Multiagents

During the lab of multiagents the goal was to create a multiagent prey/hunter environment. As mentioned during the class we needed to use the MultiAgentEnv from rllib. To transform the DQN agent from the previous lab into a multiagent we had to change a lot of things.

# Hunter and Prey Env

At first we created a prey and a hunter instance. These two used the regular rllib environment. Below a short description of both environments.

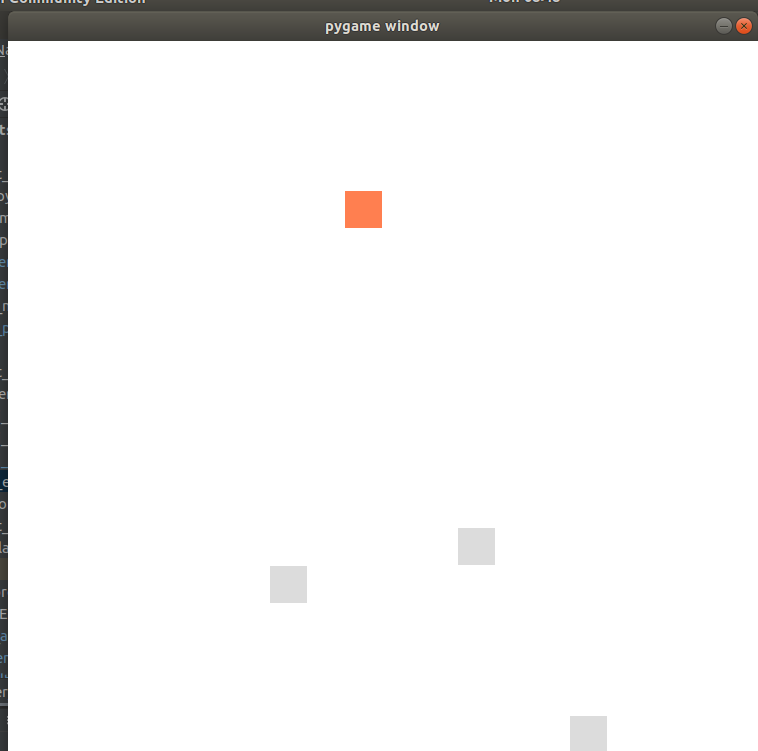
## HunterEnv(Gym.Env)

### Description

The hunter will try to catch preys in order to survive, as soon as the hunter collected enough energy it can reproduce itself. A new hunter is then born and placed on a random position of the board. A hunter is unable to move out of the pre-defined window, if a hunter chooses to move up when it is already on the top of the field this will result in a movement (a loss of energy and age +1) but there will be no position change. REWARD. A Hunter dies when he runs out of energy or when his age has reached the max age defined when starting the training.

|  |
| --- |
| Observation:  *Type: Box(4)*  *Num Observation Min Max*  *0 Age 0 max\_age defined in the simulation\_param file*  *1 Energy level 0 100*  *2 rel x to closest prey 0 width defined in the simulation\_param file*  *3 rel y to closest prey 0 Height defined in the simulation\_param file*  Actions:  *Type: Discrete(5)*  *Num Action*  *0 Reproduce, this is only possible if he has enough energy.*  *1 Move up*  *2 Move right*  *3 Move down*  *4 Move left* |

When rendering the simulation every hunter is represented by an orange dot as can be seen in the picture below:

  
Illustration 1: Representation of a hunter

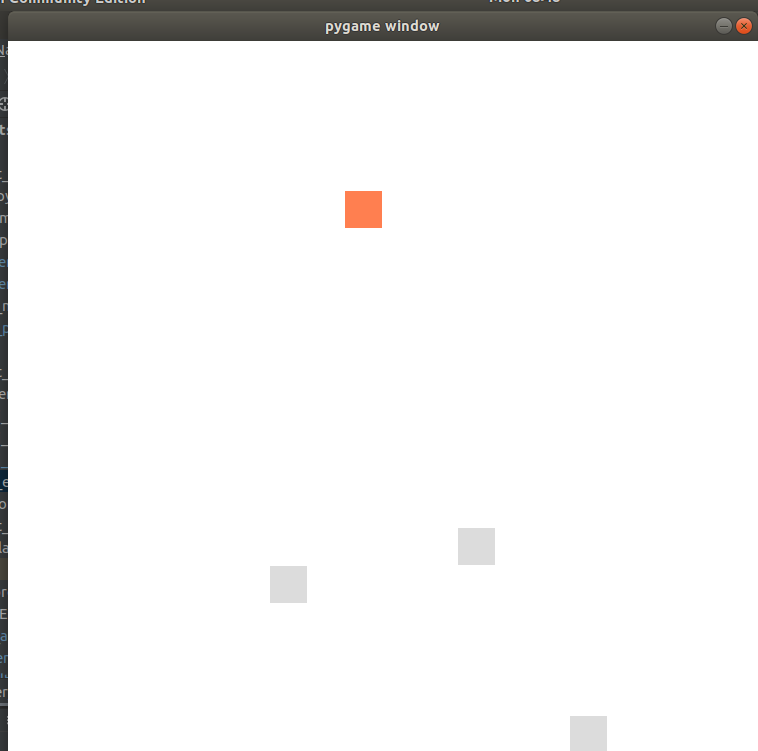
## PreyEnv(Gym.Env)

The prey environment is very similar to the hunter environment, the main difference are the observations and the amount of moves possible, a prey will always have a chance of <birthrate>/100 to reproduce. Once a new prey is born it will be placed randomly.

A prey can only die if it is eaten by a hunter or when it reached it’s max\_age.

|  |
| --- |
| Observation:  *Type: Box(4)*  *Num Observation Min Max*  *0 Age 0 max\_age defined in the simulation\_param file*  *1 rel x to closest hunter 0 Width defined in the simulation\_param file*  *2 rel y to closest hunter 0 Height defined in the simulation\_param file*  Actions:  *Type: Discrete(5)*  *Num Action*  *0 Move up*  *1 Move right*  *2 Move down*  *3 Move left* |

A prey is represented by a grey dot:

  
Illustration 2: Representation of a prey in the simulation window

# MultiAgentEnv

For the multiagent environment we based ourselfs on the [multi\_agent\_cartpole.py](https://github.com/ray-project/ray/blob/master/rllib/examples/multi_agent_cartpole.py) inside the ray/rllib package and on the code presented during the lessons. Our Multiagent environment contains both the hunters and the preys, this makes it easier to train them simultanously.

### \_\_init\_\_(self, config)

In the constructor we fill the list self.agents ans the dict self.index\_map, the list self.agents is straightforward, it contains all the agents used in the simulation. The dict self.index\_map contains all the id’s of the agents and their indexes, it makes it easier to locate the correct agent later.

If we are trying to render the simulation the [“training”] flag should be False, if so our simulation is started.

### reset(self)

When an episode has ended we reset the MultiAgentEnv, when resetting all the agents that were born during the previous training cycle get placed in the self.hunter\_wait or self.prey\_wait lists. Only the original amount of agents is again allowed inside these lists. By placing the newly generated agent in the waiting lists we can reduce the amount of unused classes and recycle them later.

### step(self, action\_dict)

The step function is rather complex, since the observations of every agent contain te closest prey/hunter according to there own type, we need a function to find the closest agent.

1. Create two maps, containg the functions of the preys and the functions of the hunters. This is done by looping over all agents and requesting their position.
   1. This function is also used to count the amount of hunters\_living, preys\_living, hunters\_total, prey\_total.
2. Find the closest agent of every agent and perform a step function for every agent. In this part of the function we also verify if there still are hunters and if not, we set done[i] in which I is the id of the agent to true.

|  |
| --- |
| for i, action in action\_dict.items():  if "hunter" in i and amount\_of\_preys\_living > 0:  dist = find\_closest(self.agents[self.index\_map[i]].get\_position(), prey\_loc)  if "prey" in i and amount\_of\_hunters\_living > 0:  dist = find\_closest(self.agents[self.index\_map[i]].get\_position(), hunter\_loc)  obs[i], rew[i], done[i], info[i] = self.agents[self.index\_map[i]].step(action, dist)  if amount\_of\_hunters\_living == 0:  done[i] = True  if done[i]:  self.dones.add(self.index\_map[i]) |

1. Reproduce, as mentioned before we ‘recycle’ our agents. If a living agente performs it’s step function it will return info about the reproduction in it’s ‘info’ field. During the same loop as in the previous snippet of code we first verify whether the amount of hunters living > 0 and whether the agent which is tryiing to reproduce is still alive. If these requirements are forefilled we can reproduce the agent. In the code snippet below the reproduction process of the hunters is shown. At first we check whether there still is an agent in the waiting line. If so, we add him to the self.agents list. If the hunter\_wait list is empty we create a new agent. The last step of the reproduction process shown in this figure is to choose a unique id. Not shown in the picture are the parts in which the agent is added to the observations, rewards and other lists.

|  |
| --- |
| if self.agents[self.index\_map[i]].type == "hunter":  if len(self.hunter\_wait) > 0:  self.agents.append(self.hunter\_wait.pop())  else:  self.agents.append(HunterEnv(self.config))  id = self.agents[len(self.agents) - 1].type + "\_" + str(amount\_of\_hunters\_total)  n = 0  while id in self.index\_map:  n += 1  id = self.agents[len(self.agents) - 1].type+"\_"+str(amount\_of\_preys\_total + n)  amount\_of\_hunters\_total += 1 |

# Training process

Both agent types have there own policy and dqn model, this is done because they will show very different behaviour between the two types but the agents of the same type will be able to perform parameter sharing. In order to train the seperate agent types we added a [“training”] parameter to the config file. When we are tryiing to train a specific type we can set this to true, else it will just return random values.

Before we started training the classes we set both training values to false, the policies would both return random values and thus we have a comparison to see if our agents are learning.

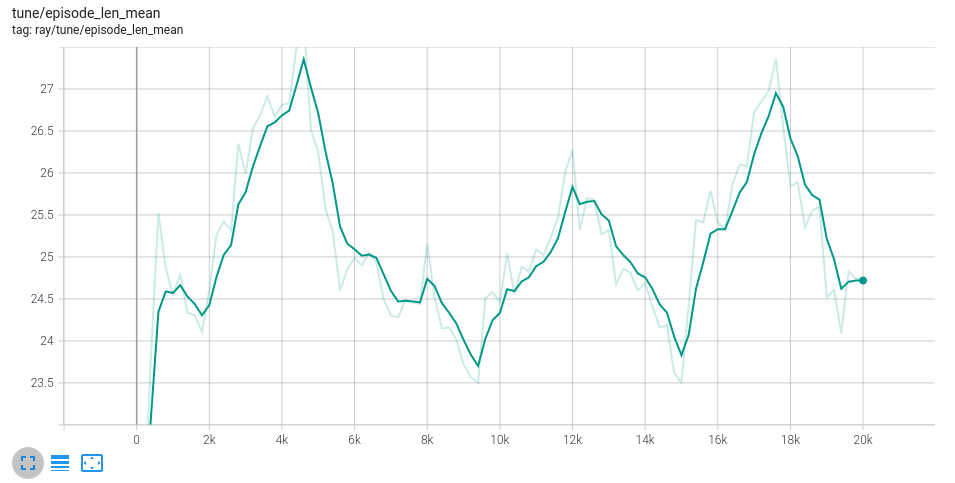
For the training we used the following config:

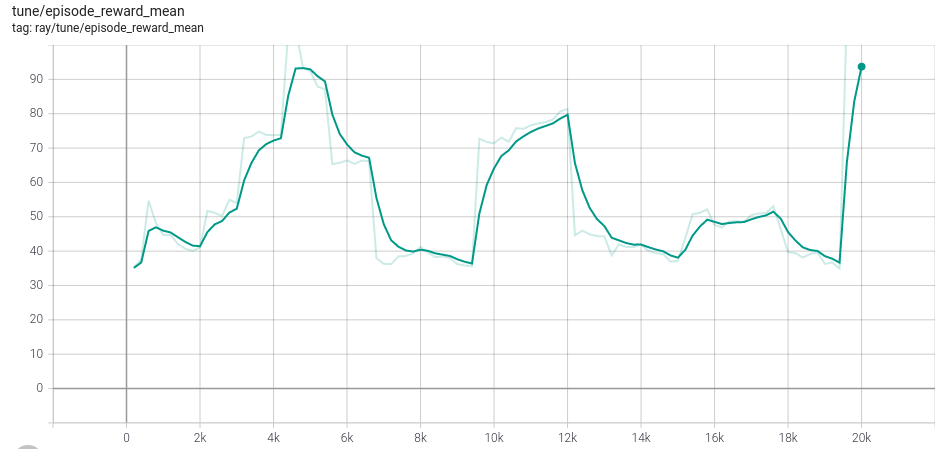
|  |
| --- |
| env\_config = {  'num\_hunters': 1,  'num\_preys': 2,  'training': True,  'hunters': {  'start\_amount': 1,  'energy\_to\_reproduce': 30,  'energy\_per\_prey\_eaten': 10,  'max\_age': 20, },  'preys': {  'start\_amount': 1,  'birth\_rate': 7,  'max\_age': 20},  'sim': {  'width': 10,  'height': 10}  } |

## Training with random values:

After training with random values only this was the result,

|  |  |  |  |
| --- | --- | --- | --- |
| reward | Episode reward max | episode\_reward\_min | episode\_len\_mean |
| 108.88 | 7529 | 20 | 24.72 |

  
Illustration 3: Mean\_len after trainin random values

  
Illustration 4: epsidode\_reward\_mean after training with random moves