Comparing AI Algorithms within a Multi-agent, Partially-Observable Survival Game

Kaden Kilburg
COM S 572: Principles of Artificial Intelligence
Iowa State University

Abstract

How directly applicable today's AI methods are to producing life-like behavior is an interesting question. We present an analysis of policy training algorithms within a multi-agent, partially-observable survival game. This game is custom-built to provide an environment that reflects common aspects of a real-life environment, such as resource scarcity and multiple actors.

Introduction

Artificial Intelligence is an increasingly popular subject of today's world. One of the main predictions concerning it features AI taking employment opportunities followed by a rise of superior beings which enslave mankind. A person might ask when hearing such views "How likely or unlikely is this scenario, and if so, when?" Such a question has no widely-accepted answer today. However, the emergence of complex behavior in life-like environments may be a good origin point for such an AI.

While the purpose of this report possesses no intention of attempting to create such a superintelligence or even highly-complex behavior, it does propose a study of various algorithms within an environment with aspects directly relatable to life on earth. In doing so, this report outlines the construction of a survival-based, partially-observable environment featuring multiple agents. The report then compares multiple policies and training methods within said environment.

Background and Related Work

Neural MMO is a multi-agent survival game inspired by MMORPGs (Massively Multiplayer Online Role-Playing Games), a video game with high levels of interaction between players and extensive game mechanics. It originates from OpenAI, where interns developed it in 2019, and it is now maintained and updated by one of those interns, Joseph Suarez, now a Ph.D. student at MIT. Neural MMO now hosts a great many features more than those displayed in its original paper (Suarez et al. 2019). Originally, Neural MMO was used to study to test the impact of the number of agents in a survival environment alongside the side of the environment. After training policies by themselves, agents using the various trained policies were placed within the same environment, and the performance of the different policies trained were analyzed through the results of these interactive episodes.

Neural MMO's environment consists of a tiled-based map where agents can move in the four cardinal directions, tiles which function as walls to present a form of a maze, agent health demands to encourage resource gathering status, a combat system allowing agents to steal food from other agents, and more minor details concerning the survival of an agent (Suarez et al. 2019).

Environment

The environment used in this report was custom built and reflects many aspects of the Neural MMO project discussed in Background and Related Work. Our environment hosts a multi-agent survival scenario with random map generation, resource gathering and demands, and combat.

Beginning with the environment map, it is laid out as a tile-based game. These tiles include null tiles for the border, grass tiles for empty tiles, water for tiles containing water, forests for tiles containing food, and depleted forests for forests that no longer have food. The tile grid is generated randomly using an algorithm that begins with an empty 16x16 grid, and then a small number of resources are placed randomly on the grid. From there, patches of resources are grown by filling randomly-picked adjacent tiles with identical resources. This was designed to give agents the opportunity to perform well locally but require exploration to use nonlocal resources. With the tile grid set, eight agents are randomly placed in occupiable tiles of the map. Nonoccupiable tiles include null and water tiles as well as tiles containing other agents. A world view of the environment can be seen in Figure 1.

The goal of our environment is for agents to survive as long as possible within their environment. At each ply for an agent, their food and water is reduced by one to a minimum of 0. If an agent's food or water is already 0, their health is reduced by one for each depleted resource. When an agent's health reaches 0, they are considered dead and are removed from the environment. Upon death, that agent is given a reward of -1 to discourage such events. One turn within the

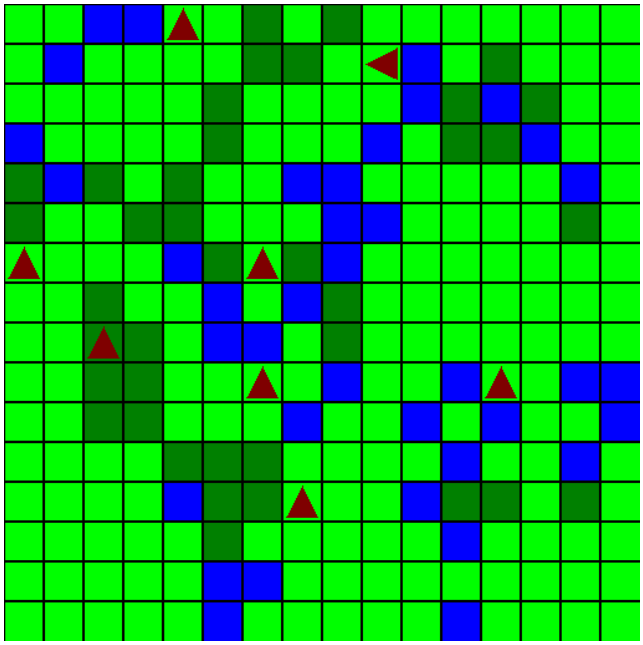


Figure 1: Screenshot of the full environment.

environment occurs once all agents have sequentially taken an action. This agent order remains constant for the entirety of the episode.

Each agent is given a partially observable view of the environment. This view consists of all tiles within a 5-tile radius square of the agent (see Figure 2) and excludes the status information of other agents. An agent's observation is received as a dictionary with four elements. Three of the elements are 11x11 matrix cutouts of the environment where the center square contains the observing agent. These matrices correspond with a grid of each tile, which tiles contain dropped food, and which tiles contain agents, respectively. The fourth element of an agent's observation is a tuple of the observing agent's health, food, and water, respectively.

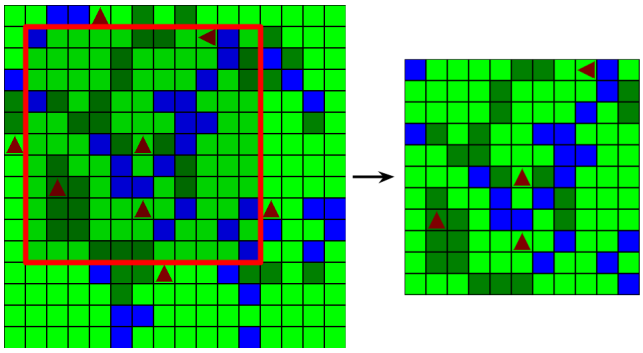


Figure 2: Area observed by one agent.

To influence the environment, an agent must take an action. An action consists of a three-element tuple conforming to the MultiDiscrete([2 4 7]) space. The discrete subactions correspond with movement (1 for move, 0 otherwise), direction (0-3 corresponding to different cardinal directions), and task. There are 7 possible tasks an agent can perform within their action: "none", "drink", "consume ground", "consume carry", "pick up", "drop", and "attack". For the "drink" action, an agent may drink from an adjacent water tile it faces, which replenishes its water status. By using "consume ground," the agent depletes a forest from their current tile and replenishes their food status if that agent is on a forest tile. Alternatively, this action works with dropped food as well. Dropped food can occur by first using the "pick up" action, which does not consume food but does deplete that food from its tile. The agent can then carry the food with them but can never carry more food until they are no longer carrying the food. Carried food may be consumed by using the "consume carry" action, which functions similarly to its ground counterpart, and carried food may be dropped by using the "drop" action, using the "attack" action, or upon death. Consuming food or drinking water restores 5 points to their respective status, but each status has a maximum of 10 points.

Architecture and Training

Our environment was built using the PettingZoo framework (Terry 2021), which provides support for multi-agent environments and interface capabilities with major reinforcement learning libraries such as Ray RLlib (Ray). The framework allows external programs to interface with the environment by first retrieving an observation o_i in the current state s_i, $O(s_i) \rightarrow o_i$, feeding this input to a policy π which generates an action $\pi(o_i) \rightarrow a_i$, and stepping through the environment with this action to get an state $S(s_i, a_i) \rightarrow s_{i+1}$. (i.e. one ply).

The policy trainers provided by Ray RLlib and used in this paper include the following: Behavior Cloning (BC), Policy Gradient (PG), Proximal Policy Optimization (PPO), PPO with an LSTM layer, and PPO with an AttentionNet wrapper. Additionally, three scripted agents were designed for testing: one which remains idle by taking null actions (zero movement with the "none" task), one which takes random actions, and one which searches all actions for the next $n = 4$ turns.

The policy for this search agent simulates a new environment using the observation space as the new environment. It then proceeds to take n actions, after which it evaluates its resulting state. This evaluation heuristic uses its negative squared difference in each max status to its current status field to increasingly penalize larger differences from the max status. Additionally, the heuristic multiplies the food and drink penalties by the agent's minimum Manhattan distance to a tile of that resource.

With a set of policies to test, the last step was to train the reinforcement learning models. For each of the policy trainers defined above, they were given a fixed number of epochs to train within. These epoch limits were defined based on run-time observations and project time constraints. These results are discussed in the following section.

Results

Having trained policies using each training algorithm, the impact of each policy on an agent's goal (longer lifespan) can be analyzed. Figure 3 presents the plots for each policy training algorithm tested. Each plot consists of two lines where each one shows to mean of that trait measured within a batch of episodes. The two traits measured consist of the average and the max lifespan within a given episode.

The first two algorithms tested were afforded significantly more epochs for training due to their speedy run-time and to monitor if unexpected improvements could be made with enough training. From Figure 3a, it is apparent that Behavioral Cloning (BC) shows no improvement of agent lifespans. Figure 3b shows that Policy Gradient (PG) training results in marginal improvement past its initialized policy. One interesting to note about this training sample is the unusually large dip in the middle of training which it then recovers from.

In contrast with BC and PG, Proximal Policy Optimization (PPO) leads to substantial improvements from its initialized model as seen in Figure 3c. A similar trend is also visible in the two extensions of PPO tested.

After plotting the lifespan changes observed during training, an additional table, Table 1, was computed, displaying the final results of training alongside the scripted policies. More specifically, the lifespan statistics shown for the reinforcement learning methods are the observed values after each training method's final epoch. For the scripted policies, these statistics represent the mean results after a batch of episodes with batch size 64. This batch size was selected to match the batch size used in the training phase.

Discussion

Conclusions

This results of this report shows that PPO is the most performance training algorithm tested for our environment. This is in contrast with its default extensions using an LSTM layer or an AttentionNet, which run significantly slower yet produce similar results, and in contrast with BC and PG which produce nonexistent or minute results.

With that said, it was surprising to see the scripted search algorithm vastly outperform the best reinforcement learning (RL) method tested, PPO. Granted, the execution of the search algorithm was significantly longer than that of the RL methods, and it was custom designed as opposed to the out-of-the-box implementations from RLlib. This shows just how performant a well-tailored algorithm can be.

Future Work

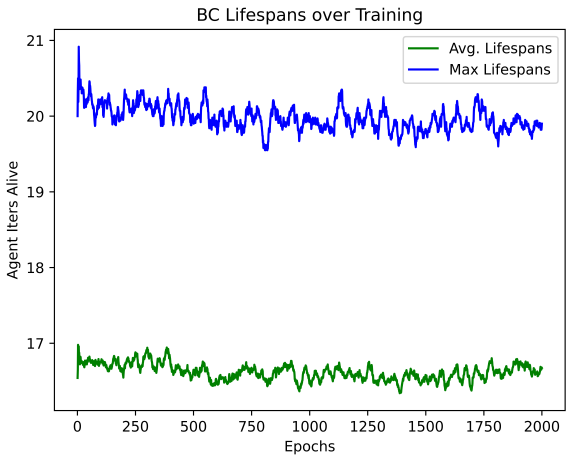
By viewing live runs of the various policies, there are obvious flaws in each of them that could be improved upon. This includes the scripted search policy which very clearly shows bugs such as taking the null action as opposed to consuming nearby resources when an agent's status is low. Additionally, a better evaluation heuristic could greatly improve the policy along with a more strategic action-space search to speed execution time. Cleaning up these current implementations should be prioritized before expanded the environment in more complex ways.

References

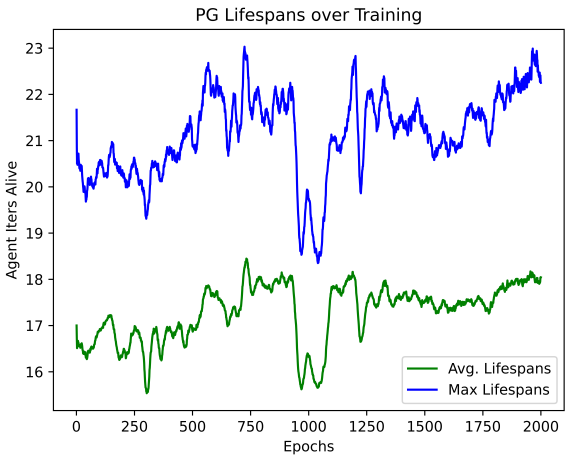
Ray. RLlib Algorithms.

Suarez, J.; Du, Y.; Isola, P.; and Mordatch, I. 2019. Neural MMO: A Massively Multiagent Game Environment for Training and Evaluating Intelligent Agents.

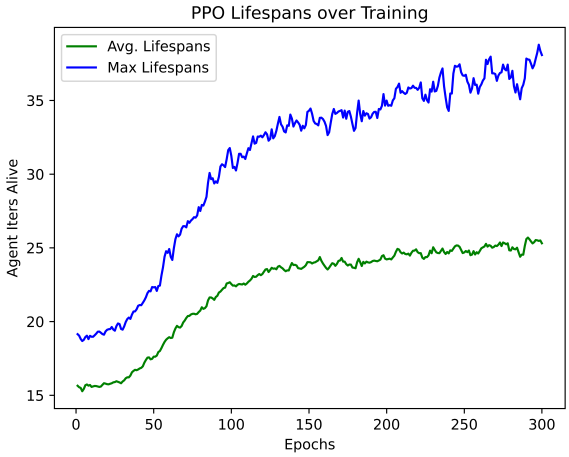
Terry, J. K. 2021. Using PettingZoo with RLlib for Multi-Agent Deep Reinforcement Learning.



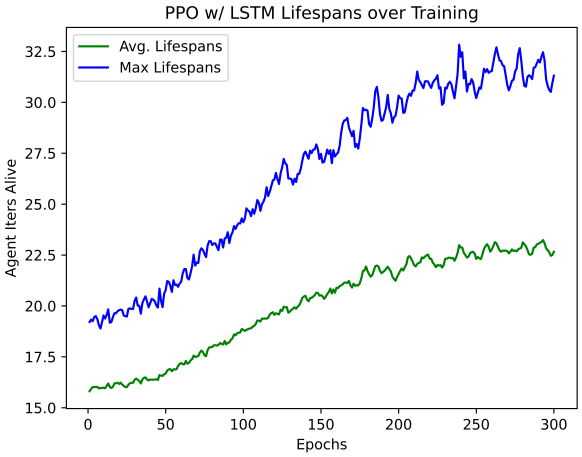
(a)



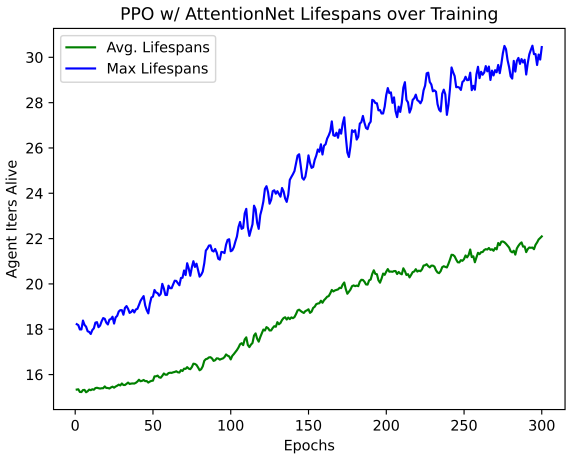
(b)



(c)



(d)



(e)

Figure 3: Agent lifespan changes during training phase.

Algorithm	Epochs	Time	Avg. Lifespan	Max Lifespan
Idle			15	15
Random			16.04	18.82
Search			31.31	84.2
BC	2000	0:15:37	16.67	19.9
PG	2000	0:10:47	18.04	22.25
PPO	300	0:20:46	25.31	38.08
PPO w/ LSTM	300	2:47:51	22.66	31.31
PPO w/ Attention	300	1:24:05	22.10	30.45

Table 1: Final training results.