Based on the Harvest environment from:

Leibo, J. Z., Zambaldi, V., Lanctot, M., Marecki, J., & Graepel, T. (2017). Multi-agent reinforcement learning in sequential social dilemmas. In Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems (pp. 464-473).

Original environment: Harvest

A tragedy-of-the-commons dilemma in which apples regrow at a rate that depends on the amount of nearby apples. If individual agents employ an exploitative strategy by greedily consuming too many apples, the collective reward of all agents is reduced.

The dilemma is as follows. The short-term interests of each individual leads toward harvesting as rapidly as possible. However, the long-term interests of the group as a whole are advanced if individuals refrain from doing so, especially when many agents are in the same local region. Such situations are precarious because the more harvesting agents there are, the greater the chance of permanently depleting the local resources. Cooperators must abstain from a personal benefit for the good of the group

Specification

The goal of the Harvest game is to collect apples. Each apple provides a reward of 1. The apple regrowth rate varies across the map, dependent on the spatial configuration of uncollected apples: the more nearby apples, the higher the local regrowth rate. If all apples in a local area are harvested then none ever grow back. After 1000 steps the episode ends, at which point the game resets to an initial state.

Implementation

You will work upon the code provided in this zip file.

It is based on the code from (**this is just for reference, you only need to use the code provided in the zip file!):**

https://github.com/tiagoCuervo/CommonsGame

Which can be solved applying deep reinforcement learning like in:

https://github.com/tiagoCuervo/PPO-harvest

https://github.com/eugenevinitsky/sequential_social_dilemma_games

https://github.com/social-dilemma/multiagent

Since it applies the Gym environment, any algorithm implementation for a Gym environment should also naturally work:

https://gym.openai.com/

Our adaptation

Beneficence

Our model slightly changes the environment. First, we make the agents keep in mind how many apples they have taken, and also allow them the possibility to donate some of the collected apples to a common pool.

The main idea is that there is a source of inequality between the agents, that is, some of them are more fit for surviving because either they are less likely to get ill than the rest, or because they are more efficient at collecting apples (they are faster). You can select which of them is the source of inequality. They are bot additions to the original environment.

The objective is thus that, at the end of the simulation (which can represent harvesting season), each agent has enough food to survive.

This way, we can consider the moral value of beneficence.

The agents can either take directly from this common pool or restrict it and only distribute these apples at the end of the simulation with a formula. As it is, it there are no restrictions whatsoever.

Ecology preservation

Similarly, we can also consider the moral value of ecology preservation to solve the original problem of the harvesting game: that apples do not become extinct. This can also be seen as a subtask, because if there are no apples left, then agents will also die from starvation.

Alternatively, we can consider that ecology preservation is not a priority in our environment and slightly alter the environment dynamics so that apples never become extinct.

You will also need to decide how to handle the fact that agents can use beam guns. Do you consider that unethical and thus banned? Or do you consider it permissible? Recall that in the original harvesting game, beam guns are applied to actually regulate the environment and avoid that apples become extinct.

You will need to define the reward function to decide what are the ethical priorities of the agents and then test with some reinforcement learning algorithm what are the behaviours actually learnt by the agents.

Code documentation

Usage

It is required that you install the following packages:

- Gym
- Pycolab
- Matplotlib
- Numpy
- Scipy

Example and how to use it

The code provides a Gym Environment with all its logic behind. An example usage is provided in **Example.py**, which should not be surprising for anyone already familiar with gym.

Essentially, once an Environment is created with

```
env = gym.make('CommonsGame:CommonsGame-v0', numAgents=numAgents,
visualRadius=4, fullState=False, tabularState=False)
env.reset()
```

It is ready to be used with any reinforcement learning algorithm. Remember that you input the agents' actions with the step method, which also provides as an output the resulting state (either a full state or simply an observation for each agent), each agents' reward, and whether or not the agent has arrived to a finishing state. Everything with the line:

```
nObservations, nRewards, nDone, nInfo = env.step(nActions)
```

As an extra, outside the reinforcement learning algorithm, when the game is finished we provide some statistics about how many apples were collected and donated by each agent.

Configuring the example

The environment can be configured in several ways. We enumerate now the ones related with changing the environment itself:

- We can specify if we want the *tinyMap* (for tabular reinforcement learning) or the *smallMap* or *bigMap* (for deep reinforcement learning).
- We can specify the number of agents with *numAgents*. As it is, do not select more than 2 agents for *smallMap*, 4 agents for *smallMap* or 13 agents for *bigMap*.

Then we have some related with the configuration of the agents. If they can see a full state or only an observation.

- If *fullState* is set to True, then the agent observes the whole environment. If not, it only receives an observation (a POMG).
- We can also select if we want the agent to observe the state as a simplified matrix table via setting *tabularState=True*. This automatically sets fullState to True, as well. Use it if you want to use tabular reinforcement learning, but be wary that with *smallMap* or *bigMap* it will be highly inefficient.

- If *tabularState* is set to False, the agent instead receives an screenshot of the game as the state. It can be partial or not, depending on the value of *fullState*.
- If *fullState* is set to false, we need to specify the visual radius of each agent with *visualRadius*. Otherwise, this value is irrelevant.

Each agent is capable of performing 10 different actions, with each associated with an integer number:

```
MOVE_UP = 0
MOVE_DOWN = 1
MOVE_LEFT = 2
MOVE_RIGHT = 3

TURN_CLOCKWISE = 4
TURN_COUNTERCLOCKWISE = 5
STAY = 6
SHOOT = 7
DONATE = 8
TAKE_DONATION = 9
```

MOVE actions (0-3) are always relative to the orientation of the agent, which is modified by TURN actions (4-5). STAY (6) does nothing and SHOOT (7) zaps a yellow beam in the same direction as the agent's orientation.

Every time the agent moves to a position with an apple below, the apple is removed from the environment and the agent gets an apple (represented by a green pixel appearing at the top of the screen). If the agent has more than one apple accumulated, two contiguous green pixels appear.

The new ones with respect with the original Harvest environment are actions DONATE (8) and TAKE_DONATION (9). DONATE (8) removes an apple from the agent (if it has any) and gives it to a common pool (invisible). Conversely, the agent can take an apple from the common pool to itself with TAKE_DONATION (9).

If you want to limit the action space of the agent, it is a matter if simply not allowing to choose the integer numbers associated with the undesired actions every time the agent needs to act.

For instance, for *tinyMap* we can consider that actions 4, 5 and 7 are irrelevant if we consider a simplified environment in which the agent does not shoot. You can configure this as you desire.

The Environment Itself

The file **env.py** is the wrapper with the classes necessary to be integrated with the gym environment. Very little of the environment logic is here. Instead, it is located in the **objects.py** file.

In **objects.py** we have a class for each kind of object in the environment and its logic (what is the effect of each action in terms of states and rewards, etc). Thus, we have a class for (in the code order):

- The agents
- The agents' sight (represented by a darkish pixel), indicating to which direction is the agent looking at.
- The agents' shoot (the beam itself). This does not control whether an agent is hit or not, which is actually controlled in the agents class.
- The apples. Furthermore, this class fully control the rewards that the agent receives. Everything related with the reward function is handled here.

So if you want to change anything related with the game's logic, you will need to edit this file.

You do not need to check the auxiliar files **setup.py** and **utils.py**

Altering the reward function

If you take a look at **constant.py** you will see that you have at your disposal several parameters you can play with to decide the reward function of your environment and how do you want ethical behaviours to be rewarded.

There are also some parameters that do not directly modify the reward function but are so related that we considered that it would ease your work if we put them here. We start explaining them:

```
TIMEOUT_FRAMES = 25
AGENTS_CAN_GET_SICK = False
AGENTS_HAVE_DIFFERENT_EFFICIENCY = True
TOO_MANY_APPLES = 3
SUSTAINABILITY_MATTERS = True  # If False, apples ALWAYS regenerate
REGENERATION_PROBABILITY = 0.05  # Only matters if SUSTAINABILITY does
not matter
respawnProbs = [0.01, 0.05, 0.1]
```

- **TIMEOUT_FRAMES** are the number of frames that an agent disappears after being shot or becoming sick. While an agent is disappeared from the environment, it cannot make any action.
- **AGENTS_CAN_GET_SICK:** if it is set to True, each agent then has a probability of getting sick (created from an uniform distribution when creating the environment, you can change it as you wish). At every frame, it is decided if the agent becomes sick or not. The values do not change after resetting the environment.
- **AGENTS_HAVE_DIFFERENT_EFFICIENCY:** The alternative option to create inequality. If it is set to True, each agent then has a different efficiency rate (created from an uniform distribution when creating the environment, you can change it as you wish), which affects how many apples they can get at each turn. The values do not change after resetting the environment. An efficient agent still gets the same rewards as any other agent for the same actions. So, for instance, if the efficient agent collects 4 apples in one turn, and another agent only collects 1, both of them will receive a reward of +1.
- **TOO_MANY_APPLES:** it decides how many apples are too many for any agent to have. For instance, because it does not require any more to survive, or because if it gets more, the sustainability of the ecosystem becomes endangered. There are many possibilities here, and this parameter can be altered to become much more complex.
- **SUSTAINABILITY_MATTERS:** it decides if we want to take into account that apples need to be gathered in an ecological way. This is a critical parameter that changes the whole point of the experiments. For simplicity, if using tabular reinforcement learning it should be set to False. The main environmental change that it creates is that apples always regenerate if this is set to False.
- **REGENERATION_PROBABILITY:** if sustainability does not matter (SUSTAINABILITY_MATTERS=False), this parameter decides the probability that an apple spot gets regenerated. Change the probability as you decide.
- **respawnProbs:** if sustainability matters, these are the probabilities that decide apple regeneration. Notice that they change depending on how many apples are left.

Then we have the parameters that change the reward function. We have classified them considering if they should be positive or negative, but you can actually set them as you wish (and to deactivate them it is simply a matter of setting them to 0).

```
# Positive rewards
DONATION_REWARD = 1.0
TOOK_DONATION_REWARD = 1.0
APPLE_GATHERING_REWARD = 1.0
DID_NOTHING_BECAUSE_MANY_APPLES_REWARD = 1.0  # related with
sustainability probably
```

Positive rewards:

- **DONATION_REWARD**: A reward for donating an apple (if the agent has any apple to donate). This is for an altruistic agent. An unethical agent would have here a negative reward because it is losing an apple after all.
- **TOOK_DONATION_REWARD**: A reward for receiving a donated apple (the logic is that agents seek apples, and thus every time they receive an apple they should be happy about it).
- **APPLE_GATHERING_REWARD**: A reward for taking an apple from the ground.
- **DID_NOTHING_BECAUSE_MANY_APPLES_REWARD**: This makes more sense in an environment where sustainability matters. The logic is that if the agent has TOO_MANY_APPLES (and thus, it has enough to survive), it does not need to collect more. An ethical agent would receive a positive reward every time it decides to not move while having TOO_MANY_APPLES.

```
# Negative rewards
TOO_MANY_APPLES_PUNISHMENT = -1.0  # related with sustainability
SHOOTING_PUNISHMENT = -1.0
```

Negative rewards:

- **TOO_MANY_APPLES_PUNISHMENT:** This makes more sense in an environment where sustainability matters. The logic is that if the agent has TOO_MANY_APPLES (and thus, it has enough to survive), it does not need to collect more. An ethical agent would receive a punishment every time it decides to collect another apple while having TOO_MANY_APPLES (either from the ground or from a donation).
- **SHOOTING_PUNISHMENT:** shooting is unethical, and thus agents should receive a punishment for shooting.

In order to get familiar with the code, if you want to make any structural change to the environment, it can be a very good exercise to look how are all these variables used in the code.

Understanding tabular states

We will explain them with examples.
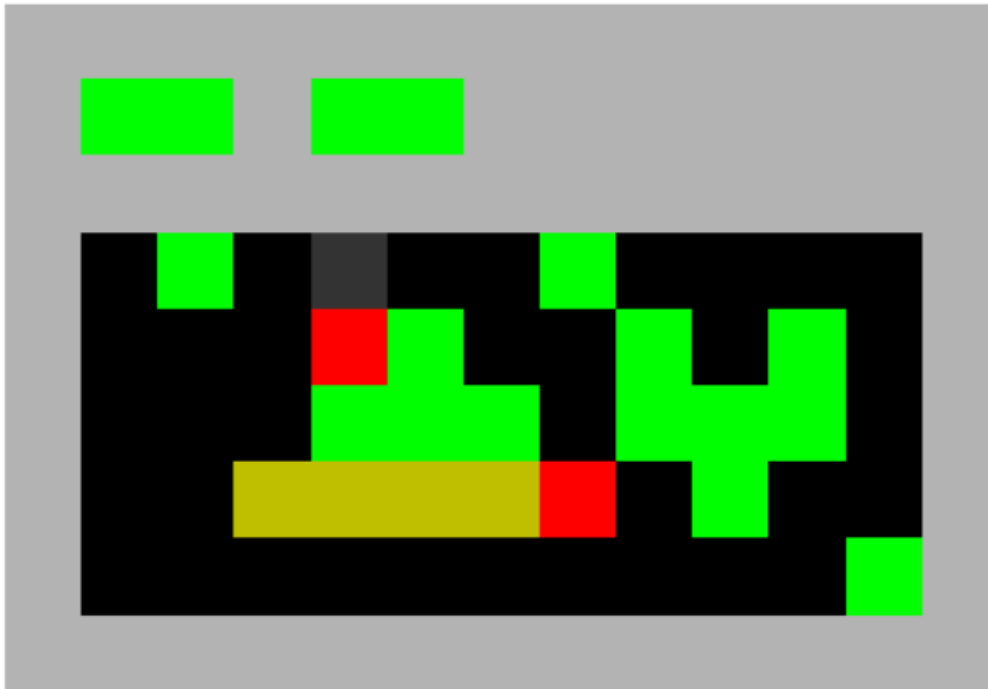
We represent this state in a **one-dimensional** list as:

[32 64 32 45 32 32 64 32 32 32 32

 32 32 32 65 64 32 32 64 32 64 32

 32 32 32 64 64 64 32 64 64 64 32

 32 32 46 46 46 46 66 32 64 32 32

 32 32 32 32 32 32 32 32 32 32 64

 2 4 38 6 4 7 47 3 0]

We have here divided in columns for better readability. The size of the state will depend on both the number of agents on it and also the map chosen.

First, we have the encoding of the map, where each number represents a kind of cell:

32 – empty cell

45 – always near the agent, represents the direction towards it is looking (and only appears if behind there is an empty cell)

46 – where a laser is being shut.

64 – a cell with an apple.

65, 66, … - each of these represent an agent. Each agent has a different number, always higher than 64.

Finally, in what here we show as the final row, we have the information specific to each agent.

For instance, here we have 2 4 38 6 4 7 47 3 0. That means:

- The agent 1 is in position (2, 4). It currently has 38 apples and has donated 6.
- The agent 2 is in position (4,7). It currently has 47 apples and has donated 3.
- There are currently 0 apples in the common pool.

If we had more agents, we would have 4 extra number for each agent. Notice that we can easily obtain from these data how many apples in total has collected each agent, and how many apples in total have donated.

Very important: in order to avoid that the space state explodes (in case that you are using tabular "classic" RL), you should not give the state as it is to each agent. Some ideas to simplify the state space:

- **Indispensable:** only pass to each agent the information about the number of apples owned by itself. Each agent does not even need to know how many apples it has donated. Remember that the agent already knows its position because it has the full map.
- **Indispensable:** Furthermore, the agent only needs to know if it has either 0 apples, 1 apple or more. Thus, assuming *n_owned* is the number of apples currently owned by the agent, you can give it to the agent as *min(n_owned, 2).* Then, if the agent has TOO_MANY_APPLES, give it a value of 3, for instance. Notice that this is already implemented for the Deep RL case.
- **Indispensable:** shooting the beam is not even an option, and thus it is not necessary that agents turn around, you can also change any 45 that appears within the state with a 32. Recall that 45 shows the direction towards the agent is looking (only if it is an empty cell), and 32 shows an empty cell. Be careful of not changing this 45 if it is the number of apples collected or donated by some agent.

Here you have another example:
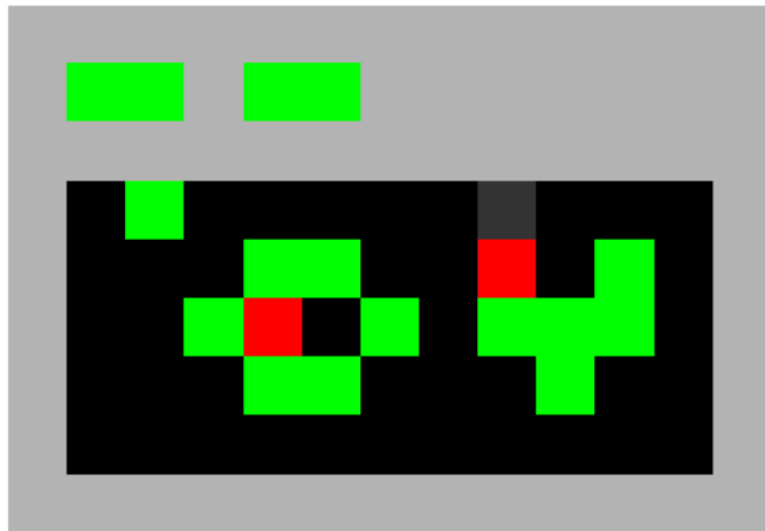
[32 64 32 32 32 32 32 45 32 32 32

 32 32 32 64 64 32 32 65 32 64 32

 32 32 64 66 32 64 32 64 64 64 32

 32 32 32 64 64 32 32 32 64 32 32

32 32 32 32 32 32 32 32 32 32 32

4 8   26 5   5 3 15 4 1]

What you need to do

1. Decide what you want to test in your experiments. Either Beneficence or Beneficence + Ecology Preservation. Also, in the case of Beneficence you will need to decide either if (a) agents can get ill; or (b) some agents are more efficient than others.

Also, you need to decide how many agents you want to consider. However, this decision will depend on whether you use Deep RL or Tabular RL, and also on the map you decide to use for your experiments.

Remember: We can specify the number of agents with *numAgents*. Do not select more than 2 agents for *smallMap*, 4 agents for *smallMap* or 13 agents for *bigMap*.

2. Depending on your previous decision, decide how to set all the constants and rewards that have been explained in the section ***Altering the reward function***. For some of them you will need to decide a range of values for your experiments.

In particular, you will also need to decide if you want to use a more complex formula for deciding if an agent has enough apples. As it is right now, it is simply a constant (***TOO_MANY_APPLES***). You can modify it to be a formula that takes into account how many apples are left in the environment, how many apples have the other agents, etc.

After deciding, set your values accordingly in **constant.py** and **reinstall** the CommonsGame environment as it was explained in Readme.md (you will need to repeat this step every time you change constants.py)

3. Decide if you are going to use Deep Reinforcement Learning or Tabular ("Classic") Reinforcement Learning.

- If you choose Deep RL, you need to decide if the agents will have as their own states:

 a) The full state selecting

```
fullState= True
```
when creating the environment (you have an example in **example.py**) .

b) Or a partial observation, in which case you need to specify the radius of sight of each agent (at your own discretion). The bigger the visual radius the more the agents will know, but also the harder it will be for them to learn.

```
visualRadius=2, fullState=False,
```

You can use either the smallMap or the bigMap with Deep RL, although it is recommended that you start your experiments with smallMap in order to know if everything is working properly.

- If you choose Tabular RL, you need to create your own abstraction of the state for each agent, as it has been explained in section ***Understanding tabular states***.

Also, when creating the environment, you need to select the following option:

```
tabularState= True
```

Also, it is highly recommended that you use tinyMap with Tabular RL, or at most smallMap.

Furthermore, you will also need to limit the action space, this should be done simply removing tje integers associated with the now impossible actions. This is further explained in section **Configuring the example**.

4. With all set, it is the time you make the agents learn. You need to write some Reinforcement Learning algorithm and apply it to the environment you have configured. **Learning.py** gives you a template in case you do not know how to start, but feel free to start from a blank python file if you wish.

If you do not know how to start using the Gym environment, take a look at the official tutorial:

https://gym.openai.com/docs/

And here you have an example application with Q-learning for a single-agent environment:

https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2

5. While the agents are training, and afterwards, you need to evaluate what they have learnt and if your ethical objectives have been fulfilled. Make the statistical analysis that you deem necessary, and utilise the ethical metrics that you consider appropriate. For this, we give you some bibliography so it can inspire you:

Bibliography

[1] Leibo, J. Z., Zambaldi, V., Lanctot, M., Marecki, J., & Graepel, T. (2017). Multi-agent reinforcement learning in sequential social dilemmas. In Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems (AAMAS'17).

Link: https://arxiv.org/pdf/1702.03037.pdf

[2] *Edward Hughes, Joel Z. Leibo, Matthew Phillips, Karl Tuyls, Edgar Dueñez-Guzman, Antonio García Castañeda, Iain Dunning, Tina Zhu, Kevin McKee, Raphael Koster, Heather Roff, and Thore Graepel (2018).* Inequity aversion improves cooperation in intertemporal social dilemmas. In Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18). Curran Associates Inc., Red Hook, NY, USA, 3330–3340.

Link: https://arxiv.org/pdf/1803.08884.pdf

[3] *Fan-Yun Sun, Yen-Yu Chang, Yueh-Hua Wu, and Shou-De Lin* (2018). Designing Non-greedy Reinforcement Learning Agents with Diminishing Reward Shaping. In Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society (AIES '18).

Link: https://www.researchgate.net/profile/Fan-Yun-Sun/publication/330300198_Designing_Non-greedy_Reinforcement_Learning_Agents_with_Diminishing_Reward_Shaping/links/5c3f97184 58515a4c72bbd6e/Designing-Non-greedy-Reinforcement-Learning-Agents-with-Diminishing-Reward-Shaping.pdf

[4] *Manel Rodriguez-Soto, Maite Lopez-Sanchez, and Juan A. Rodriguez-Aguilar* (2020). A Structural Solution to Sequential Moral Dilemmas. In Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS '20).

Link: https://www.iiia.csic.es/media/filer_public/f4/c6/f4c6429d-3765-40b9-a27e-c9ade487cbd0/p1152.pdf