Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Sistemas Inteligentes:

Proyecto final

Estudio comparativo de agentes inteligentes para recorrido de circuitos en Unity

Ayob Asrout Vargas
(alu0101350158@ull.edu.es)

Juan García Santos
(alu0101325583@ull.edu.es)

Índice:

1. Introducción.

Enlace al repositorio público del proyecto

En este estudio se comparan diferentes agentes inteligentes en su capacidad para completar varios circuitos de karting en el entorno de Unity utilizando las librerías ML-Agents y NavMesh. Se evaluarán diferentes algoritmos de aprendizaje automático, como el aprendizaje por refuerzo, el algoritmo A* y algoritmos genéticos, este último para optimizar los parámetros de cada agente al máximo y determinar cuál es el mejor en diversos términos, como la velocidad y precisión en el recorrido del circuito. Los resultados de este estudio proporcionarán información valiosa para el desarrollo de agentes inteligentes en aplicaciones de juegos y simulación en tiempo real.

2. Propuesta

Investigar y aprender sobre el uso de diferentes técnicas de inteligencia artificial en el desarrollo de agentes inteligentes para una tarea específica: recorrer de forma correcta un circuito de carreras en un entorno virtual.

2.1. Descripción

El proyecto se desarrolló para implementar diferentes tipos de agentes inteligentes y comprobar su evolución y rendimiento en diferentes entornos. El principal objetivo ha sido observar la adaptación de los agentes a los circuitos usados para entrenamiento y validación, teniendo en cuenta también otros aspectos como el tiempo que tarda el agente en completar el circuito.

2.2. Proyectos similares

- "Training my first machine learning game"(Jabrils) : El youtuber Jabrils usa redes neuronales para programar un agente capaz de dar vueltas a un circuito. Para ello, emplea la técnica del gradient-descent, específica del aprendizaje supervisado.
- "Training cars with genetic algorithms"(Holistic3D) : La youtuber Holistic3D usa algoritmos genéticos para entrenar un agente capaz de recorrer un circuito
- "ML-Agents 1.0+ Creating a Mario Kart like AI"(Sebastian Schuchmann) : El youtuber Sebastian Schuchmann usa los ML-Agents de Unity para crear agentes capaces de recorrer circuitos inspirados en mario kart

2.3. Recursos a usar

- Unity: Motor de videojuegos
- Unity ML-Agents: Paquete de Unity que permite implementar agentes de machine learning usando una API de python
- Unity NavMesh: librería empleada para implementar el algoritmo A* con un alto grado de customización
- Github : Plataforma de hosting de código para colaboración y control de versiones.
- Visual Studio: IDE preferencial para emplear en proyectos que empleen el lenguaje C#

2.4. Tecnologías de IA

- Reinforcement Learning (Proximal Policy Optimization)
- Algoritmos genéticos
- Algoritmo A*

3. Desarrollo

3.1. Lista de tareas y cronograma

- Redacción del ACP y cronograma : Redactar el Acta de Constitución del Proyecto y definir las tareas del mismo así como asignarles a cada una recursos y tiempo de ejecución ; Tiempo de ejecución : 4 Nov - 18 Nov
- Configuración del proyecto Unity : Configurar el proyecto de Unity para que funcione con los ML-Agents y su API de python ; Tiempo de ejecución : 18 Nov - 25 Nov
- Creación de los escenarios : Creación de las pistas de entrenamiento y validación de los agentes ; Tiempo de ejecución : 25 Nov - 9 Dic
- Desarrollo de lógica de juego y movimiento : Desarrollo de la lógica básica del juego, condiciones para perder, posibilidades de movimiento etc ; Tiempo de ejecución : 6 Dic - 23 Dic
- Diseño e implementación de diferentes agentes(aprendizaje automático, A*...) : Diseño e implementación de los agentes a usar, se usarán ML-Agents de Unity ; Tiempo de ejecución : 16 Dic - 2 Ene
- Estudio comparativo de los diferentes agentes : Comparar el rendimiento de los diferentes agentes usados y sacar conclusiones ; Tiempo de ejecución : 2 Ene - 16 Ene
- Elaboración de la presentación e informe final : Elaboración del informe y la presentación ; Tiempo de ejecución : 12 Ene - 23 Ene

3.2. Estado actual

Logramos desarrollar 4 escenarios donde poder comprobar la efectividad de los diferentes agentes. El agente entrenado mediante aprendizaje por refuerzo, fue entrenado en los 3 escenarios más básicos, que rotaban en cada episodio de entrenamiento de forma pseudo aleatoria, para finalmente obtener un agente capaz de completar el recorrido de forma satisfactoria.

Para la implementación del algoritmo A* empleamos la librería NavMesh, que permite registrar un área donde realizar el recorrido deseado, y calcula de forma eficiente la ruta más corta a una ubicación dentro del área, con un alto grado de configuración mediante el uso de parámetros que condicionan el movimiento y comportamiento del agente.

Finalmente, empleamos algoritmos genéticos, no para generar otro agente capaz de recorrer el circuito, si no para, partiendo de los agentes previos, optimizar al máximo sus parámetros, afinando así los agentes y comprobar si existe una mejoría significativa.

3.3. Problemas encontrados

- Dificultad para configurar el proyecto de Unity. El proceso para instalar y configurar el entorno virtual de python necesario para usar la librería es poco intuitivo, además que la documentación asociada está desactualizada en su mayoría.
- El control de versiones de github no casa bien con el desarrollo de proyectos en unity. No existe una integración fácil ni una solución cómoda para alojar archivos de más de 100 mb de peso (animaciones, modelos 3D...)
- La librería ML Agents dispone de pocos recursos formativos actualizados. Los videos y documentación más recientes suelen tener mínimo un año de antigüedad, a pesar de que la herramienta ha seguido actualizando y cambiando con el paso del tiempo.

3.4. Requisitos y Objetivos alcanzados

- Se implementaron diferentes terrenos de entrenamiento y validación (4).
- Se ha obtenido una interfaz de entrenamiento y prueba generalista de agentes inteligentes.
- Logramos implementar al menos dos agentes inteligentes diferentes para su prueba y estudio comparativo (aprendizaje por refuerzo y A*).

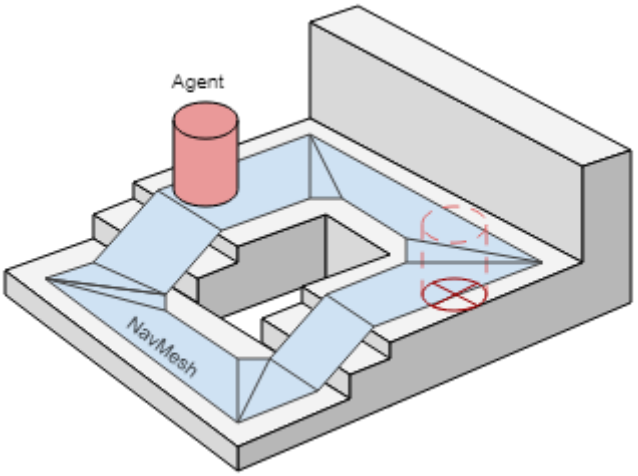
- Generamos un estudio comparativo entre diferentes agentes, con unas conclusiones asociadas, en términos de velocidad, dificultad de integración, completitud ...

4. Descripción del funcionamiento

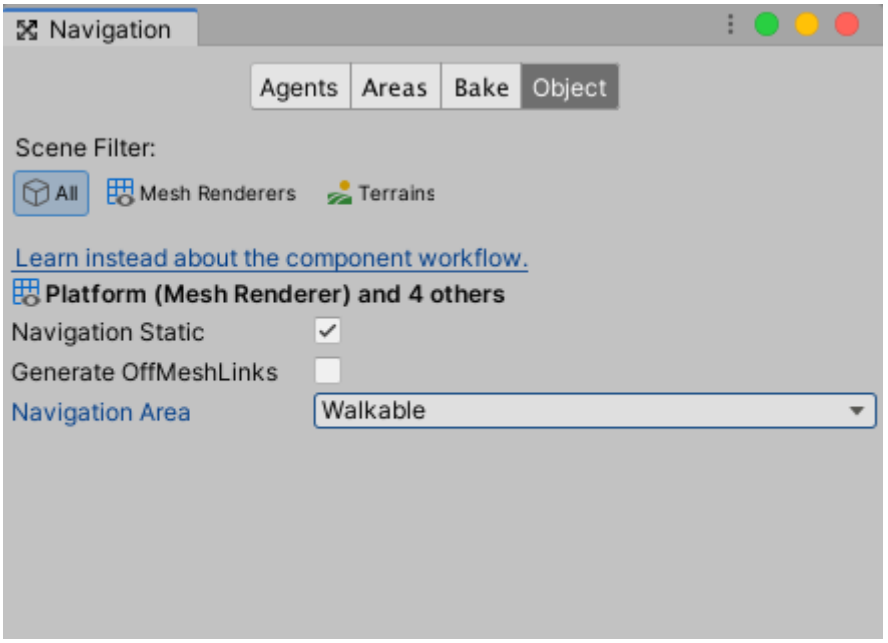
El verdadero resultado de nuestro proyecto es este informe, donde recogemos los resultados y conclusiones de nuestro estudio comparativo sobre el uso de diferentes agentes inteligentes en el ámbito del desarrollo de videojuegos.

Es por esto, que el proyecto de código aportado no resulta en una aplicación finalizada, sino más bien un entorno de desarrollo donde generar, configurar y comparar los diferentes agentes. Para poder emplearlo, es necesario tener instalado Unity Hub y la última versión de Unity. Simplemente abriendo la carpeta que contenga el proyecto descargado, desde Unity, se abrirá el mismo, mostrando los scripts desarrollados para el movimiento del Kart, los modelos 3D usados, las escenas con las pistas construidas para entrenamiento y validación, los scripts para generar y entrenar una red neuronal capaz de resolver dichos circuitos, los ficheros de configuración y el código necesario para generar un agente que se desplace empleando A* y finalmente el código empleado para optimizar los parámetros de configuración de ambos agentes previamente generados.

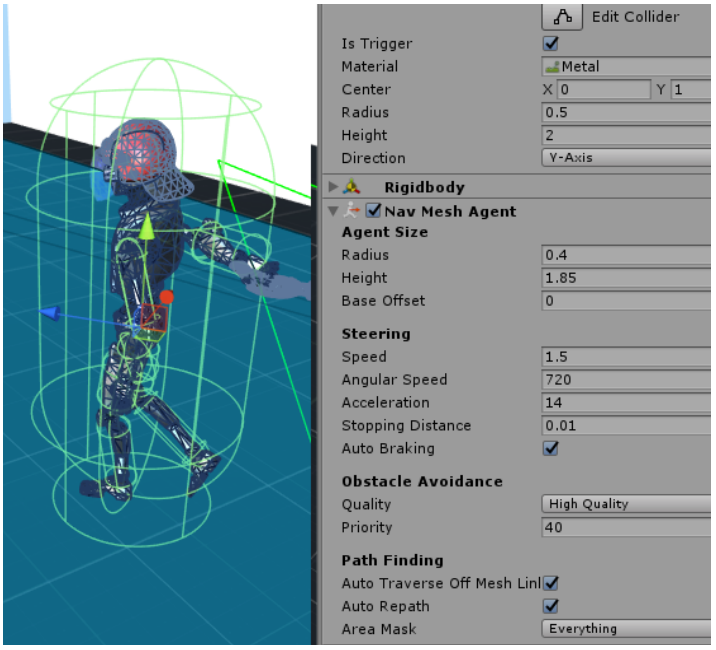
Para implementar los agentes que emplean el algoritmo A*, usamos la Librería NavMesh. NavMesh es una malla de navegación que se construye a partir de un modelo 3D del entorno. La malla se divide en varios polígonos, cada uno de los cuales representa un área accesible para los agentes. Cada polígono está conectado a otros polígonos mediante pasos o transiciones, lo que permite a los agentes moverse de un polígono a otro de forma fluida.



Para construir una malla de navegación, se utiliza el NavMesh Baker de Unity, el cual recolecta los datos del entorno, tales como la posición de los objetos, sus dimensiones y formas, para crear la malla de navegación. Esta malla se puede editar para incluir o excluir ciertos objetos o para definir áreas con diferentes características de movimiento.



Una vez construida la malla de navegación, se pueden asignar componentes NavMeshAgent a los objetos que deben moverse en el entorno. El componente NavMeshAgent proporciona una serie de funciones para controlar el movimiento del agente, como establecer un destino, evitar obstáculos, controlar la velocidad y determinar el camino más corto entre dos puntos.

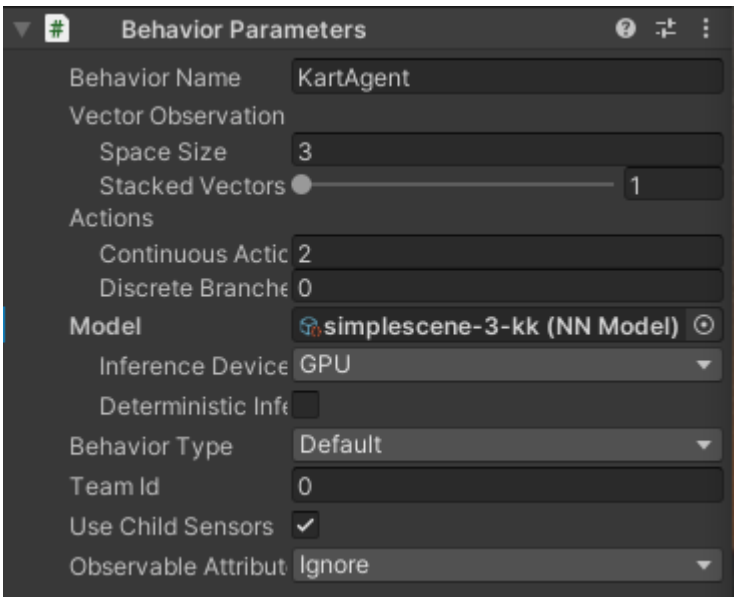


A la hora de aplicar la optimización de parámetros mediante algoritmos genéticos, comentar que este es el tipo de agente en el que más influye la optimización puesto que los parámetros disponibles a modificar influyen de manera mucho más directa en el movimiento del agente (aceleración, velocidad y velocidad angular) que aquellos disponibles en la inferencia de los agentes de ML Agents (ángulo de giro máximo, gravedad y aceleración).

Para implementar los agentes que usan machine learning, primero debemos diseñar el agente y su funcionamiento de acuerdo a cómo espera el paquete ML-Agents de Unity que funcionen. Para ello debemos crear un script que implemente de una clase llamada agent en la que deberemos definir los siguientes métodos:

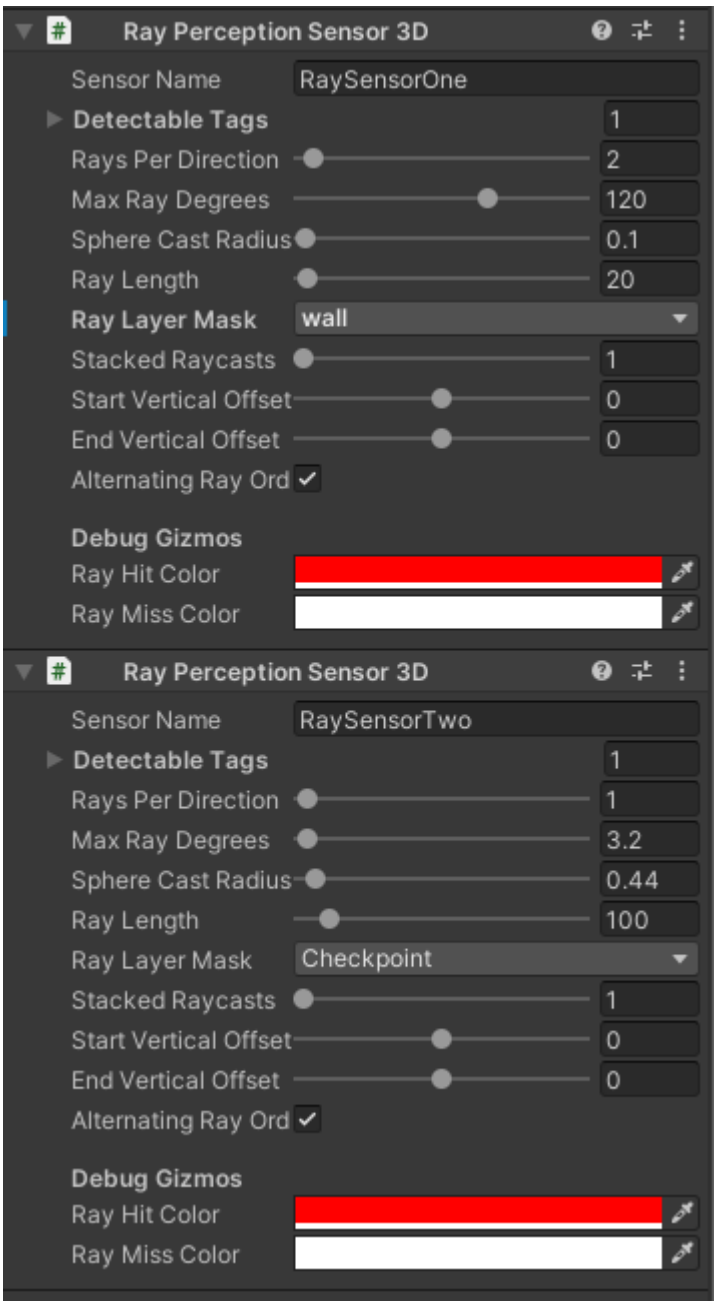
- **OnEpisodeBegin**: Encargado de definir el comportamiento a ejecutar al comenzar los episodios de entrenamiento
- **Heuristic**: Método que define el comportamiento del agente cuando se use input del usuario, esto sirve para testear que nuestro entorno de entrenamiento es correcto antes de poner a entrenar a nuestros agentes.
- **OnActionReceived**: Método que define que hacer con las acciones que se obtienen de las decisiones que toma el modelo de nuestro agente, ya sea durante el entrenamiento, o en la fase de inferencia una vez nuestro modelo haya sido entrenado.
- **CollectObservations**: Encargado de recolectar información del entorno que no sea percibida por otros sensores que podemos definir como pueden ser los raycast.

Una vez definido este script con estos métodos, se lo debemos asignar a nuestro agente, al que debemos asignar otros componentes para su correcto funcionamiento. Uno de estos sería el componente **Behavior Parameters,** en el que debemos describir varios parámetros de comportamiento de nuestro agente, como el tamaño de los vectores de observaciones y acciones, que definen el número de observaciones y acciones respectivamente, el tipo de acciones a tomar, continuas o discretas, también podemos definir si queremos usar nuestra GPU o CPU para la inferencia del modelo, el modelo a usar por nuestro agente o el tipo de comportamiento que queremos que tenga nuestro agente, si queremos que se comporte de manera heurística(input del usuario) o use el motor de inferencia para aprender o usar un modelo ya entrenado.

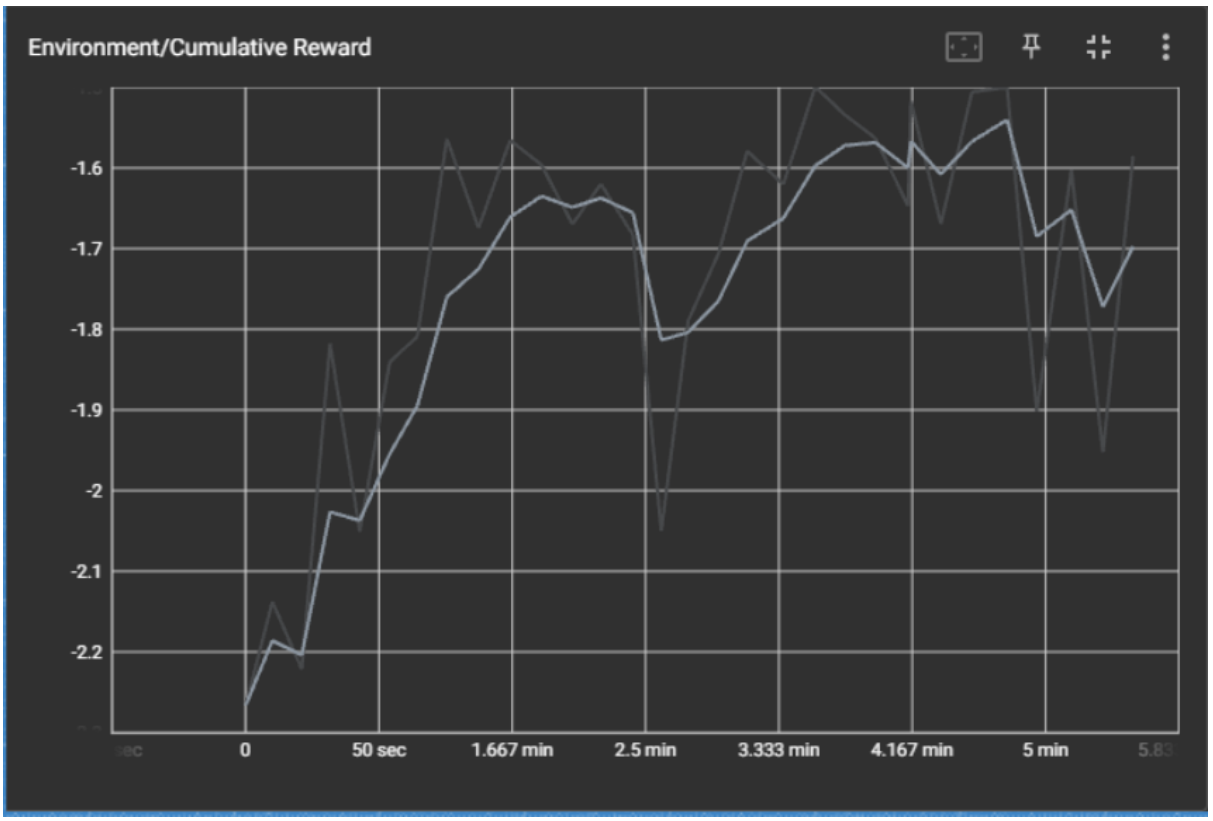


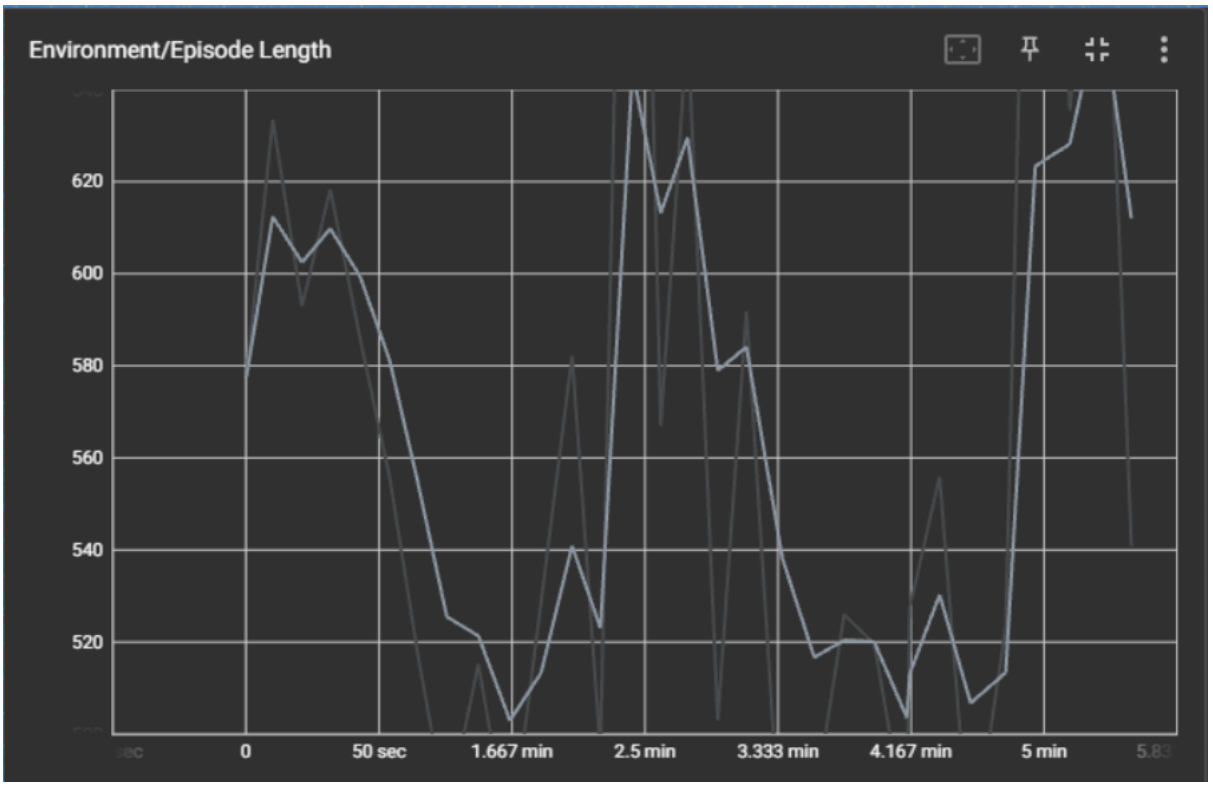
También debemos añadir el componente **Decision Requester** que será el encargado de hacer peticiones de decisiones al modelo que use nuestro agente. En este componente únicamente podemos definir el periodo que queremos usar para las peticiones de decisiones.

En nuestro caso el agente recibe una observación a través del script, esta es la distancia que hay entre si mismo y el siguiente checkpoint a alcanzar, pero además debemos añadir alguna manera de que el agente sea capaz de observar su entorno para tomar mejores decisiones. Para ello hemos optado por añadir 2 componentes llamados **Ray Perception Sensor 3D** que básicamente consisten en conjuntos de raycast que surgen desde nuestro agente y son capaces de detectar colisiones con el entorno según los configuremos. En nuestro caso hemos decidido usar 2, uno con más rayos por dirección y un mayor ángulo entre ellos capaces de detectar paredes y otros elementos del entorno, y otro con únicamente 1 rayo por direccion y angulo entre rayos restringido para que estos solo apuntes hacia adelante que únicamente sean capaces de detectar los checkpoints del circuito.



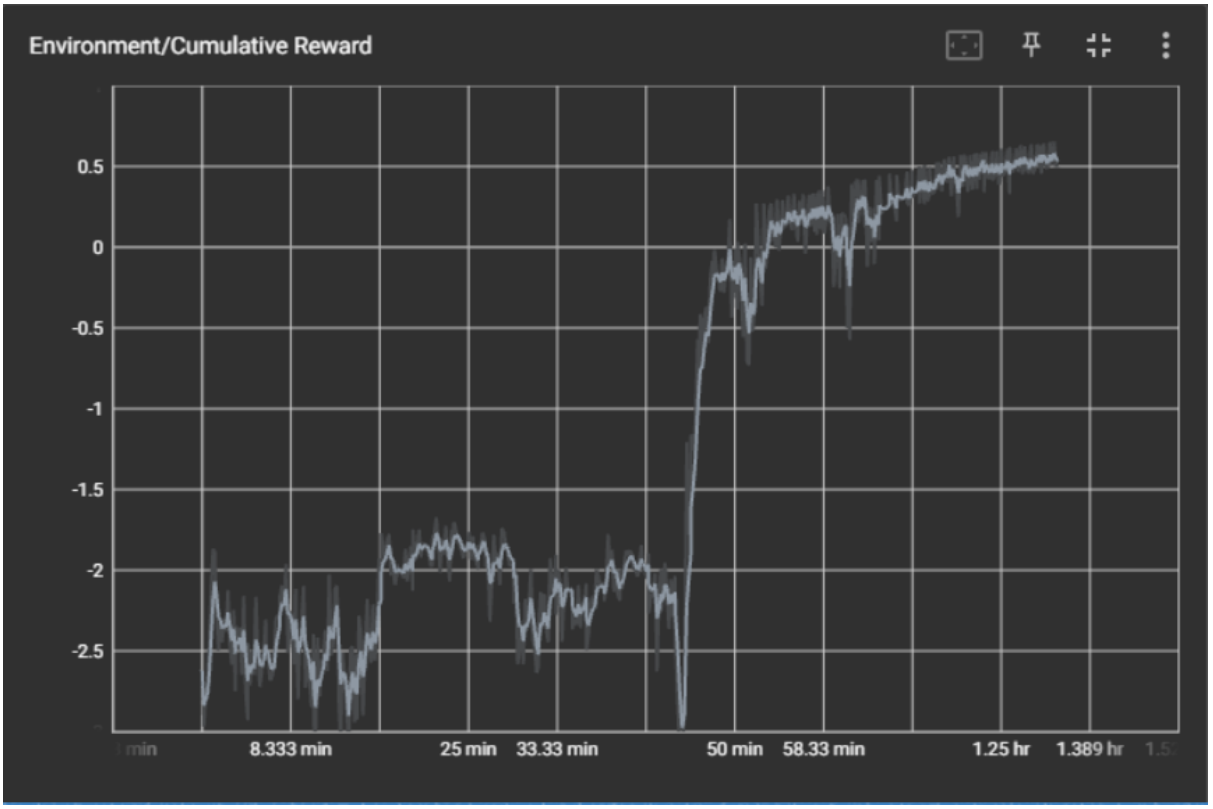
Cabe mencionar que al principio teníamos definidos los sensores para que ambos fueran capaces de detectar todos los elementos en el entorno, pero esto no era una buena decisión puesto que los sensores solo son capaces de detectar que han hecho colisión con un elemento, pero no de distinguir qué características tiene el elemento con el que han colisionado. Y por tanto, hacer una distinción en el tipo de elementos que son capaces de detectar nuestros sensores aporta información implícita a nuestro modelo siendo este capaz de tomar mejores decisiones. Como se puede observar en las siguientes gráficas.

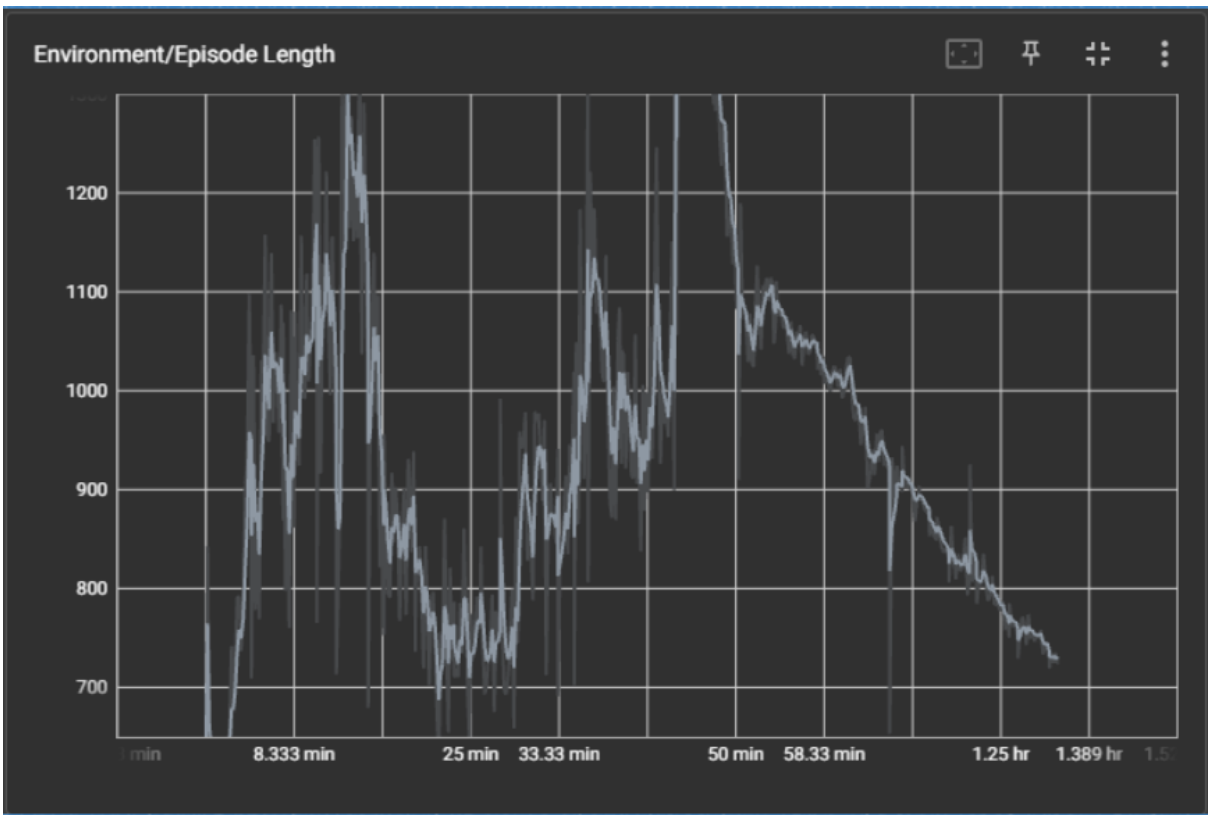




Estas 2 primeras gráficas representan la recompensa acumulada y la duración de los episodios de entrenamiento antes de ajustar los sensores. Como se puede

observar la recompensa acumulada es negativa a lo largo de toda la gráfica lo que indica un mal comportamiento de nuestro agente, además de que no se aprecia una mejoría en la duración de los episodios de entrenamiento.





Por el contrario, en estas 2 gráficas podemos observar que la recompensa acumulada a pesar de comenzar siendo negativa pasa a mejorar bastante y pasando a tomar valores positivos. Y una vez que mejora el comportamiento de nuestro agente podemos ver una reducción significativa en la duración de los episodios de entrenamiento.

En lo que respecta al uso de los ML-Agents esto sería todo lo que debemos hacer para definir un agente, pero en nuestro caso el script que define nuestra clase Agent necesita un controlador que estará encargado del movimiento del coche. Este script define métodos **ApplyAcceleration** y **Steer** que al usarlos en el **OnActionReceived** de nuestro script que define el agente seremos capaces de de aplicar aceleración o rotación a nuestro agente con las decisiones de tipo continuo que haga nuestro modelo. Esto es porque usamos el motor de físicas de Unity para el movimiento de los agentes en lugar de modificar su posición directamente aplicamos fuerzas para que estas resulten en los cambios deseados. Además el script usado como controlador del agente proporciona parámetros para la aceleración, la velocidad de giro y la gravedad a aplicar a nuestro agente. Estos parámetros suponen una cota superior para nuestro modelo a la hora de tomar decisiones, y por tanto como se mencionaba antes al intentar optimizarlos usando algoritmos genéticos una vez ya entrenado el modelo no se consigue mucha mejora.

También debemos definir las recompensas y penalizaciones que debe recibir nuestro agente. Nosotros hemos decidido que nuestro agente será recompensado cada vez que alcance un checkpoint(usando un checkpoint manager somos capaces de dar dicha recompensa únicamente al alcanzar el siguiente checkpoint del circuito, evitando dar recompensas por quedarse en el mismo sitio o ir al revés en el circuito) y cuando haga un recorrido completo del circuito. En el último caso esto supondrá también el final del episodio de entrenamiento. Y para las penalizaciones hemos decidido que nuestro agente será castigado por colisionar con las paredes y permanecer en contacto con estas, por alcanzar algún checkpoint que no sea el siguiente a alcanzar(ir al revés en el circuito o volver para atrás), al acabarse el tiempo máximo definido para un episodio de entrenamiento y además será penalizado una pequeña cantidad cada frame para fomentar una conducción más veloz a través del circuito

Una vez definido cómo serán nuestros agentes debemos definir los entornos en los que estos serán entrenados. Para ello simplemente creamos varios circuitos de carreras relativamente sencillos. Cabe mencionar que estos circuitos cuentan con paredes para evitar que los agentes se salgan del circuito, podría ser una posible mejora implementar agentes que no requieran de estas para su entrenamiento. Las paredes del circuito deben estar en su propia capa además de contar con un tag que las identifique como paredes, al igual que el suelo también deberá estar en una capa dedicada a este y tener un tag que lo identifique como tal. Esto se ha hecho para facilitar las observaciones realizadas con los raycast de los agentes, además de poder aplicar penalizaciones por colisionar con las paredes. Por último, los circuitos deberán contar también con una serie de checkpoints a lo largo de su recorrido. Y así contaríamos con un agente y un entorno en el que entrenarlo.

Por último debemos definir un fichero de configuración para el entrenamiento de nuestro agente. Este debe estar en formato YAML y debe describir los parámetros de configuración de entrenamiento de nuestro agente así como los parámetros de la red neuronal que usará este. Nosotros usamos el siguiente fichero para entrenar nuestros agentes:

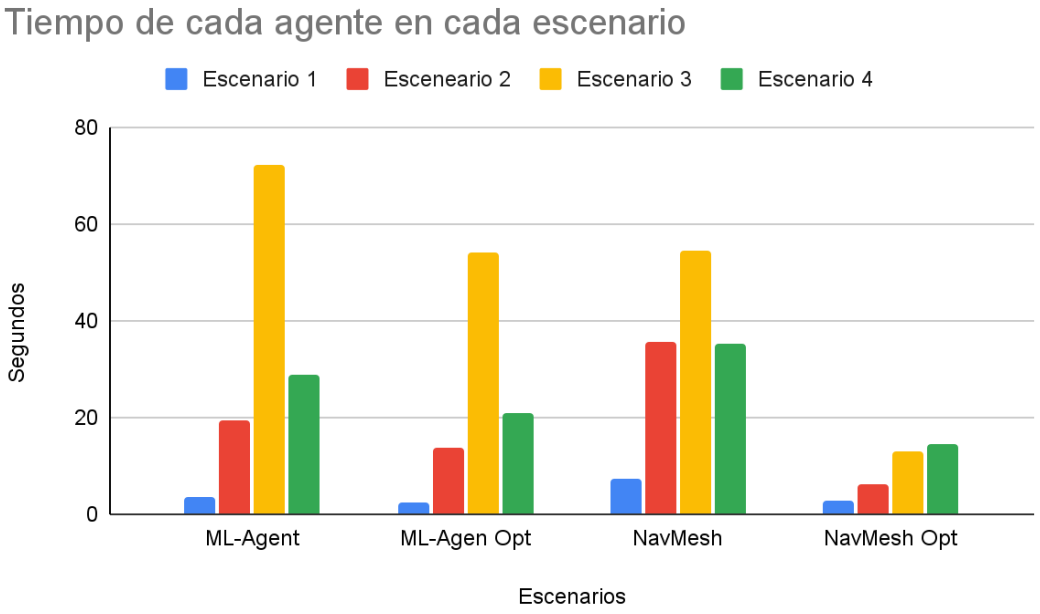
```yaml
behaviors:
  KartAgent:
    trainer_type: ppo
    hyperparameters:
      batch_size: 120
      buffer_size: 12000
      learning_rate: 0.0003
      beta: 0.001
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 256
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 500000000
    time_horizon: 1000
    summary_freq: 12000
    threaded: true
```

5. Conclusiones

A raíz de nuestro trabajo, hemos podido obtener una serie de conclusiones que aquí exponemos. Tanto los ML Agents como el NavMesh poseen sus ventajas e inconvenientes asociados a las características del entorno y finalidad de uso que se les desee dar.

Las NavMesh resultan muy sencillas de implementar e integrar, poseen un grado de configuración paramétrica suficientemente flexible sin resultar abrumador, incurren en un coste computacional medio y generan resultados de adaptabilidad satisfactorios, en tanto que el cambio a realizar entre diferentes escenarios que empleen el mismo a gente resulta en la generación de la malla, lo cual es fácilmente realizable con un par de clicks.

Por otro lado, los ML Agents poseen una complejidad tanto de implementación como de integración muy superiores al NavMesh, presentan un gran número de parámetros de configuración, generan resultados con buena adaptabilidad entre diferentes escenarios y respecto al coste computacional, mencionar que incurren en una ejecución pesada tanto en entrenamiento como en inferencia. Esta característica que poseen de necesitar un entrenamiento previo aumenta considerablemente el tiempo de desarrollo e integración, pero sí es cierto que a la hora de realizar inferencia el coste computacional disminuye en gran medida



Desde un punto de vista numérico, en la gráfica anterior podemos observar cómo las ideas generales planteadas se mantienen, pues en todos los circuitos el

mejor agente resulta ser el NavMesh con la optimización de parámetros mediante algoritmos genéticos aplicada. De todas formas, en la mayoría de los escenarios el agente de ML Agents suele obtener mejores resultados que el NavMesh sin haber aplicado la optimización de parámetros.

En conclusión, habiendo definido las características de ambos sistemas, consideramos que para la gran mayoría de casos es preferible usar NavMesh, puesto que son mucho más sencillos y rápidos de integrar e implementar, además de que generan muy buenos resultados con un coste computacional medio. Por otro lado, los ML Agents suponen una mejor alternativa en casos de uso específicos donde se tenga un gran conocimiento del dominio del problema a resolver así como el suficiente tiempo como para no solo parametrizar correctamente los agentes sino plantear un esquema de entrenamiento que evite el sobre ajuste y permita generar unos modelos capaces de obtener resultados aún mejores con el relativo coste que supone la inferencia sobre el mismo.

6. Bibliografía

1. ML Agents Toolkit (enlace).
2. Construyendo un Navmesh Unity (enlace).
3. Unity Navmesh Basics (enlace).
4. How to use genetic algorithm for hyperparameter tuning of ML models? (enlace)
5. ABMU: An Agent-Based Modelling Framework for Unity3D (enlace)
6. Machine Learning Adversaries in Video Games (enlace)