UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería en Informática

TFG del Grado en Ingeniería Informática

«Creación de un videojuego y un bot inteligente para el mismo» Documentación Técnica

Presentado por Pablo Alejos Salamanca
en Universidad de Burgos — 2 de julio de 2017

Tutores: Dr. José Francisco Díez Pastor
Dr. César Ignacio García Osorio

Índice general

Índice de figuras

Índice de tablas

Apéndice A

Planificación

A.1. Introducción

A.2. Planificación temporal

Antes de empezar a practicar la metodología basada en sprints, hubo aproximadamente un mes de planificación de las mecánicas del juego. A pesar de que fuesen evolucionando a lo largo del proyecto, era indispensable tener claro dónde quería llegar.

Se utlizó una metodoligía basada en srpints similar a *Scrum*, aunque no se ha seguido estrictamente ya que la reuniones eran semanales, no diarias. Para hacer ayudar al seguimiento se utilizó *ZenHub*, un addon para Github que nos permite:

- Establecer Milestones, a modo de Sprints
- Estimar el tiempo de las tareas en *storypoints* que, en mi caso, equivalían a horas.
- monitorizar el prograso en forma de Burndowns

Debo apuntar que no se utilizó *ZenHub* desde el principio, por lo que los primeros sprint no se monitorizaron.

A continuación se muestra la evolución del proyecto a través de los Sprints.

Sprint 0: [02/01 al 02/02]

Tareas:

- Crear la mecánica del juego mínima

En esta iteración se establece como objetivo tener ya un juego funcionando. Sólo es necesario que te deje jugar y cuente la puntuación.

Sprint 1: [02/02 al 16/02]

- Estudiar la posibilidad de poner el juego en una web.
- Barajar diferentes formas de capturar los estados del juego.
- Valorar formas de comunicar el juego con las bibliotecas de Python.

En primer lugar, se han estado analizando qué posibilidades habría de alojar el juego en una web, ya que vamos a necesitar que muchos usuarios jueguen. *Unity* te permite compilar directamente en *WebGL* lo que me facilita mucho la tarea. Los problemas llegan cuando intento salvar datos del lado del servidor, no hubo manera de conseguir que el jugo guardase un solo archivo. Por este motivo se descartó la posibilidad de *hostearlo* en un servidor web.

En este sprint también comienzo el proceso de definición de lo que conformarán las instancias. En las primeras pruebas tuve problemas al capturar las instancias, pues no había contemplado que el tamaño de estas fuese constante, requisito indispensable para el tipo de algoritmos de aprendizaje que estaba utilizando.

Para este sprint era interesante ir teniendo un primer contacto con las bibliotecas de *sckit-learn*.

Como el juego ya estabe en funcionamiento comienzo a desarrollar is propios sprites para el juego.

Sprint 2: [16/02 al 02/03]

En esta reunión se solucionó el problema de que TextMaker estaba mal instalado.

- Se ha observado que la estimación de la isues no se estaba realizando correctamente. Se han comentado los siguientes temas: Para este proyecto va a ser necesario comunicar python y C# por lo que se deberá investigar la forma de hacerlo.

En este punto es cuando realmente empiezo una dinámica similar a *Scrumm*, en la que se harán reuniones, a ser posible, semanales, ya que antes estaba trabajando sin tener un objetivo semanal concreto.

El grueso del trabajo y el tiempo de este sprint lo absorbe la «odisea» que fue, para mi, el proceso de comunicar los scripts de *Python* y Unity. Todo este proceso viene descrito en la memoria.

En el anterior sprint se observa que el jugador puede quedarse disparando continuamente, lo que quitaría emoción al juego. Por este motivo se propone que se implemente un sistema de «calentamiento» del arma, de este modo, el jugador se verá forzado a dejar de disparar. Junto con el calentamiento del arma viene, forzosamente, diseñar algún elemento gráfico que indique al jugador el estado del arma.

Llegados a este punto, se comienza a diseñar el menú de inicio.

Sprint 3: [02/03 al 14/03]

En este sprint se validaron los avances hechos. La nueva interfaz estaba lista, la barra de calentamiento del arma funcionaba y el arma funcionaba como se espraba.

Respecto a la comunicación: La opción mas viable hasta la fecha para la comunicación entre aplicaciones es la utilización de Pipes. De momento no tengo muy claro como funcionan. Los pipes en python utilizan un método (os.fork) que es específico de Linux. Me planteo descartar el uso de pipes y probar con sockets.

Sprint 4: [14/03 al 19/03]

Se ha conseguido hacer funcionar sockets en *Python*. Ahora hay que establecer la comunicación entre python y .net. Una vez conseguido esto se harán las primeras pruebas con el bot.

Para este sprint se pretende:

- Una vez se ha hecho funcionar los sockets entre dos scripts de *Python* se tiene que hacer entre una Unity y *Python*.
- Si se consigue establecer la comunicación empezar a procesar estados.
- Entrenar un agente inteligente sencillo y probar si funciona la comunicación.

Sprint 5: [21/03 al 28/03]

Durante el sprint 4 encontré algunas dificultades. Había un error, de procedencia desconocida, que hacía que el juego parase la ejecución al pulsar la barra espaciadora. Este problema se solucionó refactorizando el código de controlador del jugador. Se consiguió comunicar Unity con *Python* y ya tenía funcionando un bot simple.

Tarea para este Sprint:

- Respecto al estado:
  - Añadir TimeStamp, vida restante de los enemigos y temperatura del arma.
  - Ahora los estado pasarán a tener dos clases (Dirección del movimiento y si dispara o no).
- Permitir moverse adelante y atrás.
- Dejar el juego funcional para poder pasar a la toma de datos.
- Añadir Banda sonora.

Sprint 6: [28/03 al 12/04]

En esta reunión se mostró la nueva interfaz y el juego listo para el entrenamiento. Para este sprint se pretende tener una IA básica funcionando con el nuevo set de estados y si todo va bien distribuir el juego entre otras personas para la captura de datos.

Respecto a las mejoras del juego:

- El juego ya dispone de banda sonora. He intentado dar crédito a su autor, pero no he sido capaz de encontrarlo de nuevo (fallo mío no guardar el enlace).
- Se han incluido efectos de sonido para disparo, explosión y disparo.
- Ahora se almacenan las máximas puntuaciones a título meramente informativo.
- Los enemigos muestran, ahora, su barra de vida restante.
- Se incorporan dos tipos de «power-up»:
  - Enfriar el arma: reduce una determinada la temperatura actual del arma.
  - Boost: Duplica el arma, disparando el doble de balas en el mismo tiempo.
- Permite moverse al jugador adelante y atrás.
- Nueva forma de perder, si el jugador impacta contra un enemigo, muere.
- Se incorpora un fundido a negro cuando se pasa del menú al juego.

Sprint 7: [12/04 al 18/04]

Se ha observado que el método que captura los estado busca los enemigos aleatoriamente, de forma que si hace una lista con los enemigos de 1 al 6. El que es el enemigo 1 en un primer instante puede ser almacenado como enemigo 3 en otro. Se requiere un pre-procesado de los enemigos de tal forma que solo se tendrá en cuenta el número de enemigos hay en el eje X y la distancia que han recorrido. Este es el punto en el que se incorpora el «heat-map» que de detalla en la memoria.

En este sprint se pretende tener el Bot entrenado con el nuevo conjunto de datos. Como hemos introducido una forma de diferente de procesar los enemigos, hay que adaptar los scripts a las nuevas instancias.

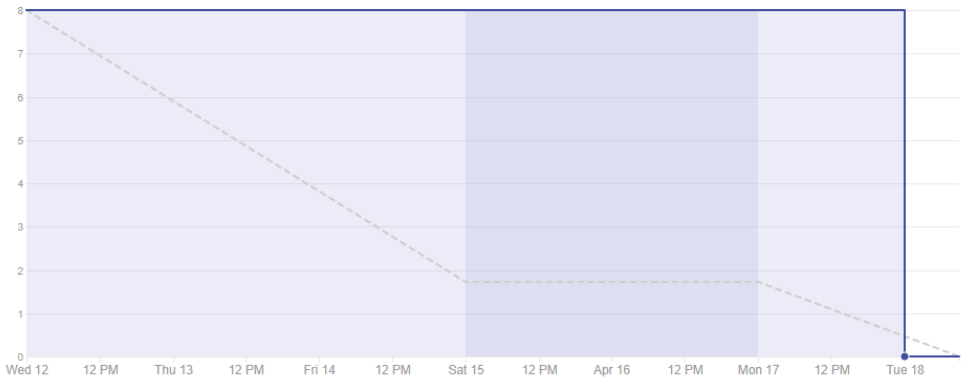

Figura A.1: Sprint 7

Sprint 8: [18/04 al 25/04]

Se tiene que hacer un preprocesado del estado del juego a la hora de pasárselo al bot entrenado. (Me entran dudas de si seguiré siendo capaz de procesar 4 estados por segundo). Finalmente si que es posible, aunque en ocasiones pierde fotogramas. Esta pérdida la atribuyo a que mi ordenador es de gama baja.

El juego ya tiene la capacidad de jugar solo aunque sin demasiado acierto. Una vez completado el preprocesado se harán grupos de n estados con el fin de tener en cuenta los n estados anteriores a la toma de decisión, tal y como lo haría un humano. Esto se ha de hacer en python ya que a partir de ahora los estados generados por el juego es conveniente que sean invariables.

Sprint 9: [25/04 al 05/05]

Se han barajado mejoras de aprendizaje supervisado:

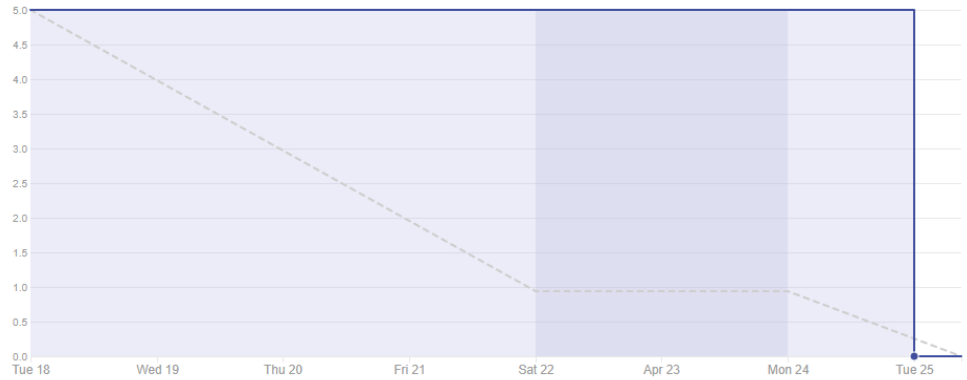

Figura A.2: Sprint 8

- Parametrizar numero de estados anteriores
- Balancear clases: Esto significa que si, por ejemplo, estamos disparando el 90 % del tiempo, deberíamos eliminar las instancias necesarias para que las instancias queden proporcionalmente balanceadas 50 % disparando 50 % sin disparar.
- Filtrar datos de buenos jugadores.

A la vista de los resultados se opta por añadir un array que indique la proximidad a los enemigos. esto en teoría debería reforzar la tendencia a conductas suicidas

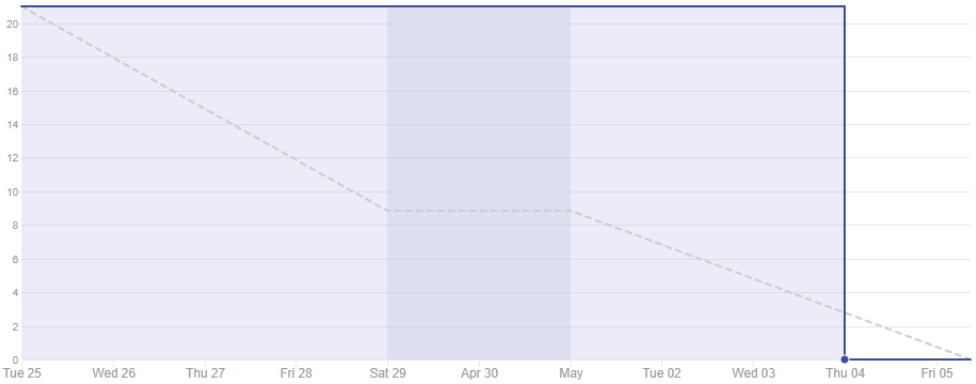

Figura A.3: Sprint 9

Sprint 10: [05/05 al 16/05]

Ésta semana no ha habido grandes avances, se ha incorporado la detección de enemigos cercanos y se han analizado las posibles alternativas para seguir mejorando el aprendizaje.

Para este sprint se pretende refactorizar el código de los «power-up» : Fijar el código en una clase propia y no en el Player Controller.

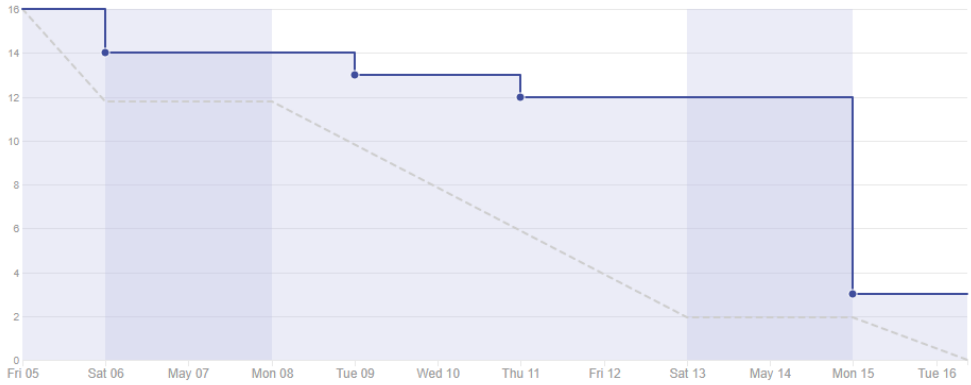

Figura A.4: Sprint 10

Sprint 11: [16/05 al 23/05]

Se han aprobado los cambios. pasamos a la fase de entrenamiento con algoritmos evolutivos. Propuestas:

- Crear una pequeña aplicación unity que permita ser ejecutada desde línea de comando para ver si podemos lanzar el juego para el entrenamiento con algoritmos evolutivos.
- Rehacer las instancias para ver si podemos mejorar un poco más el entrenamiento por imitación.
- Hacer un script que nos haga de entrenador, generando el modelo del agente inteligente.
- Implementar un perceptrón sencillo para ver de qué soy capaz.

Sprint 12: [23/05 al 30/05]

Se ha logrado utilizar un preceptrón multicapa para jugar al juego. Queda concatenar estado ya que juega bastante mal.

En este sprint se debe:

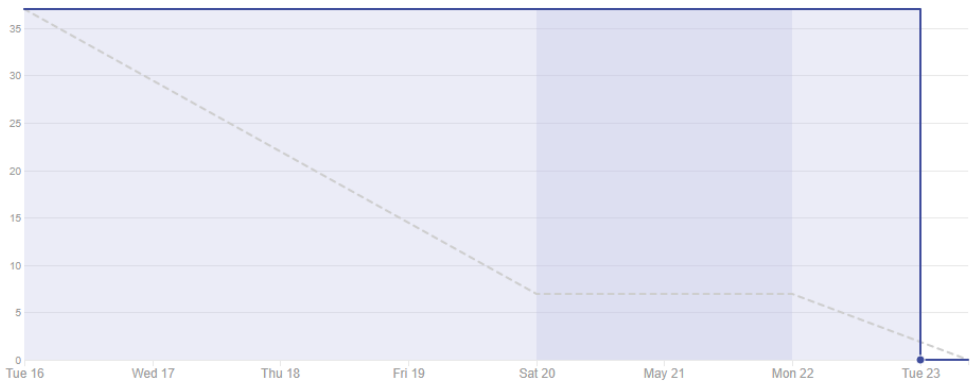

Figura A.5: Sprint 11

- Jugar con los parámetros del percetrón para ver si mejora.
- Añadir información a las instancias: Actualmente solo ve a lo enemigos y su distancia relativa, hay que añadir la distancia relativa a paredes y power ups en la instancia.

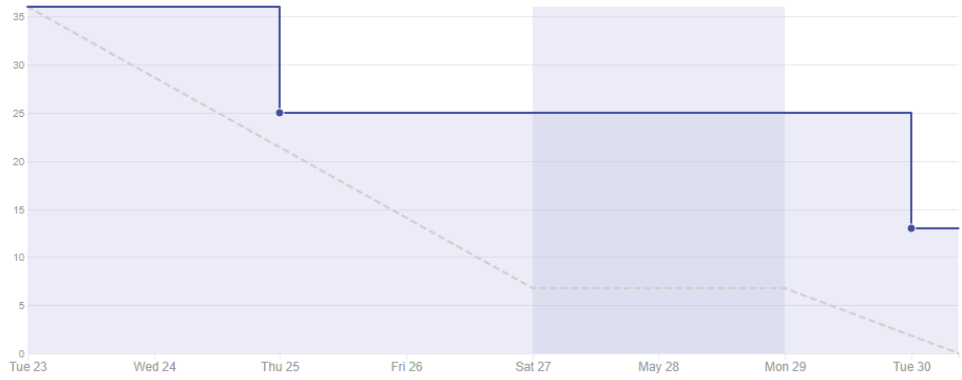

Figura A.6: Sprint 12

Sprint 13: [30/05 al 10/06]

En este sprint se trabajará principalmente en la documentación. Hacer la explicación completa de los conceptos teóricos.

Sprint 14: 10/06, 13/06

Como en el sprint anterior no se pudo hacer el apartado Conceptos teóricos, queda pendiente para este. Si es posible se acabará el apartado Aspectos

relevantes y se comenzará a trabajar con los anexos. Se han seguido haciendo pruebas con los algoritmos evolutivos.

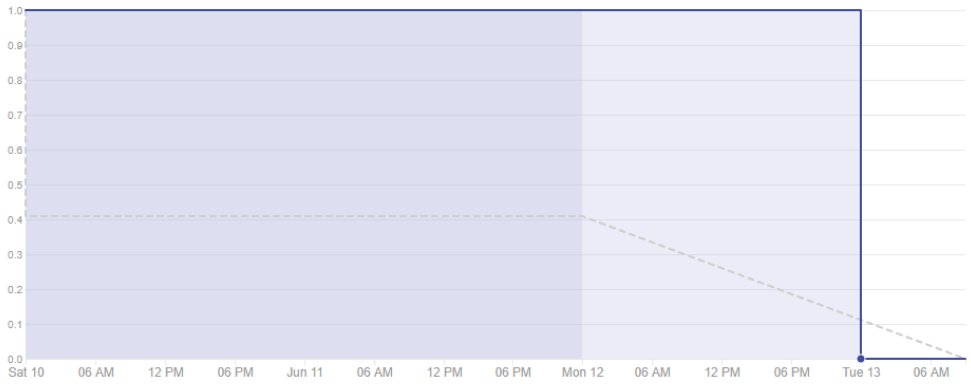

Figura A.7: Sprint 14

Sprint 15: [13/06 al 19/06]

En este sprint se han realizado las siguientes tareas:

- Se ha corregido una series de errores que calificaría como graves, ya que impedían por completo la evolución de la red neuronal.
- Se han corregido errores tipográficos y conceptuales en la documentación. Había algunos aspectos que se entendieron mal inicialmente, se ha procedido a redactarlos de nuevo.
- Se ha seguido avanzando el los aspectos relevantes y los conceptos teóricos.
- Se ha generado una versión del juego para capturar los datos de los usuarios.

Sprint 16: [19/06 al 28/06]

Se pretende completar la documentación y corregir ciertos aspectos de los scripts.

A.3. Estudio de viabilidad

Viabilidad económica

Este proyecto nunca estuvo pensado para obtener un beneficio económico. Este proyecto tenía un objetivo claro, ampliar mis conocimientos en el campo

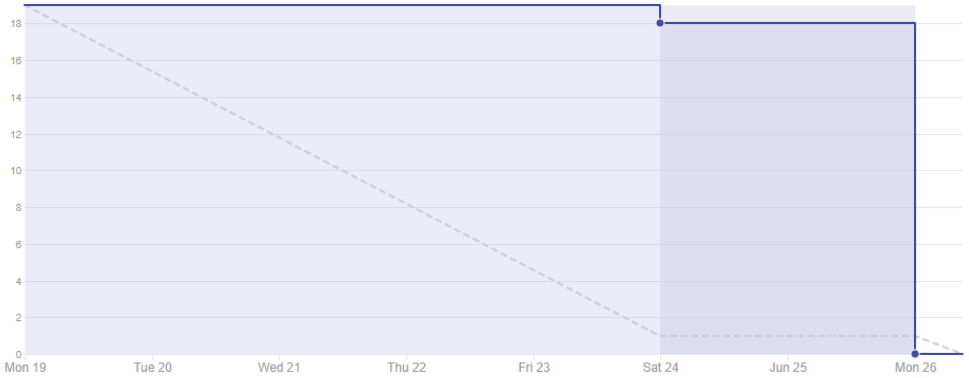

Figura A.8: Sprint 16

de la inteligencia artificial y, de algún modo, dotar a futuros estudiantes de un entorno de trabajo mas vistoso sobre el que probar sus agentes inteligentes, y así, animar a los estudiante a descubrir el maravilloso mundo de la inteligencia artificial.

Analizando los costes, el único que nos podría limitar sería Unity3D, pero el acuerdo de licencia estipula que el uso de su software gratuito siempre que se utilice con fines educativos.

Viabilidad legal

Este proyecto ha sido realizado utilizando recursos de software, cada uno con su licencia. A continuación procedo a analizar cada uno de ellos:

- Scikit-Learn: sujeto a la licencia de tipo BSD-3, que permite la redistribución con o sin modificaciones siempre que los autores originales sean mencionados y se mantenga el mismo tipo de licencia.
- Deap: se encuentra bajo licencia GNU GPL, que permite usar, copiar, modificar el software y redistribuirlo.
- Pandas: Pandas, al igual que scikit-learn, está sujeto a licencia BSD-3 y, por lo tanto, conlleva el mismo tipo de permisos y restricciones.
- Pickle: Esta biblioteca también se encuentra sujeta a los términos de BSD-3, por lo que ya hemos mencionado sus características.
- Unity3D: Unity3D ofrece varios tipos de licencia en función de la finalidad y la rentabilidad que vayamos a obtener del producto. [1]

[1] consultar detalles: https://store.unity.com/es

Debido a las restricciones de las licencias antes mencionadas mi proyecto, de ser distribuido, debería ser publicado bajo una licencia BSD [1].

Apéndice B

Especificación de Requisitos

B.1. Introducción

En este apartado se especificarán las necesidades funcionales que se pretende alcanzar. Para ello se describirán los requisitos del software que se quiere desarrollar.

B.2. Objetivos generales

- Como ya de describió en la memoria, el objetivo del proyecto es diseñar y desarrollar un videojuego de tipo «arcade». Para la implementación del juego que sirve como base para este proyecto, es necesario planear las mecánicas de juego, además de diseñar los modelos del jugador, los enemigos y otros elementos gráficos. Esta tarea también incluye el diseño de la interfaz, banda sonora y efectos de sonido.
- Diseñar e implementar un sistema inteligente que posteriormente será utilizado para interactuar con el videojuego creado. Este proceso de divide en:
  - Establecer una comunicación entre el juego y el modelo del sistema inteligente. Dado que el juego y el script que hace uso del modelo serán implementados en diferente lenguaje de programación, se debe establecer un mecanismo de intercambio de datos para la comunicación.
  - Diseñar el conjunto de datos que posteriormente servirá para el entrenamiento del modelo. Al ser el punto de partida el aprendizaje por imitación, necesitaremos un gran volumen de datos de los cuales seleccionaremos todos o solo los más representativos para el entrenamiento.

- Implementar, haciendo uso de las bibliotecas como scikit-learn (minería de datos), DEAP(algoritmos evolutivos) y Pandas y NumPy (procesamiento de datos a bajo nivel) el sistema inteligente que jugará al videojuego.

B.3. Catalogo de requisitos

Especificación de requisitos

Requisitos funcionales

Los requisitos funcionales que detallaré a continuación son so requisitos iniciales, ya que a lo largo del desarrollo, alguno de ellos se ve afectado por nuevas necesidades.

Los requisitos principales son:

- RF1: Jugar al juego.
- RF2: Entrenar el modelo.
  - RF2.1: Entrenar utilizando un Decision Tree
  - RF2.2: Entrenar utilizando un Random Forest
  - RF2.3: Entrenar utilizando un algoritmo evolutivo

  RF3: Explotar el modelo
  - RF3.1: Explotar utilizando el modelo Decision Tree
  - RF3.2: Explotar utilizando el modelo Random Forest
  - RF3.3: Explotar utilizando el modelo algoritmo evolutivo

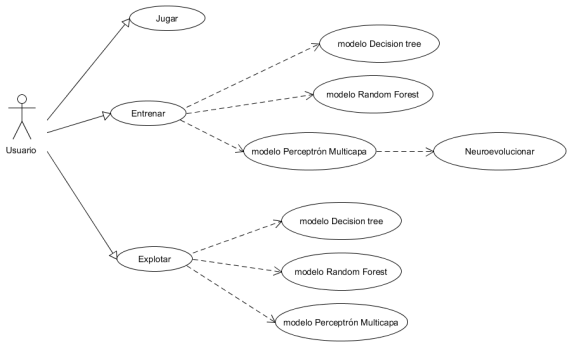

Figura B.1: Casos de uso

A continuación se procede a analiza en profundidad el RF1:

- RF-1.1: Permitir que el jugador interactúe con la nave espacial, haciendo que se traslade lateralmente B.1.
- RF-1.2: Permitir que el jugador pueda disparar B.2.
- RF-1.3: Permitir que el jugador interactúe con el menú principal. El menú principal tiene dos botones B.3.
  - RF-1.3.1: El botón jugar lleva directamente a una nueva partida.
  - RF-1.3.2: El botón salir permite salir del juego.
- RF-1.4: El juego debe tener la capacidad de escribir los estados en un fichero B.4.

R1.1: Permitir que el jugador interactúe con la nave espacial	
Versión	1.0
Autor	Pablo Alejos
Descripción	El jugador debe ser capaz de controlar la nave espacial haciendo uso de las teclas izquierda y derecha.
Precondiciones	El jugador debe estar vivo, y la partida activa
Acciones	El usuario pulsa las teclas izquierda y derecha para controlar la nave.
Postcondiciones	La nueva posición del jugador no podrá ser la misma que antes de pulsar los controles.
Excepciones	Una pared puede bloquear el movimiento, por lo que la posición seguirá siendo la misma
Importancia	Muy alta

Tabla B.1: RF-1: Permitir que el jugador interactúe con la nave espacial

R1.2 Permitir que el jugador pueda disparar.	
Versión	1.0
Autor	Pablo Alejos
Descripción	El jugador debe ser capaz de disparar pulsando la tecla espaciadora.
Precondiciones	El arma tiene una cadencia, tiene que pasar un tiempo determinado para poder volver a disparar.
Acciones	El usuario pulsa la tecla de disparo (Barra espaciadora).
Postcondiciones	Se debe producir un único dispatro.
Excepciones	Si no ha pasado el tiempo necesario desde el ultimo disparo no debe disparar
Importancia	Alta

Tabla B.2: RF-2: Permitir que el jugador pueda disparar.

RF-3: Permitir que el jugador interactúe con el menú principal.	
Versión	1.0
Autor	Pablo Alejos Salamanca
Descripción	El jugador interactúa con el menú principal
Precondiciones	-
Acciones	RF-3.1: el jugador pulsa empezar partida
	RF-3.2: el jugador pulsa el botón salir
Postcondiciones	RF-3.1: cambia de escena, pasando a la pantalla de juego.
	RF-3.2: sale de juego.
Excepciones	No tiene.
Importancia	Alta

Tabla B.3: RF-3: Permitir que el jugador interactúe con el menú principal

RF-4: El juego debe tener la capacidad de escribir los estados en un fichero.	
Versión	1.0
Autor	Pablo Alejos Salamanca
Descripción	A lo largo de la partida el juego debe registrar tanto el estado de la partida como las acciones del jugador. Estas instancias de graban en un fichero csv.
Precondiciones	Tiene que haber una partida activa que poder grabar.
Acciones	El jugador hace uso del juego para generar el fichero.
Postcondiciones	Cuando acaba la partida debe existir un fichero denominado gamestates_XX.csv
Excepciones	-
Importancia	Alta

Tabla B.4: RF-4: El juego debe tener la capacidad de escribir los estados en un fichero.

Apéndice C

Especificación de diseño

C.1. Introducción

C.2. Diseño de datos

Dado que consideraba un factor muy importante la forma de tomas los datos, se explica este apartado detalladamente el aspectos relevantes.

C.3. Diseño procedimental

A continuación, se procede a explicar algunas nociones relevantes de cómo interactúan entre sí los diferentes elementos que conforman el proyecto. Dado que no es una aplicación al uso, sino que se compone de múltiples elementos, mostraré un diagrama de despliegue y un diagrama de secuencias.

Diagrama de despliegue

La estructura que ha acabado adoptando el proyecto puede resultar algo compleja, por lo que explicaré, en primer lugar, cómo está organizado C.1.

Para empezar es muy importante aclarar que hay dos implementaciones del juego. Por un lado está el juego que recibirán los usuarios. El juego que reciben los usuarios es lo que consideraríamos un juego real, es decir, el jugador dispone del menú de inicio y banda sonora. Por el otro lado, tenemos el juego que va a utilizar el agente inteligente. Este juego no dispone de menú principal, la partida comienza directamente e intenta conectarse al *socket*. La banda sonora se decidió retirar por comodidad a la hora de trabajar, ya que acababa resultando cargante.

Para la creación de los modelos los agentes inteligentes se han seguido dos vertientes principales. Una es la implementación de un decision tree, que es la

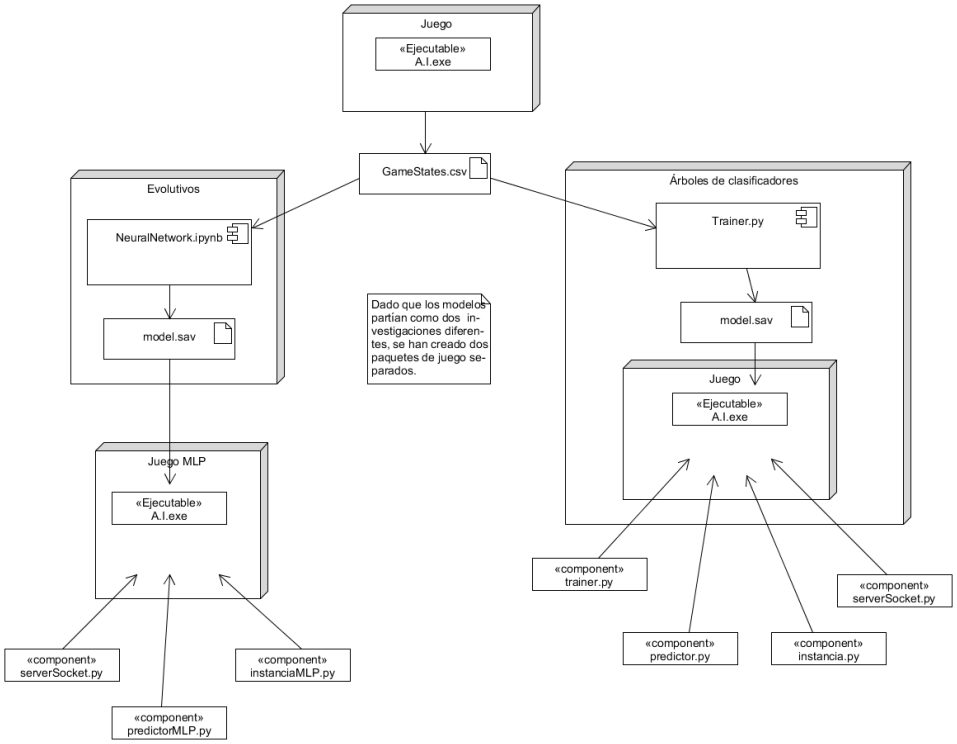

Figura C.1: Diagrama de despliegue

que primero se probó. La otra vertiente el la relativa a los algoritmos evolutivos. Por este motivo ha dos bloques que, a priori, iban a ser completamente diferentes, pero acabaron siendo muy similares. A pesar de tener cierta similitud, esas pequeñas diferencias hicieron que conservase la estructura inicial.

Diagrama de secuencias

En esta sección se pretende aclarar el funcionamiento de, al menos, uno de los agentes, ya que el otro es muy similar.

El entrenamiento y explotación del modelo de agente inteligente se realizan tres acciones. La captura de datos, para el entrenamiento por imitación, en entrenamiento utilizando el clasificador preferido y, para finalizar, la utilización del modelo entrenado sobre el juego C.2.

La captura de datos es muy sencilla: el jugador utiliza el juego normalmente y el juego crea un «log» con todas las instancias que se han ido generando a lo largo del juego. EL usuario deberá enviar al programador los resultados de la captura de datos de la partida para que este pueda trabajar con ellos.

El entrenamiento es muy sencillo y se compone de dos pasos. En primer

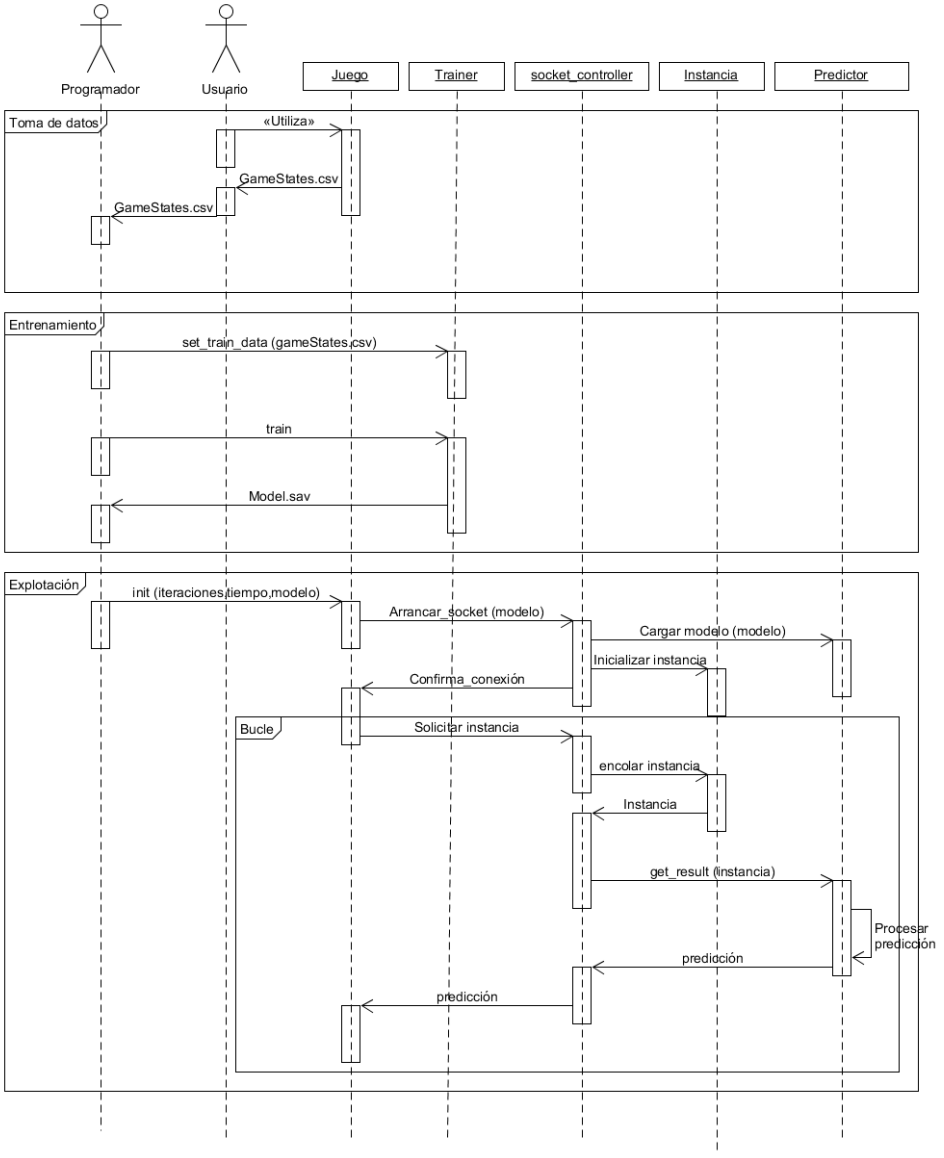

Figura C.2: Diagrama de secuencias (Arboles de decisión)

lugar decimos al entrenador qué fichero de datos queremos utilizar. A continuación, una vez ha procesado todos los datos, se le indica que inicie el entrenamiento. Como resultado deberíamos obtener un modelo entrenado.

La explotación del modelo entrenado ya es algo más sofisticada, aunque sigue siendo un flujo relativamente sencillo.

- 1 - El programador lanza el juego, indicando por parámetro el numero de rondas, el tiempo máximo por ronda y el modelo que tiene que utilizar.
- 2 - En cuanto se lanza el juego, este ejecuta un script de *Python* que abre, a su vez, un socket con el que establecer la comunicación entre el juego y el agente inteligente.
- 3 - El script de *Python*, además de abrir el socket, carga el modelo e inicializa el procesador de instancias.
- 4 - Realizados estos pasos el socket confirma la conexión y se comienzan a mandas las instancias de juego a este.
- 5 - Dado que el procesado de instancia se va a hacer en «paquetes» de cuatro, por el concatenado de instancias que se explica en la memoria, lo primero que hace el socket al recibir la instancia es encolarla.
- 6 - Una vez encolada, el procesador de instancias la devuelve junto con las tres anteriores.
- 7 - La instancia procesada se envía al «predictor» y éste, después de procesar internamente la predicción, la devuelve al servido de socket.
- 8 - EL servidor de socket devuelve la predicción al juego, que realizará la acción correspondiente.

Este proceso de envío, procesado y predicción se hace en bucle hasta que el jugador muera o se cierre el juego. El bucle se realiza cuatro veces por segundo.

Diagrama de clases

A continuación se muestra el diagrama de clases. Considero que es un diagrama de clases «especial», ya que Unity3D no trabaja con las clases tipo java que podemos conocer, Unity3D trabaja con GameObjects que, a su vez están formados por una o varias clases de .net y otros tipos de objetos, incluso un GameObject puede no tener ninguna clase .net. Este hecho me ha dificultado el desarrollo del diagrama de clases. Finalmente he decidido crear yo mi propio tipo de diagrama para poder englobar las clases que todos conocemos y los GameObject de Unity3D.

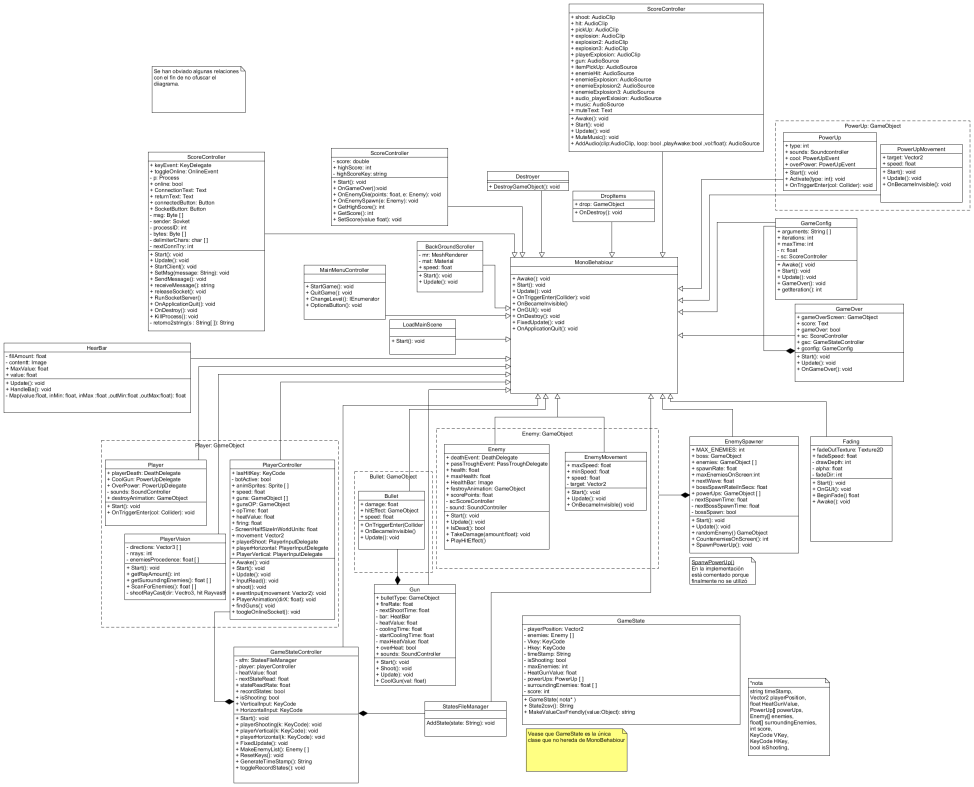

Figura C.3: Diagrama de clases

Apéndice D

Documentación técnica de programación

D.1. Introducción

En este apartado se detallará cómo entrenar nuestros agentes inteligentes y como hacerlos funcionar en el entorno del videojuego. Hay que recalca que ha sido necesario crear dos implementaciones muy similares pero diferentes, ya que uno de los problemas era de tipo *multilabel* y el otro era de tipo *multiclase*.

D.2. Estructura de directorios

A continuación se va a explicar la estructura de directorios para no perder el tiempo buscándolo todo.

Los principales directorios del proyecto son:

- Proyectos Unity
- Redes Neuronales
- DecicionTreeClasiffiers
- Ejemplos y pruebas

En el directorio de «proyectos Unity» encontraremos los dos principales proyectos de videojuego con los que se ha trabajado. En primer lugar podemos encontrar el proyecto «Juego». Este proyecto contiene el juego principal, el que van a utilizar los usuarios para obtener los datos de las partidas. Por otro lado tenemos el proyecto «Juego_lite». Esta segunda versión de la implementación es una versión reducida del primero, carece de menús y sonido, ya

que únicamente está destinado a ser jugado por la máquina. Finalmente tenemos un último directorio en el que encontramos los proyectos de que probaban funcionalidades muy concretas, como las pruebas de conexión de sockets y el lanzamiento del juego con parámetros desde terminal.

En los directorio «DecsionTreeClasiffiers» y «RedesNeuronales», encontramos una build del juego lista para funcionar, cada una referente a su tipo de algorimo. Los scripts de Python deben ir dentro del directorio AI_Data y los modelos entrenados los he colocado junto al ejecutable pero, como se va a referenciar con una ruta relativa, se pueden colocar donde más nos guste. Junto al ejecutable he creado un acceso directo al mismo, esto es porque el juego ha de lanzarse con parámetros y esto nos facilita la tarea.

Como es de esperar, la realización de este proyecto ha requerido de una gran labor de investigación, síntesis y pruebas. En el directorio pruebas y ejemplos podemos encontrar un «batiburrillo» de ejemplos realizados con ayuda del tutor y apuntes de la asignatura de «Minería de datos». Además, se han guardado en ésta carpeta varios sets de datos antiguos que fueron utilizados en las primeras pruebas.

D.3. Manual del programador

Voy a proceder a explicar cómo se debería proceder para la compilación del juego, entrenamiento de los agentes inteligentes y su posterior uso.

D.4. Compilación, instalación y ejecución del proyecto

Como que este proyecto esta dividido en tantas partes empezaré explicando cómo instalar python, qué bibliotecas necesitamos y cómo se instalan.

Instalando Python

Este royecto ha sido realizado utilizando *Anaconda*[1], una distribución de *Python*. Para obtener *Anaconda* vamos a su página web y descargamos la versión correspondiente a nuestro sistema operativo(32/64 bits).

IMPORTANTE: Actualmente hay varias versiones disponibles de Python. A lo largo de este proyecto se ha utilizado la 3.5.2, que sería compatible con cualquier versión a partir de la 3.0, pero incompatible con cualquier versión inferior D.1.

[1] https://www.continuum.io/downloads

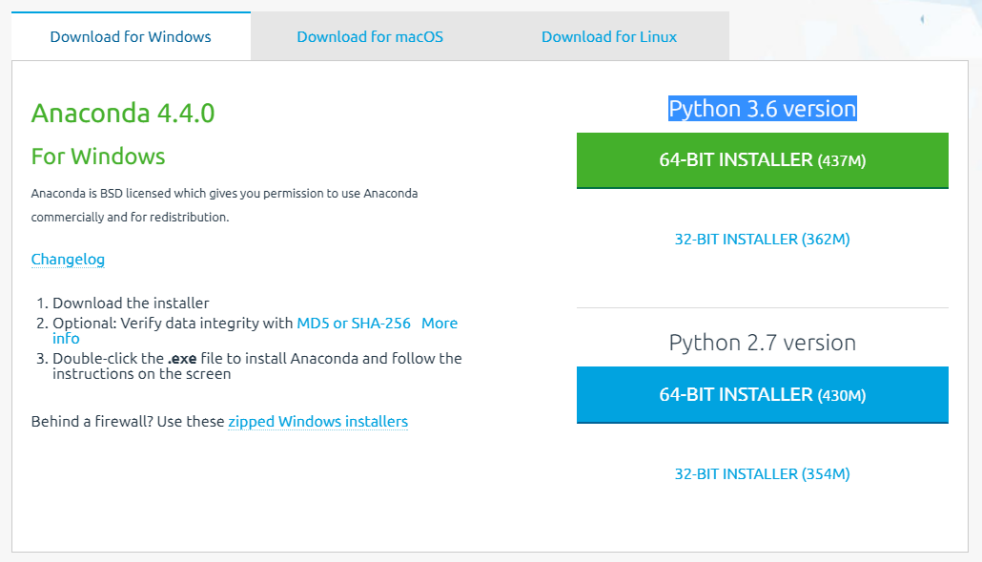

Figura D.1: Descarga Anaconda

Una vez instalado anaconda procedemos a instalar los paquetes necesarios para los scripts. *Anaconda* trae por defecto muchos paquetes de los que necesitamos, ente los que están:

- NumPy.
- pandas.
- SciPy.
- Matplotlib.
- Jupyter.

Además de estos paquetes, necesitaremos *Pickle* ,*scikit-learn* y *Deap*. La instalación de paquetes en *Anaconda* es sumamente sencilla. Abrimos el terminal de windows y escribimos:

```
conda install package-name
```

Para la instalación de Deap, en su página web nos indica que lo hagamos de la siguiente manera:

```
pip install deap
```

Cuando termine de instalar deberíamos tener el entorno de listo para trabajar en él.

Instalando Unity3D

Para cargar y compilar el juego necesitaremos Unity3D. Es software de desarrollo de videojuegos se puede descargar gratuitamente desde su página web. https://unity3d.com/es/get-unity/download

Figura D.2: Descarga Unity3D

Para este proyecto es suficiente con descargar la versión gratuita. Ejecutamos el instalador y elegimos la opción por defecto. En una de las pantallas nos permite seleccionar qué componentes se quiere instalar, con seleccionar los que se ven en D.3 es suficiente.

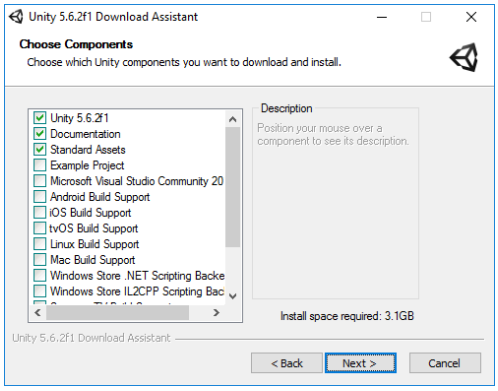

Figura D.3: Descarga Unity3D

Compilación

Para la compilación del juego hay que ejecutar Unity3D y cargar el proyecto que se va a compilar. Para ello se han de seguir los siguientes pasos:

- 1 - Lanzar Unity3D y cargar el proyecto. Al lanzar Unity3D, si no hemos modificado ninguna configuración, nos debe mostrar una ventana como D.4. En esta ventana nos muestra los proyectos recientes que, si es la

primera vez que lo lanzamos, estará vacía. Para cargar un proyecto no listado pulsamos el botón «Open» y buscamos el proyecto en nuestro disco. En este caso hay que cargar el que he denominado «Juego».

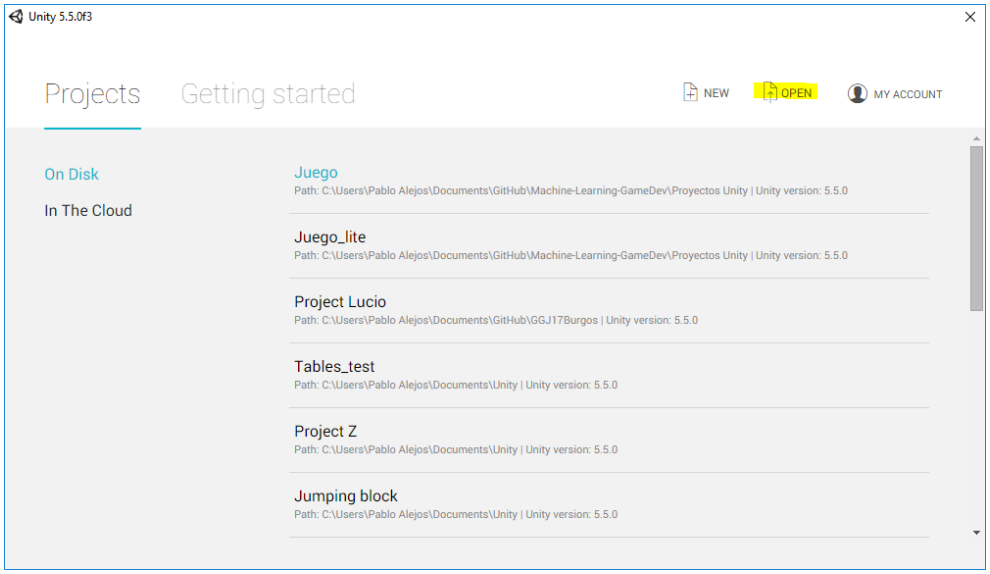

Figura D.4: Carga proyecto Unity3D

- 2- Compilación del juego. Para compilar el juego vamos a «File» ¿«Build settings» y se nos abre la ventana de compilación D.6. En la ventana de compilación deberían aparecer cargadas dos escenas, identificadas con el id 0 y 1 respectivamente. Hecha esta comprobación procedemos con la *build*. Para compilar el juego pulsamos sobre build y seleccionamos la carpeta donde lo queremos guardar.

Con el juego ya compilado no tendríamos mas que entregar al jugador el ejecutable con las carpeta de binarios generada y este podría empezar a jugar y a generar datos. Mientras nuestro «sujeto de prueba» está entretenido jugando, vamos a continuar haciendo la build del juego que utilizará nuestro agente inteligente.

Incorporación de los scripts

En este punto ya tenemos *Python* instalado, Unity3D instalado y hemos creado una build del juego.

Para la compilación del juego reducido, el juego que utilizarán los agentes inteligentes se siguen los mismos pasos mencionados anteriormente, pero esta

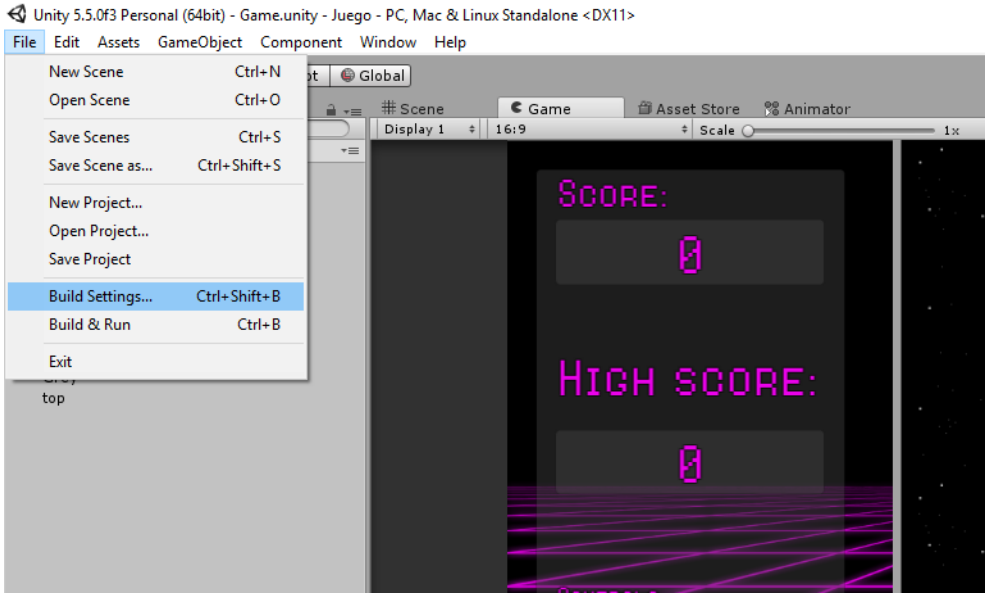

Figura D.5: Compilar proyecto

vez cargaremos el proyecto denominado «Juego_lite». Esta es una versión reducida del juego, no dispone de menú principal y de ha eliminado la banda sonora.

Una vez compilado debemos añadir a la build los scripts de python. Los scripts de python se encuentran el la carpeta «Scripts_python» que cuelga de la raíz principal y elegimos los correspondientes al tipo de agente inteligente que queremos utilizar. Disponemos de los scripts relativos a los arboles de decisión, o bien, de los relativos a las redes neuronales. Una vez seleccionados los scripts de Python que necesitamos, se copian en la carpeta «AI_Data» que acompaña al ejecutable.

Ya tenemos listo el entorno en el que vamos a probar los modelos entrenados.

Guía de uso para árboles de decisión

Entrenamiento

Para entrenar un modelo necesitamos dos cosas, el script llamado «trainer.py» y el dataset en formato *csv*, ambos han de estar en la misma carpeta. El entrenamiento es tan sencillo como abrir la linea de comandos de *windows*, ir a la carpeta en la que tengamos el script y el dataset y escribir:

```
python trainer.py
```

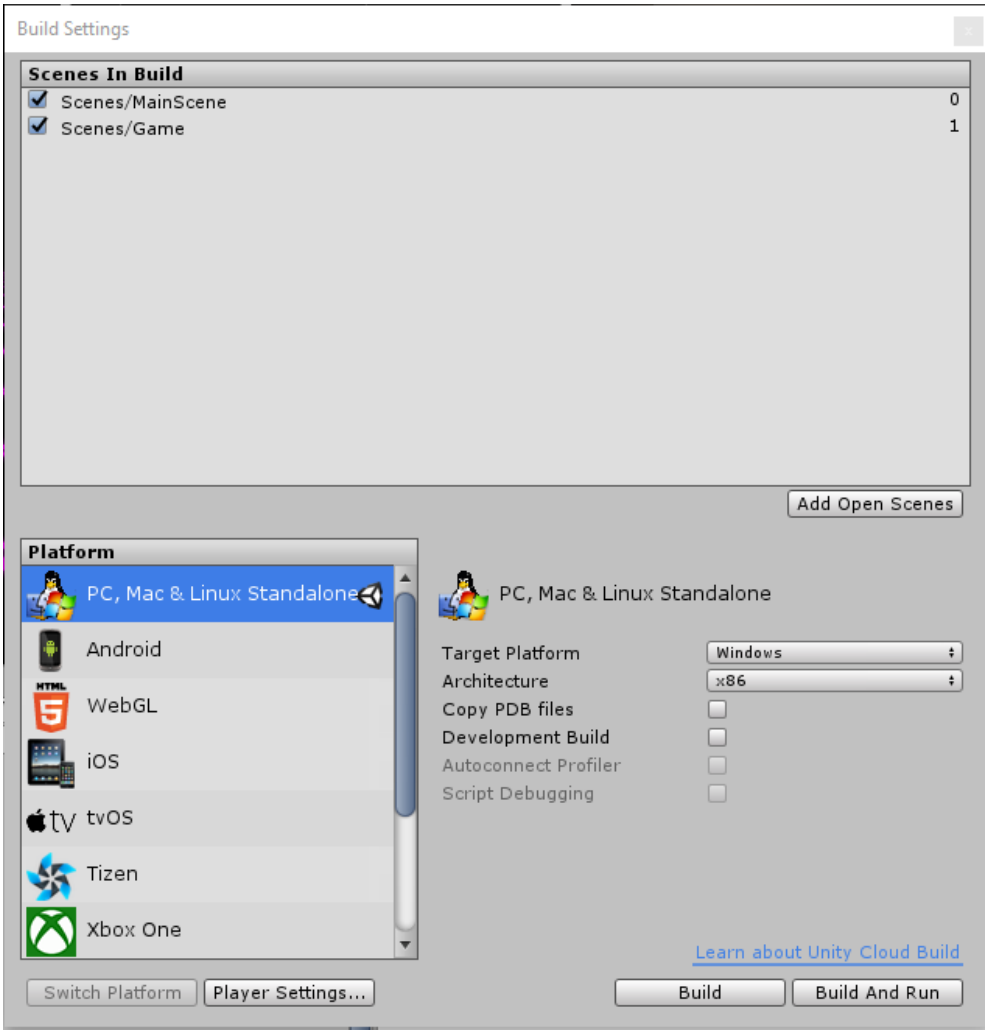

Figura D.6: Ventana de compilación

Este comando ejecutará automáticamente un entrenamiento con el modelos por defecto (Decision tree).

Este script nos permite seleccionar el tipo de clasificador que queremos utilizar y con qué nombre lo vamos a guardar. en primer lugar se debe especificar el nombre del clasificador *RandomForestClassifier* o bien, *DecisionTreeClassifier*. En segundo lugar podemos especificar el nombre del fichero en el que lo vamos a guardar. Un ejemplo de ejecución sería:

```
python trainer.py RandomForestClassifier rfc.sav
```

Este comando nos entrenaría un modelo empleando un random forest classifier y lo salvaría en el fichero rfc.sav.

Ejecución

Para lanzar el ejecutable haciendo uso un modelo entrenado tenemos varias opciones:

Una opción es abrir la consola de windows, ir hasta la carpeta que contiene el ejecutable y escribir:

```
AI.exe nro_de_rondas tiempo_de_ronda ruta_relativa_del_modelo
```

Todos los parámetros son obligatorios.

Otra forma de ejecutar el juego utilizando parámetros es crear un acceso:

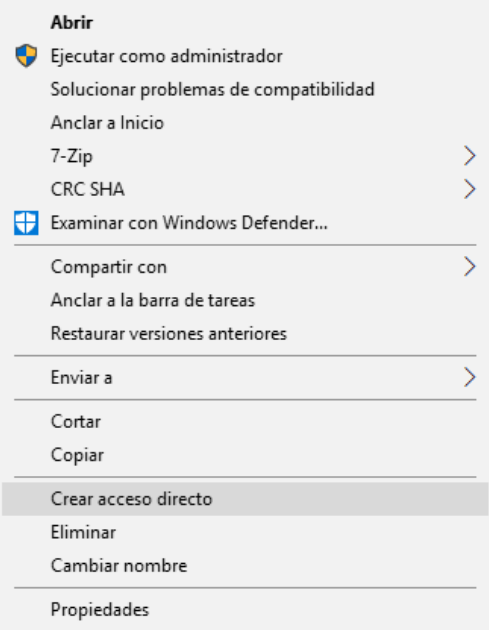

Figura D.7: Creación de un acceso directo

Una vez creado el acceso directo le hacemos clic derecho, propiedades y al final de la ruta escribimos los parámetros que le queremos aplicar como se puede ver en la figura D.8. Una vez hecho esto, siempre que hagamos doble clic en el acceso directo se ejecutará con esos parámetros.

Si todo está correctamente configurado, al hacer doble clic en el acceso directo comenzará una partida y el modelo especificado comenzará a jugar.

Guía de uso de redes neuronales

En el caso de las redes neuronales, el entrenamiento y explotación tiene algo más de trabajo. Por este motivo, he decidido hacerlo en un libro de *jupyter*.

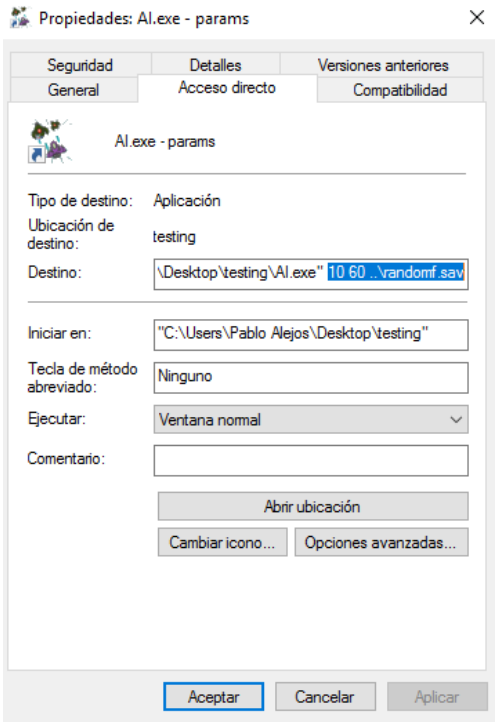

Figura D.8: Parámetros de un acceso directo

En primer lugar hay que hacer una comprobación importante. *Jupyter* utiliza también un tipo de socket para ejecutar sus servicios, y el puerto que utiliza es el 8888, el mismo que he estado yo utilizando para mis scripts. Este conflicto hace que tengamos que cambiar el puerto por defecto de *Jupyter*, ya que cambiar el del juego es algo mas engorroso. Para cambiar el puerto de *Jupyter* hay que seguir los siguientes pasos:

- 1. Abrir una consola de Windows y escribir:

  `jupyter notebook –generate-config`

  Esto genera un fichero de configuración en la ruta por defecto de *Jupyter* llamado jupyter_notebook_config.py.

- 2.Abrir el fichero que nos ha creado e ir a la línea que pone:

  c.NotebookApp.port = 8888

  y cambiar el 8888 por 7777.

- Guardamos el fichero de configuración y lanzamos *Jupyter*.

Entrenamiento y explotación Abrimos el libro *Jupyter* llamado «NeuralNetwork». Este notebook de *Jupyter* se divide en varias secciones:

- Carga de datos.
- Creación del *pipeline* inicial.
- Neuroevolición
- Calibrado de parámetros
- Bot de Telegram

Carga de datos

Este tipo de entrenamiento podría empezar sin cargar datos pero, dado que ya disponemos de un gran conjunto de datos, vamos a aprovechar los datos de los que disponemos para preentrenar un perceptrón multicapa (MLP), de esta forma se pretende reducir el tiempo de entrenamiento.

La carga de datos es sencilla. simplemente hay que ir ejecutando una a una las celdas hasta el pipeline.

En este punto tenemos dos opciones, generar un MLP con los parámetros por defecto o saltar momentáneamente al calibrado de parámetros.

Calibrado de parámetros En este proyecto se ha implementado el método de calibrado mediante random_search. Este método lo que hace es seleccionar una combinación aleatoria de entre las opciones que le da el programado y lo evalúa, al final, te dice cuál cree él que es la mejor combinación. Si se ponen demasiadas iteraciones puede tardar un buen rato, pero sabes que va a valorar más posibilidades y, por lo tanto, es más probable que de con un buen resultado.

Creación del pipeline Con el calibrado hecho, establecemos los parámetros que el calibrado nos ha proporcionado y creamos el pipeline. Un pipeline es una sucesión de operaciones que se hacen a la hora de entrenar un modelos. Este pipeline se divide en etapas, en mi caso, solo tiene dos. La primera etapa es un Standard scaler, que me ajusta los datos a una escala de forma que los datos dispersos influyan un poco menos en el conjunto. La segunda etapa es el propio MLP. Ejecutamos la celda que contiene el pipeline y tendremos un fichero nuevo llamado «pipe.sav».

Computación neuronal Para evolucionar el clasificador basta con ejecutal la celda que contiene el algoritmo evoulivo, aún así, explicaré brevemente qué es lo que hace:

Como ya se indicaba en la memoria, hay que definir la función de *fitness*, el método que nos pasa de gen a coeficientes y viceversa y la población inicial. En la sección «neuroevolución» del notebook tenemos estos tres métodos que se llaman: **gen2Coefs** y **coefs2gen** para transformar el genoma, **MLPFitness**

para el cálculo de valor de *fitness* e **initPopulation** para decir cómo ha de ser iniciarse la población inicial.

una vez definidos estos métodos hay que definir el algoritmo evolutivo. Para los algoritmos evolutivos se ha empleado la biblioteca *Deap*. Para crear nuestro modelo de algoritmo evolutivo, Deap utiliza lo que denomina *Toolbox*. Las *Toolbox* de Deap contienen la información del evolutivo como, por ejemplo, tipo de individuos, codificación del individuo, forma de mutar, tipo de cruce, etc.

Con todos los parámetros definidos ya podemos empezar a evolucionar el clasificador. Para evolucionar el clasificador, Deap nos facilita varios modelos de algoritmos evolutivos, en este caso yo utilice *eaSimple*, que es el que mas a aproxima al modelo explicado en Conceptos teóricos.

D.5. Pruebas del sistema

En el desarrollo de este proyecto, por desconocimiento del entorno de desarrollo no me sido posible realizar las pruebas automatizadas que me hubiera gustado. Por otro lado, se han realizados numerosas pruebas manuales en forma de pequeños test unitarios, con el fin de que me diesen una noción de que el funcionamiento era el esperado.

Apéndice E

Documentación de usuario

E.1. Introducción

A continuación se hace un breve comentario sobre lo que un usuario normal debería hacer para utilizar el juego.

E.2. Requisitos de usuarios

Para la utilización del juego no es necesario ningún software específico y no requiere de instalación. Para jugar solo hay que descomprimir la carpeta «AI_Juego» y ejecutar la aplicación.

E.3. Manual del usuario

Para la recogida de datos es necesario ejecutar el juego y él, automáticamente creará un fichero del tipo «GameStates_xxx» donde xxx es un valor numérico. Una vez haya completados las partidas convenientes deberá enviar el, o los ficheros generados al programador para que realice las tareas pertinetes.

Apéndice F

Datos recogidos

F.1. Introducción

En esta sección se ponen en comparativa algunos de los resultados obtenidos. Estos resultados están extraídos de un conjunto no muy grande de datos que tuve que generar yo mismo. La intención era que muchos usuarios jugasen y me entregasen los datos,pero esto no ha sido posible debido a la continua mutación del proyecto. Los datos tomados una semana a la siguiente ya no servían.

F.2. Árboles de decisión

En entrenamiento con los árboles de decisión resultó, en cierto modo, desesperanzador. El primer entrenamiento que se realizó, utilizando un random forest, resultó desastroso. En la tabla que se muestra a continuación se ha entrenado un random forest con 9000 muestras y se ha ejecutado cinco veces. Cada una de las ejecuciones conlleva diez rondas.

parrtida/ronda	1	2	3	4	5	6	7	8	9	10
1	278	-293	45	-268	-213	273	36	103	-52	257
2	164	419	1461	48	517	301	-3	86	-79	512
3	500	148	376	-84	3	-1328	44	489	-1763	119
4	222	5	220	179	878	113	-92	-1046	-72	95
5	-57	520	157	-2311	-72	-395	961	398	-72	-150

Tabla F.1: Entrenamiento con Random forest

Como se puede observar en la tabla F.1, los resultados son muy malos, tan pronto puede alcanza mas de mil puntos como acabar hundido en los valores negativos. Cuando se observaron este tipo de resultados por primera vez, se

pensó que el problema podría residir en un sobre-ajuste de la función. Esto me condujo a probar un modelo más simple, un DecissionClassifier (árbol de decisión). Este modelo, al ser algo mas simple, no se ajusta tanto.

parrtida/ronda	1	2	3	4	5	6	7	8	9	10
1	617	67	-6	95	109	261	151	-257	-76	253
2	217	66	-34	120	-174	318	200	306	-340	897
3	189	11	180	44	320	146	55	369	283	273
4	173	33	503	-394	47	-158	189	131	320	-72
5	575	426	149	-11	358	76	61	110	739	639

Tabla F.2: Entrenamiendo con DecissionTree

Estos resultados ta eran algo mejores, el ya se mueve, en cierto modo, de forma «inteligente», aunque los resultados no son los mejores. Una vez comprobado que había esperanzas de conseguir un aprendizaje, se pasó directamente a los algoritmos evolutivos.

F.3. Algoritmos evolutivos

Para el primer entrenamiento con redes neuronales se emplea un perceptrón multicapa. El algoritmo utilizado es el *eaSimple*.

En este segundo entrenamiento se ha partido como invididuo inicial el mejor del anterior entrenamiento. El modelo de algoritmo evolutivo es igual que el anterior.

Haciendo el análisis de datos me percaté de que mi función de *fitness* no era todo lo buena que yo esperaba. Al analizarlo con detenimiento me di cuenta

gen	nevals	avg	std	min	max
0	10	-738.945	1093.75	-2913.4	706.7
1	10	464.033	397.032	-394	1043.6
2	8	576.64	332.162	-67.5	1048
3	6	590.65	434.868	-549	1048
4	10	323.44	543.768	-573.7	1180.3
5	4	849.94	280.792	510.1	1180.3
6	6	729.28	580.734	-698.4	1263.5
7	8	783.804	266.751	479.909	1263.5
8	5	712.796	544.912	-312.7	1263.5
9	8	604.24	549.538	-326	1263.5
10	4	1082.16	285.419	579.091	1294.4

Tabla F.3: Entrenamiento con algoritmos evolutivos 1

gen	nevals	avg	std	min	max
0	10	-222.475	1515.29	-3194.5	2281.1
1	10	1292.45	1194.07	-1305.2	2830.4
2	8	2094.54	776.345	172.8	3143.55
3	6	2432.41	1110.06	-558.8	3650.6
4	10	1653.41	1149.34	-515.3	3162
5	4	2733.52	435.733	1907.9	3250.9
6	6	2502.06	952.354	-80	3250.9
7	8	2429.7	509.653	1850.7	3250.9
8	5	2694.2	781.888	614.5	3250.9
9	8	1929.75	1149.92	-539.7	3250.9
10	4	2952.38	515.971	1728.09	3250.9

Tabla F.4: Entrenamiento algoritmos evolutivos 2

de que no me interesa tener un individuo con mucha suerte, me interesa un individuo que siempre juega bien. Hasta el momento, toda la evolución ha tenido como meta maximizar la media de las partidas que juega un individuo, una función de evaluación a primera vista válida, pero ¿qué ocurre si dos individuos muy parecidos juegas sendas partidas, y uno es más afortunado que el otro en una única ronda? La media de este individuo sera ligeramente mayor y será marcado como más apto, y todo por azar. llegado a este punto pensé que yo no quiero saber quién tiene mas suerte, sino quién juega de una forma más regular. A partir de ahora, le voy a aplicar una penalización en función de la desviación estándar, con la esperanza de obtener como resultado individuos mas regulares.

gen	nevals	avg	std	min	max
0	10	-0.903205	2.50895	-6.84849	2.64933
1	10	1.11837	0.875644	-1.02659	2.4963
2	8	1.65532	0.904609	-0.202067	3.68612
3	6	2.40992	1.3058	-0.600178	3.68612
4	10	1.4735	1.14359	-0.479148	3.36271
5	4	2.76411	0.548491	1.93125	3.48607
6	6	2.34142	1.21775	0.115697	3.48607
7	8	2.43485	0.819595	1.40789	3.82011

Tabla F.5: Entrenamiento con nuevo fitness

Bibliografía

[1] "Wikipedia". "bsd licenses — wikipedia, la enciclopedia libre", "2016". "https://en.wikipedia.org/wiki/BSD_licenses".