad de detectar durante los primeros pasos de un entrenamiento si llegará a buen resultado o no. A partir de los gráficos de la sección de resultados se puede intuir que una estimación lineal o logarítmica de los episodios ganados o perdidos permite aproximar el resultado futuro luego de varios episodios de entrenamiento. A partir de una proyección así, podría detectarse tempranamente si los parámetros de entrenamiento son útiles o no, o compararlos contra otros.

Inclusive se podrían correr entrenamientos en paralelo y comparar en tiempo real su performance parcial, para decidir si cancelar algunos y dedicar más tiempo de procesador a otros.

7.4.9. Paralelización del entrenamiento

La herramienta actualmente permite entrenar en forma serial, es decir, procesando un único episodio, un único paso y una única acción a la vez. Las computadoras modernas tienen 2 o 4 núcleos y la capacidad de ejecutar varios flujos de código (*threads*) en simultáneo. Se podría modificar la herramienta para que ejecutara distintos episodios en paralelo, teniendo siempre cuidado con no modificar las condiciones necesarias definidas en [Wat89] para convergencia en *Q-Learning*.

7.4.10. Optimizaciones

El algoritmo actual de entrenamiento requiere de mucha capacidad de cómputo, principalmente para la búsqueda de vecinos, que se hace dentro de un KD-Tree. No se ha explorado la posibilidad de optimización de este código, u otras estructuras de datos que puedan reemplazar al KD-Tree, y hacerlo podría mejorar los tiempos de procesamiento del entrenamiento. Una posible optimización está en la heurística de división del árbol. Se optó por reconstruirlo completamente cada determinada cantidad de puntos agregados, pero esto puede no ser lo óptimo para la aplicación. También podrían buscarse optimizaciones a nivel del uso del lenguaje, por ejemplo en el manejo de memoria. Se usó Java para el desarrollo del trabajo, que administra automáticamente la memoria y realiza *Garbage Collection*, y a lo largo del código se crean continuamente pequeños objetos que el recolector de memoria debe eliminar.

7.4.11. Tiempo Variable de ejecución

En la mayoría de los videojuegos en tiempo real, el tiempo que transcurre entre cada paso de la simulación no es fijo sino que varía debido a varios factores, entre ellos la necesidad de compartir el uso de CPU con otras áreas del juego (gráficos, IA, etc.) o con otras aplicaciones que se ejecuten en simultáneo. Por lo tanto podría pensarse que entrenar al avión con un tiempo fijo no representa una situación real de uso y que lo aprendido podría no servir al momento de ser empleado en un contexto con tiempo variable entre pasos de la simulación. Sin embargo se ha probado introducir un factor de ruido en el tiempo simulado, y los experimentos parecen sugerir que se llega a iguales resultados en el aprendizaje. Esto se debe probablemente a que los datos en la memoria no están influenciados por el tiempo transcurrido, sino que representan únicamente dimensiones del estado y las acciones. La estimación por núcleos dará el mismo resultado dado un estado, sin importar cuánto tiempo transcurrió desde que se simuló el estado anterior. Un análisis más detallado de este comportamiento sería necesario para afirmar que realmente es irrelevante el ruido introducido.

Bibliografía

- [Abs11] Abzug, M. J., Larrabee E. E. *Airplane stability and control: a history of the technologies that made aviation possible*, Cambridge University Press, 2002
- [Bel57] Bellman, R. *Dynamic Programming*, Princeton University Press, 1957
- [Ben75] Bentley, J. L., *Multidimensional binary search trees used for associative searching*, Communications of the ACM, 18(9):509-517, 1975
- [Ben10] Benotti, L., Bertoa, N., *From game tutorials to game partners using natural language generation techniques*, a publicarse en las actas del AISB 2011 Symposium: AI & Games, 2011
- [Bra02] Brafman, R. I. , Tenenholz, M., *R-max - A General Polynomial Time Algorithm for Near-Optimal Reinforcement Learning*, Journal of Machine Learning Research 3, 2002
- [But10] Butler, S., *Partial Observability During Predictions of the Opponent's Movements in an RTS Game*, Proceedings of the IEEE Conference on Computational Intelligence and Games, 46-53, 2010
- [Con10] Conde, C., Moreno, M., and Martínez, D.C., *Testing real-time artificial intelligence: an experience with Starcraft game*, Actas del Primer Workshop Argentino de Videojuegos, 11-16, 2010
- [Cun10] Freitas Cunha, R L., Chaimowicz, L., *An Artificial Intelligence system to help the player of Real-Time Strategy games*, Proceedings of SBGames 2010, 74-81, 2010
- [Die00] Dietterich, T. G. (2000). *Hierarchical reinforcement learning with the MAXQ value function decomposition*. Journal of Artificial Intelligence Research, 13, 227–303, 2000
- [Diu08] Diuk, C., Cohen, A., Littman, M. L., *An Object-Oriented Representation for Efficient Reinforcement Learning*, Proceedings of the 25th international conference on Machine learning, pages. 240-247, 2008

- [Gra04] Graepel, T., Herbrich, R., Gold, J., *Learning to fight*, Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education, January 2004
- [Har87] D. Harel, *Statecharts: A Visual Formalism for Complex Systems*, Science of Computer Programming 8, 1987
- [Has07] Hastings, E., Guha, E., Stanley K. O., *NEAT Particles: Design, Representation, and Animation of Particle System Effects*, Proceedings of the IEEE 2007 Symposium on Computational Intelligence and Games, 2007
- [Has10] Hastings, E., Stanley, K.O., *Interactive Genetic Engineering of Evolved Video Game Content*, Proceedings of the Workshop on Procedural Content Generation in Games (PCG) at the 5th International Conference on the Foundations of Digital Games, 2010
- [Imm34] Immelmann, Frantz. Der Adler von Lille. Liepzig: K.F. Koehler Verlag, 1934
- [Jon99] Randolph M. Jones, John E. Laird, Paul E. Nielsen, Karen J. Coulter, Patrick Kenny, and Frank V. Koss *Automated Intelligent Pilots for Combat Flight Simulation*, AI Magazine Volume 20 Number 1 (1999) pags. 27-41
- [Kae96] Kaelbling L.P., Littman, M.L., Moore, A.W., *Reinforcement Learning: A Survey*, Journal of Artificial Intelligence Research 4 237-285, 1996
- [Kir83] Kirkpatrick, S.; Gelatt, C. D. , Vecchi, M. P., *Optimization by Simulated Annealing*. Science. New Series 220 (4598): 671–680, 1983
- [Lee09] Lee, C.S et.al *The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments*, IEEE Transactions on Computational Intelligence and AI in games, 2009
- [Man03] Manning, H., *Learning to play chess using reinforcement learning with database games*, Master thesis Cognitive artificial intelligence, Utrecht University, 2003
- [McC43] McCullock, W.; Pitts, W. *A Logical Calculus of Ideas Immanent in Nervous Activity*. Bulletin of Mathematical Biophysics 5: 115-133, 1943
- [McG08] James S. McGrew, *Real-Time Maneuvering Decisions for Autonomous Air Combat*, Msc. Thesis Massachusetts Institute of Technology, 2008

- [McP08] Michelle McPartland, Marcus Gallagher *Learning to be a Bot: Reinforcement Learning in Shooter Games*, Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, pages. 78-83, 2008
- [Mor10] Mora, A.M., Moreno, M.A., Merelo, J.J., Castillo, P.A., Arenas M.G., Laredo, J.L.J., *Evolving the Cooperative Behaviour in UnrealTM Bots*, Proceedings of the IEEE Conference on Computational Intelligence and Games, 241-248, 2010
- [Nad64] Nadaraya, E. A., *On Estimating Regression*, Theory of Probability and its Applications 9: 141–142, 1964
- [Orm99] Ormonet, D., Sen, S. *Kernel-based reinforcement learning*, Tech. Report TR 1999-8, Statistic, Standford University, 1999
- [Par10] Parpaglione, C., Santos, J.M., *Sensing capability discovering in Multi Agent Systems*, XXIX International Conference of the Chilean Computer Society, 2010
- [Pat10] Patrick, M., *Online Evolution in Unreal Tournament 2004*, Proceedings of the IEEE Conference on Computational Intelligence and Games, 249-250, 2010
- [Rot82] Roth, Scott D., *Ray Casting for Modeling Solids*, Computer Graphics and Image Processing 18: 109–144, 1982
- [Rum94] Rummery, G., M. Niranjan, *On-line Q-learning using connectionist systems*, Technical Report CUED/F-INFENG/TR 166, Cambridge University, Engineering Department, 2004
- [Rus95] Russell, Stuart J., Norvig, Peter, *Artificial Intelligence: a modern approach*, Prentice Hall, 1995
- [Sch07] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, Steve Sutphen: *Checkers is Solved*, Scienceexpress, 2007
- [Smi10] Smith, G., Avery, P., Houmanfar, R., Louis, S., *Using Co-evolved RTS Opponents to Teach Spatial Tactics*, Proceedings of the IEEE Conference on Computational Intelligence and Games, 146-153, 2010
- [Sta05] Stanley, K.O., Bryant, B. D. , Miikkulainen, R. *Evolving Neural Network Agents in the NERO Video Game*, Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games, 2005
- [Sta07] Stanley, K.O., *Compositional Pattern Producing Networks: A Novel Abstraction of Development*, Genetic Programming and Evolvable

Machines Special Issue on Developmental Systems, New York, NY: Springer, 2007

- [Sut98] Sutton, Richard S.; Andrew G. Barto (1998), *Reinforcement Learning: An Introduction*. MIT Press. ISBN 0-262-19398-1, 1998
- [Tes95] Tesauro, G., *Temporal Difference Learning and TD-Gammon*, Communications of the ACM, Vol. 38, No. 3, pags. 58-68, Marzo 1995
- [Tho06] Thompson, T., *EvoTanks II Co-evolutionary Development of Game Playing Agents*, Master Thesis, School of Informatics, University of Edinburgh, 2006
- [Toz02] Paul Tozour *The Evolution of Game AI*, AI Game Programming Wisdom 1, pags. 3-15, 2002
- [Vil10] Villar, J., Santos, J. *Robots que Aprenden: aproximación por Suavizado por Núcleos de la función Q y clases de equivalencias por Simetrías y Homotecias en espacios continuos*, Jornadas Argentinas de Robótica, 2010
- [Wat89] Watkins, Chris, *Learning from delayed rewards*, Ph.D. thesis, Cambridge University, 1989

Índice de figuras

1.1. Ejemplos de maniobras aereas: Barril (loop), giro de Immelmannn, Cobra de Pugachev	5
1.2. Ejes del avión y posibles dimensiones del movimiento	5
3.1. Captura de pantalla del juego <i>Neuro-Evolving Robotic Operatives (NERO)</i>	26
3.2. Captura de pantalla del juego <i>Tao Feng: Fist of the Lotus</i>	28
4.1. Proyección de la pirámide de lock	31
4.2. Estructura de comunicación entre agente, oponente y entorno	32
4.3. Captura de pantalla de la vista de simulación	34
4.4. Detalle de la visualización, con el agente haciendo lock sobre el oponente, lock mutuo, y diferencia de altura entre los aviones	35
4.5. Secuencia de maniobra de ataque	35
4.6. Captura de pantalla de la vista de memoria, con los controles para ajustar cada dimensión y botón para refrescar	36
4.7. Vistas de la misma memoria, usando distintos parámetros en cada dimensión (x,y)	37
5.1. Diferencia de yaw y Yaw relativo del oponente	41
6.1. Posición inicial de los agentes en el experimento preliminar	50
6.2. Detalle experimento preliminar	52
6.3. Detalle entrenamiento con oponente <i>Dummy</i>	57
6.4. Detalle entrenamiento con oponente <i>Escaping</i>	58
6.5. Detalle entrenamiento con oponente <i>Homing</i>	59
6.6. Detalle entrenamiento con oponente <i>Replicant</i>	60
6.7. Detalle entrenamiento con oponente aleatorio	61
6.8. Puntos en memoria a lo largo del aprendizaje	63
6.9. Evolución del aprendizaje con oponente Dummy y sensado y acciones completos	65

Índice de cuadros

6.1. Resultados de la evaluación cruzada de oponentes	55
6.2. Resultados del entrenamiento con sensado y acciones completas	66