Universitat de les Illes Balears

escola politècnica superior

Treball Final de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Técnicas de Inteligencia Artificial en videojuegos automovilísticos

MARIANO LEDO GONZÁLEZ

Tutor
José María Buades Rubio

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, 11 de septiembre de 2018

ÍNDICE GENERAL

ACRÓNIMOS

NPC Personaje no jugador

FSM Máquina de estados finitos

SGD Descenso de Gradiente Stocástico

ANN Red Neuronal Artificial

CAPÍTULO 1

INTRODUCCIÓN

Existen muchos campos de aplicación para la inteligencia artificial. La mayoría de ellos persiguen mejorar los resultados de diferentes cálculos complejos. Sin embargo, el campo de los videojuegos resulta muy interesante puesto que la inteligencia que se busca no consiste en mejorar cálculos específicos, sino que se pretende mejorar la experiencia de juego mediante la creación de un agente, cuyo comportamiento sea lo más parecido posible al humano.

1.1 Historia

Los primeros juegos en los que se aplicaron los estudios sobre inteligencia artificial fueron los juegos de mesa como el ajedrez, las damas o Go. La simpleza de sus reglas los hacen muy fáciles de modelar e implementar. Además, aunque estas reglas son bastante simples, estos juegos presentan una inmensa complejidad a la hora de analizar las partidas y generar un sistema que sea capaz de jugar a ellos de forma autónoma y correcta; esto los hace ideales para probar técnicas de inteligencia artificial.

Una de las técnicas más utilizadas en esta clase de juegos por turnos es el algoritmo **Minimax**, que consiste en que el agente explora el árbol de jugadas posibles, de manera que analiza las diferentes situaciones futuras que pueden darse y elige aquella que sea más conveniente para él. Este es el algoritmo que utilizó la máquina **Deep Blue**, fabricada por *IBM*, para vencer al famoso campeón de ajedrez **Garry Kasparov** en 1997 [1].

Por otro lado, el interés de la inteligencia artificial en juegos digitales apareció prácticamente desde la invención de estos. El principal uso de las técnicas de inteligencia artificial en estos juegos es el modelado del comportamiento de los **Personajes no jugadores** (NPC). Sin embargo, los primeros intentos para dotar a la máquina de capacidad para jugar de forma autónoma consistieron en implementar en el propio

código el comportamiento de los agentes. Ejemplo de esto son los alienígenas del famoso **Space Invaders** (1978) o el oponente en el famoso **Pong** (1972). El problema de este método es que los comportamientos son muy predecibles y monótonos. Entre las técnicas de inteligencia artificial aplicadas para resolver este problema se encuentran las **Máquinas de estados finitos** (FSM) [2] y los **algoritmos de búsqueda de caminos**, como el conocido como A^* (A estrella) [3].

1.2 Objetivo

Una de las principales características que hacen que un videojuego sea entretenido es la variedad. Aumentar las posibilidades mediante la creación de contenido que sea dinámico es la clave para mantener al jugador atento e interesado en el juego. En el caso de los **videojuegos de carreras**, esto se consigue principalmente con un conjunto de circuitos de diseños muy variados, vehículos con diferentes prestaciones y oponentes con comportamientos también diversos y poco predecibles.

El principal problema que se presenta a la hora de conseguir esta variedad reside en que un gran número de circuitos con diferentes características requiere de un gran esfuerzo por parte de los desarrolladores, que deben implementar el comportamiento de los oponentes para cada uno de estos circuitos. Además, los juegos en los que el circuito no está definido como una única carretera hacia la meta, sino que presentan múltiples caminos que llevan a ella, conforman un problema mayor, puesto que es necesario dotar a los oponentes de cierta libertad de decisión para evitar un comportamiento predecible y monótono.

Es aquí donde la inteligencia artificial resulta útil, puesto que si es posible crear un sistema que pueda entrenar a los vehículos de forma automática independientemente del entorno, el coste de desarrollo se reduciría de forma drástica. Además, un sistema así podría permitir al videojuego aprender del jugador y ajustar el comportamiento de los oponentes para ofrecer una experiencia más personalizada.

En este estudio se aplicarán diversos métodos de aprendizaje de máquinas para comprobar qué tan efectivos pueden resultar a la hora de afrontar el problema mencionado.

CAPÍTULO 2

ENTORNO

Antes de exponer la solución planteada es necesario exponer las consideraciones que se han tenido en cuenta a la hora de definir las características del entorno y los vehículos.

Para centrar la atención en el desarrollo de la inteligencia artificial y evitar añadir complejidad por detalles de realismo, el entorno de la simulación estará formado por un mundo en 2 dimensiones.

2.1 El vehículo

El cuerpo del vehículo está compuesto por un rectángulo. La posición de este en el espacio viene dada por un punto en coordenadas cartesianas (x, y) y se sitúa en la parte posterior central del vehículo. De esta manera dicha posición se puede utilizar como centro de rotación (ya que las ruedas traseras son fijas).

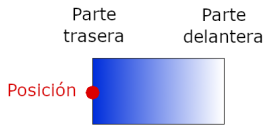


Figura 2.1: Vehículo y posición

La rotación de las ruedas se especifica en radianes: si el valor de la rotación es 0, las ruedas se encuentran centradas; si el valor es negativo, el coche gira hacia la derecha; y

por tanto, si es positivo, gira hacia la izquierda. La amplitud del ángulo de giro está se establecerá en 45 grados hacia cada lado. Para cambiar la orientación de las ruedas se deberá especificar un factor de rotación en el rango $[0,1]$, donde 0 orientará las ruedas totalmente hacia la derecha y 1 orientará las ruedas totalmente hacia la izquierda.

En este primer planteamiento se centrará la atención en generar un agente capaz de controlar la dirección del vehículo. Por ello, el aspecto de la velocidad no se tendrá en cuenta y el vehículo avanzará a una velocidad constante.

Para que el agente pueda tomar decisiones es necesario que tenga datos sobre el entorno. Por este motivo, dispone de n sensores que permiten detectar la distancia a la que se encuentra un obstáculo.

Estos sensores se disponen de forma que cada uno de ellos lanza un haz en un determinado ángulo en base a la posición del vehículo, tal como se muestra en la figura 2.2.

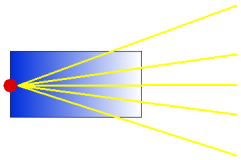


Figura 2.2: Sensores del vehículo

2.2 El circuito

Para entrenar al agente se utilizarán una serie de circuitos de diferentes características. Estos se tratarán de circuitos cerrados, cuya trazada estará definida por paredes que el vehículo podrá detectar mediante los sensores. Además, los circuitos variarán en longitud y pueden contener segmentos en los que el el camino se hace más estrecho o más ancho.

Uno de los principales aspectos del entrenamiento es el análisis del desempeño del agente, por lo que es necesario establecer unas medidas para observar y comparar dicho desempeño. La principal medida que se utilizará es la distancia recorrida por el vehículo sin estrellarse contra ningún obstáculo.

Esto plantea un problema, pues si se mide únicamente la distancia que el coche avanza se pueden obtener resultados confusos, ya que el vehículo podría quedarse dando vueltas en un punto o ir en dirección contraria. Una solución a este problema consiste en realizar el cálculo de la distancia recorrida en base a la posición del vehículo en el circuito. Sin embargo, aparece un nuevo problema, y es que puede interesar que el vehículo de múltiples vueltas al circuito, por lo que no es suficiente calcular la distancia

entre el punto inicial y la posición del vehículo, sino que es necesario mantener un histórico del espacio recorrido en el circuito.

Añadido a todo lo anterior, a la hora de comparar el desempeño de dos agentes, surge el problema de la trazada que realizan estos dentro del circuito. Esto pasa porque es posible que, aunque dos vehículos se encuentren a la misma 'altura' del circuito, uno haya realizado una trazada mayor longitud, tal como se puede observar en la figura 2.3.

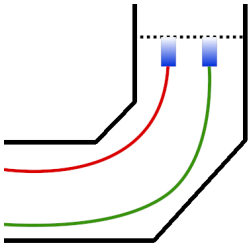
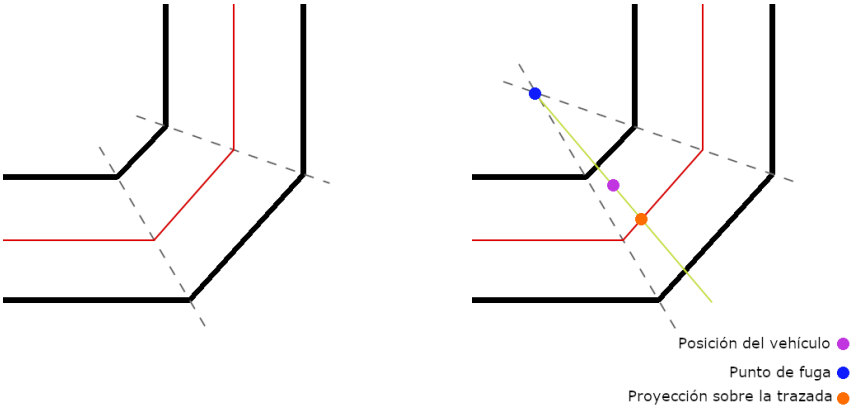


Figura 2.3: Diferencia de longitud entre trazadas

Por lo tanto, es necesario definir una medida estándar que resuelva estos casos. La solución planteada consiste en dividir el circuito en segmentos rectilíneos, obtener la línea central de dicho segmento y proyectar la posición del vehículo sobre dicha línea. De esta manera, el cálculo de la distancia recorrida se realiza siempre sobre la trazada central del circuito, obviando las diferencias entre trazadas antes mencionadas, ya que, en esta aproximación, se dará más importancia al hecho de llegar más lejos que al hecho de hacerlo en menos tiempo. Así, para un circuito como el de la figura 2.3, la trazada que se toma en cuenta es la que aparece en la figura 2.4a.



(a) Trazada normalizada (b) Proyección sobre la trazada

Figura 2.4: Normalización de la trazada

Para realizar la proyección de la posición del vehículo sobre la trazada se aprovechará el hecho de que el circuito es cerrado. Este hecho provoca que todo circuito contendrá alguna curva, de manera que, como se puede observar en la figura 2.4a, los

segmentos que componen el circuito tienen forma trapezoidal. Gracias a esta forma, es posible tomar los segmentos no paralelos del trapecio y utilizar el punto donde se intersecan como punto de fuga. De esta manera, prolongando el segmento formado por dicho punto de fuga y la posición del vehículo se obtiene una recta que corta a la trazada en un punto (ver figura 2.4b).

De esta manera, el cálculo de la distancia recorrida por un vehículo se realiza mediante un seguimiento del segmento del circuito en el que se encuentra éste en todo momento y un histórico de la longitud de los segmentos por los que ha pasado. Si el vehículo avanza del segmento i al $i+1$ se suma la longitud del segmento i al histórico y, de forma similar, si le vehículo retrocede del segmento i al $i-1$ se resta la longitud del segmento i del histórico. Para que esto funcione en el caso de múltiples vueltas al circuito, se establece que para el segmento final, el segmento $i+1$ es el segmento inicial. Así, la distancia total recorrida por un vehículo se obtiene mediante la fórmula

$$D = h + d \tag{2.1}$$

donde h es el valor del histórico y d es la distancia recorrida en el segmento actual.

CAPÍTULO 3

ENTRENAMIENTO

En este estudio se analizan dos técnicas de aprendizaje de máquinas muy utilizadas. La primera se trata de un **algoritmo genético** y la segunda hace uso de la técnica conocida como **Descenso de Gradiente Stocástico (SGD)**.

Ambas técnicas tienen en una característica en común, y es que la base de ambas es una **Red Neuronal Artificial (ANN)**. Por ello, antes de profundizar en dichas técnicas, es conveniente exponer los aspectos básicos de este sistema.

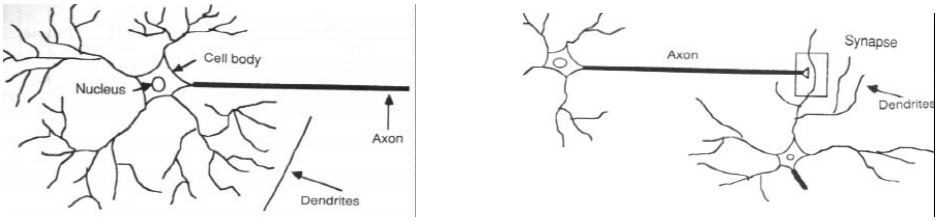
3.1 Redes neuronales

3.1.1 ¿Qué es una red neuronal?

Si bien es cierto que no se tiene un conocimiento exacto del funcionamiento de los cerebros biológicos, las ANNs se inspiran en los procesos que se conoce que tienen lugar en ellos. Al igual que los cerebros biológicos, las ANNs aprenden a partir de ejemplos, mediante la modificación de parámetros en su estructura, siendo capaces de llegar a reconocer patrones y clasificar datos.

3.1.2 Estructura de una red neuronal

Los cerebros biológicos se componen de células conocidas como **neuronas**, que reciben información a través de estructuras llamadas *dendritas*. Las neuronas envían información en forma de impulsos eléctricos a través de otra estructura que recibe el nombre de *axón*, el cual se conecta a las dendritas de otras neuronas mediante una estructura llamada *sinapsis*. Dicha sinapsis convierte la actividad del axón en efectos que inhiben o excitan a las neuronas a las que está conectado. De esta manera, el proceso de aprendizaje consiste en modificar la efectividad de las sinapsis para variar la influencia que tienen unas neuronas sobre otras [4].



(a) Componentes de una neurona (b) La sinapsis

Figura 3.1: Las neuronas del cerebro humano. Recuperada de [4]

Basándose en esta estructura, las ANNs se componen de una conjunto de 'neuronas' conocidas por el nombre de **perceptrones**. Siguiendo el concepto de las dendritas y los axones, los perceptrones básicos reciben un conjunto de entradas binarias y producen una única salida, también binaria.

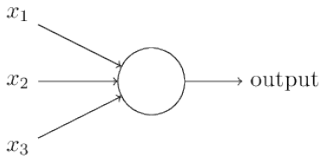


Figura 3.2: Perceptrón. Recuperada de [5, Capítulo 1]

El concepto de perceptrón fue desarrollado por **Frank Rosenblatt** en la década de los 50 [6], quien introdujo la idea de asociar pesos a cada una de las entradas de un perceptrón, los cuales expresan la influencia de cada una de las entradas sobre la salida del mismo (de forma similar a la sinapsis en los cerebros biológicos). De esta manera, el valor de la salida del perceptrón se determina realizando la suma de cada una de las entradas ponderadas por su peso, y comprobando que el resultado obtenido es mayor o menor que un determinado umbral.

$$\text{salida} = \begin{cases} 0 & \text{si } \sum_j w_j x_j \leqslant \text{umbral} \\ 1 & \text{si } \sum_j w_j x_j > \text{umbral} \end{cases} \tag{3.1}$$

La estructura en forma de red se consigue a partir de la combinación de múltiples perceptrones. Como se puede observar en la figura 3.3, la red se estructura en forma de **capas**, de manera que cada una de las entradas de la red se convierten en las entradas de un conjunto de perceptrones que constituye la primera capa y, a su vez, la salida de estos se convierte en las entradas de los perceptrones de la siguiente capa.

De esta manera, tal como expone **M. Nielsen** en su libro [5], los perceptrones de la segunda capa toman decisiones sopesando las decisiones realizadas por la primera, de forma que éstos de la segunda capa son capaces de tomar decisiones a un nivel de abstracción y complejidad mayor que aquellos de la primera.

Como se puede observar en el libro de **M. Nielsen** [5], es posible simplificar la ecuación que describe a un perceptrón mediante notación vectrial. Para ello, primero

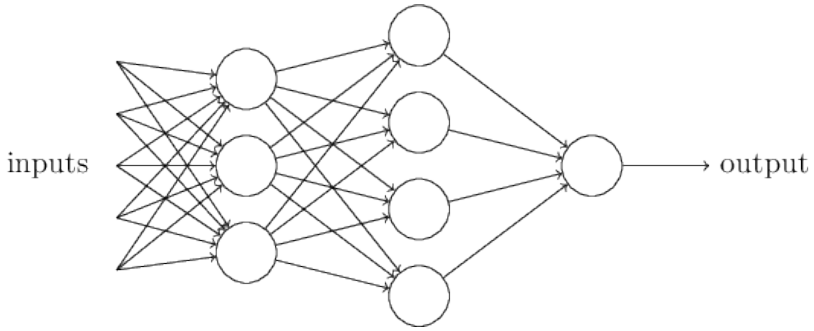


Figura 3.3: Estructura en capas. Recuperada de [5, Capítulo 1]

se reescribirá la expresión $\sum_j w_j x_j$ como un producto vectorial $w \cdot x$, donde w y x son vectores que contienen los pesos y las entradas respectivamente. A continuación, se moverá el umbral al otro lado de la in-ecuación y se reescribirá como lo que se conoce como el **bias** del perceptrón, $b \equiv -$umbral. De esta forma, la fórmula vista anteriormente 3.1 se puede expresar de la siguiente forma:

$$\text{salida} = \begin{cases} 0 & \text{si } w \cdot x + b \leqslant 0 \\ 1 & \text{si } w \cdot x + b > 0 \end{cases} \tag{3.2}$$

3.1.3 Función de transferencia

Mediante el uso de los perceptrones expuestos anteriormente es posible entrenar redes neuronales para resolver diferentes problemas. Sin embargo, el hecho de que la salida de estos perceptrones sea binaria provoca que un cambio leve en la configuración de los pesos y umbrales puede provocar un cambio considerable en la salida, como es el hecho de que pase se ser 0 a 1.

Para resolver este problema se dispone de diferentes funciones que producen diferentes tipos de salida. A esta función encargada de producir la salida final del perceptrón en base al umbral, la entrada y los pesos se le conoce como **función de transferencia**. La función 3.2 se conoce como **función escalonada**.

Otra función común es la **función identidad** o **función lineal**. En este caso la salida del perceptrón es exactamente el resultado de la expresión $w \cdot x + b$.

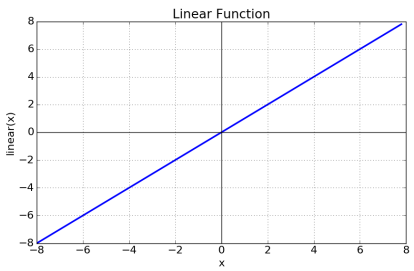


Figura 3.4: Función lineal. Recuperada de [7]

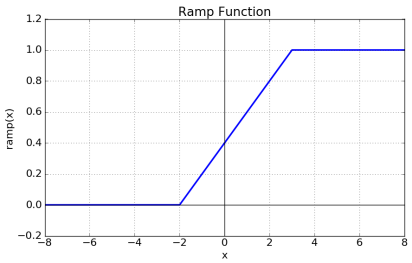


Figura 3.5: Función rampa. Recuperada de [7]

La **función rampa** es una combinación de la función escalonada y la función lineal, de manera que el paso del valor mínimo al máximo se produce de forma lineal en lugar de forma abrupta. Tomando la expresión $w \cdot x + b$ como z, y los valores T_1 y T_2 como los puntos que determinan el intervalo lineal de la función, ésta se puede formular de la siguiente manera:

$$\text{salida} = \begin{cases} 0 & \text{si } z < T_1 \\ \dfrac{(z - T_1)}{T_2 - T_1} & \text{si } T_1 \leqslant z \leqslant T_2 \\ 1 & \text{si } z > T_2 \end{cases} \tag{3.3}$$

Sin embargo, la función seleccionada para este estudio es la **función sigmoidal**. Esta función es similar a la función rampa pero con una derivada mucho más suavizada. Esta característica, sumada al hecho de que su fórmula (y la de su derivada) es más sencilla de implementar, la convierte en una elección muy conveniente para uno de los sistemas de entrenamiento que se explicarán más adelante. Esta función se define de la siguiente forma:

$$\text{salida} = \frac{1}{1 + e^z} \tag{3.4}$$

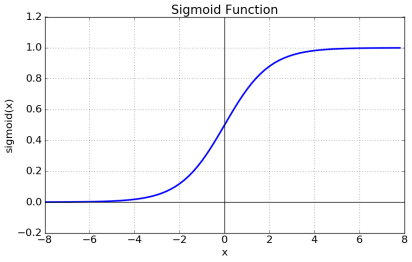


Figura 3.6: Función sigmoidal. Recuperada de [7]

Además de todas estas funciones, existen otras con características diferentes [7]. La elección de cual de ellas utilizar dependerá siempre de los requerimientos del problema.

3.1.4 Aplicación al problema

El diseño de la red neuronal utilizado en este estudio es el siguiente.

- **Entradas**. Como entradas de la red neuronal se utilizarán los valores de cada una de los sensores del vehículo y la orientación actual de las ruedas en radianes.
- **Capas internas**. La elección del número de capas internas y del número de perceptrones en cada capa es la más compleja. Para este estudio se utilizarán diferentes combinaciones y se discutirán los resultados más adelante.
- **Salidas**. Puesto que el objetivo de este estudio es entrenar el sistema de dirección, la red neuronal solo producirá un valor en el rango $[0,1]$ que se utilizará como factor de orientación de las ruedas.

3.2 Algoritmo genético

3.2.1 ¿Qué es un algoritmo genético?

Los algoritmos genéticos se inspiran en el proceso de **selección natural** propuesto por **Charles Darwin** [8]. En este proceso los individuos que muestran unas características más óptimas son los elegidos para reproducirse y propagar sus características a las generaciones siguientes.

Un algoritmo genético consta de 5 fases:

1. **Inicialización**. El primer paso consiste en generar la población de individuos con características seleccionadas de forma aleatoria.
2. **Evaluación**. En esta fase se calcula lo óptimos que son los individuos para la resolución del problema.
3. **Selección**. Una vez evaluados los individuos se deben seleccionar aquellos que propagarán sus características a la siguiente generación.
4. **Cruzamiento**. Esta es la fase de reproducción. Se combinan las características de los individuos seleccionados para producir la siguiente generación.
5. **Mutación**. Una vez generada la siguiente generación, se les aplican mutaciones aleatorias a los individuos para evitar que la evolución tienda a una configuración de características que no sea demasiado óptima y quede encallada.

3.2.2 Aplicación al problema

En el caso de este estudio se deben definir los siguientes parámetros:

- La **función de aptitud** utilizada para calcular el grado de utilidad de un individuo.

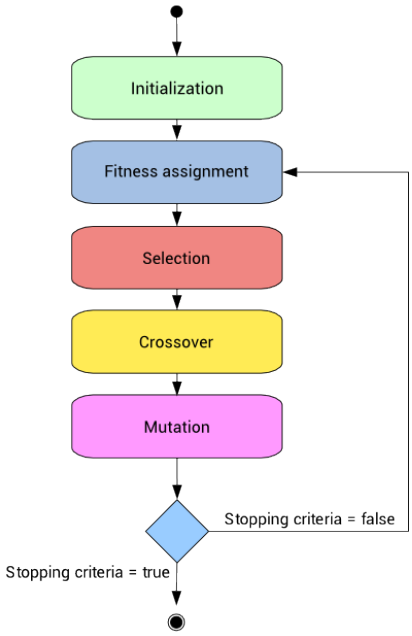


Figura 3.7: Diagrama de estados para un algoritmo genético. Recuperada de [9]

- El **criterio de selección** de los individuos más óptimos.
- El **método de reproducción** de los individuos.
- El **sistema de mutación** utilizado.

En este estudio nos centraremos en que el agente sea capaz de avanzar la mayor distancia posible sin impactar con ningún obstáculo, por lo que el grado de aptitud de un individuo se medirá en base a la D (ver 2.1) del vehículo cuya D sea mayor. Renombrando esta distancia como D_{max}, la **función de aptitud** para un vehículo i es la siguiente

$$Aptitud = \frac{D_i}{D_{max}} \tag{3.5}$$

Mediante 3.5, se define la aptitud de un individuo como un peso, de forma que el vehículo más apto (el que ha avanzado mayor distancia) tiene un peso de 1. Así, para el **criterio de selección**, se realiza una selección aleatoria de n individuos (donde n es el número de individuos que hay en la población) de forma que aquellos que tienen mayor aptitud tienen más posibilidades de ser elegidos múltiples veces. De esta manera, se mantiene la dimensión de la población y aquellos individuos más aptos serán elegidos múltiples veces, generando más descendientes. Cabe destacar, que el individuo más apto siempre es seleccionado, por lo que la selección aleatoria realmente es de $n-1$ individuos. De esta manera, se evita que la evolución degenere por casos en que la aleatoriedad provoca que el individuo más apto no sea seleccionado.

El **método de reproducción** resulta útil en problemas donde los individuos tienen características o 'genes' bien diferenciados, como puede ser que dos individuos utilicen diferentes entradas para sus ANNs. Un ejemplo de esto son los problemas de selección de características para generar modelos predictivos (Ver [9]). En el caso de este estudio, todos los individuos utilizan las mismas entradas y solo varían los pesos y umbrales de la ANN, por lo que es suficiente con la fase de mutación.

El **sistema de mutación** seleccionado simplemente consiste en aplicar un ruido gaussiano a cada uno de los pesos y umbrales que componen la ANN de los individuos seleccionados. Una vez más, para evitar degeneraciones por la aleatoriedad, el individuo más apto, que fue seleccionado automáticamente, no se le aplica ninguna mutación y pasa a la siguiente generación intacto. Sin embargo, si este individuo fue nuevamente seleccionado en el proceso aleatorio, los descendientes generados por esas selecciones sí serán mutados.

Finalmente, la implementación de este método de entrenamiento consta de cuatro fases:

1. **Inicialización**. Se generan n vehículos controlados por ANNs cuyos pesos y umbrales han sido seleccionados de forma aleatoria.
2. **Evaluación**. Se ejecuta una simulación en la que participan los n vehículos. La simulación finaliza cuando todos los vehículos han impactado con un obstáculo o uno de ellos ha completado una vuelta al circuito.
3. **Evolución**. Se realiza el proceso de selección y mutación de los individuos.
4. Si algún individuo ha completado una vuelta al circuito, se finaliza el entrenamiento. En caso contrario, se continúa a la fase 2.

3.3 Descenso de Gradiente Stocástico (SGD)

3.3.1 ¿En qué consiste?

Este método se basa en el aprendizaje a partir de ejemplos, es decir, a partir de un conjunto de datos que contiene la salida esperada para un determinado conjunto de entradas de la ANN, ésta se entrena para que sea capaz de producir la salida esperada en el mayor número de casos posible.

La explicación expuesta a continuación sobre el funcionamiento de este método es un resumen extraído del libro de M. Nielsen [5, Capítulo 1].

Un elemento del conjunto de entrenamiento está formado por el conjunto de entradas: un vector n-dimensional denotado como x; y el conjunto de salidas esperadas: un vector m-dimensional denotado como $y = y(x)$.

El objetivo del algoritmo es obtener una configuración de pesos y umbrales de manera que la salida de la ANN se aproxime lo máximo posible a la salida esperada.

Para ello, es necesario tener una función que evalúe la calidad de la salida. Esta función se nombrará como **función de coste** y se define como

$$C(w,b) \equiv \frac{1}{2n}\sum_x \|y(x)-a\|^2 \tag{3.6}$$

Donde w es el conjunto de pesos de la red, b es el conjunto de umbrales, n el número de elementos del conjunto de entrenamiento y a es la salida que produce la red cuando x es la entrada.

La meta del algoritmo es minimizar esta función de coste, esto es, encontrar el mínimo global de la función. Suponiendo que se tiene una función $C(v)$, compuesta por dos variables v_1 y v_2, como la siguiente

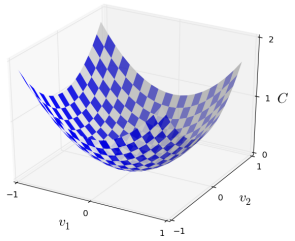


Figura 3.8: Recuperada de [5]

Un método que permite encaminar la búsqueda de dicho mínimo es aquel que toma la función como si se tratase de un valle y, colocando una bola en un punto aleatorio de este, se dejara caer hasta que alcance el fondo del valle. A partir de esta interesante analogía, el objetivo es tomar un punto aleatorio de la función y simular el movimiento descendiente de la bola, lo cual es posible conseguir mediante el cálculo de derivadas sobre la función.

Si se mueve el punto una pequeña cantidad Δv_1 en la dirección v_1 y una pequeña cantidad Δv_2 en la dirección v_2, la variación en C se puede definir como

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 \tag{3.7}$$

Para encontrar el mínimo se debe elegir Δv_1 y Δv_2 de manera que ΔC sea negativo. Antes, es conveniente reescribir esta ecuación definiendo Δv como un vector de cambios $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$, donde T es la operación de matriz traspuesta. También se definirá el **gradiente de C** como el vector de derivadas parciales

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T \tag{3.8}$$

De esta manera, la ecuación 3.7 pasa a formularse como

$$\Delta C \approx \nabla C \cdot \Delta v \tag{3.9}$$

Ahora es posible seleccionar Δv de forma que ΔC sea negativo. En particular, si se toma

$$\Delta v = -\eta \nabla C \tag{3.10}$$

Donde η es un pequeño valor positivo conocido como *factor de aprendizaje*. La ecuación 3.9 muestra que $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$. Puesto que $\|\nabla C\|^2 \geq 0$, se puede afirmar que ΔC es siempre negativo. De esta manera, mediante la ecuación 3.10 se calculará el valor de Δv y se moverá el punto v en esa cantidad

$$v \longrightarrow v' = v - \eta \nabla C \tag{3.11}$$

Aplicando de forma repetida esta regla es posible reducir C hasta llegar a un mínimo.

En esta explicación se ha utilizado una función de 2 variables. En el caso de una función C de n variables simplemente se deben tener en cuenta todas estas variables a la hora de definir los vectores $\Delta v = (\Delta v_1, \Delta v_2, ..., \Delta v_n)^T$ y $\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, ..., \frac{\partial C}{\partial v_n} \right)^T$.

Cabe destacar que, debido a que el punto inicialmente se toma de forma aleatoria, es muy probable que el algoritmo llegue a un mínimo local antes que al mínimo global. Además, puesto que estas reglas provocan que el punto siempre descienda, si el algoritmo llega a un mínimo local, no tendrá forma de continuar.

3.3.2 Aplicación al problema

Para aplicar este método al problema es necesario generar los datos a partir de los cuales se pueda entrenar la ANN. El mejor candidato para esta tarea es una red neuronal ya entrenada, es decir, una persona. Para ello, se pedirá a un usuario que controle a un vehículo de forma manual y realice diferentes vueltas al circuito con el fin de recopilar los datos de entrenamiento.

El algoritmo SGD requiere un criterio para saber cuando detenerse. El método más sencillo es definir un número n de repeticiones, de manera que el algoritmo aplicará las reglas n veces y se detendrá. También es posible aplicar métodos más sofisticados como detener el algoritmo en cuanto C se haya reducido hasta un valor determinado. Sin embargo, la primera opción es más sencilla de implementar y proporciona unos resultados muy aceptables.

CAPÍTULO 4

RESULTADOS

4.0.1 Red neuronal

Después de probar ambos métodos con diferentes diseños para las capas internas de la red, se ha observado que diseños más simples, como 1 única capa interna de 10 perceptrones, produce resultados igual de buenos que otros diseños más complejos.

4.0.2 Algoritmo genético

El proceso de entrenamiento con este método ha consistido en entrenar un agente en un circuito y, una vez entrenado, comprobar que es capaz de dar 2 vueltas a cada uno de los circuitos restantes. Los resultados obtenidos son los siguientes (Se aconseja ver la sección 4.1 para visualizar los circuitos).

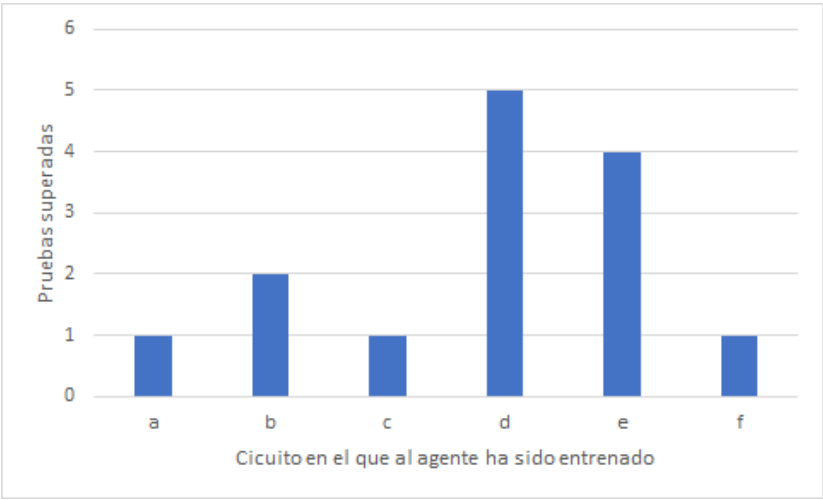


Figura 4.1: Pruebas superadas

Como se podía esperar, el entrenamiento en los circuitos *a* y *c* no ha producido buenos resultados, ya que se tratan de circuitos que solo presentan curvas a la izquierda. Similarmente, el circuito *b* no ha sido mucho más eficaz ya que, aunque presenta más curvas, sigue mostrando una complejidad muy baja. En este caso, el más efectivo ha sido el circuito *d* puesto que es el más complejo y presenta una mayor variedad de situaciones para el vehículo.

Otro aspecto que resalta bastante de este método es el rendimiento, ya que el número de generaciones que se ha necesitado para llevar a cabo los entrenamientos es realmente bajo; en el caso del circuito más complejo solo se han necesitado 12 generaciones y, en el caso de los circuitos más simples, la inicialización aleatoria ha sido suficiente para producir agentes capaces de completar el circuito.

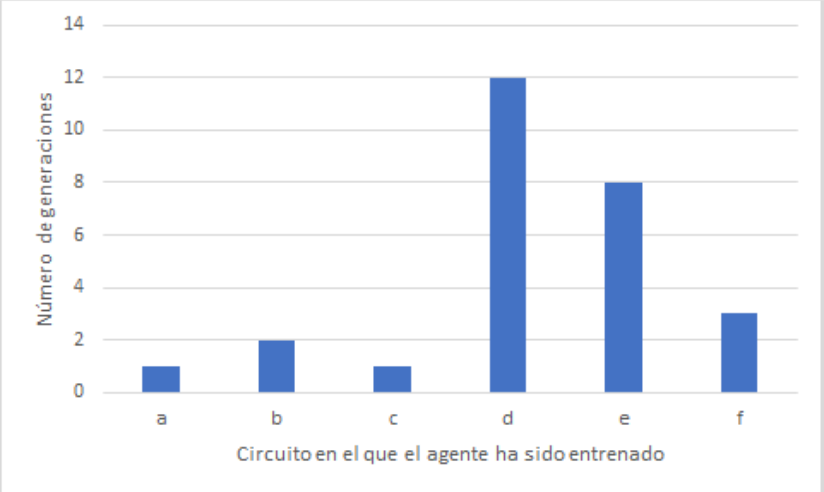


Figura 4.2: Número de generaciones necesario para completar el entrenamiento

4.0.3 SGD

Para llevar a cabo este método de entrenamiento, se han recopilado los datos de simulaciones manuales en todos los circuitos. A partir de estos datos se han generado diferentes conjuntos de entrenamiento: uno por cada circuito; se han combinado los datos de los circuitos 4.3a, 4.3b y 4.3c en un solo conjunto; y un único conjunto con todos los datos.

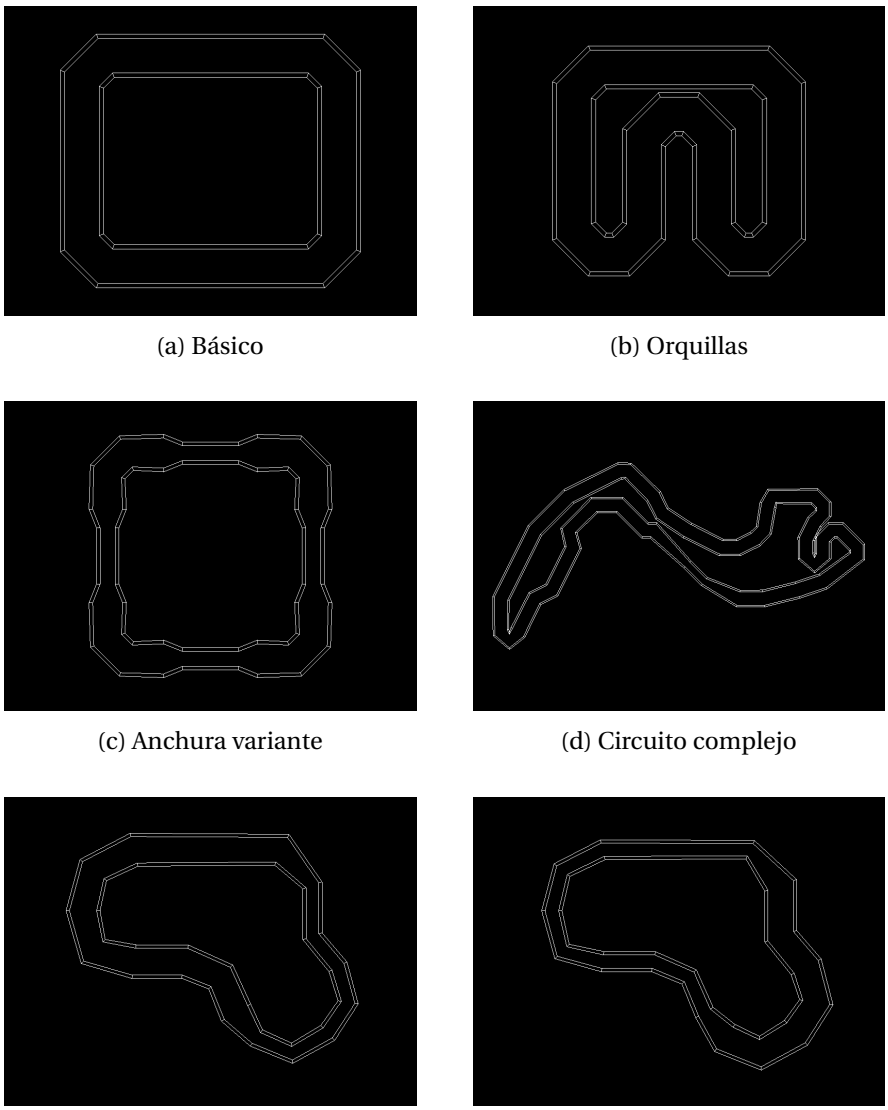
Para configurar el algoritmo SGD se han utilizado diferentes combinaciones con valores en el rango $[0'1, 2]$ para el factor de aprendizaje y $[1000, 10000]$ para el número de generaciones.

Sin embargo, aunque este método parecía muy prometedor, no ha producido ningún resultado satisfactorio. La hipótesis que se tiene es que, debido al hecho de que el control del vehículo por parte del usuario se realiza mediante el teclado, y los giros no se realizan con la tecla de dirección presionada constantemente, sino que se dan toques cortos para corregir la dirección de forma precisa, los datos generados pueden contener situaciones en las que las entradas son casi idénticas pero la salida es total-

mente diferente. Además, puesto que el usuario es capaz de seguir una trazada perfecta por el circuito, el conjunto de entrenamiento no tiene patrones de situaciones en los que el vehículo se aleje de esa trazada y, si se pide al usuario no seguir una trazada coherente para obtener datos de esas situaciones, entonces también se introducirán patrones no coherentes en el conjunto de entrenamiento que tampoco producirán resultados positivos.

4.1 Circuitos utilizados

A continuación se muestran los diferentes circuitos utilizados para probar estos métodos de entrenamiento.



(a) Básico (b) Orquillas

(c) Anchura variante (d) Circuito complejo

(e) Circuito ancho con tramo estrecho (f) Circuito estrecho con tramo ancho

Figura 4.3: Circuitos de entrenamiento

CAPÍTULO 5

CONCLUSIONES

Después de analizar los resultados obtenidos, se puede concluir que las ANNs, y el aprendizaje de máquinas en general, es una técnica bastante interesante y efectiva para implementar la inteligencia artificial en videojuegos automovilísticos. Siempre que se tengan los datos adecuados con los que entrenar estas redes.

En relación al diseño de ANNs, se ha observado que en este tipo de problemas parece ser más conveniente utilizar estructuras sencillas, ya que los resultados que se obtienen son bastante buenos a un coste computacional menor.

Finalmente, aunque el método SGD proporciona resultados bastante aceptables en problemas de clasificación y detección de patrones, este no parece ser el método más efectivo en el caso del problema tratado en este estudio.

CAPÍTULO 6

DESARROLLO

6.1 Tecnologías

Debido a la importante carga matemática a la hora de implementar los métodos de entrenamiento expuestos, el lenguaje seleccionado para el desarrollo del programa utilizado en la simulación y evaluación de dichos métodos ha sido **Python**. Este lenguaje presenta librerías y una sintaxis que permite expresar los cálculos y algoritmos matemáticos de forma muy simple.

En lo referente al apartado gráfico de la simulación, este se ha implementado en Python al igual que el resto del programa pero haciendo uso de las funcionalidades que pone a disposición la librería gráfica **OpenGL**.

Finalmente, para mantener un control sobre los cambios que se han ido realizando sobre el programa se ha hecho uso de la plataforma **Gitlab** (https://gitlab.com/).

6.2 Gestión del proyecto

Debido al falta de conocimiento sobre el tema ha sido complicado establecer objetivos y tiempos concretos. Por lo que se ha optado por seguir la línea de las metodologías ágiles que permiten reaccionar de forma rápida y efectiva a los cambios e imprevistos en la planificación. De esta manera, se ha hecho uso de la herramienta **Trello** (https://trello.com/) para establecer y gestionar las diferentes tareas y, basándose en el concepto de *sprint* expuesto en las guías de la metodología **Scrum** [10], se han agrupado las tareas en *sprints* de 1 semana de duración, de manera que al final de cada una ellas se haya conseguido un avance útil en el proyecto.

6.3 La simulación

El primer paso del desarrollo de la simulación ha consistido en llevar a cabo una investigación sobre diferentes patrones de diseño que permitan mantener los diferentes módulos del programa desacoplados. De esta manera, se facilita la tarea de añadir funcionalidades de forma progresiva y realizar pruebas aisladas sobre cada uno de estos módulos.

La arquitectura implementada sigue las líneas expuestas por *Robert C. Martin* en su libro ***Clean Architecture*** [11]. En la figura 6.1 se muestra el esquema general de esta arquitectura.

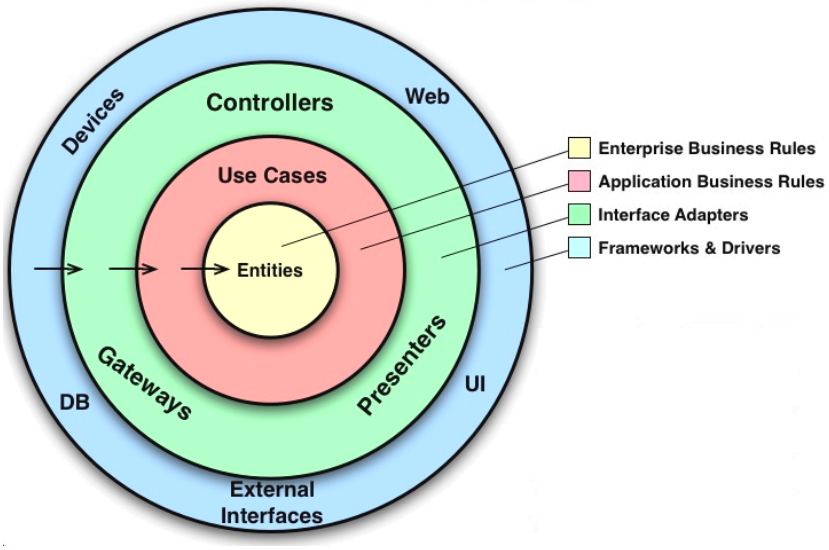


Figura 6.1: Diagrama Clean Architecture. Recuperada de [13]

Esta arquitectura establece una serie de capas alrededor de la lógica de negocio, de forma que ésta permanece al margen de los detalles de implementación, como librerías y frameworks utilizados. Además, como muestran las flechas de la figura 6.1, para asegurar la independencia de dicha lógica se establece una regla que consiste en que aquellos módulos solo pueden depender en otros que sean de capas más internas.

No obstante, la ventaja que provee esta regla de dependencias se hace más patente en lenguajes compilados y estructurados en librerías, ya que reduce el número de módulos a recompilar y distribuir cuando se añade una nueva funcionalidad. De esta manera, debido a que Python se trata de un lenguaje interpretado y este programa tiene una estructura monolítica, se ha obtado por ignorar esta regla, con el objetivo de evitar la gran cantidad de código extra que se genera al realizar la *inversión de dependencias* [11, pág. 87], necesaria para satisfacer la regla.

Con esto en mente, en la figura 6.2 se puede observar la estructura general del programa.

El módulo *Window* se encarga de abstraer la lógica relacionada con la librería

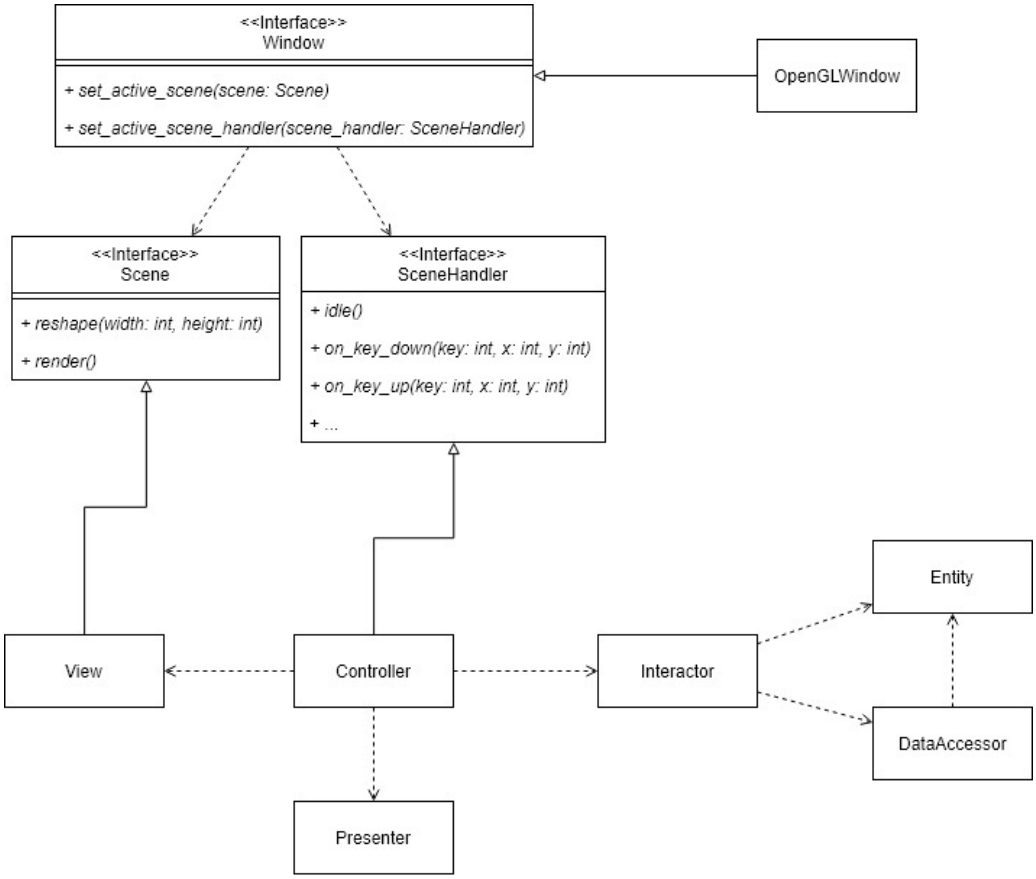


Figura 6.2: Estructura del programa

OpenGL. El módulo *Scene* permite abstraer toda la lógica de renderización de la interfaz de usuario, mientras que el módulo *SceneHandler* permite abstraer la lógica de respuesta a los eventos generados en dicha interfaz.

El módulo *View* contiene todos los detalles de renderización de la interfaz de usuario.

Los módulos de tipo *Entity* conforman la lógica de más alto nivel y, por tanto, no dependen de ningún módulo de otra capa. Estos módulos son los encargados de recoger los datos que representan entidades como una red neuronal, un coche o un circuito.

Los módulos de tipo *Interactor* contienen toda la lógica de procesamiento de las entidades, como es la lógica de conducción de un coche o la lógica de actualización del estado de la simulación.

Los módulos de tipo *DataAccessor* se encargan de abstraer toda la lógica relacionada con el almacenaje de las entidades. Permiten a los *interactors* recuperar entidades sin necesidad de conocer los detalles de cómo ni donde estan siendo almacenadas.

El módulo *Presenter* contiene la lógica necesaria para convertir las entidades en las estructuras que maneja la *vista*.

Finalmente, el módulo *Controller* contiene toda la lógica de coordinación de la simulación. Es el encargado de reaccionar a los eventos de la interfaz de usuario, realizar las acciones pertinentes mediante los *interactors*, formatear el resultado de las acciones mediante el *presenter* y enviar dichos resultados formateados a la vista para mostrarlos en la interfaz.

BIBLIOGRAFÍA

[1] F. hsiung Hsu, "Ibm's deep blue chess grandmaster chips," *IBM T.J. Watson*. [Online]. Available: http://www.csis.pace.edu/~ctappert/dps/pdf/ai-chess-deep.pdf 1.1

[2] F. Bevilacqua, "Finite-state machines: Theory and implementation," *EnvatoTuts+*, 2013. [Online]. Available: https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867 1.1

[3] A. Patel, "Amit's thoughts on pathfinding." [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html 1.1

[4] C. Stergiou and D. Siganos, "Neural networks," *Imperial College London*. [Online]. Available: https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#Contents 3.1.2, 3.1

[5] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com 3.2, 3.1.2, 3.3, 3.3.1, 3.8

[6] "Frank rosenblatt," *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Frank_Rosenblatt 3.1.2

[7] "A simple neural network - transfer functions," *MLNotebook*, 2017. [Online]. Available: https://mlnotebook.github.io/post/transfer-functions/ 3.4, 3.5, 3.6, 3.1.3

[8] "Darwin, evolution, & natural selection," *Khan Academy*. [Online]. Available: https://www.khanacademy.org/science/biology/her/evolution-and-natural-selection/a/darwin-evolution-natural-selection 3.2.1

[9] F. Gomez and A. Quesada, "Genetic algorithms for feature selection in data analytics," *Neural Designer*. [Online]. Available: https://www.neuraldesigner.com/blog/genetic_algorithms_for_feature_selection 3.7, 3.2.2

[10] K. S. y Jeff Sutherland, *La Guía de Scrum*, 2016. [Online]. Available: http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-Spanish.pdf#zoom=100 6.2

[11] R. C. Martin, *Clean Architecture. A Craftman's Guide to Software Structure and Design*. Prentice Hall, 2017. 6.3, 6.3

[12] J. McCall, "Genetic algorithms for modelling and optimisation," *Journal of Computational and Applied Mathematics*, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377042705000774

[13] J. Fierro, "Clean architecture in django," 2017. [Online]. Available: https://engineering.21buttons.com/clean-architecture-in-django-d326a4ab86a9 6.1