Artificial Intelligence for Videogames project report

Author:

Diego VALLAURI

Contents

1 Introduction and purpose of the project

The project started with reference to the **Pong** game, where there are two players whose aim is to hit the disk without being overtaken by it. Starting from this, I decided the purpose of the project, which is to create an agent who can take the place of a player. In short, my goal was to create an agent who would hit the disk every time without missing it. To solve the problem, the first idea I proposed was a trigonometric solution. Since it wasn't that easy to put into practice, the Teacher suggested to use **Machine Learning** instead. This is the solution I adopted to the project and in particular I relied on **TensorFlow** which acts as an intermediary between Unity and Machine learning itself.

At the end of the project, I created three agents. The first moves only left and right (as in the **Pong** game), the second agent and the third in each four directions. Since the third was probably not required for the project, I consider it as an experiment I did towards the end.

2 Creation of the environment

Since the aesthetic part was not part of the project, I only used some Cubes and Cylinders to create the environment, as can be seen in the image below of the agent and the disk.

I have created two types of fields. One for training and one with two agents. I will explain the use of these fields in the next section.
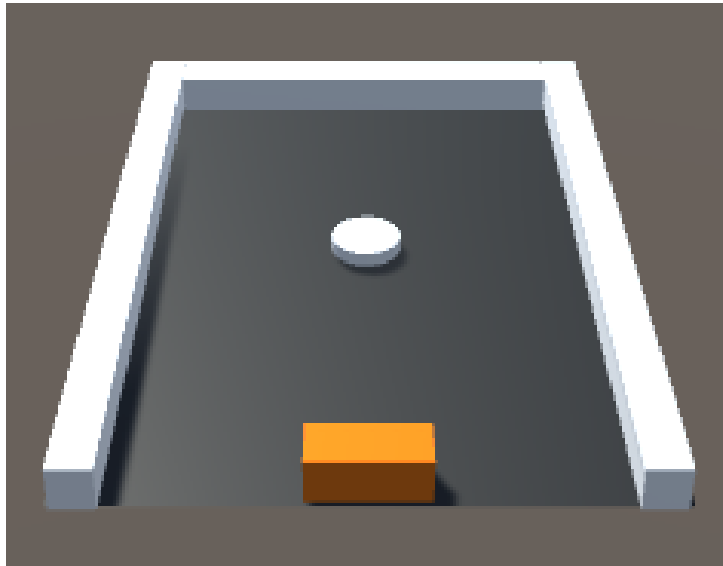

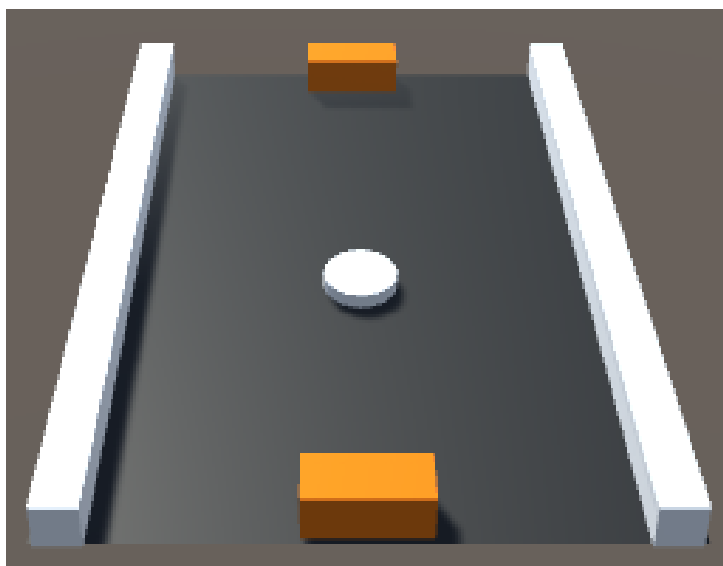Figure 1: This is the field for training with just one side open.

Figure 2: This is the normal field with two open sides.

3 First agent

The folder that describes the agent and the environment of this first example is located in **"Asset/FirstTest"**.

3.1 Environment set up.

Before starting to train the agent, I obviously need to set my environment. For the training I will use the *training field* (Figure 1). The first thing to decide is how to bounce the disk off the walls. Since every wall has a *Collider* and the disk a *RigidBody*, I can let the physics engine do it for me. Obviously I have to help it and have to make some considerations. For example, I don't want friction between the plane and the disk and between the disk and the wall. I also don't want the disk to stop moving after colliding with multiple walls. From this base, I created two different physic materials, each without friction but one with a bounciness equal to 1 and one equal to 0 (I don't want the disk to bounce on the plane). I assigned the first material to the side walls and the second to the floor.
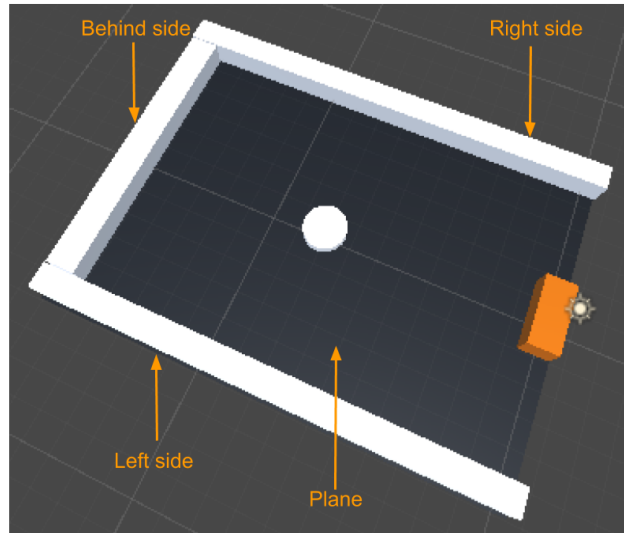

Figure 3: Representation of side walls and the plane.
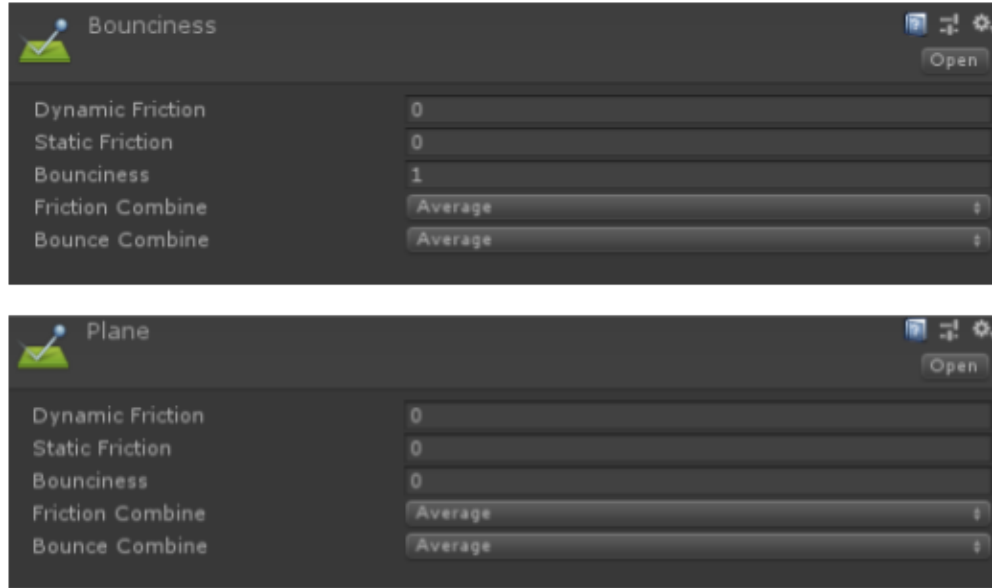
Figure 4: Physic material of the walls and the plane.

Now there is the "hardest" part: how do I make the disk bounce with the agent in a real way? First of all, I can't let this work to the physic engine because the disk will always behave the same way. I mean, if the ball goes straight through the middle to the agent, the ball will continue to change its Z coordinate but the X would remain the same (or at least, very similar. See figure 5) and in a nutshell its movement will remain the same. My solution is to look for the new destination and then add a force in that direction. For this, I used the Vector3.reflect function, which takes two parameters: in-direction and normal. In short I will calculate a new position that reflects the direction of the ball on the normal of the impact.

Figure 5: The only possible movements if the disk start straight to the agent.
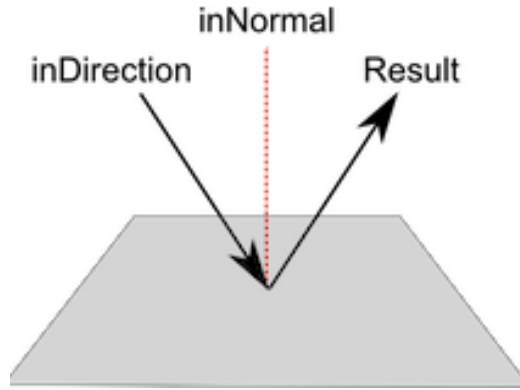
Figure 6: In summary, this is what the **Vector3.reflect** function returns.

So after find the new destination, I will apply a force based on new vector normalized and the speed of the disk. The code for this part is on the next page. As you will see, there is also an "OnTriggerEnter", whose sole purpose is to set the **haveILose** variable to true if the disk passes the agent (there is a game object in the scene called "onTriggerGoal" with the check mark on the "isTrigger" in the collider). The **haveILose** will be used on the agent for cheacking if the agent has lost and restart the episode, or not.

```
public class DiskBehaviour : MonoBehaviour{

    public float speed;

    private Rigidbody rg;
    private Vector3 destination;
    public bool haveILose = false;

    void Start(){
        rg = GetComponent<Rigidbody>();
    }

    private void OnCollisionEnter(Collision collision) {

        if (collision.gameObject.tag == "Agent") {

            ContactPoint contact = collision.contacts[0];

            rg.velocity = new Vector3(0f,0f,0f);
            Vector3 destination = Vector3.Reflect(transform.localPosition,
                                       collision.contacts[0].normal);
            rg.AddForce(destination.normalized * speed,
                        ForceMode.VelocityChange);

        }
    }

    private void OnTriggerEnter(Collider other) {
        if (other.gameObject.tag == "Goal")
            haveILose = true;
    }
}
```

Figure 7: Disk behaviour class.

One last thing I forgot to mention, is that this behaviour is attached to the
agent but also to the behind wall (figure 3), since in training it simulates a
real agent.

3.2   Training set up

Now I am ready to create my agent. As suggested, I create an empty game
object as parent of my parallelepiped. Since i wanted to use a **RigidBody**
for the physics and wanted to keep the agent part and the movement di-

vided, I simply assigned everything about the agent to the parent and the **RigidBody** and the **Collider** to the child.

- **Agent**: Has attached the **Behavior Parameters**, the **Decision Requester** and the **BAgent** (my script).

- **Player**: It's the son of **Agent** and has attached the **Box Collider**, the **RigidBody** and everything needed for rendering.

Let's now analyse the **BAgent** script. Since it extends the Agent class it has at least four methods: **Initialize**, **OnEpisodeBegin**, **CollectObservations**, **OnActionReceived**.

- **Initialize**: it has nothing special about it, just the initialization of the disk **RigidBody** and the agent child reference (that will actually move).

```
public class BAgent : Agent{
    [Header("Agent parameter")]

    public GameObject hockeyDisk;

    public float agentSpeed;

    private Rigidbody diskRB;
    private GameObject child;

    public override void Initialize() {
        diskRB = hockeyDisk.GetComponent<Rigidbody>();
        child = transform.GetChild(0).gameObject;
    }
}
```

- **OnEpisodeBegin**: I first take the value of **haveILose** and set it to false, since the true value is the trigger event for the end of the episode. So I create a different position for the agent, for the disk and calculate a new starting direction for the disk. At the end I add a force in the calculated direction.

```
public override void OnEpisodeBegin() {
    hockeyDisk.GetComponent<DiskBehaviour>().haveILose = false;
    hockeyDisk.transform.localPosition = createRandomPosition(false,false);
    float speed = hockeyDisk.GetComponent<DiskBehaviour>().speed;
    child.transform.localPosition = createRandomPosition(false, true);
    Vector3 pos = createRandomPosition(true,false);
    diskRB.velocity = new Vector3(0f, 0f, 0f);
    diskRB.AddForce(pos.normalized * speed,
                ForceMode.VelocityChange);
}
```

The **createRandomPosition** method, is only an auxiliary method
to create the position of the elements based on the parameters. (It's
not that interesting so I don't put any photos there, but it's obviously
present in the folder).

- **CollectObservations**: this is quite important and contains seven ob-
  servations. The first three are the component of the local position of
  the disk, The fourth is the x local position of the agent and the last
  three are the components of the RigidBody velocity.
  In this part I made some mistake at the beginning, for example I gave
  the global and not the local position of the elements and this create
  problems during the training because every prefabs had it's own global
  position and so the learning didn't go very well. Another mistake (not
  really a mistake) was providing the agent y and z components as well
  (which are fixed) and this only slowed down the training.

```
public override void CollectObservations(VectorSensor sensor) {
    sensor.AddObservation(hockeyDisk.transform.localPosition);
    sensor.AddObservation(child.transform.localPosition.x);
    sensor.AddObservation(diskRB.velocity);
}
```

- **OnActionReceived**: the agent will receive only a float (clamped
  beetween -1 and 1) and will first reset the RigidBody velocity of the
  agent and then add a force in the direction received before.
  Moreover the agent will receive a positive reward of 0.05 if the disk has
  not overcome it or a negative reward of -5 in the opposite case.

```csharp
public override void OnActionReceived(float[] vectorAction) {
    Vector3 destination = new Vector3(Mathf.Clamp(vectorAction[0], -1f, 1f),
        child.transform.localPosition.y, child.transform.localPosition.z);

    child.GetComponent<Rigidbody>().velocity = new Vector3(0f,0f,0f);
    child.GetComponent<Rigidbody>().AddForce(destination * agentSpeed,
            ForceMode.VelocityChange);

    if (hockeyDisk.GetComponent<DiskBehaviour>().haveILose == false) {
        SetReward(0.05f);
    } else {
        SetReward(-5f);
        EndEpisode();
    }
}
```

3.3 The training

For the learning i will use the scene **Test**.

Figure 8: Scene **Test** with 96 field prefabs.

For the .yaml file, I followed the instruction on the doc site. In particular in this first example the .yaml file is composed in that way:

As can be seen from above, the learning lasted 1000000 steps.

Learning works very well and the agent is able to hit the ball every time (see the video "**FirstTestTraining**" in the folder).

After the first training session, I panicked when I tried to play against another agent. In this case, the one in the opposite position of the training could not catch the ball. At first I thought it was my mistake, but after a few tries, I realized that the problem was only in the reference system. If we take for example the output "1" of the neural network for the normal agent it indicates to go to the left, while for the "inverse" agent it indicates to go right. I suppose that was the problem, but maybe it's something I don't catch.

So, to create the second agent, I did the same training as before with the agent in the opposite position.

```yaml
behaviors:
  Field:
    trainer_type: ppo
    hyperparameters:
      batch_size: 64
      buffer_size: 12000
      learning_rate: 0.0002
      beta: 0.001
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 128
      num_layers: 2
      vis_encode_type: simple
      memory: null
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    init_path: null
    keep_checkpoints: 5
    checkpoint_interval: 1000000
    max_steps: 1000000
    time_horizon: 1000
    summary_freq: 100000
    threaded: true
```

Figure 9: Behaviour.yaml

Now after assign the two brains to the different agents, they could play together without any problems. (see "**AgentVSAgentTest1**" video). Since I wanted to create only one brain for the two agents, I decide to do a mixed training with half normal field and half inverse field. The results is pretty good and its behaviour it's very similar to the initial brain (see "**AgentVSAgentSameBrain**" video). Obviously the learning rate it's a
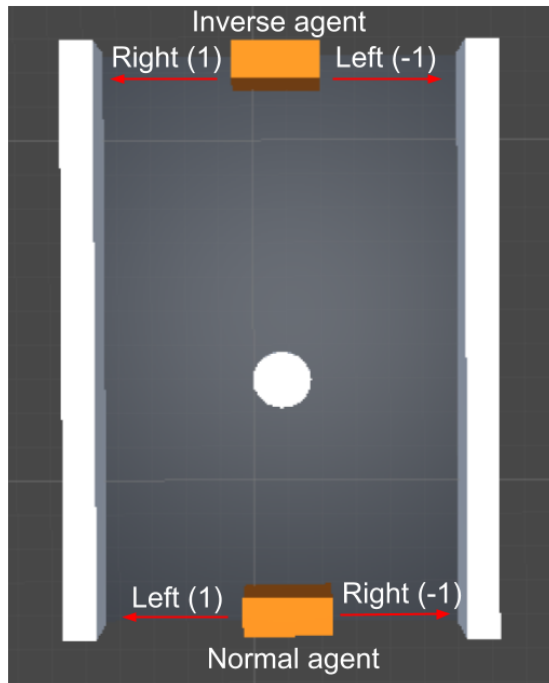
Figure 10: Different reference system for the two agents.

little bit slower and it's not as accurate as the normal training, but it's ok.

Figure 11: Scene **InverseTest** with 60 field prefabs.

3.3.1   Conclusion

In conclusion, the first experiment went very well and I created an agent that is able to hit the disk every time without any particular problems.
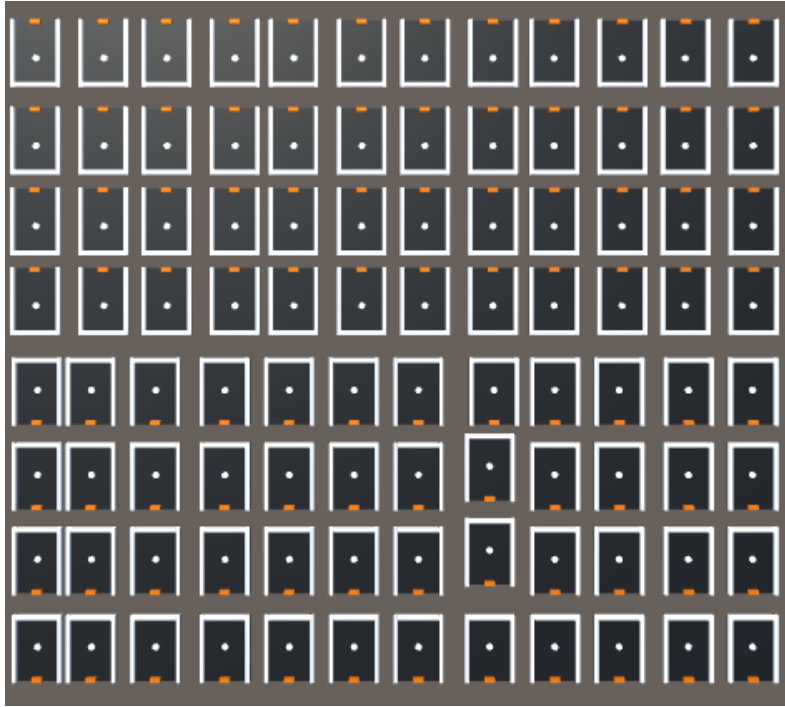
Figure 12: Scene **MixedTest** with 48 normal field prefabs and 48 inverse field prefabs.

4 Second agent

The next stage is to create an agent that can move in every four directions. The folder that describes the agent and the environment of this second example is located in **"Asset/SecondTest"**.

4.1 Environment set up.

Referring to the "**Normal**" scene, there are some small changes. There is a new empty game object called "**Middle Field**" with only a box collider to use as a limit for the movement of the player in the center of the field. There is actually a duplicate of the component used to restrict player movement at the bottom of the pitch.

Also, there is a script attached to the game object whose sole purpose is to ignore the collision for the disk. If I didn't, the disk would stop shortly after
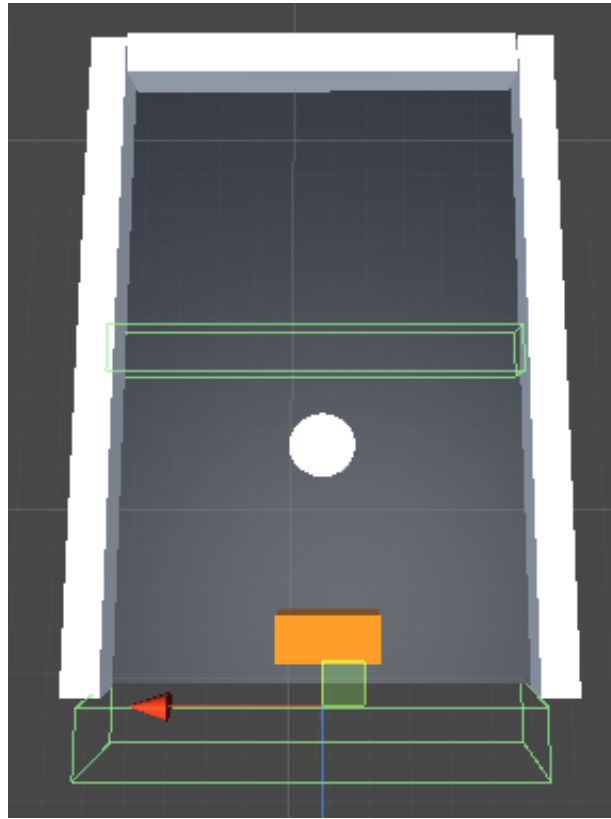
Figure 13: Gizmos of the box collider of the two new "**Middle Field**" game objects.

the episode started.

```
public class MiddleCollision : MonoBehaviour{

    public GameObject disk;

    // Start is called before the first frame update
    void Start()
    {
        Physics.IgnoreCollision(disk.GetComponent<SphereCollider>(), this.gameObject.GetComponent<BoxCollider>());
    }

}
```

Figure 14: MiddleField's script.

The second change is in "**DiskBehaviour2**". The script is very similar to

the one in the first test folder, but when the disk collides with the agent, the behind wall is randomly placed in his half court. I think it can help the agent learn better as his opponent may have different z coordinates and not sit still in the same position.
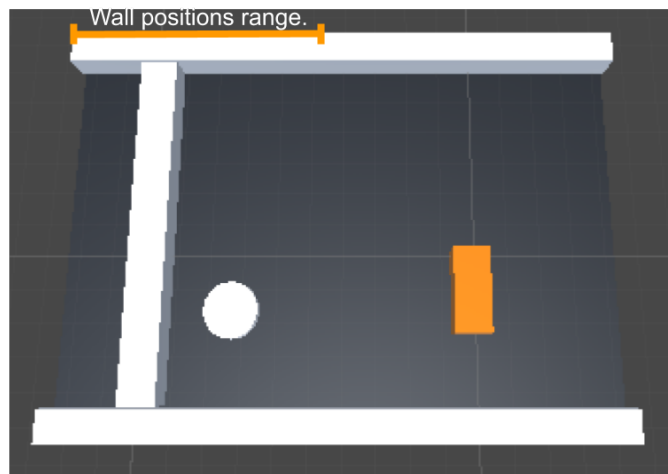

Figure 15: Representation of the different position of the wall.

The only thing that changes in the script is in the **OnCollisionEnter** method. It is just an if that checks the position of the wall and adds a random position (the if is needed to also work in the "inverse field" for the training of the second agent).

```
if (collision.gameObject.name == "Player" && behind != null) {
    if (behind.transform.localPosition.z > 0)
        behind.transform.localPosition = new Vector3(behind.transform.localPosition.x, behind.transform.localPosition.y, Random.Range(1.5f, 4.7f));
    else
        behind.transform.localPosition = new Vector3(behind.transform.localPosition.x, behind.transform.localPosition.y, Random.Range(-4.7f, -1.5f));
}
```

Figure 16: Code added in the **OnCollisionEnter** method.

4.2 Training set up

Also in the training set up there are some small changes in the different method of the agent (script: **BAgent2**).

- **Initialize**: same as before.

- **OnEpisodeBegin**: As new there is the "if" to change the position of the behind wall. The "if" is done only if we are using a field with a behind wall (if there is a second agent,I don't need to assign it a random position at the start of every episode), in the other case the "**training**" variable is set to false.

```
public override void OnEpisodeBegin() {
    if (training) {
        if(behindZ > 0)
            opponent.transform.localPosition = new Vector3(opponent.transform.localPosition.x, opponent.transform.localPosition.y, Random.Range(1.5f,4.7f));
        else
            opponent.transform.localPosition = new Vector3(opponent.transform.localPosition.x, opponent.transform.localPosition.y, Random.Range(-4.7f, -1.5f));
    }
    hockeyDisk.GetComponent<DiskBehaviour2>().haveILose = false;
    hockeyDisk.transform.localPosition = createRandomPosition(false,false);
    float speed = hockeyDisk.GetComponent<DiskBehaviour2>().speed;
    Vector3 pos = createRandomPosition(true,false);
    child.transform.localPosition = createRandomPosition(false, true);
    diskRB.velocity = new Vector3(0f, 0f, 0f);
    diskRB.AddForce(Vector3.forward * speed,
                    ForceMode.VelocityChange);

}
```

- **CollectObservations**: in addition to the seven observations I sent in the first example, I also send the z coordinate of the agent's local position. I tried to give the z coordinate of the behind wall local position as an observation, but it was irrelevant to the neural network output, so I took it out.

```
public override void CollectObservations(VectorSensor sensor) {
    sensor.AddObservation(hockeyDisk.transform.localPosition);
    sensor.AddObservation(child.transform.localPosition.x);
    sensor.AddObservation(child.transform.localPosition.z);
    sensor.AddObservation(diskRB.velocity);
}
```

- **OnActionReceived**: in addition to the float for the x coordinate, the agent will receive a float (always clamped beetween -1 and 1) for the z coordinate.

19

```
public override void OnActionReceived(float[] vectorAction) {
    Vector3 destination = new Vector3(Mathf.Clamp(vectorAction[0], -1f, 1f),
        child.transform.localPosition.y, Mathf.Clamp(vectorAction[1], -1f, 1f));

    child.GetComponent<Rigidbody>().velocity = new Vector3(0f,0f,0f);
    child.GetComponent<Rigidbody>().AddForce(destination * agentSpeed,
            ForceMode.VelocityChange);

    if (hockeyDisk.GetComponent<DiskBehaviour2>().haveILose == false) {
        SetReward(0.05f);
    } else {
        SetReward(-5f);
        EndEpisode();
    }
}
```

4.3   The training

The .yaml file used for the training is "**configuration.yaml**", which is the same file used before but the learning lasts 2000000 steps. That's probably a little too much, but since the training is a bit slower I preferred it that way. As before I created three different agent, the first is the one from the "**Test**" scene and it's the brain of the agent in the normal position. The second is the one in the inverse position and the third is the one with mixed training. Look at the different video of the agent in the folder.

4.4   Conclusion

Also the second example went very well and the agent was able to hit the ball moving in all of the different four direction.

5   Last agent

I consider this last agent as an experiment and it's similar to the second example. The folder that describes the agent and the environment of this third example is located in "**Asset/ThirdTest**".

5.1 Environment set up.

Different from before is that the pitch is a bit bigger and there is a goal to score at.
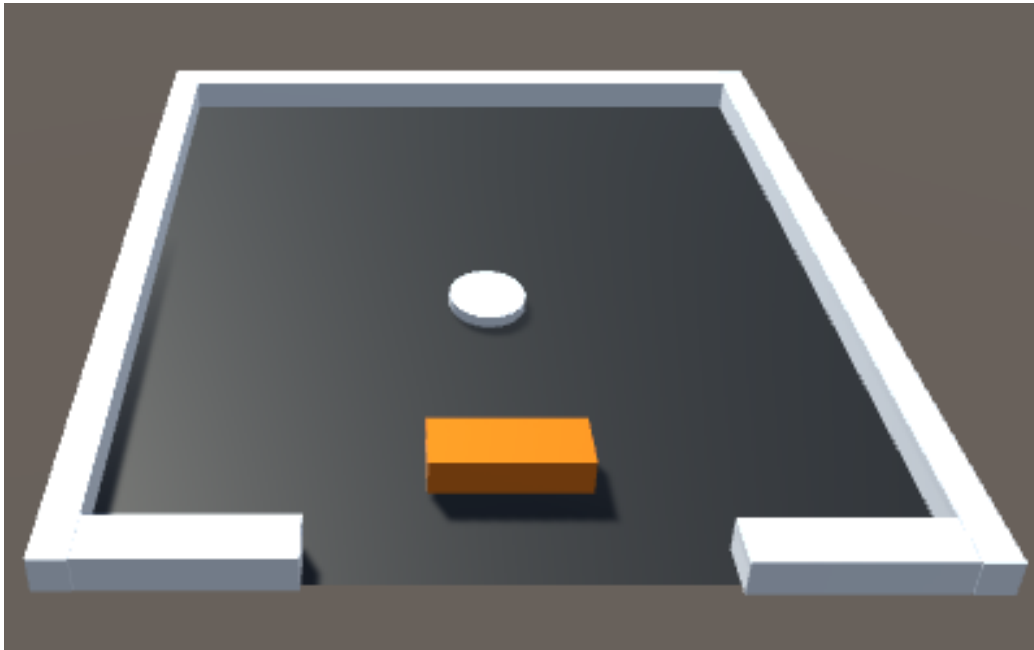
The **DiskBehaviour3** script is the same as before, the only difference is a check in the update method if the magnitude of the RigidBody velocity is less than 5. If so, it adds a force to the disk. This check is necessary in case the agent attempts to stop the disk in one of the lower side walls (and yes, it will try).

```
void Update() {
    if (rg.velocity.magnitude < 5)
        rg.AddForce((rg.velocity.normalized + new Vector3(0.1f,0.1f,0.1f)) * speed);
}
```

5.2 Training set up

There are no change from the second example.

5.3 Training

I have called this third example as an experiment, because when i tried the learning i was thinking that everything would be ok. Instead the training had some problems. Since the agent have a positive reward if the disk don't reach the goal, during the training it will try to stop the disk in in one of the lower side walls in the two following way:
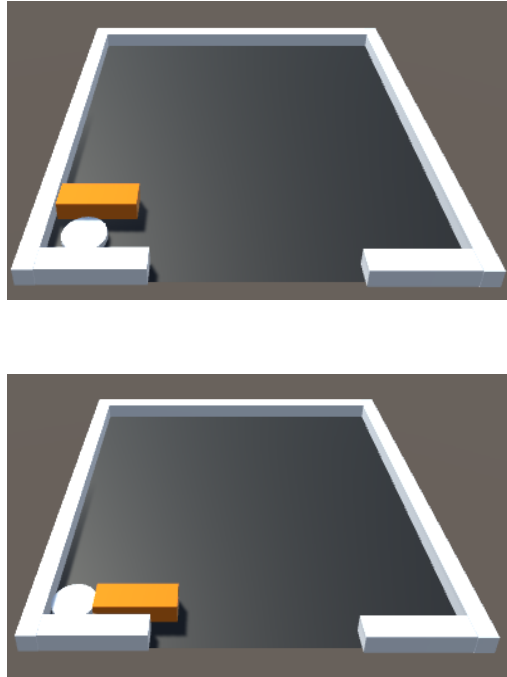
There is also a second problem. In some training sessions the agent stays close to the centre of the pitch in a corner. This is because the disk has less change to go into the goal, which is smaller than before, so agents get a positive learning reward even if it doesn't try to hit the ball. After trying the training several times, I got a decent result, but it is not comparable to the one obtained before. I left some videos in the folder and as you can see the agents try more to stop the ball than to defend the goal. I didn't have time to find a good solution, I tried something (very quickly). For example I thought of giving a negative reward not only when the disk enters the goal, but also if it passes the agent. This way the agent would learn to stay behind the disk and attempt to hit it. I try a training like this but it didn't go very well and the agent always tries to stop the disk in the side wall. I probably

didn't set up the training in the right way.

Another solution I thought of was to give a negative reward (and end the episode) if the speed of the disk drops below a certain range, to avoid stopping it.

Since I didn't have much time, I don't know if they would have been good solutions, but they could have been.

5.4   Conclusion

From this last example, I have learned that a simple problem can mutate in a difficult one just adding some small changes. My training was acceptable but sure it can be obtained more with more complex solution.

6   Conclusion

From this project I learned how to use Machine Learning in Unity with a simple environment. I noticed that it's not always that easy, as in the last test or in some example in the Ml-agents doc. However the set up is very quick and easy, and even if you don't know much about it, you can easily learn how to use it. Continuing in this direction, I'm pretty sure this is just the beginning of Machine Learning in video games.