Centro de Cálculo, Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

PROYECTO DE GRADO INGENIERÍA EN COMPUTACIÓN

Neuroevolución aplicada a la generación automática de inteligencias artificiales para verificación de videojuegos

Facundo Parodi Moraes

Sebastián Rodríguez Leopold

Abril de 2017

Supervisor del proyecto: Sergio Nesmachnow

Neuroevolución aplicada a la generación automática de inteligencias artificiales para verificación de videojuegos
Parodi Moraes, Facundo
Rodríguez Leopold, Sebastián
Proyecto de grado de Ingeniería en Computación
Instituto de Computación - Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, abril de 2017

Neuroevolución aplicada a la generación automática de inteligencias artificiales para verificación de videojuegos

Resumen

Este proyecto trata sobre la aplicación de técnicas de computación evolutiva, inferencia de objetivos y computación de alto desempeñó aplicadas al desarrollo automatizado de inteligencias artificiales para la verificación de juegos de la consola Nintendo Entertainment System. En particular, se concentró el estudio sobre ocho juegos de esta plataforma seleccionados para que cada uno presente retos distintos al proceso de aprendizaje. El sistema desarrollado consiste en un pipeline de tres fases. La primera fase aborda la inferencia de objetivos mediante un conjunto de algoritmos que se aplican sobre una serie de videos de entrenamiento con el propósito de inferir los objetivos de un juego. La segunda fase propone el refinamiento de objetivos para mejorar la inferencia del paso anterior. Por último, la fase de generación de inteligencias artificiales aplica neuroevolución para generar jugadores automáticos para al juego en cuestión, tomando en cuenta los objetivos refinados en la fase anterior. Las técnicas de generación desarrolladas produjeron inteligencias artificiales competentes para los juegos estudiados y permitieron detectar errores de programación y diseño. Los resultados sugieren que el enfoque es aplicable para realizar tareas de verificación automatizada de videojuegos.

Palabras clave: Algoritmos Evolutivos, Neuroevolución, Redes neuronales artificiales recurrentes, Inteligencia Artificial, Nintendo Entertainment System

Índice general

1 Introducción **1**

2 Descripción del problema **5**
2.1 Modelado y obtención del estado del juego 5
2.2 Evaluación de una partida 6
2.3 Inferencia de los objetivos del juego 7
2.3.1 Inferencia de prefijos 7
2.3.2 Enmascarado de valores de RAM 8
2.3.3 Inferencia de órdenes lexicográficos 9
2.4 Refinamiento de objetivos 12
2.5 Generación de inteligencias artificiales 15

3 Técnicas evolutivas, redes neuronales y neuroevolución **17**
3.1 Algoritmos Evolutivos 17
3.1.1 Elementos básicos de un algoritmo evolutivo 17
3.1.2 Individuo 18
3.1.3 Función de fitness 18
3.1.4 Inicialización 18
3.1.5 Selección 18
3.1.6 Operadores evolutivos 19
3.1.7 Criterio de parada 20
3.1.8 Uso de elitismo 20
3.2 Algoritmos evolutivos paralelos 20
3.2.1 Evaluación paralela de soluciones 21
3.2.2 Subpoblaciones distribuidas 21
3.3 Redes neuronales artificiales 22
3.3.1 Definición de una red neuronal artificial 22
3.3.2 Entrenamiento 24
3.3.3 Evaluación 25
3.3.4 Redes Neuronales Recurrentes 25
3.4 Neuroevolución 25
3.4.1 Conceptos generales 26
3.4.2 NEAT 26

4 Trabajos relacionados **35**
4.1 Panorama de técnicas de neuroevolución 35
4.2 Robocode 36
4.3 Super Mario 37
4.4 Atari 38

4.5 Playfun 39
4.6 Resumen y conclusión 41

5 Algoritmos evolutivos aplicados a la generación automática de inteligencias artificiales para NES 43
5.1 Introducción 43
5.2 Framework Evolutivo 44
5.2.1 Optimización del desempeño computacional del framework evolutivo 48
5.2.2 Ejemplos relevantes de código paralelizado 49
5.3 Emulador FCEUX 51
5.4 Inferencia de objetivos 52
5.4.1 Detección de prefijo mínimo 53
5.4.2 Detección de valores estáticos 53
5.4.3 Detección de valores erráticos 53
5.4.4 Detección de valores con bajo dinamismo 53
5.4.5 Detección de órdenes lexicográficos 54
5.4.6 Detección de valores independientes 54
5.4.7 Inferencia de pesos candidatos 55
5.5 Algoritmo evolutivo para el refinamiento de objetivos 56
5.5.1 Operadores evolutivos 57
5.5.2 Configuración paramétrica 57
5.6 Algoritmo Evolutivo para el entrenamiento de inteligencias artificiales . . 57
5.6.1 Operadores evolutivos 58
5.6.2 Configuración paramétrica 58
5.6.3 Variaciones sobre el modelo NEAT 58
5.6.4 Evaluación de la función de fitness 59
5.7 Implementación de RNNs 60
5.7.1 Representación de la red 60
5.7.2 Evaluación de la red neuronal recurrente 61
5.7.3 Funciones de activación 62

6 Análisis experimental 67
6.1 Metodología del análisis experimental 67
6.2 Experimentos de calibración paramétrica 68
6.3 Evaluación de la implementación 71
6.3.1 Comparación entre inferencia automática de objetivos y especificación manual 71
6.3.2 Comparación con jugadores humanos 81
6.3.3 Comparación con algoritmo evolutivo especializado 86
6.4 Aplicabilidad como herramienta de verificación 87
6.5 Determinación empírica de la eficiencia computacional del algoritmo de entrenamiento de Inteligencias Artificiales 89

7 Conclusiones y trabajo futuro 93
7.1 Conclusiones 93
7.2 Trabajo futuro 94

Bibliografía 95

Índice de figuras

3.1 Modelo maestro-esclavo 21
3.2 Modelo de subpoblaciones distribuidas 22
3.3 Ejemplo de ANN feed-forward 23
3.4 Esquema de un perceptrón 23
3.5 Codificación de una red neuronal en NEAT 28
3.6 Ejemplo de cruzamiento en NEAT 29
3.7 Ejemplo de mutación que agrega un nodo con peso unitario 30

5.1 Componentes del pipeline de generación de IAs 44
5.2 Diagrama de clases del framework implementado 46
5.3 Algoritmo evolutivo paralelo en 2 niveles 47
5.4 Objetivos escalados según el máximo para Battle City 56
5.5 Objetivos escalados según la tangente hiperbólica para Battle City 56
5.6 Objetivos escalados según el máximo para Contra 56
5.7 Objetivos escalados según la tangente hiperbólica para Contra 56
5.8 Función identidad 63
5.9 Función sigmoide para $k = 1$ 63
5.10 Función softsign para $k = 1$ 64
5.11 $tanh$ para $k = 1$ 64
5.12 Gaussiana para $m = 0$ y $s = 1$ 65
5.13 PReLU para $\alpha = 0{,}1$ 65

6.1 Diagrama de caja de las configuraciones del algoritmo de generación de IAs 70
6.2 Ejemplos de situaciones en las que el jugador automático logra explotar errores en la programación del juego Super Mario Brothers 72
6.3 Jugador artificial disparando una gran cantidad de balas en simultáneo en el juego Contra 74
6.4 Jugador artificial disparando a un platillo volador entre los alienígenas comunes en el juego Space Invaders 75
6.5 Rebote de la bola en el bumper en el juego Pinball 77
6.6 Gol anotado utilizando al jugador más corpulento en el juego Ice Hockey . 79
6.7 Menú para selección de juegos y tabla de puntajes 82
6.8 Tiempo requerido para la evaluación de la función de fitness en función de la generación 90
6.9 Tiempo requerido para el cruzamiento y mutación en función de la generación 90
6.10 Gráfica del tiempo de ejecución al variar el número de hilos 91

Índice de tablas

4.1 Resumen de los trabajos analizados 41

6.1 Análisis paramétrico de la etapa de inferencia 68
6.2 Análisis paramétrico de la etapa de refinamiento de objetivos 69
6.3 Resultados del análisis paramétrico de la etapa de generación de IAs . . . 70
6.4 Resultados de jugadores humanos 82
6.5 Resultados de los jugadores generados por el prototipo 86
6.6 Resultados de eficiencia computacional del algoritmo evolutivo (tiempo de ejecución por generación) 89
6.7 Tiempo de ejecución al variar el número de hilos utilizados para evaluación de la función de fitness en la versión paralela ejecutando en Xeon Phi . . 91

Índice de algoritmos

1 Esquema de un algoritmo evolutivo 18
2 Esquema de un algoritmo NEAT 27
3 Esquema del operador de cruzamiento 30
4 Búsqueda en profundidad para calcular la profundidad de la red 61
5 Evaluación de la red neuronal 62

Capítulo 1

Introducción

La industria de videojuegos es la más grande de las industrias de entretenimiento, presentando un crecimiento económico muy importante que se prevé será mantenido en los próximos años (Takahashi, 2016). Este crecimiento viene acompañado de una exigencia cada vez mayor por parte de los consumidores, que esperan juegos de mayor calidad en plazos cada vez menores. La expectativa de los jugadores genera presión en varias etapas del desarrollo, lo que puede desembocar en múltiples postergaciones de la fecha de entrega, o en la liberación de productos sin terminar. Tales decepciones para los jugadores repercuten de manera negativa en la popularidad de la marca afectada (Khaw, 2016).

Por otro lado, en las últimas décadas el desarrollo de inteligencias artificiales (IAs) para juegos ha visto grandes avances (Yannakakis y Togelius, 2015). Las IAs son utilizadas en prácticamente cualquier videojuego, desde el control de personajes secundarios (NPCs, Non-Player Characters) hasta el manejo de varios aspectos del mundo jugable, como dificultad adaptativa (Booth, 2009). Además, existen ejemplos donde IAs han mostrado potencial como herramienta de verificación automatizada, al encontrar no solo errores de codificación sino también de diseño de niveles. Un ejemplo de una IA desarrollada con fines de verificación es la utilizada por el estudio Croteam para verificar su juego *The Talos Principle* (Newman, 2014). Con la estrategia propuesta fue posible realizar la mayor parte de la verificación del juego de forma automatizada reduciendo los costos y el tiempo requerido para realizar la etapa de verificación.

Sin embargo, desarrollar IAs para juegos es un proceso complejo que requiere de gran esfuerzo por parte del desarrollador: automatizar este proceso permite reducir enormemente el esfuerzo de desarrollo de las IAs incurriendo en un mayor costo computacional (Simpson, 2012). El tiempo ahorrado puede destinarse para tareas complejas que requieren inherentemente de intervención humana, tales como aspectos relacionados con el desarrollo de la historia y aspectos estéticos del videojuego.

Para realizar eficientemente la tarea de verificación automatizada se propone en este trabajo el uso de algoritmos evolutivos. Estos algoritmos imitan el proceso evolutivo natural para lograr un refinamiento progresivo de las soluciones consideradas hasta llegar a cierto criterio de calidad o esfuerzo computacional. Los algoritmos evolutivos presentan una buena capacidad para manejar amplios espacios de búsqueda, lo que los hace idóneos para el desarrollo de IAs. Las IAs presentes en juegos modernos tienen una gran cantidad de parámetros que regulan su comportamiento, lo que genera espacios de búsqueda de alta dimensionalidad. A modo de ejemplo, Cole et al. (2004) aplicaron exitosamen-

te algoritmos evolutivos para optimizar doce parámetros de las IAs del juego Counter Strike (preferencias por armas, nivel de agresividad del jugador y uso de granadas). Los parámetros generados por el algoritmo evolutivo son comparables a los generados por humanos expertos en el juego, pero requiriendo mucho menos tiempo para encontrar buenos valores de los parámetros.

Además de la generación automática de la IA propiamente dicha, otro problema complejo es la inferencia automática de objetivos. La inferencia intenta comprender qué objetivos persigue un jugador humano en una partida dada. La complejidad de la etapa de inferencia radica en comprender la semántica del juego en cuestión. Para simplificar este problema, en este proyecto se opta por utilizar un conjunto de técnicas de inferencia en combinación con un algoritmo evolutivo para refinar los resultados obtenidos. Para estas etapas se utiliza como entrada un conjunto de videos de entrenamiento provistos por jugadores humanos. Analizando estos videos se pretende obtener una función objetivo que describa de la forma más ajustada posible los objetivos que dichos jugadores humanos persiguen durante sus partidas.

La combinación de ambas partes, inferencia de objetivos y entrenamiento de IAs, permite automatizar el proceso de desarrollo de una IA competente. Las IAs generadas pueden ser utilizadas para expandir el espectro de juegos donde puede disponerse de un jugador artificial. También pueden utilizarse para controlar jugadores secundarios, reduciendo aún más el esfuerzo y costo de la etapa de desarrollo. La automatización de la tarea de generación de IAs aumenta su viabilidad como herramienta de verificación y alivia en parte la presión generada por las demandas cada vez más grandes sobre la industria, mientras redunda en una mejora de la calidad del código producido (Röken y Frommhold, 2005).

Otra ventaja brindada por la automatización del proceso de aprendizaje es la obtención de una nueva mirada sobre el significado de un juego y las mecánicas que rigen su realidad. Una computadora no posee preconceptos humanos, por lo que puede hallar soluciones poco ortodoxas o tácticas inesperadas que resulten ser muy buenas para solucionar un juego particular. Estas estrategias, además de mejorar la habilidad de la IA para cada juego, suelen presentar un valor estético en sí mismo. Muchos jugadores disfrutan de ver a otros completando partidas de sus juegos favoritos con estrategias diferentes a las que ellos mismos emplearon para jugarlas. Esto da lugar a la existencia de fenómenos culturales como los Let's Plays, o los TAS (Tool Assisted Speedruns), y de eventos como AGDQ (Awesome Games Done Quick) (Games Done Quick, 2017), además de otras IAs que han llamado la atención de la comunidad por medio de sus resultados inesperados en diversos juegos, como Playfun (Murphy, 2013). Este valor agregado de entretenimiento puede ser suficiente para justificar la creación de IAs en juegos que de otro modo no generarían ningún interés en la época actual.

El uso de IAs puede hacer más eficiente el proceso de verificación. El proceso tradicional de pruebas de un juego recurre a jugadores humanos (conocidos como *beta testers*) que son contratados para jugar partidas reiteradamente con el objetivo de encontrar errores en el juego. El componente humano en el proceso de verificación implica que el ritmo en que se pueden encontrar defectos está condicionado por la velocidad humana de juego y la habilidad de los beta testers. El cansancio y las largas jornadas laborales perjudican la efectividad de la verificación. Las IAs, en cambio, no necesitan descansar, presentando una habilidad de juego sostenida a lo largo de las partidas. No se ven afectadas por la presión emocional de las fechas límite o diferentes factores de trabajo, logrando

un desempeño constante durante el período de verificación asignado. Presentan además otra ventaja significativa, ya que son capaces de jugar partidas a mayor velocidad que los jugadores humanos. A modo de ejemplo, un prototipo anterior del proyecto presentado en este trabajo fue capaz de jugar más de 79 años de Pinball en una semana. Esta capacidad de juego acelerada les da a las IAs una ventaja muy favorecedora en comparación al uso de beta testers, al permitir ahorrar una gran cantidad de tiempo de verificación. El tiempo ganado puede ser invertido en mejorar la calidad del producto, o en reducir el tiempo de salida al mercado realizando de igual manera una buena verificación, ventaja muy deseable en el mercado actual altamente competitivo.

La generación automática de IAs puede reducir considerablemente el esfuerzo humano involucrado en el desarrollo de dichas IAs. Además, los jugadores resultantes pueden ser utilizados para realizar tareas de verificación, lo que disminuye aún más la intervención humana. El uso de IAs también aporta una óptica muy diferente a la humana sobre los problemas tratados. Mejorar las técnicas actuales de automatización permitiría obtener mejores resultados con menos recursos computacionales, fomentando su uso en problemas cada vez más complejos.

Este proyecto presenta varios aportes principales. Se presenta un estudio del problema de inferencia de objetivos y de aprendizaje automático enfocado en aprender a jugar juegos de la consola NES (Nintendo Entertainment System). A continuación, se describe la implementación de un framework para algoritmos evolutivos con soporte para neuroevolución utilizando una variante de NEAT (NeuroEvolution of Augmented Topolgies). También se presenta una implementación de redes neuronales recurrentes y un sistema de evaluación que no impone restricciones en la estructura de la red. Finalmente, se detalla el diseño de un pipeline de aprendizaje automático utilizando videos de entrenamiento para los juegos de la consola NES.

Los resultados obtenidos sugieren que el sistema implementado es capaz de aprender de forma automática cómo jugar diversos juegos de la plataforma NES. Algunos de los jugadores generados mostraron utilidad para tareas de verificación, al encontrar y explotar errores de diseño presentes en algunos juegos para obtener cantidades ilimitadas de puntaje. También se encontraron errores de programación que permitieron obtener ventajas significativas a los jugadores artificiales. Sin embargo, no todos los juegos estudiados pudieron ser aprendidos de forma automática, principalmente por la complejidad de sus objetivos y la necesidad de una capacidad de visión a largo plazo y planificación.

El resto de este documento se organiza como se detalla a continuación. El capítulo 2 presenta los problemas tratados durante el desarrollo del proyecto. El capítulo 3 introduce las principales técnicas utilizadas para implementar el sistema. Una reseña de la literatura relacionada más relevante se presenta en el capítulo 4. El capítulo 5 explica la solución desarrollada en este trabajo mientras que el capítulo 6 resume los resultados del análisis experimental del sistema implementado. Finalmente, el capítulo 7 plantea las principales conclusiones del proyecto y las líneas de trabajo futuro.

Capítulo 2

Descripción del problema

El problema a estudiar comprende la generación automática de IAs para juegos de la plataforma NES. Teniendo en cuenta la importancia y el esfuerzo dedicado a los diferentes aspectos que lo componen, se detectaron los siguientes cinco problemas principales: el modelado y obtención del estado del juego, la evaluación de una partida, la inferencia de los objetivos del juego, el refinamiento de objetivos y la generación de IAs.

Durante el transcurso del proyecto también se identificaron otros problemas de menor complejidad, tales como: la optimización del tiempo de simulación, la ejecución concurrente de evaluaciones, la comunicación entre componentes de la solución, entre otros.

En este capítulo se detalla el modelado de los problemas más importantes encontrados durante el transcurso de este proyecto.

2.1. Modelado y obtención del estado del juego

Obtener información sobre el estado del juego es un problema complejo, que puede ser resuelto de varias maneras. Las técnicas más comunes para abordar este problema abarcan estudiar la salida en pantalla utilizando técnicas de *visión por computadora*, y la lectura directa de los datos alojados en RAM. En ambos casos se presenta el problema de la ausencia de semántica asociada a los elementos analizados (píxeles o valores de RAM). Por este motivo es necesario incorporar una etapa de procesamiento del estado del juego para inferir la semántica subyacente de los datos obtenidos en bruto, con el objetivo de poder trabajar con ellos. En particular, debe priorizarse la atención brindada a ciertos componentes del estado del juego según la importancia que tengan para el juego en cuestión, como pueden ser la vida de un personaje o la posición en el nivel de los enemigos. La interpretación del estado se encuentra entonces profundamente relacionada con la inferencia de objetivos del juego, problema tratado en la sección 2.3.

Para modelar el estado del juego se optó por leer directamente los valores de RAM mantenidos por un emulador de la consola NES, requiriendo modificaciones detalladas en la sección 5.3. El estado del emulador en un momento dado del tiempo es modelado como un arreglo de 2048 bytes, que recibe el nombre de *frame*. La NES funciona con una tasa de refresco de 59,94Hz, de acuerdo a la norma NTSC (National Television System Committee) utilizada en el mercado estadounidense, que trabaja a la tasa de refresco de 59,94Hz. Dado que el estado es actualizado en cada frame, un segundo de juego genera aproximadamente 60 frames. Cada valor de RAM contenido en el frame es direccionado por su índice en el arreglo. Teniendo en cuenta que la NES tiene 2048 bytes de RAM y es capaz de direccionar de a un *byte* de memoria, las direcciones válidas pertenecen al rango $[1 \ldots 2048]$.

Se denomina *video* a una sucesión de frames capturados a lo largo de la simulación de una partida. Utilizando la notación $\mathcal{M}_{a\times b}(D)$ para identificar al conjunto las matrices de a filas y b columnas cuyos valores pertenecen al dominio D, se define un video V como una matriz de bytes: $V \in \mathcal{M}_{frames\times 2048}(\mathbb{S})$, donde $\mathbb{S} = \{n \in \mathbb{N} : n < 256\}$. De esta forma, un frame se corresponde con una fila de la matriz.
Este modelo interpreta todos los valores de RAM como números sin signo. Si bien esto no es totalmente cierto en la práctica, la mayor parte de los valores en los juegos estudiados lo son. En particular, los valores que participan en los objetivos principales (puntaje, vidas, avance en los niveles) de todos los juegos estudiados en este proyecto no poseen signo. Existen pocas direcciones de RAM en los juegos estudiados que utilicen valores con signo, por lo que la cantidad de direcciones interpretadas de manera errónea es muy baja, afectando de manera poco significativa la calidad de los resultados.

2.2. Evaluación de una partida

Una vez modelado el estado del juego, es necesario definir un criterio de evaluación para poder discriminar la calidad de las diferentes partidas. Para ello se define una función objetivo que permite evaluar una partida y su evolución a lo largo del tiempo.

Inicialmente se consideró evaluar solamente el estado final de la partida (tomar en cuenta el último frame). Si bien puede parecer razonable en principio, esta decisión disminuye significativamente la robustez de la evaluación del juego. El problema puede verse con un ejemplo: evaluar una partida que acaba de perder luego de obtener un buen puntaje ignora todo el progreso del jugador, clasificando la partida de la peor forma posible (por haber perdido). En cambio, un jugador que no avanza del inicio del juego obtendrá un mayor puntaje, simplemente por mantenerse vivo. Además, es posible obtener valores espurios en frames específicos del juego, una situación que frecuentemente sucede en las transiciones entre niveles, lo que imposibilita obtener una evaluación certera con esta técnica. Versiones iniciales de este proyecto generaron un jugador artificial para el juego Super Mario Bros que se mantenía con vida hasta el último frame, momento en el cual perdía. Este jugador obtuvo una valoración muy elevada debido a que la cantidad de vidas restantes cambió de 0 a 255 al restar una unidad y sufrir un problema de underflow.

Para resolver estos problemas se utiliza la función F definida en la ecuación 2.2, que evalúa el progreso del jugador a lo largo de una partida, capturando la naturaleza secuencial de los juegos.

$$eval(f) = \sum_{j=1}^{2048} f_j * w_j \tag{2.1}$$

$$F(V) = \sum_{i=1}^{cF} eval(V_{(i)}) \tag{2.2}$$

El término $V_{(f)}$ en la ecuación 2.2 representa la fila f de la matriz V (el frame f del video V), el término cF es la cantidad de frames del video y w_j es un peso que permite valorar la importancia del valor f_j.

Los pesos de la función *eval* son definidos a partir de las etapas de *inferencia* y *refinamiento* que se explican más adelante. Los pesos tienen signo, lo que permite indicar que el crecimiento de ciertos valores de RAM es contraproducente para el jugador. De esta forma se posibilita que algunos frames obtengan una evaluación negativa, decrementando el valor de F ante situaciones desfavorables. Para evaluar el progreso de una partida se considera el crecimiento de la función F a medida que transcurre el tiempo, obteniendo un número único que resume el desempeño del jugador.

2.3. Inferencia de los objetivos del juego

Para realizar la inferencia de objetivos se requiere de un conjunto de videos de entrenamiento que deben haber sido previamente clasificados en buenos y malos, según su desempeño en el juego. Es necesario luego realizar un análisis para detectar las relaciones que los gobiernan y así poder inferir los objetivos del juego. A partir de esta información se genera un conjunto de pesos candidatos para cada byte de la RAM que resume dichas relaciones y sirve de insumo para la etapa de refinamiento. Cada video de entrenamiento puede alcanzar un gran tamaño, dado que una partida promedio dura varios minutos y cada segundo corresponde a aproximadamente 60 frames. Con el objetivo de reducir la cantidad de datos a analizar se emplea una etapa de filtrado que permite reducir considerablemente el tamaño de cada video para facilitar su procesamiento.

La etapa de inferencia debe ser capaz de detectar direcciones de RAM pertenecientes a cada frame que no codifican información del estado del juego. Éstas corresponden a direcciones sin usar, variables internas del motor no relacionadas con el estado del juego u otro tipo de valores que no competen a la inferencia de objetivos. De forma similar, es deseable encontrar relaciones entre variables del estado que codifiquen en conjunto un concepto particular. Resulta conveniente poder aprender la secuencia de teclas que permite iniciar una partida, dado que aprenderlo en la etapa de generación implicaría dedicar las generaciones iniciales del proceso de aprendizaje a deducir dicha secuencia. Al inferir la secuencia de teclas, es posible dedicar el proceso completo de aprendizaje en mejorar la habilidad del jugador generado para el juego en cuestión. Finalmente, para guiar la etapa de refinamiento de objetivos se debe inferir un conjunto de pesos candidatos, asignando una importancia relativa a cada dirección de RAM según los objetivos inferidos en esta etapa. Los problemas que componen a esta etapa son detallados en las subsecciones siguientes.

2.3.1. Inferencia de prefijos

Un prefijo es una secuencia de teclas común a todos los videos que permite iniciar una partida. En el proceso de inferencia se toma el prefijo más largo posible dado que el conjunto de teclas presionado en todos los videos es el mismo hasta comenzar la partida (en caso de haber menús, como sucede en el juego Tetris, pueden grabarse videos que naveguen los menús de la misma forma). Una vez dentro de la partida, el estilo personal de cada jugador y los factores específicos del juego provocan que se presionen teclas diferentes en cada video, permitiendo diferenciar las entradas usadas para jugar la partida de las empleadas para iniciarla.

2.3.2. Enmascarado de valores de RAM

Para simplificar el problema de inferencia y así mejorar la capacidad de encontrar relaciones útiles, es necesario reducir el espacio de búsqueda. Para ello se aplica el concepto de *enmascarado de RAM*, que permite ignorar direcciones cuyos valores se consideren no relevantes para comprender el objetivo del juego. El enmascarado de RAM puede realizarse aplicando diversos criterios. Los criterios utilizados en este proyecto se formalizan a continuación.

Enmascarado de valores estáticos

Dado un video V de una partida debe detectarse el conjunto de direcciones de RAM no utilizados por el juego en cuestión. Estos valores son clasificados como *estáticos*. El byte j se considera estático si cumple la condición $Var(V^{(j)}) < 0{,}17$, donde Var es la varianza y $V^{(j)}$ es la columna j-ésima de la matriz del video (el valor de una variable para todos los frames que la conforman). El valor 0,17 fue hallado empíricamente utilizando la información semántica hallada de forma manual para el juego Pinball, verificando que los valores que fueran catalogados como estáticos correspondieran con direcciones de memoria sin usar en el juego. Esto indica que el criterio funciona correctamente para el juego estudiado. Además, se corroboró que el criterio clasificara correctamente la RAM estática del resto de los juegos analizados.

Enmascarado de valores erráticos

Considerando la sucesión de frames que conforman a una partida, es posible detectar direcciones cuyo valor evolucione de manera demasiado caótica como para contener información relevante. Variables con comportamiento *errático* suelen corresponderse con contadores internos y otros tipos de variables del motor del juego. Una posición de RAM j se clasifica como errática si verifica la condición $Var(V^{(j)}) > 12100$.

El valor límite 12100 fue hallado por medio del estudio específico del juego Pinball. Pruebas sobre el resto de los juegos considerados ratificaron la validez del criterio, al enmascarar valores de RAM que no fueron identificados como relevantes para identificar los objetivos del juego.

Enmascarado de valores con bajo dinamismo

Otro criterio de filtrado corresponde a descartar variables que tomen la mayor parte del tiempo un valor específico, bajo el supuesto de que direcciones de RAM relacionadas con el juego toman muchos valores diferentes a lo largo de una partida reflejando el progreso del jugador. Una dirección de RAM d es catalogada como de *bajo dinamismo* para un video V si cumple con la ecuación 2.4, cuyos componentes se detallan en la ecuación 2.3. cF es la cantidad de frames que componen el video siendo analizado. Al igual que en los casos anteriores, el valor 0,95 surge del análisis experimental del comportamiento del Pinball, y clasifica correctamente las direcciones de RAM del resto de los juegos analizados.

$$frecuencia[i] = \sum_{j=1}^{cF} I_i(V_{j,d}) \qquad I_v(x) = \begin{cases} 1 & \text{si } x = v \\ 0 & \text{sino} \end{cases} \tag{2.3}$$

$$\frac{\max\limits_{0 \leq i \leq 255} \quad frecuencia[i]}{cF} > 0{,}95 \tag{2.4}$$

Enmascarado de valores independientes

Basándose en la comparación de valores de memoria en frames pertenecientes a distintos videos de entrenamiento se definió un nuevo criterio. Si los valores presentados por una dirección específica son los mismos frame a frame para dos partidas diferentes, la evolución de la dirección es *independiente* de las acciones del jugador. Este criterio busca eliminar variables internas del motor que no dependen del accionar del jugador y por lo tanto no pueden ser modificadas por las acciones de la IA generada.

Dados dos videos $V1$ y $V2$, se considera a una dirección d *independiente* si cumple con la desigualdad 2.7, cuyos componentes se definen en las ecuaciones 2.5 y 2.6, donde $minF$ es la cantidad de frames que posee el más corto de los videos siendo estudiados.

$$equal_d = \max_{1 \leq i \leq minF} Eq_i(d) \qquad minF = \min\left(cantFilas(V1), cantFilas(V2)\right) \tag{2.5}$$

$$Eq_f(x) = \begin{cases} f & \text{si } V1_{i,x} = V2_{i,x}, i = 0..f \\ 0 & \text{sino} \end{cases} \tag{2.6}$$

$$equal_d < cantFrames \tag{2.7}$$

El criterio requiere para poder clasificar correctamente que los frames correspondientes al prefijo, que no son parte de la partida propiamente dicha, sean removidos del par de videos estudiados. De esta forma se logra que el estado de ambas partidas esté sincronizado, dado que comienzan al principio de cada video.

2.3.3. Inferencia de órdenes lexicográficos

Para inferir relaciones entre los valores de RAM es necesario modelar los tipos de interacciones buscados. Luego de investigar las relaciones más comunes, se decidió abordar el problema bajo el enfoque de detección de órdenes lexicográficos. Este tipo de órdenes son muy frecuentes en diferentes elementos del juego que suelen guiar el progreso del jugador, como es el puntaje o el número de nivel y mundo, por lo que son de especial interés para aprender de forma automática los objetivos del juego.

No es posible inferir este tipo de órdenes dado que no existe forma de diferenciar órdenes lexicográficos que tienen un comportamiento oscilante de un conjunto no relacionado de valores. Por lo tanto, para poder automatizar la búsqueda de órdenes lexicográficos se impone la restricción de que los órdenes deben ser crecientes o decrecientes en el conjunto de frames considerados. Con este criterio se procede a realizar la definición de un orden lexicográfico decreciente y de modo análogo se procede para los órdenes crecientes.

Un orden lexicográfico se define como un conjunto ordenado de direcciones distintas de memoria $D_i, i = 0..n$, de cifra más significativa a menos significativa, como se presenta en la ecuación 2.8.

$$\{D_i\}_{i=0..n}, i \neq j \Rightarrow D_i \neq D_j, \forall i, D_i \in [1, 2048] \tag{2.8}$$

Además, se define la relación $>_D$ para dos frames F_a y F_b, formulada en la ecuación 2.9, donde el frame F_a es mayor según el orden D al frame F_b.

$$\begin{aligned} F_a >_D F_b \iff \exists k \in [0, n] \quad & F_a[D_i] = F_b[D_i], i \in [0, \ldots, (k-1)], \\ & y \; F_a[D_k] > F_b[D_k] \end{aligned} \tag{2.9}$$

La relación $>_D$ puede ser extendida a un conjunto de frames $\{F_j\}_{j=0..m}$, requiriendo que $F_i >_D F_{i+1}, i = 0..m-1$. Un orden D que cumple esta noción para todos los frames de un video es un orden *válido* para dicho video, dado que no viola la definición del orden en ningún frame. En este caso se obtiene un orden lexicográfico descendiente válido D sobre el conjunto de frames F.

Sin embargo, la definición de $>_D$ permite expandir el orden D con direcciones cuyos valores se mantengan estáticos sobre el conjunto de frames F, lo que debe ser evitado ya que dichos valores no participan del orden lexicográfico (debido a que por definición son estáticos). Para resolver este problema se define un *orden lexicográfico ajustado* como aquel en que todos sus elementos participan del mismo por lo menos una vez $\forall k = 0..n, \exists i, F_i[D_k] > F_{i+1}[D_k] \; y \; \forall j < k, F_i[D_j] = F_{i+1}[D_j]$.

Además, el objetivo de la búsqueda es encontrar órdenes lexicográficos completos sin omitir ningún elemento. Por ello deben buscarse órdenes válidos *maximales*, aquellos para los cuales no existe ninguna extensión posible que genere un orden válido.

Por lo tanto, el problema se reduce a la búsqueda de órdenes lexicográficos descendentes o ascendentes que sean válidos, ajustados y maximales. Se impone una última restricción sobre los órdenes lexicográficos: una dirección de RAM D_i puede pertenecer a lo sumo a un único orden lexicográfico. Esta restricción surge de que los órdenes codifican objetivos basados en elementos de un juego, como el puntaje. De permitir que una dirección pertenezca a más de un orden lexicográfico, se pueden generar órdenes lexicográficos separados que compartan elementos del juego, estableciendo una relación entre ellos. Esto contrasta con la intención de codificar objetivos independientes como se espera a partir de la construcción de cada orden.

Inferencia de los pesos candidatos

Para poder valorar los distintos conceptos que guían el progreso del juego, como el puntaje, el número de nivel o la cantidad de vidas, se debe inferir un conjunto de pesos candidatos para la función de valoración F definida en la ecuación 2.2. Estos pesos son luego mejorados en la etapa de refinamiento.

Se espera que un conjunto de pesos que capturen correctamente los conceptos fundamentales del juego permita a la función objetivo valorar de forma positiva a los videos de entrenamiento clasificados como buenos y calificar negativamente a los malos. Además, se desea que en una partida buena la función objetivo crezca de forma progresiva a lo largo del tiempo (al ir alcanzando distintos objetivos del juego), mientras que una partida negativa tenga evaluaciones cada vez peores producto de los errores del jugador humano. Estos requerimientos pueden ser capturados al buscar grupos de direcciones de RAM que tengan comportamiento creciente o decreciente, respectivamente. Este concepto se ve reflejado en la búsqueda de órdenes lexicográficos explicada en el apartado anterior.

Para realizar la evaluación se debe estudiar la evolución de los distintos órdenes lexicográficos sobre los videos de entrenamiento. Originalmente se consideraron las direcciones de RAM que no pertenecen a ningún orden como órdenes de un único elemento a efectos de estimar su peso asociado. Se pudo observar que a muchas direcciones de este tipo se les asignaba pesos relevantes, pero no eran verdaderos objetivos del juego, provocando que la función de evaluación no reflejara los conceptos importantes del juego. Por esto se decidió asignar un peso nulo a todas las direcciones que no sean parte de un orden lexicográfico. Los órdenes son evaluados en cada frame y sus valores son promediados para obtener una función única que describe la evolución de cada orden para un conjunto de videos. De esta forma se genera una función para el conjunto de videos buenos y otra para el conjunto de videos malos. La formulación del problema de inferencia de pesos se detalla a continuación.

Teniendo en cuenta la definición de orden lexicográfico presentada en la ecuación 2.8, se desea calcular un conjunto de pesos $\{w_i\}_{i=0..n}$ con w_i el peso asociado a la dirección de RAM D_i. Se definen las ecuaciones 2.11, 2.12 y 2.10, donde GV y BV son el conjunto de los videos buenos y malos respectivamente.

$$evalFrame(F, D) = \sum_{i=0}^{n-1} \left(F_{D_i} \times \prod_{j=i+1}^{n} (maxV_{D_j} + 1) \right) + F_{D_n} \tag{2.10}$$

$$\{\overline{G}\}_{f=1,..,minG} \ , \quad \overline{G}_f = \frac{1}{\#GV} \sum_{v \in GV} evalFrame\left(v_{(f)}, \{D_i\}_{i=0..n}\right) \tag{2.11}$$

$$\{\overline{B}\}_{f=1,..,minB} \ , \quad \overline{B}_f = \frac{1}{\#BV} \sum_{v \in BV} evalFrame\left(v_{(f)}, \{D_i\}_{i=0..n}\right) \tag{2.12}$$

De manera análoga, $minG$ y $minB$ corresponden a la longitud (en frames) del video bueno más corto y del video malo más corto, respectivamente. El valor n indica la cantidad de elementos del orden lexicográfico y $maxV_i$ es el máximo valor alcanzable por el byte i en una partida del juego. El término $(maxV_{D_j}+1)$ intenta representar el concepto de bases numéricas para los órdenes lexicográficos. Dado que se ignora la base en que están expresados dichos elementos, se toma como aproximación el mayor valor visto para la dirección de RAM en los videos de entrenamiento. Por este motivo se suma una unidad más al valor máximo de cada dirección. Este sistema puede asignar una base diferente a cada dígito del orden. Aunque parezca contraintuitivo, esto es deseable, debido a que algunos órdenes lexicográficos presentes en los juegos poseen sistemas numéricos de raíz mixta. Por ejemplo, *Super Mario Brothers* presenta un orden lexicográfico compuesto por el mundo, el nivel, el número de pantalla y la posición x en cada pantalla. Ninguno de los cuatro componentes posee una base igual a la de otro componente.

Una vez obtenidas las secuencias $\{\overline{G}\}_{f=1,..,minG}$ y $\{\overline{B}\}_{f=1,..,minB}$ que representan la evaluación promedio del orden lexicográfico para los videos buenos y malos, respectivamente, se procede al cálculo del peso W asociado al orden, según la ecuación 2.14 y sus componentes presentados en la ecuación 2.13.

$$S_G = \sum_{i=1}^{minG} \overline{G}_i \qquad S_B = \sum_{i=1}^{minG} \overline{B}_i \tag{2.13}$$

$$W = \begin{cases} \frac{S_G - S_B}{S_B} = \frac{S_G}{S_B} - 1{,}0 & \text{si } S_G \geq S_B \\ \frac{S_G - S_B}{S_G} = 1{,}0 - \frac{S_B}{S_G} & \text{sino} \end{cases} \tag{2.14}$$

En caso de que el mínimo de los videos malos ($minB$) sea menor que el de los buenos ($minG$), la secuencia $\{\overline{B}_f\}_{f=1,..,minB}$ es extendida extrapolando los datos faltantes a partir de una aproximación por medio de mínimos cuadrados. Se realiza una aproximación lineal de la secuencia $\{\overline{G}_f\}$ y se calcula el valor correspondiente al frame $minB + 1$. El valor hallado es utilizado como sustituto para todos los frames faltantes del conjunto $\{\overline{B}_f\}_{f=1,..,minB}$. El uso de mínimos cuadrados permite evitar problemas generados por valores espurios al final del video, debido a que en general los valores de RAM de los juegos de NES presentan una alta variabilidad al terminar una partida. Esto sucede a causa de la reutilización de variables por el poco espacio disponible, como también por causa del ahorro de ciclos de CPU por medio de la supresión de funciones de limpieza de RAM. Por ejemplo, en el caso del *Super Mario Brothers*, al estar en la pantalla de fin de partida las vidas valen 255 (-1 vidas por haber perdido la partida) y este valor no es reiniciado hasta volver al menú principal. Tal variabilidad es minimizada al extrapolar de la forma presentada. En caso de que $minB > minG$ se procede de forma análoga con la secuencia $\{\overline{G}_f\}$.

Usando el peso W es posible calcular los pesos w_i específicos para cada dirección de RAM que participa del orden lexicográfico siguiendo la ecuación 2.15.

$$\begin{aligned} w_n &= W \\ w_i &= W * \prod_{j=i+1}^{n} (maxV_{D_j} + 1) \ , \quad i < n \end{aligned} \tag{2.15}$$

Una vez obtenidos los pesos de cada orden se forma una función objetivo con todos los pesos calculados. Dado que cada dirección de RAM pertenece a un único orden lexicográfico, cada w_i es único para una dirección i.

2.4. Refinamiento de objetivos

Con los pesos candidatos y el conjunto de valores enmascarados de RAM es posible iniciar una etapa de refinamiento que intente mejorar de forma progresiva los resultados de la etapa de inferencia, mejorando la función objetivo a usar en la etapa de generación.

Inferir los objetivos de un juego es una tarea compleja, dado que implica comprender la semántica del juego para poder deducir lo que se espera del jugador. Además, se requiere comprender la consecuencia de las acciones realizadas para relacionarlas con el feedback provisto por el mundo virtual. Ambas acciones implican un conocimiento previo del mundo real en el que la mayor parte de los juegos se basan implícitamente. A modo de ejemplo, cuando un jugador ve un personaje en la pantalla saltando, se espera que el personaje vuelva a caer a tierra, dado que se asume la existencia de gravedad. Si bien este es un concepto cotidiano e intuitivo para los seres humanos, una IA no conoce sobre la gravedad y no tiene argumentos para no considerar otros resultados, como por ejemplo la desintegración del personaje en pleno salto.

La falta de conocimiento previo es un gran problema para las IAs. Sin embargo, las estrategias presentadas en la inferencia de objetivos inducen cierto tipo de conocimiento en forma de distintos sesgos o preferencias por un tipo de soluciones sobre otras. En las técnicas de inferencia presentadas en la sección 2.3.3, el uso de órdenes lexicográficos y distintos tipos de enmascarados de RAM inducen una preferencia por tomar como variables de decisión direcciones con un comportamiento variable moderado que posiblemente sea creciente o decreciente (respecto al orden), bajo el supuesto de que codifican un objetivo del juego.

La etapa de refinamiento intenta mejorar los resultados obtenidos siguiendo los mismos criterios presentados para la inferencia, pero agregando una nueva suposición sobre la función objetivo a construir. Se impone una penalidad en caso de que la función objetivo no sea monótona, según el criterio de que si una partida que es marcadamente buena o mala no debe mostrar oscilaciones bruscas de sus objetivos. Se busca además maximizar la diferencia entre la evaluación de los videos buenos y malos, dado que se espera que la función objetivo muestre un cambio significativo entre ambos tipos.

Formalmente, dado un conjunto $\{GV_i\}_{i=0..cantG}$ de videos buenos, se define $\widehat{G}$ como la evaluación de la función objetivo F sobre el promedio de videos buenos según la ecuación 2.16, donde $minG$ es el largo del video bueno más corto. Análogamente se nombra $minB$ al largo del video malo más corto.

$$
\begin{aligned}
\widehat{G} &= \sum_{j=1}^{minG} \frac{\left(\sum_{v \in GV} eval(v_{(j)})\right)}{\#GV} \\
&= \sum_{j=1}^{minG} \frac{\left(\sum_{v \in GV} \left(\sum_{i=1}^{2048} v_{j,i} \times w_i\right)\right)}{\#GV} \\
&= \sum_{j=1}^{minG} \frac{\left(\sum_{i=1}^{2048} \left(\sum_{v \in GV} v_{j,i} \times w_i\right)\right)}{\#GV} \\
&= \sum_{j=1}^{minG} \frac{\left(\sum_{i=1}^{2048} w_i \times \left(\sum_{v \in GV} v_{j,i}\right)\right)}{\#GV} \\
&= \sum_{j=1}^{minG} \sum_{i=1}^{2048} w_i \times \frac{\sum_{v \in GV} v_{j,i}}{\#GV}
\end{aligned}
\tag{2.16}
$$

Se denota al conjunto de los reales positivos y el 0 como $\mathbb{R}^* = \mathbb{R}^+ \cup \{0\}$ y se define el *video promedio bueno* $\overline{GV} \in \mathcal{M}_{minG \times 2048}(\mathbb{R}^*)$ como $\overline{GV}_{f,b} = \frac{1}{\#GV} \sum_{v \in GV} v_{f,b}$. Es posible entonces reescribir a $\widehat{G}$ en función de $\overline{GV}$ como puede verse en la ecuación 2.17.

$$
\widehat{G} = \sum_{j=1}^{minG} \sum_{i=1}^{2048} w_i \times \overline{GV}_{j,i} = F(\overline{GV}) = \sum_{j=1}^{minG} eval(\overline{GV}_{(j)}) \tag{2.17}
$$

Análogamente, se define el *video promedio malo* como $\overline{BV} \in \mathcal{M}_{minG \times 2048}(\mathbb{R}^*)$, $\overline{BV}_{f,b} = \frac{1}{\#BV} \sum_{v \in BV} v_{f,b}$, y $\widehat{B}$ como: $\widehat{B} = F(\overline{BV}) = \sum_{j=1}^{minG} eval(\overline{BV}_{(j)})$.

Al igual que en la inferencia de pesos de la etapa anterior, si $minB < minG$ el resultado de $eval(\overline{BV}_{(j)})$ para $j > minB$ es tomado como el valor de la aproximación por mínimos cuadrados en el frame $minB + 1$.

La expresión $\Delta = \widehat{G} - \widehat{B}$ permite evaluar la diferencia entre la calificación otorgada a los videos buenos y malos. El objetivo principal es maximizar Δ, con el criterio de que la función objetivo debe poder discernir claramente entre ambos tipos de videos.

Además, es razonable esperar que una función objetivo que describa correctamente los objetivos tenga un comportamiento creciente respecto a los videos buenos, incrementando el valor que le asigna al estado del juego conforme el jugador progresa en la partida. Para evaluar cuánto dista la función objetivo de ser monótona, se define el factor de monotonía $M = \frac{U+E/4}{minG-1}$, siendo $U = \#\{i \in [2, \ldots, minG] : eval(\overline{GV}_{(i)}) > eval(\overline{GV}_{(i-1)})\}$ la cantidad de frames en las que la evaluación del video promedio bueno aumentó su valor. Por su parte, E es la cantidad de frames en las que dicha evaluación se mantuvo constante.

En el refinamiento de objetivos se busca penalizar las funciones que exhiban un comportamiento oscilante, dado que en los videos buenos se constata un progreso continuo del jugador que debe corresponderse con un crecimiento monótono o casi monótono (y por ello se clasifican como videos buenos). Algunos ejemplos de este comportamiento son los órdenes lexicográficos compuestos por el número de nivel, el número de pantalla y la posición x en cada pantalla del juego *Contra*; el mundo, el nivel, el número de pantalla y la posición x en *Super Mario Brothers*; la diferencia de goles en el *Ice Hockey* o el puntaje en el *Pinball*, entre otros. En todos estos casos, una caída importante del valor de los órdenes lexicográficos mencionados indica una regresión en el objetivo del juego, ya sea por haber perdido una vida, o haber recibido goles en contra. Cuando la función objetivo exhibe monotonía se espera que los objetivos del juego estén siendo alcanzados sin contratiempos, mostrando un mejor desempeño en la partida.

Para calcular el factor de monotonía se utiliza solamente el promedio de los videos buenos ($\overline{GV}$), dado que estos representan el promedio de buenas partidas y se esperan avancen en sus objetivos sin contratiempos. No se exige monotonía sobre la función objetivo al evaluar los videos malos debido a que es posible que la función objetivo oscile según la naturaleza del juego. En particular, en el juego *Pinball* es necesario anotar algunos puntos antes de perder una bola, por lo que la evaluación de videos malos crece momentáneamente para luego decrecer y vuelve a crecer temporalmente al sacar la segunda bola. En definitiva, la función objetivo muestra un comportamiento oscilatorio con una tendencia decreciente hasta perder.

La función objetivo de la etapa de refinamiento (no debe confundirse con la función objetivo inferida para el juego, F) es $F_R = \Delta * M$. La etapa de refinamiento intenta maximizar el valor de F_R ajustando los pesos de la función F.

2.5. Generación de inteligencias artificiales

Una vez obtenida una versión refinada de los pesos de la función objetivo, la última etapa es la generación de IAs propiamente dicha. Con este fin, se entrena a un jugador artificial utilizando la función objetivo F basada en los pesos aprendidos como una guía que permite calificar su desempeño en base al progreso que logra el jugador en una partida.

Para entrenar al jugador artificial se propone el uso de neuroevolución. Basándose en lo expuesto en este capítulo, la función objetivo para el proceso evolutivo es la función F. El objetivo de esta etapa es obtener un jugador que maximice el valor de F para un tiempo de entrenamiento (cantidad de frames) dado.

La IA resultante debe ser capaz de jugar correctamente al juego en el que fue entrenada. La generación exitosa de esta inteligencia de forma automatizada (utilizando como entrada solamente los videos de entrenamiento) es el objetivo del presente proyecto. Además, emplear las técnicas de aprendizaje propuestas permite hallar soluciones simples a los juegos estudiados que no fueron contempladas por el desarrollador del juego, cumpliendo un rol de verificación del juego en cuestión. Los fundamentos y las técnicas empleadas para la neuroevolución son presentadas en el capítulo siguiente.

Capítulo 3

Técnicas evolutivas, redes neuronales y neuroevolución

El presente capítulo detalla las principales técnicas empleadas en el desarrollo de este proyecto, comprendiendo el uso de algoritmos evolutivos, redes neuronales artificiales y técnicas de neuroevolución. El capítulo comienza con una descripción de los algoritmos evolutivos y sus principales conceptos. A continuación, se presentan los conceptos sobre técnicas evolutivas paralelas. Luego se detallan los conceptos básicos referentes a las redes neuronales artificiales. El capítulo termina con una descripción de la técnica de neuroevolución NEAT (NeuroEvolution of Augmented Topologies).

3.1. Algoritmos Evolutivos

Los algoritmos evolutivos son una familia de algoritmos de búsqueda estocásticos e iterativos basados en los conceptos de la teoría de evolución natural. A diferencia de una búsqueda aleatoria, los algoritmos evolutivos explotan información contenida en la población para guiar la búsqueda de forma eficiente (Goldberg, 1989). Tomando como base esquema definido por algoritmos evolutivos es posible definir algoritmos específicos que permiten atacar un problema particular. La definición general de los algoritmos evolutivos permite utilizarlos para solucionar un conjunto amplio de problemas utilizando las mismas ideas básicas.

3.1.1. Elementos básicos de un algoritmo evolutivo

El esquema general de un algoritmo evolutivo sigue el principio de selección natural para realizar una búsqueda del espacio de soluciones de un problema particular. Para ello se utiliza el concepto de *población*: un conjunto de representaciones de las soluciones candidatas mantenidas por el algoritmo. Cada representación recibe el nombre de *individuo*. El esquema general de un algoritmo evolutivo es presentado en el algoritmo 1. Cada componente es detallado a continuación.

Algoritmo 1 Esquema de un algoritmo evolutivo

generación ← 0
P[*generación*] ← *inicializarPoblación()*
mientras (no se cumpla el criterio de parada) **hacer**
  evaluar(*P*[*generación*])
  Padres ← *seleccionar*(*P*[*generación*])
  Hijos ← *aplicarOperadoresEvolutivos*(*Padres*)
  P[*generación*+1] ← *reemplazar*(*Hijos*, *P*[*generación*])
  generación ← *generación* +1
fin mientras
retornar mejor solución encontrada

Cada generación es una iteración del proceso evolutivo. Como se aprecia en el algoritmo 1, los individuos de la población evolucionan con cada generación.

3.1.2. Individuo

Un individuo es una representación de una solución. El individuo puede asociarse con el concepto de genotipo en biología, mientras que la solución que representa es el fenotipo. Al definir una representación particular debe considerarse si ésta permite codificar soluciones inválidas, y en caso afirmativo, cómo serán manejadas por el algoritmo (Whitley, 1994). Se define el además el concepto de gen como la mínima unidad de información de la representación elegida.

3.1.3. Función de fitness

La *función de fitness* es la encargada de evaluar la aptitud de cada solución considerada y debe incorporar conocimiento específico del problema relacionado con la función a optimizar para poder evaluar el desempeño de cada solución candidata. El objetivo del algoritmo es maximizar el valor obtenido por la función de fitness, que permite diferenciar la calidad de las distintas soluciones.

3.1.4. Inicialización

Habitualmente la población se inicializa mediante un mecanismo aleatorio que selecciona soluciones de entre todas las posibles que conforman el espacio de búsqueda. También puede guiarse el proceso de selección utilizando información del problema a resolver. Además, es común agregar a la población un conjunto de soluciones preexistentes generadas con algún algoritmo heurístico que permita generar buenas soluciones para el problema.

3.1.5. Selección

La *selección* es el proceso responsable de seleccionar un subconjunto de individuos de la población que serán modificados por los operadores evolutivos. Existen múltiples criterios de selección que intentan preservar las buenas cualidades de las soluciones basándose

en su fitness. La mayoría de los criterios incorporan un elemento estocástico, con el objetivo de simular la posibilidad de seleccionar individuos de menor aptitud, como sucede en el proceso de selección natural.

Ejemplos de operadores de selección frecuentemente usados son la selección proporcional, la selección estocástica universal, la selección por torneo y la selección por rango, entre otros. Cada ejemplo es detallado a continuación.

Selección proporcional: este operador otorga a cada individuo una probabilidad de ser seleccionado proporcional a su fitness. Sea N el número de individuos considerados y f_i el valor de fitness del i-ésimo individuo, la probabilidad p_i de seleccionar al individuo i está dada por la ecuación 3.1.

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{3.1}$$

Puede considerarse a este operador como una ruleta con un puntero donde a cada individuo se le asigna un área de la ruleta proporcional a su fitness. Para realizar una selección se hace girar la ruleta y se selecciona al individuo correspondiente con el área a la cual apunta el puntero. Este operador tiene un sesgo hacia la selección de individuos de mayor fitness, debido a que ellos ocupan mayor área de la ruleta.

Selección estocástica universal: funciona de manera similar a la selección proporcional, pero en vez de utilizar un único puntero se utilizan N punteros equiespaciados, siendo N el número de individuos a seleccionar. Esta técnica reduce el sesgo que tiene la selección proporcional hacia elegir con más frecuencia los individuos con mayor fitness y de esta forma mantener la diversidad de la población (Baker, 1987).

Selección por torneo: divide a la población de forma aleatoria en grupos de un tamaño predeterminado y selecciona un número específico de individuos de cada grupo que poseen el fitness más alto del mismo.

Los tres ejemplos anteriores muestran el uso de elementos estocásticos en el operador de selección, para permitir la selección de individuos con menor fitness.

Selección por rango: ordena la población según su fitness y elige una cantidad predeterminada de individuos con los mejores valores de fitness de la población.

3.1.6. Operadores evolutivos

Existen múltiples operadores evolutivos que son aplicados para generar nuevos individuos a partir de los individuos seleccionados. Los operadores más frecuentes de un algoritmo evolutivo son los operadores de cruzamiento y mutación, que se describen a continuación.

Operador de cruzamiento: combina dos o más soluciones, denominadas *padres*, para obtener una o más soluciones nuevas según un mecanismo predefinido. Este operador es aplicado de forma probabilística de acuerdo una probabilidad de aplicación p_c. Al momento de aplicar el operador se realiza un sorteo de un número aleatorio en el intervalo [0,1] con probabilidad uniforme. Si el resultado es menor a la probabilidad de aplicación entonces se aplica el operador. En caso contrario el operador no se aplica y se retornan los padres sin cambios.

Algunos operadores de cruzamiento usados frecuentemente son el cruzamiento de un punto y el de n puntos. El cruzamiento de un punto sortea con distribución uniforme un punto de corte en el intervalo $[0, L-1]$, siendo L el largo del individuo, e intercambia

uno de los fragmentos obtenidos con el fragmento correspondiente del otro individuo que participa en el cruzamiento. De esta forma se obtienen un nuevo par de individuos a partir de dos padres. El cruzamiento de n puntos selecciona n puntos de corte en la representación, intercambiando la información contenida en las secciones impares con la información correspondiente del otro individuo.

Operador de mutación: modifica genes de la solución de forma aleatoria, permitiendo al algoritmo evolutivo considerar soluciones que no pueden ser producidas por el cruzamiento de individuos de la población. Este operador suele aplicarse de forma probabilística de acuerdo a una probabilidad de aplicación p_m. En general el valor de p_m es bajo (menor a 0.1), por lo que es poco probable que un gen particular sea mutado. La definición del operador de mutación depende fuertemente de la representación elegida para los individuos, debido a que trabaja con la semántica definida por dicha representación. A modo de ejemplo, si la representación es un arreglo de enteros entre 0 y 100, un operador de mutación posible consiste en el reemplazo de cada entero a mutar por otro seleccionado uniformemente en el intervalo $[0, 100]$.

3.1.7. Criterio de parada

El *criterio de parada* describe una condición que de cumplirse indica la finalización del proceso evolutivo, que retornará el mejor individuo de la población. Algunos ejemplos de criterios de parada frecuentemente utilizados son imponer un límite de generaciones a la evolución (esfuerzo computacional fijo) o la aplicación de chequeos de estancamiento (por ejemplo, detectar que la población no ha mejorado su fitness durante una cantidad dada de generaciones). Es también posible utilizar múltiples criterios de parada, terminando el proceso evolutivo cuando alguno se cumple.

3.1.8. Uso de elitismo

En el esquema de un algoritmo evolutivo presentado en el algoritmo 1 no se asegura la supervivencia del mejor individuo a lo largo de las generaciones. Para garantizar la supervivencia del mejor individuo se introduce el concepto de *elitismo*. La preservación del mejor individuo se logra utilizando una técnica de reemplazo que asegure la supervivencia de los mejores individuos (los individuos con mayor fitness). La forma más común de implementar este concepto es ordenar la población en orden descendente de fitness y tomar una cantidad específica de los mejores individuos, los cuales son asignados a la próxima generación sin aplicarles ningún operador evolutivo.

3.2. Algoritmos evolutivos paralelos

El costo computacional de un algoritmo evolutivo convencional puede ser elevado, dado que la función de evaluación puede requerir ejecutar procesos costos como la simulación de un sistema. Por este motivo la etapa de evaluación es frecuentemente la más cara del algoritmo evolutivo, dado que el costo computacional de la aplicación de los operadores evolutivos y el manejo de la población es bajo (Alba y Tomassini, 2002). Para mejorar la eficiencia es posible ejecutar simultáneamente distintos procesos del algoritmo (estrategia denominada *paralelización*). De esta forma se pueden utilizar múltiples recursos de cómputo (procesadores y núcleos) presentes en un sistema computacional

para ejecutar diferentes etapas del algoritmo evolutivo, lo que permite completar la ejecución en una menor cantidad de tiempo. Existen distintas estrategias de paralelización, siendo las más utilizadas la evaluación paralela de soluciones y el uso de subpoblaciones distribuidas, cuyos detalles se presentan en las subsecciones siguientes.

3.2.1. Evaluación paralela de soluciones

Teniendo en cuenta que la evaluación de cada individuo es independiente del resto, es posible realizar las evaluaciones en paralelo para mejorar el desempeño del algoritmo. Una estrategia de paralelización frecuentemente empleada es el modelo *maestro-esclavo*, en el cual un proceso maestro ejecuta el algoritmo evolutivo y delega la evaluación de los individuos a un conjunto de procesos esclavos (Luque y Alba, 2011). Cada proceso esclavo computa el valor de fitness para un subconjunto de individuos y retorna los resultados al proceso maestro para que pueda continuar la ejecución del algoritmo evolutivo (realizar la selección, cruzamiento, mutación). Un diagrama del modelo de evaluación paralela de soluciones se presenta en la figura 3.1. Utilizando este modelo puede distribuirse la tarea de evaluación sobre todos los procesadores disponibles en la plataforma donde se ejecuta el algoritmo, empleándolos de manera más eficiente y reduciendo el tiempo total de ejecución.

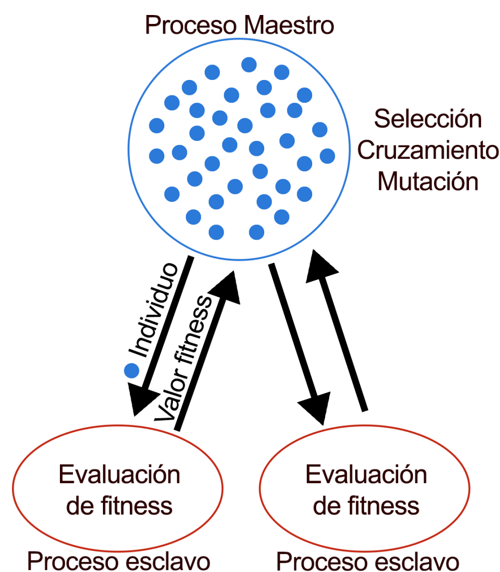


Figura 3.1: Modelo maestro-esclavo

3.2.2. Subpoblaciones distribuidas

Otro modelo de paralelización consiste en ejecutar varios procesos evolutivos simultáneamente. Estos procesos reciben el nombre de *subpoblaciones* o *islas*. Cada subpoblación es un algoritmo evolutivo que evoluciona de manera separada, compartiendo información con una frecuencia fija por medio de un nuevo operador evolutivo denominado *migración* (Luque y Alba, 2011). Este operador transfiere una cantidad específica de individuos de una población a otra según un operador de selección de los individuos a migrar, independiente del operador de selección utilizado por el algoritmo evolutivo para

la aplicación de los operadores tradicionales. El flujo de individuos entre poblaciones se define en base a una topología específica de migración. Una topología comúnmente usada es la de *anillo unidireccional*, representada en la figura 3.2.

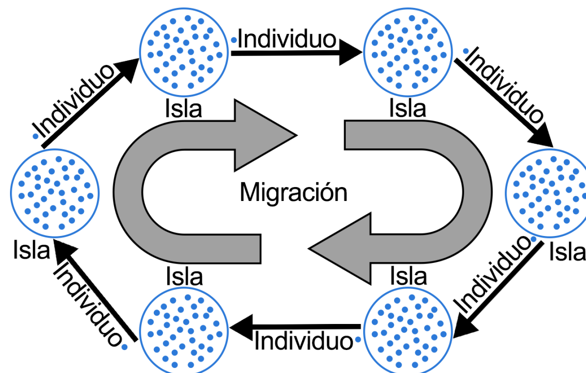


Figura 3.2: Modelo de subpoblaciones distribuidas

Además de la paralelización del proceso evolutivo, este modelo fomenta una mayor diversidad de los individuos. La diversidad refiere a qué tan diferentes son los individuos de la población. Una población compuesta por individuos muy similares presenta poca diversidad y puede llevar al estancamiento de la búsqueda realizada por el algoritmo evolutivo. El incremento de diversidad provisto por el uso de subpoblaciones es consecuencia de que cada subpoblación evoluciona de manera separada a las demás. Esto posibilita que cada subpoblación explore un subconjunto diferente del espacio de soluciones del problema. Mantener una mayor diversidad suele implicar una mejor calidad de las soluciones obtenidas, al disminuir el riesgo de que la búsqueda se estanque en óptimos locales.

3.3. Redes neuronales artificiales

Las redes neuronales artificiales (ANN por sus siglas en inglés) son técnicas de inteligencia computacional que imitan características exhibidas por las redes neuronales naturales existentes en los seres vivos. Las ANN permiten aproximar funciones complejas con un costo computacional de evaluación razonable (Mitchell, 1997). En las secciones siguientes se definen los conceptos básicos de las redes neuronales y variantes avanzadas de las mismas.

3.3.1. Definición de una red neuronal artificial

Una ANN puede definirse como un grafo dirigido cuyas aristas están ponderadas por pesos. Los nodos del grafo (también llamados neuronas) son agrupados en capas, donde todas las salidas de una capa se conectan a todos los nodos de la capa siguiente. Redes de este tipo reciben el nombre de *feed-forward networks*. Un ejemplo de ANN feed-forward se presenta en la figura 3.3. Como mínimo debe existir una capa de entrada, encargada de recibir las entradas a la red, y una capa de salida, responsable de generar las salidas de la red a partir de los resultados de las capas anteriores. En general las redes tienen una o más capas ocultas entre la capa de entrada y la de salida. Cada capa oculta genera una interpretación más abstracta de los datos de entrada que la capa anterior.

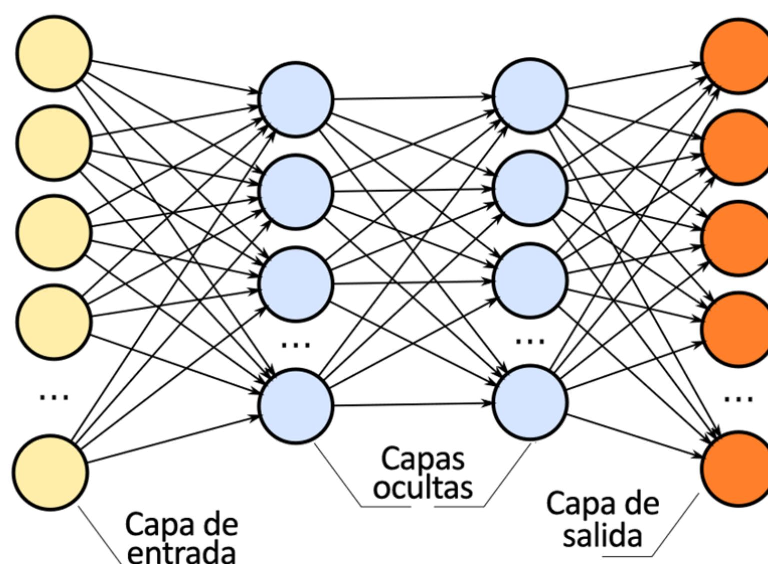


Figura 3.3: Ejemplo de ANN feed-forward

Existen varios tipos de nodos que componen una ANN. Uno de los nodos más simples es el perceptrón, cuya forma genérica puede verse en la figura 3.4. Cada perceptrón recibe las salidas de todos los nodos que conforman la capa anterior, luego de multiplicarlos por el peso w_i asociado a la arista i que lo conecta. El perceptrón suma todos estos valores y el resultado es enviado como entrada a una función de activación (también llamada función de *squashing*). Esta función decide si el perceptrón deberá emitir una salida o no, según si el valor de la suma excede cierto valor límite. Además, la función de activación limita el valor de la salida para evitar resultados demasiado grandes al sumar una cantidad alta de valores de los nodos anteriores. De no limitarse la salida puede ocasionarse un overflow en la sumatoria de un nodo posterior, lo que a su vez afectaría el resultado de la capa siguiente al continuar la propagación a nodos posteriores, alterando drásticamente los resultados de la red. El poder de expresividad de un perceptrón es equivalente al de un hiperplano de decisión en el espacio de entrada de la red en cuestión. Esto quiere decir que el perceptrón es capaz de clasificar correctamente elementos de dicho espacio que sean linealmente separables (Mitchell, 1997).

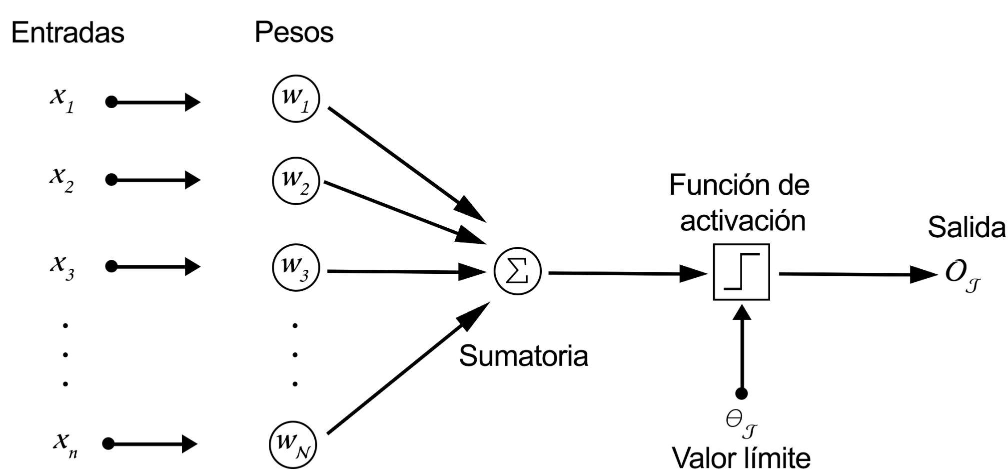


Figura 3.4: Esquema de un perceptrón

Otros tipos de nodos más complejos reemplazan la función de activación utilizada por el perceptrón (que solo devuelve 0 o 1) por funciones más complejas (sigmoide, tangente hiperbólica, softsign, etc.), logrando mejoras en la expresividad del nodo (Karlik y Olgac, 2011). Estos nodos con mayor expresividad son los más frecuentemente utilizados para crear redes neuronales.

3.3.2. Entrenamiento

El objetivo del entrenamiento es aprender una función particular que se desea aproximar con una red neuronal. El entrenamiento consiste en ajustar los pesos de la red a valores apropiados que permitan aproximar las salidas de la red a las producidas por la función que se quiere aprender. Hallar dichos valores no es una tarea sencilla. Existen varios métodos de entrenamiento diferentes. El más común es *backpropagation*, que se basa en el uso de casos de entrenamiento. Cada caso consiste es un par compuesto por la entrada que se le provee a la red y la salida esperada (valor generado por la función que se desea aprender). Dada una red definida previamente, se evalúa la entrada de cada caso de entrenamiento y se calcula el error cometido como la diferencia entre la salida obtenida y la esperada. Los pesos de la red son ajustados para minimizar el error cometido en los casos de entrenamiento. Esta técnica garantiza solamente alcanzar una aproximación a la función que constituye un mínimo local respecto al error entre la salida de la red neuronal y la salida de la función que se intenta aprender. Sin embargo, backpropagation ha presentado muy buenos resultados al entrenar redes neuronales utilizadas en varias aplicaciones del mundo real (Mitchell, 1997).

Otra técnica para el entrenamiento de redes neuronales consiste en aplicar algoritmos evolutivos para definir los pesos de la red. Como se detalla en la sección 3.3.4, el uso de algoritmos evolutivos resulta en algoritmos más simples que al utilizar backpropagation en casos de redes más complejas. Además, a diferencia de backpropagation que trabaja sobre una red cuya estructura (los nodos y sus conexiones) es fija, los algoritmos evolutivos permiten mejorar la estructura de la red durante el proceso. Otra diferencia entre backpropagation y el uso de algoritmos evolutivos es que los últimos no requieren casos de entrenamiento. Los algoritmos evolutivos utilizan en cambio una función objetivo para poder evaluar el desempeño de la red. Para el problema abordado en este proyecto se optó por el uso de algoritmos evolutivos para el entrenamiento de las redes neuronales, debido a que backpropagation no es aplicable porque no es posible definir casos de entrenamiento concretos. El impedimento radica en el componente temporal presente en las entradas de la red neuronal, que es el estado del juego. Si bien es posible considerar a cada partida en conjunto con todas las teclas presionadas por el jugador como un caso de entrenamiento, generar la cantidad de partidas necesarias para tener un conjunto de entrenamiento suficientemente amplio para utilizar por backpropagation se vuelve prohibitivo debido a que todas deben ser generadas por jugadores humanos. Incluso si se ignora el problema de generar las partidas necesarias, el uso de partidas existentes como ejemplos de entrenamiento para backpropagation generaría redes que imitan los patrones de juego exhibidos en dichas partidas, en vez de explorar otras estrategias que puedan brindar buenos resultados. Esta imitación va en contra del objetivo de verificación que se propone en el proyecto por medio de la detección de errores desconocidos en el juego que pueden ser explotados por los jugadores artificiales generados.

3.3.3. Evaluación

Para computar la función aprendida por la red se deben asignar los argumentos de la función a las entradas de la red y propagarlos a través de las neuronas ocultas hasta alcanzar las salidas de la red, obteniendo el resultado de la función. En las redes feed-forward, como es el caso de la figura 3.3, la propagación de los valores se realiza capa a capa. En cada nodo se calcula la sumatoria de las entradas del nodo y el resultado se envía a la función de activación. Redes más complejas requieren considerar aspectos que dependen de la topología del grafo que representa la red, como es el caso de las redes neuronales recurrentes, que se describen a continuación.

3.3.4. Redes Neuronales Recurrentes

Las redes neuronales recurrentes (RNN por sus siglas en inglés) son redes que soportan retroalimentación. En ellas la salida de un nodo puede dirigirse a un nodo anterior, por lo cual el grafo correspondiente presenta ciclos. Esta estructura permite a la red incorporar memoria, dado que las salidas de los nodos en un instante de tiempo son las entradas a otros nodos (o a ellos mismos) en el instante siguiente.

El uso de redes recurrentes permite computar funciones complejas sobre entradas secuenciales, entendiéndose por entradas secuenciales a aquellas entradas que poseen un componente temporal. Sin embargo, la mayor complejidad asociada a las redes recurrentes introduce un conjunto de desafíos que dificultan los procesos de entrenamiento y evaluación. En ambos casos, el problema surge de la recursividad implícita que poseen las redes recurrentes al soportar enlaces de retroalimentación. Evaluar una RNN requiere resolver dicha recursividad. Una solución posible es realizar una cantidad suficiente de pasos de propagación que permita asegurar que todas las salidas puedan ser calculadas por lo menos una vez. La cantidad de pasos necesarios para asegurar el cálculo de las salidas es igual a la cantidad de aristas existentes en el camino más largo dentro del conjunto de caminos más cortos entre las neuronas de entrada y cada neurona de salida. Formalmente, sea $d_{min}(i, j)$ la distancia mínima entre la neurona de entrada i y la neurona de salida j, se deben ejecutar al menos $\max_j \min_i d_{i,j}$ pasos para asegurar que se han calculado los valores de todas las neuronas de salida. La técnica de backpropagation presentada en la sección 3.3.2 requiere de modificaciones sustanciales para poder funcionar con redes recurrentes. Aún con dichas modificaciones, la naturaleza recurrente de estas redes implica ciertos problemas que dificultan el ajuste de sus pesos (Sutskever, 2013). Este problema puede tratarse de forma más simple al emplear algoritmos evolutivos que, a diferencia de la variante de backpropagation empleada para entrenar estas redes, no necesitan tratar de forma especial los enlaces recurrentes.

3.4. Neuroevolución

Como alternativa a los métodos de entrenamiento tradicionales basados en el procesamiento de casos de entrenamiento, se han aplicado algoritmos evolutivos para definir los pesos óptimos de una red neuronal (Simpson, 2012) (Jørgensen y Sandberg, 2009) (Hausknecht et al., 2014). Estas estrategias reciben el nombre de *neuroevolución*. Esta sección presenta las técnicas neuroevolutivas y las contrasta con el algoritmo de backpropagation, para luego introducir una técnica neuroevolutiva particular, denominada NEAT.

3.4.1. Conceptos generales

El uso de técnicas neuroevolutivas no requiere de un conjunto de casos de entrenamiento, a diferencia del algoritmo de backpropagation presentado en la sección 3.3.2. Sin embargo, los algoritmos neuroevolutivos presentan otras dificultades, como la definición de operadores evolutivos que permitan generar nuevos individuos que preserven las buenas cualidades encontradas hasta el momento. El mayor problema lo presenta el operador de cruzamiento, ya que la combinación de dos redes neuronales para generar una nueva red podría perder información estructural importante, como se explica en la sección 3.4.2. Para evitar dicha pérdida de información se deben tener en cuenta varios aspectos específicos de la estructura de las redes para poder entrenarlas de forma exitosa por medio de métodos evolutivos.

Existen numerosos artículos sobre neuroevolución. Varios autores han planteado que no es posible construir un buen operador de cruzamiento (Yao y Liu, 1998) (Angeline et al., 1994), por lo que se han enfocado en desarrollar algoritmos que no utilicen cruzamiento o recombinación, empleando solamente de la mutación. Un ejemplo de neuroevolución que no utiliza un operador de cruzamiento es la técnica *GeNeralized Acquisition of Recurrent Links* (GNARL) (Angeline et al., 1994). Otros autores, en cambio, han logrado obtener soluciones de buena calidad por medio de algoritmos de cruzamiento específicos que tienen en cuenta información estructural de la red y el histórico de cambios realizados sobre su estructura, como la técnica NEAT, que se describe a continuación.

3.4.2. NEAT

NEAT es una técnica presentada por Stanley y Miikkulainen (2002) que permite generar redes neuronales por medio de algoritmos evolutivos, evolucionando los pesos de las conexiones y la estructura de la red.

El objetivo principal de NEAT es resolver los problemas encontrados al intentar evolucionar redes neuronales con algoritmos evolutivos. Aplicar el operador de mutación o cruzamiento sin consideración por la estructura de la red provoca la pérdida del conocimiento aprendido por las redes, debido a que el conocimiento se encuentra codificado en las relaciones entre todos los componentes de la red. NEAT soluciona este problema proponiendo utilizar un operador de cruzamiento y un operador de mutación especializados que permiten preservar el conocimiento aprendido por las redes y cambiar su topología sin alterar significativamente el conocimiento aprendido por la red. El esquema básico de NEAT se presenta en el Algoritmo 2. Los conceptos principales del esquema son desarrollados en las subsecciones siguientes.

Algoritmo 2 Esquema de un algoritmo NEAT

$generación \leftarrow 0$
$P[generación] \leftarrow$ *inicializarPoblación()*
mientras (no se cumpla el criterio de parada) **hacer**
  $evaluar(P[generación])$
  $P[HijosEspecies] \leftarrow \emptyset$
  $Especies \leftarrow especiar(P[generación])$
  para cada $Especie \in Especies$ **hacer**
    $Padres \leftarrow seleccionar(Especie)$
    $Hijos \leftarrow aplicarCruzamiento(Padres)$
    $Hijos \leftarrow aplicarMutación(Hijos)$
    $HijosEspecies \leftarrow HijosEspecies \cup Hijos$
  fin para
  si faltan individuos para completar generación **entonces**
    $IndividuosExtra \leftarrow seleccionar(P[generación])$
    $HijosIndividuosExtra \leftarrow aplicarCruzamiento(IndividuosExtra)$
    $HijosIndividuosExtra \leftarrow aplicarMutación(HijosIndividuosExtra)$
    $P[generación{+}1] \leftarrow reemplazar(HijosEspecies \cup HijosIndividuosExtra, P[generación])$
  si no
    $P[generación{+}1] \leftarrow reemplazar(HijosEspecies, P[generación])$
  fin si
  $generación \leftarrow generación{+}1$
fin mientras
retornar mejor solución encontrada

Codificación de redes neuronales

Las redes neuronales son codificadas por medio de dos tipos de genes, nodos y enlaces, que se mantienen en listas separadas. Cada nodo se encuentra etiquetado con su tipo (de entrada, oculto o de salida). Cada enlace indica qué nodos conecta y tiene asociado un peso, un indicador de si ha sido deshabilitado y un número de innovación. Las redes modeladas no tienen restricciones en su topología, por lo que pueden tener enlaces recurrentes. Un ejemplo de codificación se presenta en la figura 3.5, donde se muestran las listas de nodos y de enlaces mantenidas por el individuo. La red del ejemplo de la figura 3.5 posee 5 nodos y 8 enlaces.

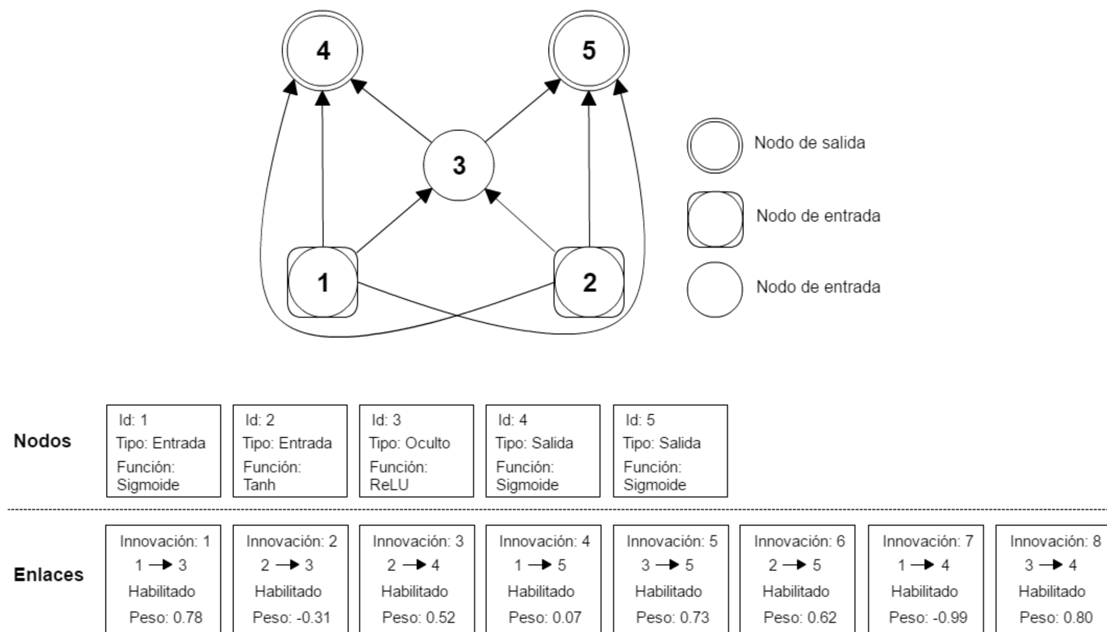


Figura 3.5: Codificación de una red neuronal en NEAT

Preservación de la innovación

La evolución de redes neuronales presenta el *problema de convenciones en competencia* (Schaffer et al., 1992), también conocido como el *problema de las permutaciones* (Radcliffe, 1993), que refiere a la existencia de varias configuraciones diferentes de una red neuronal que permiten aprender exactamente la misma función. La existencia de múltiples representaciones equivalentes dificulta la implementación de un operador de cruzamiento, dado que el cruzamiento entre soluciones equivalentes (dos redes neuronales diferentes que codifican la misma solución según el problema de las convenciones en competencia) suele destruir el conocimiento codificado en dichas soluciones. Es necesario identificar qué subconjuntos de dos redes a cruzar codifican el mismo conocimiento de formas diferentes para poder preservarlo. NEAT resuelve el problema de las convenciones en competencia definiendo el concepto de innovación, agregando a la representación de los individuos un marcador histórico que identifica a cada nuevo gen. Para ello se mantiene un contador global que es incrementado luego de mutar la estructura de una red y al enlace creado producto de cada mutación se le asigna el valor presente en ese momento en el contador. En caso de que dos mutaciones idénticas sucedan de forma independiente, se les asigna el mismo número de innovación. Los números de innovación no son cambiados una vez asignados y son preservados por el operador de cruzamiento. De esta forma es posible saber cuándo dos genes representan el mismo concepto. La información extra brindada por los marcadores históricos permite al operador de cruzamiento preservar las modificaciones introducidas en el proceso evolutivo.

Cruzamiento

La información provista por los números de innovación permite diseñar un operador de cruzamiento que respete la información codificada en los pesos de los enlaces y en la propia estructura de las redes a recombinar. El operador funciona alineando los enlaces de cada padre cuyo número de innovación coincide. Estos enlaces se denominan *coincidentes*. Los demás enlaces se denominan *disjuntos* o *de exceso*, dependiendo de si el número de innovación se encuentra dentro o fuera del rango de números de innovación del otro padre. Un nuevo individuo es formado seleccionando aleatoriamente uno de cada par de enlaces coincidentes y tomando todos los enlaces disjuntos y de exceso del padre que tiene mejor fitness. El pseudocódigo del operador de cruzamiento se presenta en el algoritmo 3. Un ejemplo de la aplicación del operador se presenta en la figura 3.6, donde el padre 2 tiene mayor fitness que el padre 1.

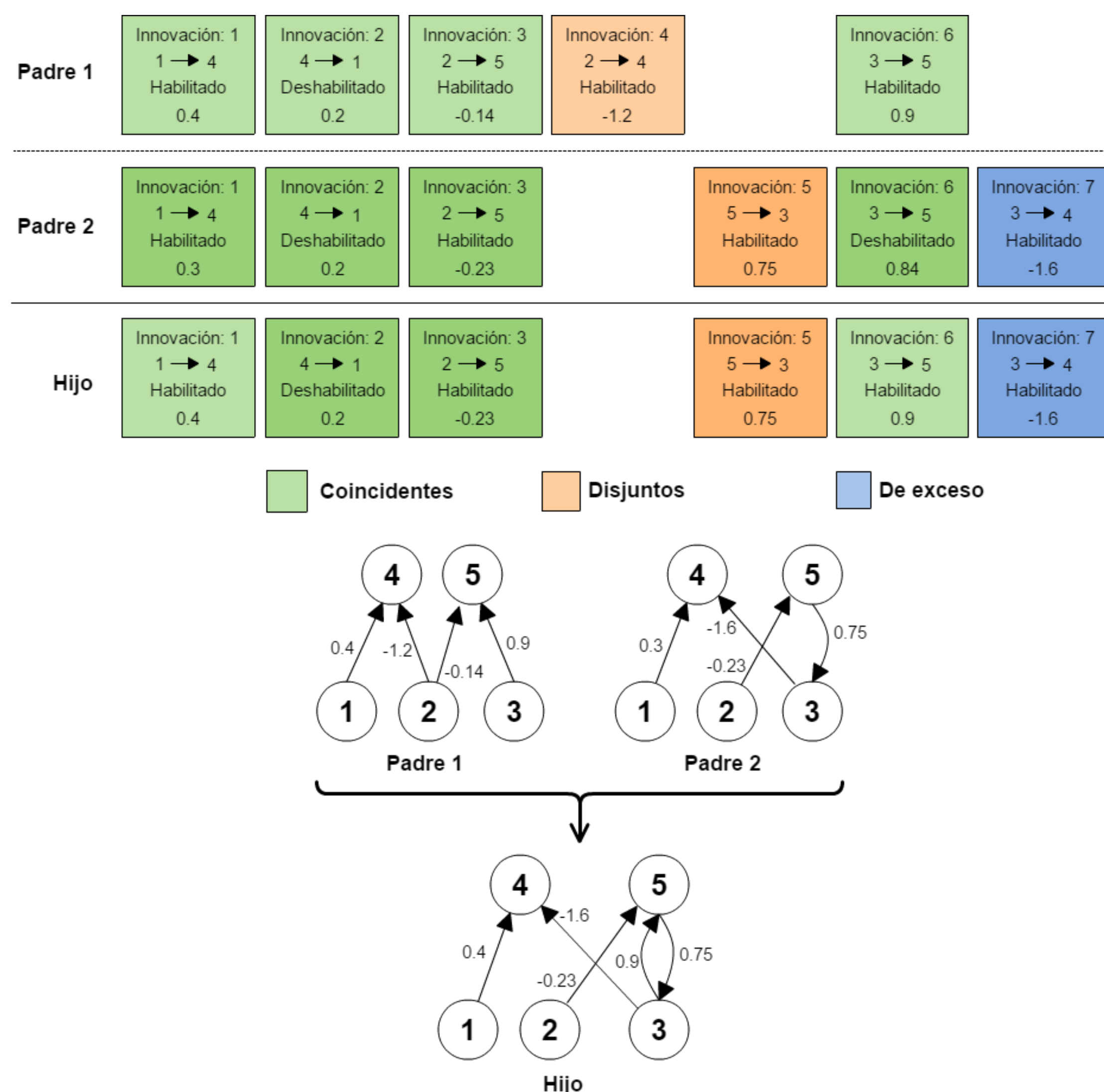


Figura 3.6: Ejemplo de cruzamiento en NEAT

Algoritmo 3 Esquema del operador de cruzamiento

Entrada: Padres Padre1, Padre2
  $Hijo \leftarrow nuevoCandidato()$
  para cada gen coincidente ($Padre1.gen1$, $Padre2.gen2$) **hacer**
    $Hijo.añadirGen(elegirAleatoriamente(gen1, gen2))$
  fin para
  $MejorPadre \leftarrow mejorPadre(Padre1, Padre2)$
  para cada gen disjunto y de exceso $gen \in MejorPadre$ **hacer**
    $Hijo.añadirGen(gen)$
  fin para
  retornar $Hijo$

Mutación

La propuesta presentada en el artículo de NEAT (Stanley y Miikkulainen, 2002) utiliza tres operadores de mutación distintos, cada uno con una probabilidad diferente de ser aplicado. El primer operador de mutación modifica de manera aleatoria los pesos de la red neuronal. Este operador es el más disruptivo, lo que permite a la búsqueda evolutiva salir con más facilidad de óptimos locales, siempre que esto sea posible según la forma de la función de fitness.

Los otros operadores realizan cambios estructurales en la red y son los responsables de incrementar el tamaño de los individuos. Se define un operador de mutación para agregar un nuevo enlace, que selecciona aleatoriamente dos nodos existentes no conectados previamente y le asigna al enlace un peso aleatorio.

El tercer operador de mutación introduce un nuevo nodo, seleccionando un enlace existente y colocando un nuevo nodo en esa posición. El enlace antiguo es deshabilitado y el operador agrega dos enlaces nuevos. El enlace de entrada al nodo recibe un peso de valor 1 y el enlace de salida toma el peso antiguo del nodo deshabilitado. Insertar el nuevo nodo de esta forma hace que el único cambio sea producido por la función de activación del nodo nuevo, ya que el peso asociado al enlace de entrada al nodo insertado no modifica el valor recibido (por tener peso 1) y el peso del enlace saliente al nodo es igual al peso original del enlace deshabilitado. El objetivo es reducir la disruptividad del operador, con el objetivo de mantener la funcionalidad de la red y así permitir que el cambio perdure, en vez de ser descartado por disminuir significativamente el fitness de la red. Un ejemplo de la aplicación de este operador puede verse en la figura 3.7.

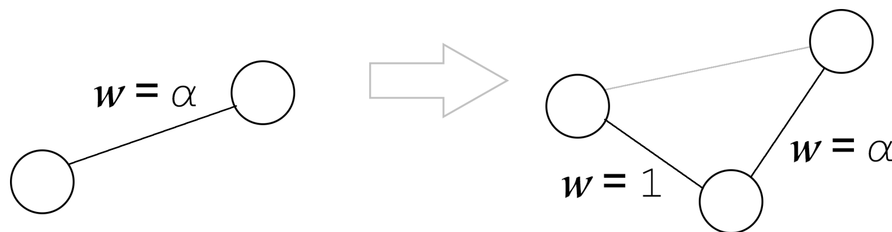


Figura 3.7: Ejemplo de mutación que agrega un nodo con peso unitario

En este proyecto se introduce un cuarto operador en NEAT, que también es aplicado según una probabilidad de aplicación definida en la configuración del algoritmo evolutivo. Este nuevo operador muta las funciones de activación de todas las neuronas excepto las

de salida. La motivación de este operador es brindar al proceso evolutivo más recursos para adaptar las redes al juego que se aprende, a diferencia de la propuesta inicial de NEAT que no modifica las funciones de activación. Al aplicar el operador se decide de forma aleatoria si se cambian los parámetros de la función de activación usada por el nodo o si se reemplaza la función por otra. En el primer caso se determina aleatoriamente si se modifican los parámetros o si se les asignan nuevos valores con una probabilidad del 1 %. En caso de modificar los parámetros de la función, al valor de cada parámetro se le suma el resultado de multiplicar dicho valor del parámetro por un número aleatorio que sigue una distribución Gaussiana de parámetros $\mu = 0$ y σ dependiente de la función de activación siendo mutada. Por ejemplo, la función de activación Gaussiana utiliza $\sigma = 0{,}1$ mientras que la función de activación sigmoide utiliza $\sigma = 0{,}3$. Finalmente, en caso de reemplazar la función de activación, se elige aleatoriamente una nueva función de activación entre las funciones soportadas y se asignan aleatoriamente valores a los parámetros que acepte la nueva función. Este nuevo operador es una contribución específica del trabajo desarrollado en este proyecto para la aplicación de NEAT al problema de generación automática de IAs para juegos del NES.

Especies

La sucesiva aplicación de los operadores de cruzamiento y mutación puede generar una población de redes con topologías muy diversas. Sin embargo, la diversidad no se mantiene por sí sola, debido a que agregar una neurona o enlace a una red generalmente disminuye su fitness hasta que el algoritmo evolutivo pueda ajustar los pesos de las nuevas conexiones. Además, el ajuste de los pesos es más rápido cuando hay menos enlaces presentes en una estructura, ya que el subconjunto del espacio de soluciones considerado por la red se ve reducido. Para permitir que redes recientemente aumentadas (por haber agregado una neurona o un enlace) tengan tiempo suficiente para ajustar sus nuevos pesos, se debe utilizar un mecanismo que las proteja durante el proceso evolutivo. El mecanismo propuesto por NEAT para lograr dicha protección es el uso de *especies*.

Una especie es una agrupación de individuos que poseen información genética similar, de acuerdo a una función de distancia predefinida. Para formalizar la noción de distancia entre individuos es necesario definir una función de compatibilidad que permite medir la similitud entre individuos. La función de compatibilidad contabiliza la cantidad de enlaces disjuntos D y de exceso E que tienen dos individuos y la diferencia promedio de los pesos de los enlaces coincidentes $\overline{W}$ (incluyendo aquellos deshabilitados), según la ecuación 3.2.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W} \tag{3.2}$$

En la ecuación 3.2, los coeficientes c_1, c_2 y c_3 ajustan la importancia de los tres factores considerados, mientras que N es el número de enlaces en el individuo de mayor tamaño, que es utilizado para normalizar los valores obtenidos. Empleando la función δ se pueden definir especies según un parámetro de tolerancia denominado *umbral de compatibilidad* δ_t. El algoritmo de asignación de individuos a especies funciona manteniendo un conjunto de especies, cada una representada por un individuo de la población. Cada individuo en una generación es asignado a la primer especie en la cual la distancia δ con el representante de esa especie sea menor a δ_t.

NEAT utiliza *compartición de fitness explícita* (Goldberg y Richardson, 1987), por lo que los individuos de una especie comparten el fitness. De esta forma es muy difícil que una especie tenga demasiados individuos en comparación con el resto, ya que cuantos más individuos tiene una especie, menor es el fitness de cada individuo que la compone. La aplicación de compartición de fitness reduce la probabilidad de que toda la población pertenezca a una sola especie. La ecuación 3.3 define el fitness ajustado $\widehat{f_i}$ del individuo i de la generación en un tiempo t dado, donde $\delta(i,j)$ representa la distancia entre los individuos i y j.

$$sh(\delta(i,j)) = \begin{cases} 1 & \text{si } \delta(i,j) < \delta_t \\ 0 & \text{sino} \end{cases} \qquad m_i = \sum_{j=1}^{n} sh(\delta(i,j))$$
$$\widehat{f_i} = \frac{f_i}{m_i} \tag{3.3}$$

En la ecuación 3.3, $sh(\delta(i,j))$ es una función que indica si los individuos i y j pertenecen a la misma especie y n es la cantidad de individuos por generación. El factor m_i corresponde a la cantidad de individuos que pertenecen a la misma especie que el individuo i, ya que las especies están construidas siguiendo la misma definición que la función sh. El cruzamiento está restringido entre individuos que pertenezcan a la misma especie. La cantidad de individuos que cada especie contribuye para la siguiente generación depende proporcionalmente de la suma de los fitness ajustados $\widehat{f_i}$ de todos sus miembros. Al momento de realizar el cruzamiento se seleccionan los individuos necesarios de cada especie para alcanzar la contribución de cada una y se utiliza el operador de cruzamiento definido en la sección 3.4.2.

Selección

NEAT utiliza un operador de selección denominado *selección intraespecie*, que selecciona aquellos individuos pertenecientes a una especie que serán cruzados. Es posible que la cantidad de individuos generados por todas las especies sea menor a la necesaria para la siguiente generación, debido a la aplicación de redondeo al calcular la cantidad de hijos a producir en función de la proporción de fitness que aporta la especie. Para generar los individuos faltantes se utiliza un segundo operador de selección denominado *selección interespecie*, que selecciona individuos de toda la población para poder ser cruzados.

Ambos operadores son implementados de la misma forma que el operador de selección tradicional de un algoritmo evolutivo. Ejemplos de implementaciones posibles son la selección proporcional, la selección estocástica universal, la selección por ranking, la selección por torneo, etc.

Generación de la población inicial

La población inicial se constituye únicamente de individuos que representan una red sin nodos ocultos, con todos los enlaces posibles que conectan nodos de entrada con nodos de salida. Los pesos de los enlaces de cada individuo se eligen de forma aleatoria. Estos individuos se denominan *minimales* debido a que no se puede tener una red con menos nodos que pueda considerarse como solución para el problema estudiado.

Crecimiento de las redes

Teniendo en cuenta que la generación inicial está compuesta exclusivamente por individuos minimales, la modificación de la representación para lograr el crecimiento de las redes sucede únicamente por el uso de operadores de mutación. Una red que sufre un cambio estructural debido a una mutación sólo sobrevivirá si el cambio introducido resulta útil según la evaluación provista por el fitness (Stanley y Miikkulainen, 2002). Por tanto, el incremento de tamaño de los individuos de una población está siempre justificado; si el aumento no es beneficioso para el aumento de fitness, el individuo será descartado por el algoritmo evolutivo. Iniciar la evolución con una población compuesta por individuos minimales agrega un sesgo hacia la generación de redes neuronales de tamaño reducido. Este sesgo logra minimizar el subconjunto del espacio de búsqueda considerado por las redes, mejorando la performance computacional del algoritmo evolutivo. Para que el algoritmo evolutivo funcione correctamente es necesario utilizar especies para proteger la innovación provista por cambios estructurales importantes en las redes.

Generalmente se requiere una probabilidad alta de mutación para que el proceso neuroevolutivo funcione correctamente. Utilizar una probabilidad de mutación alta tiene dos motivos. En primer lugar, los operadores de mutación presentados son poco disruptivos, especialmente cuando se trabaja con redes neuronales de gran tamaño. En segundo lugar, los operadores de mutación son los únicos que permiten aumentar las estructuras de las redes. Un algoritmo evolutivo convencional en estas condiciones podría degenerar en una búsqueda aleatoria, pero esto no sucede en NEAT. En un algoritmo evolutivo degenerado el mecanismo de especies no funciona, dado que cada especie tiene un solo miembro (la probabilidad de encontrar de forma aleatoria dos individuos suficientemente similares es despreciable en espacios suficientemente grandes). El correcto funcionamiento de la especiación se debe a la baja disruptividad de los operadores evolutivos y a la aplicación del operador de cruzamiento entre individuos de la misma especie, enfocando la búsqueda a una sección particular del espacio de soluciones. Por este motivo la especiación brinda robustez frente a mutaciones frecuentes (Stanley y Miikkulainen, 2002).

Capítulo 4

Trabajos relacionados

Este capítulo presenta los trabajos más relevantes encontrados en la literatura relacionada sobre algoritmos evolutivos y neuroevolución aplicados a la generación automática de IAs. Las primeras cinco secciones presentan cada una un trabajo relacionado, resumiendo brevemente el contenido del trabajo y comparándolo con el trabajo desarrollado en este proyecto de grado. La sexta sección ofrece un resumen de la bibliografía relevada y concluye sobre la relación de este proyecto de grado con la bibliografía.

4.1. Panorama de técnicas de neuroevolución

Simpson (2012) presentó un relevamiento del uso de algoritmos evolutivos para juegos. El autor comenzó presentando un análisis cualitativo comparando las ventajas y desventajas que conlleva el uso de técnicas evolutivas para desarrollar IAs para juegos. El análisis argumentó que el costo reducido de generar IAs y la mejora de calidad de los jugadores obtenidos justifican el costo de aplicar estas técnicas.

Las principales desventajas consideradas por Simpson son el costo computacional de los algoritmos evolutivos, el tiempo requerido para obtener de una IA funcional, la falta de garantías de obtener un buen resultado y el hecho de que las redes neuronales son interpretadas como cajas negras, lo que dificulta el análisis y la comprensión del comportamiento que presenta la IA. Si bien estos argumentos son válidos, el costo computacional al generar las IAs en el caso estudiado en este proyecto de grado es significativamente menor en comparación al desarrollo manual de una IA, debido al uso de técnicas de evaluación paralela y a la alta complejidad que presenta el desarrollo de IAs de forma manual.

Finalmente, el relevamiento se concentró en técnicas de neuroevolución, presentando ejemplos de casos de éxito (*Nero*, *Robocode* y *Wolfenstein 3D*) e incluyendo un breve análisis del resultado obtenido y su importancia. Nero es un juego cuyo objetivo es el entrenamiento de ANNs que controlan robots que se enfrenten de forma autónoma a enemigos. El jugador modifica interactivamente la función de fitness de un algoritmo evolutivo encargado de ajustar los pesos de las redes. El estudio realizado sobre el juego Nero evidenció la velocidad con la que pueden ser entrenadas las redes, ya que tomó entre 100 a 120 segundos luego de un cambio de la función objetivo para que se produzcan ANNs que cumplieran con el nuevo objetivo. Sin embargo, es difícil poner en contexto estos tiempos de ejecución, debido a que no se especificó la plataforma donde fue ejecutado el algoritmo evolutivo. En el caso de Robocode, se desarrolló una IA utilizando

algoritmos genéticos que fue capaz de quedar en tercer lugar en una competencia con 27 IAs. Las demás IAs fueron programadas de forma manual por humanos, por lo que el resultado fue considerado satisfactorio para los autores. El trabajo sobre Wolfenstein 3D buscó reemplazar las IAs presentes en el juego por ANNs entrenados usando algoritmos evolutivos. Como métrica para evaluar la mejora obtenida se tomó el tiempo que requirieron jugadores humanos para terminar un nivel utilizando las IAs por defecto y se lo comparó con el tiempo necesario para vencer a las nuevas IAs. Los jugadores humanos requirieron alrededor de 25 segundos más para vencer a las nuevas IAs en comparación con los niveles originales, indicando que el método utilizado para automatizar la generación de IAs fue eficaz y obtuvo resultados de mejor calidad que las IAs desarrolladas de forma manual. Todos estos resultados sugieren que la automatización del proceso de desarrollo de IAs utilizando técnicas neuroevolutivas es una herramienta válida a tener en cuenta al momento de comenzar la implementación de una IA para un videojuego.

4.2. Robocode

Hong y Cho (2004) desarrollaron una técnica para generar comportamientos emergentes complejos para una IA del juego *Robocode*. Los autores propusieron un algoritmo evolutivo que evoluciona combinaciones de comportamientos primitivos simples obteniendo como resultado patrones complejos. Robocode provee una representación del estado del juego y funciones de acceso y manipulación diseñadas para la implementación de IAs, dado que el objetivo del juego es crear una IA particular que sea capaz de derrotar a los adversarios en una batalla de tanques. El uso de estas interfaces y los datos provistos por Robocode impiden generalizar las técnicas presentadas de forma inmediata a otros juegos, debido a que se requiere un gran esfuerzo de adecuación del juego en cuestión para proveer la información necesaria a la IA generada y permitir que ésta interactúe con el entorno virtual del juego.

Los comportamientos primitivos presentados son pequeños fragmentos de código que codifican acciones básicas que el tanque controlado por la IA puede realizar, como son moverse, disparar, apuntar, evadir un disparo o buscar un enemigo. Cada acción posible tiene un número definido de variantes de comportamientos primitivos que fueron implementadas por los autores. Por ejemplo, en el caso de moverse, los autores implementaron el movimiento aleatorio, el movimiento circular, el movimiento lineal y otras variantes. El algoritmo evolutivo es usado para optimizar la combinación de los comportamientos primitivos. Cada individuo codifica el comportamiento de la IA como un arreglo de seis enteros, donde cada entero indica la variante del comportamiento primitivo utilizado para esa acción (moverse, evadir, disparar, cargar un disparo, usar el radar, apuntar). Esta codificación simple facilita el proceso de aprendizaje, mejorando la posibilidad de obtener jugadores de una calidad aceptable, pero limita la expresividad de las soluciones y la capacidad de encontrar estrategias nuevas.

El algoritmo evolutivo utilizado utilizó cruzamiento de un punto, una mutación simple para modificar el número que indica qué comportamiento primitivo es utilizado para cada acción, selección proporcional y elitismo para preservar la mejor solución encontrada en la evolución.

Para evaluar la aptitud de cada IA se ejecutaron tres simulaciones de batallas donde se enfrentó contra IAs provistas por el juego (cada una con comportamientos específicos). La función objetivo utilizada fue $f = \frac{puntaje_{propio}}{puntaje_{propio} + puntaje_{oponente}}$, donde $puntaje_{propio}$ es

la suma del puntaje obtenido por la IA entrenada con el algoritmo evolutivo en las tres simulaciones y $puntaje_{oponente}$ es la suma del puntaje obtenido por la IA provista por el juego en las tres simulaciones. Además, se enfrentó contra una de las IAs vencedoras de la competencia *Robocode Rumble* de 2002. Los resultados obtenidos mostraron que las IAs generadas con algoritmos evolutivos utilizando comportamientos primitivos fueron capaces de vencer a todos los oponentes utilizados para verificar su capacidad, incluida la IA vencedora de Robocode Rumble. Cada oponente requirió una estrategia diferente para poder vencerlo. Los autores destacaron que el algoritmo evolutivo fue capaz de encontrar varias técnicas inesperadas, con resultados positivos. Estos resultados muestran la capacidad de un método automatizado para generar jugadores competentes y con comportamiento variado que resulte en oponentes con una dificultad aceptable y que no resulten repetitivos para jugadores humanos.

4.3. Super Mario

Jørgensen y Sandberg (2009) aplicaron algoritmos genéticos y aprendizaje por refuerzos para optimizar los pesos de una red neuronal artificial, con el objetivo de aprender a jugar una variante especial del juego *Super Mario* que provee facilidades para el aprendizaje. En particular, el juego permite la llamada a funciones que devuelven una grilla representando el mapa y la distancia a los enemigos y a los obstáculos en pantalla, entre otras. El uso de información simplificada del juego facilita el entrenamiento de la red neuronal debido a que no es necesario entrenarla también para realizar el procesamiento y adecuación de los datos del juego. Por otra parte, el uso de una interfaz específica para acceder a la información del juego dificulta la generalización de las técnicas presentadas, debido a que se requieren modificaciones en otros juegos para que provean la interfaz requerida.

Se entrenaron redes neuronales de estructura fija: 14 neuronas de entrada, 20 neuronas ocultas y 5 neuronas de salida. Cada neurona presentó conexiones con todas las neuronas de la capa siguiente. El algoritmo evolutivo utilizó una función objetivo que combina la distancia recorrida, la cantidad de enemigos eliminados, el tiempo restante y las vidas perdidas. Los autores observaron que la distancia recorrida por los jugadores artificiales es altamente dependiente de la forma en que se combinen las variables de la función objetivo. Las evaluaciones de los jugadores fueron realizadas sobre mapas clasificados como simples o complejos considerando la cantidad de obstáculos y el diseño general de cada nivel. Los autores entrenaron las redes neuronales utilizando como casos de entrenamiento un solo nivel, o múltiples niveles. Los resultados fueron comparados con agentes simples utilizando estrategias predefinidas y un agente basado en el algoritmo A* (Hart et al., 1968). Las redes neuronales presentaron mejores resultados que las técnicas simples, pero se vieron superadas consistentemente por el algoritmo A*.

Al utilizar aprendizaje por refuerzo los autores eligieron la técnica de *Q-learning* (Watkins, 1989). La implementación de Q-learning propuesta otorgó premios al agente únicamente cuando logró mover a Mario hacia la derecha de la pantalla. Las redes neuronales fueron entrenadas sobre 10000 partidas utilizando la misma configuración empleada con algoritmos evolutivos. Los jugadores generados fueron capaces de completar los niveles simples. Sin embargo, para los niveles complejos se observaron desempeños cambiantes dependiendo del nivel. Para diagnosticar por qué los resultados fueron tan variados, los autores entrenaron redes sobre niveles con un solo tipo de obstáculos

(solamente enemigos, fosos o impedimentos al avance) y constataron su capacidad de aprender correctamente para todos los casos estudiados. Al trabajar con combinaciones de obstáculos, los jugadores tuvieron problemas para generalizar las estrategias individuales, siendo capaces de resolver partes del nivel pero fallando luego en algún obstáculo particular. Los autores consideraron que este comportamiento se debió a la capacidad limitada de visión del mundo provista a los jugadores. Mejorar la capacidad de visión, sin embargo, demanda un mayor costo computacional, por lo que ambos aspectos deben ser balanceados.

4.4. Atari

Hausknecht et al. (2014) presentaron un proyecto con una idea similar a la de este proyecto de grado. Por medio de técnicas neuroevolutivas, los autores entrenaron jugadores artificiales para 61 juegos de la consola Atari 2600. Para ello utilizaron un emulador especializado, *Arcade Learning Environment* (ALE), que provee facilidades para desarrollar IAs que interactúen con los juegos emulados. A diferencia de los trabajos de Jørgensen y Sandberg (2009) y Hong y Cho (2004), la metodología utilizada no depende del juego estudiado, por lo que es aplicable a una gran cantidad de juegos sin modificar la solución presentada.

Los autores propusieron el uso de cuatro técnicas neuroevolutivas. La primera, *Conventional NeuroEvolution* (CNE), evoluciona solamente los pesos de una red neuronal cuya topología es fija. CNE utiliza operadores de cruzamiento y mutación al igual que los algoritmos evolutivos convencionales. Además, los autores introdujeron el concepto de especies (similar al usado por NEAT) para mejorar la diversidad de la población. El segundo método utilizado fue *Covariance Matrix Adaptation Evolution Strategy* (CMA-ES), que evoluciona una red neuronal de topología fija utilizando un algoritmo de búsqueda sobre las políticas de juego posibles. Las políticas son seleccionadas a partir de una distribución gaussiana multivariada, modificada luego de evaluar las políticas candidatas. El tercer método estudiado fue NEAT. El último método propuesto fue *HyperNEAT*, que evoluciona una codificación indirecta utilizando una *Compositional Pattern Producing Network* (CPPN), una red que define los pesos de la ANN a entrenar. La técnica mantiene una población de CPPNs evolucionadas utilizando NEAT, que al momento de ser evaluadas generan una ANN utilizada para jugar la partida. Todas las redes evolucionadas son del tipo feed-forward, por lo que no mantienen memoria de eventos previos en el juego. Los autores propusieron el uso de redes recurrentes como un trabajo futuro para mejorar el desempeño en varios juegos que dependen de actuar en concordancia a eventos pasados. En este proyecto de grado se avanza sobre este trabajo previo, trabajando con redes recurrentes persiguiendo dicho objetivo.

Los autores utilizaron tres representaciones diferentes del estado del juego, basadas en diferentes técnicas de visión por computadora. La primera se basa en la identificación manual por parte de un humano de objetos gráficos presentes en la pantalla del juego que son clasificados en distintas clases. Al momento de jugar una partida, el sistema reconoce los objetos en pantalla en base a estas clasificaciones y provee la información procesada a la ANN. Como indicaron los autores, esta representación no es general y requiere de un trabajo manual para cada juego, lo que va en contra del espíritu de automatización total del desarrollo de IAs. La segunda representación toma la información de píxeles directamente de la salida gráfica del emulador, utilizando una paleta de colores reducida para disminuir el volumen de información a procesar. Esta representación es la más

general de todas las estudiadas en el trabajo. La última representación se basa en proveer ruido predefinido como entrada al proceso de aprendizaje. Para ello se genera de manera aleatoria información visual de entrada. El objetivo de utilizar la representación basada en ruido fue estudiar la capacidad de memoria de los algoritmos empleados al aprender secuencias de comandos que aumenten el valor de la función objetivo de cada técnica neuroevolutiva sin tener información de las acciones que están sucediendo en el juego. Además, los autores utilizaron esta representación como un resultado base para evaluar la mejora provista por las otras dos representaciones. Los autores postularon que si ambas representaciones lograban generalizar conceptos y no solo memorizar secuencias de teclas, debían obtener mejores resultados que al emplear la representación basada en ruido.

La evaluación fue realizada sobre una selección de 61 juegos de Atari, escogidos de forma de lograr una representación equitativa de distintos géneros (acción, aventura, puzles, etc.) y distintas perspectivas de juego (2D y 3D). Se utilizaron todos los algoritmos con todas las representaciones, excepto en los casos donde el problema se volvía intratable computacionalmente. La técnica CNE fue utilizada como base de comparación de resultados de las cuatro técnicas neuroevolutivas por ser la más simple. Además, los resultados fueron comparados contra algoritmos SARSA(λ) (State Action Reward State Action), algoritmos de planificación, jugadores aleatorios y jugadores humanos.

Los resultados fueron comparados entre las cuatro técnicas. NEAT superó a CNE y CMA-ES utilizando la representación de objetos gráficos y tuvo resultados comparables a HyperNEAT. Utilizando la representación de ruido, NEAT superó a CNE y HyperNEAT. Finalmente, HyperNEAT fue el único algoritmo que logró aprender a jugar utilizando los píxeles de la pantalla. Sin embargo, en las representaciones de objetos y ruido, los mejores resultados generales fueron obtenidos por NEAT. Los autores concluyeron que mientras la representación sea adecuada, NEAT es una técnica eficaz para generar jugadores para la consola Atari.

Al comparar los resultados obtenidos contra otras técnicas de automatización y jugadores humanos, se constató que NEAT fue superado por los algoritmos de planificación y por jugadores humanos expertos. El éxito de las técnicas de planificación fue atribuido a su capacidad de ver estados futuros. Sin embargo, las técnicas neuroevolutivas lograron superar a los humanos en tres juegos y encontraron formas de generar un puntaje infinito en otros tres. Al analizar los 61 juegos con detenimiento, existen varios títulos para los cuales las técnicas evolutivas superaron a los algoritmos de planificación. El punto débil de los algoritmos evolutivos surgió de la desviación de los objetivos del juego en pos de maximizar el puntaje, como constataron los autores.

Finalmente, algunos de los resultados obtenidos encontraron errores en el juego en cuestión o generaron técnicas muy difíciles de replicar por seres humanos. Los casos donde se obtuvieron puntos infinitos caen en estas categorías. La explotación de estos errores por parte de las IAs entrenadas apoya la idea de que los jugadores generados automáticamente pueden ser utilizados para tareas de verificación, como se propone estudiar en este proyecto.

4.5. Playfun

Murphy (2013) presentó un sistema de aprendizaje automático basado en videos de entrenamiento para la consola NES, con una idea similar a la del presente proyecto. Las técnicas planteadas por Murphy no dependen del juego estudiado, por lo que pueden ser aplicadas a cualquier juego de la consola NES sin modificación alguna.

El autor desarrolló un método basado en dos etapas, que requieren como entrada un video de entrenamiento donde se muestre la forma esperada de jugar al juego en cuestión. La primera etapa, implementada por el programa Learnfun, infiere los objetivos del juego a partir del video de entrenamiento. Para ello intenta detectar órdenes lexicográficos crecientes sobre valores de la RAM del juego. La función objetivo es definida como una combinación lineal de todos los órdenes encontrados. Considerar solamente órdenes crecientes implica que el método no puede representar objetivos que requieran disminuir en magnitud para ser cumplidos, como puede ser derrotar a un adversario que tiene un contador de los puntos de vida restantes. La búsqueda de los órdenes lexicográficos se realiza de manera estocástica, debido a que la mayor parte de los miembros de cada orden no se encuentran contiguos en RAM. La segunda etapa, implementada por el programa Playfun, realiza una búsqueda local combinada con backtracking para explorar el espacio de entradas posibles al juego, buscando maximizar el valor de la función objetivo hallada por Learnfun. Además, para agilizar la búsqueda Learnfun deduce *motifs*, combinaciones de teclas que son presionadas en conjunto por el jugador humano en el video de entrenamiento. Playfun utiliza estos motifs para reducir considerablemente el espacio de búsqueda. El mecanismo de búsqueda se basa en el concepto de *futures*, un conjunto de motifs que pueden ser ejecutados. El conjunto de futures es consumido a lo largo del proceso de búsqueda y debe ser rellenado con nuevos futures. Cada paso de búsqueda analiza el estado del juego en los próximos 10 frames y ejecuta la combinación de teclas que conduce a un mejor estado. Con el objetivo de evitar estancarse en máximos locales, se agregan al conjunto nuevos futures compuestos por motifs aleatorios. Estos futures permiten al algoritmo explorar otros caminos que puedan aportar una solución mejor a la que puede ser generada por los futures presentes hasta el momento en el conjunto. El componente que ejecuta backtracking intenta mejorar fragmentos de una partida, retrocediendo el estado de la simulación una cantidad predefinida de frames y ejecutando la búsqueda sobre nuevos futures candidatos por medio de diferentes técnicas de generación. El algoritmo compara todos los estados alcanzados por las diferentes ramas del árbol de backtracking y selecciona la combinación de teclas que obtenga el mejor desempeño. El resultado del proceso es la combinación de teclas necesarias para jugar una partida particular, en formato de video del emulador FCEUX (formato fm2). Para jugar una partida nueva se debe iniciar el proceso nuevamente.

La evaluación del desempeño de Learnfun y Playfun fue realizada de forma cualitativa sobre varios juegos de diferentes géneros de la plataforma NES. La combinación de búsqueda local y backtracking permitió a Playfun alcanzar muy buenos resultados en varios juegos, dado que Playfun puede ver el futuro cercano y deshacer sus acciones si éstas no son favorables. Sin embargo, Playfun requiere una gran cantidad de recursos computacionales para jugar una sola partida. Playfun mostró un buen desempeño sobre múltiples juegos, como Pinball, Super Mario Brothers y Arkanoid. Todos estos juegos requieren buenos reflejos y una capacidad limitada de planificación hacia el futuro. Además, los objetivos de estos juegos son crecientes (puntaje, progresión de niveles), lo que los hace idóneos para Playfun. La capacidad de ver estados futuros permite generar combinaciones de teclas que manipulan el generador de números aleatorios utilizado en el juego en cuestión. A modo de ejemplo, en el juego Arkanoid, Playfun fue capaz de forzar al juego a producir un potenciador que le permitió pasar automáticamente al siguiente nivel. También pudo manipular el rebote de la bola, como es evidenciado en uno de los videos publicados por Murphy, que compara dos grabaciones de la misma partida jugada

por Playfun, hasta que en una de ellas se suprimen las entradas de la IA, momento en el cual el movimiento de la bola diverge de forma favorable para la partida aún jugada por Playfun. En juegos como Dare (trivia) o Ice Hockey, que presentan objetivos oscilantes o difíciles de codificar, se obtuvieron resultados marcadamente negativos. Estos resultados son esperables debido a que la codificación de objetivos, al considerar solamente funciones crecientes, no permite expresar funciones oscilantes o que tengan componentes que decrecen con el tiempo. Estos resultados también fueron visibles en juegos como Tetris que requieren de mucha planificación a futuro (ya que las jugadas realizadas en el presente afectan significativamente el estado futuro del juego). La búsqueda local aplicada en Playfun no tiene en cuenta estados muy alejados en el tiempo, impidiendo planificar en el futuro lejano.

Finalmente, varios jugadores obtenidos mostraron técnicas poco ortodoxas de juego, explotando errores de programación y diseño. Estos resultados son extremadamente útiles para realizar una verificación automatizada de un juego particular, que permita visualizar en las partidas jugadas por la IA defectos que deben ser corregidos antes de liberar un producto al mercado. Sin embargo, el costo computacional de Playfun es demasiado alto debido a la naturaleza exponencial del algoritmo de backtracking y que se deben mantener varios estados del juego (uno por cada rama considerada). A modo de ejemplo, generar una partida de 1 minuto y 26 segundos para el juego Pinball requirió 17.5 horas de ejecución en una máquina con procesador Intel i7-920 y 6GB de RAM DDR3 (aproximadamente 733 segundos de ejecución por cada segundo de juego). El costo resulta prohibitivo para realizar varios experimentos independientes con fines de verificación.

4.6. Resumen y conclusión

Los trabajos relevados muestran que la aplicación de métodos de automatización para la generación de IAs permite obtener resultados de buena calidad, sugiriendo que existe potencial para su utilización en videojuegos. En particular, Hausknecht et al. (2014) y Murphy (2013) obtuvieron IAs que fueron capaces de explotar errores de programación y de diseño de videojuegos, logrando ventajas no previstas por el programador. Este tipo de IAs pueden ser utilizadas para verificar de forma automática un juego en desarrollo, liberando recursos humanos que pueden ser invertidos en otras áreas de desarrollo, como es la programación del motor de juego, la redacción de la narrativa, el diseño de los elementos gráficos, etc. La tabla 4.1 presenta un resumen de los trabajos estudiados.

Autores	Año	Línea de trabajo
Hong y Cho	2004	Evolución de comportamientos para el juego Robocode
Jørgensen y Sandberg	2009	Juego automatizado de Super Mario Bros utilizando algoritmos evolutivos
Simpson	2012	Panorama de técnicas evolutivas para generación de IAs para videojuegos
Murphy	2013	Aprendizaje automático de juegos de NES
Hausknecht, Lehman, Miikkulainen y Stone	2014	Aprendizaje automático de juegos de Atari utilizando neuroevolución

Tabla 4.1: Resumen de los trabajos analizados

Dentro de la investigación realizada, no fue posible encontrar un trabajo relacionado que utilizara técnicas neuroevolutivas para generar IAs de forma totalmente automatizada para la NES, ni con objetivos de verificación. La falta de trabajos previos que traten sobre verificación automatizada sugiere que el aporte realizado por este proyecto de grado es novedoso dentro del campo de la inteligencia artificial.

Los resultados recopilados por Simpson (2012) soportan la idea de que el uso de técnicas neuroevolutivas es una herramienta eficaz para producir IAs competentes.

Al considerar los resultados obtenidos por Hong y Cho (2004) se constata que el uso de elitismo resulta esencial para mejorar la velocidad de aprendizaje del algoritmo evolutivo.

La disponibilidad de información simplificada y en un formato especificado para su fácil manejo por un jugador artificial diferencian las técnicas aplicadas por Jørgensen y Sandberg (2009) del trabajo presentado en este proyecto de grado, donde la IA debe realizar las tareas de procesamiento de información y simplificación a un formato específico que permita tomar decisiones sobre qué acciones realizar en el juego.

Los resultados obtenidos por los Hausknecht et al. (2014) al utilizar NEAT impulsaron su uso en este proyecto de grado, debido a las similitudes del problema estudiado. Sin embargo, la consola NES presenta una mayor complejidad tener más RAM, cartuchos con más capacidad y un procesador más potente que la Atari, por lo que se requiere incorporar otras consideraciones para obtener buenos resultados. Además, en este proyecto de grado se incluye el uso de redes neuronales recurrentes para obtener un mejor desempeño, como proponen Hausknecht et al. como trabajo futuro.

Finalmente, con el fin de superar la limitación de poder representar solamente funciones objetivo crecientes exhibida por el trabajo de Murphy (2013), el algoritmo presentado en este proyecto de grado busca órdenes crecientes y decrecientes. Además, se complementó la búsqueda estocástica de órdenes lexicográficos al intentar localizar primero órdenes cuyos miembros son contiguos en RAM, que según pruebas empíricas existen en todos los juegos estudiados. Este proyecto también propone mejorar la capacidad de codificación de objetivos provista por los órdenes lexicográficos, al utilizar órdenes y objetivos individuales para mejorar la expresividad. Además, se intenta obtener un mejor desempeño en juegos difíciles, como Tetris o Ice Hockey, aprovechando la mayor expresividad de objetivos y la capacidad de codificar estrategias complejas y dependencias con jugadas anteriores que brindan las redes neuronales recurrentes. El uso de técnicas neuroevolutivas tiene como objetivo obtener resultados similares a los alcanzados por Playfun en su capacidad de explotar errores, pero con un costo computacional mucho menor. Todas estas consideraciones constituyen contribuciones específicas de la investigación desarrollada en este proyecto de grado.

Capítulo 5

Algoritmos evolutivos aplicados a la generación automática de inteligencias artificiales para NES

Este capítulo presenta la implementación de un algoritmo neuroevolutivo utilizando el método NEAT para automatizar la generación de jugadores artificiales para distintos títulos de la plataforma NES.

5.1. Introducción

El proyecto desarrollado en este proyecto de grado comprende la creación de un pipeline que permita la generación automática de IAs de forma eficiente. El pipeline tiene los siguientes componentes: inferencia de objetivos, algoritmo evolutivo para el refinamiento de objetivos y un algoritmo evolutivo para el entrenamiento de la red neuronal que implementa la IA.

La figura 5.1 muestra la relación entre los componentes del pipeline. La entrada del pipeline son los videos de entrenamiento provistos por jugadores humanos. La salida obtenida es la mejor red neuronal generada, que puede ser utilizada por el módulo de IA para jugar automáticamente una partida del juego en cuestión. Para ejecutar la simulación de las partidas se utiliza el emulador de código abierto FCEUX, el cual fue modificado sustancialmente con el fin de mejorar la eficiencia computacional y para poder usar la red neuronal. Finalmente, se implementó un framework evolutivo completo para tener más control sobre el proceso de optimización. En las secciones siguientes se detallan los distintos componentes de la solución.

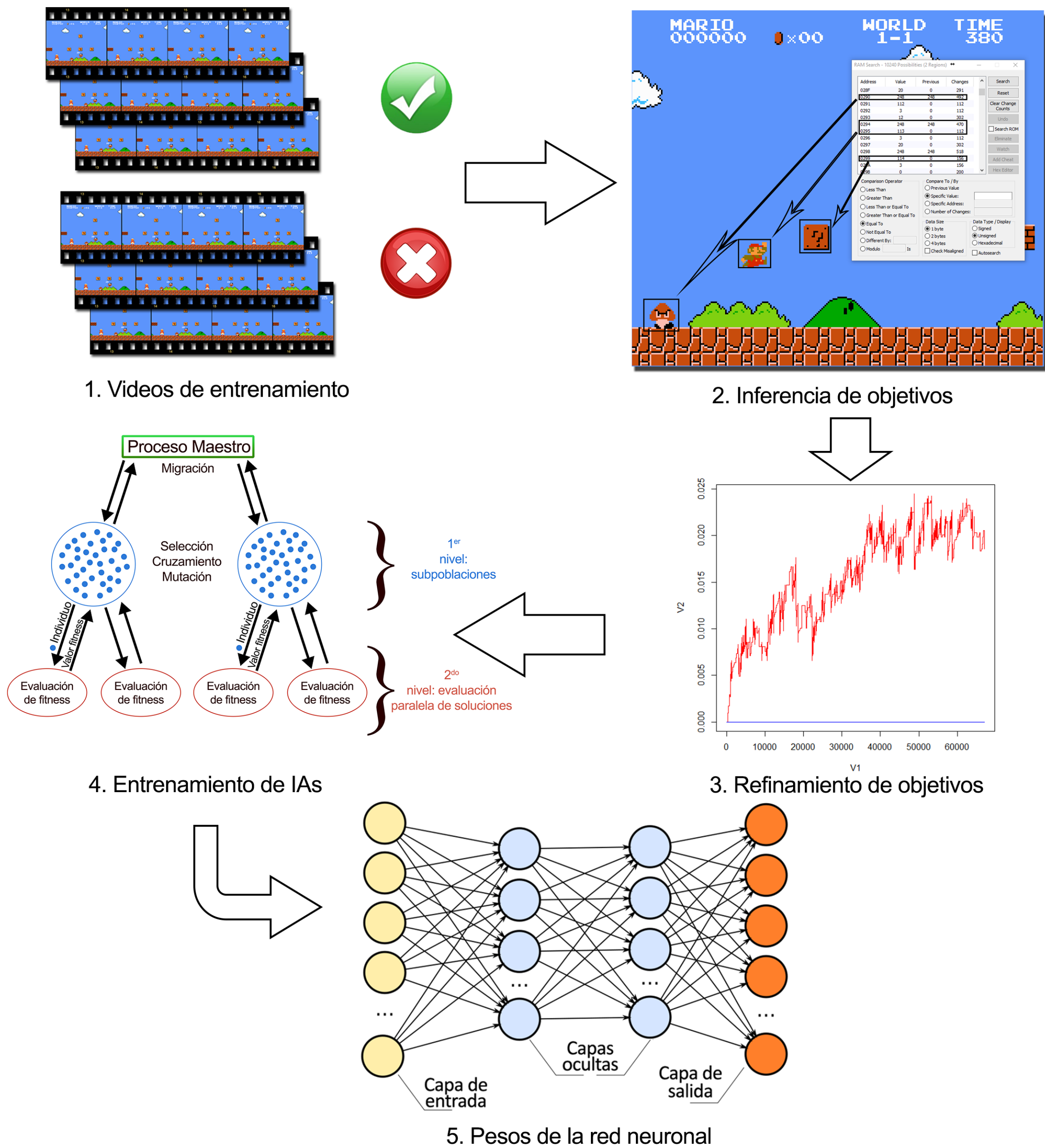


Figura 5.1: Componentes del pipeline de generación de IAs

5.2. Framework Evolutivo

Se desarrolló un framework evolutivo en C++ basándose en el diseño del framework Watchmaker (Dyer, 2006) escrito en Java. Este esfuerzo fue motivado por la necesidad de tener más control sobre la implementación realizada, pudiendo soportar tecnologías no convencionales, como la evaluación en paralelo de soluciones aplicando un modelo maestro-esclavo sobre un coprocesador Xeon Phi. El coprocesador Xeon Phi utiliza la arquitectura MIC (Many Integrated Core), y es una plataforma masivamente paralela presentada por Intel en el año 2010, orientada a la ejecución de trabajos con una alta capacidad de paralelismo.

El diagrama de clases del sistema implementado se muestra en la figura 5.2. La solución es modular y extensible, soportando varios paradigmas evolutivos y configuraciones diferentes. De los motores evolutivos incluidos, el motor *NEATEvolutionEngine* abstrae la lógica necesaria para la ejecución de un proceso neuroevolutivo basado en NEAT.

El framework desarrollado soporta procesos evolutivos secuenciales y paralelos. Para los procesos evolutivos paralelos se tienen dos variantes: algoritmos con evaluación paralela y algoritmos con subpoblaciones distribuidas. El framework soporta la combinación de ambos mecanismos de paralelismo para implementar un modelo híbrido que permita obtener un mejor rendimiento, resultando en la configuración mostrada en la figura 5.3. Se definen múltiples interfaces requeridas para especificar completamente un problema y la forma de resolverlo. Además, se proveen implementaciones genéricas de algunas interfaces, como es el caso del operador de selección estocástica universal.

A continuación se describen las interfaces definidas y las implementaciones adjuntas, en caso que existan.

EvolutionEngine: clase abstracta que provee las funcionalidades principales requeridas por cualquier motor evolutivo, como son retornar el mejor individuo del proceso (o de la generación final), configurar los parámetros básicos del proceso evolutivo, etc. Se proveen las implementaciones *SimpleEvolutionEngine* (algoritmo monohilo) e *IslandEvolutionEngine*. Este último soporta la ejecución paralela de múltiples motores del tipo EvolutionEngine, brindando soporte para diversas subpoblaciones, utilizando un modelo de migración sincrónico. SimpleEvolutionEngine fue extendido por las clases *ParallelEvolutionEngine* y *MicParallelEvolutionEngine*, las cuales proveen la funcionalidad de evaluación en paralelo de la función de fitness de los individuos. SimpleEvolutionEngine puede ejecutar en cualquier equipo mientras que MicParallelEvolutionEngine utiliza una técnica de offload para realizar la evaluación paralela de las soluciones sobre un coprocesador Xeon Phi. Por último, *NEATEvolutionEngine* es una extensión de ParallelEvolutionEngine que provee la lógica necesaria para ejecutar un proceso de neuroevolución utilizando la técnica NEAT. A diferencia del resto de los motores, NEATEvolutionEngine requiere un pipeline de operadores específico, operadores de cruzamiento y mutación especiales y una fábrica específica de individuos (NEATCandidateFactory) para garantizar la correcta generación y evolución de las redes neuronales, representadas por los genomas *NEATGenome*. Para cumplir con estos requisitos se proveen todas las implementaciones necesarias.

MigrationOperator: interfaz que define los métodos que debe ofrecer un operador de migración. Es una dependencia de IslandEvolutionEngine. La implementación *RingMigration* realiza una migración ordenando las subpoblaciones en una topología de anillo unidireccional.

TerminationCondition: interfaz utilizada para definir condiciones de parada de cualquier proceso evolutivo. Es una dependencia de EvolutionEngine. Se provee la implementación *GenerationCount*, que permite definir un criterio de parada basado en la cantidad de generaciones a evolucionar.

FinalStatistics: tipo estructurado que contiene estadísticas sobre la ejecución completa del proceso evolutivo. Es una dependencia de EvolutionEngine. Las estadísticas más relevantes incluyen: el tiempo transcurrido durante el proceso evolutivo, el fitness del mejor individuo, la primera generación donde se encontró el mejor individuo y la media y la desviación estándar del fitness de la última generación.

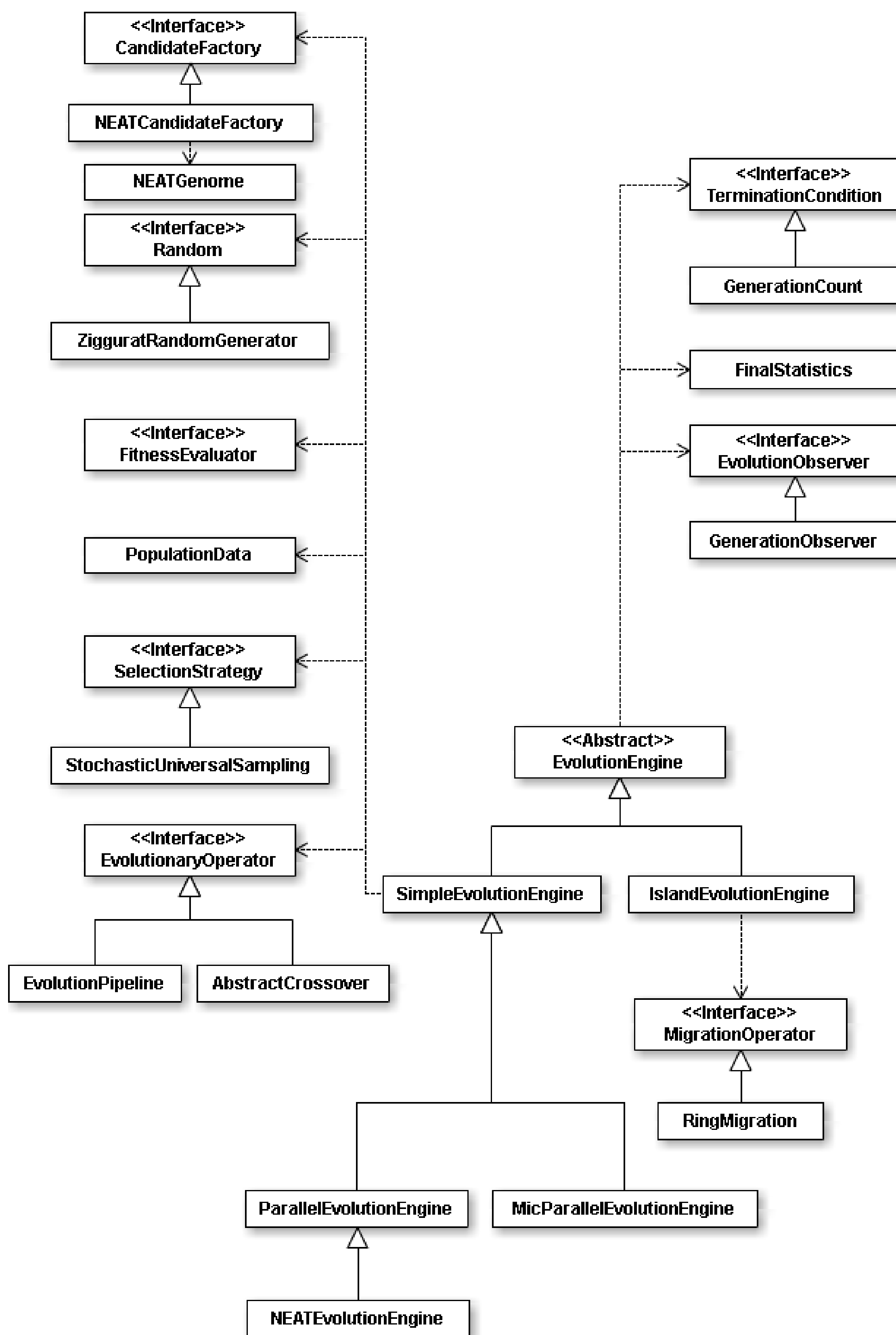


Figura 5.2: Diagrama de clases del framework implementado

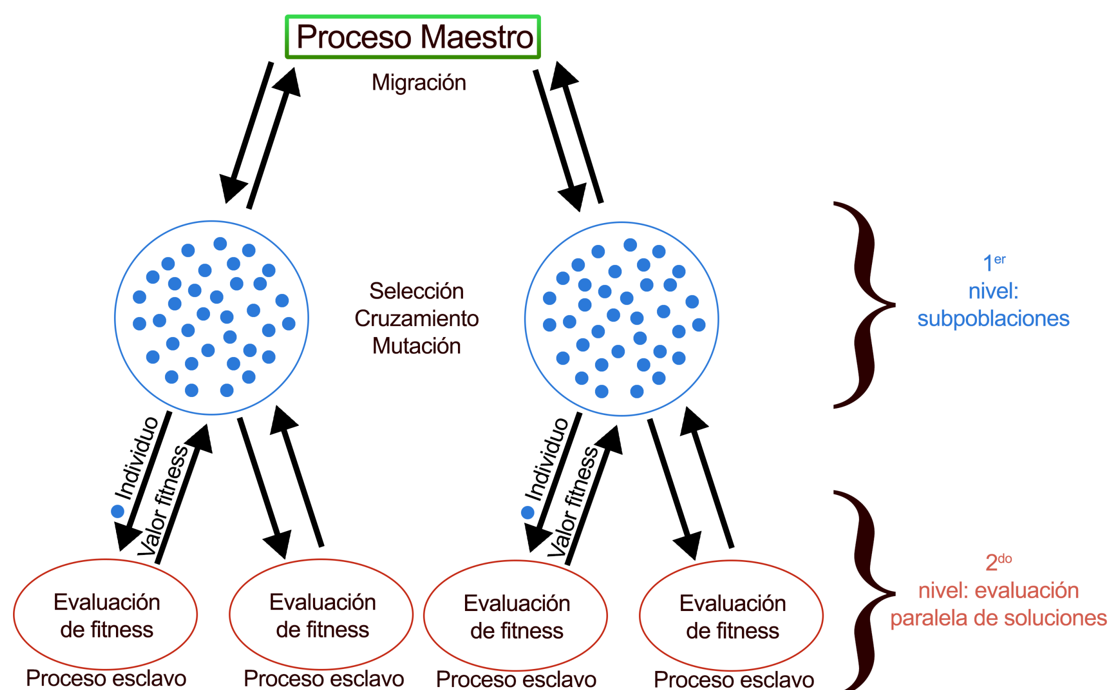


Figura 5.3: Algoritmo evolutivo paralelo en 2 niveles

EvolutionObserver: interfaz que permite definir un observador del proceso evolutivo, invocando al terminar el procesamiento de cada generación (o de cada *epoch* en el caso de utilizar un IslandEvolutionEngine). Es una dependencia de EvolutionEngine. Se provee la implementación *GenerationObserver*, que permite registrar las estadísticas más relevantes de cada generación.

CandidateFactory: interfaz que define el generador de la población inicial del proceso. Se provee la implementación *NEATCandidateFactory* necesaria para el proceso de neuroevolución. No se proveen otras implementaciones de esta interfaz dado que su definición se encuentra profundamente interrelacionada con la del problema a tratar. Esta interfaz debe implementarse de forma específica para reflejar las restricciones y requerimientos de cada problema particular.

Random: interfaz que define los servicios provistos por el generador de números aleatorios. Es una dependencia de SimpleEvolutionEngine. Se provee la implementación *ZigguratRandomGenerator*, que utiliza el método Ziggurat para el cálculo de números aleatorios con distribución gaussiana a partir de un generador de números aleatorios uniformes (Marsaglia y Tsang, 2000). El generador de números uniformes incluido en la implementación original del método Ziggurat desarrollada por Marsaglia presenta algunos problemas (Leong et al., 2005), por lo que fue reemplazado por el generador XorShift128+ (Vigna, 2017), para mejorar la calidad de los números generados.

FitnessEvaluator: interfaz que define la función de fitness a evaluar. Es una dependencia de SimpleEvolutionEngine. No se provee una implementación dado que dicha función es específica de cada problema tratado.

PopulationData: estructurado que contiene toda la información referente a la generación actual. Es una dependencia de SimpleEvolutionEngine. Las estadísticas más relevantes incluyen: el tiempo transcurrido durante el proceso evolutivo, el fitness del mejor individuo, la primera generación donde se encontró el mejor individuo y la media y la desviación estándar del fitness de la última generación.

SelectionStrategy: interfaz que define un operador de selección. Es una dependencia de SimpleEvolutionEngine. Se provee la implementación *StochasticUniversalSampling* correspondiente a la selección estocástica universal, presentada en la sección 3.1.5.

EvolutionaryOperator: interfaz que define los métodos de un operador evolutivo. Es una dependencia de SimpleEvolutionEngine. Se proveen las implementaciones *EvolutionPipeline* y *AbstractCrossover*. El primero es un pseudo-operador que permite definir un pipeline de operadores a ser aplicados en cadena. El segundo es una clase abstracta que define el esqueleto de un operador de cruzamiento. La implementación específica de cada operador de cruzamiento debe ser provista de forma separada para cada problema particular, porque requiere conocimiento de la estructura de los individuos. No existe una interfaz para un operador de mutación debido a que coincide con la interfaz provista por EvolutionaryOperator. Al igual que en el caso del operador de cruzamiento, no se provee ninguna implementación por la fuerte dependencia del operador de mutación con la estructura específica de los individuos para el problema estudiado.

5.2.1. Optimización del desempeño computacional del framework evolutivo

El objetivo fundamental de diseño del framework es lograr el mejor rendimiento computacional posible para permitir la ejecución de procesos evolutivos de tamaño considerable en un tiempo razonable. Teniendo en cuenta el alto costo del proceso evolutivo involucrado en el proyecto (mayoritariamente determinado por la evaluación de individuos), se realizó una optimización agresiva del framework para reducir al mínimo el consumo de recursos.

Se comenzó por minimizar el uso de memoria utilizando punteros a los individuos en todas las colecciones mantenidas durante el proceso evolutivo. Se tuvo especial cuidado de no generar copias de individuos de forma innecesaria y de eliminar los individuos descartados en el primer momento que fuera posible hacerlo de forma segura. También se empleó la biblioteca de programación OpenMP (OpenMP Architecture Review Board, 2008) en las secciones paralelizables del código, con el objetivo de aprovechar todos los recursos computacionales disponibles en equipos con múltiples núcleos. Todas las islas de un IslandEvolutionEngine son ejecutadas en paralelo, al igual que todas las evaluaciones de individuos de un ParallelEvolutionEngine.

Se realizó un gran esfuerzo para mantener un balance entre el tamaño del código (para minimizar los fallos del caché de instrucciones) y la cantidad de llamadas a funciones (para evitar cambios de contexto). En los casos donde el uso de funciones mejoraba la legibilidad del código pero la cantidad de llamadas era muy baja, se utilizaron declaraciones *inline*. Siguiendo el mismo enfoque, se evitó el uso de funciones accesoras y modificadoras, dando preferencia al uso de tipos estructurados y de variables con modos de acceso protegido y público. De esta forma se eliminaron los cambios de contexto asociados a dichas funciones. Además, se explotó el uso de funciones de bajo nivel, como *memcpy*, para disminuir el gasto extra de recursos generado por las operaciones más complejas. Se utilizaron todas las optimizaciones posibles que pudieran conducir a un aumento del desempeño en la plataforma objetivo (banderas O2 y O3 y la opción march=corei7 del compilador gcc). Finalmente, se utilizó Valgrind (Valgrind Developers, 2016) con el fin de detectar y corregir errores de manejo de memoria, mejorando la estabilidad y reduciendo el consumo de recursos.

Todas estas medidas influyen positivamente en el desempeño del framework evolutivo. El cuello de botella de todo el proceso recae en la evaluación de la IA que es entrenada por medio del emulador de NES.

5.2.2. Ejemplos relevantes de código paralelizado

En los listados 5.1 y 5.2 se presenta el código de las secciones paralelizadas más relevantes del framework evolutivo. Se detalla el manejo de subpoblaciones distribuidas y la evaluación concurrente de individuos.

Subpoblaciones distribuidas

El fragmento de código presentado en el listado 5.1 es el encargado de ejecutar el proceso evolutivo con subpoblaciones distribuidas. Este fragmento es ejecutado dentro de un bucle para ejecutar cada epoch hasta que una de las condiciones de parada se cumpla. Se utilizan directivas `pragma omp parallel` y `pragma omp for` para paralelizar la evaluación de las distintas subpoblaciones. En la línea 5 se inicia el proceso evolutivo de cada isla y se almacenan los resultados para un epoch. Al paralelizar el `for` se ejecuta cada isla en paralelo de forma automática, aprovechando las facilidades de OpenMP. Una vez finalizada la ejecución del epoch, las líneas 9 y 12 se encargan de eliminar información referente a los individuos del epoch anterior.

```
1  #pragma omp parallel shared(results, engines, seeds, tc)
2  {
3    #pragma omp for
4    for(unsigned int i=0; i<engines.size(); i++){
5      results[i] = engines[i]->evolvePopulation(seeds[i],tc[i]);
6    }
7  }
8
9  islandsPopulations.clear();
10 for(unsigned int j = 0; j<islands.size();j++){
11   islandsPopulations.push_back(results[j]->finalPopulation);
12   results[j]->finalPopulation.clear();
13 }
14
15 migration->migrate(islandsPopulations , migrationCount, rng);
16 mergedPopulation.clear();
17 for(unsigned int i=0; i<islandsPopulations.size();i++){
18   mergedPopulation.insert(mergedPopulation.end(),islandsPopulations[i
       ].begin(), islandsPopulations[i].end());
19 }
20 sortEvaluatedPopulation(mergedPopulation,this->isNatural);
21 this->getPopulationData(mergedPopulation,this->isNatural,0,
     currentGenerationIndex,startTime);
22 this->notifyPopulationChange(this->popData);
23 satisfiedConditions = shouldContinue(this->popData,
     terminationConditions);
```

Listado 5.1: Código para el manejo de subpoblaciones distribuidas

El bucle en las líneas 10–13 genera un vector que contiene los individuos de la generación actual de todas las subpoblaciones para realizar la migración, que se ejecuta en la línea 15. El bucle en las líneas 17–19 se encarga de dividir el vector que contiene a todos los individuos luego de realizar la migración en las islas correspondientes. La función sortEvaluatedPopulation (línea 20) realiza un ordenamiento descendente de la población de acuerdo a los valores de fitness, para facilitar el trabajo de la función getPopulationData (línea 21), que se encarga de generar la información estadística que provee el framework. La información estadística es enviada a todos los escuchas del proceso evolutivo por medio de la función notifyPopulation (línea 22). Finalmente, la función shouldContinue (línea 23) se encarga de verificar las condiciones de parada del proceso evolutivo y retorna el conjunto de condiciones satisfechas.

Evaluación paralela

El fragmento de código presentado en el listado 5.2 ejecuta la evaluación paralela de los individuos de la población en cada generación. La función *omp_set_num_threads*, en la línea 3, configura la cantidad de hilos a utilizar por OpenMP. El uso previo de la función *omp_set_dynamic(0)* en la línea 2 deshabilita el manejo de grupos dinámicos, otorgando un mayor control al especificar un número arbitrario de hilos a ejecutar. La directiva *pragma omp parallel*, en la línea 4, inicia un ambiente paralelo y define las variables a compartir, mientras que en la línea 8 se define el bucle paralelizado que realiza la evaluación utilizando OpenMP. El individuo a evaluar se almacena en un tipo estructurado EvaluatedCandidate que lo asocia al fitness obtenido. El fitness del individuo se calcula en la línea 11. La evaluación se ejecuta por medio de una llamada a la función *getFitness* del FitnessEvaluator implementado por el usuario del framework para el problema en cuestión. El constructor de vector en la línea 15 almacena los resultados en un vector, para cumplir con el tipo de retorno de la función. Finalmente, se realiza una limpieza del arreglo temporal y se retornan los individuos evaluados en las líneas 16–17.

```
1   EvaluatedCandidate<T>** resultArray = new
        EvaluatedCandidate<T>*[population.size()];
2   omp_set_dynamic(0);
3   omp_set_num_threads(this->numThreads);
4   #pragma omp parallel shared(resultArray, population)
5   {
6       EvaluatedCandidate<T>* ec;
7       #pragma omp for
8       for (unsigned int i = 0; i< population.size(); i++){
9           ec = new EvaluatedCandidate<T>;
10          ec->candidate = population[i];
11          ec->fitness = this->fitnessEvaluator->getFitness(
                i, population);
12          resultArray[i]=ec;
13      }
14  }
15  std::vector<EvaluatedCandidate<T>*> evaluatedPopulation(
        resultArray, resultArray + population.size());
16  delete[] resultArray;
17  return evaluatedPopulation;
```

Listado 5.2: Código para evaluación paralela

5.3. Emulador FCEUX

Para ejecutar las simulaciones se utilizó el emulador de código abierto de NES FCEUX (FCEUX Community). Este emulador está escrito en C++ y provee soporte para las distintas plaquetas, mandos y accesorios de la NES con el fin de poder ejecutar cualquier juego de la plataforma. Con el fin de utilizar el emulador FCEUX en este proyecto se realizaron varias modificaciones explicadas a continuación. El objetivo de estas modificaciones consistió en eliminar componentes innecesarios de la implementación para disminuir el tiempo requerido para emular un juego.

Se comenzó por implementar métodos para extraer la información del juego y permitir la interacción entre la IA y el emulador. Implementar estos métodos requirió modificar la rutina que obtiene la entrada del usuario para consultar a la biblioteca de IA, enviando como parámetro el frame de RAM existente en el momento que se procesa la entrada, según el modelo presentado en la sección 2.1. LA IA devuelve una combinación de teclas a presionar según los valores que se encuentran en la RAM emulada del NES. También se implementó una funcionalidad para definir un límite de frames a ejecutar para utilizar como criterio de parada uniforme para todos los jugadores.

Se eliminaron todas las dependencias superfluas del emulador. Se aprovechó la configuración paramétrica del sistema de compilación automática (*scons*) (The SCons Fundation, 2016) para minimizar la cantidad de dependencias de bibliotecas opcionales. Luego se modificó sustancialmente el código fuente para retirar todas las dependencias a la biblioteca SDL. Estas modificaciones permitieron suprimir las capacidades de desplegar en pantalla y de obtener entradas, lo que permite lograr una simulación sin límite de frames por segundo. Eliminar el límite de frames permite ejecutar la simulación sin destinar ciclos de CPU a enviar el proceso a dormir entre el renderizado de un frame y el siguiente, aumentando la velocidad con la que se realiza la simulación. También se suprimió el soporte para juegos comprimidos, con el fin de eliminar la dependencia con la librería ZLib.

Se eliminaron muchas funciones internas superfluas, lo que permitió disminuir el tamaño del binario y mejoró el rendimiento al eliminar llamadas a funciones que no son útiles para realizar la emulación propiamente dicha. En particular, se eliminó el soporte para grabar y reproducir secuencias de teclas, además de otras herramientas provistas por FCEUX con el fin de permitir desarrollar *Carreras Asistidas por Herramientas* (Tool Assisted Speedruns, TAS). Una TAS es una partida jugada por un jugador humano donde por medio de herramientas informáticas (como son emuladores especializados, depuradores, ensambladores de código e inspectores de memoria, entre otros) se explotan errores del juego para completar una partida en la menor cantidad de tiempo posible. Dichos errores son detectados por el jugador humano al analizar frame a frame la memoria del juego y el flujo de ejecución del mismo en varias partidas. De la misma forma, se retiró el soporte para juegos en red (que permite jugar a través de una red local o de Internet con otra persona) y se eliminó el código del *Assembler/Disassembler* integrado para X6502 (modelo de CPU de la NES). También se eliminó el soporte para el inspector de memoria. Además, fueron retiradas rutinas para almacenar una bitácora de eventos y realizar depuración, tanto de la CPU como del PPU (Picture Processing Unit). Se removió el soporte para auto-disparo de las rutinas de procesamiento de entrada, que no es utilizado por la IA pero consume un tiempo considerable de CPU. También se quitó el soporte para capturas de pantalla y guardar estados del juego.

Se suprimieron las rutinas de simulación de la PPU, manteniendo solamente las rutinas mínimas necesarias para interactuar con la CPU emulada y mantener la sincronización esperada por la consola. También se removió la emulación de audio, retirando las rutinas pertinentes en todas las plaquetas emuladas que lo requirieron, para reducir la carga de la simulación. La razón por la que se modificaron las plaquetas emuladas es que cada una simula los componentes presentes en el cartucho original que contenía el juego y algunos cartuchos contenían un circuito integrado para el procesamiento de audio.

Finalmente, en base a los resultados obtenidos en un análisis de eficiencia computacional, se retiraron varias funciones que generaban un alto consumo de recursos. Estas funciones no realizaban tareas útiles para la emulación, siendo varias de ellas funciones auxiliares para componentes removidos previamente. La supresión de estas funciones mejoró el desempeño del emulador, obteniendo una mejora de velocidad de 22,5 veces en comparación a una partida en tiempo real.

La implementación optimizada del emulador se utiliza como versión por línea de comandos para poder calcular el valor de fitness de cada individuo mientras se ejecuta el proceso evolutivo. Para lograr visualizar los resultados finales de la evolución se modificó el emulador original para permitir la utilización del módulo de IA. No se realizaron otras modificaciones de esta versión del emulador, por lo que la versión modificada posee todas las herramientas del emulador original. Se mantuvieron dos versiones del emulador completo, una para Linux y otra para Windows. La versión de FCEUX para Windows posee más herramientas de análisis y de usabilidad y mejores gráficos que la versión de Linux.

5.4. Inferencia de objetivos

La primera etapa del pipeline de aprendizaje se compone de un analizador de la RAM del emulador que captura el contenido de la RAM al reproducir los videos de entrenamiento. Esta etapa permite inferir direcciones de RAM cuyo contenido es superfluo para el juego en cuestión utilizando varias heurísticas simples. Las técnicas empleadas descartan una cantidad considerable de la RAM (hasta el 75 % para algunos juegos) y permiten reducir el espacio de búsqueda. El resultado es un aumento de velocidad de las restantes etapas del pipeline con poco esfuerzo computacional, debido a que las heurísticas empleadas requieren poca memoria y utilizan poco tiempo de cómputo. Además, el analizador infiere el prefijo necesario para iniciar una partida, lo que permite ignorar los valores espurios de la RAM al principio de cada video. Emplear el prefijo agiliza los procesos evolutivos de los pasos siguientes, al reducir la cantidad de RAM a considerar para la etapa de refinamiento y evitar tener que aprender a manejar los menús en la etapa de aprendizaje. Finalmente, el analizador es capaz de detectar órdenes lexicográficos que son utilizados para inferir los objetivos del juego. Estas funcionalidades proporcionan un mecanismo para evitar el impacto de los inconvenientes presentados en la sección 2.3. Todas las técnicas son aplicadas de forma incremental, de forma tal que los resultados de cada paso están disponibles para el siguiente.

Los pasos ejecutados en la inferencia de objetivos, en orden, son los siguientes: detección de prefijo mínimo, detección de valores estáticos en RAM, detección de valores erráticos, detección de valores con bajo dinamismo, detección de órdenes lexicográficos, detección de valores independientes e inferencia de pesos candidatos. Las técnicas utilizadas se detallan a continuación.

5.4.1. Detección de prefijo mínimo

Para poder comenzar una partida, el algoritmo evolutivo de la última etapa del pipeline debe evolucionar redes que sepan navegar los menús del juego para lograr empezar a jugar. Las redes neuronales pueden aprender a navegar por los menús, como fue constatado en las pruebas iniciales realizadas con el juego Pinball. Sin embargo, se deben dedicar las primeras generaciones del algoritmo evolutivo para aprender este comportamiento, así como un conjunto de neuronas y enlaces para codificarlo, lo que disminuye la cantidad de generaciones y componentes de la red neuronal dedicados a aprender el juego en sí. Sin perder generalidad, es posible inferir la secuencia de teclas que presionan todos los jugadores humanos en los videos de entrenamiento para comenzar una partida. Utilizando esta secuencia de teclas se puede comenzar una partida sin utilizar la red neuronal y así dedicar el aprendizaje por completo al juego propiamente dicho. Este paso resuelve el inconveniente detallado en la sección 2.3.1 por medio de un análisis de prefijo de todos los videos de entrenamiento, buscando una combinación que permite iniciar la partida.

La deducción del prefijo permite dedicar el aprendizaje por completo al juego desde la primera generación del algoritmo evolutivo. Además, permite mejorar la calidad de los resultados de la etapa de inferencia del pipeline. Utilizando el prefijo se recortan todos los videos eliminando los frames correspondientes al prefijo hallado, con lo que se logra tener todas las partidas en un mismo estado inicial (comenzando una partida nueva). Dado que todos los videos recortados comienzan en el mismo estado, es posible comparar directamente los estados de cada video, permitiendo realizar la detección de valores independientes explicada en la sección 5.4.6.

5.4.2. Detección de valores estáticos

La detección de valores estáticos se realiza implementando el análisis de varianza propuesto en la sección 2.3.2. Los valores que cumplen este criterio son descartados, debido a que no aportan a la semántica del juego (la mayor parte de ellos corresponden a lugares de memoria no utilizada).

5.4.3. Detección de valores erráticos

Para eliminar valores erráticos se implementa el criterio de análisis de varianza definido en la sección 2.3.2. De cumplir con el requisito especificado, el valor se descarta. La idea detrás de este razonamiento es que valores erráticos tampoco influyen en los objetivos, dado que la mayor parte de las veces representan datos específicos del motor de juego, como punteros a música, números aleatorios, contadores, entre otros.

5.4.4. Detección de valores con bajo dinamismo

Para eliminar valores con bajo dinamismo se realiza un análisis estadístico del histórico de cada lugar de RAM como fue propuesto en la sección 2.3.2. De cumplirse que una dirección de RAM tenga el mismo valor por lo menos el 95 % del tiempo, dicha dirección se descarta. El fundamento de este criterio es que si la dirección de RAM estudiada se encuentra la mayor parte del tiempo con un solo valor, es razonable esperar que esa dirección no forme parte del objetivo del juego, dado que el objetivo progresa a lo largo de la partida.

5.4.5. Detección de órdenes lexicográficos

Este paso del analizador soluciona el problema de detección de órdenes lexicográficos planteado en la sección 2.3.3. Sin embargo, este problema presenta una sutileza: los órdenes lexicográficos pueden estar formados por posiciones de RAM contiguas o por posiciones discontinuas. El puntaje en el juego Super Mario Brothers es un ejemplo de orden lexicográfico contiguo ya que está formado por 6 direcciones de RAM adyacentes en el rango [0x7DE. . . 0x7E2]. En cambio, el orden compuesto por el mundo, nivel, pantalla y posición x del personaje es un ejemplo de orden discontinuo ya que está formado por las direcciones 0x75F, 0x760, 0x6D y 0x86 respectivamente. Para afrontar ambos casos es necesario aplicar dos técnicas diferentes para buscar direcciones de RAM que integren órdenes.

Primero se realiza una búsqueda de órdenes lexicográficos cuyos valores se encuentren contiguos en RAM. Se utilizan cuatro variantes de este algoritmo, para buscar órdenes monótonamente crecientes y decrecientes, tanto para la representación big endian como para la little endian. La falta de una convención para la representación de valores numéricos de los juegos de NES hace necesario considerar ambos tipos. Existen incluso ejemplos de juegos que presentan ambos tipos de representación simultáneamente.

A continuación se lleva a cabo una búsqueda aleatoria sobre los valores restantes de RAM con el fin de encontrar órdenes lexicográficos que no estén contiguos en memoria. Esta técnica permite detectar órdenes como el formado por el mundo, el nivel, la pantalla y la posición x del personaje en Super Mario Brothers. El método de búsqueda aleatorio no se ve afectando por el endianess de los valores numéricos, pudiendo encontrar órdenes sobre valores codificados en cualquiera de ambas representaciones (little o big endian). Sin embargo, debe diferenciarse el tipo de orden buscado, por lo que es necesario ejecutar una búsqueda para órdenes crecientes y otra para decrecientes.

Finalmente, para captar el concepto de que algunos órdenes son temporalmente violados durante el transcurso de una partida, la búsqueda de órdenes aleatorios se realiza también sobre subdivisiones del video de entrenamiento (cada 100, 250 y 1000 frames). Esta búsqueda es importante en juegos como Super Mario Brothers o Contra, en los que es común volver hacia atrás en los niveles para realizar acciones como obtener un potenciador o matar a un enemigo. Volver hacia atrás viola temporalmente la monotonía del orden lexicográfico compuesto por la posición del jugador y la pantalla, pero la violación se da solo en un intervalo muy corto de tiempo, por lo que puede ser detectada al analizar subdivisiones de la RAM del juego. Detectar estos órdenes permite considerar valores con tendencias crecientes o decrecientes, que están generalmente relacionados con la posición del jugador dentro de un nivel.

5.4.6. Detección de valores independientes

Este paso resuelve el problema de encontrar variables independientes al juego, como fue presentado en la sección 2.3.2. Los valores que cumplen con la condición especificada son enmascarados ya que no dependen de las acciones realizadas por el jugador en los diferentes videos, lo que sugiere que no codifican parte del estado del juego.

El orden en que se ejecuta este paso en la cadena de tareas realizada por el analizador es importante, debido a que algunos órdenes lexicográficos tienen comportamientos similares o idénticos en videos suficientemente cortos, por lo que pueden ser descartados por este criterio. Por ejemplo, al considerar el mundo del Super Mario Brothers, dos partidas

que no logren avanzar al quinto nivel (mundo 2, nivel 1) descartarán la variable que codifica al mundo, por no haber variado en toda la partida. Para evitar esta situación, se detectan primero los órdenes presentes en la memoria y luego se ejecuta la detección sobre los valores que no pertenecen a ningún orden.

5.4.7. Inferencia de pesos candidatos

Para llevar a cabo la inferencia de objetivos se realiza una inferencia de pesos tentativos, siguiendo el procedimiento planteado en la sección 2.3.3. Se realizó la inferencia de pesos para todas las posiciones no enmascaradas de RAM. Sin embargo, esta estrategia produjo funciones de fitness con ruido, que no evaluaban correctamente las partidas. Mediante estudios informales se encontró que considerar solamente los órdenes lexicográficos como componentes de la función objetivo mejoraba la calidad de la evaluación. Por este motivo se decidió aplicar un procedimiento de inferencia que genera pesos solamente para los órdenes hallados previamente.

Además, al momento de generar los pesos es posible que el valor inferido para un peso particular se encuentre fuera del rango válido para cada peso. Por este motivo se implementó una técnica de escalado para mantener los pesos acotados. Inicialmente se hizo un escalado global, dividiendo el valor de todos los pesos por aquel de mayor valor, escalando todos los valores al rango $[-1 \ldots 1]$. Sin embargo, debido a que los pesos presentan diferencias sustanciales de magnitud entre sí (como es el caso de los pesos relacionados con el puntaje y con la cantidad de vidas), la técnica descrita asignaba valores cercanos a cero a la mayor parte de los pesos. Para solucionar este problema se empleó en cambio una técnica de escalado local, calculando la tangente hiperbólica del peso inferido y tomando el valor resultante como el peso escalado. De esta forma se logran obtener valores en el mismo rango $[-1 \ldots 1]$, pero se evita descartar pesos de pequeña magnitud.

El efecto de ambas técnicas de escalado puede verse en las figuras 5.4, 5.5, 5.6 y 5.7. La línea azul representa la valoración de la función objetivo inferida sobre los videos malos, mientras que la línea roja representa la valoración de la función objetivo inferida sobre los videos buenos. La figura 5.4 muestra que al dividir por el valor máximo la función objetivo obtiene valores muy cercanos a 0 y, en este caso particular, valora de mejor forma a los videos malos en comparación con los buenos, indicando que la inferencia fue incorrecta. Los resultados obtenidos para el resto de los juegos muestran valores muy pequeños de la función objetivo, dado que casi todos los pesos valen 0 al utilizar la técnica de escalado que divide por el máximo. Este comportamiento dificulta la discriminación de las partidas buenas y malas. Un ejemplo notorio de esta cercanía son los valores obtenidos por el juego Contra, que se presentan en la figura 5.6. Si bien la gráfica de la función objetivo es creciente, la diferencia más grande entre la valoración de los videos buenos y malos es de 0,007, un resultado que dificulta la comparación entre ambos. Al utilizar la tangente hiperbólica los resultados son distinguibles, como se presenta en la figura 5.5 para el Battle City. La diferencia entre las valoraciones de los videos positivos y negativos es de dos órdenes de magnitud (mayor) en comparación a los resultados de la técnica que emplea el máximo. Este resultado también se observa en los resultados presentados en la figura 5.7 para el juego Contra, donde la diferencia es de tres órdenes de magnitud (mayor) que la obtenida al utilizar el máximo. Esta diferencia permite distinguir exitosamente ambos tipos de partidas, por lo que se seleccionó para ser utilizada en este proyecto.

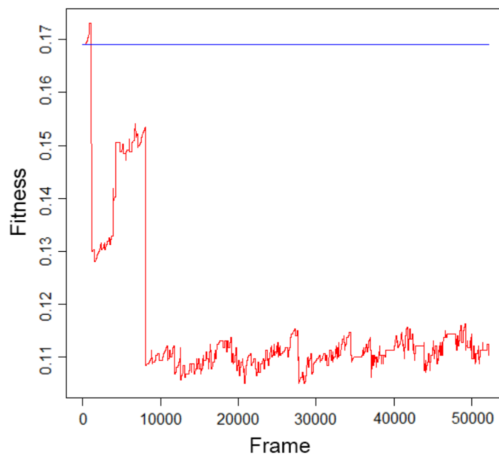


Figura 5.4: Objetivos escalados según el máximo para Battle City

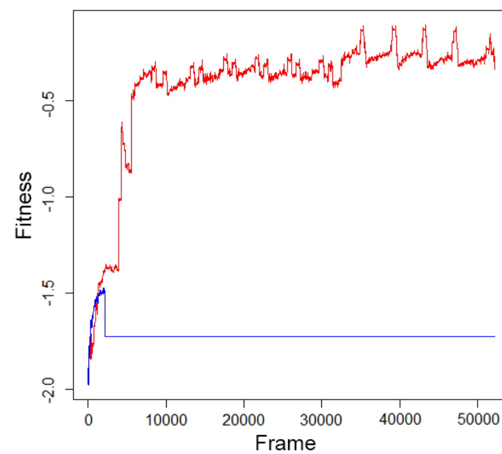


Figura 5.5: Objetivos escalados según la tangente hiperbólica para Battle City

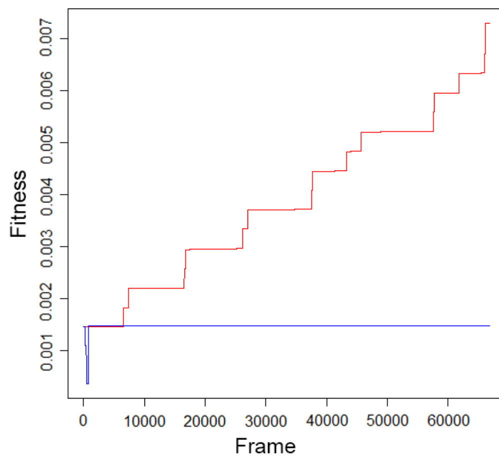


Figura 5.6: Objetivos escalados según el máximo para Contra

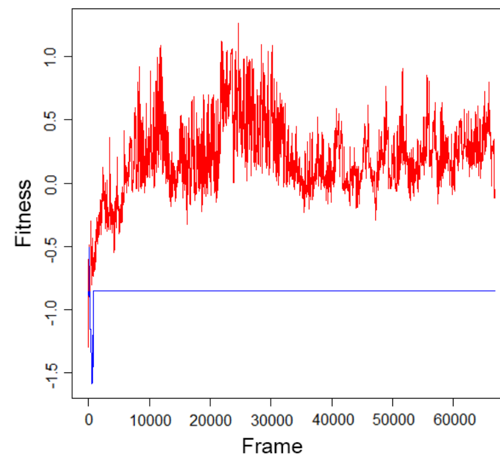


Figura 5.7: Objetivos escalados según la tangente hiperbólica para Contra

5.5. Algoritmo evolutivo para el refinamiento de objetivos

Los objetivos obtenidos en la primera etapa del pipeline pueden ser mejorados para obtener una función que describa de manera más ajustada el progreso del jugador a lo largo de la partida. Para resolver el problema de optimización relacionado se propuso aplicar un algoritmo evolutivo que toma como entrada los videos de entrenamiento, la máscara de RAM y los pesos tentativos producidos en la etapa anterior. A partir de estos datos de entrada se siguen los pasos planteados en la sección 2.4, cuyo resultado es una función de fitness mejorada para la última etapa del pipeline.

El uso de una combinación lineal de los objetivos se prefiere a un MOEA (algoritmo evolutivo multiobjetivo) explícito debido a la alta dimensionalidad del espacio de soluciones. Una dimensionalidad alta presenta desafíos de escalabilidad para los MOEAs más populares, tanto en el desempeño computacional como en la uniformidad del muestreo del frente de Pareto (Khare et al., 2003). La inferencia de objetivos permite detectar por lo menos 20 objetivos, identificados en el juego que presenta menos órdenes lexicográficos de los ocho juegos considerados.

5.5.1. Operadores evolutivos

El operador de selección utilizado es la selección estocástica universal (Baker, 1987). Se utiliza un operador de cruzamiento de un punto y la mutación se implementa como la modificación por medio de una variable aleatoria gaussiana de los pesos que componen al individuo. Además, se emplea elitismo para preservar al mejor individuo de cada subpoblación.

La generación de la población inicial se realiza de la siguiente manera: a partir del individuo conformado por los pesos obtenidos en la primera etapa del pipeline se generan 15 mutantes modificando cinco pesos elegidos aleatoriamente. Estos pesos son modificados utilizando la fórmula $w_{nuevo} = w * (1 + G)$, donde G es una variable aleatoria con distribución gaussiana de media 0 y varianza 0.01. El conjunto de individuos formado por los 15 mutantes y el individuo sin mutar son distribuidos de forma equitativa entre las 4 subpoblaciones, asignando 4 individuos por subpoblación. El resto de la población se completa generando individuos con pesos aleatorios.

5.5.2. Configuración paramétrica

El algoritmo implementado utiliza las técnicas de evaluación paralela y subpoblaciones distribuidas para obtener un mejor desempeño y mantener una mayor diversidad. Se utiliza una configuración como la del modelo de paralelismo presentado en la figura 5.3, con una población de 256 individuos distribuidos en 4 subpoblaciones (64 individuos por subpoblación). La migración entre subpoblaciones se realiza cada 50 generaciones, utilizando una topología de anillo unidireccional. Se migran 5 individuos en cada epoch, seleccionados de forma aleatoria. Se ejecutan 40 epochs, para un total de 2000 generaciones. Estos parámetros fueron fijados por medio de análisis que mostraron que la configuración elegida obtiene resultados satisfactorios para los juegos estudiados.

5.6. Algoritmo Evolutivo para el entrenamiento de inteligencias artificiales

La última etapa del pipeline genera las IAs buscadas por medio de un algoritmo evolutivo para entrenar redes neuronales recurrentes, que implementan los jugadores automáticos resultantes de todo el proceso de aprendizaje. La etapa toma como entrada la función objetivo inferida en la etapa anterior y la máscara de RAM. Se mantiene un conjunto de redes neuronales que son evaluadas con la función objetivo presentada en la sección 2.2. La evaluación se realiza en base a la memoria asociada a una partida jugada por la red que es evaluada, utilizando el emulador FCEUX. La salida de esta etapa es la mejor red neuronal encontrada según la función objetivo inferida. De esta forma se resuelve el problema presentado en la sección 2.5.

Para esta etapa se utiliza una estrategia de neuroevolución basada en el modelo NEAT explicado en la sección 3.4.2. Se fija el conjunto de neuronas de entrada de la red para coincidir con los valores no enmascarados de RAM. Independientemente del juego en cuestión, todas las redes tienen ocho neuronas de salida, cada una asociada con uno de los botones disponibles en el mando del NES. La activación de una neurona de salida representa la acción de presionar el botón asociado.

5.6.1. Operadores evolutivos

La generación de la población inicial se realiza siguiendo el proceso presentado en la sección 3.4.2. Se aplica selección estocástica universal para la selección interespecie y para la selección intraespecie. El cruzamiento y la mutación son las provistas por NEAT, presentadas en las secciones 3.4.2 y 3.4.2. Además, se utiliza elitismo para preservar al mejor individuo del proceso evolutivo.

5.6.2. Configuración paramétrica

El algoritmo implementado utiliza la técnica de evaluación paralela para obtener un mejor desempeño computacional. No es posible utilizar subpoblaciones distribuidas debido a la existencia de un repositorio centralizado de números de innovación (base de datos de innovación). Realizar consultas concurrentes a dicha base requeriría ejecutar operaciones de bloqueo que reducirían la mejora de desempeño obtenida por el uso de subpoblaciones. En el modelo de evolución propuesto por NEAT la mejora de diversidad es aportada por el uso de especies.

El proceso evolutivo mantiene una población de 64 individuos que es evolucionada durante 800 generaciones.

5.6.3. Variaciones sobre el modelo NEAT

La solución presentada se basa en el diseño original de NEAT propuesto por Stanley y Miikkulainen (2002). Sin embargo, se decidió modificar algunos de los operadores y extender las capacidades del algoritmo con la intención de mejorar el rendimiento para esta tarea particular. Los cambios realizados se presentan a continuación.

En primer lugar, se permitió al operador de mutación cambiar las funciones de activación de las redes neuronales durante el proceso evolutivo. El objetivo de este cambio es que la red pueda modificar la representación de los conceptos aprendidos a medida que observa más elementos del juego (dado que la habilidad de la IA resultante aumenta y logra progresar más en los niveles). La evaluación realizada sobre prototipos iniciales del algoritmo mostró que emplear diferentes funciones de activación permite una mayor expresividad de las redes, lo que conlleva una mejora en la calidad de los jugadores resultantes.

Se agregaron más tipos de funciones de activación, lo que permite a las redes neuronales representar de forma simple más conceptos. El resultado es una mejora del desempeño de los jugadores resultantes al entrenarse en juegos de diversos géneros, que requieren habilidades diferentes para jugar de forma competente. Las funciones consideradas son presentadas en la sección 5.7.3.

Finalmente, se agregó soporte para elitismo para asegurar la supervivencia de los mejores individuos de la población global. Si bien NEAT introduce nociones de elitismo en el uso de especies y compartición de fitness explícita, evaluaciones preliminares permitieron constatar que la pérdida del mejor individuo de la población se produce con frecuencia. Utilizar elitismo de forma explícita permitió mejorar los resultados al asegurar la supervivencia del mejor individuo de toda la población.

5.6.4. Evaluación de la función de fitness

La evaluación de la función de fitness no es realizada por el proceso maestro, debido a que no posee la información necesaria para calcular el fitness de cada individuo. En cambio, cada red es serializada por un proceso esclavo y enviada al emulador FCEUX por línea de comandos, en conjunto con los pesos de la función objetivo. Dicho emulador carga el módulo de IA y utiliza sus servicios para deserializar e instanciar la red.

```
1     char filename[50];
2     NEATGenome* sol = population[index];
3     sprintf(filename, "tempSols/%d%d%d.sol", index, this->
          islandIndex, this->params->evolutiveProcessID);
4     char* serialChromosome;
5
6     serialChromosome = new char[1000000];
7     serializeChromosome(sol, serialChromosome, params);
8
9     FILE* file = fopen(filename, "w");
10    if(file != NULL){
11            fputs(serialChromosome, file);
12            fclose(file);
13    }
14    else{
15            printf("%s\n", "No es posible escribir el archivo de
                  solucion.\n");
16    }
17    delete [] serialChromosome;
18
19    char path[200];
20    sprintf(path, "cd \"Fceux Headless/bin/\"; ./fceux -l %d -e
          ../../%s %s", this->params->frameCount, filename,this->
          params->ROM);
21
22    FILE *fp = popen(path,"r");
23    double fitness;
24    fscanf(fp,"%lf",&fitness);
25    pclose(fp);
26    if(fitness < 0){
27    fitness = 1.0 + 1.0/(fitness-1.0);
28    }
29    else{
30    fitness+=1.0;
31    }
32    return fitness;
```

Listado 5.3: Código para evaluación de una red neuronal

El código de evaluación utilizado por el proceso esclavo se presenta en el listado 5.3. La serialización de la red la realiza la función serializeChromosome (línea 7 en el listado 5.3) y la red completa se escribe en un archivo en la línea 11. Si bien el uso de archivos es menos eficiente que enviar los parámetros directamente como argumentos, Linux define un límite máximo del tamaño de todos los argumentos que puede recibir un

ejecutable. Análisis utilizando Fedora 23 indicaron que el tamaño de las redes neuronales serializadas exceden el límite del tamaño de argumentos que soporta dicha distribución (2087286 bytes). El mismo resultado se obtiene sobre CentOS 6.5, presente en los equipos utilizados para realizar la calibración y evaluación del pipeline. Por este motivo, la única opción viable consistió en utilizar archivos para comunicar el proceso de optimización con el emulador, enviando como argumento el nombre del archivo que contiene la red serializada. Se construye el comando a ejecutar concatenando un comando para ubicarse en la carpeta que contiene al ejecutable del emulador y una llamada a dicho ejecutable, pasando como parámetros al emulador la cantidad de frames a emular, el nombre del archivo que contiene la red serializada y el nombre del ROM que contiene el juego a emular (línea 20). La ejecución se realiza con una llamada al sistema utilizando el comando popen (línea 22). Además, se captura la salida estándar asociada al proceso lanzado con el fin de recuperar el valor de fitness que retorna el emulador. Posteriormente, se lee el valor de salida (línea 24). Una vez obtenido el valor, se maneja la posibilidad de haber obtenido un valor negativo de fitness realizando un mapeo al rango (0.,1) (líneas 26-28). En caso de que el valor sea positivo, es desplazado al rango $[1 \cdots + \infty)$ (línea 30).

Del lado del emulador, al simular cada frame se consulta la red neuronal enviando la RAM asociada al frame actual como entrada. Las salidas de la red son interpretadas como la acción de presionar cada botón del control de la NES. Las neuronas de salida utilizan una función de activación sigmoide, para tener salidas entre 0 y 1. Los resultados son discretizados de la siguiente manera: si una salida es mayor a 0.5 se considera que el botón correspondiente debe ser presionado y en caso contrario no se presiona dicho botón. Este esquema soporta combinaciones arbitrarias de botones, requeridas en varios juegos. Al mismo tiempo, se evalúa el frame actual utilizando la función objetivo presentada en la sección 2.2 y los pesos enviados por línea de comandos. El resultado es agregado a un acumulador que representa el valor de fitness calculado hasta el frame que es considerado. Al finalizar el tiempo establecido, la IA retorna el valor final de la función de fitness por la salida estándar.

5.7. Implementación de RNNs

Trabajar con redes neuronales recurrentes presenta varios desafíos en comparación al uso de redes feed-forward, debido a la presencia de enlaces recurrentes. Varios métodos optan por limitar la topología de la red, dando lugar a redes especializadas como *Long short-term memory* (LSTM) (Hochreiter y Schmidhuber, 1997) o *Echo State Network* (ESN) (Jaeger, 2001). En este proyecto se utilizan redes sin restricciones que surgen del proceso evolutivo NEAT. Para soportar redes con topologías arbitrarias se define una representación específica y un algoritmo de evaluación, utilizados por el módulo de IA. Esta representación difiere de la utilizada por los individuos, con el fin de mejorar la eficiencia de la evaluación. La representación utilizada y el proceso de evaluación son presentados a continuación.

5.7.1. Representación de la red

La red es representada como cuatro listas de nodos, según su tipo: de entrada, ocultos, de salida y de sesgo (*bias*). Los nodos de sesgo permiten tener una entrada fija a la red para posibilitar que la red codifique conceptos constantes en base a esta entrada artificial. Para facilitar la propagación de las entradas por la red, cada nodo mantiene una lista de

enlaces salientes, indicando a qué nodo conectan y el peso asociado al enlace. Los nodos mantienen información de la función de activación que poseen. Además, se almacena la *profundidad* (P) de la red, valor necesario para realizar la evaluación. Se utiliza una definición no estándar del concepto de profundidad, definido como el camino más largo en cantidad de enlaces de los caminos más cortos entre cada nodo de salida y cualquier nodo de entrada o de sesgo. Este concepto se formaliza en la ecuación 5.1, donde EB es el conjunto de los nodos de entrada y de sesgo, y S es el conjunto de los nodos de salida. El objetivo de la métrica P es conocer el número mínimo de pasos de propagación que garantiza la activación de todas las salidas de la red utilizando información de al menos una de las entradas. El cálculo de la profundidad se realiza por medio de una búsqueda en profundidad (*Breadth First Search*, BFS), marcando los nodos visitados para para que la búsqueda no quede atrapada en los ciclos. El pseudocódigo del cálculo de P se presenta en el algoritmo 4.

$$\begin{aligned} d(i,j) &= \text{cantidad de enlaces entre el nodo } i \text{ y el nodo } j \\ P &= \max_{j \in S} \min_{i \in ES} d(i,j) \end{aligned} \tag{5.1}$$

Algoritmo 4 Búsqueda en profundidad para calcular la profundidad de la red

$C \leftarrow nuevaCola()$
$encolarNodosEntradaYBias(C)$
$V \leftarrow nuevaCola()$
$profundidad \leftarrow 0$
mientras no se visiten todos los nodos de salida **hacer**
  para cada nodo $N \in C$ **hacer**
    si N aún no visitado **entonces**
      $marcarVisitado(N)$
      $encolar(V, Vecinos(N))$
    fin si
  fin para
  $C \leftarrow V$
  $V \leftarrow nuevaCola()$
  $profundidad \leftarrow profundidad + 1$
fin mientras
retornar $profundidad$

5.7.2. Evaluación de la red neuronal recurrente

Debido al uso de enlaces recurrentes, la propagación de valores dentro de la red puede ocasionar oscilaciones en nodos intermedios. Para evitar este problema se discretiza la evaluación de la red en pasos temporales. De esta forma, los enlaces recurrentes se tienen en cuenta en las neuronas de destino en el próximo paso temporal, proporcionando información del cómputo del paso anterior.

Los enlaces recurrentes permiten a las redes almacenar memoria de las entradas anteriores. Para realizar la propagación por pasos discretos dentro de la red se utiliza una búsqueda en profundidad, colocando inicialmente en la cola los nodos de entrada y de sesgo.

También pueden presentarse oscilaciones en la activación de los nodos de salida, producidos al computar el valor de salida en base al valor de los nodos intermedios cuando se presentan oscilaciones en dichos valores intermedios. Por este motivo, es necesario definir un criterio de parada en el proceso de propagación de señales a través de la red. El criterio elegido para la implementación desarrollada en el proyecto es propagar hasta lograr que todas las salidas hayan recibido información de alguna de las entradas. Esto se logra ejecutando la búsqueda en profundidad durante un número fijo de pasos, coincidente con la profundidad de la red.

Una vez terminada la propagación, se leen los valores de salida y se retornan como salidas de la red. Dada la definición de profundidad de una red, es posible que existan caminos entre nodos de entrada y nodos de salida que sean más largos que la profundidad, por lo que algunos nodos de estos caminos no se activarán en una sola propagación. Para que los caminos más largos puedan ser activados se mantiene la cola de la búsqueda en profundidad entre propagaciones distintas, agregando los nodos de entrada y de sesgo al comienzo de una nueva propagación. De esta forma, algunos nodos pueden requerir más de una llamada a la función de propagación para propagar su información hasta las neuronas de salida. Estos nodos funcionan como memoria de entradas anteriores.

El algoritmo de propagación resultante es una variante del algoritmo BFS utilizado para el cálculo de profundidad de la red. El pseudocódigo del algoritmo de propagación se presenta en el algoritmo 5.

Algoritmo 5 Evaluación de la red neuronal

Entrada: profundidad de la red
  $Q \leftarrow nuevaCola()$
  $encolarNodosDeEntradayBias(Q)$
  $V \leftarrow nuevaCola()$
  para $i \in [1 \ldots profundidad]$ **hacer**
    para cada nodo $N \in Q$ **hacer**
      si N aún no visitado **entonces**
        $marcarVisitado(N)$
        $encolar(V, Vecinos(N))$
        $computarSalida(N)$
      fin si
    fin para
    $Q \leftarrow V$
    $V \leftarrow nuevaCola()$
  fin para
  $salidas \leftarrow leerSalidasDeNodosSalida()$
  retornar $salidas$

5.7.3. Funciones de activación

Además de la función sigmoide utilizada en el artículo original de NEAT, en el algoritmo desarrollado en este proyecto se agregaron nuevas funciones de activación que permiten tener redes más expresivas. Las funciones soportadas son: identidad, sigmoide, softsign, tangente hiperbólica, gaussiana y rectificador paramétrico. A continuación se describe cada una de ellas.

Identidad

La función identidad devuelve el mismo valor que se le pasa como argumento y no posee parámetros. Su expresión matemática se muestra en la ecuación 5.2, y su gráfica en la figura 5.8.

$$id(x) = x \tag{5.2}$$

Sigmoide

Esta función toma su entrada y la escala al intervalo $[0, 1]$. La función depende de un parámetro k que regula la curvatura de la gráfica y permite acotar los valores de salida de un nodo de manera suave (todas sus derivadas son continuas). Su expresión matemática se muestra en la ecuación 5.3. Cuando $k = 1$ se tiene la función logística, cuya gráfica se muestra en la figura 5.9.

$$S(x) = \frac{1}{1 + e^{-kx}} \tag{5.3}$$

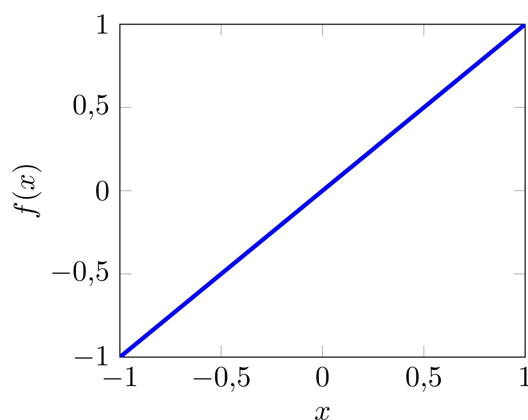


Figura 5.8: Función identidad

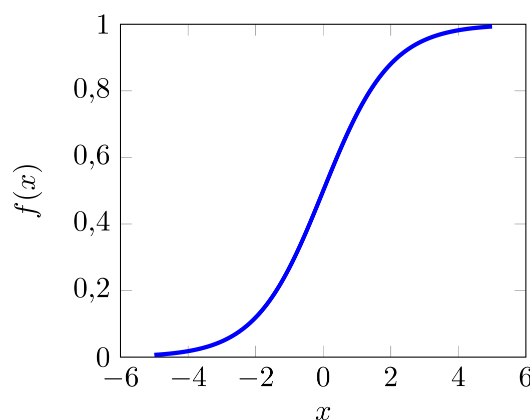


Figura 5.9: Función sigmoide para $k = 1$

Softsign

Esta función de activación tiene una forma similar a la sigmoide pero mapea un valor al intervalo $[-1, 1]$. Las redes que utilizan funciones softsign exhiben mayor robustez que las redes basadas en tangentes hiperbólicas y obtienen mejores resultados en algunas tareas que redes que emplean sigmoides o tangentes hiperbólicas (Glorot y Bengio, 2010). La función softsign requiere un parámetro k que modifica su curvatura. Su fórmula se presenta en la ecuación 5.4, y la gráfica para $k = 1$ puede verse en la figura 5.10.

$$SS(x) = \frac{kx}{1 + |kx|} \tag{5.4}$$

Tangente hiperbólica

La tangente hiperbólica ($tanh$) también mapea sus valores al rango $[-1, 1]$ pero se acerca mucho más rápido a los extremos del intervalo que la función softsign. Esta velocidad permite codificar un punto de quiebre, donde el resultado es cercano a -1 o 1 luego de distanciarse del 0. La función toma un parámetro k de escalado, permitiendo modificar su curvatura. La fórmula se presenta en la ecuación 5.5 y la gráfica para $k = 1$ puede verse en 5.11.

$$tanh(x) = \frac{e^{kx} - e^{-kx}}{e^{kx} + e^{-kx}} \tag{5.5}$$

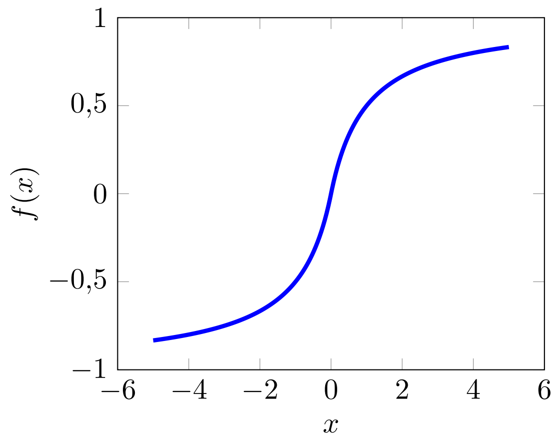


Figura 5.10: Función softsign para $k = 1$

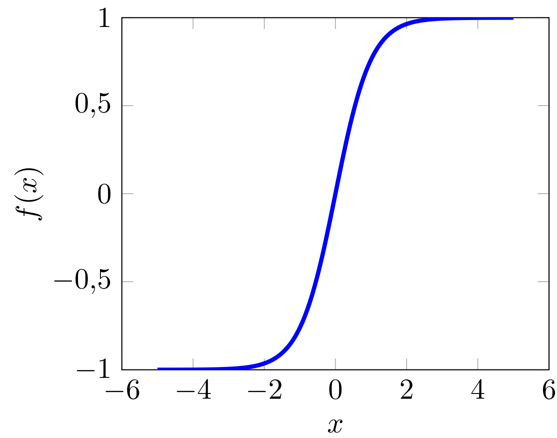


Figura 5.11: *tanh* para $k = 1$

Gaussiana

La función de activación gaussiana sigue una campana de Gauss de media m y desviación estándar s. A diferencia del resto de las funciones presentadas, la función gaussiana exhibe un comportamiento simétrico alrededor de un eje. Esta característica le permite codificar comportamientos diferentes a los de las funciones sigmoide, softsign y tangente hiperbólica. En el caso particular del aprendizaje automático de juegos, la función de activación gaussiana permite modelar comportamientos simétricos exhibidos por juegos como Pinball, en el cual existen sectores de la pantalla que tienen simetría axial (un ejemplo de disposición simétrica es el posicionamiento de los flippers). La fórmula de la función de activación gaussiana se presenta en la ecuación 5.6 y la gráfica para $m = 0$ y $s = 1$ se presenta en la figura 5.12.

$$gauss(x) = \frac{1}{\sqrt{2\pi}s} e^{-\frac{1}{2}\left(\frac{x-m}{s}\right)^2} \tag{5.6}$$

Rectificador paramétrico

La función rectificador paramétrico (He et al., 2015) es una función quebrada que tiene una expresión lineal con coeficiente α para valores de x negativos y una expresión lineal de coeficiente 1 para x positivos. Un nodo que utiliza esta función de activación recibe el nombre de *Parametric Rectified Linear Unit* (PReLU). Esta función es una generalización sobre el rectificador tradicional, donde $\alpha = 0$. Al ser una función partida permite codificar decisiones a partir de un valor crítico. La función se presenta en la ecuación 5.7 y la gráfica para el caso $\alpha = 0{,}1$ se presenta en la figura 5.13.

$$rect(x) = \begin{cases} x & si\ x > 0 \\ \alpha x & sino \end{cases} \tag{5.7}$$

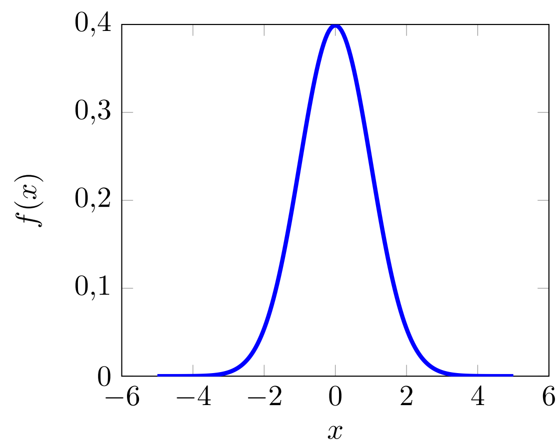


Figura 5.12: Gaussiana para $m = 0$ y $s = 1$

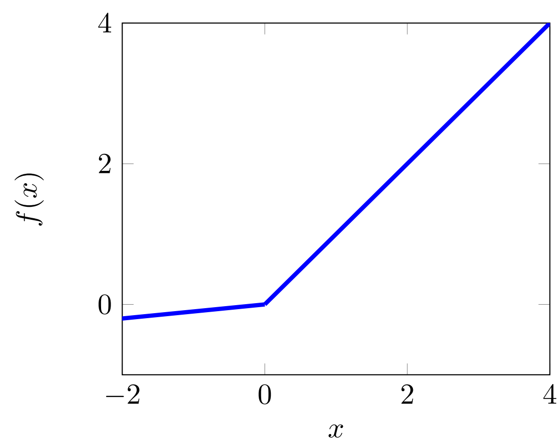


Figura 5.13: PReLU para $\alpha = 0{,}1$

Capítulo 6

Análisis experimental

Este capítulo presenta el análisis experimental del pipeline de aprendizaje automático desarrollado en el proyecto. Se realizaron análisis cuantitativos para calibrar las diferentes etapas del pipeline, así como un análisis del rendimiento computacional de los jugadores artificiales considerando como métrica su tiempo de entrenamiento. Además, se evaluó la aplicabilidad del sistema desarrollado para su uso en tareas de verificación automática de juegos. Los resultados obtenidos y su análisis se describen en este capítulo.

6.1. Metodología del análisis experimental

La evaluación del proyecto se realizó en dos etapas. La primera fue una etapa de calibración de los distintos pasos del pipeline, hallando los mejores valores de los parámetros considerados. Una vez completada la calibración, se realizó la evaluación del pipeline propiamente dicha utilizando los valores paramétricos que permitieron calcular los mejores resultados en la etapa de calibración. Se compararon los resultados obtenidos utilizando funciones objetivo halladas de forma manual contra las funciones objetivo halladas por el pipeline. Además, se contrastaron los resultados obtenidos por el sistema implementado con los resultados de Playfun y con el puntaje obtenido por jugadores humanos que asistieron al evento Ingeniería DeMuestra 2016, en el que se presentó el proyecto.

La interpretación de los resultados obtenidos es difícil, dado que no existe un criterio absoluto que permita evaluar la calidad con la que un jugador se desempeña en un juego dado. Es posible que existan varias estrategias diferentes, todas válidas, que lleguen al final del juego. Para hacer énfasis en el enfoque de verificación automatizada de videojuegos, el análisis de resultados se concentra en las estrategias que explotan errores en los juegos estudiados. Muchas de estas estrategias no representan intentos de jugar la partida de la forma esperada por el desarrollador, sino intentos de progresar en el juego de formas inesperadas. Estas estrategias permiten encontrar errores en casos borde y problemas de diseño. Al encontrar estas estrategias se explica el motivo de los resultados obtenidos y su importancia para verificar juegos automáticamente.

En cambio, algunas IAs desarrollaron estrategias similares a las que se esperaría ver en partidas humanas, llegando incluso a dominar algún aspecto del juego como lo haría un jugador experto. Estos jugadores permiten verificar el correcto funcionamiento del juego al demostrar que existe una estrategia que permite progresar en el juego según lo esperado por el desarrollador.

6.2. Experimentos de calibración paramétrica

La calibración se realizó sobre los parámetros más relevantes de cada etapa del pipeline. La etapa de inferencia posee un único parámetro a estudiar. Para la etapa de refinamiento se estudiaron dos parámetros que influyen significativamente en la calidad de los resultados. Para la tercera etapa se estudió la incidencia de los cambios introducidos al modelo NEAT a través de la calibración de los parámetros que controlan las modificaciones presentadas en la sección 5.6.3. El objetivo de los experimentos de calibración paramétrica fue encontrar los mejores valores posibles para cada parámetro según los resultados obtenidos en los experimentos realizados. A continuación se explican los parámetros estudiados para cada etapa y sus resultados.

Inferencia de objetivos

La etapa de inferencia de objetivos presentó un solo parámetro a calibrar: la cantidad de intentos para buscar un orden lexicográfico disjunto en la RAM. Debido a que la búsqueda de órdenes disjuntos es un proceso estocástico, se realizaron diez experimentos para cada configuración para obtener resultados estadísticamente significativos. Los valores estudiados fueron 100, 1000 y 10000 intentos por cada estrategia de búsqueda aleatoria. Dado que existen ocho estrategias de búsqueda (cuatro que buscan órdenes ascendentes y cuatro que buscan órdenes descendentes, ambas examinando cada 1, 100, 250 y 1000 frames), en total se realizan 800, 8000 y 80000 intentos respectivamente. Los resultados son presentados en la tabla 6.1. Todas las pruebas fueron realizadas sobre el juego Pacman, debido a que fue el que exhibió la mayor cantidad de órdenes lexicográficos de todos los juegos estudiados.

Cantidad de intentos	Órdenes encontrados	
	Ascendentes	Descendentes
800	23.0	0.0
8000	23.0	0.0
80000	22.0	0.0

Tabla 6.1: Análisis paramétrico de la etapa de inferencia

Los resultados de la tabla 6.1 indican que los diez experimentos obtuvieron la misma cantidad de órdenes para todas las configuraciones (800, 8000 y 80000 intentos. Sin embargo, se encontraron menos órdenes lexicográficos al realizar 80000 intentos. Examinando las máscaras generadas pudo verse que la mayor parte de los órdenes detectados no son contiguos en RAM. Este resultado sugiere que se requieren pocos intentos para encontrar los órdenes presentes en memoria. Por este motivo, se optó por trabajar con 800 intentos de búsqueda de órdenes aleatorios.

Refinamiento de objetivos

Para la etapa de refinamiento se estudió la probabilidad de aplicación de los operadores de mutación y cruzamiento del algoritmo evolutivo propuesto para refinar la función objetivo generada en la etapa de inferencia. Se estudiaron todas las combinaciones posibles de los valores de probabilidad de cruzamiento $p_C = \{0{,}8; 0{,}9\}$ y la probabilidad de mutación $p_M = \{0{,}05; 0{,}1\}$. Se realizaron diez experimentos para cada combinación. Los resultados del análisis paramétrico, incluyendo el fitness obtenido, el tiempo en minutos de ejecución y el *p-value* del test de Shapiro-Wilk (S-W) se presentan en la tabla 6.2.

(p_C,p_M)	f				$T(minutos)$				p-value
	min	*max*	*avg*	σ	*min*	*max*	*avg*	σ	*S-W*
(0.9,0.1)	6017696.12	7141679.64	6712986.69	316489.64	18.82	20.50	19.52	0.58	0.55
(0.8,0.1)	6051893.56	7055972.05	6624413.63	276822.30	18.87	20.20	19.55	0.43	0.91
(0.9,0.05)	5640657.93	6947332.75	6351489.08	383028.20	18.77	20.12	19.32	0.40	0.59
(0.8,0.05)	5573519.80	6927328.08	6272696.37	373244.62	18.73	20.42	19.45	0.50	0.85

Tabla 6.2: Análisis paramétrico de la etapa de refinamiento de objetivos

Se aplicaron múltiples tests estadísticos para determinar la mejor configuración del algoritmo. En todos los casos se utilizó un nivel de significancia $\alpha = 0{,}05$. Se comenzó por aplicar el test de Shapiro-Wilk sobre los valores de fitness y se obtuvo como resultado que no se puede descartar la hipótesis nula de normalidad en ninguna de las muestras.

Debido a que el test de Shapiro-Wilk sugirió que todas las muestras de fitness siguen una distribución normal, se empleó el test ANOVA para analizar la relación de las distribuciones subyacentes de cada muestra. El test reportó un p-value de 0.001592, resultado que permitió rechazar con significancia estadística que las medias de cada muestra fueran iguales. Esto indicó que los resultados son estadísticamente diferentes, existiendo una configuración mejor que las demás. Asimismo, se aplicó el test de Bartlett y se obtuvo un p-value de 0.6127, lo que indicó que no se puede rechazar la hipótesis nula de que las varianzas son iguales. Este resultado aporta una nueva evidencia del hecho de que una de las configuraciones es mejor que las demás. Dado que no fue posible descartar que todas las distribuciones tengan la misma varianza y tomando en cuenta que se encontró que tienen distintas medias, se infirió que existe una configuración que en promedio obtiene mejores resultados que el resto (la configuración que tiene media más grande).

Finalmente, dado que se estudiaron múltiples hipótesis, se realizó la corrección de Bonferroni con el objetivo de estudiar la posibilidad de haber cometido un error de tipo 1 entre todos los tests realizados. La corrección ajustó todos los p-values a 1.0 exceptuando el del test ANOVA, cuyo resultado fue 0.009552. Este resultado indica que las conclusiones de los tests no se ven afectadas por errores en las muestras.

Respecto al tiempo de cómputo, los valores obtenidos indicaron que todas las configuraciones requirieron aproximadamente la misma cantidad de tiempo para ejecutar, teniendo valores promedio muy cercanos. Además, las muestras presentaron una desviación estándar baja de alrededor de medio minuto. Estos resultados sugieren que el tiempo de ejecución no impacta en la elección de la mejor configuración.

A partir de los tests estadísticos realizados se concluyó que la configuración $p_C = 0{,}9$ y $p_M = 0{,}1$ permite obtener los mejores resultados de fitness, por lo que fue la configuración utilizada en el resto del análisis experimental.

Entrenamiento de IAs

Para el entrenamiento de IAs se evaluaron dos parámetros específicos del modelo NEAT, la cantidad máxima de enlaces diferentes que pueden tener individuos de una misma especie y el uso o no de las funciones de activación agregadas sobre el modelo original, presentadas en la sección 5.7.3. Debido a que ambos parámetros son independientes entre sí, se estudió primero el uso de funciones de activación adicionales y para el mejor resultado se calibró la cantidad de enlaces que discrimina entre especies. Los resultados se reportan en la tabla 6.3. La figura 6.1 presenta un diagrama de cajas comparativo de la media y la desviación estándar obtenidas para las configuraciones estudiadas.

Funciones extra	δ	f *min*	*max*	*avg*	σ	$T(minutos)$ *min*	*max*	*avg*	σ	*p-value S-W*
No	6	47160.24	59824.01	51116.26	3938.02	48.12	51.37	48.84	0.94	0.08
Sí	6	52480.04	90396.43	67904.55	14261.47	47.88	55.47	49.52	3.32	0.11
Si	12	34360.61	86901.23	61101.85	16316.22	46.83	48.72	47.67	0.66	0.31

Tabla 6.3: Resultados del análisis paramétrico de la etapa de generación de IAs

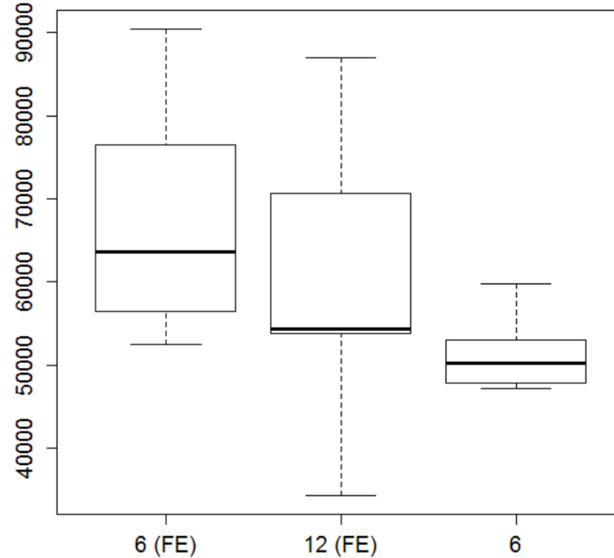


Figura 6.1: Diagrama de caja de las configuraciones del algoritmo de generación de IAs

Al aplicar el test de normalidad de Shapiro-Wilk a las distribuciones de fitness obtenidas no se descartó la hipótesis de normalidad para ninguna muestra.

Para analizar la significancia de los resultados obtenidos para los experimentos sobre el uso de funciones de activación extra se comenzó realizando un test ANOVA, cuyo resultado fue 0.002115, indicando que las medias difieren. Luego se utilizó el test de Bartlett, cuyo resultado de 0.0008128 indicó que las varianzas difieren. El ajuste de Bonferroni no modificó los resultados de los tests. Por lo tanto, se eligió la configuración cuya media es mayor, que corresponde a utilizar las funciones de activación adicionales no contempladas por el modelo original de NEAT. El uso de estas funciones adicionales produjo una mejora del 33 % en el valor de fitness en el caso promedio.

Luego de determinar que el uso de funciones de activación extra mejora la calidad de los resultados, se realizó otro estudio variando el parámetro que define la cantidad de enlaces diferentes necesarios para que los individuos sean considerados de especies diferentes. Se estudiaron los valores 6 y 12 para el número de enlaces. El test ANOVA sobre las distribuciones de los resultados obtuvo un $p-$value de 0.334, que no permitió descartar que las medias sean iguales. El test de Bartlett obtuvo un p-value de 0.6947 que no permitió rechazar que las varianzas sean iguales. Finalmente, Bonferroni aumentó todos los p-values, por lo que no se detectó un error de tipo 1.

Al considerar el tiempo utilizado por los experimentos se constató que el uso de doce enlaces requirió en promedio dos minutos menos que el uso de seis enlaces. Esto se debe a que utilizar doce enlaces redujo la cantidad de especies mantenidas por el proceso,

al permitir que individuos con una diferencia de hasta doce enlaces integren la misma especie. Esta reducción de especies disminuyó el costo del algoritmo, al realizar menos trabajo de selección, mutación y cruzamiento intraespecie. De todas formas, la diferencia de tiempo es de un 3.88 %, en comparación a la diferencia de fitness promedio que es del 11.13 %. La ganancia de fitness significativamente mayor en comparación al tiempo extra utilizado, por lo que se considera que es preferible usar la configuración con seis enlaces.

Considerando los resultados de todos los tests se optó por tomar la configuración cuya distribución mostró una media mayor, correspondiendo a utilizar seis enlaces para diferenciar los individuos de especies diferentes.

6.3. Evaluación de la implementación

Para evaluar la implementación se decidió utilizar dos técnicas diferentes. Se comenzó aplicando un análisis cualitativo de los resultados obtenidos tanto para los jugadores que fueron entrenados de manera totalmente automática como para los jugadores que fueron entrenados con la función objetivo definida manualmente. Luego se realizó una comparación de los mejores jugadores artificiales contra los jugadores humanos que asistieron a Ingeniería DeMuestra 2016.

6.3.1. Comparación entre inferencia automática de objetivos y especificación manual

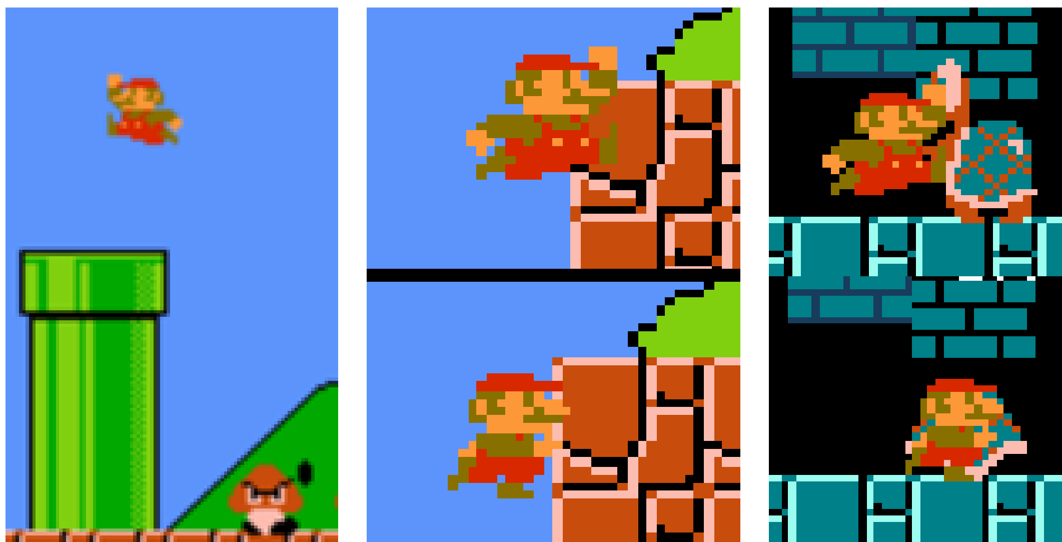
Se procedió a realizar una comparación de los resultados obtenidos por los jugadores cuya función objetivo fue aprendida de forma automática y los jugadores que utilizaron una función objetivo especificada manualmente, para cada juego estudiado. El análisis de los resultados obtenidos se presenta a continuación.

Super Mario Brothers

Super Mario Brothers presenta una jugabilidad sencilla. El juego tiene como objetivo dirigir al personaje (Mario) hacia la derecha de cada nivel, evitando los obstáculos del mapa. Sin embargo, la necesidad de caminar hacia atrás en algunos puntos de los escenarios y las variaciones de altura generan dificultades para los jugadores artificiales. Los jugadores automáticos obtuvieron peores resultados que los jugadores entrenados con el objetivo manual. Esto es esperable, debido a la imposibilidad de incluir las vidas y otros elementos importantes del juego que no forman órdenes lexicográficos en los objetivos automáticos. Sin embargo, algunos de los jugadores automáticos lograron obtener resultados interesantes, como ingresar al atajo presente en el primer nivel, e incluso uno de ellos logró completar el primer nivel exitosamente.

Los jugadores que utilizaron la función manual lograron llegar más lejos en el segundo nivel y fueron en general más consistentes. La mitad de los jugadores generados llegaron al segundo nivel, ya sea utilizando el atajo o recorriendo rápidamente el nivel completo. Muchos jugadores, tanto automáticos como manuales, mostraron habilidad para derrotar enemigos y también para evadirlos. Algunos esperaron a que los enemigos se desplazaran para saltar sobre ellos. Además, algunos jugadores lograron tomar el hongo presente al principio del nivel, que permite romper bloques del escenario con Mario al cambiar a su forma grande y otorga al jugador un punto más de vida, lo que le permite sobrevivir a un ataque enemigo. Si Mario recibe daño mientras es grande, vuelve a su tamaño normal y pierde la capacidad de destruir bloques.

La capacidad de encontrar múltiples estrategias válidas sugiere que los jugadores automáticos son útiles para la verificación del juego, dado que exploraron múltiples estrategias para solucionar los niveles. Esta capacidad facilita encontrar errores en el juego, hecho confirmado al observar jugadores generados que lograron explotar errores de programación. Varios de los jugadores descubrieron que caminar de espaldas con Mario les permitió realizar saltos sin tener la velocidad necesaria para alcanzarlos, debido a un error de programación de la lógica encargada de la física en el juego. Un ejemplo de este salto puede verse en la figura 6.2(a). Uno de los jugadores generados fue capaz de explotar un error que requiere una precisión de un frame (esto es, el error es explotable solamente si se presiona la tecla correcta en un frame particular). El error en cuestión permite a Mario ubicarse dentro de una pared durante un solo frame, debido a un defecto del algoritmo de detección de colisiones. Durante ese frame es posible saltar, debido a que el juego interpreta que Mario se encuentra apoyado sobre el piso. El jugador artificial explotó este error para saltar fuera de un pozo y evitar perder una vida, como puede verse en la figura 6.2(b). Otro error encontrado consistió en que mientras Mario está cayendo el juego considera que cualquier enemigo que colisione con él debe ser aplastado. Esto fue explotado para matar una tortuga en una sección donde no era posible saltar sobre ella en el segundo nivel, debido a que solo hay un bloque de espacio entre el suelo y el techo, como se ejemplifica en la figura 6.2(c).



(a) Mario salta hacia atrás

(b) Mario salta fuera de un pozo al explotar un error en la física del juego

(c) Mario aplasta una tortuga al caer en un espacio reducido

Figura 6.2: Ejemplos de situaciones en las que el jugador automático logra explotar errores en la programación del juego Super Mario Brothers

Ningún jugador logró avanzar más allá del segundo nivel. Esto se explica por la limitada cantidad de tiempo de entrenamiento, dado que se simulan partidas con una condición de parada de 4000 frames (aproximadamente un minuto y 6 segundos de juego en tiempo real). Durante el tiempo de la simulación, los jugadores entrenados no pudieron experimentar mecánicas del juego que son vistas por primera vez luego de la mitad del segundo nivel, como son nuevos enemigos, nuevas plataformas y una disposición del mapa más compleja.

Finalmente, los experimentos permitieron determinar la importancia de los objetivos aprendidos (o especificados manualmente) a la hora de obtener partidas de buena calidad. A modo de ejemplo, uno de los prototipos iniciales trabajó con una función objetivo que valoraba significativamente las vidas, lo que llevó al jugador generado a maximizar las vidas explotando un desbordamiento de enteros en el final del tiempo de entrenamiento. El contador de vidas se mantiene con un entero sin signo. Al perder todas sus vidas, el contador sufre un underflow al restar la última vida al contador, quedando con 255 vidas. Una partida en la que Mario logra morir en los últimos frames era valorada muy positivamente por el prototipo, que solamente evalúa el último frame de cada partida. Si bien para un ser humano la partida fue mala, para los objetivos provistos la estrategia resultó en valores mucho más altos de la función de fitness que jugar la partida en cuestión. Estos inconvenientes fueron resueltos en versiones posteriores del modelo de entrenamiento, eliminando las vidas de la función de fitness. Este nuevo modelo logró modelar una partida de modo razonable desde el punto de vista humano, evaluando solamente los valores presentes en el último frame de la partida.

Contra

Contra es un juego de acción cuyo objetivo es pasar de nivel eliminando enemigos y evadiendo obstáculos. Al final de cada nivel se debe vencer a un "jefe", un enemigo más poderoso que los demás personajes del nivel, que tiene algún poder especial. A diferencia de Super Mario Brothers, el avance de los niveles no es siempre hacia la derecha. El juego presenta niveles con una vista trasera en los que el jugador debe avanzar hacia el interior de la pantalla y niveles verticales donde el objetivo es subir. El jugador comienza con tres vidas y puede adquirir una vida extra al obtener 20.000 puntos y luego cada 60.000 puntos. El puntaje no se muestra en pantalla hasta que el jugador completa un nivel, pero las vidas son otorgadas ni bien se consiga la cantidad de puntos necesaria.

Para combatir enemigos se dispone de un arma de fuego semiautomática que puede ser mejorada o reemplazada por otra arma al adquirir objetos especiales a lo largo del juego. Las mejoras se obtienen al tocar unos elementos gráficos que indican a qué mejora refieren con una letra: S para *spread*, M para *machine gun*, R para *Rapid Fire*, entre otros. El arma más valorada es spread (dispersión) que permite disparar cinco balas simultáneas en diferentes direcciones. Además, el juego permite mantener hasta diez balas de spread en pantalla al mismo tiempo, mientras que las demás armas no aceptan más de cuatro. Por estas razones, si en algún momento se obtiene una S, generalmente es contraproducente obtener otra arma.

Los resultados obtenidos por las IAs en este juego fueron muy variados. Cuando se utilizaron objetivos inferidos por la aplicación se obtuvieron jugadores capaces de avanzar en el primer nivel eliminando enemigos. Sin embargo, los jugadores quedaron atrapados rápidamente en una zona con agua de la cual no fueron capaces de escapar. Otros jugadores pausaron el juego apenas comenzar, permitiéndoles conservar las tres vidas iniciales al no tener que combatir contra ningún enemigo. En cambio, los jugadores generados con objetivos definidos manualmente lograron completar la mayor parte del primer nivel, e inclusive finalizarlo.

La mayoría de las soluciones obtuvieron el arma spread y luego evitaron cambiarla. Utilizando esta arma son capaces de eliminar casi todos los enemigos rápidamente al explotar ciertas particularidades en los algoritmos de generación de balas y de cálculo de daño. Los objetos en el juego sólo pueden recibir daño de una única bala a la vez. Esto quiere decir que si un enemigo es golpeado por varias balas al mismo tiempo, sólo

una de ellas hará daño y las restantes atravesarán al enemigo, como puede observarse en el tanque de la esquina inferior derecha de la figura 6.3. Al disparar cinco balas que se dispersan, es normal que un enemigo sea golpeado por más de una bala a la vez, reduciendo efectivamente el daño potencial que se podría realizar. Además, cuando se tienen menos de diez balas de spread en pantalla al mismo tiempo es posible realizar otro disparo que genere tantas balas como sean necesarias para completar las diez (hasta un máximo de cinco). Si se generan menos de cinco balas, el algoritmo de generación prioriza aquellas direcciones más cercanas a una línea recta. Por lo tanto, si se realiza un disparo cada vez que existen exactamente nueve balas en pantalla, es posible generar un flujo recto de balas. Esto permite realizar la mayor cantidad de daño ya que nunca se golpeará al mismo enemigo con más de una bala al mismo tiempo. Los jugadores artificiales logran reproducir estas circunstancias al accionar el botón de disparo cada dos frames y de esta forma disparar un número reducido de balas cada vez. La figura 6.3 muestra un flujo de balas en línea recta disparadas desde el personaje principal. Utilizando esta estrategia, un jugador artificial fue capaz de eliminar al jefe del primer nivel instantáneamente.

Figura 6.3: Jugador artificial disparando una gran cantidad de balas en simultáneo en el juego Contra

Otra estrategia seguida por algunos jugadores artificiales consistió en encontrar un lugar en el mapa donde el jugador no fuera alcanzado por los enemigos pero fuera posible eliminarlos y el juego continuamente los regenerara. De esta forma el jugador pudo ganar puntos indefinidamente, lo que implicó obtener vidas extra. Estas soluciones son teóricamente perfectas cuando el objetivo buscado es la maximización del puntaje.

Es importante destacar que la solución que llegó al segundo nivel no fue capaz de aprender a jugarlo. En dicho nivel existe un cambio de perspectiva en la cámara y el personaje pasa de tener que moverse hacia la derecha a tener que avanzar hacia el interior de la pantalla. Este cambio de perspectiva no fue aprendido por la IA, que al comenzar el nivel intentó moverse a la derecha siguiendo la misma estrategia utilizada en el primer nivel. De todas formas, el personaje se mantuvo saltando en el costado derecho de la pantalla, donde los enemigos no pudieron dispararle, evitando perder. Como consecuencia, el jugador artificial no avanzó más en el juego pero tampoco perdió la partida.

Finalmente, el jugador que alcanzó el segundo nivel terminó el primero con gran velocidad, llegando a mostrar una habilidad comparable a los *speedrunners* del Contra (jugadores humanos que intentan terminar el juego lo más rápido posible). El resultado obtenido presenta un valor estético que ha gozado de popularidad en los últimos años en sitios como YouTube, donde se comparten videos de partidas similares. Además, existe interés en dicho sitio por videos de IAs jugando juegos de forma poco ortodoxa. El primer video de Playfun obtuvo más de un millón de visitas en YouTube y tuvo cobertura de algunas revistas digitales especializadas en el tema.

Space Invaders

En Space Invaders el jugador controla un cañón láser y tiene como objetivo eliminar oleadas de alienígenas para obtener la mayor cantidad de puntos posible. El cañón está ubicado en la parte inferior de la pantalla y puede moverse únicamente de derecha a izquierda. Los alienígenas se acercan al borde inferior a medida que pasa el tiempo, moviéndose en un patrón de zigzag y disparando al cañón. La velocidad con la que descienden los alienígenas incrementa a medida que se eliminan más enemigos de la pantalla. El jugador cuenta con cuatro búnkeres que sirven de resguardo contra los ataques enemigos. Estos búnkeres pueden ser destruidos gradualmente por los disparos enemigos así como por los disparos del propio jugador. Cada cierto tiempo aparecen platillos voladores que vuelan horizontalmente en el borde superior de la pantalla y otorgan una cantidad aleatoria de puntos al ser destruidos, generalmente alta en comparación al puntaje de los alienígenas comunes.

Los jugadores artificiales generados con los objetivos inferidos no fueron capaces de aprender a jugar este juego. En cambio, simplemente accionaban la pausa al comenzar. Por su parte, al utilizar objetivos manuales se obtuvieron IAs capaces de avanzar el primer nivel hasta casi completarlo. La mayoría de las soluciones dejaron de atacar a los alienígenas comunes para atacar a un platillo volador cuando estaba por aparecer y de esta forma obtuvieron una mayor cantidad de puntos. Inclusive hubieron jugadores que destruyeron platillos disparando entre medio de los demás alienígenas como puede observarse en la figura 6.4.

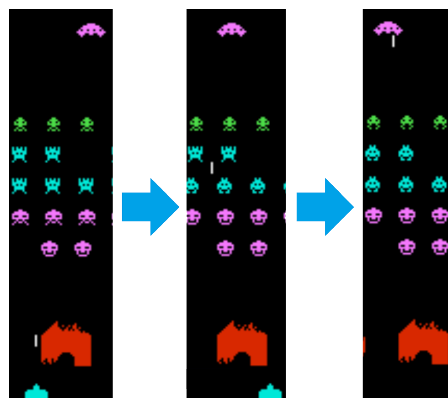


Figura 6.4: Jugador artificial disparando a un platillo volador entre los alienígenas comunes en el juego Space Invaders

Dado que los jugadores artificiales fueron evaluados con simulaciones de partidas de largo fijo, perder la partida antes de que la simulación acabe supone una caída en la evaluación del jugador. Por esta razón, muchas IAs pausaron el juego hasta el final de la simulación cuando estuvieron a punto de perder o pausaron e inmediatamente reanudaron el juego para consumir frames hasta terminar la simulación. Esta última estrategia permitió realizar acciones durante los frames en los que el juego no estaba pausado. En efecto, un jugador artificial aprovechó dichos frames para destruir un platillo volador antes de que la simulación terminara. Estas estrategias representan partidas anómalas que no son útiles para la tarea de verificación, por lo que deben ser impedidas. El comportamiento fue detectado al analizar los resultados finales del proyecto, por lo que no pudo ser solucionado a tiempo. Una solución posible es impedir el uso de la tecla

start por parte de los jugadores artificiales para evitar el desarrollo de estas estrategias. Debido a que la tecla start es necesaria para el avance en otros juegos no estudiados (como Zelda o Contra Force, donde es necesario interactuar con el inventario o el menú del juego durante el transcurso de una partida), puede implementarse la capacidad de prohibir el uso de ciertas teclas para cada juego particular, con el fin de no perder generalidad.

Pacman

El clásico juego Pacman presenta dificultades inesperadas para el aprendizaje por parte de los jugadores artificiales. El juego es simple, ya que consta solo de una pantalla que contiene un laberinto lleno de píldoras que deben ser consumidas por el personaje mientras evade a los cuatro fantasmas presentes. El jugador comienza con 3 vidas y pierde una cuando el personaje es alcanzado por un fantasma. Sin embargo, existen cuatro píldoras especiales, más grandes que las demás, que permiten al personaje comer a los fantasmas durante cierto tiempo después de ser consumidas.

A pesar de su simplicidad, el juego no provee suficiente información al jugador para que éste pueda inferir el objetivo del juego de manera simple. El objetivo obvio de aumentar el puntaje favorece comer a los fantasmas, que otorgan un puntaje que es un orden de magnitud mayor que el que se obtiene al comer las píldoras presentes en el laberinto, yendo en contra del objetivo real del juego. Teniendo en cuenta la dificultad para inferir el objetivo del juego, los resultados de los jugadores con la función objetivo manual superaron a los de los jugadores automáticos. Sin embargo, debe considerarse que la función objetivo manual no tomó en cuenta varios valores que intuitivamente deberían formar parte del objetivo, para evitar confundir el objetivo de comer las píldoras del laberinto con el de comer fantasmas, como fue descrito en el párrafo anterior. Los jugadores automáticos no lograron jugar satisfactoriamente.Los jugadores con la función manual, en cambio, mostraron comportamientos interesantes, aprendiendo a seguir a los fantasmas y evitar ser comidos por ellos en el último momento. Varios jugadores parecieron ser capaces de memorizar comportamientos estáticos de la IA del juego, como es el caso del fantasma rojo que siempre sale hacia la esquina superior izquierda, y explotaron este conocimiento a su favor. Sin embargo, los jugadores no hicieron esfuerzos por ganar la partida, intentando en cambio maximizar el puntaje. Priorizar el puntaje responde a la dificultad encontrada para expresar el objetivo real del juego en términos de la función de fitness, que se relaciona con la confusión entre comer los fantasmas para generar más puntaje y comer las píldoras para terminar el nivel.

Los jugadores entrenados con la función manual lograron explotar errores en el juego. En particular, lograron generar combinaciones de teclas que resultan en oscilaciones de Pacman frente a un fantasma, logrando que el fantasma se desvíe y no lo coma. Además, encontraron formas de quedarse quietos en una esquina luego de comer una de las cuatro píldoras grandes y que los fantasmas se dirijan hacia Pacman para ser comidos por él. El hecho de que los jugadores automáticos explotaron debilidades de las IAs encargadas de manejar los fantasmas sugiere que Pacman puede ser verificado de forma automática. Además, corregir las debilidades detectadas de la IA de los fantasmas puede mejorar la calidad del juego, al ser estas debilidades explotables por jugadores humanos para obtener una ventaja en la partida.

Pinball

Pinball es un juego donde se debe proyectar una bola hacia un tablero utilizando un resorte y controlar un conjunto de paletas o *flippers* para golpear la bola intentando que no se pierda por el fondo del tablero. Dicho tablero contiene varios elementos que otorgan diferentes puntajes al ser golpeados con la bola. Además, ciertas combinaciones de golpes permiten que se accionen recompensas en forma de puntos, que se multiplique el del puntaje obtenido durante un cierto período de tiempo o que aparezcan elementos en el tablero que se interponen a la caída de la bola. El jugador comienza con tres bolas y el juego termina cuando las pierde todas.

Al utilizar objetivos manuales fue posible generar IAs que lograron puntajes en el rango de los 70000 a los 110000 puntos (puntajes altos para jugadores humanos), además de aprender a activar varias recompensas. Las mejores soluciones obtuvieron la mayor parte de los puntos utilizando una estrategia que consistió en colocar la bola encima de un *bumper* de forma que rebote contra él repetidas veces. Cada vez que la bola rebotó sobre el bumper se obtuvieron puntos y además ocasionó que pasara sobre un gatillo que otorgó 1000 puntos cada vez que fue accionado, como se observa en la figura 6.5. Muchas soluciones lograron activar una recompensa que duplica el puntaje otorgado por todos los elementos del juego durante cierto período de tiempo. Al combinar esta recompensa con la estrategia anterior se obtuvieron 2000 puntos cada vez que se accionó el gatillo, incrementando el puntaje. Otras recompensas obtenidas consistieron en ganar una bola extra al sobrepasar los 50000 puntos y activar elementos del tablero que dificultan la pérdida de bolas. Al examinar los jugadores artificiales generados con objetivos inferidos se detecta un comportamiento igual que el constatado en el juego Space Invaders. Los jugadores artificiales pausaron el juego al comenzar y no retiraron la pausa por el resto de la partida. La solución también es análoga, basta con prohibir el uso de la tecla start para este juego particular.

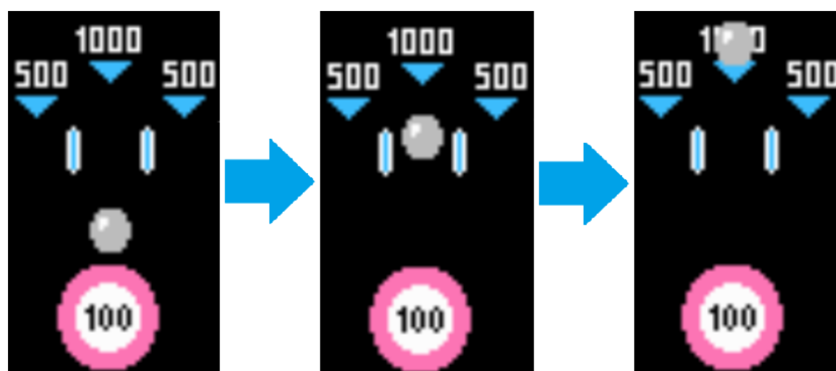


Figura 6.5: Rebote de la bola en el bumper en el juego Pinball

Battle City

El juego Battle City consiste en controlar un tanque de guerra y defender una base militar de un conjunto de tanques enemigos. Existen 35 niveles y cada nivel se completa al destruir todos los enemigos. El jugador comienza con tres vidas. El juego se gana al completar satisfactoriamente los 35 niveles. En cambio, las condiciones de derrota son la destrucción de la base militar o perder todas las vidas. La base puede ser destruida por el propio jugador.

Algunos jugadores entrenados a partir de funciones objetivo manuales lograron pasar el primer nivel destruyendo a los tanques enemigos. En estos casos se observaron estrategias muy simples que probaron ser efectivas, como mantenerse siempre en el mismo lugar disparando a intervalos constantes y destruir los demás tanques cuando estos cruzan frente a las balas. Los enemigos nunca fueron tras la base militar ni atacaron al jugador por el costado, lo que muestra que las estrategias utilizadas por el juego no presentan gran dificultad para el jugador. El hecho de que pueda terminarse el primer nivel solamente utilizando el botón de disparar puede considerarse como un error en el diseño de los niveles, ya que no es necesario mover al jugador hasta que se avanza en el segundo nivel. Este resultado sugiere la utilidad del proceso de aprendizaje como herramienta de verificación automática, al haber encontrado un error substancial de diseño. El jugador artificial fue derrotado al ser destruida la base militar. Esto sucedió luego de los 4000 frames de entrenamiento, por lo que el hecho de que la destrucción de la base implica la pérdida de la partida no pudo ser aprendido por las IAs.

Las soluciones generadas con los objetivos inferidos no lograron aprender satisfactoriamente los objetivos del juego. Al comenzar la partida dirigen el tanque hacia su base y la destruyen, perdiendo el juego de inmediato. Un comportamiento similar fue observado en algunos jugadores artificiales generados con objetivos manuales, que a diferencia de los jugadores generados con la función inferida, navegaron por los menús hasta seleccionar el nivel 35 luego de perder y aparecer la pantalla principal del juego. Dado que los objetivos manuales premiaban el pasaje de niveles, los jugadores que comenzaron el juego en el nivel 35 obtuvieron un gran valor de fitness. Estos resultados sugieren que es necesario las acciones que los jugadores artificiales pueden realizar en el juego, para suprimir estos casos anómalos, de manera similar a como se planteó para el caso del Space Invaders y del Pinball.

Ice Hockey

Ice Hockey es un juego deportivo basado en hockey sobre hielo que posee controles avanzados en comparación con otros juegos de NES, permitiendo pasar, bloquear, amagar, controlar a más de un jugador a la vez, lanzar el disco con distintos niveles de intensidad y respetar varias reglas del hockey sobre hielo, como los penales y las faltas. Además, los jugadores en el campo se dividen en tres clases diferentes, cada uno con sus fortalezas y debilidades. Estas características hacen al juego difícil de aprender para los jugadores artificiales, que deben mostrar capacidades de planificación para logar sus objetivos.

Los jugadores entrenados con la función objetivo manual lograron mejores resultados que los jugadores automáticos. Sin embargo, los jugadores automáticos lograron aprender algunas reglas del juego (como generar faltas y lograr que expulsen jugadores del adversario) y el preferir un empate a una derrota. Uno de los jugadores automáticos encontró técnicas para retener el disco, inhibiendo el saque automático por parte del golero e impidiendo que el oponente controlado por la computadora le robara el disco. De esta forma logró llevar el partido a penales, que por suceder luego del tiempo de entrenamiento la IA no supo cómo jugar y perdió 2 a 0. Estos resultados son muy positivos considerando la complejidad del juego y sugieren que se pueden obtener aún mejores resultados utilizando heurísticas más efectivas para la inferencia de objetivos.

Los jugadores que utilizaron la función manual aprendieron a marcar goles muy rápidamente, logrando realizar buenas jugadas en equipo y venciendo fácilmente al oponente. La velocidad con la que las IAs lograron marcar goles fue muy alta, mostrando un buen dominio de las tres clases de jugadores presentes en el campo. Una de las tácticas utilizadas empleó al jugador más corpulento del equipo para realizar el saque de media cancha y llevar el disco hasta el arco rival, anotando un gol en pocos segundos. Esta jugada es muy difícil de realizar debido a que una de las debilidades del jugador corpulento es su baja capacidad de realizar saques exitosos. La jugada puede visualizarse en la figura 6.6. Después de marcar algunos goles (en general teniendo un resultado favorable de 4 a 0), las IAs decidieron poner la pausa y no jugar más. Esto sucede por no estar prohibido el uso de la tecla start por parte de la IA.

De los errores detectados, el error que permite retener la pelota es significativo, ya que puede ser explotado para asegurar la victoria a un jugador que logre anotar un gol, debido a que el otro equipo no tiene posibilidad de quitarle la pelota y anotar. Errores de este tipo pueden arruinar la experiencia de los jugadores y empeorar la opinión que estos tienen del juego.

La detección de estos errores ejemplifica la capacidad de verificación de las IAs generadas, dado que permite al equipo de desarrollo del juego detectar errores rápidamente que impacten negativamente en la calidad del producto para poder corregirlos antes de liberarlo al mercado.

Figura 6.6: Gol anotado utilizando al jugador más corpulento en el juego Ice Hockey

Tetris

Tetris es un juego de puzzles basado en fichas denominadas tetrominos (por estar compuestas por cuatro bloques) que deben ser apiladas mientras caen sobre el tablero para retirar líneas completas de bloques. Este juego requiere de habilidades de planificación para colocar las piezas de la mejor forma posible. Un buen jugador debe especular con las piezas que pueden llegar en el futuro cercano para maximizar la cantidad de líneas eliminadas. Además, a medida que el jugador avanza en los niveles la velocidad del juego aumenta. En los últimos niveles se requieren buenos reflejos y agilidad para poder jugar correctamente. Si bien los jugadores artificiales poseen la agilidad necesaria, la planificación a futuro es un problema complejo, especialmente dado que en Tetris una jugada puede depender de una ficha que puede aparecer muy distante en el tiempo. Este fuerte componente de planificación a largo plazo limita significativamente el desempeño de los jugadores artificiales.

En este juego particular no es posible comparar el desempeño de ambos tipos de jugadores artificiales de forma directa, debido a que cada uno opta por jugar un modo de juego diferente. El mejor jugador automático jugó la variante de juego A, debido a que fue la variante elegida por los jugadores humanos en los videos de entrenamiento, por lo que la etapa de inferencia generó como prefijo para iniciar la partida una combinación de teclas que inicia una partida en la variante A. El prefijo, además, indica que se debe seleccionar el nivel 9 en el menú de selección de niveles. El jugador automático intentó apilar de la manera más rápida posible una ficha sobre otra, debido a que esto otorga algunos puntos, hasta llegar al último frame donde todavía no perdió la partida (la nueva ficha que excede la capacidad del tablero aparece montada sobre otra ficha ya presente). En ese momento pausó el juego, quedando en este estado por el resto de la partida. Al buscar maximizar el puntaje y los niveles, decidió que la mejor jugada posible era nunca salir de la pausa, evitando perder. Esta acción señala un problema del modelo utilizado, donde la IA debe darle prioridad a no perder la partida para mantener su valor de fitness. Prohibir el uso de la tecla start puede impedir este comportamiento. Se destaca que este problema es también encontrado por Playfun al intentar jugar el juego Tetris.

El jugador entrenado con la función manual encontró una combinación no documentada de teclas que le permitió evitar jugar la variante de juego A. Al iniciar la partida presionó las teclas select, start, B y A simultáneamente en el mismo frame, lo que reinició el juego. De esta forma descartó el prefijo provisto por la etapa de inferencia de objetivos. A diferencia del resto de los juegos estudiados, este jugador artificial aprendió a navegar por los menús de forma autónoma hasta llegar a la modalidad de juego B. En este menú el jugador automático encontró una secuencia de teclas que le permitió iniciar la partida en el nivel 19, el cual no es alcanzable dentro del juego normal. Para ello seleccionó el nivel 9 en la pantalla de selección de nivel y comenzó la partida apretando en el mismo frame el botón A y start. De esta forma el jugador automático logró maximizar el nivel más allá de lo posible jugando de forma normal. Apiló todas las fichas rápidamente y perdió, volviendo al menú de selección de nivel. Luego el jugador automático aplicó la combinación de teclas nuevamente para entrar en el nivel 19, volvió a perder, y así sucesivamente.

Si bien ninguno de los jugadores jugó correctamente al Tetris, este resultado era esperable debido a la necesidad de planificación por parte de los jugadores. Si bien las IAs mostraron comprender el objetivo de progresar en los niveles del juego y la condición de derrota, la falta de capacidad de planificación llevó a estrategias muy ineficientes (apilar bloques) y a pausar la partida antes de perder, evidenciando que las IAs no fueron capaces de aprender las estrategias necesarias para jugar correctamente.

Al considerar la utilidad de los jugadores con fines de verificación, el mejor jugador entrenado con la función manual logró hallar combinaciones de teclas no documentadas que le permitieron obtener una ventaja inesperada, lo cual lo hace un mejor candidato para realizar verificación automatizada. La detección de este tipo de combinaciones permite al equipo de desarrollo identificar códigos de trampa y otros comandos definidos con fines de depuración que deben ser removidos antes de liberar la versión final del juego al mercado. A modo de ejemplo, el célebre código Konami fue inicialmente un código de trampa introducido en la fase de beta testing del juego Gradius, debido a que los beta testers encontraban al juego muy difícil y no podían verificarlo de forma completa. Al momento de publicar el juego, el equipo olvidó eliminar el código, quedando presente en todas las copias vendidas.

Conclusiones del análisis experimental

El uso de técnicas de inferencia automática y definición manual de objetivos presenta distintas ventajas y desventajas. Las técnicas de inferencia automática son las más generales, muestran versatilidad al lograr aprender varios juegos, y pueden ser aplicadas sin modificación a todos los juegos considerados. El proceso es completamente automático, sin requerir intervención humana. Sin embargo, la calidad de los resultados obtenidos varía en función al juego estudiado. En algunos juegos, como Super Mario Brothers, los resultados alcanzados son muy buenos, mientras que en juegos como Tetris, no fue posible entrenar jugadores completamente automáticos competentes con los objetivos inferidos.

La definición manual de los objetivos permite incluir conocimiento específico de cada juego particular, mejorando la calidad de los resultados. Sin embargo, las funciones deben ser definidas de forma independiente para cada juego. Además, debe realizarse una inspección manual de la RAM de cada juego a lo largo de una partida para poder identificar las variables relevantes. Esta tarea implica que el tiempo requerido para definir cada función es significativamente mayor al de inferencia automática de objetivos.

Al analizar las partidas jugadas por los jugadores automáticos fue posible detectar errores en los ocho juegos estudiados. Esto sugiere que los jugadores automáticos son útiles para tareas de verificación automática, permitiendo al equipo de desarrollo de un juego reducir el costo (económico como temporal) de la etapa de verificación. El menor costo fomenta una etapa de verificación más robusta, con más pruebas, que redunda en un juego de mejor calidad y en jugadores satisfechos.

6.3.2. Comparación con jugadores humanos

Para poder contextualizar los resultados obtenidos por los jugadores artificiales se realizó una comparación cualitativa entre los mejores jugadores artificiales y los jugadores humanos que se presentaron a Ingeniería DeMuestra 2016, para cada juego estudiado.

Recolección de datos

Todos los datos utilizados para la comparación con humanos fueron obtenidos de jugadores humanos que participaron en la exposición Ingeniería DeMuestra 2016, organizada por la Facultad de Ingeniería y la Fundación Julio Ricaldoni para presentar al público general proyectos de grado e investigaciones desarrolladas en la facultad. La exposición se realizó durante tres días, durando cuatro horas por día. Se almacenó un video en formato fm2 de cada partida para poder analizar el desempeño de cada jugador y realizar una comparación cualitativa con los jugadores artificiales.

Para recolectar los datos se montó un stand en la exposición donde además de presentar el proyecto se organizó un espacio donde los asistentes podían sentarse a jugar cualquiera de los ocho juegos estudiados en este proyecto. Para ello se dispuso de una laptop conectada a un monitor y se les otorgó a los participantes un teclado y un mouse para registrar su nombre y seleccionar uno de los ocho juegos disponibles. Una vez seleccionado el juego los participantes disponían de un joystick para interactuar con él.

La recolección de datos fue automatizada por medio de una aplicación Java con un menú en JavaFX que ofrecía a los participantes los juegos disponibles y solicitaba su nombre. Una vez que se había elegido un juego la aplicación Java realizó la llamada de forma automática a una versión modificada del emulador FCEUX que permitía iniciar la grabación de la partida de forma transparente. Además, se desarrolló un nuevo módulo

que permitió al emulador retornar el puntaje obtenido por cada jugador al cerrar el emulador. Una vez terminada la partida y retornado el control a la aplicación Java, ésta almacenó los valores obtenidos en una base de datos sqlite, registrando la fecha y hora de la partida, el nombre del jugador, el juego y el puntaje en cuestión. Para fomentar la competencia entre los participantes, la aplicación Java mantuvo una tabla de los mejores puntajes obtenidos en el día para cada juego. Al seleccionar un juego particular, la aplicación mostró los puntajes del juego para el día en cuestión, al costado del emulador. El menú y la tabla de puntajes se presentan en la figura 6.7. Además, al iniciar una partida se ocultó el menú para no obstaculizar el juego. Finalmente, para fomentar la motivación de los participantes, se ofreció un premio al jugador con el mayor puntaje en cada juego al final de cada día.

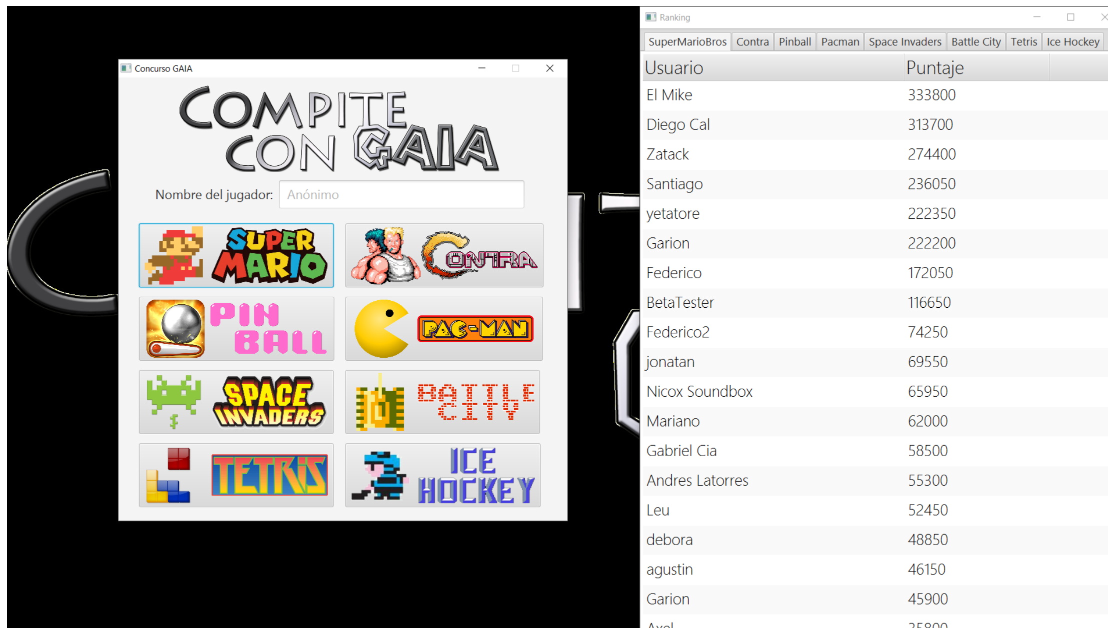


Figura 6.7: Menú para selección de juegos y tabla de puntajes

A partir de los datos recabados durante los tres días de la muestra fue posible estimar el puntaje promedio y la desviación estándar para cada juego. Los datos calculados se presentan en la tabla 6.4.

Juego	*#Partidas*	*Puntaje*			
		min	*max*	*avg*	σ
Battle City	6	1000	11300	4116.67	3790.60
Contra	11	600	28400	8400.00	7877.29
Ice Hockey	2	-8	-6	-7.00	6.93
Pacman	28	930	19900	4590.71	4511.55
Pinball	16	10200	75910	33545.63	33199.21
Space Invaders	8	150	540	373.75	369.83
Super Mario Bros.	45	700	333800	65564.44	64254.93
Tetris	2	15937	25110	20523.50	20004.45

Tabla 6.4: Resultados de jugadores humanos

Super Mario Brothers

Super Mario Brothers fue el juego más popular, atrayendo 44 jugadores a lo largo de la exposición. Además, debido a la gran fama del juego, varios de los jugadores mostraron un profundo conocimiento de múltiples estrategias efectivas para las distintas clases de enemigos y niveles, como también de la disposición de los potenciadores y secretos en cana nivel. A diferencia de los otros juegos de la muestra, los mejores jugadores tuvieron la clara intención de obtener la mayor cantidad de puntos posible, con el fin de competir por el premio otorgado al final del día.

Los mejores jugadores humanos superaron a los jugadores artificiales, obteniendo una cantidad mayor de puntaje y superando más niveles. El mejor jugador humano logró acumular una gran cantidad de puntos, llegando hasta el nivel 5-3. Este jugador mostró un gran conocimiento previo de todos los niveles, aprovechando todas las monedas y otros elementos ocultos para maximizar su puntaje. Para lograr este cometido, optó por jugar de manera muy conservadora, utilizando todo el tiempo necesario para alcanzar con seguridad el puntaje requerido. Este comportamiento contrasta con los objetivos de los jugadores artificiales, que debido al tiempo de entrenamiento limitado tienen un sesgo a jugar lo más rápido posible. Este sesgo produce jugadores que obtienen una gran cantidad de puntos en muy poco tiempo. El mejor jugador artificial obtuvo 13 veces menos puntaje que el mejor jugador humano, pero obtuvo un mayor puntaje por segundo de partida. Esta métrica indicó que los jugadores artificiales son más eficientes que los jugadores humanos, obteniendo más puntaje por tiempo jugado. Es posible que con suficiente cantidad de tiempo de entrenamiento fueran capaces de superar a los jugadores humanos en partidas de la misma longitud.

Contra

Contra atrajo a una cantidad considerable de jugadores que mostraron distintos niveles de habilidad sobre el juego. La mayor parte de los jugadores humanos obtuvieron peores resultados que el mejor jugador artificial (que se ubica tercero en el ranking completo).

El mejor jugador humano logró llegar hasta el final del segundo nivel, utilizando tácticas conservadoras para prolongar la partida. Mostró conocimiento de los potenciadores presentes en el juego y de las estrategias para vencer a los enemigos. El mejor jugador artificial también mostró conocimiento sobre los potenciadores del juego, en particular, sobre la conveniencia de mantener el arma spread una vez adquirida. A diferencia de los jugadores humanos, el mejor jugador artificial optó por una estrategia muy agresiva, terminando el nivel en muy poco tiempo y por tanto superando la cantidad de puntos por segundo obtenida en comparación a todos los jugadores humanos. Si bien el jugador artificial no aprendió a jugar el segundo nivel (que al tener un cambio de perspectiva se juega de forma diferente al primero), encontró una forma de mantenerse vivo de forma indefinida. En este juego el jugador artificial mostró una mayor capacidad sobre el juego en el primer nivel en comparación a los jugadores humanos. Es posible que con el tiempo de entrenamiento suficiente se pueda generar un jugador artificial competitivo con los jugadores humanos en todos los niveles del juego.

Space Invaders

En la muestra Space Invaders atrajo a jugadores poco experimentados, que no lograron superar el primer nivel del juego. Esto no es sorprendente debido a la gran dificultad de este juego, producida por la gran velocidad con la que descienden las filas de enemigos. Si bien todos los jugadores humanos intentaron eliminar las fuerzas enemigas, pocos utilizaron estrategias consistentes para mantener la mayor distancia posible entre el personaje del jugador y las filas de enemigos y tampoco lograron evadir ataques inesperados de dichos enemigos.

El mejor jugador artificial logró superar al mejor jugador humano por el doble de puntaje. Si bien no pudo superar el primer nivel, logró manipular de manera ingeniosa el puntaje provisto por los platillos voladores que aparecieron en la parte superior de la pantalla y manipuló la pausa para prolongar su supervivencia. La capacidad de manipulación del juego que mostraron los jugadores artificiales y su mayor velocidad de reacción al poder presionar teclas en todos los frames resultaron en una ventaja significativa sobre los jugadores humanos.

Pacman

Pacman fue el segundo juego más popular entre los jugadores de Ingeniería DeMuestra, por lo que fue posible estudiar las partidas de varios jugadores. Algunos de ellos demostraron una gran habilidad para el juego, superando los resultados de los mejores jugadores artificiales. El jugador humano con mayor cantidad de puntos logró alcanzar el cuarto nivel y obtener diez veces más puntos que la mejor IA. Sin embargo, otro jugador con menor puntaje llego al quinto nivel, resaltando nuevamente que en el juego Pacman el puntaje no es un buen indicador del progreso a través de los niveles. De manera curiosa, el jugador humano que alcanzó el quinto nivel utilizó algunas estrategias parecidas a las exhibidas por el mejor jugador artificial, atrayendo a los fantasmas a las esquinas para comerlos en el último segundo. También mostró un comportamiento temerario similar al de la IA, moviéndose muy cerca de los fantasmas.

Los resultados del mejor jugador artificial se ubicaron en el promedio de los jugadores humanos, que no lograron pasar del primer nivel y obtuvieron menos puntos comiendo la mayor parte de las píldoras del laberinto. El jugador artificial ganó más puntos por unidad de tiempo al dedicarse a comer fantasmas. Debe tenerse en cuenta que el objetivo dado a este jugador artificial fue maximizar su puntaje en el tiempo de entrenamiento, lo cual contrasta con la mayor parte de los jugadores humanos, que intentaron progresar por los niveles.

Si bien los jugadores humanos obtuvieron mejores resultados que los artificiales, el hecho de que el mejor jugador artificial se posicionó en el promedio de los jugadores humanos y logró exhibir algunas técnicas utilizadas por el mejor jugador humano sugiere que Pacman puede ser automatizado de manera exitosa con fines de verificación. Los jugadores artificiales encontraron comportamientos deseables y explotaron debilidades de la IA de los fantasmas de forma similar a como lo hicieron los humanos. Con el uso de mejores heurísticas para la inferencia de objetivos y una función de fitness que fomente el avance a través de los niveles puede ser posible obtener mejores resultados.

Pinball

Los participantes tuvieron resultados muy variados en el juego Pinball. Los mejores jugadores humanos no fueron capaces de superar los resultados obtenidos por las IAs. El mejor jugador humano obtuvo 75910 puntos en su partida, mientras que el mejor jugador artificial obtuvo 102000 puntos con la primera bola. Tan solo dos jugadores humanos lograron obtener una bola extra y ninguno pudo activar la recompensa de puntaje doble o explotar en gran medida los 1000 puntos otorgados por el gatillo superior. Además, el jugador artificial alcanzó su alto puntaje en menos de la mitad del tiempo que les llevó a los mejores jugadores humanos completar sus partidas. Los resultados observados sugieren que Pinball puede ser automatizado satisfactoriamente con el fin obtener un jugador artificial capaz de sobrepasar al humano promedio.

En comparación a los resultados obtenidos con un prototipo anterior de este proyecto (que fue desarrollado para automatizar el aprendizaje del juego Pinball), la mejor IA generada por el prototipo obtuvo 310570 puntos. De esta forma superó por un poco más del triple el puntaje del mejor jugador artificial de este proyecto. La inclusión de conocimiento del problema permitió al prototipo obtener mejores resultados con menos tiempo de entrenamiento, evidenciando una capacidad de aprendizaje sustancialmente mayor. Sin embargo, la técnica actual es más general que la del prototipo. Al igual que con el proyecto actual, el prototipo superó a todos los jugadores humanos con los que se lo evaluó. Una comparación detallada del prototipo y del proyecto actual se presenta en la sección 6.3.3.

Battle City

Durante la muestra Battle City atrajo algunos jugadores con distinto conocimiento del juego. Los mejores jugadores humanos lograron superar los resultados de los jugadores artificiales, además de exhibir técnicas más complejas de juego y adelantarse a la IA simple de los tanques controlados por la máquina.

El jugador artificial con mayor puntaje logró llegar al segundo nivel del juego, ubicándose en el medio del ranking de puntajes humanos. A diferencia de los jugadores humanos, el jugador artificial encontró una estrategia extremadamente simple, requiriendo solo el uso del botón de disparo, para vencer el primer nivel completo y llegar a la mitad del segundo. Este estilo de juego se basó en explotar mejor que los jugadores humanos las debilidades de los tanques controlados por el ordenador, dejando en evidencia problemas graves de diseño del juego. Si bien ninguno de los jugadores artificiales entrenados logró superar a los jugadores humanos en la cantidad de puntaje obtenida o la cantidad de niveles logrados, los resultados obtenidos encontraron técnicas alternativas poco obvias para los jugadores humanos, en especial en el primer nivel, al eliminar a todos los enemigos sin mover el tanque.

Ice Hockey

Ice Hockey fue el juego menos popular de toda la muestra; solamente cuatro jugadores lo jugaron durante los tres días. Esto puede atribuirse a que se trata de un título poco conocido y a que presenta una gran dificultad al jugarlo por primera vez. Como se explicó en la sección 6.3.1, el juego presenta muchos elementos que generan una curva de aprendizaje pronunciada (diferentes clases de jugadores, controles complejos, etc.).

Los jugadores artificiales lograron superar el desempeño de los jugadores humanos en todas las partidas. La mayor parte de los jugadores artificiales lograron victorias jugando solamente el primero de los tres tiempos del partido. Incluso, el único jugador artificial que completó una partida (perdiendo 2 a 0 por penales) logró resultados más favorables que los jugadores humanos, que recibieron entre 6 a 8 goles en contra. Comparando las estrategias logradas por los humanos y las IAs, las últimas realizan jugadas más rápidas y efectivas, concretando goles en los primeros segundos de la partida, mientras que los humanos tienen grandes problemas para concretar algún gol. Para este juego los jugadores artificiales lograron encontrar una estrategia efectiva y consistente que les permitió vencer al oponente con facilidad.

Tetris

Tetris tuvo muy pocos jugadores durante la muestra. Sin embargo, los jugadores humanos obtuvieron resultados muy superiores a los artificiales, tomando en cuenta la habilidad para jugar. Los mejores jugadores mostraron un buen nivel de planificación a corto, mediano y largo plazo, esperando a veces más de 20 fichas antes de obtener un tetromino de 4 bloques en línea para liberar una gran porción de la pantalla. Estas técnicas están fuera de la capacidad de aprendizaje de las redes neuronales que pueden ser generadas por la IA con el tiempo de entrenamiento provisto, dado que las redes no pueden crecer lo suficiente para generar estructuras que puedan recordar eventos tan alejados en el tiempo. La falta de complejidad suficiente tampoco permite a las redes manejar los diversos aspectos de planificación necesarios para jugar correctamente al Tetris.

6.3.3. Comparación con algoritmo evolutivo especializado

Previo al trabajo de este proyecto se desarrolló un prototipo inicial para analizar la viabilidad de la propuesta, considerando un enfoque manual para la determinación de los objetivos. Dicho prototipo se desarrolló considerando un modelo especializado en evolucionar IAs para jugar al Pinball (Parodi et al., 2016). Las técnicas utilizadas en el prototipo son más simples y son específicas para el juego estudiado. Se utilizó un algoritmo evolutivo para evolucionar un conjunto de parámetros utilizados por un algoritmo fijo que codificó las acciones posibles para el Pinball (mover un flipper o sacar la bola) y conocimiento general del juego (posición de los flippers y del pistón). Además, se definió un algoritmo heurístico estático (no usa parámetros ajustados por el algoritmo evolutivo) para jugar el bonus stage. Se ejecutó un conjunto de pruebas variando la cantidad de frames emulados en el cálculo de la función de fitness. En todas las pruebas se utilizó una población de 40 individuos y se ejecutaron 1000 generaciones. Los resultados obtenidos se presentan en la tabla 6.5.

Frames	*Puntaje*			
	min	*max*	*avg*	σ
10000	20930	223320	113682.4	59657.7
30000	77620	284540	157909.2	83061.2
50000	23360	310570	127367.7	118515.5

Tabla 6.5: Resultados de los jugadores generados por el prototipo

El prototipo fue comparado además contra los resultados obtenidos por Playfun (Murphy, 2013). El prototipo superó significativamente la eficiencia computacional de Playfun, al jugar una partida completa de 4 minutos y 38 segundos antes de perder, obteniendo 65420 puntos. Para ello entrenó durante 25 generaciones, requiriendo un tiempo total de entrenamiento en una computadora con un procesador Intel i7-920 y 6 GB de RAM. En cambio, Playfun requirió un total de 17 horas y media de cómputo en el mismo equipo para producir una partida de un minuto y 26 segundos, con un puntaje de 24730 puntos y dos bolas extra. La ejecución completa del algoritmo evolutivo utilizado por el prototipo (1000 generaciones) requiere tres horas y 14 minutos en el mismo equipo. Además, los jugadores generados por el prototipo pueden jugar un número indefinido de partidas, mientras que Playfun debe ser ejecutado nuevamente para cada partida que se quiera jugar (Playfun no genera IAs, sino que retorna un video de la partida jugada). La IA generada por el prototipo tiene un costo computacional constante, por estar compuesta por sentencias if y la lectura de un número fijo de datos de entrada. Finalmente, Playfun requiere de un video de entrenamiento provisto por un jugador humano para aprender, mientras que el prototipo requiere de la definición de un esqueleto para la IA y la función objetivo del algoritmo evolutivo. Al comparar los resultados de este proyecto con Playfun, se observa una situación similar. Si bien el pipeline generado en este proyecto requiere más tiempo de cómputo que el prototipo, el pipeline es sustancialmente más rápido que Playfun. El costo de ejecutar la IA es bajo.

Al tener información específica del problema y un espacio de búsqueda más reducido, el prototipo logró mejores resultados que las técnicas desarrolladas en este proyecto. Además, debido a su menor costo computacional, el prototipo pudo entrenar utilizando desde 10000 hasta 50000 frames en comparación a los 4000 frames empleados en el proyecto actual. Los resultados del algoritmo evolutivo especializado sugieren que el proyecto actual sería capaz de obtener resultados similares en el caso que utilizara el mismo tiempo de entrenamiento. El mejor jugador generado por el prototipo alcanzó 310570 puntos, ubicándose cuarto en el mundo en el ranking online UberNES (UberNES, 2017), en comparación a los 102000 obtenidos por el mejor jugador generado con este proyecto. El prototipo inicial y el sistema presentado en este proyecto superaron a los jugadores humanos, tanto en la cantidad de puntos que obtienen por segundo, como en el puntaje obtenido. Al automatizar el proceso de aprendizaje se mejoró sustancialmente la generalidad y la versatilidad de la generación de IAs para otros juegos, pero las IAs resultantes exhiben menor precisión y especificidad. Además, el tiempo de entrenamiento es mayor, al tener que explorar un espacio de soluciones más amplio, debido a que el modelo es general y no realiza suposiciones específicas a cada juego.

6.4. Aplicabilidad como herramienta de verificación

Este proyecto consta de varios objetivos principales. En primer lugar, se busca automatizar el desarrollo de jugadores artificiales con fines de verificación. Además, se busca disminuir el costo y el tiempo requerido en la etapa de verificación por medio del uso de jugadores artificiales. Finalmente, se busca mejorar la calidad del proceso de verificación al utilizar jugadores artificiales que pueden emplear estrategias poco ortodoxas, lo que permite detectar errores no esperados por los desarrolladores o el equipo de verificación. Esta sección presenta un análisis cualitativo de los jugadores obtenidos en relación a la capacidad de alcanzar dichos objetivos.

En primer lugar, el análisis experimental indicó que los mejores jugadores para verificación son los resultantes del uso de una función de fitness definida manualmente. Esto es razonable, debido a que la definición del objetivo del juego con un método manual es mucho más precisa que la obtenida por el método automático y no posee ningún tipo de ruido (no considera variables no relacionadas al objetivo). Sin embargo, algunos jugadores cuyo objetivo fue inferido automáticamente mostraron buenas habilidades de verificación al explotar errores del juego. Los resultados obtenidos con funciones definidas manualmente e inferidas sugieren que es posible mejorar la capacidad de los jugadores entrenados con funciones inferidas por medio de mejoras en el algoritmo de inferencia de objetivos, que permitan inferir funciones objetivo similares a las definidas manualmente.

Los jugadores encontraron errores en todos los juegos susceptibles de ser verificados automáticamente. Este resultado prueba la factibilidad de esta técnica como herramienta de verificación. Además, la mayor parte de los errores encontrados fueron severos, ya que impactan negativamente en la competitividad entre jugadores al permitir a quien los explote obtener una ventaja significativa sobre el resto. Algunos de los errores detectados son de diseño, como es el caso de la IA que controla los tanques enemigos en el juego Battle City. Los errores de diseño son más difíciles de corregir e indican problemas importantes que afectan al diseño completo del juego. Si los errores detectados fueran corregidos, mejoraría sustancialmente la calidad del juego. En este aspecto, se considera que el sistema desarrollado fue exitoso.

A lo largo de los experimentos realizados se obtuvieron algunos resultados anómalos, como fueron el uso de la pausa en el juego Space Invaders y el uso del menú de selección de niveles en el juego Battle City. Para impedir estos comportamientos no deseados se requiere una modificación del proyecto actual para prohibir ciertas acciones de la IA al interactuar con un juego particular. En los casos mencionados, la solución es prohibir el uso de la tecla start. Sin embargo, algunos juegos requieren el uso de la tecla start para acceder al inventario del juego o a una pantalla de selección de personajes, como son el caso del juego Legend of Zelda y Contra Force, respectivamente. La solución más simple es limitar las teclas que la IA puede presionar para cada juego en particular. Implementar esta funcionalidad es sencillo, ya que basta con eliminar las salidas de la red neuronal que corresponden a los botones que se desean prohibir.

Al considerar las estrategias aprendidas por los jugadores artificiales, se constató una preferencia por estrategias poco ortodoxas que explotan errores del juego (como en el caso del Super Mario Brothers) y jugadas arriesgadas para avanzar por los niveles lo más rápido posible (como en el caso del Contra). Estas estrategias contrastan con las jugadas más conservadoras de los jugadores humanos. Además, los jugadores humanos no explotaron errores para su beneficio, exceptuando los problemas de diseño de la IA del Battle City. Estas diferencias de estrategias hacen de las IAs buenas candidatas para la verificación automática de juegos, al explorar estrategias que los humanos no explorarían.

Finalmente, el tiempo necesario para realizar la verificación de cada juego es relativamente bajo, requiriendo alrededor de dos horas para definir la función de forma manual (identificar los valores relevantes de RAM y asignarle pesos) y entre 4 y 12 horas de entrenamiento, dependiendo de la cantidad de generaciones y la cantidad de frames simulados. Este costo computacional resulta competitivo con la verificación manual, sin incurrir en los costos económicos asociados al personal y a los equipos utilizados para la tarea de verificación. Además, un desarrollador, al poseer el código fuente y conocimiento del funcionamiento interno del juego, requiere mucho menos tiempo para definir una

función de fitness de forma manual (al no necesitar deducir el significado de los distintos valores de RAM). Asimismo, la función objetivo resultante probablemente sea mejor, al tener en cuenta el desarrollador aspectos del funcionamiento del juego que no pueden ser encontrados fácilmente por una inspección del estado del juego. Estas ventajas por parte del desarrollador permiten reducir aún más el tiempo necesario para realizar el entrenamiento, mejorando la eficiencia del proceso. Si bien el proyecto solamente trabaja con la consola NES, es posible formalizar el API definido para trabajar con la NES con el fin de usar el pipeline con un juego arbitrario. Realizar esta modificación es sencilla debido a que la interfaz actual se diseñó teniendo en mente la posibilidad de modificarla para un juego arbitrario.

En conclusión, el análisis experimental indicó que el proyecto desarrollado logró explotar errores de programación y de diseño de forma consistente en varios de los juegos estudiados. En combinación con el bajo costo económico del pipeline en comparación a la verificación manual y la nueva óptica aportada por las IAs, esta técnica resulta prometedora para la verificación automatizada de juegos. Para mejorar la capacidad de verificación se propusieron cambios menores que permiten mitigar la generación de jugadores anómalos y facilitar el uso del pipeline con juegos arbitrarios.

6.5. Determinación empírica de la eficiencia computacional del algoritmo de entrenamiento de Inteligencias Artificiales

Con el fin de obtener información detallada de la eficiencia computacional del sistema implementado, se realizó un análisis de la eficiencia del algoritmo evolutivo para la generación de redes neuronales, que es la etapa que domina el tiempo de cómputo del pipeline evolutivo. El análisis se basó en la instrumentación manual del código para determinar las funciones que requieren más tiempo de ejecución. Los resultados se presentan en la tabla 6.6. Todos los resultados se computaron al comprar el costo de cada una de las 800 generaciones consecutivas del algoritmo evolutivo.

Función	*Tiempo(s)*			
	min	*max*	*avg*	σ
Evaluación de fitness	0.85	31.16	15.92	8.72
Cruzamiento y mutación	0.10	0.29	0.14	0.01

Tabla 6.6: Resultados de eficiencia computacional del algoritmo evolutivo (tiempo de ejecución por generación)

Los resultados obtenidos muestran que el tiempo de cómputo es dominado por la evaluación de la función de fitness. Esto es esperable, debido al alto costo de la simulación. La aplicación de los operadores evolutivos demanda un tiempo despreciable y crece muy lentamente a lo largo de las generaciones, como se muestra en la figura 6.9. El aumento del tiempo de ejecución responde al crecimiento paulatino de las redes, que al avanzar las generaciones poseen cada vez más nodos y enlaces. Al continuar el análisis se constató que el uso de más generaciones resulta en un crecimiento cuadrático del tiempo de cómputo de la evaluación de la población de una generación (y no lineal como es frecuente al usar algoritmos evolutivos), como se reporta en la figura 6.8. El incremento del tiempo de

ejecución de la evaluación sucede debido a que la simulación de la población asociada a una generación completa es siempre mayor al anterior, provocado por el crecimiento de las redes generación a generación. El factor cuadrático de crecimiento se dedujo de forma empírica. La figura 6.8 sugiere un crecimiento lineal del costo de evaluar la población generación a generación, coincidente con el costo cuadrático del algoritmo sobre la cantidad de generaciones. Para verificar la relación entre la cantidad de frames simulados y el tiempo total de cómputo, se repitió el mismo experimento con una cantidad fija de generaciones mientras se variaba la cantidad de frames simulados. Estos experimentos mostraron una dependencia lineal entre ambos elementos.

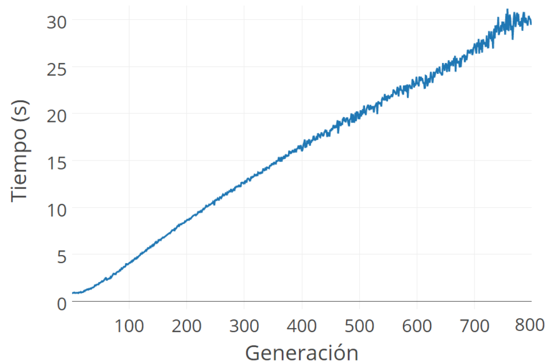


Figura 6.8: Tiempo requerido para la evaluación de la función de fitness en función de la generación

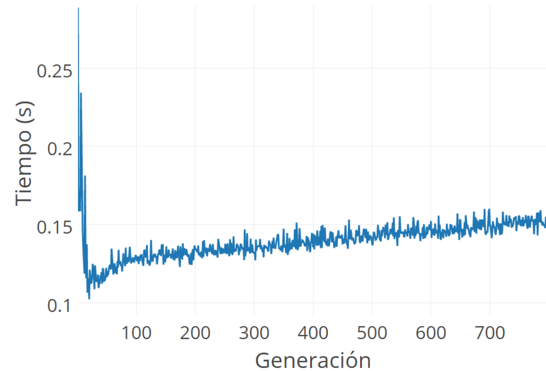


Figura 6.9: Tiempo requerido para el cruzamiento y mutación en función de la generación

Para el prototipo presentado en la sección 6.3.3 se experimentó con el uso de un coprocesador Xeon Phi, con el fin de paralelizar la evaluación de la función objetivo (Rodríguez et al., 2016). Las tarjetas Xeon Phi están equipadas con núcleos basados en una modificación de la arquitectura Pentium, soportando la ejecución de software x86 por medio de la recompilación del código fuente. Para la ejecución del prototipo se utilizó una tarjeta Xeon Phi modelo 31S1P que cuenta con 57 procesadores capaces de ejecutar 4 hilos simultáneos, con una frecuencia de reloj de 1.1GHz, 28.5MB de memoria caché L2 compartida y 8GB de memoria RAM GDDR5.

Para analizar el rendimiento del prototipo al utilizar la Xeon Phi, se trabajó con una sola población de 200 individuos, evolucionando 100 generaciones. Se simuló una partida de 4000 frames para cada individuo. Se realizaron experimentos utilizando 55, 110, 165 y 220 hilos dedicados a la evaluación de los individuos. Los resultados de eficiencia computacional se presentan en la tabla 6.7 y en la figura 6.10.

#Hilos	*Tiempo (minutos)*	
	avg	σ
55	181.47	2.18
110	93.63	0.62
165	65.68	0.93
220	50.20	0.31

Tabla 6.7: Tiempo de ejecución al variar el número de hilos utilizados para evaluación de la función de fitness en la versión paralela ejecutando en Xeon Phi

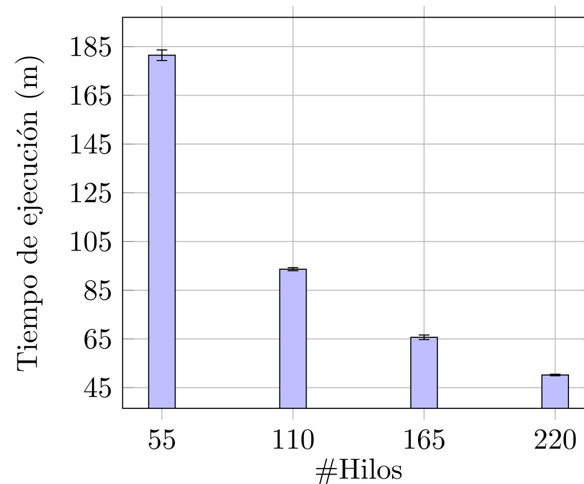


Figura 6.10: Gráfica del tiempo de ejecución al variar el número de hilos

Los resultados de eficiencia computacional indican que el algoritmo logró escalar satisfactoriamente al incrementar la cantidad de hilos utilizados para la evaluación de la función de fitness, hasta llegar al máximo de hilos físicos soportados por el coprocesador (220 hilos). La capacidad de escalado del prototipo sugiere que la implementación del proyecto desarrollado en este trabajo puede obtener un buen desempeño al ser portado a la arquitectura MIC. Sin embargo, la posible mejora del desempeño depende de la capacidad de portar de forma eficiente el código de evaluación de la red neuronal a la arquitectura MIC. En particular, debe explotarse la unidad de vectorización presente en la Xeon Phi para poder obtener una mejora del desempeño. En caso de lograr portar el código de evaluación a la placa con un desempeño aceptable, el soporte para realizar el offload del código de evaluación hacia el coprocesador ya se encuentra implementado en el framework evolutivo, por lo que realizar experimentos de evaluación de la eficiencia computacional al ejecutar en la Xeon Phi será una tarea sencilla.

Capítulo 7

Conclusiones y trabajo futuro

Este capítulo resume las principales conclusiones obtenidas del desarrollo del proyecto y presenta las principales líneas de trabajo futuro.

7.1. Conclusiones

En el proyecto se desarrolló un sistema que permite generar de forma automática jugadores para distintos juegos de la plataforma NES. Para ello se toman como entradas del proceso videos de jugadores humanos sobre el juego a entrenar, previamente clasificados como videos buenos o malos. A partir de estos videos el sistema intenta inferir los objetivos del juego y entrenar un jugador artificial utilizando técnicas neuroevolutivas.

Los resultados obtenidos indicaron que las técnicas utilizadas obtienen buenos resultados, logrando aprender de forma automática a jugar al Super Mario Brothers y presentando resultados interesantes para el Tetris. Sin embargo, la inferencia automática tiene problemas con varios de los juegos estudiados, obteniendo resultados inferiores a los alcanzables al proveer una función objetivo manualmente. Utilizando funciones objetivo provistas por el usuario los resultados mejoran significativamente, logrando aprender los controles de todos los juegos y jugando partidas que alcanzan objetivos del juego, como son avanzar de niveles o derrotar enemigos, para la mayor parte de los ocho juegos estudiados. La calidad de los jugadores obtenidos depende sustancialmente de la función objetivo provista y de la complejidad del juego, lo que implica que para algunos juegos las estrategias aprendidas sean poco ortodoxas, como es el caso de Tetris o Battle City, donde las IAs deciden no jugar y en cambio manipulan los menús para maximizar el nivel donde comienzan.

Respecto al objetivo de verificación perseguido por este proyecto, el sistema fue capaz de generar jugadores automáticos que explotaron errores de diseño o programación en todos los juegos estudiados. Este resultado es muy prometedor, sugiriendo que las técnicas presentadas pueden ser utilizadas con buenos resultados para realizar verificación automática de juegos. Además, la velocidad con la que las IAs progresan por las partidas es muy ágil, permitiendo encontrar errores rápidamente.

Al comparar el desempeño de los jugadores artificiales contra los participantes de la muestra Ingeniería DeMuestra 2016 se constata una notoria ventaja por parte de los jugadores humanos en lo que respecta a progresar a través de los niveles del juego. Sin embargo, los jugadores artificiales logran una eficiencia mucho mayor, obteniendo mayor puntaje que los humanos en el mismo tiempo. Este resultado responde a un sesgo hacia

jugadores más rápidos introducido en el sistema de aprendizaje al fijar un límite en el tiempo de entrenamiento disponible para los jugadores artificiales. Además, los jugadores artificiales fueron los únicos en explotar de forma consistente errores del juego para su provecho.

Al considerar todos los resultados obtenidos se concluye que el sistema presentado cumple los objetivos de verificación automática propuestos. Los resultados son prometedores, especialmente al utilizar una función objetivo definida por el usuario. Proporcionar dicha función es una tarea fácil para un desarrollador que tenga un conocimiento profundo del funcionamiento interno del juego, por lo que no representa una dificultad para utilizar el sistema como herramienta de verificación. La inferencia automática también obtiene resultados prometedores, pero requiere de mejores heurísticas para poder ser aplicada exitosamente a todos los juegos.

A lo largo del proyecto se generaron prototipos que resultaron en publicaciones de artículos científicos. Los artículos publicados fueron: Optimizing a pinball computer player using evolutionary algorithms, presentado en la Latin-Iberoamerican Conference on Operations Research (CLAIO), Santiago de Chile, 2016 (Parodi et al., 2016) y Evaluation of a Master-Slave Parallel Evolutionary Algorithm Applied to Artificial Intelligence for Games in the Xeon-Phi Many-Core Platform, presentado en la Latin America High Performance Computing Conference (CARLA), Ciudad de México, 2016 y en la serie de Communications in Computer and Information Science de Springer (Rodríguez et al., 2016).

7.2. Trabajo futuro

A lo largo del proyecto surgieron varias líneas de trabajo futuro.

Una línea de trabajo a seguir se relaciona con mejorar las heurísticas utilizadas en la etapa de inferencia de objetivos. Si bien las heurísticas propuestas lograron obtener resultados prometedores, la inferencia de objetivos es clave para el proceso de aprendizaje automático. El uso de nuevas heurísticas que logren descartar más lugares de RAM permite reducir el espacio de búsqueda, simplificando la tarea de optimización y aprendizaje del algoritmo evolutivo. Además, el uso de nuevas heurísticas que permitan inferir correctamente relaciones entre las direcciones no enmascaradas de la RAM permitiría mejorar la calidad de la población inicial, enfocando el proceso evolutivo a un subconjunto reducido del espacio de soluciones considerado que según la etapa de inferencia considera solamente las direcciones de RAM relevantes para los objetivos del juego. Esta etapa del pipeline es la que puede permitir obtener mejores resultados, al influir en todas las etapas posteriores.

Una posible mejora relacionada a la inferencia de objetivos es perfeccionar el refinamiento de los objetivos inferidos, realizado por la etapa 2 del pipeline. La técnica actual presenta un sesgo marcado a funciones objetivo que sean monótonas. Sin embargo, existen juegos donde los objetivos claramente oscilan, y deben ser optimizados con una estrategia diferente (como es el caso del Pinball, donde la posición de la bola varía a medida que esta rebota con los demás objetos del tablero).

Para mejorar la utilidad del proyecto como herramienta de verificación, es deseable formalizar la interfaz de comunicación entre el juego siendo aprendido y el pipeline de aprendizaje. La existencia de una API bien documentada permitiría implementar rápidamente un sistema de aprendizaje automático para cualquier juego con solo cumplir con los requerimientos del sistema. Las funciones necesarias para crear dicha API son tres, según la experiencia obtenida en el transcurso de este proyecto, por lo que implementarlas dentro de un juego que deba ser verificado no requiere una cantidad significativa de esfuerzo por parte del desarrollador. Sin embargo, para maximizar la velocidad de verificación el desarrollador debe proveer la capacidad de ejecutar el juego sin límite de frames y sin salida de video, con el objetivo de dedicar todos los recursos a la simulación de la lógica del juego. La dificultad de implementar esta modalidad depende fuertemente de la arquitectura del motor de juego y un juego bien diseñado debería permitir realizar los cambios fácilmente.

Finalmente, pueden utilizarse técnicas de GPGPU (General-Purpose Computing on Graphics Processing Units) para acelerar el pipeline utilizando placas de video. Esta línea de desarrollo requiere un esfuerzo mayor para portar el código que el uso de la Xeon Phi, dado que la arquitectura de las placas de video es muy diferente a x86, mientras que la arquitectura MIC de la Xeon Phi es compatible con x86. Sin embargo, el uso de GPGPU tiene como ventaja que la mayor parte de los equipos comerciales y personales actuales disponen de placas de video preparadas para realizar tareas de GPGPU.

Bibliografía

E. Alba y M. Tomassini. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, págs. 443–462, 2002.

P. Angeline, G. Saunders, y J. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, págs. 54–65, 1994.

J. Baker. Reducing bias and inefficiency in the selection algorithm. En *Proceedings of the Second International Conference on Genetic Algorithms*, págs. 14–21, 1987.

M. Booth. The AI Systems of Left 4 Dead. URL `http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf`. Accedido en: 14/03/2017.

N. Cole, S. Louis, y C. Miles. Using a genetic algorithm to tune first-person shooter bots. En *Congress on Evolutionary Computation*, págs. 139–145, 2004.

D. W. Dyer. Watchmaker Framework for Evolutionary Computation. URL `http://watchmaker.uncommons.org/`. Accedido en: 14/03/2017.

FCEUX Community. The all in one NES/Famicom Emulator. URL `http://www.fceux.com/web/home.html`. Accedido en: 14/03/2017.

Games Done Quick. Games Done Quick: Speedrunning Marathons for Charity. URL `https://gamesdonequick.com/`. Accedido en: 14/03/2017.

X. Glorot y Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. En *International Conference on Artificial Intelligence and Statistics*, págs. 249–256, Mayo 2010.

D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Artificial Intelligence. Addison-Wesley Publishing Company, 1989.

D. Goldberg y J. Richardson. Genetic algorithms with sharing for multimodal function optimization. En *Genetic algorithms and their applications: Second International Conference on Genetic Algorithms*, págs. 41–49, 1987.

P. Hart, N. Nilsson, y B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, págs. 100–107, 1968.

M. Hausknecht, J. Lehman, R. Miikkulainen, y P. Stone. A neuroevolution approach to general Atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, págs. 355–366, 2014.

K. He, X. Zhang, S. Ren, y J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. En *IEEE International Conference on Computer Vision*, págs. 1026–1034, 2015.

S. Hochreiter y J. Schmidhuber. Long short-term memory. *Neural computation*, págs. 1735–1780, 1997.

J.-H. Hong y S.-B. Cho. Evolution of emergent behaviors for shooting game characters in Robocode. En *Congress on Evolutionary Computation*, págs. 634–638, 2004.

H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks-with an erratum note. *German National Research Center for Information Technology Technical Report*, pag. 34, 2001.

L. Jørgensen y T. Sandberg. Playing Mario using advanced AI techniques. *Advanced AI in Games, IT University of Copenhagen*, 2009.

B. Karlik y A. Olgac. Performance analysis of various activation functions in generalized MLP architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, págs. 111–122, 2011.

V. Khare, X. Yao, y K. Deb. Performance scaling of multi-objective evolutionary algorithms. En *International Conference on Evolutionary Multi-Criterion Optimization*, págs. 376–390. Springer, 2003.

C. Khaw. Assassin's Creed may finally stop being an annual franchise. URL `http://arstechnica.co.uk/gaming/2016/02/assassins-creed-may-finally-stop-being-an-annual-franchise/`. Accedido en: 14/03/2017.

P. Leong, G. Zhang, D.-U. Lee, W. Luk, y J. Villasenor. A Comment on the Implementation of the Ziggurat method. *Journal of Statistical Software*, págs. 1–4, 2005.

G. Luque y E. Alba. *Parallel genetic algorithms: theory and real world applications*. Springer, 2011.

G. Marsaglia y W. Tsang. The ziggurat method for generating random variables. *Journal of statistical software*, págs. 1–7, 2000.

T. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1997.

T. Murphy. The first level of Super Mario Bros. is easy with lexicographic orderings and time travel. *seventh Annual SIGBOVIK Conference*, págs. 112–133, 2013.

H. Newman. The Talos Principle underwent 15,000 hours of playtesting—but not by humans. URL `http://tinyurl.com/aitesting/`. Accedido en: 14/03/2017.

OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.0, Mayo 2008. URL `http://www.openmp.org/mp-documents/spec30.pdf`.

F. Parodi, S. Rodríguez, S. Iturriaga, y S. Nesmachnow. Optimizing a pinball computer player using evolutionary algorithms. En *XVIII Latin-Iberoamerican Conference on Operations Research*, págs. 649–656, 2016.

N. Radcliffe. Genetic set recombination and its application to neural network topology optimisation. *Neural Computing & Applications*, págs. 67–90, 1993.

S. Rodríguez, F. Parodi, S. Nesmachnow, y E. Mocskos. *Evaluation of a Master-Slave Parallel Evolutionary Algorithm Applied to Artificial Intelligence for Games in the Xeon-Phi Many-Core Platform*, En *Communications in Computer and Information Science*, págs. 161–176. 2016.

F. Röken y D. Frommhold. Automated Tests and Continuous Integration in Game Projects. URL `http://www.gamasutra.com/view/feature/2269/automated_tests_and_continuous_.php`. Accedido en: 14/03/2017.

J. Schaffer, D. Whitley, y L. Eshelman. Combinations of genetic algorithms and neural networks: a survey of the state of the art. En *International Workshop on Combinations of Genetic Algorithms and Neural Networks*, págs. 1–37, Jun 1992. doi: 10.1109/COGANN.1992.273950.

R. Simpson. Evolutionary Artificial Intelligence in Video Games. *University of Minnesota*, 2012.

K. Stanley y R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, págs. 99–127, 2002.

I. Sutskever. *Training recurrent neural networks.* PhD thesis, University of Toronto, 2013.

D. Takahashi. PwC: Game industry to grow nearly 5 % annually through 2020. URL `http://venturebeat.com/2016/06/08/the-u-s-and-global-game-industries-will-grow-a-healthy-amount-by-2020-pwc-forecasts/`. Accedido en: 14/03/2017.

The SCons Fundation. Scons: A software construction tool, 2016. URL `http://www.scons.org/`. Accedido en: 14/03/2017.

UberNES. UberNES Leaderboards - High Scores. URL `http://www.ubernes.com/current-nes-high-scores.html#Pinball`. Accedido en: 14/03/2017.

Valgrind Developers. Valgrind Home. URL `http://valgrind.org/`. Accedido en: 14/03/2017.

S. Vigna. Further scramblings of marsaglia's xorshift generators. *Journal of Computational and Applied Mathematics*, págs. 175–181, 2017.

C. Watkins. *Learning from delayed rewards.* PhD thesis, University of Cambridge England, 1989.

D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, págs. 65–85, 1994.

G. Yannakakis y J. Togelius. A panorama of artificial and computational intelligence in games. *IEEE Transactions on Computational Intelligence and AI in Games*, págs. 317–335, 2015.

X. Yao y Y. Liu. Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, págs. 83–90, 1998.