El proyecto consiste en la aplicación de algoritmos de refuerzo a un brazo robot para que complete ciertas tareas dentro de un entorno virtual. El entorno virtual que se ha utilizado ha sido Unity, ya que cuenta con ML-Agents, un framework para aplicar aprendizaje por refuerzo de forma sencilla a proyectos de Unity.

El brazo robot que se ha usado ha sido el “Universal Robot UR3” el cual en el [github](https://github.com/Unity-Technologies/articulations-robot-demo/tree/mlagents) del proyecto ya cuenta con una implementación de este framework con el objetivo de que la pinza del robot toque la pieza. El objetivo de este proyecto siempre ha sido que partiendo de esta base el hacer que el robot pudiera agarrar la pieza, depositarla e incluso interaccionar con otros agentes de una forma colaborativa.

ML-Agents

Es un proyecto open source que permite que diferentes juegos y simulaciones sirvan como entornos de entrenamiento para agentes inteligentes. Ofrece implementaciones, basadas en pytorch o tensorflow, de diferentes algoritmos para permitir a los desarrolladores de videojuegos entrenar de forma sencilla agentes para sus videojuegos. Estos agentes entrenados se pueden usar para múltiples propósitos, controlar personajes no jugables (PNJ) en entornos adversariales o multiagente, testeo automatizado de juegos e incluso diferentes decisiones de diseño antes de la salida de un videojuego. Aunque no solo es beneficioso para los desarrolladores de videojuegos, sino que los investigadores de inteligencia artificial también pueden usar ML-Agents como una plataforma donde se pueden evaluar los diferentes avances en esta materia.

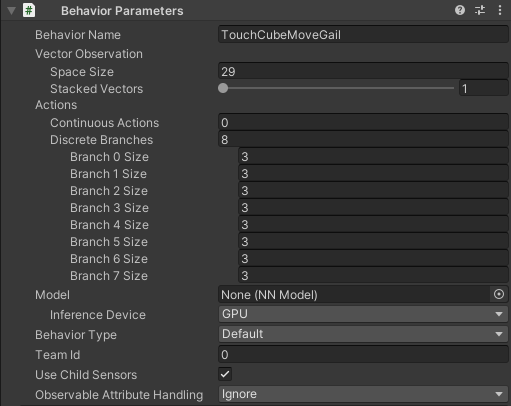
Características:

* Soporte para múltiples configuraciones de entorno y escenarios de entretenimiento.
* Un SDK de Unity flexible que puede ser integrado en un juego o escena
* Soporte para entrenar en entornos de agente único, multiagente colaborativos o multiagente adversariales gracias a algoritmos de aprendizaje profundo como PPO, SAC, MA-POCA
* Soporte para aprendizaje mediante ejemplos gracias a los algoritmos Behavioral Cloning y GAIL(Generative Adversarial Imitation Learning)
* Poder definir escenarios de aprendizaje por curriculum para tareas mas complejas.
* Entrenar agentes robustos gracias a la aleatorización.
* Entrenar agentes usando múltiples instancias de entrenamiento de forma simultánea.



Los diferentes algoritmos de aprendizaje mediante refuerzo, siguen un ciclo de recibir observaciones, calcular decisiones, realizar esas acciones y finalmente recibir las recompensas. Este ciclo se repite en cada frame del entrenamiento y ML-Agents tiene herramientas específicas para cada una.

En los diferentes scripts de los agentes de ML-Agents deben heredar del objeto Agent, para poder implementar los diferentes métodos para seguir con el ciclo mencionado anteriormente. Además de esto, en Unity se debe añadir el script que herede de este objeto, ya que añade automáticamente otro script llamado *Behaviour Parameters* que serán importantes a la hora de obtener las Observaciones y la cantidad de acciones.



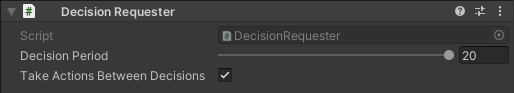
Observaciones:

El método que hay que implementar es *CollectObservations()*, estas serán todas las entradas que va a recibir la red neuronal mediante un vector de sensores. Mediante *sensor.AddObservation()*, se puede añadir cualquier tipo de valor como entrada a la red neuronal. Se pueden introducir cualquier tipo de datos, lo que hay que tener en cuenta es que en Unity existen unos tipos de variables como los Vector3 que son utilizados para posiciones en entornos tridimensionales, por lo que si se añade una entrada de este tipo, se añaden 3 variables distintas a la entrada de la red neuronal.

Lo que hay que configurar en el *Behaviour Parameters* es la del tamaño del espacio de observaciones.

Decisiones:

Para que el agente haga sus propias decisiones, hay que añadir el script *Decision Requester* en el cual indicamos cada cuánto se va a pedir una nueva decisión a la red neuronal y si se pueden hacer o no acciones durante este tiempo de decisiones.



En el caso de que queramos realizar nuestras propias acciones de cara a probar que el entorno que se ha diseñado es correcto en el script de *Behaviour Parameters* en la parte de *Behaviour Type* se debe seleccionar la opción *Heuristic Only* y en el codigo implementar la función *Heuristic* y implementar las decisiones a mano.

Acciones:

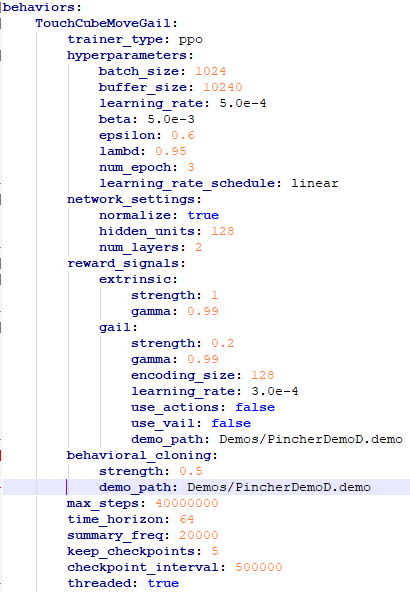
Para realizar las acciones hay que hacer dos cosas, primero definir en el *Behaviour Parameters* la cantidad de acciones que existen para el agente y el tipo del que son, continuas o discretas, y segundo, implementar el método OnActionRecieved() donde se reciben las decisiones y se deben realizar las acciones en el agente.

Recompensas:

Finalmente, para terminar este ciclo, en el mismo método anterior se deben implementar las diferentes condiciones para otorgar las recompensas al agente. Existen 2 métodos distintos en relación a esto, AddReward() y SetReward(). El primero sirve para definir la recompensa del episodio enero, mientras que el segundo es para el último step que se ha hecho.

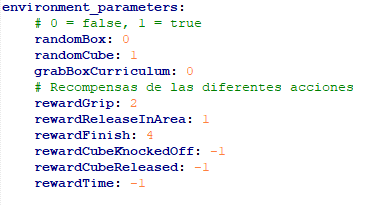
Archivo de configuración YAML:

A la hora de querer entrenar un agente se necesita de un archivo de configuración en formato yaml en el que se tienen que especificar los diferentes algoritmos hiperparametros, señales de recompensa



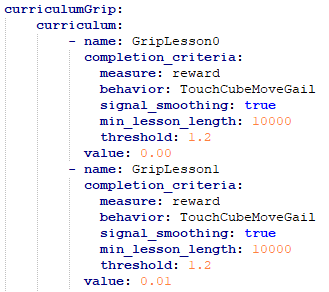
En este ejemplo no se muestra, pero permite activar funcionalidades antes mencionadas, como el GAIL, Behavioral Cloning, Curiosity… Es muy importante tener a mano la [documentación de ML-Agents](https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-ML-Agents.md) para investigar a fondo cuales son las posibilidades de entrenamiento que ofrece esta herramienta.

Además de esto permite la definición de variables de entorno que se pueden trasladar directamente a los scripts de unity.



En este caso, los parámetros de entorno que tenemos, son para definir la aleatoriedad de elementos del entorno y las recompensas que otorgamos por diferentes eventos como agarrar la pieza, soltarla, tirarla fuera de la mesa…

Finalmente, el yaml de configuración permite crear escenarios de curriculum en los que los parámetros van cambiando según avanza el entrenamiento. Se asigna el nombre de la variable de entorno, se pone que es curriculum y en forma de lista se van añadiendo las diferentes lecciones del curriculum. Primero se le pone nombre a la lección, lo siguiente es especificar el criterio para considerar que ha sido completada, en este caso lo que medimos es la media de la recompensa sea de 1.2 durante 10000 episodios y al final del todo se especifica el nuevo valor del parámetro de entorno.



**Escenario Brazo Robot en Mesa**

En este escenario, nos encontramos con el brazo robot, el cubo y la caja de destino todos en la misma mesa. El objetivo es que el agente del brazo robot aprenda a agarrar el cubo para que después lo deposite en la caja de destino.

Script del Agente Brazo Robot (Escenario de la mesa)

Al script del agente se le pasan varios elementos:

* La pinza
* El cubo
* El Robot
* La mesa
* Y varios elementos sobre la caja donde debe terminar el cubo

En la función *Awake()* se reciben los parámetros de entorno del yaml y se guardan en un objeto *SaveRewards* que hemos creado con ese mismo objetivo. También se obtienen los controladores del robot y la pinza, y varios scripts para la detección de colisiones.

En la función *OnEpisodeBegin()* que se ejecuta al inicio de cada episodio se encarga de parar el robot, devolverlo a su posición original, resetear variables y de hacer aparecer el cubo de forma aleatoria y en rotaciones aleatorias si está definido en el archivo yaml.

En la función *CollectObservations()* lo primero que se recibe es si el cubo está agarrado o no y la fuerza que ejerce la pinza. Después de esto, si la pieza no está agarrada, se introduce la rotación de la pieza y de la pinza, lo siguiente, es la posición relativa del robot con la pieza que se obtiene de restar estas dos posiciones. Se repite lo mismo con la posición relativa del robot con la pinza y la resta de estas dos. Finalmente, hay 3 vectores de posiciones que están a 0, pero que si la pieza está agarrada se hacen los mismos cálculos que se hacen con la pieza, pero con los valores de la caja y los valores relativos de la pinza están a 0.

En el método *OnActionRecieved()*, se recibe un vector de acciones en las que están las decisiones que ha tomado el *Decision Requester* que se ha mencionado anteriormente. Para cada joint del robot, primero se pasa por un método llamado *ActionIndexToRotationDirection()* para obtener la dirección que se debe rotar el joint, y después se llama a *robotController.RotateJoint()* para rotar una joint concreta hacia esa dirección (el valor false de esa función es importante, por que hace que a cada joint que movamos, el robot no se pare). Lo siguiente es el movimiento de la pinza, si se abre o se cierra, y se le hace un tratamiento similar a las joints con la función *GripStateForInput().*

A partir de este punto, que se han realizado las diferentes acciones en el agente se comprueban las diferentes condiciones que hacen que se reciban recompensas:

* Si se agarra el cubo (+2)
* Si tira el cubo fuera de la mesa (-1)
* Si suelta el cubo dentro del área de destino (+1)
* Si el cubo toca el destino (+4)
* Si el robot suelta la pieza (-1)
* Por tiempo (-1 / MaxStep)
* Si se agarra el cubo:

Se considera que el robot ha agarrado el cubo si:

* El punto central de la pinza (que se obtiene con *pincherController.CurrentGraspCenter()*) se encuentra dentro de los límites del cubo.
* Si el valor de agarre de las pinzas (*pincherController.grip*) es mayor que 0.5.
* Los colliders individuales de las pinzas están tocando el cubo
* Si el cubo está en el aire

Todo esto, está definido en la función *EstaAgarrado()*.

En el caso de que estemos entrenando por curriculum, primero se entrena a que agarre el cubo, y después se le deja terminar el escenario completo, por lo que hay if que si el flag de curriculum está activo termina el episodio.

* Si tira el cubo fuera de la mesa:

Se comprueba si el cubo está fuera de los límites de la mesa, si ocurre se da la recompensa negativa y se termina el episodio.

* Si la pieza se suelta dentro de los límites del área de destino:

Se calculan los límites del área de destino, y se da una recompensa, para evitar que se de múltiples veces, activa un flag llamado *cubeIsReleased*.

* Si el cubo toca el destino:

Se da la recompensa por tocar el área verde del entorno y se termina el episodio.

* Si el robot suelta la pieza:

Si se detecta que el robot ha soltado la pieza, se da la recompensa negativa por soltarlo, y se termina el episodio.

* Recompensa por tiempo:

En problemas de aprendizaje por refuerzo se da una recompensa negativa al agente a cada paso del entrenamiento, esto se hace para forzar a que el agente busque que es lo que tiene que hacer para recibir recompensas, y una vez ha descubierto esto, optimizarlo.

Estrategias

* Imitation Learning

Una de las primeras estrategias que hubo en el proyecto fue la de usar grabaciones de las acciones para acelerar el proceso de entrenamiento, ya que pensábamos que este era un entorno demasiado complicado como para dejarle a él solo aprender una estrategia desde cero. Para ello, usamos dos herramientas que nos proporciona ML-Agents que son GAIL (Generative Adversarial Imitation Learning) y Behavioral Cloning que si lo alimentamos con grabaciones hechas por un componente que hay que añadir al Agente llamado Demonstration Recorder.

Los resultados que obtuvimos de las pruebas realizadas con Imitation learning fueron positivas, aprendiendo rápidamente que es lo que es lo que debía hacer, pero cuando intentamos complicar un poco más el entorno añadiendo posición del cubo aleatoria no fue capaz de aprender y por ello tuvimos que recurrir a Curriculum Learning.

* Curriculum Learning

Curriculum learning se basa en empezar con una tarea sencilla que el agente sea capaz de realizar, y una vez lo haya logrado, ir complicando cada vez más y más. Para esto decidimos no usar Imitation learning, por lo que tuvo que aprender a agarrar desde 0 lo que aumenta el tiempo de aprendizaje. Para evitar esto, realizamos unos pasos de currículum iniciales, en los que posicionamos la pinza próxima al cubo, con el objetivo de que aprenda a agarrar el cubo y que lo haga de forma horizontal. Una vez logrado esto, lo vamos alejando hasta que la posición inicial del robot esté totalmente estirado.

Una vez se ha logrado esto, lo siguiente es ir añadiendo que la posición del cubo sea aleatoria en incrementos pequeños, a cada paso del curriculum lo aumentamos en 1cm a cada dirección. Finalmente, se introduce la rotación de la pieza siguiendo un planteamiento similar.

Los resultados de usar curriculum learning fueron muy satisfactorios, aprendiendo a agarrar la pieza desde gran parte de la mesa, y además teniendo en cuenta que la rotación era totalmente aleatoria. Algo que no hubiera sido posible mediante la estrategia de imitación. Aunque, hay que tener en cuenta que al dejar al agente que explore de forma totalmente libre, el tiempo de entrenamiento es mucho mayor.

**Escenario Brazo Robot en Carro**

En este escenario el objetivo sigue siendo el de agarrar el cubo y depositarlo en la caja de destino, sólo que en vez de estar estático, se encuentra sobre un carro que puede transportar al brazo de un lado a otro. Además de esto, ahora tenemos dos mesas distintas separadas, las cuales una tiene el cubo y la otra es la caja. La idea general de este escenario es que el agente del carro transporte al brazo a la mesa del cubo, una vez se acerque el brazo detecta el cubo y lo agarra. Una vez se ha hecho el agarre el carro transporta al brazo junto con la caja para que lo deposite.

Script del Agente Brazo Robot (Escenario del carro)

Este script es muy similar al antes descrito para el escenario del brazo en la mesa. La mayoría de cambios que se deben hacer para este escenario son en los maximos y minimos para la normalización y que una vez el brazo haya aprendido a que debe agarrar el cubo no baje cualquiera de sus partes por debajo de la mesa, para enseñarle a que no haga esto, cada vez que una joint del robot baja por debajo de la mesa se le da una recompensa negativa y se reinicia el entorno.

Además, también se le pasa el GameObject del carro, para que tenga acceso al script del agente y poder terminar de forma simultánea el entorno.

Script del Agente Carro

Al script del agente carro se le pasan varios elementos:

* El carro
* El Robot
* La guia se que se usa para marcar la dirección
* El cubo
* La caja donde depositar el cubo

En la función *Initialize()* se reciben los diferentes rigidbodies y colliders que se van a utilizar en el agente.

En la función *OnEpisodeBegin()* se usa para resetear todos los parámetros del agente al inicio de cada episodio, se devuelve la posición y la rotación a los estados iniciales. En el caso que estemos enseñando al robot a agarrar el cubo, esta posición y rotación pueden no ser las del centro del escenario, sino junto a una mesa cerca del cubo.

En la función *CollectObservations()* se le pasan al agente todos los valores que el agente observa. En el entorno final tendría acceso a ver si el cubo está agarrado o no por el brazo robot, pero para el entrenamiento del carro se sustituye por una variable que marca si el carro se ha acercado o no al cubo y es lo primero que le entra como observación al agente del carro. Lo siguiente es la rotación del carro y finalmente la dirección normalizada hacia el cubo o la caja dependiendo de si se ha “agarrado” previamente el cubo.

En la función *OnActionRecieved()* se hacen varias cosas. Lo primero es mover al agente, para ello se mantiene la velocidad angular (en todos los ejes) y rotación en los ejes X y Z a 0 debido a que el carro se puede girar debido a las interacciones con el brazo robot. Se llama al método *MoveAgent()* para mover al carro y finalmente se teletransporta al brazo robot encima del carro para que se mueva el brazo al mismo tiempo que el carro. Lo siguiente es hacer el cálculo de la recompensa, que a cada paso se le da una recompensa desde 0 a 0.001 dependiendo de la distancia con el cubo o la caja dependiendo si se ha “agarrado” el cubo. Además de esto, una vez se ha acercado lo suficiente recibe una recompensa única mucho mayor, +1 por acercarse al cubo y +4 por acercarse a la caja una vez se ha acercado al cubo. Finalmente se añade una recompensa negativa por tiempo para que intente optimizar la tarea.

En cuanto al entrenamiento de este agente, se hace de forma independiente, es decir, el robot no hace ningún tipo de movimiento y el agente solamente tiene que moverse por el escenario para acercarse primero al cubo y después a la caja. Para entrenar se usa el archivo “EntornoCarro.yaml"