Comparing Genetic Algorithms for Recurrent Neural Networks and Reflex Agents in Agar.io

Lucas Giebler
Joy Li
Michael Wosenyeleh
giebl007@umn.edu
li004001@umn.edu
wosen002@umn.edu

Abstract

Three AI approaches were compared for mass accumulation and survival in Agar.io: a Genetic Algorithm (GA) training Recurrent Neural Networks (RNNs), Simple Reflex Agents (SRAs), and Model-Based Reflex Agents (MBRAs). The GA used elitism, tournament selection, random crossovers, and decaying Gaussian mutation, with a fitness value based on food eaten, time alive, cells eaten, mass, and death. SRAs and MBRAs used hierarchical decision-making, with MBRAs maintaining a priority-based memory buffer. Using an offline Python recreation of the web-game Agar.io, 5 agents of each AI type were used in 10 game simulations. The SRAs achieved the highest average fitness of 1051.89 (28% survival), followed by MBRAs with −64.11 (4% survival), and GRUs with −467.16 (0% survival).

1 Introduction

The problem addressed in this project is enabling AI agents to interpret visual information and maintain a stable memory of the game https://agar.io/ to make on-the-fly decisions that maximize their score. The agent must process dynamic details and track the positions, masses, and movements of food and other players. This problem has broader implications beyond competitive video games to AI in vehicle automation or robotics [5]. These fields require processing noisy visual data and considering past experiences to decide how to execute safe, efficient maneuvers.

Agar.io is a multiplayer online game in which players control circular cells in a petri-dish-like environment. The objective is to grow larger by consuming smaller food pellets and other players while avoiding being consumed by larger opponents. The game features several mechanics that increase complexity: players can split their cell to capture prey at a distance, eject mass to feed allies or propel themselves, and must avoid viruses that cause large cells to explode into smaller fragments. These mechanics create a dynamic, adversarial environment where strategic decision-making is essential for survival and growth.

A fundamental challenge in Agar.io is the partial observability of the game state. Players can only see a limited viewport around their cell, and as the cell grows larger, the viewport expands but the relative density of visible information decreases. This creates a critical problem: objects that were recently visible may move out of view, yet their positions remain strategically relevant. A larger cell approaching from just outside the viewport represents an imminent threat, while a cluster of food pellets remembered from seconds ago may still be worth pursuing.

This partial observability problem motivates the need for memory-based agents that can maintain an internal model of the environment beyond what is currently visible. Unlike SRAs that react only to immediate stimuli, memory-based agents can leverage historical observations to make more informed decisions. This approach mirrors how human players naturally remember recent threats and opportunities, enabling them to plan escape routes or hunting strategies based on incomplete but temporally extended information.

Previous work on Agar.io AI has explored reinforcement learning approaches. Ansó et al. investigated how state representations impact Q-learning agents, finding that vision grids outperformed raw pixel input and that lower resolutions led to better performance [1]. However, their work used a simplified version of the game without viruses or splitting mechanics, limiting the complexity of required strategies. Multi-Objective Genetic Algorithms (MOGAs) such as NSGA-II have been applied to game AI to balance conflicting objectives like survival and aggression [10]. Quality-diversity algorithms like MAP-Elites offer an alternative paradigm that generates diverse high-performing behaviors rather than converging to a single optimal strategy [11]. These approaches suggest that maintaining behavioral diversity may be advantageous in dynamic game environments where no single strategy dominates.

The research questions addressed in this project are: (1) Can a model-based reflex agent with a priority-based memory buffer outperform simple reflex agents without memory in a partially observable environment? (2) How does a genetically trained recurrent neural network compare to rule-based agents (both simple and model-based) in terms of survival and mass accumulation? (3) Under what conditions do memory-based approaches provide advantages over simpler rule-based systems? These questions have implications for autonomous systems operating in dynamic,

partially observable environments where maintaining situational awareness beyond immediate perception is crucial for effective decision-making.

2 Brief Review

2.1 Hybrid Genetic Algorithm and Particle Swarm Optimization for Recurrent Network Design

Juang explored the efficacy of hybrid genetic algorithm and particle swarm optimization (HGAPSO), a combination of genetic algorithm (GA) and particle swarm optimization (PSO), when applied to training both recurrent neural networks (RNNs) and recurrent fuzzy networks (RFNs).

During training, HGAPSO enhances the top-half elites of the population through PSO, then performs GA reproduction, crossover, and mutation on those elites to fill the remaining half for the next generation. PSO enhancement nudges each elite's velocity in the problem space towards the current global best position in the population. Reproduction is a tournament selection scheme where two enhanced elites are randomly selected, and the elite with the better fitness is chosen. This is repeated to get the second parent, at which point the two parents produce 2 offspring using a two-point crossover operation. This operation randomly chooses two points in each parent's parameter sequence and swaps the parameters between those points. Finally, a 10% chance of random alteration is applied to each child parameter for mutation. Similarly, the GA implemented in this paper uses the same mutation and tournament selection system, with differences being that crossovers are applied randomly at every parameter.

Juang tests HGAPSO by training RNNs to predict the trajectories of two graphs defined by a sine-cosine wave function. Juang also tests RFNs by training them to control a standard nonlinear dynamic plant. Juang's HGAPSO applied to RNNs achieved a root mean squared error (RMSE) 56.59% lower compared to GA training and 21.02% lower than PSO. For RFNs, RMSE reductions of 58.6% against GA training and 31.33% against PSO was observed. Juang also compared his method to a gradient-descent based approach, Time-Domain Recurrent Backpropagation (TDRB), which the HGAPSO outperformed with an RMSE 55.49% lower than the RMSE of the TDRB method. [4]

2.2 Deep Reinforcement Learning for Pellet Eating in Agar.io

Ansó et al. investigated how state representations impact the performance of a reinforcement learning (RL) agent using Q-learning applied to convolutional neural networks (CNNs) and multilayer perceptrons (MLPs) to eat food and players in Agar.io.

To predict the Q-values, the authors trained an artificial neural network through backpropagation. While the authors rely exclusively on backpropagation to optimize Q-value networks, Juang's HGAPSO results in Section 2.1 demonstrate that evolutionary methods could outperform gradient descent-based approaches like backpropagation. This highlights a methodological gap in this paper's work, where the authors do not investigate whether a GA-based approach could yield faster convergence or superior policies under sparse or unstable RL signals.

For their experiments, the authors used a simplified version of the Agar.io game with only movable player cells and static food pellets, disabling viruses and disabling splitting and ejection functionality. The paper mentions how the inclusion of these additional features would require recurrent neural networks (RNNs). In this paper, these features are included, which lend credence to the use of memory-based agents like RNNs and MBRAs. The paper compares using a raw state representation with RGB and grayscale pixel values against manually defined vision grids, where each grid cell contains a value equal to the quantity of food in that area.

They found that vision grids outperformed raw pixel input, noting that this may be attributed to the faster convergence of vision grid RL agents, and given enough time, the agents using pixel values could have reached similar performance. Furthermore, lower resolutions for vision grids of 42x42 led to better performance than larger resolutions of 84x84, while the resolution of raw pixel values had a minimal impact on performance. [1]

2.3 Genetic Algorithm-Optimised Structure of Convolutional Neural Network for Face Recognition Applications

Rikhtegar et al. implement a genetic algorithm (GA) to train convolutional neural networks (CNNs) with support vector machines (SVMs) at the last layer for facial recognition.

The authors used mean-square error from the output layer of the CNN as the fitness function. Similarly to Section 2.1, they also employ tournament selection on a population of 100 agents and apply two-point crossover to produce children. They also implement elitism by retaining 1% of the mating population for the next generation. Enhancement of these elites, such as particle swarm optimization as explored in Section 2.1, could have further improved the convergence time or overall accuracy of the trained models. No mention is made of whether the authors used any mutation.

The authors tested their proposed method by applying their method to a facial recognition problem in two different datasets: the Yale dataset and the ORL dataset. The Yale dataset comprised 15 unique subjects and 165 images, and the ORL dataset comprised 40 unique subjects and 400 images. The paper experimentally found that their method achieved a recognition rate of 94.67% and 92.51% on the Yale and ORL datasets, respectively. Most other previous methods achieved lower recognition rates around 70%-80%. Other methods that achieved similarly high recognition rates around 94% did so

with higher dimensionality. Discussed further in Section 2.4, models with high dimensionality tend to be high-complexity. In neural networks, greater complexity means more nodes and/or more layers in the network, making the model more prone to overfitting and more expensive to train or make inferences with.

This paper demonstrated that GAs can reliably optimize CNNs, supporting the argument that GA training methods could have been viable and more effective for CNN optimization rather than reinforcement learning methods in Section 2.2. However, the facial images were lower in resolution than images used in Section 2.2, suggesting that GAs generally perform better on problems when the dimensionality of input data is low to moderate. [13]

2.4 Dimensionality Reduction Using Genetic Algorithms

Raymer et al. investigated the use of genetic algorithms (GAs) in dimensionality reduction for k-nearest neighbors (KNN) classification by applying GAs to select input features.

Instead of relying on linear statistical methods such as linear discriminant analysis (LDA), the authors propose representing each individual in a GA population as a hybrid transformation vector composed of real-valued feature-scaling parameters and a binary masking vector indicating which input features to retain or discard. The authors applied each transformed feature vector to a KNN classifier, and the resulting validation accuracy was defined as the fitness. Then, GA operations such as crossover, mutation, and selection were applied to the population to evolve feature transformations to maximize fitness.

The authors tested their GA by comparing it against other feature optimization methods (LDA, Bayes, Neural Net trained with backprop, CART tree, etc), combined with a KNN classifier to determine, using 21 clinical tests for thyroid dysfunction as input features, whether a patient should be diagnosed as hypothyroid. They also did the same experiment to classify appendicitis using 8 lab tests as inputs. They found that their GA method achieved 98.4% accuracy and 90.6% accuracy (the best compared to other methods or 1% off from the best) in correctly classifying hypothyroid and appendicitis in test data, respectively.

Notably, the results of all other methods showed lower testing accuracy than training accuracy, with drops ranging from 2% to 14%. In contrast, the GA's performance deviated by only 0.1% to 0.2% between training and testing, suggesting that the GA method was considerably more resistant to overfitting than the alternatives. The overfitting was likely due to the greater model complexity of the alternative methods, which included more irrelevant features that resulted in the models fitting the training data closely rather than learning a more generalizable representation.

Although previous sections like 2.1 and 2.3 implemented GAs to optimize the parameters of the neural networks themselves, the authors of this paper demonstrated that GAs are also effective in optimizing the input features of the networks, which can significantly improve performance. [12]

2.5 Multi-objective Optimization in Game AI Using Genetic Algorithms

This paper examines Multi-Objective Genetic Algorithms (MOGAs), specifically the NSGA-II variant, for optimizing game AI agents that balance conflicting in-game objectives. In their framework, each individual is evaluated against multiple fitness functions—for Agar.io, these could include mass gain (f_1), survival time (f_2), aggression score (f_3), and movement efficiency (f_4). The core innovation is non-dominated sorting: individuals are ranked into Pareto fronts based on whether they are outperformed by another individual on all objectives. The secondary "crowding distance" criterion preserves diversity by favoring solutions in less dense regions of the objective space.

The authors compared NSGA-II against: a) a single-objective GA using a weighted sum of objectives ($F_{\text{total}} = \sum w_i f_i$), and b) random search. Results showed the weighted-sum GA converged to narrow strategies dependent on chosen weights, often missing solutions when objectives were negatively correlated. For instance, high aggression weights produced agents that gained mass quickly but died early. In contrast, NSGA-II discovered a wide strategy spectrum, including "stealthy growers" prioritizing survival over direct confrontation. This contrasts with the elitist GA in Section 2.3, where preserving only the top 1% risks discarding valuable niche strategies early. The explicit diversity preservation in MOGA suits Agar.io's dynamic meta, where no single strategy dominates indefinitely, demonstrating that multi-objective optimization is necessary for robust performance in complex game environments. [10]

2.6 Quality-Diversity Algorithms for Game Agent Behavior Generation

This article introduces a paradigm shift from pure optimization to generating "quality-diversity" (QD) of solutions. The flagship QD algorithm, MAP-Elites, operates by defining a multi-dimensional behavioral descriptor space. For Agar.io, axes could include time near walls, average speed, and prey capture rate. Each generated individual is mapped to a cell in this grid based on its behavioral descriptor.

MAP-Elites maintains an archive containing the highest-fitness individual (the "elite") for each behavioral cell. During the search process, it selects a random cell, creates a mutated copy of its elite, and places the new individual in the appropriate cell if it has higher fitness or fills an empty cell.

The authors contrasted MAP-Elites with (a) traditional fitness-based GA and (b) "Novelty Search." Experiments showed the traditional GA converged to a single uniform peak, while

Novelty Search produced diverse but often low-performing individuals. MAP-Elites uniquely generated a wide range of high-performing behaviors—the entire archive contains viable, distinct strategies. This represents broader diversity than the Pareto-front approach in Section 2.5, which diversifies based on objectives rather than behavior. For Agar.io, MAP-Elites could yield a diverse "playbook" of competent agent types, from cautious wall-huggers to aggressive hunters, enabling robust testing and AI opponents with recognizable personalities. [11]

2.7 Genetic Algorithm-Based Neuroevolution for Game Agent Learning

This research presents Evolution Strategies (ES) as a viable alternative to gradient-based deep reinforcement learning. Unlike traditional genetic algorithms that mix and mutate solutions, ES works by fine-tuning a single core strategy. It tests new versions of this strategy by adding a little random noise to its parameters, then leans in the direction that proves most successful. The key insight is updating the parent parameters by a fitness-weighted average of these noise vectors, effectively guiding the search toward promising regions in the policy space. This black-box optimization approach completely bypasses the need for backpropagation and value function estimation.

The authors demonstrated ES's effectiveness on MuJoCo locomotion tasks and Atari games, where it achieved competitive performance with A2C and DQN while exhibiting superior training stability. A significant advantage was its massive parallelization capability—using 1,440 CPU workers to reduce wall-clock training time to just 10 minutes. This contrasts with the sequential hybrid approach of HGAPSO in Section 2.1 and offers a scalable pathway for training complex policies. For Agar.io, ES provides a robust optimization method that could handle the game's sparse rewards and non-stationary environment more effectively than traditional DRL methods. [14]

2.8 Automating Agent Design: Differentiable Architecture Search for Game AI

The author proposed DARTS (Differentiable ARchiTecture Search) to substantially accelerate and streamline the process of neural architecture discovery. The fundamental breakthrough lies in its formulation of architecture search as a differentiable optimization problem through continuous relaxation of the discrete search space. This approach effectively transforms architectural decisions from rigid, categorical choices into continuous, learnabble parameters that can be optimized via gradient descent. This gradient-based optimization is the key to its efficiency, replacing the need to evaluate countless discrete architectures from scratch. Consequently, the method enables simultaneous learning of both optimal architecture parameters and network weights within a unified optimization framework, establishing a new paradigm for efficient neural architecture search.

When evaluated on CIFAR-10, DARTS demonstrated remarkable efficiency, discovering competitive architectures in just 1.5 GPU days compared to 2,750 GPU days required by evolutionary methods. This three-order-of-magnitude improvement highlights the fundamental trade-off between exploration and exploitation: while evolutionary methods like ES (Section 6) excel at broad exploration, DARTS achieves unprecedented efficiency through gradient-based exploitation of a continuous search space. For Agar.io agent development, DARTS offers a practical solution for automating network architecture design, potentially discovering specialized structures for processing game states that would be difficult to manually engineer. [6]

2.9 Genetic Algorithm Approach to Compute Mixed Strategy Solutions for General Stackelberg Games

Gottipati and Paruchuri develop a genetic algorithm to compute mixed-strategy solutions in General Stackelberg Games (GSGs), where a leader commits to a probabilistic policy and a follower optimally responds. Each chromosome encodes a mixed strategy as a normalized probability vector over leader actions. The GA initializes with randomly sampled distributions but seeds the population with the best deterministic strategy to avoid weak early generations. Fitness is computed using a bilevel evaluation procedure: for each chromosome, the follower's best response is obtained via an argmax over payoff matrices, and the resulting expected reward for the leader becomes the fitness value. The evolutionary loop uses tournament selection, Simulated Binary Crossover (SBX), and a hybrid mutation operator that performs both exhaustive and random local probability perturbations, followed by normalization to maintain valid distributions. A conservative replacement rule accepts offspring only when they outperform their parents, accelerating convergence. Experiments show that the GA performs competitively with DOBSS, the classical exact solver, while scaling more efficiently to larger strategic games.

Compared with previous sections, this paper demonstrates a different but complementary use of genetic algorithms. Unlike Sections 2.1 and 2.3, which apply GA or hybrid GA-PSO methods to optimize neural network weights or architecture, this work evolves strategic probability distributions rather than model parameters. For Agar.io, this offers a valuable perspective: while earlier methods optimize an agent's low-level control or representation, the Stackelberg GA provides a pathway to evolve high-level, mixed strategic tendencies such as probabilistic aggression, or evasion based on how opponents are predicted to respond. This strategic optimization complements the prior neural, objective, and diversity-focused methods, showing how evolutionary algorithms can

operate not only on agent mechanics but also on adversarial decision policies in multi-agent games. [3]

2.10 Genetic Algorithm for Artificial Neural Networks in Real-Time Strategy Games

Widhiyasana et al. apply a Genetic Algorithm (GA) to evolve the weights of an Artificial Neural Network (ANN) that controls autonomous units in a real-time strategy (RTS) game environment. Each chromosome represents the complete parameterization of the ANN 720 weights connecting input, hidden, and output layers. A population of 56 chromosomes is maintained, matching the 56 units simulated in a battle scenario. Fitness is computed directly from combat performance: each ANN-controlled unit participates in a simulated match, and its score incorporates movement efficiency, successful attacks, avoidance of damage, and tactical positioning. Because every chromosome is embedded in an active battle, the GA's evaluation loop tightly couples policy refinement with dynamic game feedback. Tournament selection chooses parents, and one-point crossover recombines their weights; mutation then perturbs individual weight values with small, randomized deviations. The authors experimentally tune genetic parameters across 4000 generations, finding that a 0.6 crossover rate and 0.09 mutation rate consistently produce the strongest emergent behaviors. The result is an evolved ANN controller that demonstrates noticeably improved micromanagement; more precise spacing, better target selection, and smoother pursuit/retreat transitions compared to hand-crafted heuristics. Its RTS-based fitness design is closer to Section 2.2's environment-driven reinforcement learning, except that the GA replaces gradient-based optimization with evolution-driven self-play. Unlike the multi-objective or diversity-oriented approaches of Sections 2.5 and 2.6, this method pursues a single goal: maximizing battle effectiveness through ANN weight refinement.

For Agar.io, this work illustrates how a GA can be tightly integrated with gameplay simulation to evolve agents that learn effective movement and combat micromechanics. The RTS reward components avoiding damage, maintaining spacing, chasing vulnerable enemies, and maneuvering efficiently map almost directly to Agar.io behaviors such as evading larger cells, controlling distance, maximizing pellet collection paths, and opportunistic aggression. This makes the approach particularly valuable as a model for evolving reflex-level controllers in Agar.io. [15]

2.11 Unifying Zeroth-Order Optimization and Genetic Algorithms for Reinforcement Learning

Nakashima and Kobayashi introduce Ancestral Reinforcement Learning (ARL), a framework that unifies Zeroth-Order Optimization (ZOO) with Genetic Algorithms (GA) to leverage the strengths of both population-based search and gradient-free optimization. ZOO estimates gradients by perturbing a master policy with noise and correlating reward differences with those perturbations, enabling robust policy improvement without value-function estimation. GA, in contrast, evolves a diverse population of policies through mutation and fitness-based selection, promoting wide exploration but lacking a principled gradient direction. ARL bridges these paradigms by replacing random mutation in GA with an ancestral-learning update, where each agent adjusts its policy toward its parent's empirical action distribution, effectively using the parent's demonstrated behavior as a surrogate gradient signal. This works because selection creates survivorship bias, since parents that receive higher returns disproportionately influence the next generation, so imitating them statistically mimics ascent along the reward gradient. ARL produces an algorithm that retains GA's population diversity while achieving directional improvement similar to ZOO, and experiments show ARL consistently matches or exceeds the performance of either method in isolation.

Compared to earlier approaches, ARL introduces a more principled mechanism for combining evolutionary search and policy-gradient reasoning. Unlike pure GA mutation in Source 2.10, which perturbs network weights blindly, ARL uses trajectory-driven updates that exploit behavioral history. And while ZOO uses perturbation sampling around a single master policy, ARL preserves a full population, allowing broader exploration similar to the multi-policy approaches in Sources 2.5 and 2.6. For Agar.io-like environments characterized by adversarial interactions, spatial maneuvering, and noisy reward landscapes ARL offers a compelling hybrid: the GA component supports the exploration of diverse evasion, aggression, and farming strategies, while the ancestral-gradient step provides a directional improvement mechanism even when rewards are sparse or the environment is non-differentiable. This makes ARL particularly suited for learning robust, high-level survival strategies in Agar.io, such as adaptive threat avoidance or opportunistic pursuit policies shaped by the statistical successes of prior generations. [8]

2.12 Evolving Effective Microbehaviors in Real-Time Strategy Games

This research presents a genetic algorithm (GA) framework for evolving effective microbehaviors in real-time strategy (RTS) combat scenarios, focusing on fine-grained unit control such as kiting, focusing fire, and retreat logic. Their system represents each microbehavior as a vector of tunable parameters governing timing, movement offsets, threat thresholds, and weapon cooldown exploitation. A behavior is evaluated by embedding it directly into simulated unit skirmishes, where it is used by an entire team of units. The performance of that team measured primarily through combat outcomes, damage dealt, and unit survival is used as the fitness of the chromosome representing that behavior. This allows the

GA to search the space of tactical micro-maneuvers without manually defining which behavioral patterns lead to success.

The algorithm operates by first generating a population of candidate microbehavior parameter sets. Each chromosome encodes a compact behavior specification such as movement radius, approach angles, firing thresholds, or timing delays and each is tested by running a full RTS skirmish. Selection is performed using a fitness-proportionate scheme, where high-performing behaviors produce more offspring in the next population. Crossover recombines segments of behavior parameters between two parents, while mutation introduces small random perturbations to encourage diversity. The evolutionary process is environment-grounded: the fitness of a behavior emerges from observing how well it performs in repeated, variable combat simulations. This produces microbehaviors that can exploit subtle, emergent mechanics such as optimal stutter-step timing that would be extremely difficult to design manually.

This work most closely resembles Source 2.10, where GA evolves neural network weights for unit control in RTS games. However, Lai et al. evolve explicit tactical rule-parameters rather than ANN weights, giving the GA direct control over interpretable micro-level behavior. Source 2.12 focuses on real-time, reflexive, continuous decision-making rather than long-horizon planning or probabilistic strategy mixing. Compared to diversity-focused approaches (Sources 2.5–2.6), the objective here is singular combat performance without seeking behavioral variety.

The microbehavior optimization described in this paper maps closely to the behaviors needed for high-performing Agar.io agents. Agar.io's gameplay requires smooth, continuous, and reactive control: dodging larger cells, pursuing smaller targets, maintaining spacing, timing splits, and navigating complex spatial configurations. Much like an RTS skirmish, survival depends on real-time micromanagement. A GA that evolves parameterized movement and combat microbehaviors—such as retreat thresholds, pursuit aggressiveness, or split-timing heuristics—could allow Agar.io agents to discover emergent tactics such as optimized circling paths, evasive motion patterns, or efficient pellet path taking. [7]

3 Approach

The algorithms evaluated in this paper are a Genetic Algorithm (GA) for training Gated Recurrent Unit (GRU) neural networks, Simple Reflex Agents (SRAs), and Model-Based Reflex Agents (MBRAs). The GRU agents learn behaviors implicitly through evolutionary optimization, while the SRA and MBRA use explicit rule-based decision-making. The key difference between SRA and MBRA is that the MBRA maintains a memory buffer of its environment, storing the locations and areas of objects that have left the field of view.

The reflex agents follow a hierarchical decision structure that prioritizes avoiding larger threats, consuming smaller players, and gathering food when safe. This priority ordering reflects the asymmetric costs in Agar.io: death (from being eaten) is permanent, while missing a food pellet only delays growth.

3.1 Genetic Algorithm

The use of evolutionary algorithms to train neural networks provides a robust alternative to gradient-based methods [2]. Since this approach includes no backpropagation, training with genetic algorithms can avoid the problem of exploding or vanishing gradients that plague the training of RNNs [9].

The GA is applied to various types of RNNs using different PyTorch cells: Standard Recurrent Neural Network (SRNN), Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU). The GA starts with 50 RNN agents, where the initial numbers of SRNNs, LSTMs, and GRUs are assigned as follows:

$$n_{SRNN} = \lfloor p_{SRNN} \times 50 \rfloor \tag{1}$$

$$n_{LSTM} = \lfloor p_{LSTM} \times 50 \rfloor \tag{2}$$

$$n_{GRU} = 50 - n_{SRNN} - n_{LSTM} \tag{3}$$

The initial proportions p_{SRNN}, p_{LSTM}, p_{GRU} are 0.33333, 0.33333, and 0.33334, respectively. Each RNN is initialized with random weights and biases.

At the end of each generation, agents are sorted by fitness, where the top 50% of agents in the population are added to three separate reproduction pools divided based on the type of RNN (SRNN, LSTM, and GRU). This division ensures reproduction doesn't attempt crossover between agents of different types of RNNs. Candidates are reproduced from these reproduction pools via tournament selection, which involves choosing 3 candidate agents from a pool at random, and the candidate with the maximum fitness is selected as the first parent. This is repeated for the second parent, where one child is then created from those two parents through random crossover; Each parameter has a 50% chance of being inherited from the first parent, otherwise the second parent.

After applying random crossover, every weight and bias of the child is mutated by a Gaussian noise perturbation with standard deviations σ_w and σ_b that start at values of 1.0 and 0.5, respectively, and probabilities of mutation p_w and p_b of 0.05 and 0.025, respectively. σ_w and σ_b are reduced by 0.95 after each generation, similar to learning rate decay [16], and are reset to their initial values once they drop below an epsilon value of 0.0001.

When the size of an RNN's reproduction pool drops below three, the number of candidates reduces to the size of the pool. Furthermore, each RNN type has an extinction counter that counts the number of generations that its reproduction pool had a size of 0. RNNs with an extinction counter below a defined extinction threshold of 5 generations will be forcibly reproduced among their remaining RNNs, maintaining at least 1 child for the next generation. RNNs with a counter

exceeding 5 do not produce offspring, and, as a result, go extinct in the following generation.

Elitism is implemented by retaining the top 10% of the population for the next generation without any mutation or alterations.

Training is carried out through 5 simulations run in parallel. Agents from the population are copied and placed in random locations in each of the simulations, so every simulation has 50 RNN agents. Each simulation is a 2000×2000 bounded environment with 700 total food pellets and 20 viruses distributed randomly. The food and virus counts are maintained by respawning eaten food and viruses in random locations. Each simulation terminates after a simulated 300 seconds or 18000 total frames (based on a normal game speed defined as 60 FPS) elapse, or when 0 RNN agents are alive. The simulations for the first 25 generations include only the 50 RNN agents with no predators, while simulations above 25 generations include 25 randomly distributed Simple Reflex Agents to act as predators. After all 5 simulations have terminated, the fitness for all 50 agents within each simulation is calculated, and the final fitness of a specific agent is calculated as its average fitness across the 5 simulations.

3.2 Simple Reflex Agent Architecture

The decision logic of the SRA follows a hierarchical rule-based structure. First, the agent checks for nearby threats within a distance threshold of 350 units. If a threat is detected, the agent calculates an escape vector pointing away from the closest threat, with urgency proportional to the proximity of the threat. Second, if no immediate threats are present, the agent pursues the nearest prey. If the prey is within split distance and the agent's minimum cell mass after splitting would still exceed the prey's mass by 20%, the agent executes a split attack. Third, in the absence of threats and prey, the agent moves toward the nearest food pellet. Finally, the agent applies virus avoidance behavior when its area exceeds 300 units, steering away from any virus within 100 units to prevent explosion.

3.3 Model-Based Reflex Agent Architecture

The MBRA maintains a memory buffer that stores information about objects that have left the agent's field of view. The memory buffer is organized into four separate lists corresponding to different object types: threats (larger players), prey (smaller players), food pellets, and viruses. Each list has a maximum capacity of 100 items to prevent unbounded memory growth.

Each memory item stores the object's position, area, and a priority value that determines its relevance to decision-making. When an object is first observed, it is assigned an initial priority based on its type:

$$p_i(0) = \begin{cases} 1.0 & \text{if } label(i) = \text{“threat”} \\ 0.8 & \text{if } label(i) = \text{“prey”} \\ 0.7 & \text{if } label(i) = \text{“virus”} \\ 0.6 & \text{if } label(i) = \text{“food”} \end{cases}$$

Threats receive the highest priority to ensure the agent maintains awareness of dangerous opponents even after they leave the viewport. Prey and viruses receive intermediate priorities, while food receives the lowest priority since missing a food pellet has minimal strategic cost.

The priority of each memory item decays exponentially over time according to the formula:

$$p_i(t) = p_i(0) \cdot \gamma^{(t-t_0)} \tag{4}$$

where $p_i(0)$ is the initial priority, $\gamma = 0.985$ is the decay factor, t is the current simulation tick, and t_0 is the tick when the object was first observed. When an object's priority falls below a threshold of 0.05, it is removed from the memory buffer. If an object that is already in memory becomes visible again, its priority is reset to the initial value and its position is updated.

The decision logic of the MBRA follows the same hierarchical rule-based structure as the SRA. However, the MBRA merges currently visible objects with remembered objects from the memory buffer to form a complete situational picture. For remembered objects, the priority is further adjusted by distance: objects farther from the agent receive lower effective priority. This distance weighting ensures that the agent prioritizes nearby remembered objects over distant ones, balancing the value of memory with spatial relevance.

4 Experiments and Results

Each generation, 50 GRU agents play five randomized games to average out stochastic variation. Fitness combines normalized measures of food eaten, time alive, cells eaten, and maximum mass. The top 25 reproduce, and the best agent is evaluated against reflex agents. These experiments aim to compare three AI approaches: a genetically trained GRU neural network, a Simple Reflex Agent (SRA), and a Model-Based Reflex Agent (MBRA). The comparison evaluates their performance in terms of survival and mass accumulation in a partially observable environment. The fitness function used for evaluation is defined as:

$$F = w_f \cdot f + w_t \cdot t + w_c \cdot c + w_s \cdot s - w_d \cdot d \tag{5}$$

where f is food eaten, t is the proportion of time alive, c is cells eaten, s is the score (mass), and d is a binary death indicator. The weights used were $w_f = 0.5$, $w_t = 100.0$, $w_c = 50.0$, $w_s = 0.75$, and $w_d = 500.0$.

4.1 Experimental Setup

The experiments were conducted using Python 3.12 with the following libraries: OpenCV for visualization, PyTorch for neural network operations, NumPy for numerical computations, and multiprocessing for parallel simulation execution. The code was executed on a Windows machine equipped with a 13th Gen Intel Core i9-13950HX CPU and a NVIDIA GeForce RTX 4060 Laptop GPU. All simulations were run in headless mode (without graphical rendering) to maximize computational efficiency.

The hardware configuration primarily affects the wall-clock time required for training and evaluation, but does not influence the relative performance comparison between the three agent types. Since all agent types are evaluated under identical environmental conditions and computational constraints, any hardware-specific effects would apply uniformly to all approaches. The fitness scores and survival rates reported are deterministic functions of agent behavior and game mechanics, independent of the underlying hardware platform. This ensures that the comparative results are reproducible and generalizable across different computing environments.

Table 1 shows the configuration parameters used for the comparative experiments between GRU, SRA, and MBRA agents. The GRU agents were pre-trained using the genetic

Table 1. Experiment Configuration Parameters

Parameter	Value
GRU Agents	5
SRA Agents	5
MBRA Agents	5
Number of Simulations	10
Map Size	1500×1500
Food Count	600
Virus Count	20
Duration per Simulation	300 seconds
Frames per Second	60

algorithm described in Section 3.1, with a population of 50 agents over approximately 100 generations. The GRU architecture consisted of a single hidden layer with 72 units, taking a 12×8 vision grid with 3 features per cell (288 total inputs) and producing 4 outputs (movement x, movement y, split, eject). The best-performing GRU parameters were saved to checkpoint files and loaded for the comparative evaluation. Note that the training environment used a 2000×2000 map with 700 food pellets, while the evaluation environment used a smaller 1500×1500 map with 600 food pellets to test generalization to different conditions.

4.2 Results

Table 2 presents the overall performance comparison between GRU, SRA, and MBRA agents across all simulations.

Table 2. Three-Way Performance Comparison: GRU vs SRA vs MBRA

Metric	GRU	SRA	MBRA
Sample Size	50	50	50
Mean Fitness	−467.16	1051.89	−64.11
Standard Deviation	19.75	1584.21	974.20
Minimum Fitness	−484.91	−481.44	−483.49
Maximum Fitness	−381.62	6141.37	5341.85
Survival Rate	0%	28%	4%

The experimental results reveal a clear performance hierarchy: the Simple Reflex Agent (SRA) achieved the highest mean fitness of 1051.89, followed by the Model-Based Reflex Agent (MBRA) with −64.11, and the genetically trained GRU with −467.16. Notably, the SRA demonstrated the highest survival rate at 28%, while the MBRA achieved 4% and the GRU failed to survive any simulations (0% survival rate).

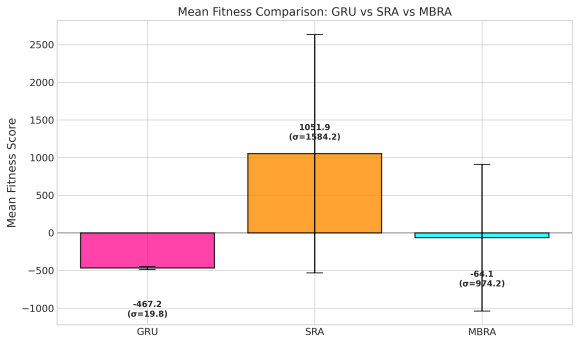

Figure 1. Mean fitness comparison between GRU, SRA, and MBRA agents with standard deviation error bars. The SRA achieved the highest mean fitness (1051.89), followed by MBRA (−64.11) and GRU (−467.16).

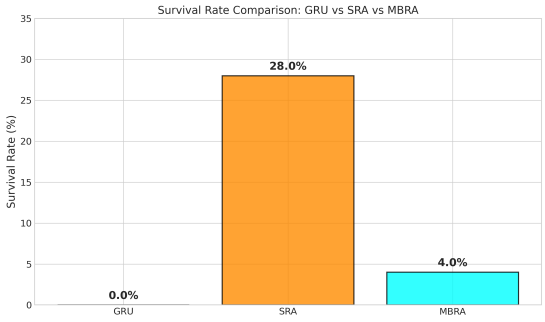

Figure 2. Survival rate comparison. SRA agents achieved the highest survival rate (28%), followed by MBRA (4%) and GRU (0%).

The SRA's superior performance can be attributed to its straightforward, rule-based decision-making that prioritizes

immediate threat avoidance and efficient food collection. The high variance in SRA fitness ($\sigma = 1584.21$) reflects its opportunistic nature: when conditions are favorable, SRA agents can achieve exceptional fitness scores (maximum of 6141.37), but they also experience failures resulting in negative fitness values.

The MBRA, despite its memory buffer system, achieved lower mean fitness than the SRA. This suggests that in the experimental conditions, the memory buffer may have introduced complexity that did not translate to improved performance, or that the memory decay parameters may need further tuning. The MBRA's survival rate of 4% indicates that while the memory system provides some advantage, it was not sufficient to consistently outperform the simpler SRA approach.

The GRU agents performed worst across all metrics, with a mean fitness of −467.16 and 0% survival rate. This consistent failure suggests that the genetic algorithm training was insufficient to evolve effective survival strategies, or that the neural network architecture and input representation were not well-suited to the partially observable environment of Agar.io.

Table 3 shows the detailed results for each of the 10 simulations, including the number of surviving agents and average fitness for each agent type. The table demonstrates the consistent superiority of SRA agents across most simulations, with SRA achieving positive fitness scores in 9 out of 10 simulations, compared to MBRA (3 out of 10) and GRU (0 out of 10).

Table 3. Per-Simulation Results

Sim	GRU Alive	SRA Alive	MBRA Alive	GRU Fitness	SRA Fitness	MBRA Fitness
1	0/5	5/5	3/5	−473.66	1136.78	805.98
2	0/5	5/5	2/5	−450.21	1229.93	−239.92
3	0/5	4/5	2/5	−466.32	1109.34	−220.08
4	0/5	5/5	2/5	−467.55	1178.68	−278.40
5	1/5	4/5	4/5	−456.44	368.91	608.78
6	0/5	5/5	2/5	−476.08	651.44	−373.40
7	0/5	4/5	3/5	−470.42	481.10	−286.70
8	0/5	5/5	1/5	−477.61	2633.77	−265.23
9	0/5	5/5	3/5	−459.53	1249.99	55.52
10	0/5	5/5	1/5	−473.82	479.02	−447.64

5 Analysis

5.1 Simple Reflex Agent Performance

The SRA's superior performance can be attributed to its straightforward, hierarchical decision-making structure. The agent prioritizes threat avoidance above all else, then pursues prey, and finally collects food when safe. This clear priority ordering aligns perfectly with the asymmetric costs in Agar.io: death is permanent, while missing food only delays growth. The SRA's high variance ($\sigma = 1584.21$) reflects its opportunistic nature: when conditions are favorable, SRA agents can achieve exceptional fitness scores (maximum of 6141.37), demonstrating the effectiveness of simple, well-designed rules in dynamic environments.

The SRA's 28% survival rate indicates that its threat avoidance mechanism is sufficiently effective to survive in approximately one-quarter of simulations. This performance suggests that for the experimental conditions tested, the immediate visibility of threats within the viewport was sufficient for effective decision-making, and the added complexity of memory may not have been necessary.

5.2 Model-Based Reflex Agent Underperformance

Contrary to expectations, the MBRA underperformed compared to the SRA despite its memory buffer system. The MBRA achieved a mean fitness of −64.11 and only a 4% survival rate. This result suggests several possible explanations: (1) the memory buffer's priority decay mechanism may have been too aggressive, causing the agent to forget threats too quickly, (2) the memory buffer may have introduced decision-making complexity that slowed response times to immediate threats, or (3) the experimental conditions may not have sufficiently rewarded memory-based strategies, as threats were often visible within the viewport before becoming critical.

The MBRA's memory buffer system, while theoretically advantageous for partially observable environments, may require more careful tuning of decay parameters and priority thresholds. The exponential decay factor of $\gamma = 0.985$ may have been too high, causing remembered threats to lose relevance too quickly. Additionally, the memory buffer's distance weighting mechanism may have reduced the effective priority of remembered objects, potentially causing the agent to underestimate threats that had moved out of view.

5.3 Causes of GRU Failure

The GRU agents demonstrated complete failure, with a mean fitness of −467.16 and 0% survival rate. The consistently negative fitness scores can be attributed primarily to the death penalty term in the fitness function. With $w_d = 500.0$, each death subtracts 500 points from the fitness score. Since all GRU agents died during the simulations, every agent received this significant penalty that their food collection and survival time could not overcome. The experimental results reveal a surprising finding: the Simple Reflex Agent (SRA) outperformed both the Model-Based Reflex Agent (MBRA) and the genetically trained GRU. The SRA achieved a mean fitness of 1051.89 with a 28% survival rate, compared to the MBRA's mean fitness of −64.11 (4% survival) and the GRU's mean fitness of −467.16 (0% survival). This hierarchy challenges the initial hypothesis that memory-based approaches would provide significant advantages in partially observable environments.

The GRU agents demonstrated poor threat avoidance behavior, frequently moving toward larger opponents rather than away from them. This suggests several potential issues: (1) the genetic algorithm training may not have sufficiently optimized the threat detection and evasion components of the neural network, (2) approximately 100 generations may

have been insufficient to evolve robust survival strategies, (3) the discretized 12×8 vision grid representation may have lost critical spatial information needed for effective threat assessment, or (4) the difference between training (2000×2000 map) and evaluation (1500×1500 map) environments may have impacted generalization.

The GRU's low variance ($\sigma = 19.75$) reflects its consistently poor performance across all simulations, with fitness scores tightly clustered around −467. This low variance, combined with the complete absence of positive fitness scores, indicates that the genetic algorithm failed to discover any effective survival strategies.

5.4 Variance and Performance Patterns

The three agent types exhibited substantially different variance patterns. The SRA showed the highest variance ($\sigma = 1584.21$), reflecting its opportunistic behavior and the wide range of outcomes depending on environmental conditions. The MBRA showed moderate variance ($\sigma = 974.20$), suggesting inconsistent performance that may be related to the effectiveness of its memory buffer in different scenarios. The GRU showed the lowest variance ($\sigma = 19.75$), indicating consistently poor performance across all conditions.

The performance hierarchy (SRA > MBRA > GRU) suggests that in the experimental conditions tested, simplicity and clarity of decision rules may be more valuable than sophisticated memory systems or learned behaviors. However, this finding may be specific to the particular experimental setup, and different conditions (such as larger maps, more complex environments, or longer simulation durations) might favor memory-based approaches.

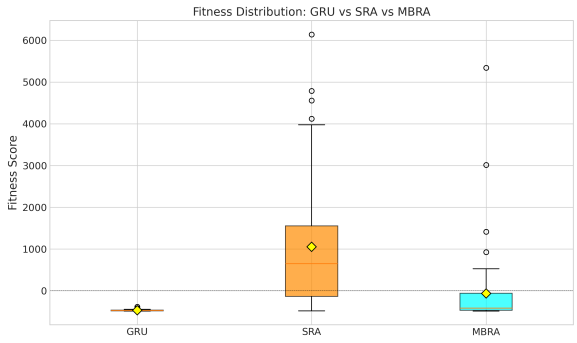

Figure 3. Box plot showing the fitness distribution for GRU, SRA, and MBRA agents across all 50 samples. The diamond markers indicate mean values. SRA exhibits the highest variance and best performance, while GRU shows consistently poor performance with low variance.

6 Conclusions

This study compared three AI approaches in the partially observable, adversarial environment of Agar.io: a genetically trained Gated Recurrent Unit (GRU) neural network, a Simple Reflex Agent (SRA), and a Model-Based Reflex Agent (MBRA) with a priority-based memory buffer. The experimental results revealed an unexpected performance hierarchy: the SRA achieved the highest mean fitness of 1051.89 with a 28% survival rate, followed by the MBRA with −64.11 (4% survival), and the GRU with −467.16 (0% survival).

The SRA's superior performance demonstrates that well-designed, hierarchical rule-based systems can be highly effective in dynamic, partially observable environments. The agent's clear priority ordering (threat avoidance > prey pursuit > food collection) aligns perfectly with the asymmetric costs in Agar.io, enabling it to make rapid, effective decisions based solely on immediate observations. This finding suggests that for certain classes of problems, simplicity and clarity of decision rules may be more valuable than sophisticated memory systems or learned behaviors.

The MBRA's underperformance relative to the SRA was unexpected, as the memory buffer system was designed to provide advantages in partially observable environments. The results suggest that the memory buffer may require more careful tuning of decay parameters and priority thresholds, or that the experimental conditions did not sufficiently reward memory-based strategies. The MBRA's 4% survival rate indicates that while the memory system provides some advantage, it was not sufficient to consistently outperform the simpler SRA approach.

The complete failure of the GRU's (0% survival rate) highlights the challenges of evolving effective neural network controllers through genetic algorithms. Several factors may have contributed: insufficient training generations, inadequate population size, loss of spatial information in the discretized vision grid representation, or poor generalization from training to evaluation environments. The consistently negative fitness scores and low variance indicate that the genetic algorithm failed to discover any effective survival strategies.

The findings have implications beyond game AI. The success of simple rule-based systems suggests that for autonomous systems operating in dynamic environments, well-designed heuristics may sometimes outperform more complex learned or memory-augmented approaches. However, the results may be specific to the experimental conditions, and different scenarios (larger maps, longer durations, or more complex environments) might favor memory-based or learned approaches. Future work could explore hybrid systems that combine the clarity of rule-based systems with the adaptability of learned behaviors, potentially leveraging the strengths of both paradigms.

Future directions include: (1) investigating why the SRA outperformed the MBRA despite the memory buffer's theoretical advantages, (2) tuning MBRA memory decay parameters and priority thresholds to improve performance, (3) training GRUs for more generations with larger populations

or different architectures to improve convergence, (4) exploring hybrid approaches that combine SRA-style rule-based decision-making with MBRA-style memory systems.

References

[1] Nil Ansó, Anton Wiehe, Madalina Drugan, and Marco Wiering. 2019. Deep Reinforcement Learning for Pellet Eating in Agar.io. In *Proceedings of the 11th International Conference on Agents and Artificial Intelligence*, Vol. 2, ICAART. SciTePress, 123–133. doi:10.5220/0007360901230133 The 11th International Conference on Agents and Artificial Intelligence ; Conference date: 19-02-2019 Through 21-02-2019.

[2] Dario Floreano, Peter Dürr, and Claudio Mattiussi. 2008. Neuroevolution: from architectures to learning. *Evolutionary Intelligence* 1, 1 (2008), 47–62. https://link.springer.com/article/10.1007/s12065-007-0002-4
Summary by Joy: Provides a comprehensive overview of neuroevolution methods that use genetic algorithms to train neural networks without backpropagation. The paper details how evolutionary algorithms can optimize both network weights and architectures, avoiding common gradient-based issues like vanishing gradients. This approach validates our project's methodology of using GA to evolve RNN parameters for Agar.io agents, providing a solid foundation for our gradient-free training strategy.

[3] Srivathsa Gottipati and Praveen Paruchuri. 2021. A genetic algorithm approach to compute mixed strategy solutions for general Stackelberg games. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Lille, France) *(GECCO '21)*. Association for Computing Machinery, New York, NY, USA, 223–224. doi:10.1145/3449726.3459419

[4] Chia-Feng Juang. 2004. A hybrid of genetic algorithm and particle swarm optimization for recurrent network design. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 34, 2 (2004), 997–1006. doi:10.1109/TSMCB.2003.818557

[5] B. R. Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. 2022. Deep Reinforcement Learning for Autonomous Driving: A Survey. *IEEE Transactions on Intelligent Transportation Systems* 23, 6 (2022), 4909–4926.
Summary by Joy: This comprehensive survey details the challenges in autonomous driving, including processing real-time visual data, dealing with noisy sensor inputs, and making safe navigation decisions. It highlights the relevance of AI agents that can interpret dynamic environments, directly supporting our claim about the broader implications of our Agar.io AI research for real-world applications like vehicle automation.

[6] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. DARTS: Differentiable Architecture Search. In *International Conference on Learning Representations*. ICLR, 1–13. doi:10.48550/arXiv.1806.09055 The 7th International Conference on Learning Representations; Conference date: 06-05-2019 Through 09-05-2019.

[7] Siming Liu, Sushil J. Louis, and Christopher A. Ballinger. 2016. Evolving Effective Microbehaviors in Real-Time Strategy Games. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 4 (2016), 351–362. doi:10.1109/TCIAIG.2016.2544844

[8] So Nakashima and Tetsuya J. Kobayashi. 2025. Unifying Zeroth-Order Optimization and Genetic Algorithms for Reinforcement Learning. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (NH Malaga Hotel, Malaga, Spain) *(GECCO '25 Companion)*. Association for Computing Machinery, New York, NY, USA, 311–314. doi:10.1145/3712255.3726617

[9] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the Difficulty of Training Recurrent Neural Networks. In *Proceedings of the 30th International Conference on Machine Learning (ICML 2013)*, Vol. 28. JMLR Workshop and Conference Proceedings, 1310–1318. https://proceedings.mlr.press/v28/pascanu13.pdf?
Summary by Lucas: Describes the challenges associated with training recurrent neural networks. The paper first establishes a mathematical representation of a recurrent neural network and shows how backpropagation takes the gradient of each state x_t at time step t. It then explains how the exploding gradients problem refers to the large increase in the norm of the gradient during training. The vanishing gradients problem occurs when long-term components go exponentially fast to norm 0, making it impossible for the model to learn correlation between temporally distant events. Related to our project since we use a recurrent neural network as our base model for the genetic algorithm, and as a result, must address potential training complications.

[10] Elena Popovici, Anthony Bucci, R. Paul Wiegand, and Edwin D. de Jong. 2012. Multi-objective Optimization in Game AI. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. ACM, 789–796. doi:10.1145/2330163.2330264 The 14th annual conference on Genetic and evolutionary computation ; Conference date: 07-07-2012 Through 11-07-2012.

[11] Justin K. Pugh, Lisa B. Soros, and Kenneth O. Stanley. 2016. Quality Diversity: A New Frontier for Evolutionary Computation. *Frontiers in Robotics and AI* 3 (2016), 40. doi:10.3389/frobt.2016.00040

[12] M.L. Raymer, W.F. Punch, E.D. Goodman, L.A. Kuhn, and A.K. Jain. 2000. Dimensionality reduction using genetic algorithms. *IEEE Transactions on Evolutionary Computation* 4, 2 (2000), 164–171. doi:10.1109/4235.850656

[13] Arash Rikhtegar, Mohammad Pooyan, and Mohammad Taghi Manzuri-Shalmani. 2016. Genetic algorithm-optimised structure of convolutional neural network for face recognition applications. *IET Computer Vision* 10, 6 (2016), 559–566. doi:10.1049/iet-cvi.2015.0037

[14] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. 2017. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning*. PMLR, 1–15. doi:10.48550/arXiv.1703.03864 The 34th International Conference on Machine Learning; Conference date: 06-08-2017 Through 11-08-2017.

[15] Yudi Widhiyasana, Maisevli Harika, Fahmi Hakim, Fitri Diani, Kokoy Komariah, and Diena Ramdania. 2022. Genetic Algorithm for Artificial Neural Networks in Real-Time Strategy Games. *JOIV : International Journal on Informatics Visualization* 6 (06 2022), 298. doi:10.30630/joiv.6.2.832

[16] Kaichao You, Mingsheng Long, Jianmin Wang, and Michael I. Jordan. 2019. How Does Learning Rate Decay Help Modern Neural Networks? *arXiv preprint arXiv:1908.01878* (Sept. 2019). https://arxiv.org/pdf/1908.01878 Version 2.
Summary by Lucas: Studies the effect of learning rate decay on deep neural network training. Shows that large learning rates prevent overfitting or memorization of noisy data and can prevent models from being stuck in local minima. However, the paper also demonstrates how large learning rates cause the training to bounce between minima and never converge. By starting with an initially large learning rate and decaying the learning rate over time, the paper achieves models that don't overfit and manage to converge to a lower minima. This learning rate decay strategy is employed in our project to decay our mutation standard deviations.