PEC 3 – Aprendizaje en  
videojuegos

Joaquín López Soriano

Inteligencia artificial para videojuegos

Universidad Oberta de Cataluña

Contenido

[Actividad 1: Roller Ball tutorial 3](#_Toc155550659)

[Instalación 3](#_Toc155550660)

[Realización 4](#_Toc155550661)

[Bibliografía 13](#_Toc155550662)

[Actividad 1 13](#_Toc155550663)

[Instalación 13](#_Toc155550664)

[Realización 13](#_Toc155550665)

# Actividad 1: Roller Ball tutorial

Para realizar esta actividad, se ha de usar el módulo *ML-Agents*, una librería que ofrece un conjunto de algoritmos de aprendizaje por refuerzo e imitación diseñados para su uso con entornos *Unity* compilado en el lenguaje de programación *Python*.

## Instalación

Primero hay que importar la librería *ML Agents* al proyecto de Unity para definir el agente a entrenar. Se ha de realizar mediante la secuencia de acciones:

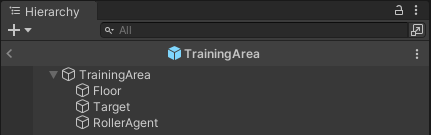
1. En el menú, dar click sobre la opción *Window*
2. Seleccionar la opción *Package Manager*
3. Seleccionar la opción *Packages: Unity Registry* de la lista desplegable
4. Buscar la librería *ML Agents*
5. Instalar e importar la librería en el proyecto

A continuación, se ha de instalar la librería de Python *mlagents* en el proyecto para poder entrenar la IA mediante el aprendizaje por refuerzo. Se ha de realizar mediante la secuencia de comandos:

1. python -m venv venv //crear un entorno virtual
2. venv\Scripts\activate //activar el entorno virtual
3. //instalar el gestor de paquetes de Python  
   python -m pip install --upgrade pip
4. //instalar biblioteca de aprendizaje automático  
   pip3 install torch~=1.13.1 -f <https://download.pytorch.org/whl/torch_stable.html>
5. pip3 install mlagents //instalar el módulo mlagents
6. //instalar el módulo protobuf para serializar datos estructurados  
   pip3 install protobuf==3.20.3
7. //mostrar la ayuda para comprobar su correcta instalación  
   mlagents-learn –help

## Realización

Para realizar esta actividad, se ha seguido el tutorial de la siguiente página web: <https://unity-technologies.github.io/ml-agents/Learning-Environment-Create-New/>.

Para ello, se ha creado el *prefab* *TrainingArea* con los siguientes objetos:

* *Floor*: un suelo, una superficie cuadrada sólida y llana donde rodar la esfera y ubicar el cubo.
* *Target*: el cubo que la esfera ha de alcanzar
* *RollerAgent*: la esfera, el agente que se ha de entrenar por la inteligencia artificial para lograr llegar hasta el cubo.

Por último, se asigna una textura a cada uno de los objetos mediante el recurso *PBR Materials Sampler Pack* obtenido de la *Unity Asset Store*. El resultado final del *prefab TrainingArea* es el siguiente:

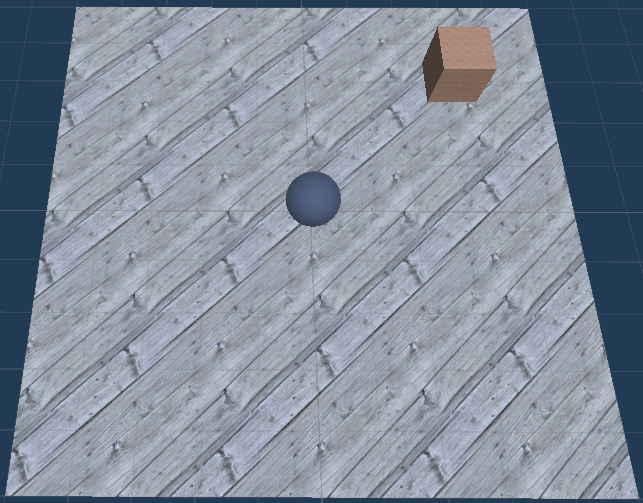


Ilustración : Prefab TrainingArea

Para el control y testeo del comportamiento del objeto *RollerAgent* mediante el comportamiento heurístico se ha agregado el módulo *Input System* para la gestión de los controles de entrada.

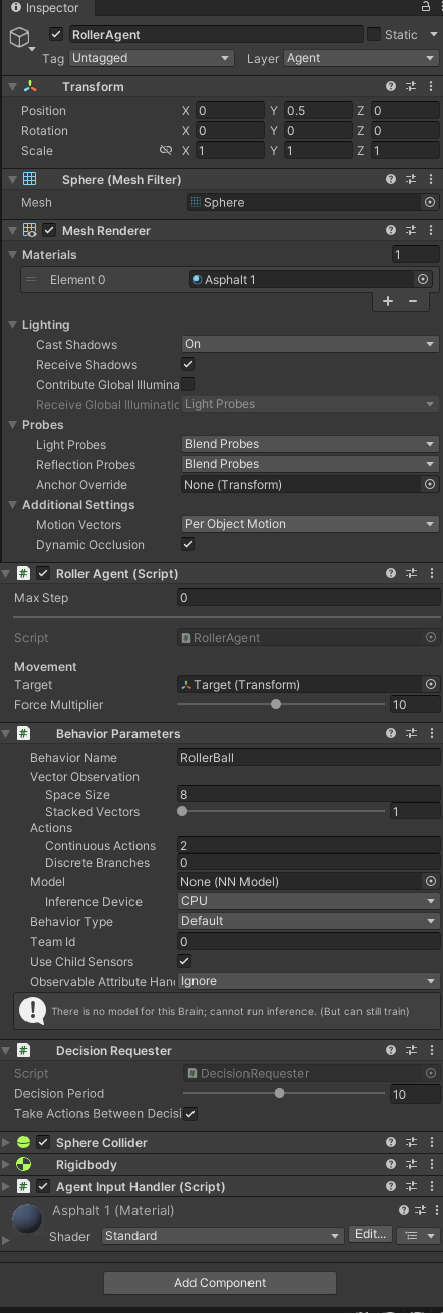
Los componentes necesarios para asociar las acciones con los controles de entrada se han definido en la carpeta *Input* y el script *AgentInputHandler* que mapea las acciones de entrada se encuentra en la carpeta *script/Input*.

Este script se ha asociado al objeto *RollerAgent.*

Después, se define el código del agente en el script *RollerAgent* asociado al objeto *RollerAgent*. El código del script es el siguiente:

*/\*  
\* Handles the behavior of the RollerAgent and sets the implementation \* of the agent  
\*/*public class **RollerAgent** : Agent  
{

[Header("Movement")]  
 *// the objective of the agent, the gameObject to reach* [SerializeField] private Transform **target**;  
 *// the force applied to move it* [SerializeField] [Range(1f, 20f)] private float **forceMultiplier** = 10;  
 *// how near to the target is has to be to reach it* [SerializeField] [Range(0.5f, 2f)] private float **minDistanceTarget** = 1.5f;  
   
 *//properties  
 // the rigid body* private Rigidbody \_rBody;  
 *// the input key manager* private AgentInputHandler \_agentInputHandler;  
   
 *// Start is called before the first frame update* void **Start**()  
 {  
 *// sets an instance of the components* \_agentInputHandler = GetComponent<AgentInputHandler>();  
 \_rBody = GetComponent<Rigidbody>();  
 }  
   
 */\*  
 \* When the agent reaches the target or falls ->  
 \* Reset the agent position and the target properties  
 \*/* public override void OnEpisodeBegin()  
 {  
 *// If the Agent fell, zero its momentum* if (this.transform.localPosition.y < 0)  
 {  
 \_rBody.angularVelocity = Vector3.zero;  
 \_rBody.velocity = Vector3.zero;  
 transform.localPosition = new Vector3( 0, 0.5f, 0);  
 }  
  
 *// Move the target to a new spot* target.localPosition = new Vector3(Random.value \* 8 - 4, 0.5f,  
 Random.value \* 8 - 4);  
}  
  
 */\*  
 \* the information we collect for the training of the IA  
 \*/* public override void CollectObservations(VectorSensor sensor)  
 {  
 *// Target and Agent positions* sensor.AddObservation(target.localPosition);  
 sensor.AddObservation(transform.localPosition);  
  
 *// Agent velocity* sensor.AddObservation(\_rBody.velocity.x);  
 sensor.AddObservation(\_rBody.velocity.z);  
}  
  
 *// ReSharper disable Unity.PerformanceAnalysis  
 /\*  
 \* Receives actions and assigns the reward  
 \*/* public override void OnActionReceived(ActionBuffers actionBuffers)  
{  
 *// Actions, size = 2* Vector3 controlSignal = Vector3.zero;  
 controlSignal.x = actionBuffers.ContinuousActions[0];  
 controlSignal.z = actionBuffers.ContinuousActions[1];  
 \_rBody.AddForce(controlSignal \* forceMultiplier);  
  
 *// Rewards* float distanceToTarget = Vector3.Distance(this.transform.localPosition,   
 target.localPosition);  
 *// Reached target ++ positive reward* if (distanceToTarget < minDistanceTarget)  
 {  
 SetReward(1.0f);  
 EndEpisode();  
 }   
 *// Fell off platform -- negative reward* if (transform.localPosition.y < 0)  
 {  
 EndEpisode();  
 }  
}  
   
 *// ReSharper disable Unity.PerformanceAnalysis  
 /\*  
 \* Control the agent ant the heuristic behavior  
 \*/* public override void Heuristic(in ActionBuffers actionsOut)  
{  
 Vector2 movement = \_agentInputHandler.GetMoveInput();  
 var continuousActionsOut = actionsOut.ContinuousActions;  
 continuousActionsOut[0] = movement.x;  
 continuousActionsOut[1] = movement.y;  
}  
   
}

 A continuación, se ha asignan el resto de componentes al objeto *RollerAgent* obteniendo el resultado de la imagen.

Se crea un documento rollerball\_config.yaml con los parámetros que utilizará el programa de entrenamiento de la IA.

El contenido se copia del publicado en el tutorial.

Para acelerar el proceso de simulación, se ha copiado el prefab *TrainingArea*, obteniendo un total de 25 prefabs en la escena.

Por último, se ejecuta el programa de entrenamiento de la IA cuyo identificador es *RollerBallTest01* mediante el siguiente comando:

mlagents-learn config/rollerball\_config.yaml --run-id=RollerBallTest01

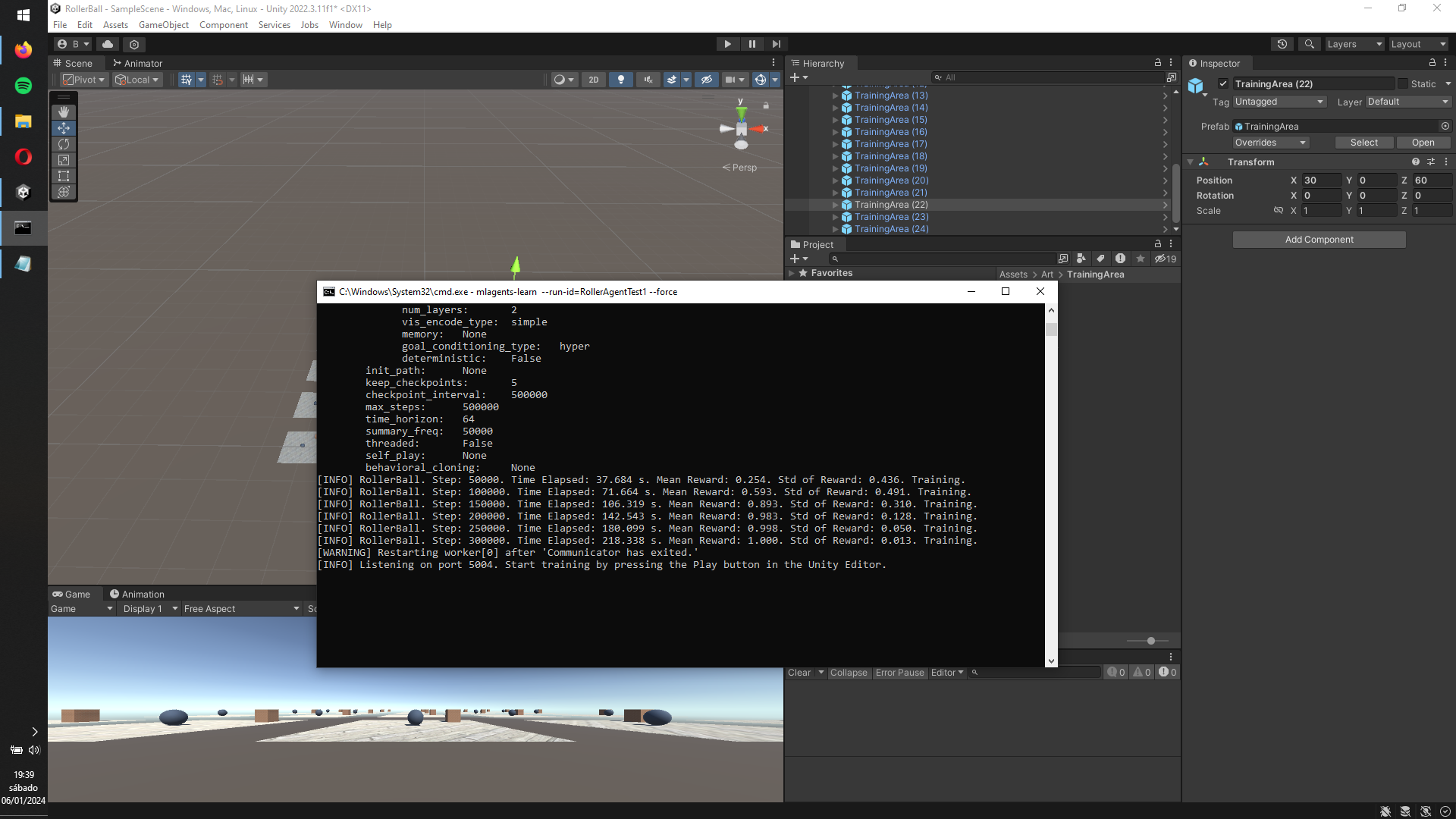
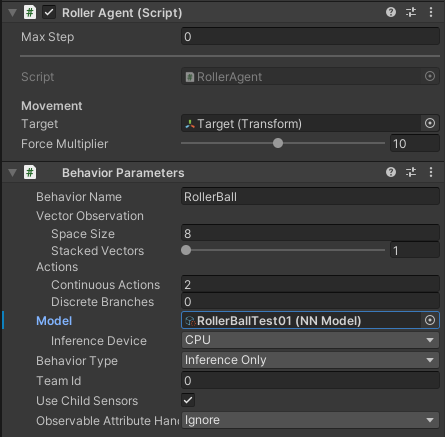


Ilustración : resultados entrenamiento IA

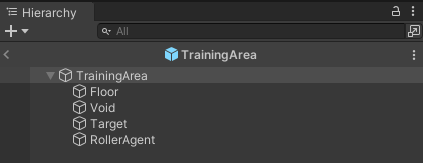
ejecutando un total de 300.000 simulaciones.

**Para comprobar que el entrenamiento ha sido correcto, se asigna el modelo generado al agente y se cambia el comportamiento a inferencial en los parámetros del script *RollerAgent.*

El resultado es el esperado y se muestra en el siguiente video: <https://youtu.be/s4iUtUaHMG8>

El video se ha realizado dejando activo un solo prefab de *TrainingArea*.

También se ha comprobado comprobando la colisión de la esfera con otros objetos usando el componente *Sphere Collider* como se hace en los videotutoriales utilizados para realizar esta actividad.

A continuación, se van a describir los cambios realizados sobre el tutorial para que la esfera alcanza al cubo de esta nueva forma.

Respecto al *prefab TrainingArea*, se ha agregado un nuevo objeto:

* Void: plano inferior para comprobar si la esfera se ha caído del suelo.

El resultado final del *prefab TrainingArea* es el siguiente:

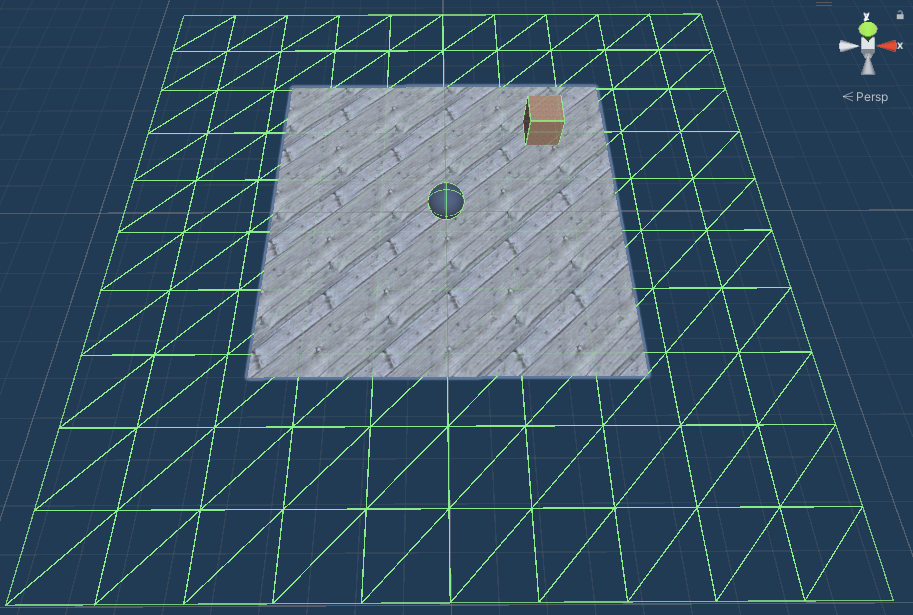


Ilustración . Nuevo prefab TrainingArea

Se ha asignado un *tag* y un *layer* a cada objeto del prefab *TrainingArea* como muestra la siguiente tabla:

|  |  |  |
| --- | --- | --- |
| Objeto | Tag | Layer |
| Floor | Default | Floor |
| Void | Void | Void |
| Target | Target | Target |
| RollerAgent | Player | Agent |

Para definir las colisiones, se ha definido en la matriz de colisiones del proyecto que layers pueden colisionar, marcando solo las definidas en el prefab como se muestra a continuación:

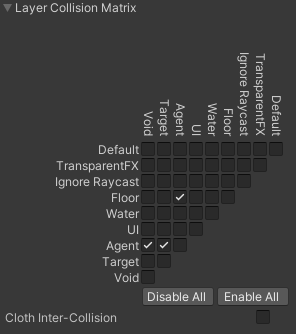
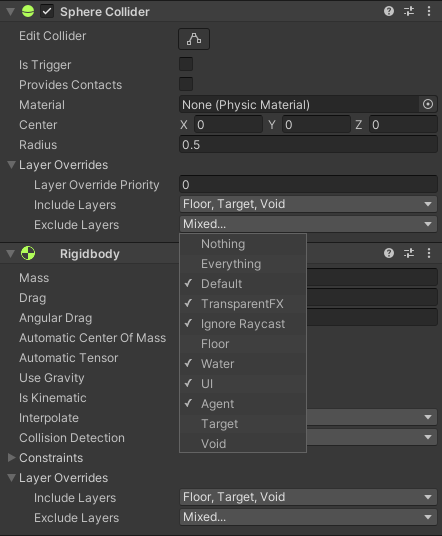


Ilustración : Matriz de colisiones

Asimismo, se ha defido también dichos layers en los componentes *Sphere Collider* y *RigidBody* del objeto *RollerAgent* como se muestra en la a continuación:



El cambio realizado en el script *RollerAgent* ha sido mover el código de la recompensa al método que se ejecuta cuando un objeto entra en colisión con otro OnCollisionEnter.

El código modificado es el siguiente*:*

*// ReSharper disable Unity.PerformanceAnalysis  
/\*  
 \* Receives actions and assigns the reward  
 \*/*public override void OnActionReceived(ActionBuffers actionBuffers)  
{  
 *// Actions, size = 2* Vector3 controlSignal = Vector3.zero;  
 controlSignal.x = actionBuffers.ContinuousActions[0];  
 controlSignal.z = actionBuffers.ContinuousActions[1];  
 \_rBody.AddForce(controlSignal \* forceMultiplier);  
}

*/\*  
 \* Triggers when the object hits another one  
 \* Set the rewards according to the object hitted  
 \*/*private void **OnCollisionEnter**(Collision collision)  
{  
 switch (collision.transform.tag)  
 {  
 *// Rewards* case Constant.Tag.**TARGET**:  
 SetReward(1.0f); *// reach the goal and get a positive reward* EndEpisode();  
 break;  
 case Constant.Tag.**VOID**: *// not reached the goal, so no reward* EndEpisode();  
 break;  
 }  
}

Se ejecuta el programa de entrenamiento de la IA cuyo identificador es *RollerBallTest02* ejecutando un total de 300.000 también.

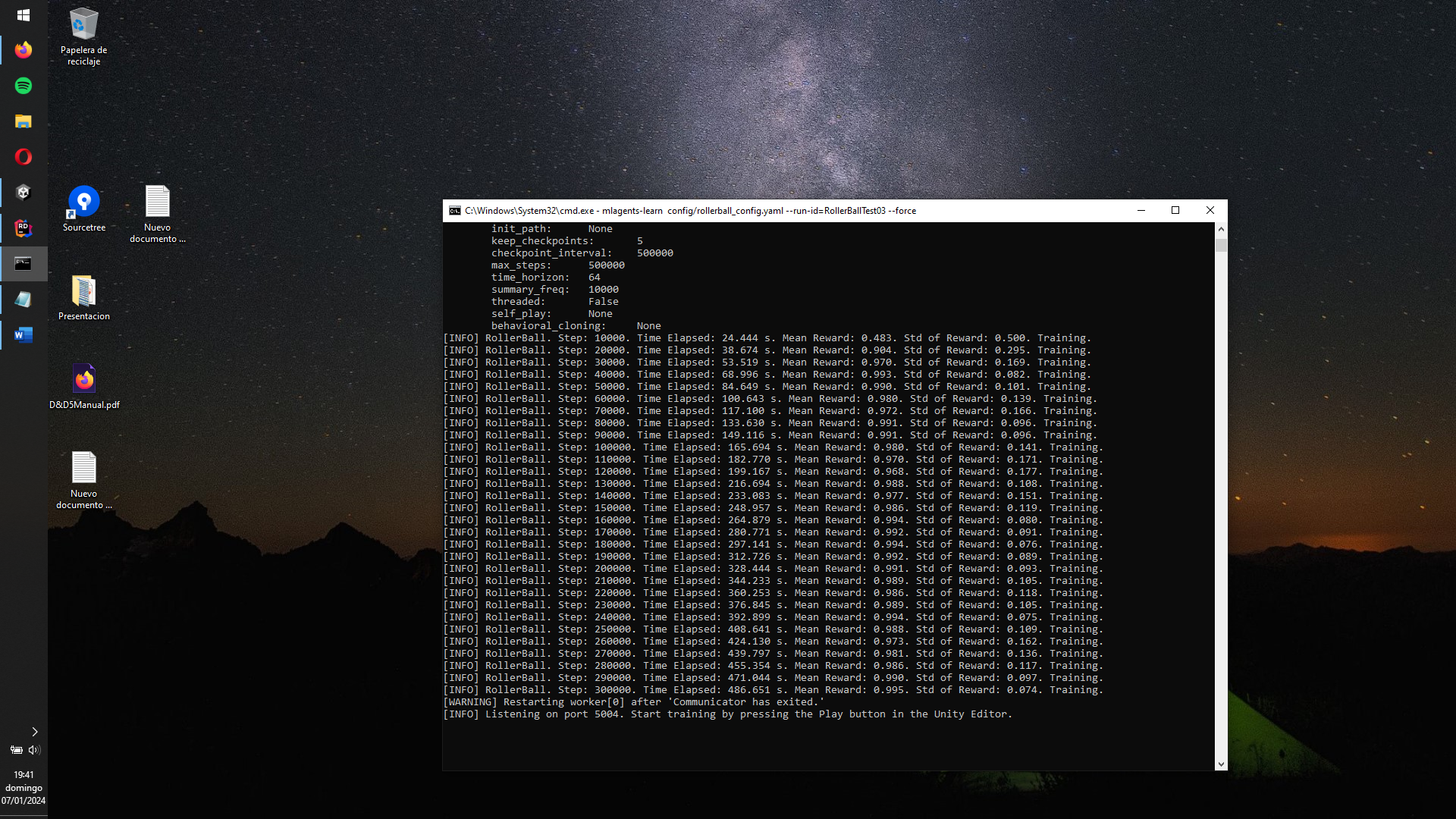


Ilustración : nuevos resultados entrenamiento IA

El resultado es el esperado y se muestra en el siguiente video: <https://youtu.be/70rF_wDgTLg>

# Actividad 2: Roller Ball tutorial

*/\*  
 \* Handles the behavior of the Dummy and sets the implementation of the agent  
 \*/*public class **DummyAgent** : Agent  
{  
 *// Fields* [Header("Movement")]  
 [Tooltip("The speed at which the npc walks")] [SerializeField] [Range(0, 1.5f)]  
 protected float **walk** = 1f;  
   
 [Tooltip("The speed at which the npc runs")] [SerializeField] [Range(0, 4)]  
 protected float **sprint** = 3f;  
   
 [Tooltip("Max speed of movement transitions")] [Range(0, 20)]  
 [SerializeField] private float **maxAcceleration** = 10f;  
  
 [Tooltip("The max distance to move")] [SerializeField] [Range(0, 15)]  
 protected float **movementDistance** = 7f;  
  
 [Tooltip("The max distance for detecting the destination point")] [SerializeField] [Range(0, 2)]  
 protected float **detectionDistance** = 0.5f;  
   
 [Tooltip("The max time until reach the destination point")] [SerializeField] [Range(1, 10)]  
 protected float **waitingTime** =5f;  
   
 [Tooltip("The max distance for detecting other scene elements")] [SerializeField] [Range(5, 20)]  
 protected float **detectionRadius** =10f;  
  
 [Tooltip("The area to search the acceptable destination points")]  
 [SerializeField] private Constant.NavMesh.AreaMask **areaMaskName** = Constant.NavMesh.AreaMask.**Walkable**;  
  
   
 *//properties  
 // the input key manager* private AgentInputHandler \_agentInputHandler;  
 *// animator* private Animator \_animator;  
 private CharacterController \_controller;  
  
 *//variables* private float \_speed;  
 private float \_movementSpeed; *// maximum speed* private float \_accelerationSpeed; *// maximum acceleration* private float \_animationBlend; *// speed value of the animation* private Vector3 \_destinationPoint; *// point to go to* private const float **RotationVelocity** = 360; *// maximum rotation angle* private int \_areaMask; *// walkable layer area number* private int \_maxSpaceSize; *// maximum elements to save while training the IA* private float \_reachTime; *// maximum time for reaching the point  
   
   
 // Start is called before the first frame update* private void **Awake**()  
 {  
 *// sets an instance of the components* \_agentInputHandler = GetComponent<AgentInputHandler>();  
 \_animator = GetComponent<Animator>();  
 \_controller = GetComponent<CharacterController>();  
 \_controller.detectCollisions = true;  
 *// get maximum character speed between the walk and the sprint value* \_movementSpeed = Random.Range(walk, sprint);  
 *// get maximum character acceleration between half the acceleration and the acceleration value* \_accelerationSpeed = Random.Range(maxAcceleration / 2, maxAcceleration);  
 *// get the mask to get the acceptable points* \_areaMask = 1 << NavMesh.GetAreaFromName(areaMaskName.ToString());  
 *// the maximum size of the vector observation* \_maxSpaceSize = GetComponent<BehaviorParameters>().BrainParameters.VectorObservationSize;  
  
 }  
  
 */\*  
 \* Reset the agent position  
 \*/* public override void OnEpisodeBegin()  
 {  
 *// max speed between the walk and the sprint value* \_movementSpeed = Random.Range(walk, sprint);  
 *// max acceleration between half the acceleration and the acceleration value* \_accelerationSpeed = Random.Range(maxAcceleration / 2, maxAcceleration);  
 \_animationBlend = 0;  
 \_speed = 0;  
 *// loop until find a random point in the walkable area* while (!RandomPoint(transform.position, movementDistance, out \_destinationPoint))  
 {  
 }  
 *// set the beginning reach time* \_reachTime = Time.time;  
  
 }  
  
 */\*  
 \* The information we collect for the training of the IA  
 \*/* public override void CollectObservations(VectorSensor sensor)  
 {  
 *// Destination and Agent positions* sensor.AddObservation(\_destinationPoint);  
 sensor.AddObservation(transform.localPosition);  
  
 *// Agent velocity* sensor.AddObservation(\_controller.velocity.x);  
 sensor.AddObservation(\_controller.velocity.z);  
   
 *// other object to avoid: another citizen or a non walkable area* var colliders = Physics.OverlapSphere(transform.position, detectionRadius);  
 *// get the maximum elements to save at the vector observation* int max = \_maxSpaceSize-4;  
 if (max > colliders.Length)  
 {  
 max = colliders.Length;  
 }  
  
 for (int i = 0; i < max; i++)  
 {  
 Collider colliderElement = colliders[i];  
 switch (colliderElement.transform.tag)  
 {  
 case Constant.Tag.**NO\_WALKABLE**: *// save only the position for still objects like trees or stones* sensor.AddObservation(colliderElement.transform.position);  
 break;  
 case Constant.Tag.**CITIZEN**: *// save the position and velocity for moving objects, the citizens* sensor.AddObservation(colliderElement.transform.position);  
 sensor.AddObservation(colliderElement.GetComponent<NavMeshAgent>().velocity.x);  
 sensor.AddObservation(colliderElement.GetComponent<NavMeshAgent>().velocity.z);  
 break;  
 }  
 }  
  
 }  
  
 *// ReSharper disable Unity.PerformanceAnalysis  
 /\*  
 \* Receives actions and assigns the reward  
 \*/* public override void OnActionReceived(ActionBuffers actionBuffers)  
 {  
 *// Actions, size = 2* Vector3 controlSignal = Vector3.zero;  
 controlSignal.x = actionBuffers.ContinuousActions[0];  
 controlSignal.z = actionBuffers.ContinuousActions[1];  
 Movement(controlSignal); *// move and animate the dummy* if (CheckDistanceToDestination()) *// if the dummy reaches the point* {  
 SetReward(1.0f); *// reach the goal and get a positive reward* EndEpisode(); *// start again* }  
  
 if ((waitingTime + \_reachTime) < Time.time) *// if the dummy is stuck and no reached the point in a period of time* {  
 EndEpisode(); *// not reached the goal, so no reward* }  
 }  
  
 *// ReSharper disable Unity.PerformanceAnalysis  
 /\*  
 \* Control the agent ant the heuristic behavior  
 \*/* public override void Heuristic(in ActionBuffers actionsOut)  
 {  
 Vector2 movement = \_agentInputHandler.GetMoveInput();  
 var continuousActionsOut = actionsOut.ContinuousActions;  
 continuousActionsOut[0] = movement.x;  
 continuousActionsOut[1] = movement.y;  
 }  
  
 */\*  
 \* Triggers when the object hits another citizen, bounds or a non walkable area  
 \* Set the rewards according to the object hitted  
 \*/* public void **OnControllerColliderHit**(ControllerColliderHit hit)  
 {  
 switch (hit.transform.tag) *// the dummy walks over or hit an avoidable object: no walkable area, a citizen* { *// or walks away from the park  
 // Rewards* case Constant.Tag.**NO\_WALKABLE**:  
 case Constant.Tag.**CITIZEN**:  
 case Constant.Tag.**VOID**:  
 EndEpisode(); *// not reached the goal, so no reward* break;  
 }  
 }  
  
  
 *//movement* private void Movement(Vector3 move)  
 {  
 if (move.Equals(Vector3.zero)) *// the dummy is still, reset the speed* {  
 \_animationBlend = 0;  
 \_speed = 0;  
 }  
 else  
 { *// is moving* Vector3 inputDirection = move.normalized;  
 \_speed = Mathf.Lerp(\_speed, \_movementSpeed,  
 \_accelerationSpeed \* Time.fixedTime);  
 *// move and rotate the dummy in the movement direction* Quaternion toRotation = Quaternion.LookRotation(inputDirection, Vector3.up);  
 transform.rotation = Quaternion.RotateTowards(transform.rotation, toRotation, **RotationVelocity** \* Time.deltaTime);   
 \_controller.Move(inputDirection \* (\_speed \* Time.deltaTime));  
 transform.position = new Vector3(transform.position.x, 0.5f, transform.position.z); *// reset the vertical position* }  
   
 *//animation* \_animationBlend = Mathf.Lerp(\_animationBlend, \_speed,Time.fixedTime \* \_accelerationSpeed);  
 \_animator.SetFloat(Constant.Animation.**SPEED**, \_animationBlend);  
   
 }  
   
 *// Find a random point in the walkable area* private bool RandomPoint(Vector3 center, float range, out Vector3 result)  
 {  
 *// get a random 3D point and extends it with a certain range* Vector3 randomPoint = center + Random.insideUnitSphere \* range;  
 *// checks if the random 3D point is in the walkable area* if (NavMesh.SamplePosition(new Vector3(randomPoint.x, 0.25f, randomPoint.z), out var hit, 0.5f, \_areaMask))  
 {  
 result = hit.position; *// if found, returns the random 3D point* return true;  
 }  
 *// returns vector zero if not* result = Vector3.zero;  
 return false;  
 }  
   
 */\*  
 \* Checks if the the citizen position is close enough to the destination point   
 \*/* private bool CheckDistanceToDestination()  
 {  
 return detectionDistance >= Vector3.Distance(transform.position, \_destinationPoint);   
 }  
   
}

venv\Scripts\activate

mlagents-learn config/dummy\_config.yaml --run-id=DummyTest01 --force

# Bibliografía

## Actividad 1

### Instalación

* <https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Installation.md>
* <https://www.youtube.com/watch?v=zPFU30tbyKs&list=PLzDRvYVwl53vehwiN_odYJkPBzcqFw110>
* <https://www.youtube.com/watch?v=RANRz9oyzko>

### Realización

* <https://unity-technologies.github.io/ml-agents/Getting-Started/>
* <https://github.com/Unity-Technologies/ml-agents/blob/release_10/docs/Training-ML-Agents.md>
* <https://unity-technologies.github.io/ml-agents/Learning-Environment-Create-New/>
* <https://assetstore.unity.com/packages/2d/textures-materials/pbr-materials-sampler-pack-40112>
* <https://unity.com/es/features/input-system>

## Actividad 2

### Realización

* <https://www.youtube.com/watch?v=BJzYGsMcy8Q>
* <https://www.youtube.com/watch?v=UJsaEVPntMg>