Adversarial Environment Generation for Multi-Agent Algorithm Evaluation

Kavya Kopparapu* **Eric Lin*** **Lucy Liu***

[1]Harvard University Department of Computer Science
Cambridge, MA, 02138

Abstract

Multi-agent algorithms are increasingly becoming a staple in the real world, solving complex tasks through coordination. The complexity of these algorithms' behaviors also makes them less interpretable than their single-agent counterparts. Therefore, there is a significant need to understand their behavior, potential edge-cases, and failure scenarios in a variety of environments. Intuitively evolving these adversarial environments that are "difficult" for a group of multi-agent algorithms and iteratively improving on their implementation based on identified weaknesses is the focus of this paper.

We present the first implementation of adversarial environment generation for multi-agent algorithms. Using a genetic algorithm (GA) for environment generation enables easier interpretation of swarm and deep Q-learning algorithms and illuminates their failure scenarios. We also propose and demonstrate a framework for the continued improvement of multi-agent learning algorithms through co-evolution. We implement an environment simulator representing a foraging scenario with baseline `RandomAgent`, ant foraging-inspired `SwarmAgent`, and deep Q-learning `DQNAgent` algorithms. Experiments show that our Genetic Algorithm successfully evolves environments which consistently reduce foraging to less than an average of 0.3 food particles collected per agent [1].

Introduction

The rising adoption of multi-agent algorithms in the real world has transformed our way of life. Going forward, we are likely to see such algorithms increasingly employed for tasks like navigation, supply chain logistics, and decision making. However, safely using an algorithm in the real world, especially when the setting involves physically interacting with humans, requires a deep understanding of the algorithm's weaknesses and limitations.

Today's algorithms are complicated and computationally-intensive, making it difficult to fully understand the basis behind their behaviors and in which scenarios they may fail. The interpretability of an agent can be broken down into two components: the interpretation of the algorithm deciding the agent's behavior and the interpretation of the agent's behavior in a specific environment. While neither is readily interpretable for most state-of-the-art single-agent algorithms, the latter is significantly more difficult for multi-agent algorithms, whose behavior in an environment is not only reliant on the implementation of their internal algorithm but is also influenced by other members of the population. Even though we may consider all the agents in an environment as part of a single multi-agent swarm, the behavior of the swarm is quite complex compared to a single individual.

Therefore, a methodology with which researchers can evaluate the performance of their multi-agent swarms through generating difficult examples that the swarms perform poorly on would be very advantageous [2]. These fail-cases, difficult examples, and edge-cases can not only be used for validating and testing performance but also then be used to improve the implementation of the swarm agents to account for the realized poor performance. To further showcase the utility of this methodology, we chose an environment and application– multi-agent foraging– that highlights a significant coordination problem between agents that can be uninterpretable without specially-designed examples. This forms the motivation of our work in generating adversarial environments for multi-agent programs.

Background and Related Work

Adversarial Learning

Improving an advanced AI program already trained on a large amount of data is difficult. Often, it requires finding increasingly specialized training data or training settings which the model cannot succeed on yet. Such "challenge settings" provide information about an algorithm's weaknesses and limitations. Continuing to train or modify algorithms in response to the challenge settings can produce more robust models. Even if the algorithms cannot be improved to succeed on the challenge settings, simply being aware of a model's weaknesses helps practitioners understand its error boundaries and ensure safer, more effective real world deployment. However, finding enough challenge settings to train on is often a resource consuming and difficult task,

*These authors contributed equally.

[1]Results overview and animations: https://kavyakvk.github.io/adversarial-environment-generation/

[2]Repository with code implementation, analysis notebooks, and raw results: https://github.com/kavyakvk/adversarial-environment-generation

since many state of the art algorithms, including neural nets and reinforcement learning, are black box models with limitations in interpretability and explainability.

For agents learning policies for achieving some task, a natural approach to generating adversarial settings is training an opponent that plays against the agent being trained. AlphaGo, a reinforcement learning program for playing Go, was trained using self-play (Silver et al. 2017). (Pinto, Davidson, and Gupta 2017) used a physical robot adversary while training a protagonist robot to grasp objects. The adversary shakes or grabs at the protagonist's arm to destabilize its grasp, and training with the adversary resulted in a more effective grasping model.

Environment Design

In simulations without explicit opponents, adversarial generation can be applied to the environment itself to produce challenging — or simply interesting — problems for agents to act on. As a video game agent advances through a level, (Gisslén et al. 2021) generates the features of the level, segment by segment. (Dennis et al. 2020) treats the adversarial environment generator as an agent itself, which uses an LSTM model to learn a policy and is given 50 timesteps to place blockades in the environment. (Wang et al. 2019) uses a genetic algorithm to evolve the parameters encoding a bipedal walker environment.

We also utilize a genetic algorithm to produce difficult environments, but we extend the problem to swarm agents and introduce a crossover element. Furthermore, the work cited here focuses on using adversaries to train models for robustness and generalizability, while our work's main focus is interpretability.

The idea of interpreting an algorithm by observing how it responds to changes in the environment is also used by (Finkelstein et al. 2021), who explain unexpected agent behavior by transforming an environment until the agent behaves as expected. Previous work has also studied how to design an optimal environment according to a specified agent utility function, or for the purposes of influencing the agent's behavior in a particular way (Keren et al. 2017), (Zhang, Chen, and Parkes 2009).

Genetic Algorithms

In genetic algorithms, a population of candidates is evolved based on a desired fitness function. For instance, to evolve a simulated robot that can walk using a genetic algorithm, a suitable fitness function could be distance travelled. In our setting, where we seek to generate challenging environments for a swarm algorithm, a suitable fitness function would measure how difficult the swarm found the environment.

Evolution proceeds using mutations and crossover, which simulate features of natural evolution. In each generation, some population members are filled in using high-performing members from the last generation, and the rest are produced by mutating or combining the genomes of other members of the previous generation. The genome combination process makes it possible to combine strong features that evolved independently. It mimics the crossover that occurs during meiosis. Computer scientists have leveraged crossover by swapping sections of data structured as vectors, trees, or matrices, such as in the LEGO structures represented as trees by (Funes and Pollack 1999).

Our Contribution

In this work, we demonstrate a methodology for adversarial environment generation that differs from current literature in several key ways:

- The environment which is optimized is multi-agent, which poses a significant computational challenge. For instance, one challenge with interpreting swarm algorithms is that the swarm can be everywhere in the environment at once, so optimizing locally in the environment is no longer an effective way to reduce computational burden.
- We build from scratch a modular environment simulator to experiment on different multi-agent algorithms and adversarial environments. Our proposed methodology is impactful in identifying weaknesses in inherently less interpretable multi-agent algorithms.
- We compare multiple multi-agent algorithms in light of adversarial environment generation, including a biologically-inspired swarm algorithm and a deep Q-learning algorithm.
- The genetic algorithm we employ for environment generation leads to more interpretable results. We use the adversarially-generated environments to illuminate intuition behind how different foraging algorithms operate and their failure scenarios, including algorithms like deep Q-learning that are traditionally difficult to explain.
- We utilize a genetic algorithm that both mutates and crosses over a matrix-represented environment, which provides the potential for preserving promising meta-structures in generated environments.
- We demonstrate the improvement of multi-agent algorithms through co-evolution with a genetic algorithm generating adversarial environments.

Methods

Swarm Foraging Environment

While our adversarial environment generation methodology can be applied to any multi-agent environment, a significant proof-of-concept would be in a cooperative setting, where there is more complex behavior in the coordination between agents that may make evaluating the performance of the agents difficult.

To this end, we chose ant foraging as a proof-of-concept environment. The environment is defined as an `N` by `M` grid with several key elements:

- Hive: The location from which the agents are spawned from and must return food to.
- Food: Left in "piles" across the environment, each grid cell with food has a certain number of food units that agents can pick up to carry back to the hive.

- Barriers: Grid cells may be denoted as barriers; locations where agents may not move onto.
- Pheromone: At every time step, agents in the environment can deposit pheromone onto the grid cell they currently occupy. Environment parameters include the evaporation rate, the rate at which the pheromone decreases at every time step, and diffusion, the proportion of a grid cell's pheromone that is distributed to neighboring grid cells.

The introduction of pheromone drives part of the difficulty in interpreting multi-agent algorithms in this environment. While our algorithms have quite simple rules for depositing pheromone, the way they follow the pheromone and the role that the pheromone plays in the agent's decision-making is quite unclear, especially for the deep Q-learning agent implementation. This makes a foraging environment, and the algorithms we have deployed in the environment, a perfect example of highlighting the complexity of behavior that multi-agent algorithms have over their single-agent counterparts.

Environment Simulator

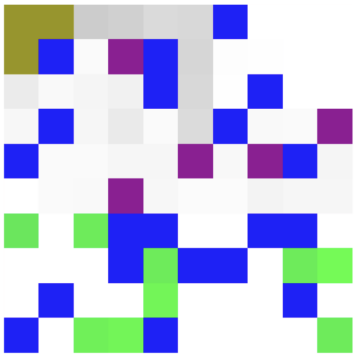

Figure 1: Example step from our foraging simulation. The hive is yellow, ants are purple, food is green, and blockades are blue. Higher pheromone levels are marked in darker shades of gray.

There exist several environment simulators for multi-agent problems and interfaces for training machine learning algorithms for single-agent scenarios. Unfortunately, at the time of writing, there was no robust multi-agent simulation software capable of interfacing with the training of machine learning algorithms in a non-competitive environment. The PettingZoo model environments represent the most prolific example of multi-agent simulators with sample environments, but represent largely competitive scenarios (Terry et al. 2020). OpenAI's Gym API is a staple for the implementation of algorithms and environments for reinforcement learning (RL) but does not readily support multi-agent environments (Brockman et al. 2016). Thus, we implemented from scratch our own environment simulator for our multi-agent cooperative setting.

As an implementation detail, the simulator maintains 3 grids that uniquely define the current state of the environment:

- `agent_grid`: Stores the locations of the agents currently interacting in the environment.
- `static_grid`: Stores the locations of the hive, food, and blockades.
- `dynamic_grid`: Stores the amount of pheromone currently deposited in each cell of the grid.

Similar to most environment simulators, each environment is initialized independently of the agents operating in it, instead interfacing with the agents through `get_observation(location)`, `step(agents)`, and `get_valid_movements(agent)` methods. The agents, in turn, share a common `get_action(observation)` function that is utilized by the environment's step function. The observation window is the $k \times k$ grid around the agent with a radius of r where $k = 2 \cdot r + 1$. The main metric returned by an episode of the environment is the total amount of food collected (picked up and returned to the hive) by the agents.

The environment simulator also implements partial observability, using a parameter for the radius of the observation window, `observation_radius` to construct a $2*$ `observation_radius`$+1$ length square observation centered around the agent as the input for each of the implemented algorithms.

Algorithm 1: Environment Update

```
Spawn agents
Update agents' observations
Update pheromone levels using evaporation rate
for agent ∈ active agents do
    loc = agent location
    movement, pheromone = agent.get_action(obs)
    loc ← loc + movement
    loc_p, loc_f = pheromone level at loc, food at loc
    loc_p = min(loc_p + pheromone, pheromone_cap)
    if loc has food then
        agent.food ← True
        loc_f -= 1
    end if
    if loc at hive then
        agent enters hive, placed in spawn queue
        food_collected += agent.food
    end if
end for
```

Swarm Algorithms

For the purposes of evaluating the Genetic Algorithm-generated environments, we implemented 3 different swarm agents: `RandomAgent` (a lower bound baseline for performance), `SwarmAgent` (based on ant swarm behavior), and `DQNAgent` (a deep learning-based agent). `SwarmAgent` and `DQNAgent` have unique policies to navigate each agent towards the food source, but follow a BFS path back to the hive once the agent has picked up food.

Random Agent As a lower-bound baseline for performance, the first algorithm implemented in the environment selected a random valid action to take and laid a random amount of pheromone at every timestep, as shown in Algorithm 2.

Algorithm 2: Random Agent Step

```
Lay random amount of pheromone
Take random valid action
```

Swarm Agent Algorithm We also test a traditional swarm foraging algorithm based off of the ant foraging literature. In particular, we draw inspiration from the algorithm presented by (Panait and Luke 2004), which differentiates ants' decision-making between before and after it finds food. If ants are not carrying food, they prefer to continue traveling forward with respect to the location they just came from. Other than this preference, they choose a direction probabilistically based on pheromone level in the forward side of the observation window. If there was no bias against the agents traveling backwards, they would follow pheromone they had just laid and make no progress towards finding new food sources. If ants are carrying food, they take an optimal step towards the hive, as calculated using a breadth-first search.

To summarize the major differences between the (Panait and Luke 2004) implementation and ours:

- Our agents can only move in the four cardinal directions, while the Panait et al. agents can also move diagonally.
- Our agents take ϵ-greedy actions in the environment, while the Panait et al. agents always move probabilistically based on pheromone levels.

Algorithm 3: Swarm Agent Step

```
if agent.food then
    Lay amount of pheromone corresponding to hive-seeking
    Take an optimal step back to hive
else
    Lay amount of pheromone corresponding to food-seeking
    if Random p < ε: then
        Take random valid action
    else
        Probabilistically choose valid target location in ant's forward direction in observation box, weighted by pheromone level
        Choose random cardinal direction that moves ant toward selected target location
        if n o such target location exists then
            Select a random valid movement
        end if
    end if
end if
```

Deep Q-Learning Deep Q-learning was introduced as a methodology for training agents to play Atari games (Mnih et al. 2013). Q-learning involves estimating values for each (state, action) pair in the environment, representing the total future discounted reward expected from taking a certain action from a certain state. Deep Q-learning was introduced to estimate these Q-values through the inference of a trained deep learning network rather than through the update of the Bellman Equations (Jang et al. 2019). This "policy network", i.e. the network whose predictions dictate the agent's policy, is the key component of deep Q-learning. Since then, numerous applications and improvements have evolved based on the core ideas of estimating the optimal policy for an agent in a defined environment through the inference of a learning network (Li 2017). Since our work focuses on the adversarial environment generation and coevolution with a genetic algorithm, we implement a relatively simple deep Q-learning approach.

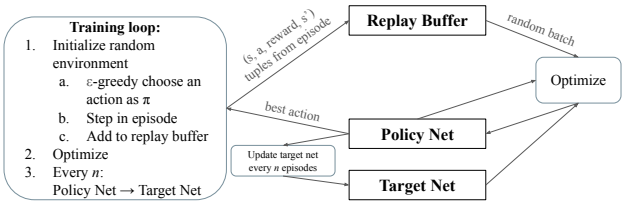

Figure 2: DQN training scheme, with replay buffer and target network.

A key component of deep Q-learning implemented in our training process is experience replay, the idea of storing previous (state, action, reward, state') tuples in a buffer for future sampling when training the learning network (Lin 1992). Experience replay is a valuable addition to the training of a Deep Q-learning Network (DQN) to make the training data independent and identically distributed, a necessary requirement for the stochastic gradient descent (SGD) that trains the DQN. Without experience replay, instead just batching the tuples as they are observed in the environment, the tuples would not be independent as they are observed by an agent in sequence.

Another key component of deep Q-learning is the introduction of a separate target network with an identical architecture to the policy network that dictates the agent's actions in the environment. Estimating a new value of Q(state, action) ultimately changes the estimated Q(state', action') for adjacent states, which makes training a network to predict these values significantly unstable. Introducing a target network, whose parameters are not trained but periodically synced with the policy network, produces a "delayed" stable estimate of Q values for use as the Q(state', action') values in the Bellman Equation.

The final component of deep Q-learning is the assigning of rewards to different states in the environment to aid in the identification of "more desirable" states. For this environment, the reward structure is as follows:

- +5 reward for collecting food
- +2 reward for depositing food
- +0.1 reward for observing food

While the first two rewards are intuitive, the small reward for observing food is added as a shaping reward to help the agent have a smoother gradient to the large reward for collecting food.

The training of the DQN followed the loop illustrated by Figure 2 and was implemented in the PyTorch deep learning framework (Paszke et al. 2019). Due to time constraints, default parameters were used for the learning rate, optimizer, epsilon-greedy decay, network architecture, batch size, and other training parameters.

Algorithm 4: DQN Evaluation Agent Step

if agent.food **then**
  Lay amount of pheromone corresponding to hive-seeking
  Take optimal step back to hive
else
  Lay amount of pheromone corresponding to food-seeking
  Action ranking $\leftarrow$ Inference of policy network with observation
  Valid action with highest ranking from policy network
end if

Genetic Algorithm

The adversarial environment generation is mediated by a genetic algorithm (GA), which manipulates the initialization of the environment through the locations of the food and blockades. Genetic algorithms optimize a defined fitness function through the evolution of a population of solutions, which are randomly initialized and evolved through manipulation such as crossover and mutation (Man, Tang, and Kwong 1996). The amount of food and blockades in the environment is predetermined as a parameter passed to the GA, where the food parameter is an exact requirement and the blockade parameter is an upper bound.

The population manipulated by the GA is a collection of 2D-arrays representing the initialization of the foraging environment. Crossover is defined by a tile size, which is used to randomly select groups of grid cells from either one of the parents to combine into their children. Mutation is defined as the probability with which each empty, food, or blockade cell is changed (to either make the cell empty or change the item initialized in the cell). Feasibility in the grid involves checking whether there is a path from hive to each of the food sources, which utilizes the same breadth-first search function implemented for agent navigation back to hive from any location in the environment.

Our implementation of a GA, shown in Algorithm 5, maintains an "elite" population preserved from generation to generation whose fitness outranks the other members of the population. Measuring the fitness of this elite population in contrast to the fitness of the rest of the population provides a valuable metric for the convergence of the GA.

The fitness function to maximize for the GA is defined as $\frac{-1 * food\_collected}{num\_agents}$ and calculated from running the agents in an environment initialized with the evolved grid for a fixed number of timesteps.

Algorithm 5: Genetic Algorithm

Population $pop \leftarrow \alpha$ random valid feasible grids
for generation $g \in$ generations **do**
  Calculate fitness for pop
  $pop_{elite} \leftarrow$ elitism $\beta * \alpha$ best-fitness individuals from pop
  Select pop_{cross} as $(1 - \beta) * \alpha$ individuals to cross from pop
  $pop_{cross} \leftarrow$ Randomly select pairs in pop_{cross} and perform crossover
  Select $(1 - \beta) * \alpha$ individuals from pop_{cross} and mutate each cell with mutation rate γ
  $pop \leftarrow pop_{cross} + pop_{elite}$
  Replace infeasible grids in pop with random valid feasible grids
end for

Co-evolution of DQN and GA

The eventual use for this methodology is that, following the evolution of difficult environments for the multi-agent algorithm by the GA, the algorithm can be modified or re-trained to be more robust. The co-training scheme for the Deep Q-learning Agent is described in Algorithm 6.

Algorithm 6: Co-evolution methodology

$Grids \leftarrow$ previously evolved GA population of size α
for Iteration i in iterations **do**
  Train DQN over e epochs evolved elite population from $Grids$ and at least $\alpha * \beta$ random valid feasible grids
  Evolve $Grids$ over g generations using updated DQN
end for

Results

Environment Parameters

The number of timesteps in an episode was chosen to be 300, as it was long enough for variation in the amount of food collected but not long enough for all food to be depleted from the environment. Other fixed parameters include a 12x12 square environment grid (this was the largest size that stayed under 18 hours of run time for one simulation), a spawn rate of 2 for agents at the hive, and 0.05 pheromone evaporation. For genetic algorithm environment generation, we ran each simulation for 50 generations at an elitism rate at 0.2.

Experiments

Overview We conducted three separate rounds of experiments to validate and improve our foraging and adversarial environment generation algorithms:

- Experiment A: We assess the first iterations of our foraging algorithms on randomly generated environments.

- Experiment B: We generate adversarial environments for each of our foraging algorithms using our genetic algorithm (GA). We explore the GA's hyperparameters to increase the effectiveness of adversarial environment generation.
- Experiment C: We improve the swarm and DQN foraging algorithms against the toughest adversarially generated environments.

Experiment A: Foraging

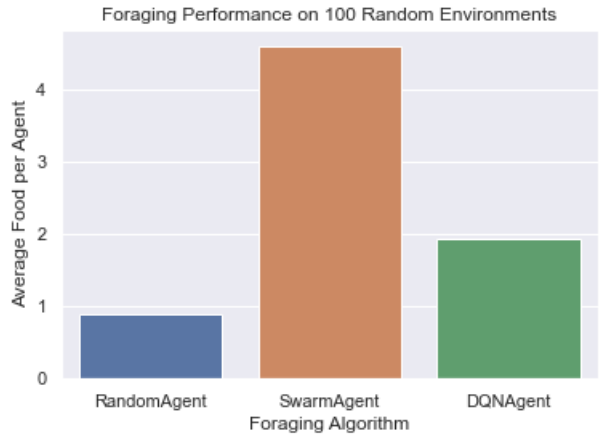

Figure 3: Foraging algorithm performance averaged over 100 randomly generated 12x12 environments.

To demonstrate the effectiveness of our foraging algorithms, we evaluated their performance on 100 randomly generated environments. Each environment was a 12x12 grid with 20 barriers and 40 food particles. As shown in Figure 3, `SwarmAgent` was found initially to be the most successful in bringing food back to the hive, with an average of 4.6 food particles foraged per agent over 300 time steps. This is more than double the performance of `DQNAgent`, which in turn doubled the performance of `RandomAgent`. One result of interest from the GIFs (included in our GitHub) of our simulations is that `SwarmAgent` follows pheromones more closely than `DQNAgent`, which seems to approach the problem through a mix of pathfinding and pheromone-based decision making. Harkening back to our background section, since the DQN is initially a black box model that is hard to interpret, the adversarial environments generated by the GA in Experiment B may reveal more insight into how the DQN algorithm works.

These performance numbers form a baseline to evaluate the effectiveness of our GA approach, which aims to decrease the average food per agent to 0 (GA fitness is equivalent to multiplying average food per agent by -1). As our focus in this paper is to investigate the development of adversarial environments and not to propose a novel DQN approach, we only needed baseline numbers as a benchmark and didn't fully optimize the DQN algorithm. However, we do show later in Experiment C the effectiveness of co-evolution in training the DQN to perform well even when facing against difficult environments generated by the GA.

Experiment B: Adversarial Environment Generation

We begin our testing of the GA with evaluating average fitness of elite environments generated over time. We run the GA for 50 generations, where each generation includes simulations of 100 individual environments. Since our GA's elitism rate is 0.2 and our total population is 100, this means at every generation the fitness is averaged across 20 different environments.

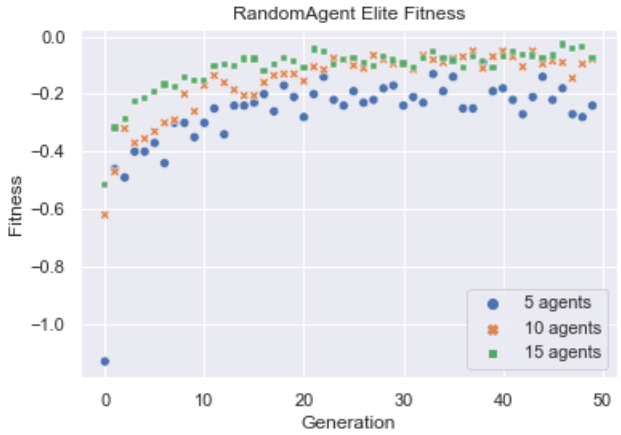

Figure 4: Average `RandomAgent` fitness of elite population over 50 generations with varying number of agents.

Figure 5: Average `SwarmAgent` fitness of elite population over 50 generations with varying number of agents.

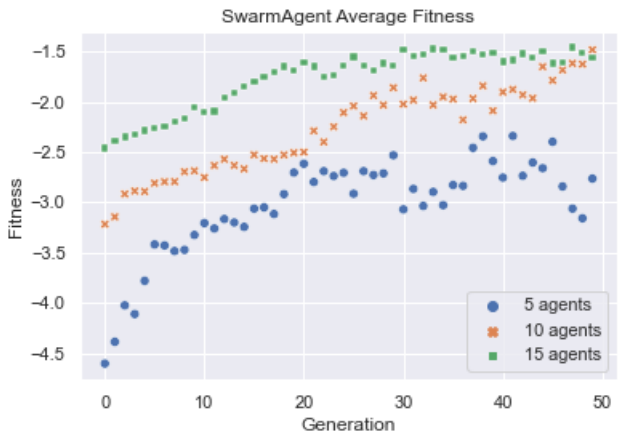

Figure 6: Average `SwarmAgent` fitness of entire population over 50 generations with varying number of agents.

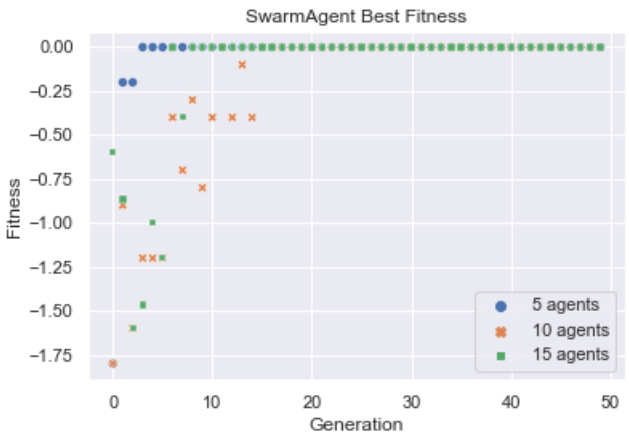

Figure 7: Best `SwarmAgent` fitness over 50 generations with varying number of agents

Figure 4 shows that although the foraging algorithm employed here is random, the GA is able to find environments which on average limit the amount of food collected per agent to almost 0. Moreover, there is smaller variance in the fitness from one generation to the next when the number of agents is increased.

Figures 5, 6, and 7 show the effectiveness of GA against the swarm agent algorithm in 3 different metrics – elite fitness (averaged across the 20 elite), average fitness (averaged across entire population of 100), and best fitness (the greatest fitness achieved per generation). Compared to a random walk, the swarm algorithm starts off with a much higher amount of food collected per agent but is also quickly mitigated to close to 0 food per agent over 300 time steps.

Generally, the average fitness of the elite population converges much faster than the average fitness of the remainder of the population because mutations and crossovers of members of near-optimal grids may result in an infeasible solution. In our GA implementation, infeasible solutions are replaced with randomly initialized solutions, which would significantly decease the average fitness of the population. We see in 7 that the GA is able to find the most difficult environments faster when there are only 5 agents (the best fitness is 0.0 by generation 6) whereas increasing the number of agents delays this by 5 to 10 generations, indicating longer times before convergence.

Hyperparameter Tuning Having established the success of the GA, we explored tuning four parameters for greater effectiveness:

1. We compared tuning the tile size vs. the mutation rate used in the genetic algorithm. This aimed to see which parameters resulted in faster convergence.

Figure 8: Average Random agent fitness of elite population after 50 generations.

Random agent algorithm Figure 8 shows a heatmap of the average elite fitness at the end of each GA simulation (on generation 50) with varying tile size and mutation rate. There aren't any clear differences in fitness, perhaps due to the randomness of the `RandomAgent` algorithm making it difficult for the GA to increase performance.

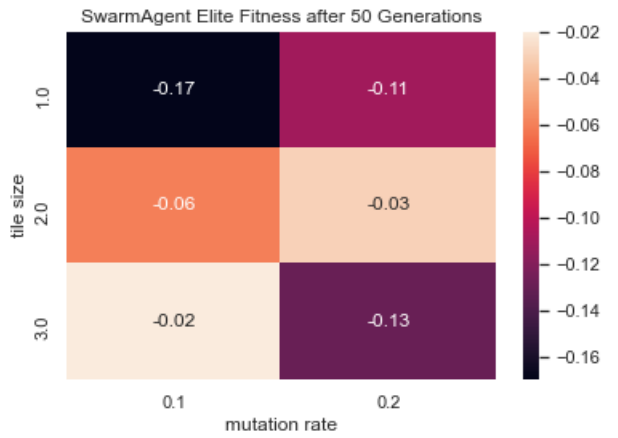

Figure 9: Average Swarm agent fitness of elite population after 50 generations.

Figure 10: Average Swarm agent fitness of elite population with varying tile size.

Swarm agent Algorithm On the other hand, we see from Figure 9 when against the `SwarmAgent` algorithm the GA tends to achieve higher fitness with a tile size of 2. Furthermore, Figure 10 shows that tile sizes of 2 or 3 not only achieve higher performance but also converge much faster than when the tile size is 1. This may be an indication that the most effective barriers are diagonal chains of blocks, which can only be achieved with greater tile size.

Mutation rate also plays a major factor in convergence speed for GA fitness. Figure 11 shows that a higher mutation rate of 0.2 slows the rate of convergence considerably (almost to a linear relation after the first few generations). Although a higher mutation rate is often used to prevent converging too early to a suboptimal fitness level, we don't need it here since the GA already averages close to 0.0 for a fitness level with only a 0.1 mutation rate.

Figure 11: Average `SwarmAgent` fitness of elite population with varying mutation rate.

Figure 12: Average `DQNAgent` fitness of elite population with varying tile size.

DQN agent Algorithm When running the GA against the `DQNAgent`, all combinations of tile size and mutation rate resulted in an average of 0.0 for elite fitness after 50 generations. In fact, as supported by Figure 12, we see that the GA is able to figure out how to counteract and fully block the `DQNAgent` algorithm from *obtaining any food* in less than 15 generations. The resounding success of the GA against `DQNAgent` may be due to the predicability and consistency of the deep Q-learning algorithm's policy as a result of its straightforward rewards.

2. We also evaluated the effects from varying values of the exact number of food particles and maximum number of barriers the GA was allowed to utilize.

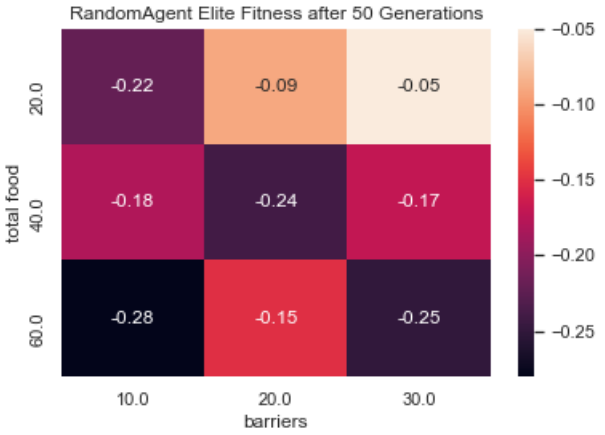

Figure 13: Average `RandomAgent` fitness of elite population with tile size 2 and 0.1 mutation rate.

Random agent algorithm From Figure 13, we see that the GA performs well even with drastic increases in total food it needs to place down into the environment (with no increase in grid size). Again due to the algorithm's randomness, it appears that there aren't any clear trends for changes in performance when the number of barriers is increased.

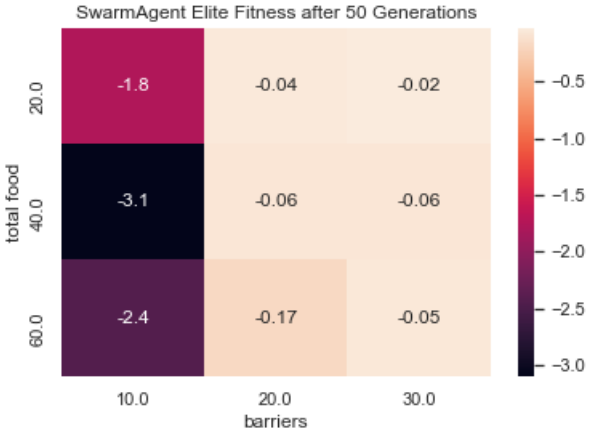

Figure 14: Average `SwarmAgent` fitness of elite population with tile size 2 and 0.1 mutation rate.

`SwarmAgent` Algorithm In contrast to the results for `RandomAgent`, Figure 14 demonstrates that the GA gets a huge boost in performance against `SwarmAgent` when the number of barriers is increased from 10 to 20. This indicates that the GA can find meaningful use with around 20 barriers, although further increasing the number of barriers seems to come with diminishing returns.

Figure 15: Average `SwarmAgent` fitness of elite population with varying number of food particles.

Figure 15 shows us that similar to `RandomAgent`, `SwarmAgent` struggles to forage any food after 40 generations of the GA, even when increasing the number of food particles in the environment. Although the ending fitness is around the same regardless of the amount of food, the rate of convergence for the GA is slower when there are only 20 food particles, which is surprising given that intuitively it should converge at least as fast as when there are 40 food particles.

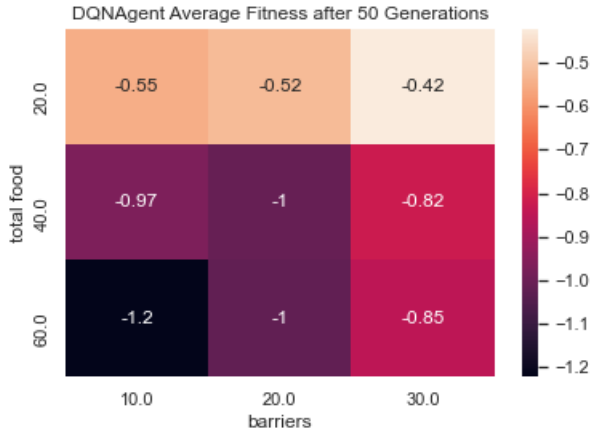

Figure 16: Average DQN agent fitness of the final population with tile size 2 and 0.1 mutation rate.

Figure 17: Average DQN agent fitness of elite population with varying number of food particles.

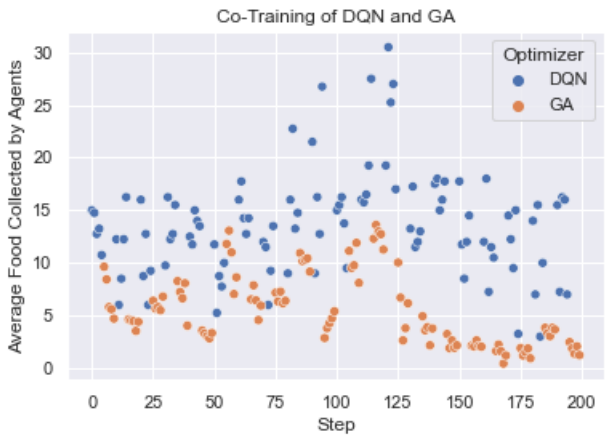

Figure 18: Average food collected at every step of the co-training process. A "step" of the DQN Agent represents 4 epochs of training, while a "step" of the GA represents a generation.

`DQNAgent` algorithm As shown in Figure 17, by the end of the 50-generation evolution of the GA, the elite population was saturated with grids in which the `DQNAgents` collected 0 food. This is in contrast to the `SwarmAgent` simulations, which continued to collect some food by the end of the same evolution period. There are several likely explanations for this phenomenon, but the most likely is that there is no ϵ-greedy random exploration associated with steps of the `DQNAgent` in the environment outside of the training process. In addition, the DQNAgents were only trained with random grids, and not with grids especially evolved to pose difficult training problems. These explanations also shed light as to why the overall magnitude of the fitness (and the food collected in the environment) is much lower for the `DQNAgent`.

Since the average `DQNAgent` fitness of the elite population was 0 at the end of the GA evolution process, Figure 16 depicts the average fitness of the entire population. The trends are quite expected: food collected decreases with increased barriers and with decreased food.

Experiment C: Co-Evolution – Foraging Algorithm Improvement

`DQNAgent` For the co-evolution training, we utilize the following parameters:

- Default environment parameters with 40 food and 20 blockades
- Default GA parameters of 0.2 elitism, 0.1 mutation with 5 generations and population size $\alpha = 30$
- `DQNAgent` training parameters of 20 training epochs
- Random grid proportion $\beta = 0.2$ for selecting the 20 training grids for the `DQNAgent`.

Figure 18 illustrates the co-training process over 20 iterations, where each iteration represents 20 training epochs of the `DQNAgent` and 5 generations of the GA. There does seem to be a back-and-forth, whereby during the `DQNAgent` steps, average food collected increases and during the GA steps, average food collected decreases.

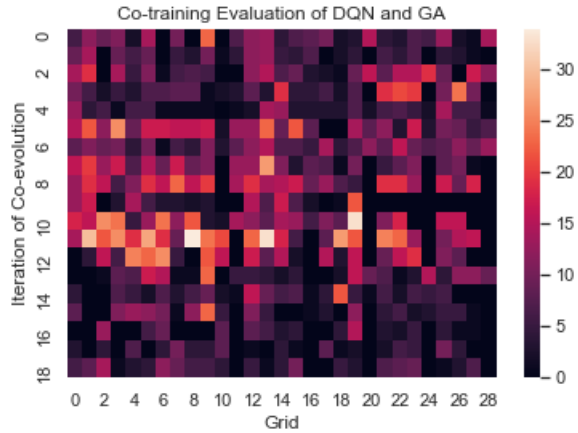

Figure 19: Food collected at every iteration of the co-training process (y-axis) on each grid in the evaluation set (x-axis). Grids 0-9 are random grids, 10-19 are grids sampled from the non-elite population of the GA prior to co-training, and 20-29 are grids sampled from the elite population of the GA prior to co-training.

We hypothesize the learning process is quite cyclical, which is demonstrated on the evaluation set of grids illustrated in Figure 19. There was clearly a point in time where the `DQNAgent` performed incredibly well even on the more difficult grids (approximately 11 iterations, 110 steps). Following this point, the average food collected by the GA sig-

nificantly decreased, presumably due to the evolution of several really difficult grids that were then evolved into other grids, and resulted in an equivalent significant decrease in food collected. Once the GA evolved a significantly difficult enough set of environments, the DQNAgent could not intelligently locate the food sources and gather reward, which scrambled the total discounted reward estimates for each state.

Discussion

Interpreting GA Results

Figure 20 illustrates an average of the elite population of the "hardest" grids generated for each of the multi-agent algorithms.

The elite population of the `RandomAgent` has more variance since the agents move randomly without a defined strategy with regards to the placement of the blockades and food. It's clear through the evolution of the grids that, in the aggregate, blockades move toward the hive and food moves away from the hive, which makes it more difficult to both collect food and navigate back to the hive when performing a random walk.

`SwarmAgent`'s preference for moving forward rather than backwards to avoid backtracking on previously deposited pheromone leads it to become trapped in some structures. In the dark blue structures that emerge in Figure 20 for `SwarmAgent`, moving forward whenever possible means missing the only sideways exit out of the trap structure. The trap is exacerbated because the ants only lay pheromone in the squares they have already visited, creating positive feedback that further cements them in only moving forward. Even by generation 19, the GA beings to exploit this trap, which is more or less fully evolved by generation 39 — the saturation of the blue color in the averaged elite grid means that nearly all members of the elite population have evolved this structure. There are several modifications we could make to the environment and `SwarmAgent` algorithm to help it evade this trap, but we find it impressive that the GA evolved such an effective and non-intuitive structure. (In an additional experiment, we tried adding random noise to `SwarmAgent` actions, and found that while ants occasionally exited the trap, this was rare and very little food was still collected.)

The evolved grids for `DQNAgent` are quite unlike the other two algorithms. First, the GA converges very quickly; as previous explained, this is likely due to the lack of ϵ-greedy randomness during evaluation. This agent, as is the case with many reinforcement-learning algorithms, learns a quite specific and non-generalizable strategy, so while it could evade any single specific configuration of blockades surrounding the hive, we think it struggles in general maze-like settings. This is especially the case since it was only trained on random grids, not on any "intelligently" chosen difficult examples. Food generally moves towards the center of the grid, where the random placement of blockades surrounding the food may make it quite difficult to readily access. Another rationalization for the grouped food locations is that because the `DQNAgents` receive a partial reward for observing food, the optimal action when the observable space is crowded with a significant amount of food is quite confusing.

In Figures 21 and 22, we fix default environment parameters and quantify some of the differences between challenge settings generated for different algorithms. First, we observe that a common feature in elite environments seems to be placing blocks next to the hive, which limits how ants can enter and exit. In 21, we observe that for `SwarmAgent` and `DQNAgent`, the elite population quickly evolves so that almost all members have the maximum permitted number of hive-adjacent blocks. The same trend does not hold for `RandomAgent`, for which most elite environments have only 1 of 2 possible hive blocks. Our interpretation of this result is that blocking the hive is less effective for `RandomAgent` because the random movement ensures that ants will not be trapped by any particular configuration.

In Figure 22, we visualize how food blocks and obstacle blocks' average distance from the hive evolves for elite environment populations. We see that generally, the genetic algorithm evolves to place obstacles closer to the hive and food farther from the hive, but that this evolution occurs at different paces and to different extends for the three algorithms. Adversarial environments generated for `DQNAgent` seem to rely the least on distance-based object placement, and `SwarmAgent` food distance seems to spike, fall, and then rise again, perhaps as the GA approaches different local optima. We also note that these two figures are based on only one run of the GA.

Parameter Significance

There are two ways of interpreting the elitism population in the context of an optimization problem: a high elitism rate at the end of evolution would guarantee a population full of difficult grids, while a low elitism would evolve a population of grids representative of the range of difficult environments. An extension of our current experiment useful for co-evolution training of the `DQNAgent` would be to decrease the elitism rate over time as a sort of "learning rate" for GA environment generation.

Conclusion

In our work, we present the first known genetic algorithm (GA) approach in generating adversarial environments for multi-agent problems. We demonstrate the GA's success in completely blocking both swarm-based and deep Q-learning-based algorithms from foraging virtually any food in a multi-agent cooperative setting. Moreover, we explore multiple parameters such as tile size and mutation rate which successfully enable the GA to reach convergence faster (within 10 generations) without sacrificing ending performance. The GA allows us to interpret algorithms like deep Q-learning that are typically hard to understand, giving us the ability to generate more difficult environments for training. Finally, we utilize this ability to co-evolve deep Q-learning and genetic algorithms together, demonstrating the potential of such approaches in helping similar algorithms to prevent overfitting.

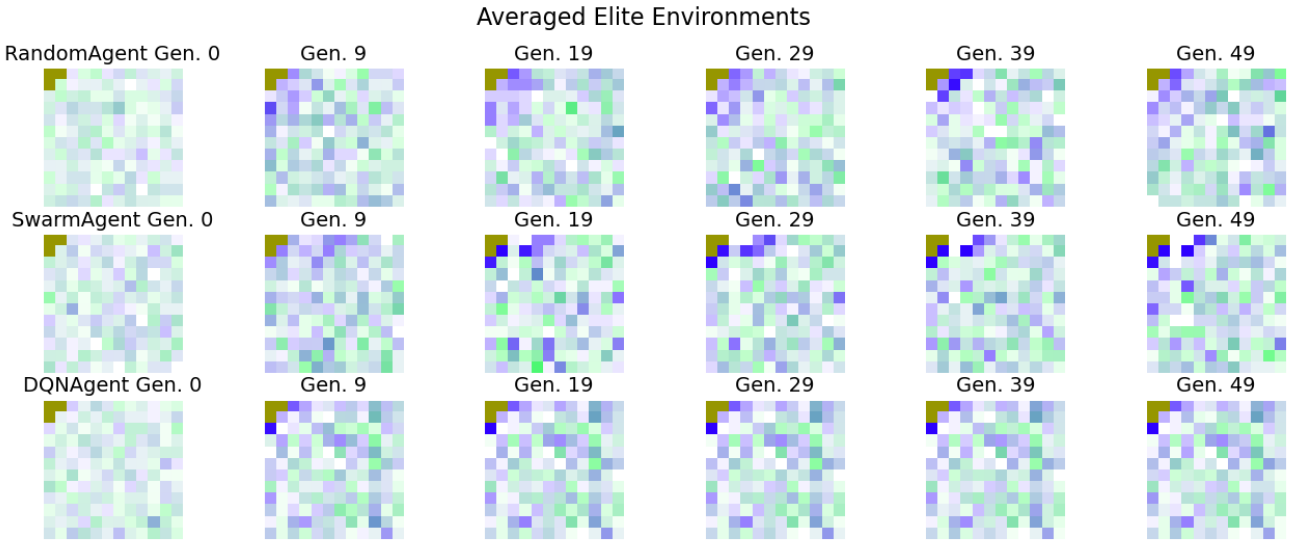

Figure 20: Evolution of hardest adversarially-generated environments.

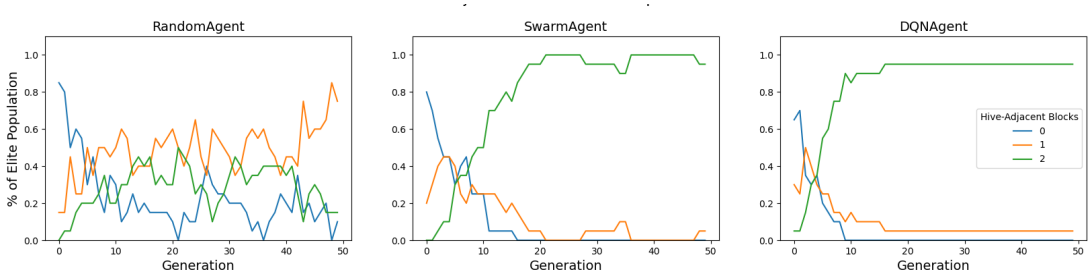

Figure 21: Evolution amongst elite population of blockade counts in the 3 adjacent grid locations to the hive.

Future Work

Our work uncovered several tracks for potential future work. One is extending our method for challenge setting generation to other multiagent or GA-compatible environments. It could be extremely beneficial to generate challenging settings for traffic control or autonomous driving problems. A webpage navigation environment could also be interesting, as the tree structure of the DOM makes it suitable for mutations and crossover.

Another track is further formalizing our interpretation methods to more automatically infer an algorithm's weaknesses, such as by more rigorously quantifying concepts like certainty and diversity. For instance, we could apply clustering to the elite population to try and understand how many distinct challenge settings the GA has uncovered. We noticed that (especially for `SwarmAgent`), the GA seems to repeatedly generate specific structures. This is useful for emphasizing the difficulty these structures pose for the algorithm, but we may desire a GA that produces more diverse challenge settings, which would give practitioners a broader picture of an algorithm's weaknesses. Thus, another direction for future work is incentivizing diversity in the elite population.

Finally, it would be natural to bridge our method with existing adversarial training papers by continuing to explore co-evolution between our DQN algorithm and the GA.

Contribution Breakdown

All three group members contributed equally to this project. Everyone worked on developing the research idea, planning environment implementation, and writing the paper. Eric implemented the environment simulator, `RandomAgent`, and ran experiments A and B along with the visualizations of their results. Lucy implemented the `SwarmAgent` algorithm and webpage, and generated visuals for the discussion section. Kavya implemented the `DQNAgent` and GA as well as the co-evolution between them as shown in experiment C.

References

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

Dennis, M.; Jaques, N.; Vinitsky, E.; Bayen, A.; Russell, S.; Critch, A.; and Levine, S. 2020. Emergent complexity and zero-shot transfer via unsupervised environment design. *arXiv preprint arXiv:2012.02096*.

Figure 22: Evolution of average distance of food and blockades from hive.

Finkelstein, M.; Liu, L.; Levy Schlot, N.; Kolumbus, Y.; Parkes, D. C.; Rosenschein, J. S.; and Keren, S. 2021. Deep Reinforcement Learning Explanation via Model Transforms. *NeurIPS Workshop on Deep Reinforcement Learning*.

Funes, P.; and Pollack, J. B. 1999. Computer evolution of buildable objects for evolutionary design by computers. *Evolutionary design by computers*, 387–403.

Gisslén, L.; Eakins, A.; Gordillo, C.; Bergdahl, J.; and Tollmar, K. 2021. Adversarial reinforcement learning for procedural content generation. *arXiv preprint arXiv:2103.04847*.

Jang, B.; Kim, M.; Harerimana, G.; and Kim, J. W. 2019. Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*, 7: 133653–133667.

Keren, S.; Pineda, L.; Gal, A.; Karpas, E.; and Zilberstein, S. 2017. Equi-reward utility maximizing design in stochastic environments. *HSDIP 2017*, 19.

Li, Y. 2017. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*.

Lin, L.-J. 1992. *Reinforcement learning for robots using neural networks*. Carnegie Mellon University.

Man, K.-F.; Tang, K.-S.; and Kwong, S. 1996. Genetic algorithms: concepts and applications [in engineering design]. *IEEE transactions on Industrial Electronics*, 43(5): 519–534.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Panait, L.; and Luke, S. 2004. Ant foraging revisited. In *proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32: 8026–8037.

Pinto, L.; Davidson, J.; and Gupta, A. 2017. Supervision via competition: Robot adversaries for learning tasks. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 1601–1608. IEEE.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *nature*, 550(7676): 354–359.

Terry, J. K.; Black, B.; Grammel, N.; Jayakumar, M.; Hari, A.; Sulivan, R.; Santos, L.; Perez, R.; Horsch, C.; Dieffendahl, C.; Williams, N. L.; Lokesh, Y.; Sullivan, R.; and Ravi, P. 2020. PettingZoo: Gym for Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:2009.14471*.

Wang, R.; Lehman, J.; Clune, J.; and Stanley, K. O. 2019. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*.

Zhang, H.; Chen, Y.; and Parkes, D. C. 2009. A general approach to environment design with one agent. In *Twenty-First International Joint Conference on Artificial Intelligence*.