ESCUELA DE INGENIERÍA DE

FUENLABRADA

GRADO EN INGENIERÍA DE ROBÓTICA SOFTWARE

TRABAJO FIN DE GRADO

DESARROLLO DE JUGADOR AUTÓNOMO DE CATÁN MEDIANTE TÉCNICAS DE APRENDIZAJE REFORZADO PROFUNDO

Autor: Rubén Montilla Fernández

Tutor: David Gualda Gómez

Curso académico 2023 / 2024

Agradecimientos

Cuando se da la oportunidad de trabajar en lo que te gusta, es mucho más fácil perseverar y obtener mejores resultados. Esa es la sensación que yo he tenido al realizar este trabajo de fin de grado.

Quiero dar las gracias a todos mis profesores de la carrera que en sus asignaturas me han enseñado cosas que estoy seguro me serán de mucha utilidad en el futuro. En especial a mi tutor David, que ha estado conmigo en este proyecto y me ha dado ánimos a continuar y sacar el proyecto adelante. Gracias por enseñarme tanto.

Gracias a mis amigos por acompañarme en los duros años de estudio y por hacerme los días más amenos. Gracias a mi familia por ayudarme a seguir luchando por terminar y enseñarme a forjar un futuro mejor.

Resumen

En este Trabajo Fin de Grado (TFG) se ha desarrollado un entorno de simulación en Python del juego Colonos de Catán con jugadores que toman decisiones aleatorias y que permite realizar un análisis de la partida en cualquier momento.

El entorno de simulación desarrollado ha servido como fundamento para la creación de una versión base de jugador autónomo generado a partir de la utilización de técnicas de aprendizaje por refuerzo cuyo objetivo es superar a jugadores que toman decisiones aleatorias.

Por último, los resultados del proyecto han sido satisfactorios ya que se ha conseguido crear un entorno de juego en el cual un jugador autónomo es capaz de usar sus recursos para aprender a jugar.

Palabras Clave

Catán, Aprendizaje Automático, Inteligencia Artificial, Redes Neuronales, Python

Abstract

In this End of Degree Project, a simulation environment has been developed in Python for the game Settlers of Catan with players making random decisions and allowing an analysis of the game at any time.

The developed simulation environment has served as the basis for the creation of an autonomous player base version generated from the use of reinforcement learning techniques whose goal is to outperform players making random decisions.

To conclude, the results are satisfactory given the fact that the implementation of the simulation gives the autonomous player the needs for learning how to play the game.

Keywords

Catan, Machine Learning, Artificial Intelligence, Neural Networks, Python

Índice de contenidos

Índice de figuras

Índice de tablas

1 Introducción

1.1 Presentación

Los colonos de Catán es un juego de mesa creado por Klaus Teuber en los años 90 y que dio lugar al juego que hoy todos conocen por el nombre de Catán [1]. Esta nueva versión creada en 2010 por la empresa de juegos de mesa Devir ha dado la vuelta al mundo y es conocida por millones de personas.

A partir del juego de mesa, varias empresas de desarrollo de videojuegos se han unido para crear la aplicación oficial de Catán: Catán Universe. Esta aplicación simula partidas del juego de mesa y permite jugar online con gente de todo el mundo tanto en su versión original como en muchas de sus variantes [2].

Con el auge actual de las inteligencias artificiales que se entrenan para un único propósito, se genera una pregunta, ¿se puede crear una inteligencia artificial que aprenda a jugar al Catán?

1.2 Motivación personal

Desde pequeño he sentido fascinación por los juegos de mesa, me encantaba jugar a juegos de mesa con mi familia o con amigos. A pesar de que he jugado a muchos juegos de mesa a lo largo de los años, sin duda alguna el Catán es mi favorito.

Dado que en asignaturas de la carrera se enseña aprendizaje automático y se pone en práctica realizando diversos ejercicios, pronto nace la ilusión de usar dicho conocimiento en alguno de los juegos que me han acompañado durante mi infancia. Mi primer pensamiento me condujo a crear este trabajo que tanto me ha ilusionado.

1.3 Estructura de la memoria

En esta sección se describen cada uno de los capítulos para facilitar su orden y comprensión:

1. **Introducción:** a través de varios apartados se presenta el proyecto, así como la motivación personal que ha logrado la creación este proyecto y una breve descripción del contenido de la memoria.

2. **Objetivos:** se comentan los objetivos que se pretenden conseguir y se detallan las fases en las que se ha estructurado el proyecto.
3. **Definiciones y conceptos:** este apartado trata de describir las herramientas utilizadas para el proyecto, así como de explicar conceptos que en capítulos posteriores se van a usar y pueden ser desconocidos, tales como: el uso del lenguaje de programación Python, el juego del Catán, la librería de creación de juegos Pygame, etc.
4. **Desarrollo del trabajo:** este capítulo se divide en dos apartados principales: la implementación de la simulación del juego creada desde cero en Python y el entrenamiento del agente inteligente. En cada capítulo se explica la estructura del código utilizado para la implementación, utilizando como apoyo los flujogramas de las correspondientes funciones.
5. **Resultados:** en este capítulo se muestran y comentan los resultados obtenidos.
6. **Conclusiones:** se recogen las conclusiones obtenidas del proyecto, las competencias empleadas, competencias adquiridas y por último se sugieren algunas líneas de mejora para la aplicación en un futuro.
7. **Bibliografía:** se referencian las fuentes consultadas.

2 Objetivos

En este apartado se detallan los objetivos y las fases del proyecto que se han adoptado para lograr los objetivos

2.1 Objetivos del proyecto

Los objetivos del proyecto son:

1. Crear desde cero una implementación del famoso juego de mesa en Python sobre la cual se pueda extraer en cada momento la información necesaria para realizar un análisis de la partida y que sea compatible con una inteligencia artificial que resuelva el problema de jugar una partida completa y conseguir la mayor puntuación posible.
2. Generar un agente que mediante técnicas de inteligencia artificial sea capaz de superar la puntuación media obtenida por agentes que toman decisiones completamente aleatorias. Es decir, el agente debe ser capaz de ganar como mínimo un 25% de las partidas para considerar que en efecto es mejor que un agente arbitrario.

2.2 Fases de desarrollo del proyecto

2.2.1 Fase de investigación

Esta fase ha sido constante y recurrente a lo largo del desarrollo del proyecto. Se centra en la obtención de la información y conceptos relacionados con las herramientas que se han usado para crear la simulación y el entrenamiento del agente.

2.2.2 Fase de desarrollo

Esta fase a su vez se podría dividir en dos fases:

- **Implementación de la simulación**

  El primer paso del trabajo es crear la base sobre la que entrena el agente. El desarrollo se hace en Python y permite un análisis visual a la vez que también se diseña una representación visual que permite ejecutar partidas de manera fácil de seguir por el usuario

- **Entrenamiento del agente**

  Esta parte también se realiza en Python usando la simulación anterior para ejecutar miles de partidas rápidamente. El entrenamiento se hace usando redes neuronales combinadas con algoritmos genéticos.

2.2.3 Fase de evaluación de resultados

Tras finalizar las dos fases anteriores, se recogen los resultados de los entrenamientos en forma de documento de texto y se generan conclusiones analizando las acciones que se toman en el trascurso de la partida y el resultado final de la misma.

2.2.4 Fase de documentación

Esta fase de documentación ha sido llevada a cabo en diferentes tramos del proyecto:

• Al comienzo, con la estructura de la memoria y la definición de los objetivos.

• Al finalizar la fase de desarrollo para explicar los progresos realizados y adjuntar imágenes sobre la representación visual.

En la parte final del proyecto, documentando los resultados obtenidos, las conclusiones y líneas futuras, así como el resto de los apartados.

2.3 Diagrama de Gantt

Meses	1	2	3	4	5	6	7	8	9
Desarrollo Juego de Catán									
Desarrollo del Agente Inteligente									
Estudio Resultados									
Documentación									

Tabla 1. Diagrama de Gantt.

3 Definiciones y conceptos

En este apartado se definen los elementos y conceptos que se usan a lo largo del proyecto.

3.1 Colonos de Catán

Colonos de Catán (o simplemente Catán) es un juego de mesa creado por Klaus Teuber que consiste en colonizar una isla llamada Catán. Contiene un tablero modular que cambia cada partida y figuras de cuatro colores. Cada uno de los jugadores tiene la posibilidad de construir cinco poblados y cuatro ciudades que se usarán para colonizar la isla y trece carreteras que servirán para expandirse por el terreno [1].

Es un juego de estrategia y gestión de recursos, pero también es un juego de negocio al comerciar con otros jugadores. Las partidas suelen durar 90 minutos y a pesar de que es fácil de entender, la estrategia hace que sea un juego un poco difícil para los niños.

Fig. 1. Juego de mesa Catán [3].

3.2 Python

Python es un lenguaje de programación creado en 1989 [4]. Este lenguaje es uno de los más usados en este momento por su sencillez a la hora de aprender a programar desde cero. Es un lenguaje no tipado, es decir que las variables que se usan no requieren de una previa definición de qué van a contener. Esto permite almacenar fácilmente grandes grupos de datos y tener un acceso rápido a ellos.

A la hora de ejecutar un programa, el flujo de ejecución en Python es de arriba a abajo con uso de funciones y bucles o condicionales que varían el curso del código. Un ejemplo sencillo sería el siguiente:

```python
# El siguiente comando imprime un mensaje en la pantalla.
# La función print() toma el mensaje entre comillas y lo muestra en la pantalla.
print("¡Hola, mundo!")

# También puedes imprimir números y realizar operaciones matemáticas simples.
numero1 = 5
numero2 = 3
suma = numero1 + numero2
print("La suma de", numero1, "y", numero2, "es igual a", suma)

# Puedes asignar valores a variables y usar esas variables en tus operaciones.
nombre = "Juan"
edad = 30
print("Hola,", nombre, "tienes", edad, "años.")

# Puedes pedir al usuario que ingrese información.
nombre_usuario = input("Por favor, introduce tu nombre: ")
print("¡Hola,", nombre_usuario, "!")

# Las estructuras de control permiten tomar decisiones en tu programa.
# Aquí, comprobamos si un número es positivo o negativo.
numero = int(input("Introduce un número: "))
if numero > 0:
    print("El número es positivo.")
elif numero < 0:
    print("El número es negativo.")
else:
    print("El número es cero.")

# Python también es útil para bucles, como este bucle for que imprime números del 1 al 5.
for i in range(1, 6):
    print(i)

# Finalmente, puedes definir tus propias funciones para realizar acciones específicas.
def saludar(nombre):
    print("¡Hola,", nombre, "!")

saludar("María")
saludar("Carlos")
```

Fig. 2. Ejemplo sencillo Python.

Al ejecutar el programa anterior obtenemos el siguiente resultado:

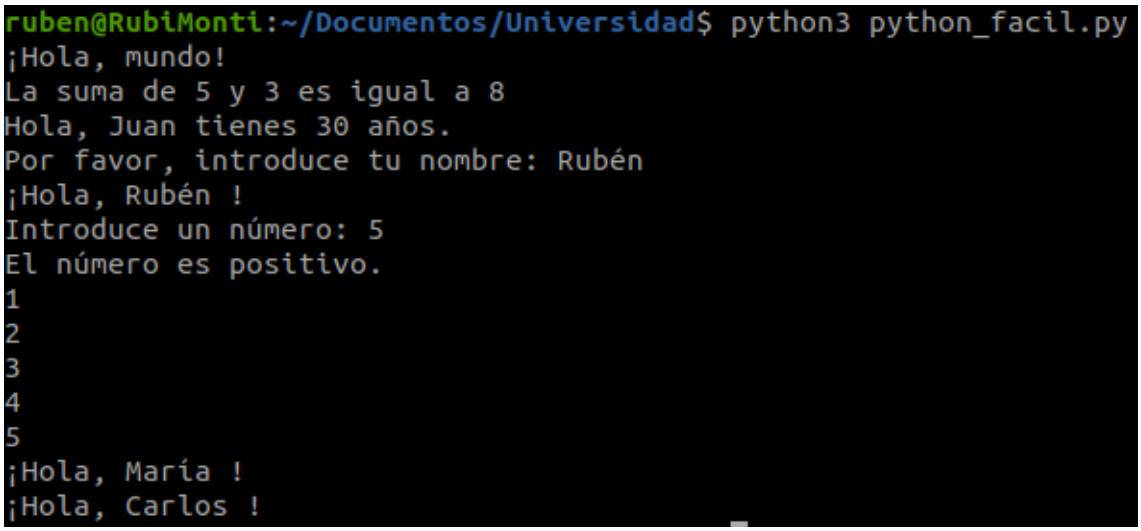

Fig. 3. Resultado ejemplo sencillo Python.

A lo largo del proyecto se usa el término clase y objeto, estos elementos forman parte de la programación orientada a objetos.

Programación orientada a objetos se denomina a la programación que se basa en crear un conjunto de datos que se almacenan en una estructura concreta que puede almacenar variables de distintos tipos y puede contener funciones especiales para ella. Lo especial de estas clases es que una vez definidas, se pueden crear objetos, que simplemente son referencias a ellas, y de esta forma almacenar varios conjuntos de datos de una manera sencilla y fácilmente operable. Un ejemplo sencillo puede ser el siguiente:

```python
# Definición de la clase Coche
class Coche:
    # Método constructor que se llama al crear un nuevo objeto Coche
    def __init__(self, marca, modelo):
        # Atributos de la clase Coche
        self.marca = marca
        self.modelo = modelo
        self.velocidad = 0

    # Método para acelerar el coche
    def acelerar(self, incremento):
        self.velocidad += incremento

    # Método para frenar el coche
    def frenar(self, decremento):
        self.velocidad -= decremento

    # Método para obtener la información del coche
    def obtener_informacion(self):
        return f"Coche: {self.marca} {self.modelo}, Velocidad: {self.velocidad} km/h"

# Creación de objetos de tipo Coche
coche1 = Coche("Toyota", "Camry")
coche2 = Coche("Honda", "Civic")

# Usando los métodos de los objetos
coche1.acelerar(20)
coche2.acelerar(15)

coche1.frenar(5)
coche2.frenar(10)

# Mostrando información de los coches
print(coche1.obtener_informacion())
print(coche2.obtener_informacion())
```

Fig. 4. Ejemplo Python orientado a objetos.

El resultado del código anterior es el siguiente:

```
ruben@RubiMonti:~/Documentos/Universidad$ python3 python_orientado_a_objetos.py
Coche: Toyota Camry, Velocidad: 15 km/h
Coche: Honda Civic, Velocidad: 5 km/h
```

Fig. 5. Resultado ejemplo Python orientado a objetos.

3.3 Pygame

Dentro de Python se permite el uso de librerías que aportan nuevas funcionalidades o constantes que ayudan a la creación de código reduciendo la redundancia de código. Una de estas librerías es Pygame [5].

Esta librería implementa un entorno de programación cómodo para crear juegos dentro de Python. Un ejemplo de esto es que proporciona funciones que permiten crear una ventana en la que se puede dibujar, imprimir imágenes o realizar animaciones.

Además, Pygame también implementa una gestión de eventos que permite analizar el teclado y saber si alguna tecla se ha presionado o se ha mantenido pulsada, función muy útil a la hora de crear juegos que requieran entradas por parte del usuario.

3.4 Redes Neuronales

Una red neuronal es un método matemático que se basa en el cerebro humano y que trata de imitar el comportamiento de las neuronas para transmitir los datos. La funcionalidad de este diseño será la de analizar información para reconocer patrones y de esta manera poder realizar tareas específicas usando la salida de la red. Como se puede ver más adelante, las redes neuronales se usan en varios algoritmos de aprendizaje automático, entre ellos en el que se ha usado para resolver el problema del Catán.

Las redes neuronales están compuestas de capas de neuronas que serán las encargadas de realizar operaciones a los datos de entrada y proporcionar una salida que se ajuste a lo deseado. Cada capa de la red tendrá todas sus neuronas conectadas a las neuronas de la capa anterior y a las de la posterior, exceptuando a la salida y la entrada que solo estarán conectadas a la capa anterior y a la capa posterior respectivamente:

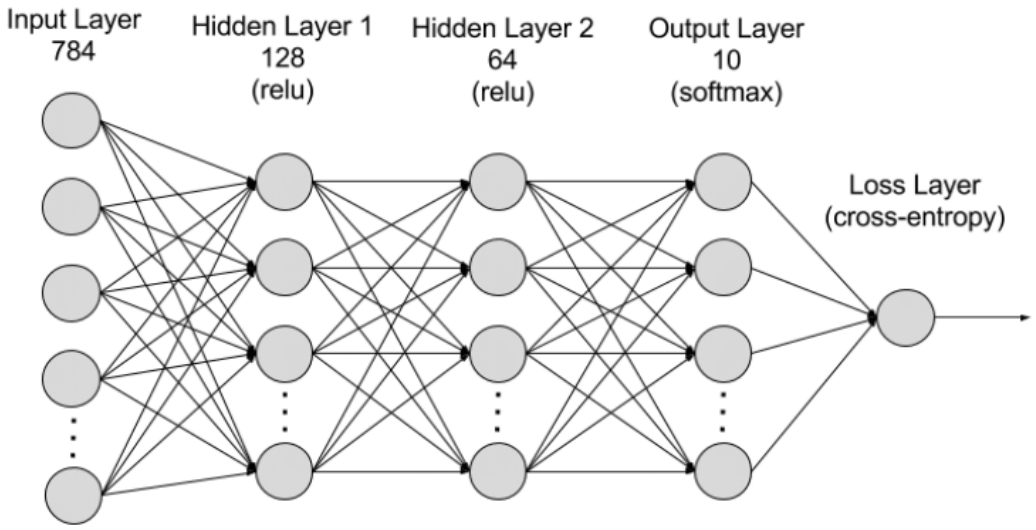

Fig. 6. Estructura de las redes neuronales [6].

Podemos diferenciar las capas en tres tipos:

- **Capa de entrada**

  Esta capa es la encargada de recibir los datos e introducirlos en la red. El tamaño de la capa de entrada viene definido por el número de datos que se introducen. Por tanto, a mayor cantidad de información, mayor complejidad tendrá la red.

- **Capas ocultas**

  Estas capas que se encuentran entre la entrada y la salida son las encargadas de realizar las operaciones y de extraer los patrones en los datos que determinarán la salida de la red. Cuanto mayor sea el número de capas mayor profundidad tendrá la red y por tanto mayor complejidad.

- **Capa de salida**

  Esta capa es la encargada de generar la salida de la red neuronal. Esta salida dependerá de la aplicación de la red y, por tanto, estará definida por las variables que se quiera tener en la salida.

Como se ha mencionado previamente, cada capa tiene un número de neuronas. Una neurona es la función encargada de transmitir los datos de la entrada y, realizando las operaciones oportunas, llevarla hasta la salida. Una neurona estará compuesta por varias partes:

- **Entradas**

  Las entradas de la neurona serán la salida de todas las neuronas que hay en la capa anterior. En caso de que la capa sea la primera, las entradas de la neurona serán las mismas que las de la red.

- **Pesos**

  Los pesos serán valores constantes que serán los encargados de variar la salida y son los valores que los algoritmos de aprendizaje cambiarán para que la red neuronal aprenda a resolver el problema.

- **Suma ponderada**

  La suma ponderada es la operación que, usando las entradas, los pesos y un parámetro opcional, que son los términos independientes, devolverá un valor que se pasa a la función de activación.

- **Función de activación**

  La función de activación es la encargada de sacar la salida de la neurona usando la suma ponderada. Algunas funciones usadas comúnmente son la función sigmoide o la función tangente hiperbólica.

- **Salida**

  La salida será conectada a la entrada de cada neurona de la capa siguiente o en el caso de ser la última capa, será directamente conectada a la salida de la red.

El ejemplo más sencillo de una red neuronal es el perceptrón [7]**¡Error! No se encuentra el origen de la referencia.**. Esta red es la más simple ya que contiene únicamente una neurona y una salida:

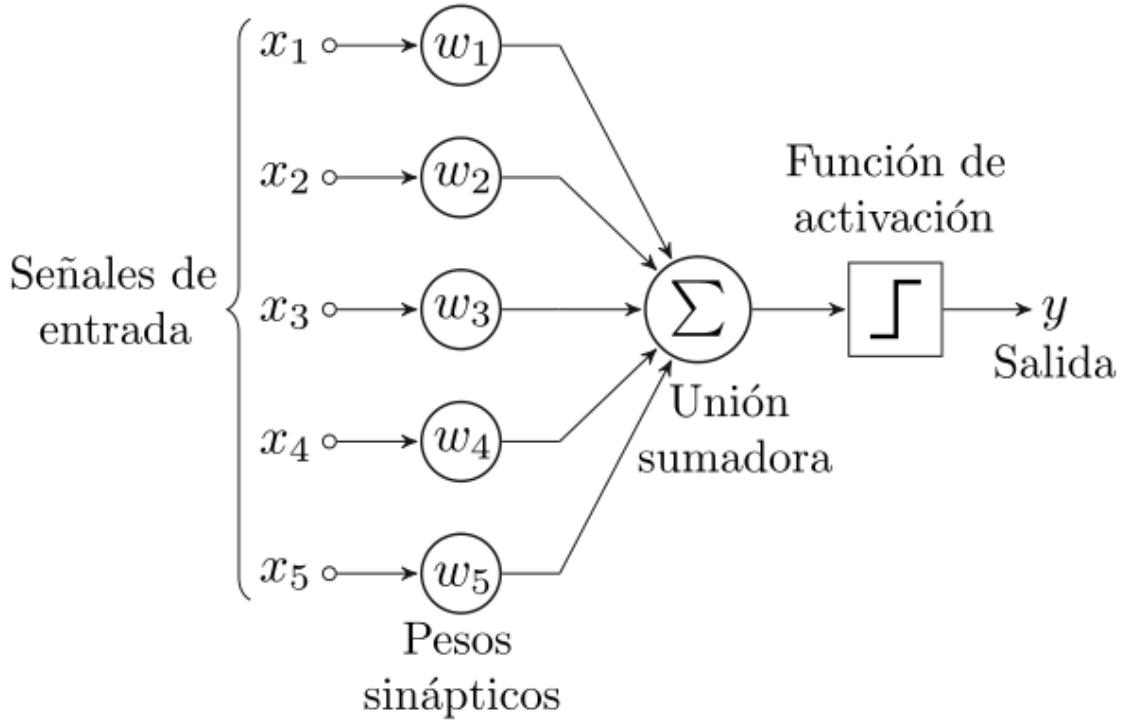

Fig. 7. Perceptrón [8].

3.5 Aprendizaje automático

El aprendizaje automático es una rama de la inteligencia artificial cuyo propósito es mejorar el rendimiento de las computadoras a la hora de realizar tareas específicas como pueden ser la clasificación o la identificación de elementos, o el ajuste de una función inventada a la salida de otra función de la cual tenemos los datos [9]. Para

obtener los resultados, se tienen en cuenta patrones y se realizan predicciones basadas en dichos patrones.

Existen tres tipos principales de aprendizaje automático:

- **Aprendizaje supervisado**

  En este tipo de aprendizaje, se proporciona al modelo un conjunto de datos que ha sido previamente tratado y etiquetado para que el modelo aprenda a etiquetar nuevos datos según la nueva entrada que se pueda proporcionar. Algunos ejemplos de este tipo de aprendizaje son Regresión Lineal, Máquinas de Soporte Vectorial (SVM) y Redes neuronales [10].

- **Aprendizaje no supervisado**

  En este tipo de aprendizaje, también se proporciona al modelo un conjunto de datos, pero esta vez, los datos no son tratados de ninguna manera previamente y es el modelo el que tiene que encontrar los patrones necesarios para adaptar su salida a la deseada. Este tipo de aprendizaje se utiliza para agrupación de datos en grupos con algoritmos como K-means y también para disminuir la complejidad de los datos reduciendo sus dimensiones [10].

- **Aprendizaje por refuerzo**

  En este tipo de aprendizaje, no se le proporcionan datos de entrada, sino que un agente interactúa con un entorno y dependiendo de las acciones o decisiones que se tomen, el entorno devolverá una recompensa que se irá acumulando. El algoritmo es el encargado de maximizar esa recompensa mediante prueba y error para obtener una solución óptima al problema planteado en el entorno. Este tipo de aprendizaje se usa principalmente en robótica, por ejemplo, a la hora de imitar movimientos humanos como puede ser andar, toma de decisiones autónomas o aprendizaje de juegos, como es el caso de este proyecto. Algunos algoritmos de este tipo son Q-Learning o Deep-Q-Learning [10].

A pesar de que todos los algoritmos mencionados son muy potentes, el único que se usa es el algoritmo genético ya que para el problema del Catán hay demasiadas variables y acciones posibles para utilizar cualquier otro.

3.6 Algoritmos Genéticos

A pesar de que todos los algoritmos mencionados son muy potentes, para resolver un problema tan complejo como el Catán se requiere de un elemento extra. Los algoritmos genéticos.

El algoritmo genético es una técnica de aprendizaje que se ha basado en la biología y en la teoría de la evolución. Aunque este algoritmo esté presente en muchas técnicas de aprendizaje automático, por sí solo no se considera un algoritmo de aprendizaje automático [9].

Como se menciona en el párrafo anterior, el algoritmo creará lo que se denomina por población, que está compuesta por distintas posibles soluciones al problema, también denominados cromosomas. Posteriormente evaluará cada una de las posibles soluciones y ejecutará una serie de operaciones para seleccionar, reproducir y finalmente hacer evolucionar la población hacia mejores soluciones. Las distintas fases del algoritmo genético son:

- **Inicialización de la población**

  Al principio de la ejecución, la población se inicializa de forma aleatoria. Cada una de las posibles soluciones será un cromosoma que posteriormente se evaluará y será importante para el cruzamiento.

- **Evaluación de Aptitud**

  En esta fase se evalúa el comportamiento de los cromosomas frente al problema. Para ello habrá que crear una función de aptitud o *fitness function* que ejecutará el programa y devolverá una recompensa o evaluación que indicará como de buena es la solución.

- **Selección**

  Una vez que se han evaluado todos los cromosomas será el momento de elegir cuales son los mejores y los que serán seleccionados para el cruzamiento. El número de cromosomas elegidos puede variar dependiendo del problema [11].

- **Cruzamiento**

  De los cromosomas seleccionados se crearán unos nuevos cromosomas también denominados soluciones descendientes. En el cruzamiento, los genes de las soluciones evaluadas, también llamados cromosomas padre, se mezclan entre ellos para crear lo que se denomina cromosomas hijos. De esta manera, se crearán hijos que mantengan las características buenas de los padres [12].

- **Mutación**

  En cada cruzamiento hay una probabilidad ajustable de introducir una mutación en el proceso. La mutación consiste en cambiar uno o más cromosomas hijos por una nueva solución aleatoria. De este modo, el algoritmo evita quedarse atascado en una solución que podría servir para resolver el problema, pero no sería óptima.

- **Reemplazo**

  La nueva población compuesta por los cromosomas hijos tomará el relevo y esta nueva generación volverá a la fase de evaluación y continuará hasta convertirse en los padres de la siguiente generación, así sucesivamente.

- **Criterio de parada**

  El algoritmo mantiene el bucle hasta que se cumple alguno de los criterios de parada que se hayan configurado. Estos criterios pueden ser alcanzar un número máximo de generaciones, superar un tiempo de ejecución o alcanzar una recompensa deseada.

Debido a su capacidad de mejora a través de las generaciones, es un algoritmo muy útil a la hora de hacer evolucionar arquitecturas de redes neuronales y, por tanto, también son útiles a la hora de realizar aprendizaje por refuerzo.

3.7 PyGad

Una vez explicado lo que es un algoritmo genético, tendremos que ver como implementar algoritmos genéticos en Python.

PyGad es una librería de Python que permite implementar algoritmos genéticos de una manera sencilla. Esta librería proporciona varias funciones que automatizarán los procesos que hemos visto antes [13].

Lo primero de todo es crear una instancia de la clase que contiene todas las funciones necesarias para el correcto funcionamiento del algoritmo. A la hora de crear la instancia, se podrá determinar varios factores como el número de padres que queremos seleccionar para la mutación, el número máximo de generaciones que se van a ejecutar, el número de cromosomas que va a tener cada generación, la función de evaluación que se va a utilizar…

Posteriormente, se ejecuta la función *run* que es la encargada de realizar todas las fases y de realizar correctamente el bucle. Una vez la ejecución de esta función acabe, dentro de la instancia se encontrará la información de la mejor solución que se ha encontrado, la cual se puede extraer con la función *get_best_solution*.

3.8 Biblioteca Random

En el proyecto será necesaria la obtención de valores aleatorios para realizar algunas operaciones. Dentro de Python, necesitaremos la ayuda de una biblioteca para obtener dichos valores.

La biblioteca Random de Python implementa varias funciones que permiten obtener valores pseudoaleatorios siguiendo varias distribuciones de probabilidad. La función que se usa en el proyecto es *randint*, una función que devuelve un número entero aleatorio entre dos valores que se introducen por parámetros [14].

Esta biblioteca nos permite aumentar la aleatoriedad del juego, pero al tratarse de computación es difícil obtener un valor completamente aleatorio. Es por lo que la biblioteca Random nos asegura valores pseudoaleatorios que son generados a partir de un elemento del ordenador que le produce una aleatoriedad.

4 Desarrollo del trabajo

4.1 Explicación del juego

Catán es un juego de mesa estratégico de colonización y desarrollo. En él, los jugadores intentan asentarse en una isla en la que se producen distintos recursos que usarán para desarrollar una colonia. Ganará el juego el primer jugador que sea capaz de alcanzar los 10 puntos de victoria.

4.1.1 El tablero

El juego se desarrolla en un tablero compuesto por 19 hexágonos colocados al azar, asignándose a cada uno de ellos un número del 2 al 12 exceptuando el 7. Estos números se colocan en un orden especial para evitar la acumulación de probabilidades, la cual se explica más tarde. Los hexágonos están colocados de forma similar a la siguiente:

Fig. Tablero del Catán [15].

En el mapa podemos observar distintos elementos importantes:

Los hexágonos de materiales son el elemento principal de la partida. Cada uno de ellos es el encargado de proporcionar un recurso a los jugadores; el hexágono rojo produce arcilla, el verde oscuro produce madera, el verde claro produce lana, el amarillo produce cereales y el gris produce mineral. También se puede apreciar que existe un hexágono especial, este es el desierto que no tiene ningún número asignado y que por tanto no produce ningún recurso.

En los límites del tablero se puede observar que la isla formada por los hexágonos está rodeada por el mar. Dibujados en él hay unos puertos marítimos con números y materiales. Estos se explican más tarde y forma una parte importante para el comercio que se explica más tarde.

Por último, se puede observar que hay una ficha en el desierto, se trata del ladrón. El ladrón es otro factor importante que puede decidir partidas y que también es mencionado y explicado más tarde.

Los jugadores disponen de tres posibles construcciones que pueden colocar en distintas partes del tablero. La primera de ellas es la carretera, la cual permite expandir la colonia por el mapa y así alcanzar hexágonos o puertos que no eran alcanzables previamente. La segunda es el poblado, que permite al jugador obtener recursos de los hexágonos adyacentes y además le otorga un punto de victoria. Por último, las ciudades son una mejora de los poblados y se coloca en el tablero sustituyendo un poblado ya existente. Este cambio hace que la producción de recursos sea el doble y el edificio otorgue 2 puntos de victoria en vez de 1 que correspondían al poblado.

En el tablero se pueden distinguir dos zonas con las que interactúan los jugadores: los caminos o aristas que unen dos hexágonos y sobre las que se colocan las carreteras de los jugadores, y las encrucijadas o vértices que son el cruce entre tres hexágonos o de uno o dos hexágonos con el mar. Sobre estas encrucijadas se colocan los poblados y las ciudades.

En la siguiente ilustración se puede ver como los poblados de los jugadores se colocan en las encrucijadas (*intersection*) y las carreteras en los caminos (*path*). También se pueden localizar las encrucijadas que otorgan el beneficio de los puertos (*harbor*) y la figura del ladrón en el centro (*robber*):

Fig. 8. Tablero del Catán detallado [15].

4.1.2 Fase de asentamiento

El juego comienza con una fase de asentamiento en la que los jugadores tienen la oportunidad de colocar sus primeras fichas en cualquiera de las casillas del tablero.

Los jugadores toman turnos en el sentido de las agujas del reloj para poner un poblado en cualquier encrucijada libre y respetando la regla de la distancia. Esta regla dicta que, para poder construir un poblado, todas las encrucijadas adyacentes a la que queremos construir deben estar libres. Posteriormente pondrán una carretera en cualquiera de los tres caminos adyacentes al poblado. Cuando todos los jugadores hayan puesto un poblado y una carretera, toman turnos en el sentido contrario a las agujas del reloj para poner un segundo poblado. Es decir, el último jugador en poner el primer poblado tiene la opción de elegir donde colocar el poblado y la carretera dos veces seguidas y el primer jugador será el último en poner el segundo poblado. Al colocar este segundo poblado, a cada jugador se le otorgan las cartas de materia primas correspondientes al último poblado colocado en la fase de asentamiento. Para repartir

las cartas iniciales a cada jugador se tienen en cuenta los hexágonos que rodean el último poblado colocado por cada jugador y se le otorga una carta de la materia prima correspondiente a dicho hexágono. Esto significa que los jugadores pueden iniciar la partida con un máximo de 3 cartas.

Una vez que el primer jugador coloca su segundo poblado y carretera, éste es el que inicia la partida tirando los dados.

4.1.3 Desarrollo del turno

La producción de recursos sucede una vez al inicio de cada turno y se resuelve tirando dos dados. Únicamente los hexágonos que tengan asignados el número resultante de la suma de los dos dados producen recursos en ese turno. Todos los jugadores que tengan asentamientos o ciudades construidos en los vértices de los hexágonos que han producido materiales en este turno reciben una o dos cartas respectivamente del material perteneciente a dicho hexágono.

Siguiendo la colocación del tablero en la ilustración 2, en caso de que el resultado de los dados fuese un 10, independientemente de qué jugador haya lanzado los dados, el jugador rojo recibiría una carta de mineral por su poblado etiquetado como A y el jugador amarillo recibiría una carta de arcilla por su poblado etiquetado como B

Otro factor para tener en cuenta si nos fijamos en la figura anterior es que en ninguna casilla está la ficha correspondiente al número 7. Esto se debe a que, si el resultado de los dados es el 7, se resolverá el evento del ladrón. El ladrón es la ficha negra (representada con un círculo) que robará cartas de materia prima a los jugadores cada vez que salga dicho resultado. En el momento en el que sale el resultado, todos los jugadores que posean más de 7 cartas de materia prima en la mano, están obligados a devolver la mitad de sus cartas a la banca. Las cartas que devuelvan quedan a su elección. Una vez que los jugadores se han descartado de las cartas, el jugador que ha realizado la tirada de los dados es el encargado de poner el ladrón en una casilla nueva y podrá robar una carta de materia prima a uno de los jugadores que tenga un poblado o una ciudad colindante a la nueva ubicación del ladrón. La carta robada se escoge al azar de entre todas las que posea el jugador en ese momento. El ladrón permanecerá en esa posición hasta que salga otro siete en los dados o hasta que una persona utilice una carta de caballero que se explica a continuación. Mientras que el ladrón permanezca en una

casilla del tablero, dicha casilla no producirá recursos a pesar de que salga el número que hay debajo del ladrón.

Una vez repartidas las cartas de materiales, el jugador tiene la posibilidad de hacer tres acciones el número de veces que quiera y en el orden que quiera: Construir, comerciar y jugar cartas de desarrollo.

- **Construir**

  En la construcción, el jugador puede colocar una de las tres fichas que hemos visto previamente en el tablero a cambio de los materiales necesarios. Se pueden construir todos los edificios que el jugador se pueda permitir en un mismo turno. Para construir una carretera basta con una carta de madera y una de arcilla y la podremos poner en cualquier camino adyacente a una carretera en propiedad del jugador. Para un poblado se necesitan una carta de madera, una de arcilla, una de cereal y una de lana, y se puede construir en cualquier encrucijada adyacente a una carretera que pertenezca al jugador y que respete la regla de la distancia. Por último, para una ciudad se necesitan tres cartas de mineral y dos de cereal, y se construye sustituyendo un poblado ya existente. Otra opción es la de comprar una carta de desarrollo (los distintos tipos y sus funciones se explican a continuación) a cambio de una carta de lana, una de cereal y una de mineral.

Fig. 9. Tabla de costes [16].

Como se ha explicado previamente, a la hora de construir fichas en el tablero hay que seguir unas reglas de construcción: la regla de la distancia, y una regla que dicta que, para poder construir en una encrucijada, el jugador ha debido encontrar la forma de llegar hasta dicha encrucijada construyendo carreteras y formando un camino hasta ella.

- **Comercio**

La segunda opción que tiene el jugador es comerciar tanto con la banca como con otros jugadores. Se pueden realizar tantos intercambios como el jugador desee con la única condición de que el jugador del turno actual siempre tiene que estar incluido en el trato, no está permitido realizar intercambios entre dos jugadores los cuales no sea su turno.

Para intercambiar con la banca, denominado comercio marítimo, se intercambian un número de cartas del mismo recurso a cambio de una carta del recurso que se desee. Los intercambios con la banca se realizan a razón de 4 a 1, es decir, el jugador debe entregar cuatro cartas de la misma materia prima a cambio de una de su elección. También existe la posibilidad de que el jugador posea un poblado o ciudad en un puerto que le permita una razón de intercambio mejor como puede ser 3 a 1 o los puertos específicos en los que se puede intercambiar a razón de 2 a 1, pero el requisito es que únicamente necesita entregar dos cartas del recurso que hay en el dibujo para obtener el recurso deseado.

En cambio, a la hora de realizar intercambios con otros jugadores, denominado comercio doméstico, el trato puede ser cualquier cantidad de cartas que sea acordado por ambos jugadores con dos únicas reglas: no se pueden regalar cartas, es decir, siempre tiene que haber algo en los dos lados del intercambio y el mismo recurso no puede estar en los dos lados del intercambio.

- **Jugar una carta de desarrollo**

Por último, el jugador tiene la posibilidad de jugar una única carta de desarrollo que posea y siempre que no se haya adquirido en ese mismo turno. Hay 3 tipos de cartas de desarrollo:

El caballero es la carta más común y cuando es usada, el jugador está obligado a mover el ladrón de la casilla en la que está a cualquiera de los hexágonos que hay en el tablero. Posteriormente tiene la posibilidad de robar una carta al azar de uno de los jugadores que tengan un poblado o ciudad adyacente a ese hexágono si es que hubiese alguno. Las cartas usadas se mantienen delante del jugador que las ha jugado porque serán importantes para conseguir un bonus de puntos de victoria.

El segundo tipo de cartas son los puntos de victoria. Hay cartas de desarrollo que otorgan al jugador que las compre un punto de victoria que se mantendrá oculto hasta el final de la partida. Es decir, no es necesario jugar esta carta, sino que cuando el jugador posea los puntos de victoria necesarios para sumar 10 con las cartas de puntos de victoria, se acabará la partida y este jugador será el ganador.

Por último, existen las cartas de progreso, que son 3 cartas que otorgan una ventaja inmediata al jugador que las usa. Las tres cartas son:

- Carreteras: al usarse, el jugador tiene la posibilidad de construir dos carreteras de manera gratuita siguiendo las reglas de construcción que se han explicado previamente.
- Invento: esta carta permite al jugador que la use elegir dos cartas de materia prima a su elección que podrá robar de la banca.
- Monopolio: cuando se juega esta carta, el jugador nombra una materia prima y el resto de los jugadores están obligados a entregar todas las cartas de dicha materia prima que tengan en la mano en ese momento.

Una vez que el jugador haya realizado todas las acciones que quiera y en el orden que quiera, tomará la decisión de pasar el turno al siguiente jugador. Los turnos se suceden hasta que uno de los jugadores obtiene su punto de victoria número 10.

4.1.4 Final del juego

En el momento que cualquier jugador llegue a 10 o más puntos de victoria ya sea en su turno o en el de cualquier otro jugador la partida acaba con la victoria de dicho jugador.

Aparte de los puntos de victoria que ya se han explicado previamente (los edificios colocados en el tablero y los puntos de victoria de las cartas de progreso) existen dos bonificaciones que otorgan 2 puntos de victoria cada una:

La primera es la bonificación por mayor ejército, que se le otorga al jugador con más cartas de caballero usadas hasta el momento. El primer jugador en alcanzar los tres caballeros activados toma la tarjeta de bonificación que en cualquier momento puede robar otro jugador en caso de superar el número actual de caballeros del jugador con la tarjeta.

La segunda bonificación es la de mayor ruta comercial. En este caso, en vez del número de caballeros activados se cuentan el número de carreteras consecutivas que cada jugador posea. Al igual que la anterior bonificación, también está asociada a una tarjeta que irá cambiando de dueño dependiendo quien tenga el mayor número de carreteras. Para contar el número de carreteras se empezará por una carretera de un extremo y continuaremos siguiendo las carreteras una a una hasta que no podamos avanzar. Es posible que siguiendo ese camino haya carreteras que no se cuenten porque al colocarlas, se genere una bifurcación (se explica más a fondo a continuación).

4.2 Diseño del juego

El juego está completamente diseñado desde cero en el lenguaje de programación Python. Dentro de la creación de los elementos del juego se pueden distinguir dos partes, siempre habrá una parte de software que incluirá todos los datos y demás subelementos que se encargan de que el juego funcione, y la parte gráfica que representa los datos almacenados en forma de juego para que el usuario pueda observar la partida e incluso interactuar con ella.

4.2.1 El tablero

Como se ha explicado anteriormente, el tablero esta principalmente compuesto por hexágonos que son los encargados de producir las materias primas necesarias para la construcción de elementos que hacen que los jugadores puedan progresar durante la partida.

Lo primero de todo es crear dos clases primitivas que serán las clases que almacenen la información básica y que son necesarias para crear el resto de los

elementos. Estas clases son *Corner* o encrucijada y *Edge* o camino. Estas clases en términos de software no almacenan muchos datos, pero son muy útiles para dibujar el tablero de una forma sencilla.

La clase *Corner* es la encargada de almacenar cuales son las encrucijadas adyacentes y cuáles son las casillas que producen materia prima que tiene alrededor. De esta manera se puede extraer la información en cualquier momento del juego. También será la encargada de saber si en la encrucijada hay un puerto o si hay algún edificio de alguno de los jugadores. Para saberlo, se crea una nueva clase llamada *Building* que almacena en su interior el tipo de edificio que se ha construido y a qué jugador pertenece. A cada esquina se le asigna un objeto de tipo *Building* que puede estar vacío, o con la información de alguno de los jugadores. Por último, tambien almacena los datos necesarios para que se puedan imprimir correctamente los edificios en el mapa.

La clase *Edge* es parecida a la anterior, pero algo más sencilla. Únicamente almacena si tiene alguna carretera de algún jugador en ella y una lista de todos los caminos adyacentes. En la parte gráfica se almacena la posición que tiene dentro del tablero y la orientación que este camino tiene, ya que, a diferencia de las encrucijadas, las carreteras tienen tres orientaciones debido a la geometría de los hexágonos que componen el tablero.

Posteriormente se crea la clase *Tile* o casilla, que sirve para identificar el recurso que esa casilla proporciona, el número que tiene que salir en los dados para que la casilla sea producente en este turno y si el ladrón está en esa ubicación. Además, también almacena información sobre las 6 encrucijadas y los 6 caminos que la componen. Las 6 encrucijadas y los 6 caminos se almacenan referenciando a los objetos que hemos creado anteriormente para que sean modificables desde cualquier parte del programa. Con esta organización, se obtiene una manera sencilla de almacenar también la información de los poblados y ciudades de los jugadores y realizar más rápidamente el reparto de materias primas al inicio de cada turno. En la parte gráfica de la clase se encuentra una única función que, a partir de unas coordenadas de inicio, genera un hexágono en la pantalla de Pygame del color del recurso que otorga y con un círculo blanco indicando que número es el que proporciona el recurso previamente mostrado.

A continuación, se crea la clase Tablero que almacenará los datos de todos los hexágonos, encrucijadas, caminos, puertos…. En la parte de software, esta clase

contiene una lista con todos los objetos *Tile* que se han mencionado anteriormente, en concreto 19 casillas. Estas 19 casillas siempre están compuestas de 4 bosques, 4 pastos, 4 sembrados, 3 cerros. 3 montañas y un desierto. Al inicio de la partida, estos hexágonos se colocan aleatoriamente. Gracias a la biblioteca "random" se puede lograr un orden pseudoaleatorio para colocar los hexágonos y formar la isla que se convertirá en el tablero de juego. Posteriormente, se asigna un número a cada hexágono esta vez en un orden determinado para evitar acumulaciones de probabilidades. Los números estarán colocados para que nunca dos hexágonos colindantes tengan números iguales y que esté equilibrado, es decir, las probabilidades de producir recursos en este turno esta equilibrado en todo el tablero. La probabilidad de los dados sigue el siguiente esquema:

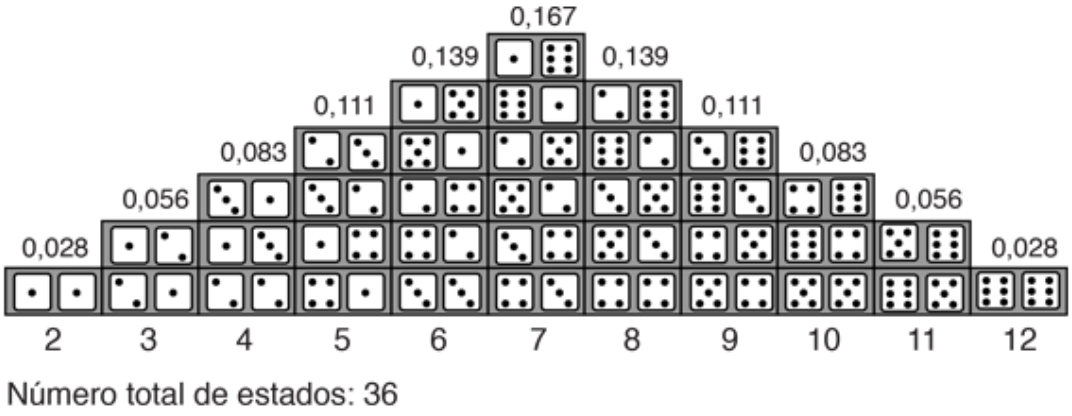

Fig. 10. Probabilidad de dos dados [17].

Siguiendo el esquema anterior, se puede ver que el 7, que es el número del ladrón, es el número que más posibilidades de salir tiene. Después le siguen por parejas en orden ascendente y descendente del 6 y el 8 hasta el 1 y el 12 respectivamente. Es por esto por lo que en una partida si hubiese un jugador que tuviese un poblado rodeado de números muy frecuentes tendría ventaja frente a un jugador que solo tenga poblados en números con probabilidad baja. El orden de colocación de los números es el siguiente:

5,2,6,3,8,10,9,12,11,4,8,10,9,4,5,6,3,11.

Asignando el 5 en la esquina superior izquierda y siguiendo en forma de espiral hacia abajo acabando en la casilla del centro del tablero con el 11. En caso de que la casilla sea el desierto, esa casilla se saltará y el número que se le hubiese asignado a dicha casilla pasará a asignarse a la siguiente.

Fig. 11. Orden de la numeración [15].

Por último, se almacenan los datos de los puertos que hay en el tablero. En total hay 9 puertos, 5 puertos específicos que permiten el cambio de 2 a 1, uno de cada material y 4 puertos genéricos que permiten el intercambio de 3 a 1 cuando un jugador tiene un poblado o una ciudad en él.

En la parte gráfica del tablero únicamente se dibuja el fondo azul que indicará el agua y sobre este fondo se pintarán los hexágonos uno a uno utilizando la función que se ha explicado anteriormente de la clase *Tile* y así se forma la tierra de la isla. Para completarla añadiremos los puertos que siempre tienen un lugar concreto por eso simplemente se indicarán las coordenadas en las que hay que dibujarlo.

Una vez concluidos los elementos que componen el tablero y el juego visual, es el turno de los elementos que hagan que la partida avance. Estos jugadores son los que realizan las acciones y almacenan todos los datos necesarios para tomar dichas acciones. Los cuatro jugadores que componen la partida son objetos de una clase llamada *Player*. Esta clase servirá para almacenar información como cuántas fichas de carreteras poblados o ciudades le quedan al jugador, cuántas cartas de cada materia prima posee o cuántas cartas de desarrollo tiene y cuales son dichas cartas. En la parte de software, dentro de esta clase encontramos una función que será muy útil y que dada la

información que tiene el objeto del jugador y con la entrada del estado actual del tablero y de la partida, es capaz de devolver una lista con todas las posibles acciones que puede realizar el jugador en ese momento. La función irá comprobando si cada una de las acciones que existen dentro del juego se pueden realizar o no. Empezará comprobando si está en la fase de fundación. En caso de estarlo, únicamente devuelve una lista de todas las variantes de la acción que corresponde a construir un poblado y una carretera contigua en cualquier parte válida del tablero. En caso de no estar en esta fase, se procede a añadir la acción de saltar el turno que se puede realizar siempre, y se continuará por orden numérico[1] comprobando los requisitos de cada acción. En caso de cumplirse, se añadirán todas las posibles combinaciones de parámetros de dicha acción. Por ejemplo, la función es capaz de ver si el jugador tiene las cartas de materia prima para construir un poblado y, en el caso de tenerlas, devolver una lista indicando en qué posiciones se podría colocar dicho poblado si existe alguna casilla libre. Por último, la lista de posibles opciones que se ha ido generando a lo largo de la ejecución de la función se devolverá con tipo fijo de lista de tuplas siendo cada tupla una posible acción a desarrollar.

En la parte de diseño del tablero esta clase Player es capaz de crear un cuadro en el cual se imprime la información de cada jugador en cada momento de la partida. Dentro de este cuadro del color de cada jugador se podrá ver de manera clara cuantas cartas de materia prima y cuantas cartas de cada tipo tiene cada jugador. También se imprime la información relativa a las cartas de desarrollo y los bonus de mayor ejército y mayor ruta comercial. Por último, se imprime en grande el número de puntos de victoria que tiene el jugador en el momento.

Fig. 12. Estadísticas jugador.

[1] El orden numérico viene dado por unas variables constantes que se crean al principio del programa y que se mantendrán a lo largo de la partida. Estas variables se encuentran en el fichero constantes.py del código fuente.

4.2.2 El juego

Por último, se crea una clase Game que es la más compleja y la que coordinará todos los demás elementos para que se pueda jugar una partida de Catán. Lo más importante de esta clase por tanto será la parte de software ya que la parte gráfica es muy simple y usará las funciones que ya se han descrito previamente. La parte gráfica consiste únicamente en una función que actualiza el mapa usando los datos que tiene almacenados sobre la posición de todos los elementos del tablero y usando las funciones de los objetos Player que imprimen sus datos en la pantalla y, por último, imprime la tirada actual de los dados. La parte del software es la parte más compleja del programa ya que es la encargada de hacer los cálculos necesarios para que la partida transcurra.

Al iniciar la clase Game se crean todos los objetos Corner y Edge que se requieren para crear el tablero e iniciar la partida, almacenándose en una lista para que posteriormente puedan ser referenciados y usados en otras funciones que ya se han explicado previamente para consultarlos o modificarlos. También se crea una lista con los cuatro objetos Player y otra con la cantidad de cartas de desarrollo que quedan. Una vez definida la clase, lo siguiente es definir las funciones que se usan a lo largo del transcurso de la partida.

El primer paso consiste en definir una función que determina si la partida ha acabado. Esta función se ejecutará después de cada acción realizada por un jugador para ver si alguno de los cuatro jugadores ha conseguido alcanzar los diez puntos de victoria. En caso de que exista un jugador que cumpla esas condiciones, la partida llega a su fin y él será el ganador, en caso contrario, la partida sigue su curso con normalidad.

La siguiente función es la encargada de repartir las cartas tras tirar los dados al inicio de cada turno. Esta función comprueba cada una de las casillas dependiendo del número que tenga asociado y si coincide con el número que hay en los dados se revisan las seis encrucijadas de dicha casilla en busca de algún poblado o ciudad que pudiese haber colocado y en caso de encontrar alguno, reparte las cartas pertenecientes al propietario del edificio. Esta función solo se usa en caso de que el número que haya salido en los dados sea distinto de siete porque como se ha explicado previamente, si la suma de los dados es igual a siete, ninguna casilla producirá recursos y sucederá el evento del ladrón.

A continuación, se crea una función cuyo propósito es simular una tirada de dados. Para lograr un resultado lo más aleatorio posible se usa la biblioteca de Python "Random", una biblioteca que incluirá varias funciones de obtención de valores pseudoaleatorios que aseguran un resultado que sigue la distribución previamente descrita. La forma de obtener el resultado aleatorio es pedir a la biblioteca dos números aleatorios del uno al seis con todos los posibles resultados con igual probabilidad de aparición, así simulamos dos dados y la suma de los dos será el resultado para el turno en juego. Se completa la funcionalidad de este código añadiendo la función descrita anteriormente, así como el algoritmo que procede en caso de que el resultado de los dados sea un siete. En este probable caso, primero se comprueba que ninguno de los jugadores posee más de siete cartas de materia prima, en caso de que hubiese algún jugador con más de siete cartas, está obligado a descartarse de la mitad de las cartas de materia prima antes de que continue la partida. En este caso, el agente inteligente puede elegir qué cartas descartarse, llegando a una variedad inmensa de opciones de descarte cuando se poseen un gran número de cartas. Posteriormente, como se ha explicado en la introducción, el jugador que ha realizado la tirada de dados es el encargado de mover el ladrón a cualquier casilla distinta de la actual y como recompensa puede robar una carta de materia prima a cualquiera de los jugadores que tuviese un poblado o ciudad adyacente a la casilla de destino.

Relacionado con el ladrón se encuentra la función de descartar cartas que entra en juego cuando el resultado de los dados sea un siete y alguno de los jugadores cuente con más de siete cartas de materia prima en sus manos. Esta función es algo compleja y devuelve una lista con todas las posibles opciones de descarte del jugador para que se elija posteriormente en otra función que se describe más adelante. Como la función devuelve todas las posibles combinaciones de cartas que posee en la mano, nos encontramos con una lista de posibilidades con un número muy alto de elementos. Es un problema de combinatoria que se resuelve creando una lista de todos los recursos que posee el jugador y aplicando la fórmula de la combinatoria sin repetición:

$$C_{n,x} = \binom{n}{x} = \frac{n!}{x!\,(n-x)!} \tag{1}$$

Tabla 2. Ecuación de Combinatoria [18].

Como se puede ver en la fórmula, se trata con factoriales de números y eso hace que el número de posibilidades que se añadan a la lista pueda crecer muy rápidamente.

Para ello se creó un pequeño programa de pruebas que imprimía el número de posibilidades que devolvía la función con distintos números de cartas en la mano y se comprobó que a partir de 26 cartas en la mano el número es demasiado elevado para calcular todas las opciones disponibles, en concreto para 27 cartas habría más de 20 millones de posibilidades de descarte. Este suceso llevó a que se tomaran medidas que se explican más adelante para reducir la posibilidad de encontrar a jugadores con más de 20 cartas en la mano.

A continuación, se desarrollan una serie de funciones simples que ejecutaran las acciones disponibles por los jugadores, como pueden ser construir ciudades, carreteras o poblados.

Lo primero de todo es crear dos funciones, una para las carreteras y otra para los poblados, que será de ayuda cuando un jugador quiera construir un edificio. Las funciones tienen como parámetros de entrada el jugador que quiere realizar la acción y la casilla deseada, la función devuelve si es posible o no. En el caso de la función dedicada a las carreteras, únicamente comprueba que no haya ninguna carretera en la ubicación solicitada y si en alguna de los caminos colindantes existe otra carretera del mismo jugador que lo solicita. En cambio, en el caso de los poblados, la función debe comprobar si la encrucijada solicitada y todas las adyacentes están vacías y también si en alguno de los caminos adyacentes se encuentra alguna carretera del mismo jugador que quiere construir el poblado. Por último, para las ciudades únicamente es necesario comprobar que en la casilla solicitada haya un poblado, por lo que dicho requisito se comprueba directamente en la función que se ha explicado previamente de la clase Player.

Lo siguiente será crear tres funciones que, tras haber comprobado que el lugar está disponible, construyan el edificio solicitado por el jugador. En estas funciones únicamente se cambian las variables que existen dentro de la clase como puede ser el objeto de la clase *Building* que se encuentra dentro del objeto de la clase *Corner* de la encrucijada donde se va a construir el poblado o ciudad. En estas funciones también se actualizan los puntos de victoria y el número de edificios disponibles por los jugadores para seguir construyendo.

A continuación, se crea una función que sirva para repartir las cartas de progreso. Dichas cartas están representadas en una lista dentro del programa que contiene todas

las cartas que existen al inicio de la partida. La función en cuestión saca una carta de desarrollo al eliminar uno de los elementos al azar de la lista. Al hacerlo el número de cartas restantes será menor. De esta manera se asegura un reparto aleatorio de las cartas y también que no se repartan más cartas de un tipo en concreto ya que en el caso de robar todas las cartas de un mismo tipo, ya no quedarían cartas de dicho tipo dentro de la lista.

También se crea una función únicamente para robar una carta de manera aleatoria de uno de los jugadores y entregársela al jugador del turno actual. Esta función es de utilidad cuando se mueve el ladrón a una casilla. Del mismo modo, se crea otra función que comprueba si al activar un caballero, el jugador que lo ha activado es el jugador con más caballeros en frente suya y por tanto le pertenece el bonus por el mayor ejército.

Siguiendo con los bonus, también es necesaria una función que contabilice el número de carreteras consecutivas tiene cada jugador para así asignar el bonus al jugador que tenga el mayor número. Para realizar un seguimiento del número de carreteras se implementará un algoritmo de búsqueda que, partiendo de una carretera de un jugador, recorre todas las carreteras adyacentes hasta que no queda ninguna del mismo color unida al camino de carreteras, y devuelve el mayor número de carreteras seguidas. El algoritmo de búsqueda que se ha usado es *Breadth First Search*, un algoritmo que va recorriendo todos los nodos del problema que estén al mismo nivel (en este caso, el mismo número de carreteras del mismo color consecutivas) hasta que se han comprobado todos los nodos de este nivel y a continuación pasa a los nodos que se hayan descubierto en el siguiente nivel. Un ejemplo del algoritmo se puede ver en el *GIF* que se encuentra en los anexos (Breadth-First-Search-Algorithm.gif) y en las referencias [19].

Este algoritmo tendrá algunas modificaciones para adaptarlo al problema de las carreteras. La modificación más significativa es añadir una condición que permita analizar si alguna de las carreteras adyacentes a la carretera a analizar era también adyacente a la última carretera que ha sido analizada, si esta condición se diera, se estaría ante una bifurcación y por tanto no contaría para el mayor número de carreteras consecutivas:

Fig. 13. Explicación de la ruta comercial [15].

Por ejemplo, en la Fig. 13. Explicación de la ruta comercial [15].**¡Error! No se encuentra el origen de la referencia.** se puede ver que el jugador rojo tiene colocadas ocho carreteras sobre el tablero, pero al haber una bifurcación solo se podrán contar seis carreteras consecutivas como máximo.

Acabando con la parte de software se necesita crear una función que tenga como parámetro un elemento de la lista de todas las posibilidades que se han comentado anteriormente y dado cualquier elemento pueda realizar la acción correspondiente. Para ello será necesario usar todas las funciones que hemos descrito hasta ahora. La función consistirá en un bucle condicional que analizará cual es la acción que se ha pasado como argumento y la ejecutará dependiendo de qué acción sea y con qué argumentos. Existirán un total de catorce posibles de acciones a realizar, entre las cuales se puede destacar el caso de jugar carta de desarrollo que no tenga función a parte como puede ser el monopolio o las carreteras gratis, o el hecho del comercio marítimo que hará el intercambio de materiales dependiendo de si el jugador actual tiene puertos o no para establecer el número de cartas que intercambiar.

La última función que tendrá la clase *Game* será una que decida qué acción tomar. Esta función será la más importante del proyecto y por eso se explicará más a fondo más adelante. Dentro de este algoritmo podemos distinguir dos partes, una que será el agente inteligente y que irá aprendiendo y otra parte que será la que estará predeterminada al resto de jugadores que tendrán acciones pseudoaleatorias ya que estarán guiadas por un algoritmo creado para que las acciones tengan un cierto criterio de selección dentro de la aleatoriedad de la decisión.

La función principal es *get_next_action* cuyo propósito es devolver el índice de la acción que se quiere realizar de la lista de todas las acciones posibles. Para ello se realiza una operación que separa la lista de las posibles acciones y crea otra que contiene únicamente los tipos de acciones que se pueden hacer sin tener en cuenta sus argumentos. De esta manera se divide la decisión de qué acción tomar en dos partes: el tipo de acción a tomar y, posteriormente qué parámetros elegir para esa acción. Un ejemplo del algoritmo en ejecución es el siguiente:

```
La lista de posibles acciones es:
 [0, (1, 40), (1, 41), (1, 44), (1, 46), (1, 51), (1, 52), (1, 64
), (1, 65), 9, (12, 1, 1, 2, 1), (12, 1, 1, 3, 1), (12, 1, 1, 4,
1), (12, 1, 1, 5, 1), (12, 2, 1, 1, 1), (12, 2, 1, 3, 1), (12, 2,
 1, 4, 1), (12, 2, 1, 5, 1), (12, 2, 2, 1, 2), (12, 2, 2, 3, 2),
(12, 2, 2, 4, 2), (12, 2, 2, 5, 2), (12, 2, 3, 1, 3), (12, 2, 3,
3, 3), (12, 2, 3, 4, 3), (12, 2, 3, 5, 3)]
La lista de tipos de acciones es:
 {0, 1, 12, 9}
```

Fig. 14. Listas de acciones y tipos.

Una vez que se han obtenido estas dos listas, se procede a tomar la decisión. En este momento se divide la función a su vez en dos funciones, una ejecutada por los jugadores normales y otra que ejecuta el agente inteligente.

La primera función se llama *AlmostRandomDecision* ya que es la encargada de tomar acciones aleatoriamente, pero con algunos criterios para evitar partidas muy largas. Partiendo de la lista de tipos de acciones que se ve en la Fig.15, se establecen unas ponderaciones que se pueden ajustar según conveniencia antes de que empiece la partida que hacen que la decisión no sea completamente aleatoria y de esta manera dar más peso a algunas acciones que son más importantes que otras dentro del juego. Esta es una manera de reducir las posibilidades de que los jugadores almacenen cartas en sus manos y posteriormente pueda haber problemas a la hora de descartar cartas. Al aumentar las probabilidades de que un jugador construya edificios o intercambie cartas con la banca se reducen el número de cartas que tiene el jugador al finalizar el turno y por ende el número de cartas que puede llegar a tener al iniciar los turnos de los demás jugadores. Esta medida también ayuda a que las partidas sean más rápidas al aumentar las posibilidades de que un jugador construya un poblado o una ciudad para así conseguir más puntos de victoria. También se ha añadido una medida de seguridad para evitar que los jugadores acaben su turno con demasiadas cartas de recursos en la mano.

Esta medida será hacer que la probabilidad de pasar de turno cuando el jugador tenga más de doce cartas en la mano sea nula. Siempre podrá hacer otras acciones como construir o comerciar, pero estará obligado a descartarse cartas por el algoritmo para así evitar el problema que se ha mencionado previamente con la combinatoria.

La siguiente función llamada *NNdecision* es la más importante ya que es la que determina el aprendizaje del agente. Esta función es la parte más importante del trabajo ya que es la encargada de aprender a ejecutar las acciones correctas. El desarrollo y la explicación del algoritmo se explica a continuación en el siguiente apartado.

Para concluir con este apartado, se dispone de un diagrama de flujo que explica cómo se desarrolla el control del programa que ejecuta la partida de Catán.

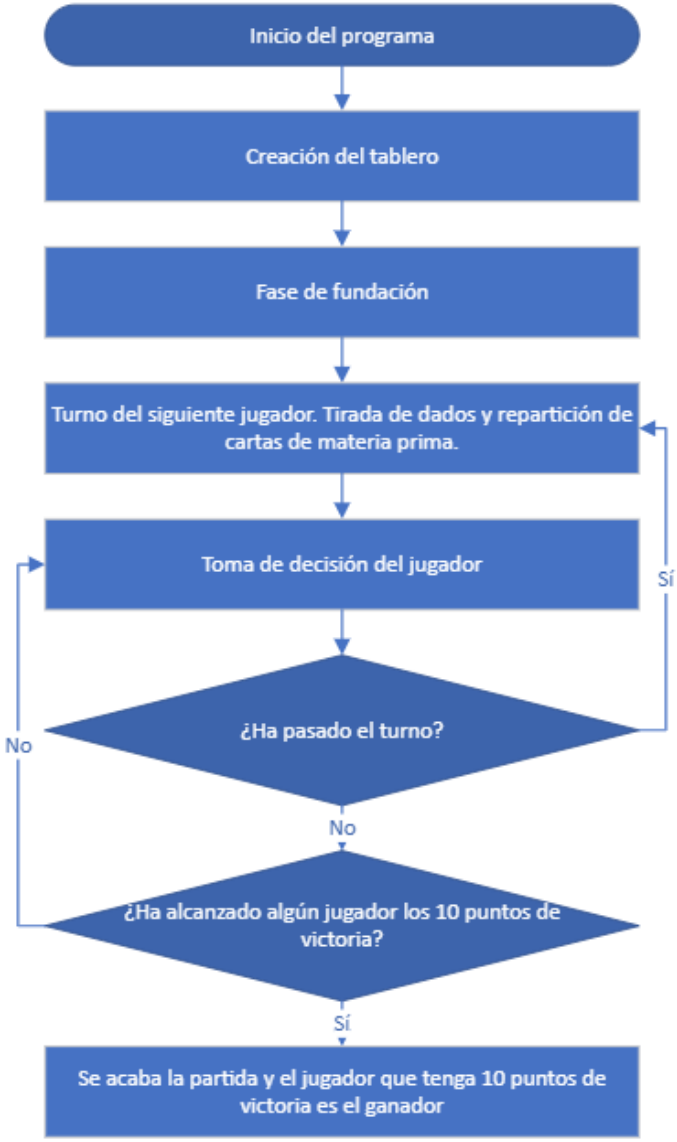

Fig. 15. Diagrama de control del programa.

4.3 Aprendizaje del agente

Una vez diseñado todo el mecanismo de juego y una interfaz visual, es el momento de crear un agente inteligente que sea capaz de ganar a los jugadores que se han creado que juegan de manera casi aleatoria. Para ello se hace uso de los algoritmos de aprendizaje automático que ya hemos explicado previamente.

4.3.1 Redes neuronales

A pesar de que existen librerías que contienen redes neuronales en Python, son librerías muy grandes que implementan muchas funciones y características a las redes que no son útiles para este proyecto y que hacen que la ejecución sea más lenta. Es por lo que se ha optado por realizar una implementación de las redes neuronales creada desde cero.

Para crear las redes neuronales hay que separar el trabajo en dos partes, la creación de la estructura, y la ejecución y obtención de la salida de la red.

- **Creación de la estructura de la red neuronal**

  Para crear la red neuronal hay que tener en cuenta qué es lo mínimo necesario para que la red proporcione una salida fiable. En el caso del Catán lo único que debe tener una red neuronal son las capas con un cierto número de neuronas, y para cada capa, una matriz que contenga los pesos de cada neurona para cada entrada y un vector con las ganancias estáticas.

  Dentro de Python se crea una clase llamada Capa que únicamente contendrá el tamaño de sí misma, el tamaño de la capa anterior y los dos elementos mencionados en el párrafo anterior. De esta manera tenemos un tipo de datos lo más sencillo posible y así se evita el procesamiento de datos innecesario.

  También se crearán cuatro funciones dentro de la clase, dos para calcular el tamaño de la matriz y del vector basándose en los tamaños de la capa actual y de la anterior, y dos funciones que se encargan de rellenar la matriz y el vector a partir de un vector de valores introducido por parámetro.

  La red por tanto consistirá en una lista de instancias de clase Capa que cada una tendrá un número de neuronas distinto y con pesos y ganancias distintos. Para crearla únicamente es necesario proporcionarle un vector con los valores de los

pesos y de las ganancias estáticas, el tamaño de las capas ocultas en forma de vector (cada posición del vector indica el número de neuronas en una capa), el número de entradas y el número de salidas. Esta función aporta mucha versatilidad ya que con una misma función se pueden crear redes de todos los tamaños y con cualquier número de capas.

- **Ejecución de la red neuronal**

Para la ejecución de la red, se ha creado una función que recibe como parámetros una red neuronal como la que se ha explicado en el párrafo anterior y una lista de entradas. A partir de estos parámetros se obtiene la salida deseada.

Para llegar a este punto es necesario crear también la función de activación de la red neuronal que en este caso es la función sigmoide:

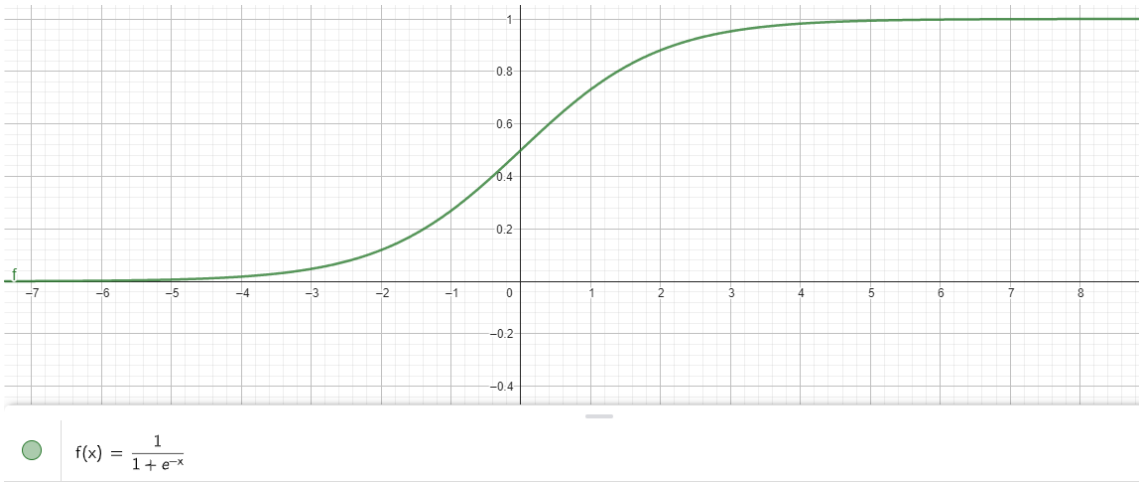

Fig. 16. Función sigmoide [19].

Esta función es útil porque siempre devuelve un valor entre 0 y 1, lo cual se puede usar para representar el nivel de certeza de realizar una acción o un valor normalizado de un rango de valores entre los cuales se puede elegir. Por su versatilidad a la hora de ser interpretado, es una función perfecta como función de activación.

Para implementarla se creará una función que realice la operación matemática una a una del vector de entrada de la neurona hasta devolver un vector de salida que estará compuesto por la salida de la función sigmoide de todos sus elementos.

Combinando los dos apartados anteriores se pueden crear redes neuronales de cualquier tamaño y que al introducirle una entrada es capaz de obtener una respuesta fiable. Ahora es el turno de configurarlas y entrenarlas.

4.3.2 Estructura de la red neuronal

Como el problema que se plantea es muy complejo, el algoritmo que se desarrolla no tendrá una única red neuronal, sino que para cada acción posible a realizar se creará una subred que será la encargada de proporcionar los argumentos para realizar la acción más adecuada.

El algoritmo deberá seguir el siguiente diagrama:

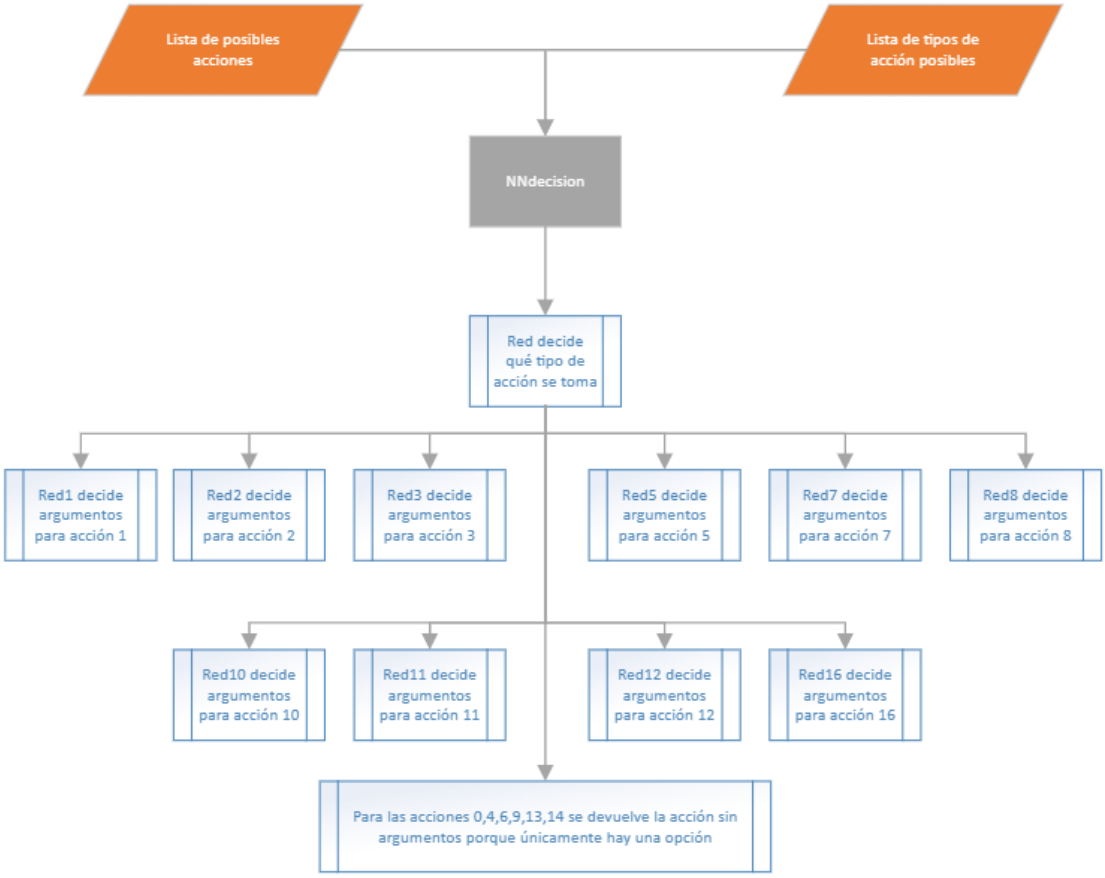

Fig. 17. Estructura de las redes neuronales.

Como se puede ver en la imagen anterior, el algoritmo primero solicitará una acción a la red neuronal general que devolverá una clase de acción entre las 17 que hay disponibles para hacer:

0. Saltar el turno
1. Construir una carretera

2. Construir un poblado
3. Construir una ciudad
4. Comprar una carta de desarrollo
5. Jugar una carta de monopolio
6. Jugar una carta de carreteras
7. Jugar una carta de caballero
8. Jugar una carta de invento
9. Mover el ladrón
10. Descartarse de cartas
11. Comercio marítimo
12. Comercio doméstico
13. Aceptar trato
14. Rechazar trato
15. Construir una carretera gratis
16. Construir poblado y luego una carretera

Una vez que la red principal ha decidido qué acción se va a realizar, las diferentes subredes se encargan de decidir qué argumentos se añaden a la acción para optimizar la cantidad de puntos de victoria que se obtienen a lo largo de la partida.

Hay algunas acciones que requieren un único argumento como pueden ser las tres funciones de construcción, cuyo único parámetro sirve para indicar donde se construye, existen otras que requieren más argumentos como las dos acciones del comercio o la acción de usar la carta de invento que requieran dos argumentos o más. Habrá algunas acciones que no requieran una subred ya que no necesitan ningún argumento como pueden ser la acción de saltar el turno o la acción de comprar una carta de desarrollo.

Una vez que se ha obtenido la salida de las redes, la salida es una acción que debe estar en la lista de posibles acciones que se ha pasado por parámetro a la función *NNdecision*. En el caso de estarlo, la función devuelve la posición de dicha acción en la lista y de esta manera, la función *get_next_action* que se ha explicado previamente devolverá la acción correspondiente. En caso de que la acción que ha salido de la red no exista, el programa debe acabar con error y de cara al entrenamiento recompensar el algoritmo negativamente para evitar estos errores.

4.3.3 Parámetros de la red neuronal

Una vez definidas las redes neuronales que se van a utilizar, el siguiente paso es el entrenamiento de la red, lo primero que se ha tenido que concretar es el número de entradas, el número de salidas, el número de capas intermedias y el número de neuronas que debe haber en cada capa.

Empezando por el primer elemento, la entrada de la red neuronal debe ser el estado actual de la partida en el momento de realizar la acción. Para ello, los datos se extraerán de objetos de las clases que se han explicado previamente que almacenan todos los datos de la partida como pueden ser el contenido de una encrucijada en concreto, los puntos de victoria de los jugadores o el número de cartas de cada tipo de recurso que tiene el jugador en la mano. Al juntar todas las variables que constituyen el estado del tablero y los jugadores en el momento de la acción, se cuentan un total de 279 variables. A parte de estas variables de entrada que serán comunes para todas las redes neuronales, cada una de las subredes tiene una lista de entradas específicas para tomar la decisión de elegir los argumentos de la acción a realizar. Existen redes, como por ejemplo las subredes de construcción, que no necesitan información extra ya que tienen toda la información que necesitan en la lista de entradas que se ha descrito antes. A pesar de que el número de variables de entrada cambie con respecto a la red concreta, casi todas las redes tienen el mismo número de entradas.

Posteriormente hay que concretar el número de salidas y, al igual que en el apartado anterior, depende de la red en concreto que tome la decisión. En el caso de la red principal, tiene una lista de 17 salidas, una por cada tipo de acción a realizar. Esta salida indicará el porcentaje de seguridad de la red de realizar dicha acción en ese momento. Una vez que la red ha propuesto su salida, el algoritmo irá probando a realizar la acción que tenga más seguridad y en caso de no poder pasará a la siguiente con mayor seguridad y así hasta encontrar una que sea posible. En el caso de las subredes dependerá de los argumentos necesarios de las acciones que representan cada subred. Las redes tendrán una salida por cada argumento que requiera la acción, llegando desde uno como en el caso de mover al ladrón, hasta cinco como es el caso de la acción de descartar cartas. Estas salidas se tratan de distinta manera en cada caso ajustándose a las necesidades de cada parámetro.

En el caso del número de capas y el número de neuronas, estos valores se podrán ir cambiando a lo largo del entrenamiento ya que cuanto más altos sean, mayor volumen de información se podrá tratar, pero más complejo se volverá el entrenamiento. A base de prueba y error, se irán ajustando dichos números. El volumen de neuronas es independiente entre la red principal y todas las subredes, pero al tener una lista de entradas muy parecida, será muy similar entre todas las redes.

4.3.4 Función de recompensa (*Fitness Function*)

Una parte muy importante del entrenamiento es la definición de la función de recompensa que se introduce en el algoritmo genético y que será la encargada de analizar el éxito o el fracaso de la red neuronal en el problema.

Al ser una función dentro del algoritmo genético, está obligada a tener una forma en concreto para poder usarse correctamente. La función debe tener tres parámetros que son: una instancia del algoritmo genético, una solución (o cromosoma) de la generación actual que pueda resolver el problema, y el índice de esta misma población dentro del cromosoma. En este caso se ha creado una función que creará la red principal y todas las subredes a partir de un vector que se introduce en la población, es decir, las variables que almacenan los pesos y las ganancias de la red principal y todas las subredes irán concatenados en el vector de población.

El siguiente paso es separar las variables que corresponden a cada red y utilizar la función que hemos explicado previamente para crear redes del tamaño necesario para cada subred. Estos tamaños podrán ir cambiando a lo largo del entrenamiento por lo que también irá cambiando la función de recompensa a lo largo del entrenamiento.

Por último, se ejecuta el juego introduciendo estas redes como parámetro y se obtiene la salida. Si la salida es que el jugador inteligente ha ganado, se suma un punto a la recompensa de la población si no, la recompensa se mantiene igual. Este proceso se realizará 100 veces con la misma población para que así se prueben distintos escenarios con la misma configuración. Al final de la ejecución del bucle, se comprueba que la recompensa es un porcentaje de victorias frente a los jugadores aleatorios. Este valor es el que se devuelve al algoritmo genético

4.3.5 Entrenamiento de la red con Algoritmo Genético

Como hemos mencionado previamente, los algoritmos genéticos se pueden usar en combinación con redes neuronales para resolver problemas de aprendizaje por refuerzo como es el caso. Haciendo uso de la librería PyGad se creará una función que ejecute el algoritmo que hará que el agente inteligente aprenda.

Se crea un programa que realiza todo el proceso. Este no requerirá de muchas líneas de código, ya que, en cuanto al algoritmo todas las operaciones son ejecutadas por la librería. Esta función simplemente crea una instancia del algoritmo introduciendo como parámetros el número máximo de generaciones que debe ejecutar el algoritmo, el número de cromosomas que se crearán por generación, cual es la función de recompensa, el número de variables a optimizar, los límites inferiores y superiores de dichas variables, la probabilidad de mutación…Finalmente se ejecuta la función *run* y el propio algoritmo es el encargado de ejecutar la función de recompensa para cada población de cada generación y de realizar todos los cruzamientos y de obtener una solución óptima.

Al final, se ejecutarán un número de soluciones por generación y un número concreto de generaciones elevando el número de partidas jugadas por la inteligencia artificial a un número considerable. Es por esa razón por la que la implementación del juego debe ser fluida y no consumir muchos recursos. Para ello, como se ha explicado anteriormente, el desarrollo se ha realizado de manera que hubiese una separación entre la ejecución del juego y la visualización de este. De esta manera, se puede deshabilitar la visualización del juego y lograr que la partida no consuma demasiados recursos y, por tanto, se pueda ejecutar más rápidamente. De cara a acelerar el entrenamiento, se ha logrado conseguir una velocidad de alrededor de las 100 partidas por minuto.

5 Resultados

En este apartado se exponen los resultados obtenidos a lo largo de la creación del proyecto, tanto en la parte gráfica como en el entrenamiento del agente.

Los primeros resultados se obtienen al acabar de desarrollar la parte gráfica del juego, cuando se implementa correctamente todos los elementos que se han descrito en el apartado 4.2. Al ejecutar el código se puede ver una partida entre cuatro jugadores aleatorios de manera gráfica. Estos jugadores juegan de manera automática y al ir cambiando las variables que almacenan los datos de la partida, la imagen en la pantalla se va actualizando. Cada juego tiene una generación aleatoria, es decir, cada partida que se ejecuta tiene un tablero distinto con materiales y números distintos lo que hace que la probabilidad de conseguir un recurso en concreto cambie mucho entre partidas. No será tan fácil construir una carrera y un poblado en una partida en la que todas las casillas de la arcilla tienen un número bajo que una partida en la que las probabilidades de los números se hayan repartido entre todos los tipos de material.

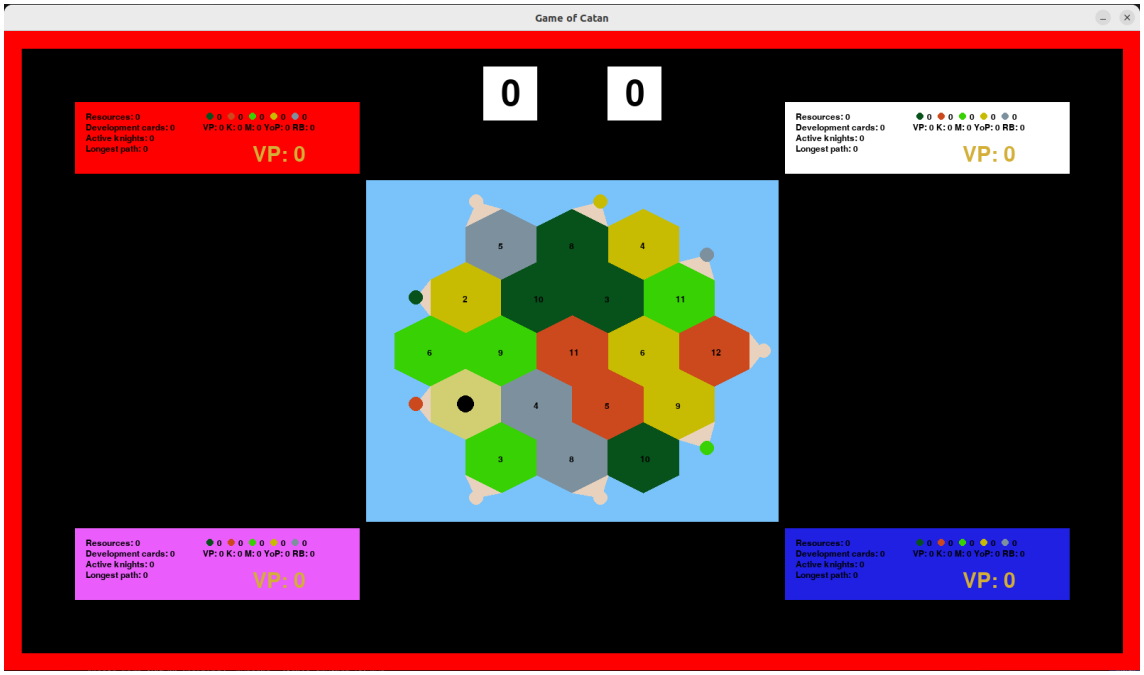

Fig. 18. Tablero vacío.

En la imagen anterior se puede ver el resultado de la implementación cuando el tablero está vacío. A medida que los jugadores vayan poniendo sus poblados se irá actualizando como se ve en las siguientes figuras. La figura anterior explica perfectamente como es una partida equilibrada en cuanto a materia prima se refiere.

Todas las casillas de materia prima tienen al menos un número cuya probabilidad no sea muy baja y por tanto es muy probable que a lo largo de la partida todos los jugadores puedan obtener todas las materias primas. En cambio, si en el ejemplo anterior el hexágono de arcilla que tiene un 5 se hubiese colocado en el hexágono que tiene un 2, toda la producción de arcilla quedaría limitada a una probabilidad muy baja y, por tanto, existiría durante toda la partida una deficiencia de arcilla que afectaría a todos los jugadores.

Otro factor a tener en cuenta es la posición inicial de los jugadores. En la fase de asentamiento se determina qué materiales son los más probables que el jugador consiga a lo largo de la partida. Es por lo que una mala colocación de los poblados iniciales es determinante a la hora de obtener la victoria en una partida. En concreto, en una partida, realizar una buena colocación de los poblados iniciales puede otorgar la posibilidad de obtener todos los tipos de materia prima en el segundo turno de juego, en cambio una mala colocación de los mismos puede provocar que haya algún recurso que sea imposible de obtener por la tirada de dados al inicio de la partida y, por tanto, difícil de conseguir durante el resto de la partida:

Fig. 19. Colocación inicial del tablero.

En este ejemplo se puede ver que la colocación de los jugadores puede decidir la partida. En el caso del jugador azul, su colocación le ha llevado a únicamente poder obtener dos materiales de materia prima a través de la tirada de los dados, lana y

madera. Además, la probabilidad de que los números que proporcionan esas materias sean el resultado de la tirada de dados es muy baja. En esta partida, la victoria del azul es prácticamente imposible. Por otro lado, el rosa ha colocado sus dos poblados de manera que a través de la tirada de dados pueda obtener todas las materias primas, aunque para conseguirlo haya tenido que ponerlo en un sitio que la probabilidad sea baja como el caso del cereal que solo lo puede conseguir con el 12. Aunque la estrategia más interesante es la del jugador rojo (el agente inteligente). El jugador rojo ha decidido poner un poblado en el puerto de cereales para poder intercambiar con la banca a razón de 2:1 y de esta manera, con el otro poblado, situarlo en un sitio que produzca mucho cereal como es el caso. De esta manera, a pesar de que no pueda obtener todas las materias primas, sí que podrá cambiarlo por el exceso de cereal que es muy probable que obtenga a lo largo de la partida.

Por otro lado, se obtuvo un resultado inesperado que obligó a realizar cambios en el código. A pesar de las medidas implementadas para evitar la acumulación de cartas en las manos de los jugadores, se comprobó que, en los turnos más avanzados, con el aumento de edificios en el tablero, las cartas de materia prima se obtienen de manera más rápida y por tanto es más difícil evitar la acumulación de cartas. Es por lo que, a pesar de que las reglas sí lo permitan, se establece un límite de 20 cartas que un jugador puede tener en la mano. A partir de ese número, a pesar de que la tirada de dados sea fructuosa para el jugador, solo se repartirán cartas hasta que tenga un máximo de 20 cartas y las demás se descartarán. De esta manera, se evita el aumento de cálculos a realizar en caso de que salga el siete en los dados y haya que realizar un descarte de cartas.

Una ejecución del código se puede ver en el video que está dentro del repositorio del anexo [A.4].

A continuación, se entrena el agente para que aprenda a jugar al Catán. Las primeras pruebas se hacen con una estructura compleja. Cada una de las redes tiene como estructura 279 entradas (algunas más como hemos mencionado previamente), 4 capas ocultas con 300, 150, 80 y 8 neuronas cada una respectivamente y una salida o varias dependiendo de la red. Una vez definida la estructura, se configura el algoritmo genético de manera que ejecute 100 generaciones con 10 soluciones cada una.

Tras la primera ejecución del algoritmo se comprueba que la puntuación obtenida es nula debido a que la red al no estar entrenada devuelve en la mayoría de los casos una acción que no existe dentro de las posibles acciones a realizar. Es por eso que, a pesar de haber jugado más de un millón de partidas, el entrenamiento llega a su fin sin haber concluido correctamente ninguna de ellas.

La primera solución que se adoptó fue simplificar la red. En el momento de la primera ejecución, el número de pesos y ganancias de las redes que el algoritmo tenía que ajustar era de más de 30000. Al tener una estructura tan compleja, se buscaba un comportamiento que se adapte bien a las distintas fases de juego, pero requería de un entrenamiento muy grande y que ejecutase demasiadas partidas. Es por eso que la primera medida que se adoptó fue reducir el número de capas de la red y también el número de neuronas en cada capa. A través de diversos entrenamientos fallidos se alcanzó un tamaño mínimo de dos capas por red neuronal con 4 neuronas en la primera capa y 10 neuronas en la segunda. Con esta configuración alcanzamos un número de variables a optimizar de 12060. A pesar de reducir el número de parámetros que se introducen al algoritmo, el resultado sigue siendo el mismo, la red no es capaz de acabar una partida sin devolver una acción que exista dentro de la lista de posibles acciones. Se emplearon alrededor de 200 horas de entrenamiento hasta llegar a la conclusión de que ninguna de las configuraciones funciona.

Para seguir reduciendo la implementación de la red, se optó por reducir aún más la estructura de la red. Y para ello, se eliminaron las subredes que elegían los parámetros de las distintas acciones, ya que eran las que devolvían acciones que no existen. De esta manera, se reducen en gran medida la cantidad de variables a optimizar y se permite la finalización de las partidas y de esta manera obtener la recompensa positiva en caso de que el agente logre conseguir los 10 puntos de victoria. La estructura del algoritmo sigue el siguiente diagrama:

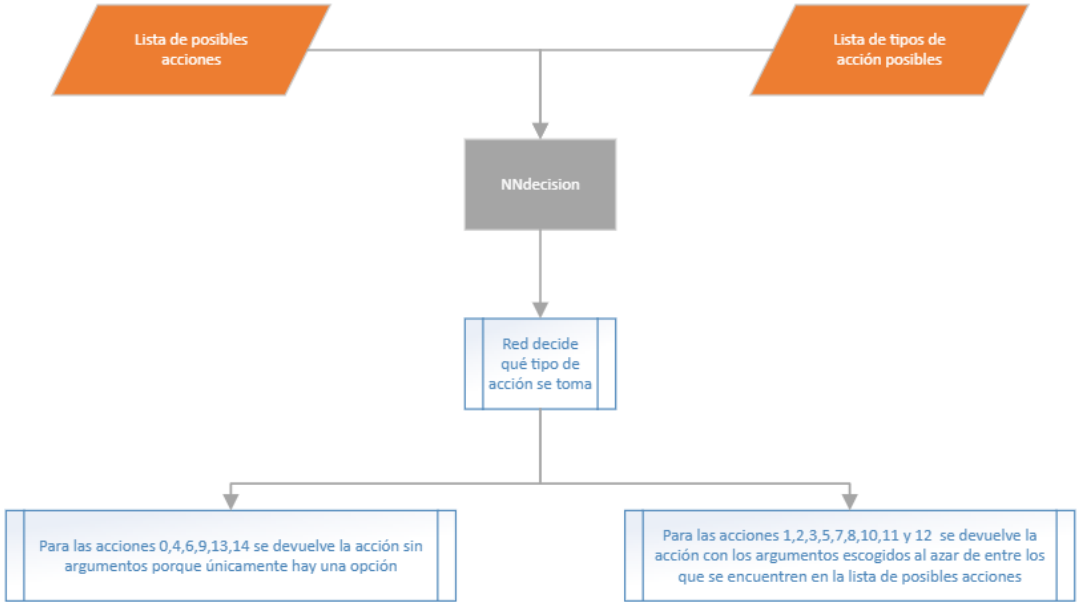

Fig. 20. Diagrama de la nueva red

Habiendo realizado estos cambios, la red es capaz de finalizar el entrenamiento con los mismos argumentos que se habían introducido en el algoritmo genético al inicio del entrenamiento. Al haber arreglado el problema de las acciones que no existen, las partidas que se jueguen con esta estructura siempre acabarán. A consecuencia de esto, el tiempo de entrenamiento se multiplica y se requiere de más recursos para llegar a una solución que, en efecto, pueda ganar a jugadores aleatorios.

El siguiente resultado obtenido es que, tras haber entrenado la red durante 50 horas aproximadamente, existen partidas que no finalizan y por tanto el entrenamiento se interrumpe y no acaba nunca. Esto se debe a un suceso que no se había contemplado previamente y es que debido a la aleatoriedad del mapa y de los jugadores, la partida llega a un estado en el que ninguno de los jugadores tiene la oportunidad de ganar independientemente de las acciones que elija, llegando a jugar una partida de 300000 turnos sin que pueda acabar. Es un suceso que no se contempla en las reglas debido a que es tan remoto que, a pesar de jugar muchas partidas, en el entrenamiento únicamente sucedió 2 veces en 150000 partidas aproximadamente. Cuando se da este caso, la medida que se opta para solucionarlo es simplemente limitar los turnos dentro de una partida a 1000. Siguiendo los ejemplos del entrenamiento, se diferencia que todas las partidas acaban antes de los 1000 turnos con una victoria de alguno de los

jugadores, por eso, si en alguna partida se supera ese número, significa que se encuentra en un punto muerto y se declara que todos los jugadores han perdido.

Una vez solucionado el problema, se entrena la red durante aproximadamente otras 200 horas hasta obtener un resultado donde al menos se obtiene un porcentaje de victorias de un 25%. A pesar de que el propio entrenamiento ya realiza una evaluación de la solución, se crea otro pequeño programa que ejecutará 100 partidas y devolverá el porcentaje de partidas ganadas con unos valores de pesos y ganancias obtenidos de un entrenamiento concreto.

A continuación, se ejecuta una partida con el agente entrenado. El video de la partida se encuentra dentro del repositorio del anexo [A.4].

En la partida podemos ver cómo el jugador rojo es capaz de expandirse por el mapa, pero, sobre todo, compra muchas cartas de desarrollo. Esta estrategia es la que hace que el agente gane la partida, a través de los puntos de victoria que le otorgan las cartas y el bonus por ser el jugador con más caballeros logra ser el primer jugador en llegar a los 10 puntos de victoria.

Tras realizar varias pruebas durante muchas horas de entrenamiento se obtienen diversas soluciones al problema que se quedan cerca del 25% en el porcentaje de victorias dependiendo de la ejecución y la aleatoriedad de las partidas. El porcentaje máximo obtenido es que, en 100 partidas, durante el entrenamiento una generación logró obtener 30 victorias.

Para concluir, encontramos que a pesar de la aleatoriedad, sí que hay casos en los que el agente logra superar a los jugadores aleatorios como es el caso anterior en el que obtiene un 30% de victorias.

6 Conclusiones

En este último capítulo, se describen las conclusiones que se han obtenido de este proyecto, las competencias que se han empleado y las nuevas competencias adquiridas. Además, se describen algunas posibles líneas de mejora para implementar en el futuro.

6.1 Conclusiones finales

Este TFG ha profundizado en la simulación de juegos con Python, creando un simulador del juego de mesa Catán que es un juego extenso y con mucha interacción y toma de decisión.

El mayor reto que se ha encontrado al realizar el proyecto es la complejidad del problema. El Catán es un juego con mucha estrategia y gestión de recursos que al final hacen que la ejecución y sobre todo el aprendizaje sean complicados.

Para ejecutar el agente se realizó una simulación fluida del juego que es capaz de representar el juego de manera visual a tiempo real de partida, sirviendo como soporte para el aprendizaje del propio agente.

Por último, se ha comprobado que la complejidad del problema es muy alta ya que a pesar de las múltiples horas empleadas entrenando al agente con la simulación, la estructura necesaria para que el agente consiga una tasa de victorias mayor, es demasiado extensa y no es viable entrenarla en un tiempo razonable con los recursos disponibles.

6.2 Competencias empleadas

Esta memoria recoge toda la información relevante recabada a la hora de realizar el Trabajo de fin de grado, es por ello por lo que muchos conocimientos que se habían adquirido durante los años de estudio han sido empleados.

Uno de los conocimientos empleados más importantes, sin el cual este proyecto no se podría haberse llevado a cabo es la programación. Desde el inicio de la carrera las asignaturas dedicadas a la programación han sido numerosas y se han estudiado muchos tipos de programación en distintos lenguajes como Introducción a la Programación que fue la primera asignatura en Python, Algoritmos y Estructuras de Datos en la cual vimos programación orientada a objetos en Java o Sistemas Operativos en la cual se aprendió a

manejar el sistema en lenguaje C. Estas son algunas de las asignaturas que han fundado una base de programación suficiente para realizar un proyecto tan extenso como el que se ha explicado en esta memoria.

Dentro de la programación, hay que destacar el aprendizaje de diseño gráfico que se impartió en las asignaturas de Sensores y Actuadores y Diseño y Simulación de Robots cuyas enseñanzas a pesar de estar orientadas a los robots han sido muy útiles a la hora de realizar la parte del diseño gráfico del Catán.

Por último, el conocimiento más empleado a la hora de realizar este trabajo ha sido el que se ha estudiado en las asignaturas de Inteligencia Artificial y Aprendizaje Automático. En estas asignaturas se enseñaron conceptos como Algoritmo de Búsqueda, Redes Neuronales, Algoritmo Genético y por supuesto Aprendizaje Automático. Cursar estas asignaturas fue la inspiración de este proyecto debido a la teoría y sobre todo las prácticas realizadas durante el curso.

6.3 Competencias adquiridas

Tras la realización del proyecto se han adquirido algunas competencias, como consecuencia del desarrollo de éste.

Lo primero de todo es la mejora de nivel de programación en Python ya que, a pesar de ser un lenguaje de programación muy utilizado, durante la carrera sólo se impartió en cuatro asignaturas y no se profundizó en las especificaciones del lenguaje. Gracias a este proyecto se ha realizado un aprendizaje más exhaustivo del lenguaje.

Ligado a esta competencia se encuentra la capacidad de desarrollar juegos y diseños gráficos en Python. Antes de este proyecto no se había realizado la creación de juegos en este lenguaje y no se tenía mucho conocimiento sobre el tema.

Otra competencia que se ha mejorado notablemente es la relacionada con el aprendizaje automático, ya que, a pesar de haber estudiado una asignatura sobre el tema, al realizar el trabajo se ha profundizado mucho más en las redes neuronales y algoritmos genéticos de una manera que no se había tratado en el curso.

Por último, una de las competencias que más se ha notado la diferencia es la capacidad de redacción y de búsqueda de información que requiere realizar un Trabajo de Fin de Grado y que en ninguna otra asignatura de la carrera se ha enseñado.

6.4 Trabajos futuros

En cuanto a líneas de mejoras para el futuro de este proyecto se van a comentar varias ideas:

- **Eliminar las limitaciones.**

  La principal mejora que se le puede hacer al proyecto es eliminar las distintas limitaciones que se han ido añadiendo por las pruebas y el entrenamiento. Por ejemplo, los jugadores en el juego normal pueden tener más de 20 cartas de materia prima en la mano sin mayor problema que venga el ladrón.

- **Añadir complejidad al algoritmo.**

  Debido a las largas sesiones de entrenamiento de la red neuronal se ha tendido a simplificar la red y a agilizar el entrenamiento. En el caso de disponer de más equipos y más tiempo, se podría aumentar la complejidad del algoritmo y así obtener mejores resultados.

- **Mejorar el diseño gráfico.**

  El tablero original de Catán está diseñado con materiales con texturas que aumentan la profundidad del tablero y, a pesar de que las herramientas que proporciona Pygame son limitadas, los colores y los diseños empleados son mejorables de cara a hacer un entorno más llamativo al jugador.

- **Añadir interacción del jugador humano.**

  La librería Pygame ofrece una serie de funciones que permiten la interacción con el usuario humano. De esa manera, se podría implementar un sistema de juego entre el agente inteligente, los agentes aleatorios y el usuario humano.

- **Añadir multijugador.**

  Una idea que en un principio se pensó es realizar el juego a través de un bróker que fuese independiente de los jugadores, de esta manera centrar todo el juego en un equipo y que cualquier jugador se pudiese conectar a través de un servidor TCP y poder jugar en modo multijugador contra el agente desde cualquier parte.

- **Ampliar el juego para añadir variantes.**

  El juego de mesa Catán es conocido por sus expansiones que aportan nuevos elementos al juego y crean partidas completamente nuevas. El código está creado de una manera que sería fácilmente ampliable a las expansiones que existen del juego. Creando así un nuevo agente que también es capaz de jugar a las expansiones.

7 Bibliografía

[1] colaboradores de Wikipedia. (2023a). Los colonos de Catán. *Wikipedia, la enciclopedia libre*. https://es.wikipedia.org/wiki/Los_colonos_de_Cat%C3%A1n

[2] *Sumérgete en el mundo de CATÁN UNIVERSE: descarga gratuita*. (2023, 26 enero). Catán. https://Catánuniverse.com/es/

[3] Albertini. (2020). Vikingos, Islandia y un dentista: cómo Catán se convirtió en todo un clásico de los juegos de mesa. *Xataka*. https://www.xataka.com/videojuegos/como-Catán-se-convirtio-clasico-juegos-mesa

[4] colaboradores de Wikipedia. (2023c). Python. *Wikipedia, la enciclopedia libre*. https://es.wikipedia.org/wiki/Python

[5] *About - Pygame Wiki*. (s. f.). https://www.pygame.org/wiki/about

[6] *¿Qué es una red neuronal? - Explicación de las redes neuronales artificiales - AWS*. (s. f.). Amazon Web Services, Inc. https://aws.amazon.com/es/what-is/neural-network/

[7] Russell, S., & Norvig, P. (2016). *Artificial intelligence: A Modern Approach, Global Edition*.

[8] colaboradores de Wikipedia. (2023b). Perceptrón. *Wikipedia, la enciclopedia libre*. https://es.wikipedia.org/wiki/Perceptr%C3%B3n

[9] Abu-Mostafa, Y. S., Magdon-Ismail, M., & Lin, H. (2012). *Learning from data: A Short Course*.

[10] Mitchell, T. M. (1997). *Machine learning*.

[11] *Genetic Algorithms - parent selection*. (s. f.). https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm

[12] Wikipedia contributors. (2023). Crossover (genetic algorithm). *Wikipedia*. https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)

[13] *PyGAD - Python Genetic Algorithm! — PyGAD 3.2.0 Documentation*. (s. f.). https://pygad.readthedocs.io/en/latest/index.html

[14] *Random —Generar números pseudoaleatorios*. (s. f.). Python documentation. https://docs.python.org/es/3/library/random.html

[15] *Game rules*. (s. f.). CATÁN. https://www.Catán.com/understand-Catán/game-rules

[16] Tammy. (2021, 25 mayo). *Tabla de Costes Catán*. Pinterest. Recuperado 9 de agosto de 2023, de https://www.pinterest.es/pin/626844841888012166/

[17] *Statistics of dice throw*. (s. f.). http://hyperphysics.phy-astr.gsu.edu/hbasees/Math/dice.html

[18] *COMBINACIONES SIN REPETICIÓN*. (s. f.). https://matematicatranquila.blogspot.com/2015/07/combinaciones-sin-repeticion.html

[19] *File: Breadth-First-Search-Algorithm.gif - Wikimedia Commons*. (2009, 26 marzo). https://commons.wikimedia.org/wiki/File:Breadth-First-Search-Algorithm.gif

[20] *Calculator Suite - GeoGebra*. (s. f.). https://www.geogebra.org/calculator

A. Anexos

A.1. Instalación y uso

Para ser posible la ejecución del programa se requerirá de un ordenador que sea capaz de ejecutar Python, en concreto su versión 3. Lo siguiente será descargarse el código fuente desde el siguiente repositorio:

[A.1] https://github.com/RubiMonti/Neural-Network-Catan

A continuación, únicamente será necesario acceder a la carpeta *Game* y ejecutar el programa catan.py y por la pantalla se imprimirá el juego en movimiento entre el agente entrenado y los tres jugadores aleatorios.

También es posible entrenar al agente manualmente y posteriormente jugar una partida con la salida obtenida del entrenamiento. Para ello será necesario ir a la carpeta *Training* y ejecutar el programa *Genetic_Algorithm.py*. El entrenamiento del agente dependerá de la capacidad de cómputo del ordenador, pero de media será de unas 15 horas. Tras el entrenamiento, se genera un fichero llamado *Best_solution.txt* que contendrá la mejor solución y que se puede introducir como parámetro al programa catan.py de la carpeta Game para ver los resultados del entrenamiento.

A.2. Publicación

El código fuente del simulador gráfico se encuentra en el repositorio:

[A.2] https://github.com/RubiMonti/Neural-Network-Catan/tree/main/Game

El código relacionado con el entrenamiento del agente se encuentra en el siguiente repositorio:

[A.3] https://github.com/RubiMonti/Neural-Network-Catan/tree/main/Training

Por último, la presente memoria y las imágenes y videos utilizados en la misma se encuentran disponibles en el repositorio:

[A.4] https://github.com/RubiMonti/Neural-Network-Catan/tree/main/Anexos