Predator/Prey Simulation with Reinforcement Learning

By Arlo Rostirolla

Table of Contents

List of Figures

Abstract

The aim of this project was to attempt to implement artificial intelligence algorithms in our 2D top down covid simulator game. We attempted Neuro-Evolution of Augmented Topologies (NEAT), Soft Actor-Critic (SAC), and Proximal Policy Optimization (PPO). The problem was simplified, and agents were trained over a gradually increasing level of difficulty. Inputs were gained by 3 raycast sensors, tuned to either obstacles, the ACA, or civilians. Both SAC and PPO learned to collect civilians, however only PPO was effective at also learning to avoid the ACA. Neuro-evolution using NEAT was unsuccessful at learning either behaviour.

Introduction

For this assignment, the main techniques investigated were Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC) reinforcement learning using the Unity ML agents framework. The Neuro-Evolution of Augmented Topologies (NEAT) algorithm was also explored, albeit to less successful results. We used our original game from assignment 1 as the environment for these experiments. The differences being that there was no longer human control. The AI controlled the player, and all other agents were controlled by their original assignment 1 implementations. The problem was simplified, and gradually made more difficult as a manual implementation of curriculum learning.

Problem Formulation

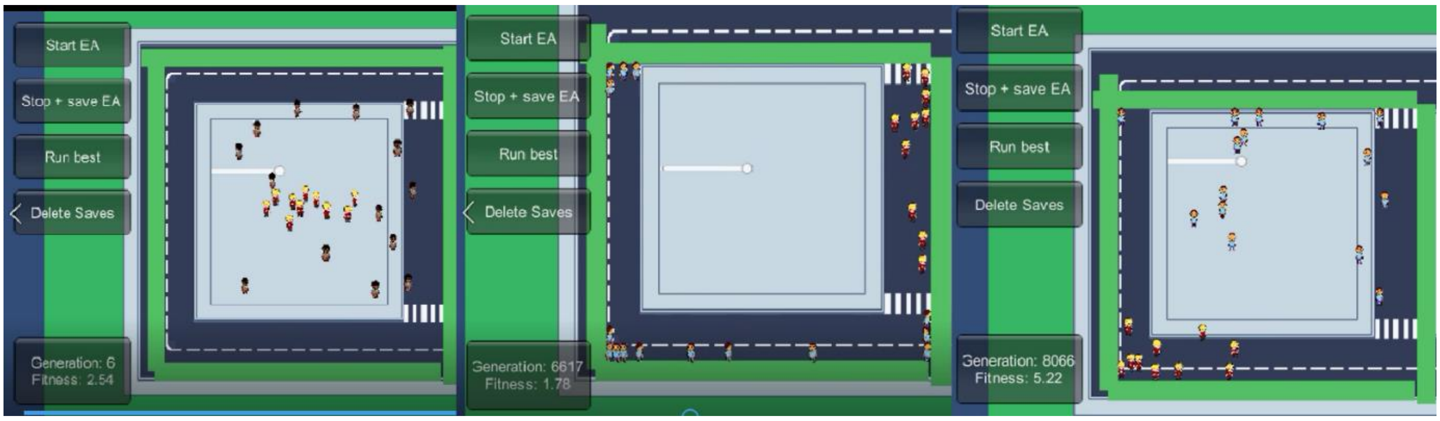
At its most basic level, with continuous actions and a continuous location space, the state space of our game would be infinite. When discretized into a grid with S nodes, and N agents, the resulting state space is N^S. In the case of our game, this is 44^{1875} or approximately $3.00e^{3081}$. If we consider that agents can die, then the state space is $N^S + N\text{-}1^S + \ldots + 4^S$ (there are four agents that cannot die without the game ending). This number is far too big to calculate. Instead of relaxing our problem representation to comply with simpler RL techniques such as tabular Q learning, we decided to use neural networks as a reward function approximator.

Neuro-evolution of Augmented Topologies

We attempted to incorporate the NEAT algorithm into our game using an C# implementation by github user flo-wolf [1]. The inputs to the algorithm were defined as the XY locations of civilians, and a float distance measurement. This was implemented by looping through all nearby civilians and adding their x coordinate, y coordinate, and distance to the input sensor. Outputs were 8 continuous variables corresponding to the eight available discretized directions. Initially, this was set to the maximum output from the algorithm. After a lack of success with this, a softmax function was used to make the results more probabilistic. The evaluation function was simple to design, being the number of civilians infected in an episode. For training, multiple agents were spawned in the same simulation, and for each civilian that is infected, another was spawned to keep their population constant.

Initial runs demonstrated little to no evolution. An attempt was made to implement novelty search, by creating a list of paths taken, and rewarding phenotypes which took novel paths. This did not improve the learning and was removed. The original code was modified to change the time scale, and to fix a bug with the stop + save EA button, which did not save the genome. Although the genome was saved, it was quickly evident that the algorithm was not improving. Figure 1 shows the results of these attempts.

Figure 1. *Results of the NEAT algorithm run over multiple generations*



Analysis of the results showed that none of the output probabilities were reaching more than 0.20, leading to highly random behaviour. A sanity check was conducted, by changing the reward function to be equivalent to the number of upward actions taken. After 8000 generations, it was evident that the agents were taking the up action more than others. However, given the more difficult problem, they failed. Given the difficulty and lack of customizability of the implementation, this attempt was abandoned.

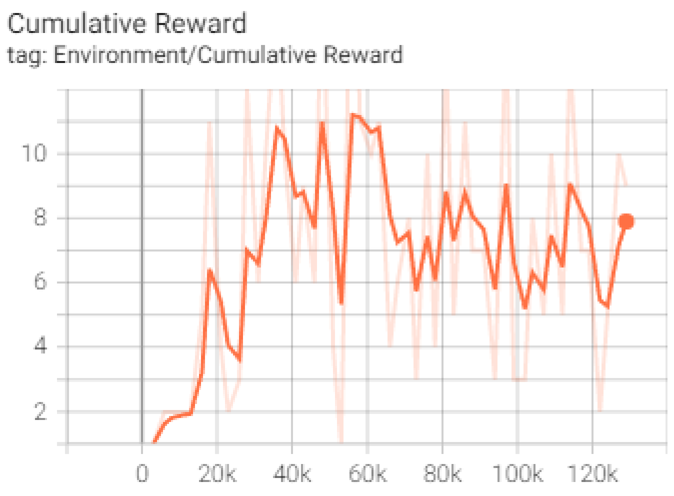
Reinforcement Learning

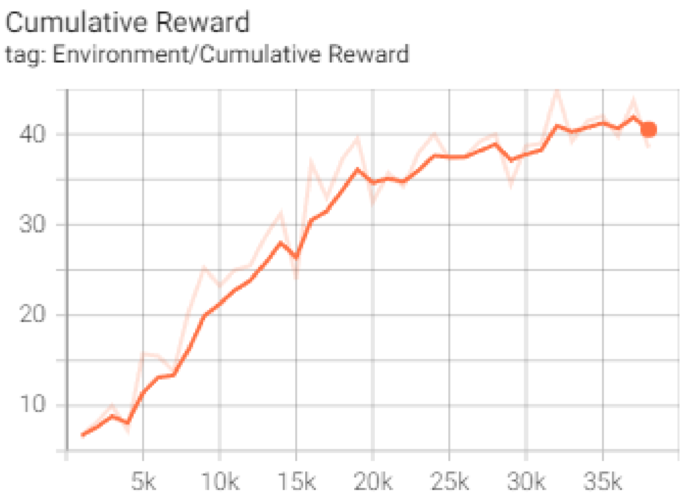
Soft Actor-Critic

The soft actor critic algorithm was implemented to test the differences between it and the PPO algorithm in unity ml agents. Initial attempts were unsuccessful as can be seen in Figure 2a. However, after increasing the gamma parameter, increasing the buffer size, and lowering the batch size, the agent quickly learned to increase its reward over the course of ten minutes. The results of which can be seen in figure 2b. The agent appeared not to follow a strategy other than moving towards civilians.

Figure 2. *Tensorboard graphs for the initial attempts at implementing SAC without enemy agents.*

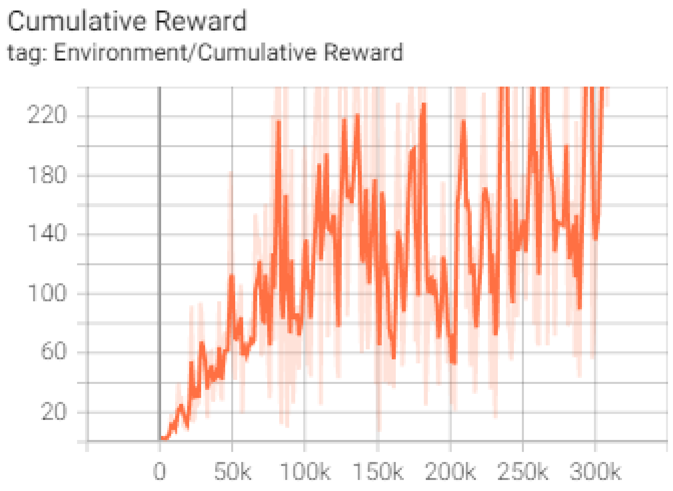
(A)First attempt *(B) Second attempt with different hyperparameters*





After having success with the hyperparameters chosen, an attempt was made to make the problem more difficult to test the limits of this algorithm. Initially, an anti-covid agent was introduced, and the map was increased to be the initial size of our assignment one implementation. This caused the reward to be far more variable, which was highly correlated with the average trial period as can be seen in Figure 3/Figure 4c/Figure 4d. This was to be expected given the correlation between surviving longer and obtaining a higher reward. The summary frequency was increased from 1000 to 5000 steps to smooth out the graph.

Figure 3. *First attempt at introducing an anti-covid agent with SAC algorithm*



This agent learnt an interesting strategy. It tended to loop around the ACA agent, either in a circular motion or figure of eight; this allowed it to sweep through and collect rewards while ensuring that the enemy agent did not get close. This strategy is visualised in Figure 4a/4b.

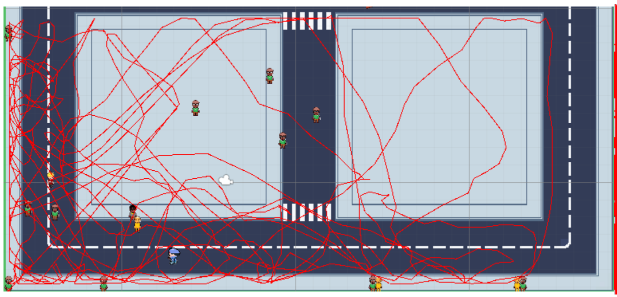
During this experiment there were many times that training had to be restarted due to unity crashing. It became noticeable that whenever the training was restarted, performance dropped rapidly before slowly increasing again. In the entropy graph, the entropy spiked, and policy loss dropped every time this occurred (See figure 4e/4d). Given this, an attempt was made to make the unity environment more stable so that learning could continue without interruptions. Setting threading to false, and exporting the game to an executable that could be run directly from ML-Agents fixed the crashes.

Although this was successful, once obstacles were added, learning plateaued. Given previous success with PPO, and the fact it could be restarted without impacting training, the algorithm was swapped, using hyperparameters that had been successful tuned to work in a traffic simulation. Interestingly, learning occurred much the same as in the other domain (See Figure 5).

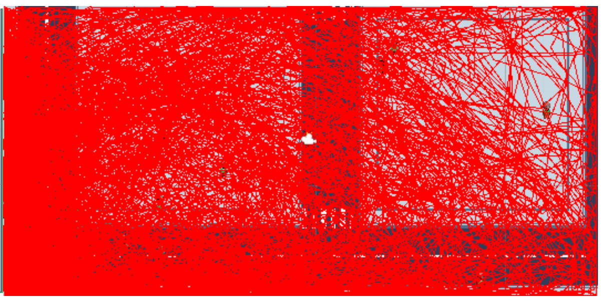
Figure 4.

Illustration of the path taken by the agent over a single episode (a) vs 10 episodes (b); Plot of reward over time (c) and plot of episode length over time (d) for first attempt; Entropy graph (e) and policy loss (f) for second attempt.

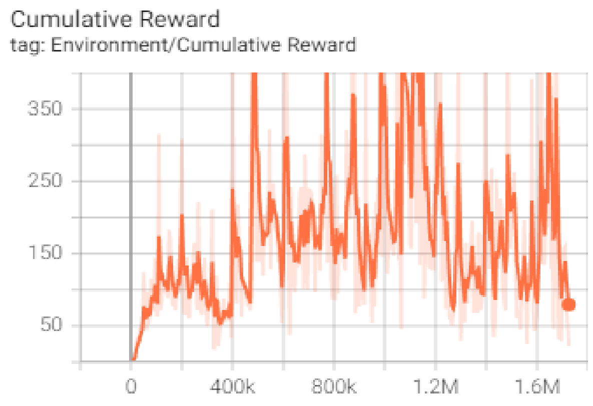
(a)



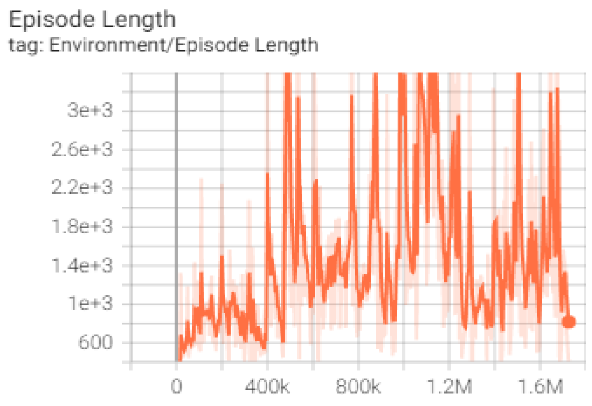
(b)



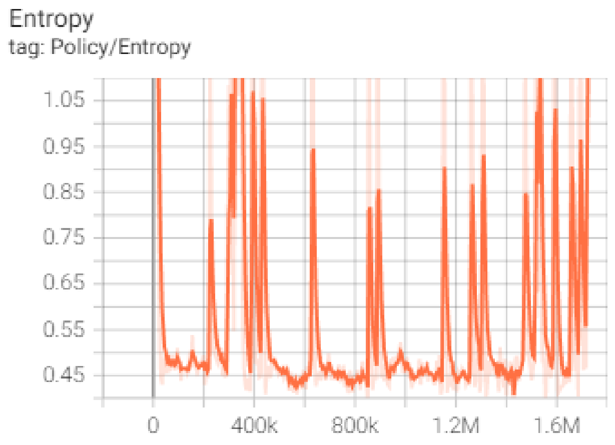
(c)



(d)



(e)



(f)

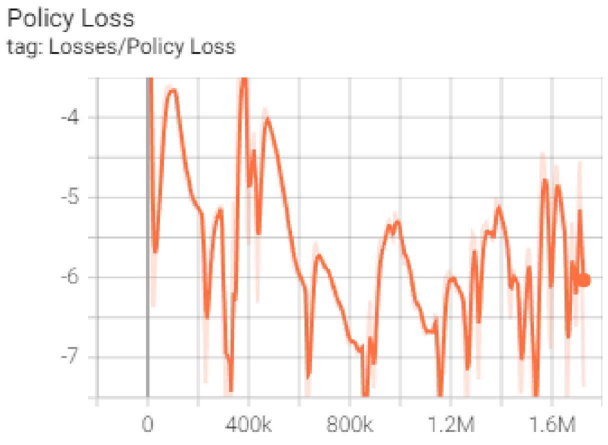
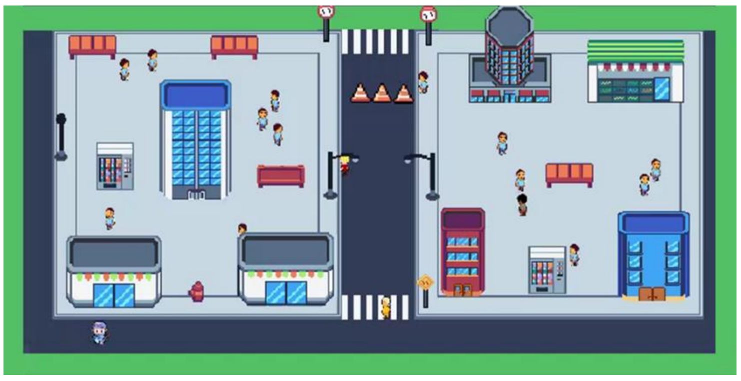
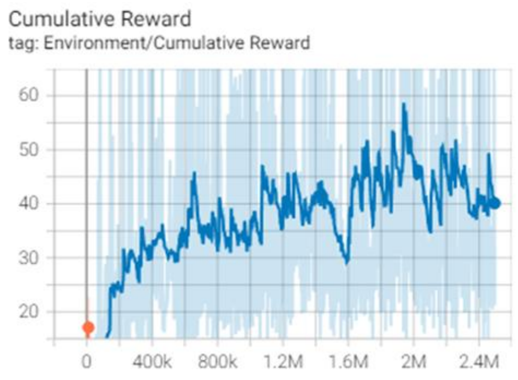


Figure 5. *Final map, along with cumulative reward over 2.4 million steps (PPO)*

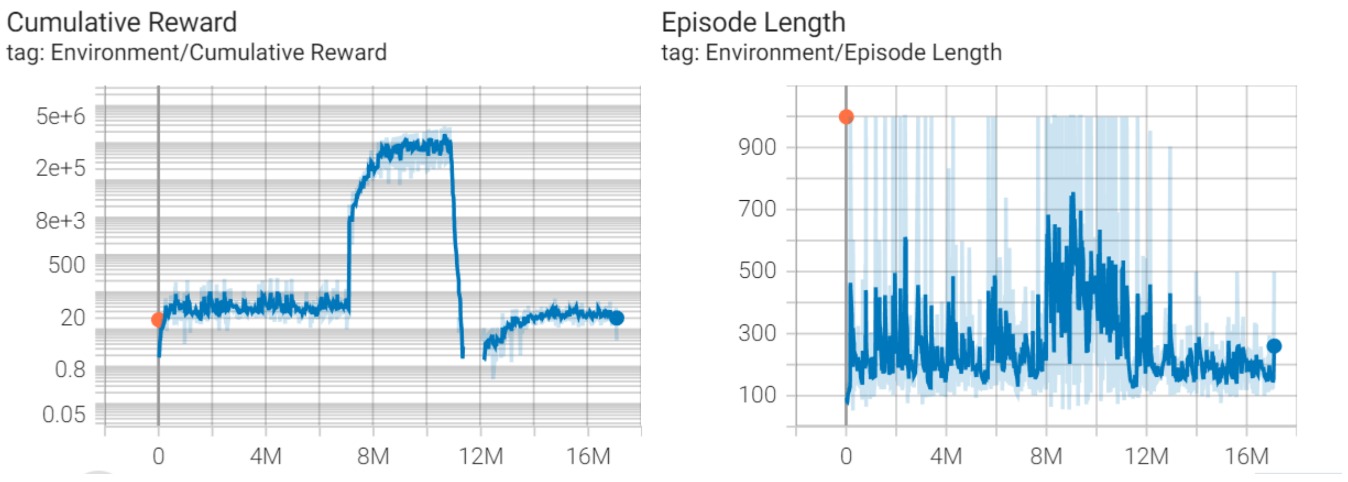




Proximal Policy Optimization

The agent had learned to collect civilians and avoid obstacles, however did not avoid the ACA. No punishment had been set for colliding with the ACA, as it was assumed the lesser reward resulting from a shorter episode would be enough. Two attempts were made to fix this issue. First, a “surviving reward” was added, such that a reward of one was added for every second spent existing. This did increase the episode time, along with dramatically increasing reward, as can be seen in Figure 7.

Figure 6. *Cumulative reward and episode length at 16 million steps, with introduction of survival reward at step 7 million, and capture punishment at 12 million.*



Upon inspection of the simulation, it was evident that the agent had learned that hiding in small corners where the ACA could not go could give it more reward than actively collecting civilians. This did not reflect the ideal global optimum reward, which is an episode where the agent collects as many civilians as possible while never colliding with the ACA. This feature was removed, and instead a reward of -20 was given upon contact with the ACA. This was reduced to five when it was noticed that the mean reward was negative. This can be seen after 12 million steps in Figure 6. Additionally, the decision period was increased to 20, given that the agent had unnaturally sharp and fast movements. The agent relearnt to collect civilians, and was effective at avoiding the ACA, however it did not always successfully avoid it, and often the trials ended early.

Discussion

It was evident from the results that RL was far more effective than NEAT. The RL algorithms took minutes to learn problems that NEAT couldn’t learn even over the course of a day. While it is likely that the NEAT implementation was not ideal, RL seemed to be the more optimal choice for this environment, both with regards to both ease of use and effectiveness.

The most difficult aspect of implementing RL was tuning the hyperparameters, both within Unity, and within the configuration file. Initially for SAC, the action decision time was set too fast, at 50 actions per second. This meant with the gamma reduction, future rewards were not contemplated at all. After changing this to once every 10 frames, learning became far more stable. It was interesting to note that more success was gained by decreasing hyperparameters than increasing them. Attempts to increase the time horizon, num layers, hidden units, batch, and buffer size did not result in improved learning. In fact the simplest configuration, with 1 layer, 128 hidden units and batch size of 128, which had been used with the traffic light control domain, was the most effective.

It was also interesting to witness ‘cheating behaviour’, where the agent would find a hiding spot to earn rewards for living. Although not ideal, it meant the agent was learning. It was just learning the wrong thing. With SAC, it would have had a negative impact on training to stop and restart. However with PPO, training could be stopped and started to customize expected behaviour and implement a form of manual curriculum learning.

References

[1] https://github.com/flo-wolf/UnitySharpNEAT/tree/main/Assets/UnitySharpNEAT