SVS 2022

Assignment 1

Project description

The first exercise aims to implement a simple drone behavior to learn how to take advantage of reinforcement learning for this type of task. We have a set of four drones, each one controlled by a separate policy, placed in an empty space near ground level with random flying spheres generated in front of them. The objective of this exercise is to train the drones to fly between spheres till the end of the path while avoiding obstacles (spheres and themselves) without leaving the defined area.

How we solved it

The proposed exercise sounds easy, but hides a lot of problems, derived from our poor initial knowledge of reinforcement learning and the libraries used to implement it. But this is part of the next section; here are the main ideas we implemented until we reached the final form we delivered:

The easiest way to start was to totally ignore flying spheres, and we just gave a reward to the drone every time it could move forward. Learning to move straight was initially a good step, but problems were encountered: the drones kept flying forward and backward without ever moving on. We fixed this by giving drones a reward only when they actually beat their maximum distance reached.

- The easiest way to start was to totally ignore flying spheres, and we just gave a reward to the drone every time it could move forward. Learning to move straight was initially a good step, but problems were encountered: the drones kept flying forward and backward without ever moving on. We fixed this by giving drones a reward only when they actually beat their maximum distance reached.

- Drones go straight but slowly. To fix this, we multiplied the reward with the straight speed they have in the correct direction if they get a reward for every little movement they make.
- Drones started acting strange and kept moving on with a strange "swing" movement to maximize speed when getting the reward. To fix this, we added a sub-goal: now only a minor reward is gained when moving on, and a major one when reaching a sub-goal, which is a portion of the total path.
- Drones act better, but still exploit the speed fact. To fix this, we stopped multiplying the reward based on speed. Instead, the minor reward is gained based on the distance from the next sub-goal. In this way, the drone is actually motivated to move forward.

Nice! Now drones move forward and pretty fast. A nice long job was done, but now we need to make them avoid obstacles! Drones don't have eyes but only data perceived, which are the ten nearest spheres' positions. So we had to modify the reward to make them avoid obstacles. The first solution was to give negative reward based on the distance to spheres, but again, problems were found:

- Drones kept flying high out of the world, on top of it, in order to avoid all the spheres and reach the goal easily. We fixed this by adding world boundaries. Now, if the drone leaves the designated area, it gets a negative reward and is killed.
- Drones won't leave the initial area, too afraid of spheres. To fix this, we decreased the negative reward but still not working, so we set a minimum distance to be near in order to effectively receive negative reward.
- Drones act good, but we observed we could optimize performance. We saw that with this reward most of the time, only one sphere was considered to calculate reward. So in order to divide by a factor of ten the calculation needed, we started considering only the nearest sphere. Actually, performances stayed the same, but training time got a nice decrease.

Problems encountered

A lot! Mostly caused by the strange behavior of the library we used, or else caused by our initial difficulties with training:

- Spheres were poorly rendered: all the spheres had the same size. After hours of debugging, we could find it was a problem of cache. Spheres were actually generated well, but rendered poorly. It wasn't a problem for training because it was only a problem of vision, but drones worked with coordinates and physics, which were good. So we left cache on for training and off for visualize the result, and it was good.
- Drones death causing problems: if a drone was declared dead, the very next step it would cause a fatal crash of the training, that problem was pretty hided and took hours to be found out, our solution after other hours of debugging was to simply make all drones die whenever any drone was dead, that fixed the problem and didn't cause problems with the training performances.
- A lot of minor issues that summed up took away many many other hours of debugging, can't really describe all of them, but the feeling was not really good, every time we had to find solutions to an incredibly apparently random crash of training… considering always a not meaningful error message and a total absence of documentation, well, it was a pain…

Assignment 2

Project description

The second exercise is based on the concepts learned in the first one. We also have to use reinforcement learning to train drones, but this time, the task is much harder.

Description of the setting: A static arena surrounds the area of battle. There are 4 drones lined up facing each other in 2 rows, representing the blue team and the red team. Spheres are used as bullets.

Task: The objective is to win by destroying the enemy team by throwing spheres at them, making them "die." Each drone has 5 bullets, which they can shoot no sooner than 3 seconds from the start of the battle and no more than once per second. Drones can fly freely inside the arena, but any collision (with a sphere, the arena, another drone...) causes them to "die."

How we solved it

This exercise was very difficult and took a long time to solve. We cannot describe all the steps taken, as we did with Assignment 1, because we had to take many steps before reaching the final version. Therefore, for the sake of clarity, we will only describe the major decisions we made and the final results.

There were many collisions to be accounted for, as we can see from the exercise text. We had to check collisions between spheres and drones, spheres and the arena, arena and drones, etc. This was a lot of work and difficult to identify. We also noticed that some collisions were avoided due to the step not being continuous, so sometimes spheres hitting drones or the arena were not revealed, causing a nonsensical evaluation and leaving open possibilities for exploiting this fact. Due to this, we spent hours searching for an event we could subscribe to in order to perceive collisions directly generated from the battle, but we were unsuccessful in finding any kind of event we could exploit. Taking this into consideration, we decided to switch to another solution: we avoided checking for collisions. Yes, we explicitly ignored collisions and limited ourselves to simply checking the height of objects. Every step, we checked every object's distance from the ground. If it is below a certain level, it means it should be destroyed. For spheres, they are simply removed, but for drones, it means they are "dead" and destroyed.

The observation space had to include the actual position and rotation of the drone, augmented with data from its camera, the number of spheres it has left to shoot (initially 5 for every drone), and the time elapsed from the beginning of the episode (in order to make policies understand how they receive negative rewards if they shoot in the first 3 seconds, and how they have to shoot at maximum once per second). This observation space is quite large, due to the presence of the camera image of each drone, which caused a real increase in training time. To stay within the project limits, we included the image in its entirety within the observation space, but we lowered the resolution of the observed image, keeping the training time at a decent level.

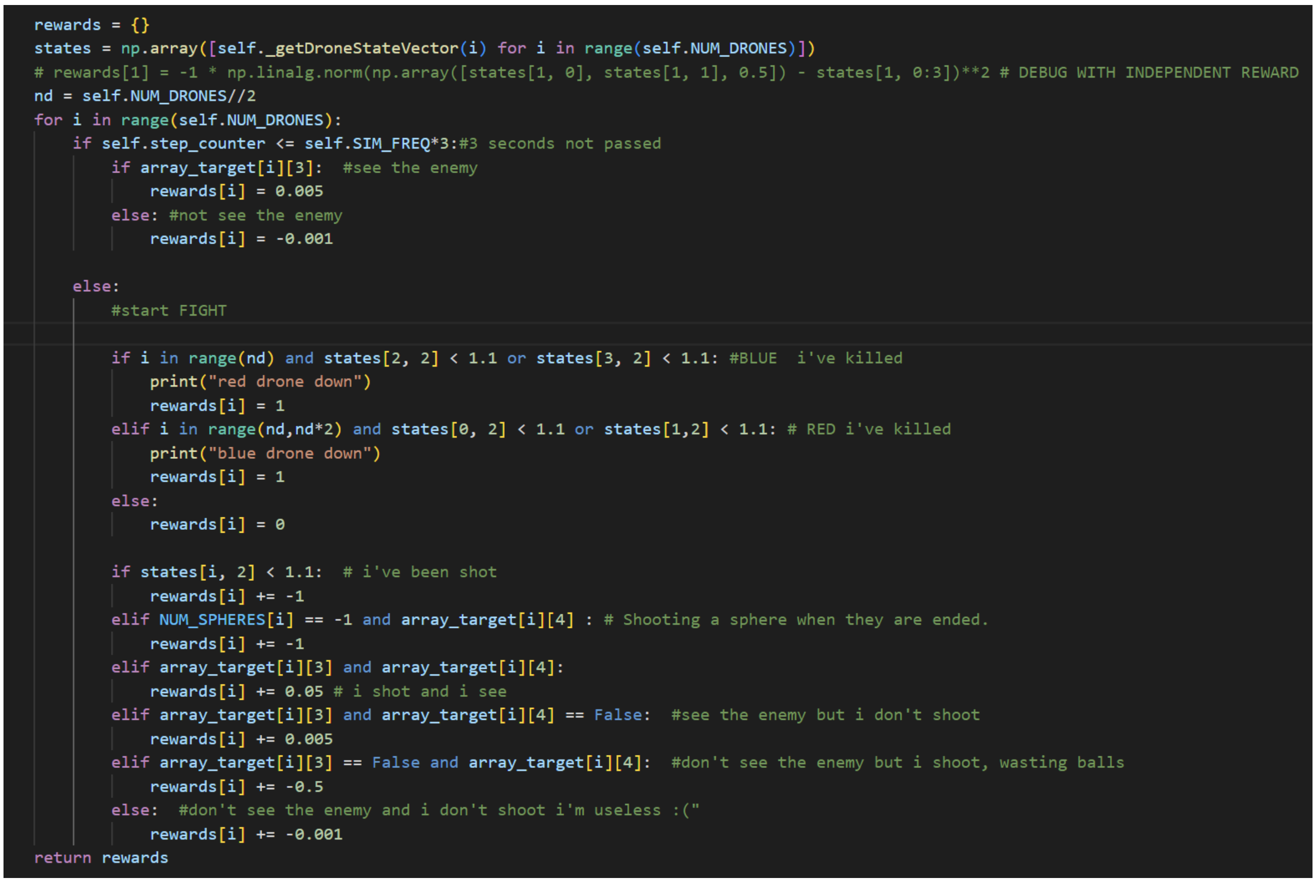
The exercise was originally meant to have 4 drones fighting against 4 other drones. This worked well for most of the time, but at a certain point in the project, 8 drones became too many to be trained. For some reason we didn't find, the RAM of the training PC was filled and the training crashed because of it. So, we had to lower the number of drones to a total of 4, with 2 against 2. This worked well and resolved all the memory problems.

Drones shooting:
In the "step" function, the "RED_ARRAY_TARGET" and "BLUE_ARRAY_TARGET" are reset, and the "setupForStep" function is called for each drone to detect other drones. Then, it checks if any drone wants to shoot, if 3 seconds have passed and if a drone has a target in its sight, then it calls the "_drone_shoot" function to shoot a sphere.

In the setupForStep function, the drone elaborates his image to check if it sees an opposite drone. The transformation runs as follows:
The first point is to convert the droneBGRA image to HSV format and then we can filter this new image with the color of the opposite drone, for example the blue one tries to filter the red color in the image, the opposite for the other one.
The filtering result produces a mask containing 1 in the pixels where the searched color was present (red in the example case), or 0 where it was not present.
With this mask we are able to understand if a drone is seeing an opposite one, simply by checking if in the mask there is at least a 1.

To decide where to shoot we used the camera target. We know that the camera is watching at the target [100, 0, 0] in front of the drone and we also know the index of the matrix where the other drone was spotted.
The drone processes the pixel index set to 1 (opponent drone spot) considering the center of the image as [0, 0] and the axes will be decreasing from the upper left corner to the lower right corner.
With this configuration it will not be possible to shoot outside the confines of the vision camera, as the target calculation will be as follows:
TARGET = [100, (CAMERA_VISION[0] // 2) - j, (CAMERA_VISION[1] // 2) - i]

Reward function: A lot of time was spent on this. Many different attempts were made, starting from simple rewards in order to find the best one. We cannot summarize all the decisions made, but we can certainly describe the final form we got.



```python
rewards = {}
states = np.array([self._getDroneStateVector(i) for i in range(self.NUM_DRONES)])
# rewards[1] = -1 * np.linalg.norm(np.array([states[1, 0], states[1, 1], 0.5]) - states[1, 0:3])**2 # DEBUG WITH INDEPENDENT REWARD
nd = self.NUM_DRONES//2
for i in range(self.NUM_DRONES):
    if self.step_counter <= self.SIM_FREQ*3:#3 seconds not passed
        if array_target[i][3]:  #see the enemy
            rewards[i] = 0.005
        else: #not see the enemy
            rewards[i] = -0.001

    else:
        #start FIGHT

        if i in range(nd) and states[2, 2] < 1.1 or states[3, 2] < 1.1: #BLUE  i've killed
            print("red drone down")
            rewards[i] = 1
        elif i in range(nd,nd*2) and states[0, 2] < 1.1 or states[1,2] < 1.1: # RED i've killed
            print("blue drone down")
            rewards[i] = 1
        else:
            rewards[i] = 0

        if states[i, 2] < 1.1:  # i've been shot
            rewards[i] += -1
        elif NUM_SPHERES[i] == -1 and array_target[i][4] : # Shooting a sphere when they are ended.
            rewards[i] += -1
        elif array_target[i][3] and array_target[i][4]:
            rewards[i] += 0.05 # i shot and i see
        elif array_target[i][3] and array_target[i][4] == False:  #see the enemy but i don't shoot
            rewards[i] += 0.005
        elif array_target[i][3] == False and array_target[i][4]:  #don't see the enemy but i shoot, wasting balls
            rewards[i] += -0.5
        else:  #don't see the enemy and i don't shoot i'm useless :("
            rewards[i] += -0.001
return rewards
```

Problems encountered

We encountered many crashes, struggles, and spent a lot of time trying to solve them by following bad error descriptions to find a solution. The observation space caused a lot of problems, as sometimes it wasn't properly normalized, causing data to exceed the minimum or maximum initially set. However, the error encountered wasn't specific about what was exceeding, making it difficult to fix the normalization. This issue occurred frequently due to various changes made to the observation space, resulting in a significant amount of time being spent on it.

We also encountered many small errors throughout the code, which is common in a long, complex exercise like this. However, unlike other times, the errors printed right after a crash were often useless, sometimes only pointing to code inside libraries not written by us. Additionally, errors were not always encountered, making it difficult to locate the source of the problem. This resulted in heisenbugs being found, with some bugs only appearing when certain print statements were included and disappearing when they were removed.