

# Efficacy of Deep Neuroevolution for Emergent Control of Multi-Agent Swarms

Russell Bingham

*Electrical and Computer Eng. Cornell Tech*  
New York, New York

Elijah Roussos

*Electrical and Computer Eng. Cornell Tech*  
New York, New York

Darren Midkiff

*Electrical and Computer Eng. Cornell Tech*  
New York, New York

Everett Key

*Electrical and Computer Eng. Cornell Tech*  
New York, New York

**Abstract**—A functional evolutionary algorithm system for improving the behavior of pheromone-dropping “ant” agents in simulation is presented. Given a semi-degenerate set of initial conditions with biased food locations, the agents learn to go to locations where there will likely be food, but when food is distributed truly randomly the genetic algorithm is unable to find a solution. Traditional backpropagation-based RL methods are tested as alternate training options, but no neural network agent is able to approach the performance of a simple pre-programmed agent, suggesting that the foraging problem is fundamentally difficult to train a neural network decision agent for.

## I. INTRODUCTION

In nature, ants are able to efficiently gather food as a collective using a combination of random scouting behaviors and pheromone-marking of paths to food sources. This has inspired a variety of optimization methods, most famously Ant-Colony-Optimization (ACO) [2][6]. However, the agents in these systems have predefined behavior and cannot adapt new techniques on their own. This may present an issue in more complex or mutable problem spaces, as it is difficult for agent designers to know the optimal behaviors for agents in a given task environment. A more generalizable method for designing agent behavior would focus on allowing agents to learn optimal behaviors over time in response to their performance on the task at hand. Existing algorithms for decision agent learning like reinforcement learning (RL) are difficult to apply to the multi-agent case because of the challenge of accurately attributing discrete rewards to agents that cooperate to reach a goal [10].

Evolutionary algorithms circumvent this reward-attribution problem by sampling the solution space broadly and improving on those individuals that perform well in simulation. The formal implementation of such a method involves designing the “brain” of each agent as a neural network, then using a genetic algorithm to evolve the weights of that network using mutation and crossover to move through the solution space towards a goal. This approach is referred to as “deep neuroevolution” in the academic literature, and the primary system presented here is a functional implementation of this method. Prior work on this method has generated impressive training results, even outperforming backpropagation-based RL for some tasks, but in practice the largely stochastic food

foraging problem seems to be fundamentally difficult for a neural network to solve, regardless of training methodology.

## II. RELATED WORK

### A. Deep Neuroevolution

Work using genetic algorithms to search for the optimal network weights for a deep RL agent network has been ongoing since at least 2018 [13]. Studies have proven that this strategy is very effective for deep RL tasks, and can outperform traditional methods on some learning tasks, even using a completely naive genetic algorithm without crossover [16]. After broad review, it seems that application of this to swarm systems where agents have limited sensing capabilities or learnable stochasticity has not been widely explored. This work seeks to fill this gap.

### B. Multi-Agent Deep Neuroevolution

While the behaviors of agents in swarms are typically of bespoke, application-specific design, they may also be learned via machine learning or evolutionary algorithms. The concept of applying machine learning to multi-agent systems is not new, with several applications to vehicle monitoring, air traffic control and network routing [4], and to problems in reinforcement learning [7]. Evolutionary computation methodologies in multi-agent systems are also not novel, and have proven to be effective optimizers for agent-based simulations [3]. However, combining these techniques is fairly novel and uncommon in the literature. Current methods operate on the NERO game environment [5] through the NEAT algorithm, which generates recurrent neural networks for agents by adding and removing network nodes and changing network weights [8]. While the work simulated foraging agents, each food source emitted a signal to the agents, and no stigmergic or inter-agent interactions were considered. Other research has focused on deep RL methods for multiagent systems, but ant foraging was not included [10].

## III. SIMULATION ENVIRONMENT

The basis for all ant simulations presented herein is a custom-built testing environment that simulates and can graphically display the actions performed by a range of different

agent types, as shown in Figure 1. The environment is structured such that agents move freely in continuous 2D space while the locations of food and pheromone are tracked using a grid of discrete cells. The agents are initialized at a central nest location (shown in black on the grid), and their decision-making fitness is evaluated based on the number of food particles they can carry back to the nest given a static number of time steps ( $\text{max-steps} = 300$  unless otherwise noted). The environment size and agent density shown in Figure 1 are also representative of the large majority of learning simulation parameters, with 10 agents exploring a 30 by 30 cell grid.

#### IV. NAIVE COMPARISON (DETERMINANT)

In order to validate the environment and provide some baseline as to what stigmergic behavior could look like in it, a simple, pre-programmed agent was developed. This agent is referred to as *DeterminAnt*, though it could be argued this name is a bit of a misnomer, as its behavior involves a random walk and is not entirely deterministic. Nonetheless, it is far more pre-determined than that of the neural network-based agents that will be discussed later.

Determinant follows a simple set of rules, inspired by a previous ant simulation implementations [2][6]. At a high level, each agent conducts the following actions:

- Randomly explore the environment in search of food.
- If food is located, pick it up and walk back to the nest, dropping pheromone.
- If pheromone is encountered while exploring, follow the pheromone away from the nest.

These rules readily give rise to stigmergic behavior in the simulation environment, as shown in Figure 1. The food gathering fitness of this agent is used in later figures as a reference for high-quality food collection.

#### V. NEURAL NETWORK AGENT BRAIN

##### A. Input

In this work, the ant agents' brains are modeled as neural networks which take in a variety of inputs from the environment. These inputs are based upon what real ants may perceive, including sensing features as well as features relating to the ant's orientation and heading. The sensing features were extracted from the arc of cells in front of the agent. For all agent architectures, decision-making inputs were the following:

- *has\_food* - A flag stipulating whether the agent is holding food.
- *adjacent\_food* - A list of food items in the 5 cells in front of the agent.
- *adjacent\_pheromone* - A list of pheromone items in the 5 cells in front of the agent.
- *global\_orientation* - The angle of an agent's current position relative to the nest location.
- *local\_orientation* - The current heading of an agent.

The orientation inputs were tested as both raw angles in radians as well as the equivalent trigonometric representation via sine and cosine.

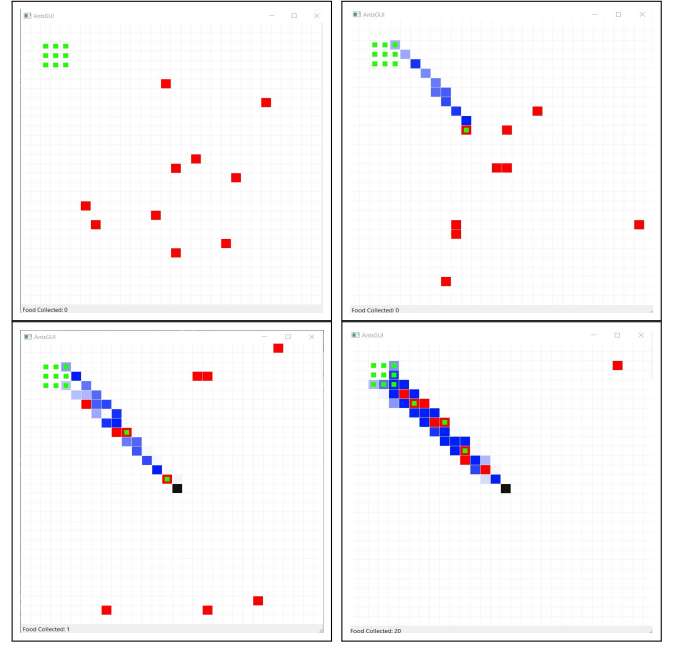


Fig. 1. Example foraging simulation using pre-programmed *DeterminAnt* agent. Agents are shown in red, food in green, pheromone concentration in blue, and nest in black.

##### B. Output

Just like the *DeterminAnt* baseline, this agent should decide at every time step how to change its local orientation and how much pheromone should be dropped. The first two network outputs follow naturally: change in local angle and amount of pheromone to drop in the current cell. There is, however, a third, less-intuitive output: the *randomness* of the agent's walk, which serves to balance between environmental exploration and other learned behaviors.

- *orientation\_delta* - The amount by which to alter the local orientation (in radians).
- *pheromone* - The amount of pheromone to drop in the current cell.
- *randomness* - The amount of random noise added to the agent's walk.

##### C. Network Architectures

In designing any neural network architecture, there are three key design choices that affect the performance of the network: the number of hidden layers, the size of those hidden layers and the desired output action space. Larger network topologies are harder to train due to the exponential increase in parameters; however, if the network is not large enough, it may not be able to approximate the desired behavior regardless of training. A similar tradeoff applies to the choice of output space. Continuous action spaces require the model to be well fit to the problem so as to not emit noisy output. Discrete output spaces, on the other hand, must be designed in such a way that enough information is retained for the agent to take a suitable action.

In this vein, two key architectures were created, the *DominAnt* network and the *DiscreteAnt* network. The main difference between the two networks is the former outputs actions in continuous space with a smaller network topology while the latter outputs actions in discrete space corresponding to predetermined values for *orientation\_delta* and *pheromone* using a larger network topology.

1) *DominAnt*: The *DominAnt* network is a four-layer neural network with a continuous action space output. It contains two hidden layers of size 15 and 8, respectively, with each layer making use of the *tanh* activation function. The network outputs three values, *orientation\_delta*, *pheromone* and *randomness*, which are all passed through another *tanh* activation to ensure each output is between -1 and 1. To calculate the final outputs, the values are scaled to the ranges below.

$$-\frac{\pi}{2} < \textit{orientation\_delta} < \frac{\pi}{2}$$

$$0 < \textit{pheromone} < 5$$

$$-\frac{\pi}{4} < \textit{randomness} < \frac{\pi}{4}$$

2) *DiscreteAnt*: The network architecture for *DiscreteAnt* is slightly more complex, as the continuous action space needs to be discretized appropriately. This includes accommodations to the output activations of the network, such that all discrete actions for both laying pheromone and calculating direction are each chosen from a separate probability distributions.

To discretize the agent action space, the  $-\frac{\pi}{2}$  to  $\frac{\pi}{2}$  radian range of possible orientation deltas was divided into seven equal buckets (0-6), while the range of possible pheromone buckets was discretised into five buckets (0-4). Each discrete actions' value corresponds to the midpoint between the lower and upper bound of the bucket for *orientation\_delta*, while for pheromone was left as the index of the bucket. Finally, an eighth bucket is added to the discretized *orientation\_delta* action space, which if selected causes the agent to take a turn a random amount in the  $-\frac{\pi}{2}$  to  $\frac{\pi}{2}$  range. This is the discrete version of the *randomness* output from *DominAnt*.

To accommodate this change to the action space, the network was modified to be larger in layer size while retaining the same number of layers as *DominAnt*. The two hidden layers are both of size 32 and each makes use of the *mish* activation function. *Mish* is used here, as the output no longer needs to be constrained between -1 and 1. The network outputs a total of 13 values, corresponding to every action the agent may take, with the first eight outputs corresponding the *orientation\_delta* output and the last 5 corresponding to the *pheromone* output. As the *orientation\_delta* actions and pheromone actions should be independent from one another, the use of network branching is employed on the last layer of the network. This is done by running two separate softmax activation layers on the neurons corresponding to the *orientation\_delta* and *pheromone* outputs respectively, such that the output probabilities for all possible discrete direction change actions and pheromone actions add up to one in each case. This allows the agent to sample actions

for both *orientation\_delta/randomness* and and pheromone drop while utilizing a single network.

## VI. GENETIC ALGORITHM

As noted in Section I, the primary focus of this work was implementing a genetic algorithm (GA) to iteratively update the component weights of each of the different neural network-based agent types, thus conducting deep neuroevolution. In broad strokes, the algorithm maintains a large population of potential solutions to the problem and, at each iteration, tests every individual in the population to determine its fitness (measured as total food gathered). Those individuals which score the highest are selected to reproduce and are mutated and crossed over with other highly fit individuals to create new potential solutions. In theory, the method eventually generates good solutions to the problem by leveraging the sheer size of its population and random exploration to stochastically move towards an optimal solution, creating a simulacrum of gradient following.

### A. Structure of Individuals

Each individual in the population represents the initialization parameters for an agent's neural network, and is thus stored as a list of arrays, where each array makes up the full set of parameters for one layer of the agent's brain.

### B. Scoring

At each iteration, every individual is selected in order, and the weights stored in the chromosome are used to initialize a PyTorch neural network of the given size and shape. For each testing simulation, only one network is initialized because every individual in the swarm has an identical brain. The brain is evaluated on its effectiveness at making cooperative decisions. The given number of agent copies are initialized in the environment, and they are updated with decisions made by the master network for the given number of maximum time steps (usually 300, as noted above). Because the agents are hard coded to move the same amount at each time step, this encoding of swarm fitness captures both food gathering skill and movement efficiency because the swarm is incentivized to make optimal use of the finite time horizon.

### C. Mutation

After the training environment generates scores for every individual in the population, the best individuals (usually those which score at or above the 75th percentile) are selected for reproduction. The first process to create new individuals is mutation, where individual network weights are modified using Gaussian noise based on some mutation probability. This mutation process expands the population's coverage of the search space into new domains.

### D. Crossover

After a chromosome is mutated, the reproduction subroutine also has some probability of switching out sections of the network weights. A newly mutated chromosome will inherit the weights of an entire node from a different high-scoring

member of the population. This routine should expand and improve the diversity of the search space, and hopefully transfer learned relationships between inputs from one quality individual to the next, further increasing the fitness of new individuals.

## VII. RESULTS

To determine the performance of the various agent architectures and training methods, several hundred processor-hours of training simulations were conducted. Even with highly-optimized and multi-threaded code running on a rented cloud-processing virtual machine, simulations with population size of 800 scored using the median of 3 different trials took on the order of 12 hours for 250 epochs. Though these training times are far longer than direct backpropagation techniques, the GA training system was drastically simpler to code and understand at a conceptual level. This simplicity versus runtime comparison is the primary tradeoff of using the GA paradigm when compared to traditional RL.

### A. GA Training Efficacy

The earliest tests of the GA training process were conducted using the *DominAnt* architecture, but a code typo caused the food to never spawn on the left side of the environment and spawn with a strong preference for some randomized position along the right side of the environment. The environment also bounced agents off walls in a predictable and exploitable way and provided input headings to the agent with a large discontinuity at  $2\pi$ . When applied to this significantly flawed environment, the GA training procedure produced impressive score improvement in just 200 epochs of training, as seen in orange in Figure 2.

However, when the highest quality agents from epoch 200 are visualized in the environment GUI, it is clear that the learned behaviors are in large part clever ways of exploiting the flaws in the environment. The agents consistently go straight to the right side, where the chance of food spawning is artificially high, and they choose the upper part of the right side specifically to exploit the way that the environment will bounce them off the far wall back towards the nest (the lower part would bounce them into empty space). Ants that miss the nest on the way back then exploit the discontinuity in input headings to make it home by spiraling back along the discontinuity line. In all, these behaviors represent clear and frankly humorous "solutions" to the problem tasked by the GA, and prove that the training procedure does work. That said, though high scoring, these chromosomes are not real solutions to the generalized foraging problem, and after fixing the flaws in the environment, median training scores from these chromosomes went to nearly zero within 5 epochs. As shown in Figure 3, this exploitative solution did not match the gathering rate of the *DeterminAnt* agent, though it did vastly outperform agents trained on the fully random environment.

### B. Fully Randomized Problem Space

The failure of the *DominAnts* trained in an exploitable environment was the first example of a pattern that would continue

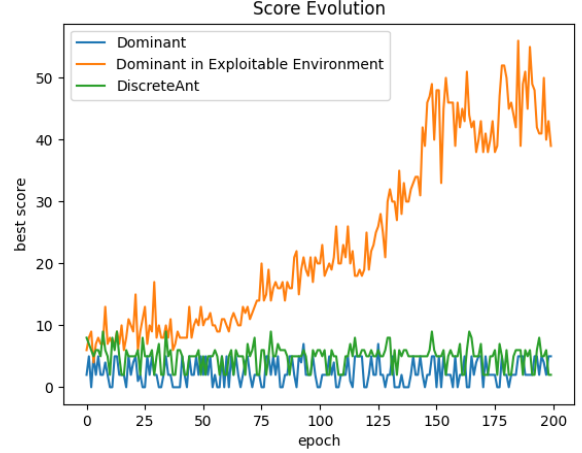


Fig. 2. Comparison of training curves for different agent implementations

unabated when conducting GA training on the corrected and fully randomized environment. After the success of the GA in finding an exploitative solution in its first pass, no further GA training tests using any agent architecture were able to achieve any meaningful score improvement above that of the randomly initialized chromosomes, as shown in blue and green in Figure 2.

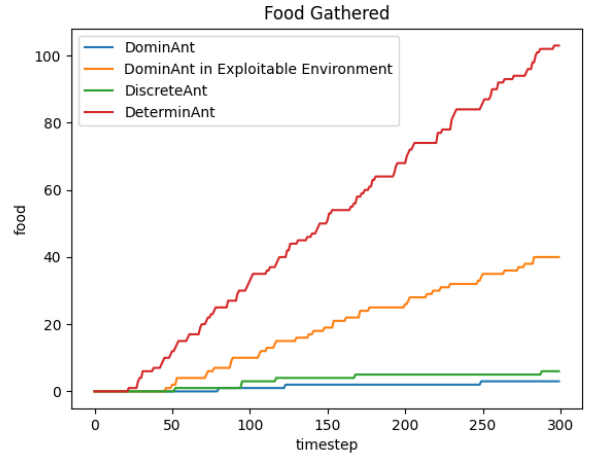


Fig. 3. Comparison of food gathered over time for different agent implementations

### C. Comparison with Gradient-Based RL Methods

To ensure that this dearth of quality chromosomes for the generalized case was not caused by some fundamental flaw in the GA training method, a standard policy-gradient RL method, REINFORCE, was applied to the *DiscreteAnt* architecture. The architecture was simplified to only output the *orientation\_delta* discrete action, with pheromone output being removed. This was done to simplify the foraging problem, and to ensure the REINFORCE algorithm was applied to a

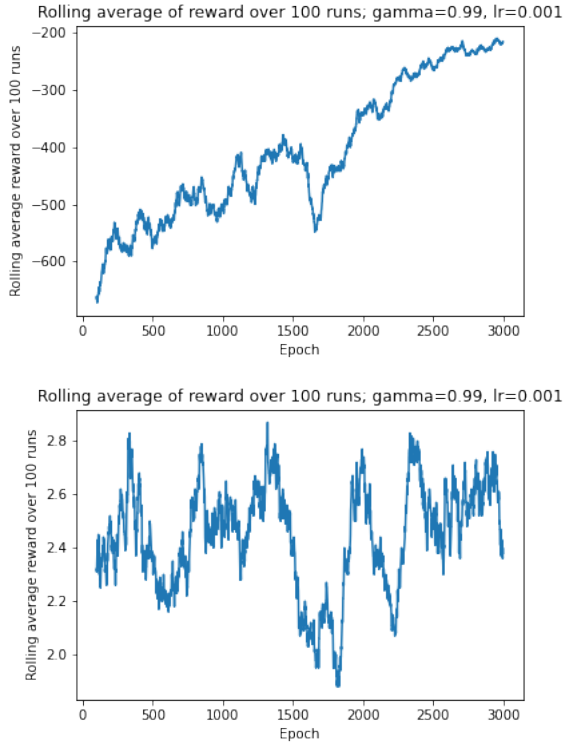


Fig. 4. Average scores over epochs for MountainCar (Top) and DiscreteAnt (bottom)

maximally simple action space to improve the fairness of the comparison in Figure 4. To compensate for this change, pheromone was automatically dropped into the agent’s current cell should the agent have food.

As a comparison, the REINFORCE algorithm was applied to the OpenAI MountainCar gym environment [9, 1]. While the environment is a single-agent system, the application of REINFORCE to *DiscreteAnt* is much the same, with the action trajectory of each agent acting as a training instance for the network in every training epoch. In both cases, the discount factor  $\gamma$  was set at 0.99 and the network used the ADAM optimizer to conduct backpropagation with a learning rate of 0.001. The mean training scores over 3000 episodes is shown in Figure 4.

While the actual reward scales are not comparable, Figure 4 does show that the training trend for a typical RL problem will trend upwards over training epochs. However, REINFORCE as applied to *DiscreteAnt* in the foraging environment failed to demonstrate that same trend, instead hovering consistently around a mean of 2.4. This further demonstrates the sheer complexity of generating emergent control for the ant foraging problem using a neural network. This result suggests that neural networks may be a mathematically suboptimal solution for this task, at least with current training techniques.

#### D. Reasons for Training Failure

Given the hundreds of processor-hours of training time poured into generating ultimately ineffective solutions to the

fully randomized foraging case, theories abound for why exactly the GA training procedure works so well on the exploitable case while neither the GA nor any traditional RL method works on the full problem formulation. The true answer is likely a combination of various factors and hundreds more hours of training will be required to conclusively validate these theories, but possibilities include:

1) *Continuous output spaces are too complex:* The *DominAnt* architecture (ironically named in hindsight) attempts to generate heading directions for the agent using a continuous activation function on a single neuron. By contrast, many RL implementations split the action space into discrete actions, and take the most likely of several output nodes at every time step. This distinction inspired the *DiscreteAnt* architecture which does perform marginally better than *DominAnt*, but changing this alone does not solve the problem.

2) *Behavior discontinuity is prohibitively large:* Optimal behavior for the foraging agents would involve entirely separate decision-making paradigms based on having or not having food. Though all the architectures seek to account for this by separating the degree of random walk from other outputs, an optimal weight distribution would still need to treat the `has-food` input as a behavioral switch. Such a large discontinuity may be hard for networks to encode. Solutions to this could include splitting the network into two in order to handle the different cases, but this was left unimplemented to try and conserve the single-network goal case.

3) *Solution space is very narrow:* Similar to the point above, it is quite possible that the solution space for optimal behavior would require the weights of the network to have extremely specific magnitudes with respect to one another. This would mean that finding the optimal solution would take an extremely long time with the GA given its stochastic search trajectory—possible future work could involve leaving the GA to run for thousands instead of hundreds of epochs over weeks instead of days.

4) *Regions surrounding optima do not form gradient:* Lastly, it may well be that the regions of the search space surrounding the optimal solution do not form any kind of gradient towards the solution. Though the GA does not require an explicit gradient to function, it does require that individuals close to the solution perform well in simulation so that those individuals are preferred for reproduction. A search space where the gradient pushes the algorithm away from the solution until actually at a narrow optima would be nearly impossible for any known network training algorithm to solve.

## VIII. FUTURE WORK

It is evident that there is a long way to go in order to train a neural network to approximate an optimal food foraging decision function. As noted above, much depends on the structure of inputs and outputs, and so more exhaustive testing is required to determine what configuration of local knowledge and movement capabilities are necessary to ensure that the agents are even able to reach an optimal solution. Given a network configuration that is provably able to reach

an optimal solution, further investigation of bleeding-edge algorithms for multi-agent RL using backpropagation could provide a path to a truly robust training solution, as could a more effective formulation of the action space and reward attribution for GA training. For RL algorithms, promising first steps would include gradient-based algorithms such as Multi-Agent Proximal Policy Optimization (MAPPO) [11], Independent PPO (IPPO) [14] and QMIX [12].

Beyond implementing new algorithms, converting the simulation environment to be compatible with the petting zoo environment paradigm would increase the impact of future work on this problem formulation, as petting zoo is a widely available and open-source RL project that provides a range of pre-programmed training algorithms [9][15].

## IX. CONCLUSION

Even though the GA training procedure was never able to generate an optimal or even particularly good solution for the generalized case with truly randomized food, the hundreds of hours of training results on at least five meaningfully different agent architectures did lead to a wealth of valuable knowledge about the dynamics of GA training and several good directions for future work, as noted above. Primary contributions of this work will eventually be verified answers for why this problem is so difficult for a neural network to solve—likely culprits include the stochasticity of the problem and large discontinuity in optimal behavior between having food and not having food, but more robust results are required to prove these conjectures. In all, the environment and training procedures are a useful platform for studying and learning more about the foraging problem and for investigating what kinds of learning solutions will be best for this use case.

Cooperative swarm behaviors represent an important part of the future of robotics and neural networks represent the cutting edge of decision making agents, so work that investigates the merging of these two new and important techniques has the potential for significantly improving the state of the art in both. The case for this merging of disciplines is made even stronger by the architecture’s similarity to biological reality—if nature can evolve organisms that cooperate based on the decisions of tiny node-based brains, it makes logical sense that a working computer simulacrum should be possible and powerful.

## CROSS-LISTING NOTE

For full transparency’s sake, the authors must note that this project was cross-listed by half the team for another Cornell Tech class, CS5787 Deep Learning. The RL and network design applications were the focus for those authors as part of the work for that project submission.

## REFERENCES

[1] Andrew William Moore. “Efficient memory-based learning for robot control”. In: (1990).

[2] M. Dorigo and Luca Maria Gambardella. “Ant colonies for the travelling salesman problem”. In: *Biosystems* 43 (2 1997).

[3] Benoît Calvez and Guillaume Hutzler. “Automatic tuning of agent-based models using genetic algorithms”. In: *International Workshop on Multi-Agent Systems and Agent-Based Simulation* (2005).

[4] Liviu Panait and Sean Luke. “Cooperative multi-agent learning: The state of the art”. In: *Autonomous agents and multi-agent systems* 11.3 (2005), pp. 387–434.

[5] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. “Real-time neuroevolution in the NERO video game”. In: *IEEE transactions on evolutionary computation* 9.6 (2005), pp. 653–668.

[6] M. Dorigo, M. Birattari, and T. Stutzle. “Ant colony optimization”. In: *IEEE Computational Intelligence Magazine* 1.4 (Nov. 2006), pp. 28–39.

[7] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. “Multi-agent reinforcement learning: An overview”. In: *Innovations in multi-agent systems and applications-1* (2010), pp. 183–221.

[8] Risto Miikkulainen et al. “Multiagent learning through neuroevolution”. In: *IEEE World Congress on Computational Intelligence* (2012).

[9] Greg Brockman et al. “Openai gym”. In: *arXiv preprint arXiv:1606.01540* (2016).

[10] Jayesh K. Gupta, Maxim Egorov, and Mykel Kochenderfer. “Cooperative multi-agent control using deep reinforcement learning”. In: *International Conference on Autonomous Agents and Multiagent Systems* (2017).

[11] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[12] Tabish Rashid et al. “Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 4295–4304.

[13] Kenneth O. Stanley et al. “Designing neural networks through neuroevolution”. In: *Nature Machine Intelligence* 1.1 24–35 (2019).

[14] Christian Schroeder de Witt et al. “Is Independent Learning All You Need in the StarCraft Multi-Agent Challenge?” In: *arXiv e-prints* (2020), arXiv:2011.

[15] Justin K Terry et al. “PettingZoo: Gym for Multi-Agent Reinforcement Learning”. In: *arXiv preprint arXiv:2009.14471* (2020).

[16] Felipe Such et al. “Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning”. In: *Uber AI Labs* (). URL: [arxiv.org/pdf/1712.06567.pdf](https://arxiv.org/pdf/1712.06567.pdf).