

Simulating Natural Selection through AI Agents of Neural Network-based Intelligence honed through Evolutionary AI Algorithm

Pilli, Rohan Saketh
BASIS Independent Fremont Upper School

Abstract

This project explores the simulation of natural selection within artificial ecosystems using Artificial Intelligence (AI) agents equipped with neural network-based intelligence. By integrating evolutionary AI algorithms (EA) with neural networks, the study investigates how environmental factors and agent-specific attributes influence evolutionary outcomes and emergent behaviors. AI agents, designed with variable traits (agent-specific attributes) such as size, speed, intelligence, and eyesight, are placed in a dynamic environment where survival and reproduction depend on acquiring energy through food consumption. Evolutionary principles are applied across generations, such as mutations introducing variability in neural network configurations, repopulation of agents under the successful criterion (acquiring food), and elimination of agents not under the successful criterion.

The research utilizes Python to develop the simulation, combining generational AI algorithms with an evolved neural topology framework to drive the decisions of the agent. Environmental variables (environmental factors) such as food abundance, proportions of rotten food, and spatial dimensions are manipulated to observe their effects on population dynamics and behavioral evolution. Initial investigations focus on basic survival behaviors, while subsequent experiments introduce more complex phenomena, including aggression, competition, and altruism.

The study provides insights into the dynamics of natural selection and the emergence of complex behaviors from simple computational rules. The application of this simulation demonstrates a model for biological processes like the adaptation of bacteria, single-celled organisms with limited function and simplistic behavior. Future extensions could explore advanced strategies for simulating cooperative behaviors and multi-agent interactions. This work contributes to artificial life, evolutionary computation, and the design of adaptable AI systems, bridging computational simulations with insights into biological and psychological behaviors.

Introduction

The project extends the original proposal of a neural-evolution multi-agent system. The initial plan included an energy budget for each agent, but this was dropped to simplify the model; I instead focusud on spatial interactions and size-based dynamics. The core idea remains: agents sense their local environment, process observations through neural controllers, and evolve over generations. I retained the two modes of interaction (altruistic vs competitive food sharing) as proposed. My work is closely related to studies of emergent cooperation in agent simulations: multi-agent systems can produce complex collective behaviors from simple rules. In particular, neuroevolution—evolving neural network weights by genetic algorithms—provides a suitable framework. My implementation uses PyTorch for batched neural computation on GPU, and Pygame for interactive visualization. Below we describe the environment setup, agent sensing and action pipeline, batch processing and evolution, food mechanics, and the UI visualization.

Methodology

The simulation world is a fixed-size square (side length ENV_SIZE=1000 units) with hard boundaries. Agents incur a small fitness penalty if they touch the walls (see below) and are kept inside the region. Three species of agents differ by physical size: *Large* (radius 6.0), *Medium* (3.0), and *Small* (1.5). These radii also determine movement speed (smaller agents move slightly faster, see below) and how much food they absorb. Food items are point-like objects of radius FOOD_RADIUS=2 placed randomly (100 initially) in the world. I also spawn a fixed number (NUM_CHASERS=10) of hostile chasers (radius 4.0) that pursue agents. These chasers move in cardinal directions (north/south/east/west) towards the nearest agent and “catch” it if they collide, freezing for 15 seconds and applying a fitness penalty (CHASER_PENALTY=30) to that agent. The world update code enforces the bounds by clamping positions to [radius, ENV_SIZE-radius] and subtracting EDGE_PENALTY points whenever an agent crosses a boundary. Thus agents must remain inside the arena to survive.

Each agent maintains its own (x, y) position and a neural “brain”. At every frame, an agent constructs an 11-dimensional input vector representing its local situation. The first four features are normalized distances to the four walls (environment boundary). Next, the agent finds the nearest food: it computes the Euclidean distance to all food objects and takes the smallest; it then encodes (normalized) distance and angle-to-food as two features. Similarly, it locates the nearest *other* agent in the entire population (excluding itself) and computes that agent’s distance, direction, and relative size (scaled by a base radius). Finally, it finds the nearest chaser and records its distance and angle. These are concatenated into a single tensor of shape [num_agents, 11]. If no food or no chasers are present, default values (max distance, zero angle) are used.

This state tensor is fed into the agent’s neural network. I’ve implemented a custom PyTorch module BatchNeuralNetwork that holds weight matrices for all agents simultaneously so that each agent has its own weight matrix. In the forward pass, I unsqueeze the input and perform batched matrix multiplies via torch.bmm with PyTorch’s ReLU activations.

(repeated for all 10 hidden layers, finally multiplying by self.w11). The output is a tensor of shape [num_agents, 4], giving logits for four discrete movement directions (up, right, down, left). The program takes argmax to pick the action for each agent. Converting these to motion vectors, they then interpret the index as an angle (multiplying by $2\pi/4$) and compute the displacement ($\cos \theta$, $\sin \theta$) multiplied by the agent’s step size. (Step sizes were scaled inversely with radius so smaller agents move faster.) The new position is clamped to stay within bounds, and any agent touching the wall has EDGE_PENALTY subtracted from its fitness.

When an agent comes within contact range of a food item (specifically, if the squared distance to the food center is $\leq (\text{agent_radius} + \text{FOOD_RADIUS})^2$), we mark that agent as an “eater” of that food for this frame. Each food tracks which agents have touched it (food_spot.eaters). A food has a 30-frame timer (EAT_FRAMES = 30); each frame that at least one agent touches it, the timer increments. When the timer reaches 30, the food is consumed and removed, and we distribute points to agents according to the current mode. In *altruism mode*, the agents split the food proportionally to their sizes.

Thus each eater’s reward is FOOD_SCORE scaled by (its radius / total radii). In *competitive (steal) mode*, instead only the largest agent gets the full reward (unless there is a tie, in which case no one scores). The code checks the maximum radius among fs.eaters and assigns FOOD_SCORE to that single agent. (These code snippets match the altruistic vs competitive behavior described in the proposal.) In either mode, multiple small agents can collectively benefit if near a food and share it; otherwise big agents can dominate a food alone.

After a fixed number of time-steps (one “generation”), each agent has accumulated a fitness score (sum of points from food, minus penalties). We then evolve each species’ population independently. Within each population, the program sort agents by fitness (higher is better) and select the top 10% (at least one) as “masters”. To form the next generation of N agents, for each new agent we randomly choose one master and copy its neural network weights. In implementation, this means each weight tensor uses tensor.copy_(tensor[choices]) to vectorize all copying at once. Finally, it applys mutation: for each weight element there is a small probability (rate=0.1) to add Gaussian noise (std = 0.25).

This perturbs some weights by random noise, injecting variation. If no agent scored (all fitness zero), it simply resets the population to random positions to avoid extinction. All populations then continue to the next generation. This mutation-based approach (selection + perturbation) is conceptually similar to other neuroevolution methods.

I built a separate Pygame-based visualizer to watch the simulation in real-time. The visualizer has a top panel of buttons (Start, Pause, Toggle Altruism, Save, Load, etc.), and renders the world and agents below. Agents are drawn as colored circles (green=large, blue=medium, yellow=small) at their (x,y) positions. The user can also enter a manual placement mode, select a species, and click to add agents one by one. The visualizer loads pretrained “best brains” so that the user can watch a single agent act with its neural weights fixed. The code saves each species’ best network to files named e.g. best_agent_radius_60_altruism.pth (where 6.0 radius is saved as 60). The load function then reads these files and reconstructs the weights (squeezing away the batch dimension) for a SingleAgentNetwork. For example, to load radius 6.0 in altruism mode we load best_agent_radius_60_altruism.pth. The Agent class in the visualizer mirrors the sensing logic of the simulation: it builds the same 11-feature input from nearby foods, agents, and chasers, runs its loaded network to pick a direction, and updates its position. The visualizer thus confirms and illustrates the behaviors discovered by the simulator’s evolutionary runs.

Results

Over many generations, I observed clear differences between the two modes. In steal (non-altruistic) mode, large agents overwhelmingly dominated food consumption. Since only the largest eater receives points, large agents acquired almost all the fitness and reproduced. Small agents tended to steer clear of contested areas and rarely scored. This matches intuition and validates the implementation. By contrast, in altruistic mode agents of different sizes clustered together near food patches. Because points are shared by size, even small agents gained some score by being in the vicinity when a larger neighbor ate. In practice I saw emergent “feeding groups” of mixed sizes around each food. As predicted, no single small agent could overpower a large one, but the cooperative reward sharing allowed lower tiers to survive and score. These behaviors were not hand-coded but arose naturally from the rules—an example of emergent cooperation in an evolving multi-agent system (consistent with principles of agent-based modeling).

This project demonstrates a modular simulation framework combining PyTorch-based neural agents, GPU batching, and Pygame visualization. The system is flexible: one can adjust parameters like the number of agents, mutation rate, etc., and observe the effects. The emergent clustering in altruism mode suggests rich dynamics can arise even in a simple world; analyzing these patterns might draw on evolutionary game theory or swarm models.

The batched network design proved efficient: processing hundreds of agents per frame with one PyTorch forward pass and a single CUDA kernel for each layer is much faster than naive Python loops. It leverages the GPU’s strength in parallel matrix operations (similar to data-parallel deep learning). I avoided low-level CUDA and simply relied on PyTorch tensors and torch.bmm, making the code straightforward and cross-platform.

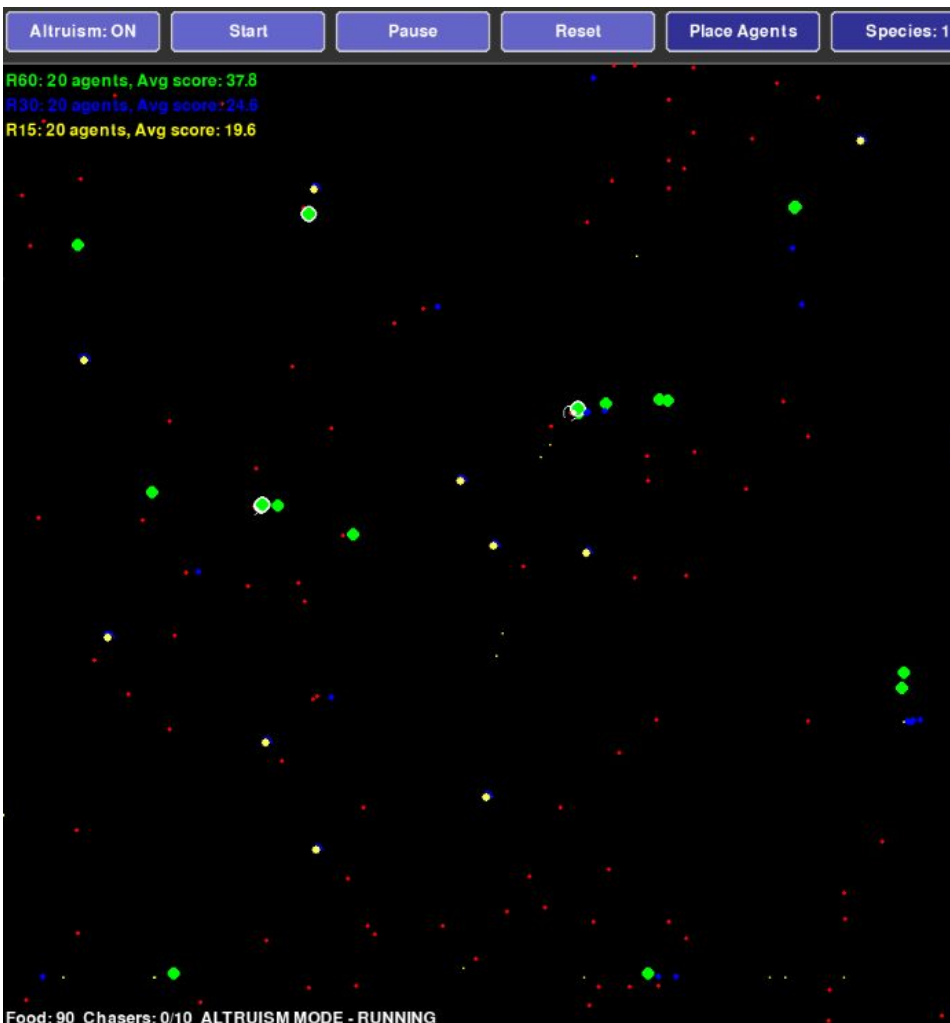
Future extensions could make the simulation more realistic. For example, one might allow continuous actions (smooth 2D velocity) instead of four fixed directions, possibly using a small drift component in the network output. Adding agent memory (e.g. recurrent networks or explicit state variables) could enable strategies like remembering past food locations. Introducing static obstacles or terrain would test navigation and pathfinding. We could also refine the evolution by incorporating crossover or more sophisticated selection (e.g. tournament selection).

Conclusion

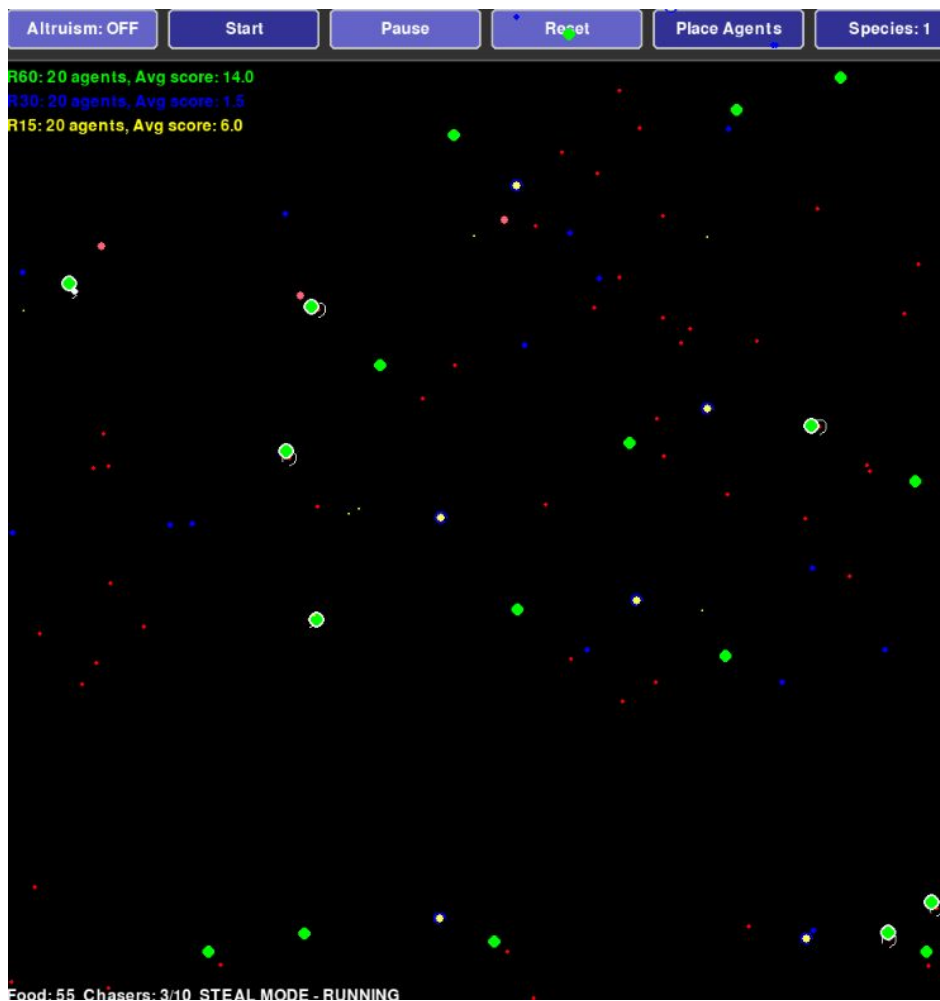
In summary, I’ve built and evolved a GPU-accelerated multi-agent neural simulation. Agents sense a 2D world (walls, food, neighbors, and chasers) and act via a ten-layer neural network. By processing all agents’ networks in parallel, PyTorch on CUDA made the simulation efficient. Food is scored either by sharing (altruism) or by a single winner, and we apply standard genetic selection + mutation each generation. I visualized the results in a Pygame GUI with controls and the ability to save/load neural “brains”. The system exhibited clear emergent behavior: altruism led to clustering strategies that benefited even small agents, whereas competition favored only the largest. This confirms the system’s validity and modularity. In future work, I hope to explore richer behaviors with added memory, continuous motion, or complex environments, leveraging this scalable PyTorch-based framework. The modular design (separate sensing, neural controller, and visualization components) makes such extensions straightforward to implement.

Images

ALTRUISTIC:



NON-ALTRUISTIC:



Acknowledgements

Many many thanks to Ms. Nagpal for organizing the Senior Project, and the BASIS community for fostering a strong and intellectual environment to learn and grow in.