IAJ Project 4 - ML Agents

This report has been made in accordance to the guidelines provided for the IAJ's (Inteligência Artificial para Jogos) fourth project on the topic of Unity's MLAgents module.

Authors

- Diogo Silva (98776)

Index

- Concept
- Game
    - Gameplay
    - Ship
    - Enemies
    - Waves
    - GameManager Settings
- ML Agents
    - Methodology
    - Tensorboard & Results
    - First Scenario - Just Shoot
    - Second Scenario - ShootNRotate
        - Single Static Asteroid
        - Inbound Asteroids
    - Third Scenario - MoveNShoot
        - Inbound Asteroids w/ Movement
        - Single Non-Splitting Asteroid
        - Single Splitting Asteroid
- Curriculum Learning
- Final Model
- Conclusion
- References & Assets

Concept

The concept for this final project was the application of Unity's ML-Agents framework to the classic

game - **Asteroids**. As such, the elaboration of this work was done in 3 phases:

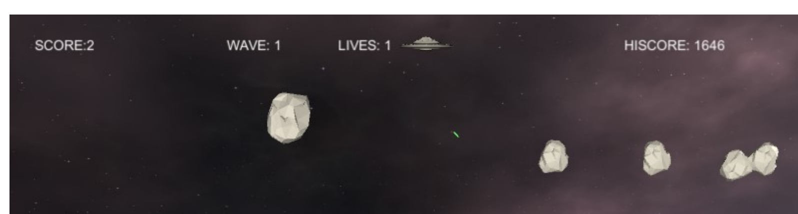
- **Create the Game**
  - First, the game had to be built from scratch and had to emulate the original game's mechanics as best as possible.
- **Adapt the Game**
  - Afterwards, the game had to be adapted in order to allow for the integration with the ML-Agents framework.
- **Train the Agent**
  - Finally, the kernel of this project. Several scenarios were used to train the agent in incremental stages of difficulty.

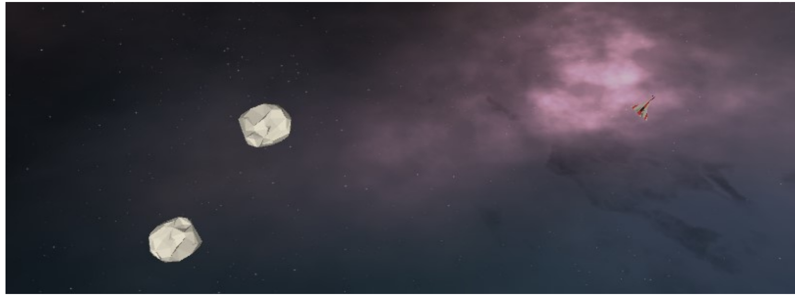
Game

Gameplay

The game is basically a clone of the classic 1979 game **Asteroids** developed by Atari. The player controls a ship, being able to **shoot**, **accelerate** forward, **deaccelerate** and **turn left** or **right**. The aim of the game is to survive against an onslaught of **asteroids** that, when shot, depending on their size, may split, creating even more hazards. Alongside the asteroids, after some time a new type of enemies start spawning, dubbed **UFOs** that shoot at the player. There's no real "*win state*", as the objective of the game is to try and go for the highest score possible (as is usually the case with these types of games). A specific property of the game is that the world "warps" around, meaning, if the ship flies off the left of the game-space, it should reappear on the right. This mechanic was implemented in our game using a technique called "*Euclidean Torus*".

The game was built from the ground up in the Unity Engine, trying to emulate the original game's properties as closely as possible.





Ship

The **ship** works exactly as expected. The player can press **w** to accelerate forward or **s** to deaccelerate, noting that there is a maximum velocity after which point the ship will not accelerate further. This movement is done through the application of forces to the ship rather than kinematically (which prooved to be an extra layer of complexity for the ML Agent to learn). If the player stops accelerating, or deaccelerating, the ship will carry on the momentum with the velocity dropping over a certain period of time (until it stops moving).

The player can also turn the ship left or right by pressing **a** or **d** respectively. This is also done through the application of a force (i.e using angular velocity), but this velocity drops so fast after the player stops pressing that the momentum is barely noticeable. This was done on porpuse, as even on the original game, whilst moving is supposed to feel like it has some weight to it, turning should be able to be done fast and snappy.

Finally the player can **shoot** a bullet (by pressing the **Spacebar**) that travels in a straight line, at a certain velocity, disappearing after some time (or if it collides against an enemy).

Enemies

There are basically two types of enemies in the game - UFOs and Asteroids.

Asteroids are the main and most common enemy. They spawn in a random location of the game space (granted they can't spawn too close to the ship's position), with a random rotation and random direction of movement (moving in a straight line without velocity decrease) and random velocity. There are three types of asteroids that differ both in size and number/type of asteroids they spawn upon death.

- **Big Asteroids**
    - Split into 2 *Medium Asteroids* upon death
- **Medium Asteroids**
    - Split into 3 *Small Asteroids* upon death
- **Small Asteroids**
    - Don't split into anything.

UFOs are rarer enemies that only move horizontally, going either left to right or right to left. Whilst being easy to track, they have the capability of shooting at the player in random intervals of time.

Waves

Rather than having random enemies spawn at random intervals of time, the game was implemented in a "wave" based manner. After each wave is completed (i.e all enemies of that wave are killed), a new wave begins. From waves 1 through 15 we incrementally spawn a number of **Big Asteroids** equal to the number of the wave. After wave 5 we additionally spawn one **UFO** per round.

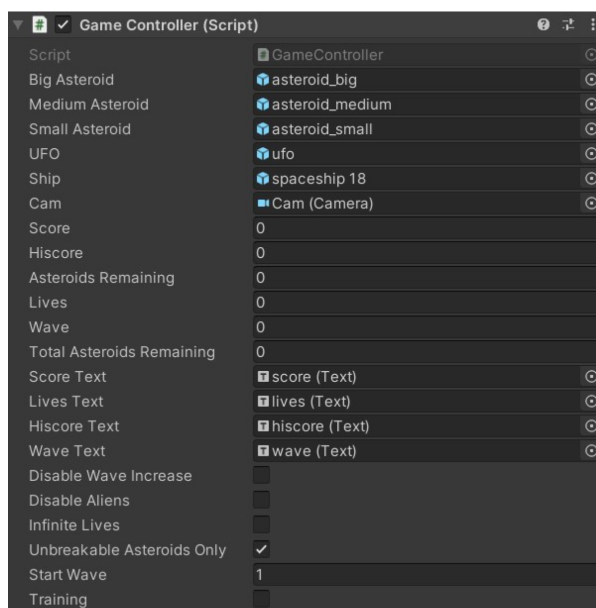
After wave 15 we spawn 15 Big Asteroids plus a random number (1 - 5) of random enemies (Small, Medium or Big Asteroids or UFOs).

If the ship dies (by coming into contact with one of the enemies or enemy bullets), the game resets back to wave 1. Effectively (by default) the player has only "1 life".

For debug purposes the wave can be increased or decreased using the numpad's **+** and **-** keys, respectively.

GameManager Settings

The GameManager posesses an assortment of settings, most of them having been created to help set the training scenarios or other debug utilities.



- **Disable Wave Increase**
  - Makes it so the number of enemies spawned doesn't increase per wave
- **Disable Aliens**
  - Makes it so UFOs don't spawn
- **Infinite Lives**
  - Makes it so dying doesn't reset the game back to wave 1.
- **Unbreakable Asteroids Only**
  - Makes it so asteroids don't split into smaller asteroids

- **Start Wave**
  - Sets the initial start wave.
- **Training**
  - Used for curriculum training. Spawns enemies according to the *Academy*'s *Environment Settings* rather than the preset ones.

ML Agents

Methodology

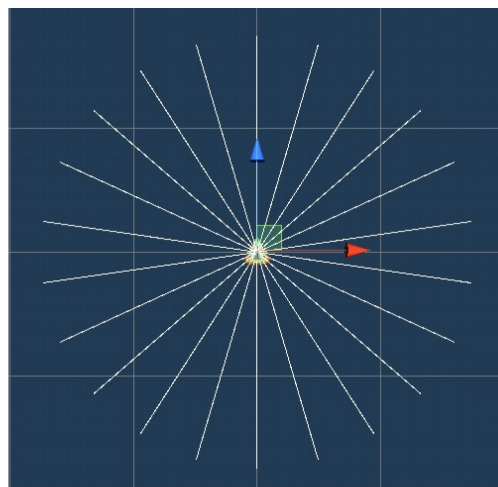
The first step to applying ML-Agents to this game was to first adapt it to the framework. Since my aim was to have multiple instances of the game running (to speed up training and avoid overfitting) most classes had to be reworked in order to make them easily "cloneable".

A tough one in particular was the **Euclidean Torus** class responsible for the "warping around the screen" effect. This class heavily relies on the game's camera in order to determine whether an object goes out of bounds and should have it's position reset. This isn't an issue when dealing with a single instance of the game (since then we would only have one camera we could, and originally, used the game's renderer to check whether or not the objects were visible). In the end the solution found was to have each instance of the game have its own camera (passed to that instance's GameManager), and use *GeometryUtility*'s *TestPlanesAABB* function to check whether the colliders of the object where within the camera's bounds or not. Similar modifications had to be done for other factors of the game, but this was perhaps the major one.

Alongside this I also disabled the *canvas* text that showed the score, wave, and so on. It was simply uncessesary for training.

Finally, I placed a camera, GameManager and Ship prefab in an empty game object. Called it instance and cloned it several times, making sure each instance was sufficiently distant from each other so as to not cause issues with an object leaving the bounds of an instance, and entering another's (imagine if for example, an asteroid from instance 2 flew into the space of instance 3 and destroyed that instance's ship!).

The **ship** prefab also had to suffer some alterations in order to have a brain and be able to be controlled and trained as an Agent. Besides this I also gave it some **RayPerception3D**s. I was originally having a lot of issues using these due to the ship's collision boxes not having been properly sized (basically they weren't centered, meaning the rays were spawning outside these colliders and immediately hitting them causing a whole slew of problems) and for a few iterations used my own implementation of raycasts (it was a jumbled mess but I couldn't find out the reason as to why my the ray perception wasn't working). Eventually I figured out the problem and erased my rays. The observations passed to the ship will be discussed when talking about each training scenario, but I found it appropriate to document the initial issues I had with the **RayPerception3D**s in this section.



Finally, in terms of training time, I ran each training run either until it stabilized or noticed a bug that warranted the restart of the training.

Tensorboard & Results

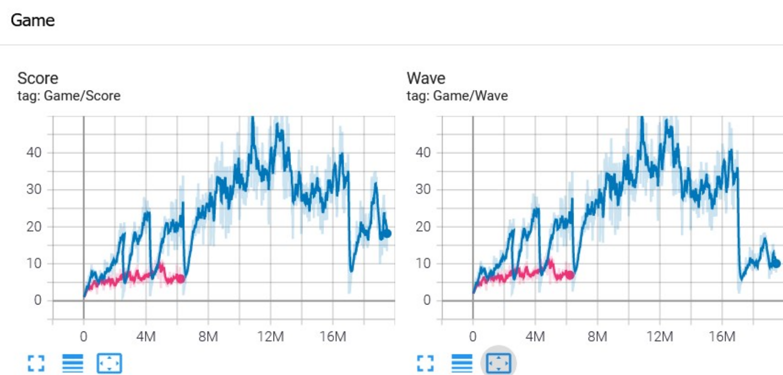
All training results and progress charts can be seen using *Tensorboard* (if installed). Simply navigate to the main project folder and use the command `tensorboard --logdir results` . All results are stored within the folder **results** which contains several subfolders pertaining to specific training scenarios/objectives/methodologies (*Shoot*, *ShootNRotate*, *MoveNShoot*, and *MoveNShoot_Curriculum*).

Each training run was named following the same pattern: `runNumber_objective_[changes made since last run]` . So for example the run *6_MoveNShoot_GivenClosestAsteroidInfo_Curiosity* was the 6th run of training the agent to Move and Shoot, and its different from the previous run because new observations were added (Closest Asteroid Info), and Curiosity was added to the configuration file. By sticking to this pattern, it became easier, both to analyze what had and had not yet been tried, as well as made comparing changes more manageable. More so, not all training runs/changes applied will be presented in this report, so I encourage the reader to at least glance at the names (or preferably see the graphs in Tensorboard) of the several runs, as these are more

or less indicative of all efforts done.

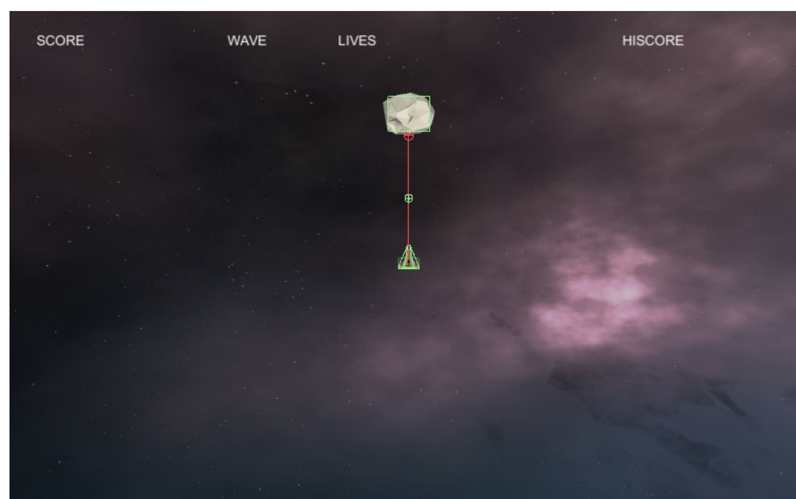
Additionally, in folder **BestResults** are some of the best runs achieved for certain objectives. Do note, however, that these models may not be suitable to the scenario that is currently set to run. For example, a ShootNRotate model was trained in a scenario where all asteroids spawn around the player, either moving towards it, or statically on a set radius, so that model may showcase some less than optimal behaviours on the scenario that is set to run normally (i.e asteroids spawn randomly and moving in a random direction).

Finally, two additional metrics were added to be captured by Tensorboard - *Score* and *Wave*. These information are captured at the end of each episode, and served moreso as a test and trial of how Tensorboard worked/new metrics could be captured through the ML-Agents framework, but never the less they do provide some information that may be (and was) taken into consideration when comparing models.



First Scenario - Just Shoot

The first scenario I put the agent through was as simple as could be. I gave the agent a single (discrete) action - *Shoot* - a single observation - a single front facing **RayPerception3D** ray - and always spawned an asteroid directly in front of the ship.



Initially I gave the agent a reward of 1.0 upon destroying the asteroid but after training for about an hour, I went to see its behavior and noticed it wasn't particularly optimal. The ship was shooting, yes, but it wasn't doing so as fast as it could. As such I gave it one additional observation

- whether it could shoot or not - and added a small time penalty at each frame. Honestly I was expecting that with these changes the observed graphical improvement would be noticeable, but the graphs seem to go pretty much hand in hand. Testing the model itself, however, this second one did manage to learn to shoot basically every time it could so it got the job done! It should also be said that at this point, the configuration file used for training basically used the default values for everything.

- **Observations**
  - Single **RayPerception3D** ray coming out of the front of the ship
  - Whether the agent was ready to shoot again or not
- **Rewards**
  - +1.0 when hitting the asteroid
  - -0.0001 each (fixed update) frame
- **Actions**
  - Shoot (Discrete)
- **Final Outcome**
  - Agent capable of optimally shooting at a fixed target infront of it

Second Scenario - ShootNRotate

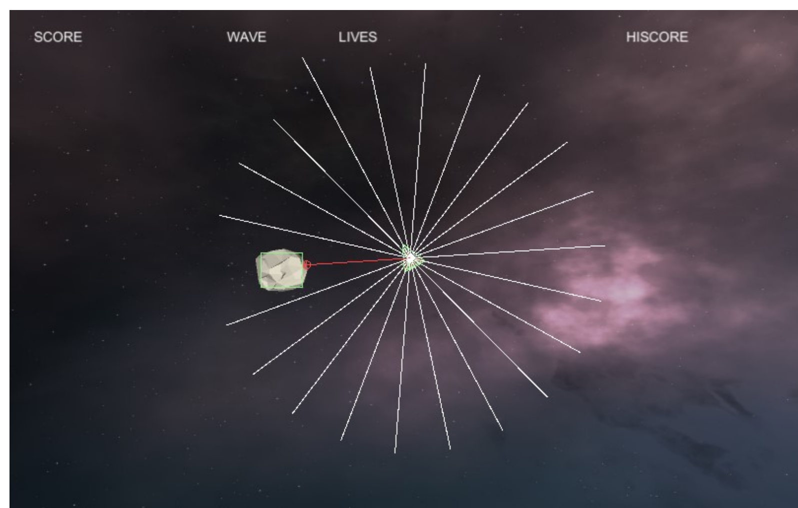
After using the last training scenario to make sure that training was actually working and the agent was capable of learning, it was time to add some complexity. Since movement is dependent on where the ship is facing, I decided it made more sense to start with a scenario where the agent only had to rotate towards a nearby enemy, and shoot (with no movement involved). By doing this I could be sure that the agent was capable of learning how to properly identify enemies, know how to optimally rotate, and know to only shoot when actually facing an enemy (rather than bullet spamming everywhere).

Single Static Asteroid

Starting off simple, I kept the last scenario (i.e the asteroid always spawned in front of the agent) but gave the agent the capability of rotating. So, optimally, the agent would learn that he didn't actually need to rotate at all, just shoot and be done with it! And that's exactly what it learned

(after reducing *Beta* a bit, since Entropy wasn't seemingly going down). Funnily enough the agent's final behavior was to start by turning left as far as possible while still keeping the asteroid detectable by the Ray perceptor and then stick to that position and just start blasting. This made me realize I should probably reset both rotation and position at the end of each episode in order to fully refresh the environment.

Next, I made it so the asteroid could spawn in a certain radius around the agent and not just straight infront of it. After the first training in this scenario, the agent's strategy was to simply start rotating either left or right, and keep rotating and shooting until it hit something.



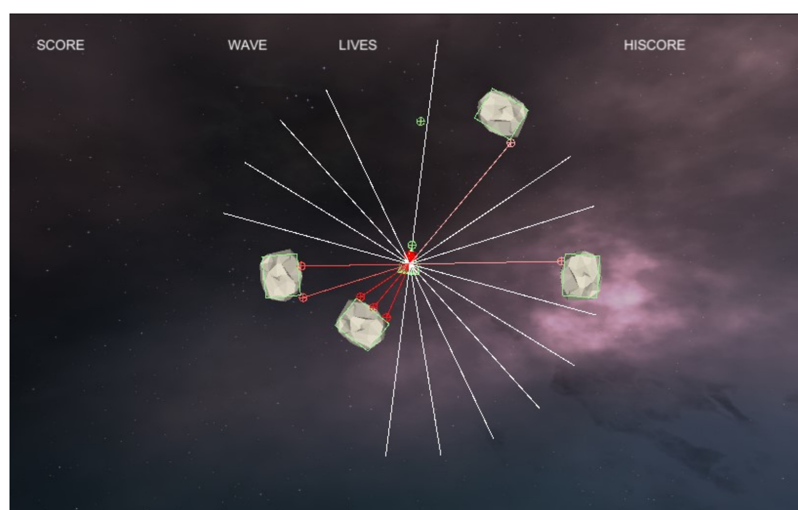
The remedy for this was two-fold. I started by added more **RayPerceptionSensor3D** sensors surrounding the agent so that it could be aware of the asteroid's position. Then I added a small penalty for each shot taken. I experimented with the way this penalty was applied. I started by applying the penalty each time the shot was taken and refunding the agent if the shot actually landed. Then I tried having the penalty only be applied if the bullet disappeared (i.e timed out without hitting anything), but with this the agent kept a sort of bullet spammy approach, probably because the bullet takes a few seconds to time out, so if he spammed in a certain way he could shoot several bullets that wouldn't land and not be penalized for it (since the episode would've ended by then). So I reverted back to the first approach. After some tweaking with several paremeters to increase the complexity (as rotating is a continuous action) - namely *Batch Size*, *Hidden Units* and *Time Horizon* - the agent was then capable of picking the rotation direction that would make it reach the asteroid the fastest, and shoot only when it knew the shot would land. I also had to alter some parameters to make the training more stable since at the beginning I noticed the agent wasn't learning to start rotating in the optimal direction (for example an asteroid spawned in a way that would make more sense to start rotating left but the agent would start rotating right). I warranted this to the agent getting stuck in a local maximum so I tweaked things like *epsilon*, *beta* and the *learning rate* until I was happy with the results. Perhaps increasing the penalty applied each frame would have also helped with this issue but I didn't want the agent to constantly be getting penalizations (especially since the final game itself doesn't really have any time constraints)

- **Observations**

  - Single **RayPerception3D** ray coming out of the front of the ship (uncapable of detecting enemy bullets since these aren't destructable)
  - Whether the agent was ready to shoot again or not
  - Several **RayPerception3D** rays surrounding the ship
- **Rewards**
  - +1.0 when hitting the asteroid
  - -0.0001 each (fixed update) frame
  - -0.1 for every bullet shot
  - +0.1 for every bullet landed (applied at the time the bullet hits the asteroid)
- **Actions**
  - Shoot (Discrete)
  - Rotate Left and Right (Continuous)
- **Final Outcome**
  - Agent capable of optimally rotating towards an asteroid spawning around it and shooting it without spamming bullets

Inbound Asteroids

So up until now, the only urgency the agent had to destroy the asteroid was to get the reward as fast as possible (and minimize the penalties infringed in each frame). Besides wanting to make sure the rotation was optimized, I also wanted to see if the agent could learn to properly prioritize targets. To do this I added some urgency to the last scenario. Rather than having a single static asteroid spawning around the player, I started spawning 4 that would spawn around the player at random distances and, with a random velocity, start moving towards the player. If any asteroid collided with the ship, it was game over.



Funnily enough, I didn't even need to make any major changes to the previous trainings. I experimented with different time horizons and such, but in the end, what worked best for the previous scenario also prooved to work really good for this one! I was actually impressed by how well the agent did. It managed to survive most onslaughts of asteroids and properly prioritize

which asteroids to shoot first!

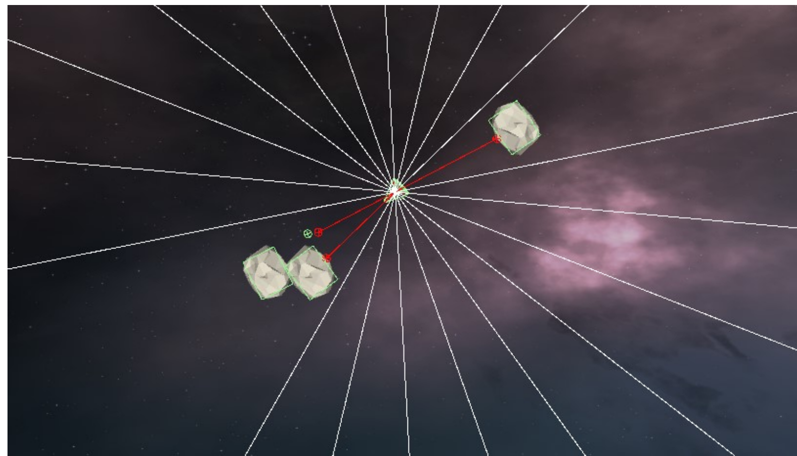
- **Observations**
  - Single **RayPerception3D** ray coming out of the front of the ship (uncapable of detecting enemy bullets since these aren't destructable)
  - Whether the agent was ready to shoot again or not
  - Several **RayPerception3D** rays surrounding the ship
- **Rewards**
  - +1.0/4.0 when hitting the asteroid
  - -0.0001 each (fixed update) frame
  - -0.1 for every bullet shot
  - +0.1 for every bullet landed (applied at the time the bullet hits the asteroid)
  - -1.0 if hit by an asteroid (dies)
- **Actions**
  - Shoot (Discrete)
  - Rotate Left and Right (Continuous)
- **Final Outcome**
  - Agent capable of prioritizing which asteroid to shoot at first in order to survive enemies flying towards it

Third Scenario - MoveNShoot

Finally it was time to add the final action to the agent. We were in the endgame. If the agent could shoot, rotate AND move it would basically be done! It would be able to play the full game! But right from the get go I knew it wouldn't be easy to have the agent learn how to properly move. There were several challenges ahead. The agent had to learn how to track an enemy, how to move towards it with a movement system that's based on acceleration and direction, it had to learn that enemies aren't static and move around, it had to deal with the whole warping aspect of the game and it had to deal with not crashing into anything. The task was daunting, and the fear of failure terrifying.

Inbound Asteroids w/ Movement

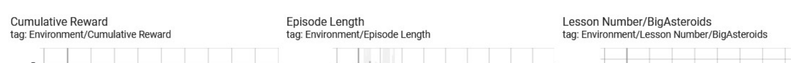
I started off with the same approach I followed when transitioning from just shooting to shooting and rotating. I used the same scenario as last time but added the movement action. After a few iterations of tweaks, mostly related to dealing with the increased complexity brought upon by the addition of another continuous action, the agent was actually behaving pretty well! The agent actually learned that it could sort of exploit the physics of the game and be able to rotate faster by also accelerating, which made for some interesting trick shot tactics! Alongside this the agent also learned to dodge asteroids if they came too close.

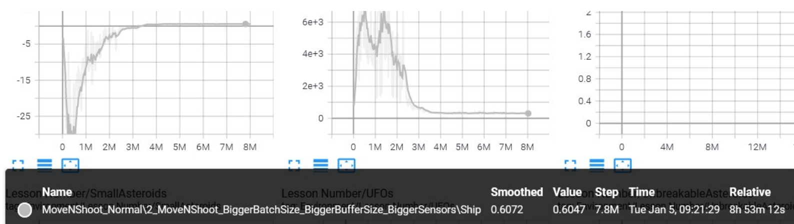


During this phase I also extended the perception rays. I did this because I was afraid the agent would lose track of the asteroids if he managed to dodge them as they continued to drift away. I quickly realized why this wasn't the smartest move as will be explained in the next sections.

- **Observations**
  - Single **RayPerception3D** ray coming out of the front of the ship (uncapable of detecting enemy bullets since these aren't destructable)
  - Whether the agent was ready to shoot again or not
  - Several **RayPerception3D** rays surrounding the ship
  - Forward vector (so that the agent knows in which direction it'll move if it accelerates)
- **Rewards**
  - +1.0/4.0 when hitting the asteroid
  - -0.0001 each (fixed update) frame
  - -0.1 for every bullet shot
  - +0.1 for every bullet landed (applied at the time the bullet hits the asteroid)
  - -1.0 if hit by an asteroid (dies)
- **Actions**
  - Shoot (Discrete)
  - Rotate Left and Right (Continuous)
  - Move by Accelerating or Deaccelerating (Continuous)
- **Final Outcome**
  - Agent capable of prioritizing which asteroid to shoot at first in order to survive enemies flying towards it and apply some acceleration boosts in order to speed up rotation and avoid collisions

In terms of training, it took about 6 hours until the agent stabilized. The fact asteroids were coming towards the agent meant each episode's length was pretty limited, and the rewards reacurrent, hence the environment was very "reward dense", which helped probably helped the agent train faster, despite the added movement complexity.

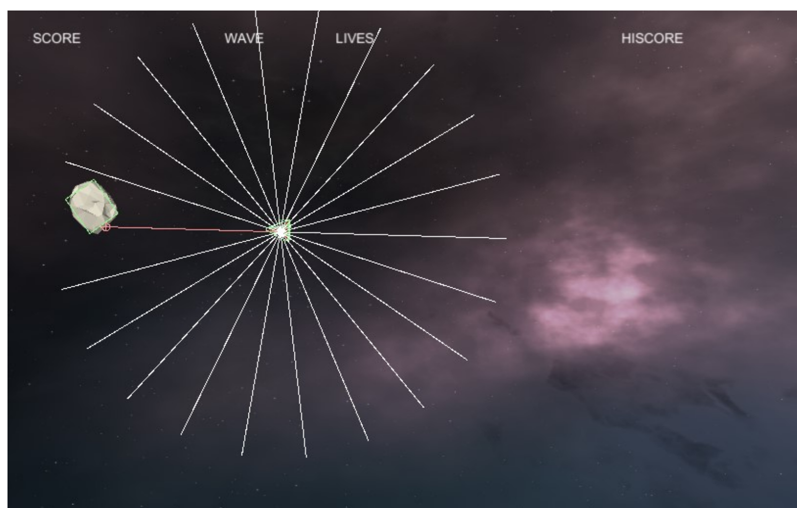




Single Non-Splitting Asteroid

So the agent was able to conquer the last scenario even with the added complexity of movement. Next step was to have it seek enemies and actually play in a game-like scenario! So what I did next was spawn 6 asteroids randomly moving in different directions. I added more observations - Velocity (so the agent knew how fast it was going) and Position (in the hopes the agent would learn what happens if he goes off any side of the screen). This failed. Hard. The agent wasn't learning and just kept pulling off random actions, no matter the tweaks I did to the learning configuration file. At first I atributed it to the big ray perception method I implemented in the last scenario. So I took it out. But even then the agent wasn't really learning much. I think what it came down to was the fact that I put it in a scenario that was way too hazardous and hard, which hindered the learning process. It was time to take a step back and simplify.

I started by spawning just a single asteroid. This time the agent managed to learn something, albeit not optimal. Since the only sensors it had were close-ranged, if the asteroid was too faraway the agent didn't really know what to do. So it adopted a strategy of moving forward whilst rotating around and shooting randomly in the hopes of hitting something. Not what I wanted, but the fact it learned a semi-viable strategy gave me back some much needed hope (I had already restarted the project once because I got to a point similar to this one and, in desperation started adding too many nonsensical observations and network complexity, to the project was so jumbled, I decided to just start over).

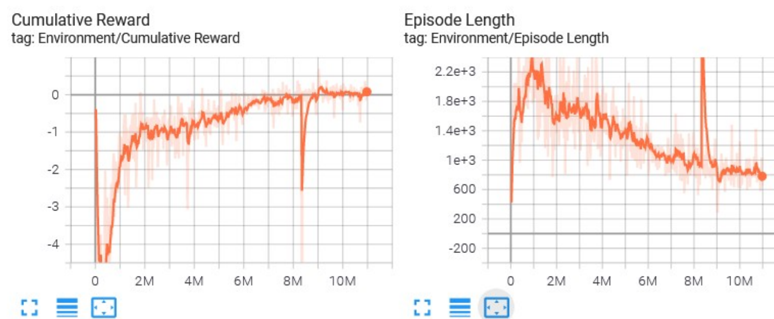


To solve this issue I added what I called *Closest Asteroid Information*. I had the GameManager keep track of all enemies in the game's instance and, when collecting new observations, made the agent check which of the enemies was closest in order to collect, originally, it's position and direction differential (i.e the angle between the agent's forward vector and the vector going from the ship to the enemy), and after some experimentation, it's velocity and distance.

After some tweaking of the training parameters and further experimentation (notibly with Curiosity, and the addition of a timeout to try to condense the maximum time length of each episode), I ended up with a model that very aptly chases down an asteroid and destroys it! Furthermore, testing prooved that this model actually worked well even if there was more than one asteroid meaning the agent was actually finally apt to play the final game!... If not for the fact that the asteroids the agent trained against didn't actually split, and the agent had yet to go against UFOs.

- **Observations**
  - Single **RayPerception3D** ray coming out of the front of the ship (uncapable of detecting enemy bullets since these aren't destructable)
  - Whether the agent was ready to shoot again or not
  - Several **RayPerception3D** rays surrounding the ship
  - Forward vector
  - Velocity
  - Position
  - Closest Enemy Velocity
  - Closest Enemy Position
  - Closest Enemy Distance
  - Signed Angle between ship's forward vector and vector going from the ship to the closest enemy
- **Rewards**
  - +1.0/4.0 when hitting the asteroid
  - -0.0001 each (fixed update) frame
  - -0.1 for every bullet shot
  - +0.1 for every bullet landed (applied at the time the bullet hits the asteroid)
  - -1.0 if hit by an asteroid (dies)
- **Actions**
  - Shoot (Discrete)
  - Rotate Left and Right (Continuous)
  - Move by Accelerating or Deaccelerating (Continuous)
- **Final Outcome**
  - Agent capable of moving towards an enemy and destroy it without dying. Also capable of performing well versus multiple enemies (granted they don't split).

Going from run 8 (the best in the last scenario) to run 10, I was surprised how little iterations/tweaking were necessary to have the agent seeking the target. It should be noted however, that episodes were longer, and training was much slower, with stabilization starting about 10 hours into training.

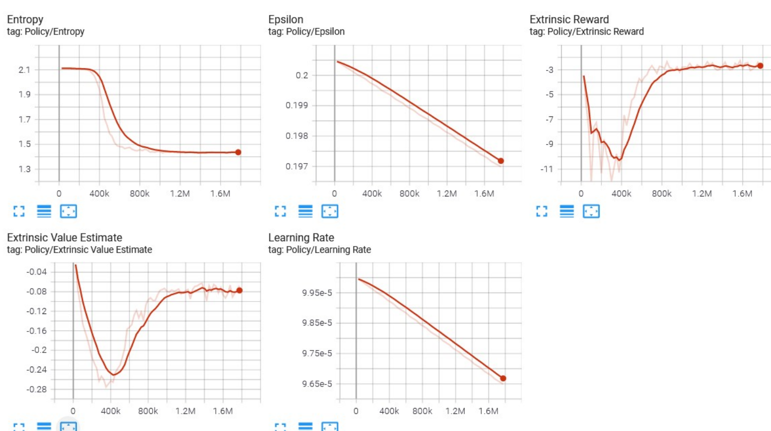


Single Splitting Asteroid

The agent obtained in the last scenario was able to adequately play the game against non-splitting asteroids. Against asteroids that when broken split into several ones however, the agent wasn't as good. Sometimes it performed really well and adequately adapted to new asteroids spawning from broken ones, but most of the times the agent would destroy the big asteroid, and immediately crash into the ones that spawned (since it was used to asteroids fully disappearing after being shot).

So back to training. This time I used the same scenario as last time, but with an asteroid that actually split. This didn't work out too well, however. The problem was the same as when I tried to train with too many asteroids at once. The scenario was too hard. The episodes took too long to complete since, in total, the agent would have to destroy 9 different asteroids (the big one splits into two mediums which in turn split into 3 small each).

Note, in the following image, how stabilization occurred without the agent really learning anything. Even after tweaking values to make entropy drop slower, and messing with other parameters, I wasn't being able to train the agent using a breakable asteroid (probably for the aforementioned reasons). It was clear I wouldn't be able to get anywhere with this methodology. It was time for **Curriculum Learning**

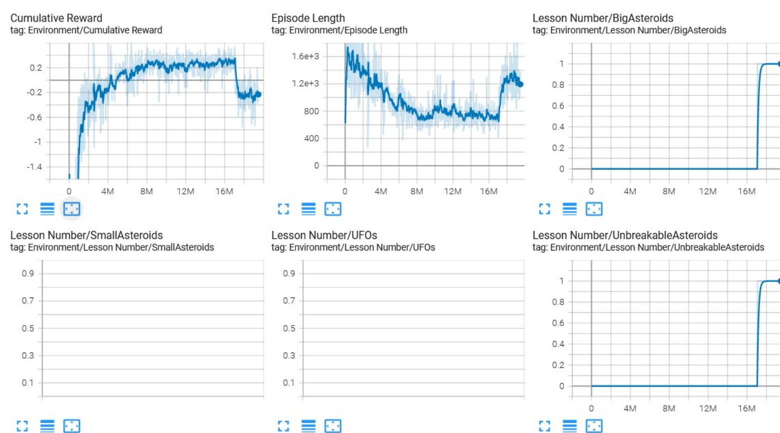


Curriculum Learning

The last thing that was attempted to make the agent learn how to properly play the game was through the usage of **Curriculum Learning**. The issue we were facing in the last section was that, for the best model learned, the agent was trained with a single non breaking asteroid. As such, it

would learn how to zone in on its target and destroy it quite aptly. But since our asteroids split, meaning that after their destruction, the agent can't just assume the game to be over, it needs to slow down, not go into the newly created asteroids, and so on. Furthermore the agents learned movement was a bit "cocky". Since it only trained with one asteroid, it never really learned to dodge asteroids, it just learned to go towards them and shoot. Immediatlely setting out to train with harder environments from the start was also flawed since the agent just wasn't learning that way. Using Curriculum Learning, however, prooved to be a bit underwhelming... and it should be noted that it did make training have to run for much, much longer comparatively.

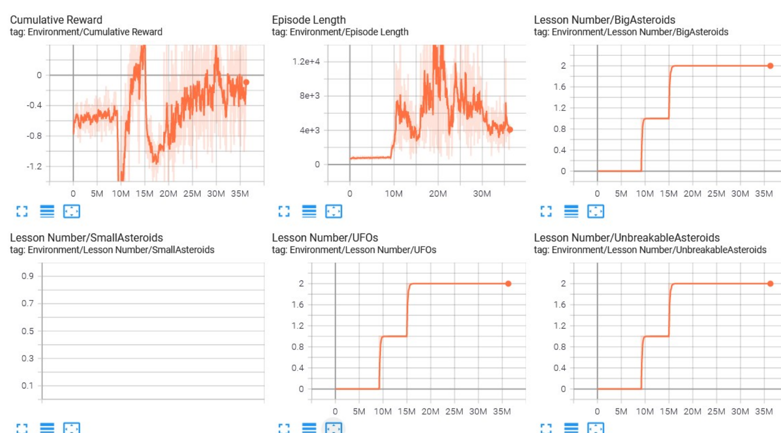
Around 20 different runs were tried, changing the lessons in terms of number of unbreakable asteroids, breakable asteroids and UFOs. I tried different reward systems (per recommendation of the teacher), such as giving the agent a 1.0 reward only after having deleted all asteroids, I tested different complexity levels in the network, different parameters, but the problem persisted. The agent would learn on easier levels, then as soon as more complexity was added (for example going from a lesson with 1 asteroid, to one with 3), the reward per episode would drop drastically, and the agent wouldn't learn. This is exemplified in the following graph from run 18 which was one of the best runs (managing to play decently well, but still too reckless to be able to survive versus a lot of asteroids).



After some consideration, a different strategy was attempted. So the agent was managing to learn how to seek and destroy asteroids, but didn't learn to be careful enough actually, avoid being hit by them. So, what if we created a "LessonZero". A lesson where the agent would be deprived of its ability to shoot, and 10 asteroids would be spawned around him. If he survived for 1000 steps, it would be awarded a 1.0 reward, else, it would be penalized with a -1.0. The idea is that, after this lesson, the agent would be better prepared to tackle harder lessons, and immediately be placed against 6 asteroids in Lesson One.

After training with this methodology, the results were promising. After a few hours of training the agent was masterfully avoiding and dodging asteroids! It still died regularly mind you, mostly due to getting to close to corners of the map, and an asteroid warping to it, not giving it enough time to dodge. But this was enough to allow it to train versus a lot of asteroids. And so it did. After Lesson One versus 6 asteroids (now back to the normal environment/reward system and with the ability to shoot) the agent was managing to destroy all asteroids and not die pretty nicely. There was a problem however..since it immediately started training versus a lot of asteroids, it wasn't

managing to learn how to actually seek and approach an asteroid to destroy, instead it was flying around, and when close to an asteroid, shooting it. This works when the screen is filled with asteroids, but when there's only one left, the agent would take ages to destroy it, and eventually time out (a 25000 step timeout was added, to avoid negative rewards racking up too high and to force the closure of episodes if they took too long). It was almost like we had the inverse problem. This, run 31, showcases some interesting behaviour, such as the agent stopping at a safe distance from the asteroid before shooting it, in order to not be hit by the splitting debris, but it also has some downsides, such as the aforementioned ones, as well as a strange pattern of always starting the episode by doing a weird 360 turn (probably due to some bias during the lesson 0 training).



For our final curriculum (due to time constraints) - train 32 - we had the following lessons:

- **Lesson Zero**
  - 10 Asteroids
  - No shooting
  - +1.0 if agent survives for 1000 steps
  - -1.0 if agent dies
- **Lesson One**
  - 1 Unbreakable Asteroids
- **Lesson Two**
  - 6 Unbreakable Asteroids
- **Lesson Three**
  - 2 Breakable Asteroids
- **Lesson Four**
  - 2 Breakable Asteroids
  - 1 UFO
- **Rewards from Lesson One onwards**
  - +1.0/#OfAsteroids when bullet hits the asteroid
  - -1/12500 each (fixed update) frame
  - -0.05 for every bullet shot
  - +0.05 for every bullet landed (applied at the time the bullet hits the asteroid)

    - -1.0 if agent dies
    - Episode forcefully ends after 12500 steps (timeout)

The plan was flawless...but disaster struck. I left the agent training the day before delivery. It managed to pass Lesson Zero and was able to aptly avoid asteroids all day long. Then I left it training during the night, and when I woke up I realized my PC had shut down because it wasn't properly connected to the battery..and as such, the training stopped. As it was the last day I didn't have time to fully have it go through all the lessons, and as such run 32 ended with a model that is capable of evading asteroids, and thats about it.

Final Model

All in all, despite certain shortcomings, we ended with several apt models. Model 10 and 18 perform best when the number of asteroids is minimal as they're the best at seeking and destroying asteroids as fast as possible. Model 31 is pretty good when the number of asteroids is high, but it didn't properly learn to search and destroy.

To see these agents in action there are several scenes that can be played:

- **Main Scene**
    - No A.I
    - Used to play the game normally
- **ML_10**
    - ML Agent from training run 10
    - Trained without curriculum learning
    - Good versus low number of asteroids/unbreakable asteroids, but movement is not very fluid
- **ML_18**
    - ML Agent from training run 18
    - Trained with curriculum learning
    - Good versus low number of asteroids/unbreakable asteroids
- **ML_31**
    - ML Agent from training run 31
    - Trained with curriculum learning
    - Good versus higher number of asteroids
    - Has some weird patterns
- **ML_32**
    - ML Agent from training run 32
    - Trained with curriculum learning
    - Training interrupted due to crash

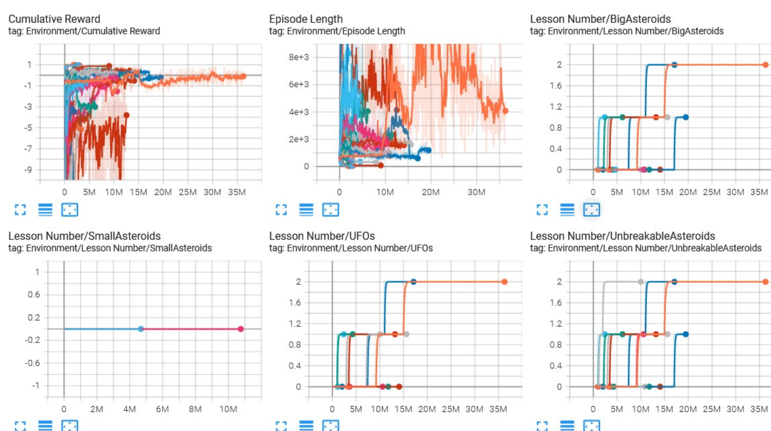
    - Good at dodging asteroids
- **Training Scene**
    - Scene used for training

Globally, Runs 18 and 31 both perform decently, although they're complimentary in a way. Run 18 manages to win waves with ease, granted the asteroids are unbreakable (it struggles against breakable asteroids). Run 31 is decent at breakable asteroids and for when a lot of enemies clutter the screen, but when theres little asteroids left it basically roams around randomly trying to find a target. Both perform adequately against UFOs which is surprising since they had very limited training against them (run 18 trained for a few hours with a UFO, and run 31 didn't even each that lesson).

Conclusion

This project was a pretty fun! It allowed me to apply knowledge gained from other studies about Reinforcement Learning and Machine Learning in general to the realm of videogames for the first time and better than that, the final result was an agent that was actually capable to, at least decently, play the game! This isn't to say it wasn't without its gripes however. Whilst training agents via machine learning is one of the most rewarding feelings when they actually learn what you want and start doing cool things like pulling off trick shots, on the majoraty of trainings the agent's lack of brain is both depressing and demotivating (there were points where I honestly didn't think the agent was going to be able to learn more than to randomly move and shoot in the hopes of hitting a target) and I didn't even mention the one time where I scrapped the entire ML part of the project and started over because I was at a point where I added so many observations and tweaked so many configuration parameters that I was lost and didn't know what else to do (and good thing I did reboot it or else I may have not have been able to regain control of the situation in a more methodical way). That, and the fact that my computer had to stay turned on for *weeks* training model after model did lead me to some dread. But in the end it all worked out, and I managed to add a new framework and A.I technique to my arsenal.

To close off Im leaving an image with all training runs side to side.



References & Assets

For the elaboration of this project the official ML-Agents documentation was vital for understanding what each of the settings of the configuration file pertained to.

Some tutorials by Sebastian Schuchmann were also a good introduction to how to use the ML-Agents framework. Of note are the videos Unity ML-Agents 1.0 - Training your first A.I and ML-Agents 1.0+ | Create your own A.I. | Full Walkthrough | Unity3D.

The model meshes used for the game were graciously provided by the following authors (under CC):

- **Spaceship** by *Liz Reddington*
    - Makes it so UFOs don't spawn
- **Asteroid** by *Poly by Google*
- **Flying Saucer** by *Poly by Google*