

GANNs for Strategic Decision-Making: A Neuroevolutionary Framework

Cos 498: Introduction to Game AI

Sophie Walden

April 6th, 2025

Introduction:

Four and a half years ago before I came to UMaine I first created a self-built neural network approach to problem solving: GANNs. GANNs, or Genetic Algorithm Neural Networks, are a mass amount of multi-layer perceptrons slowly learned solutions to problems through supplied inputs and fitness functions. This framework was able to solve basic tasks: dodging objects, playing board games, or driving in a 2d world on unseen tracks.

In the past 4 years since I created these models I have taken (including this class) 4 Artificial Intelligence classes spanning topics from machine learning to planning. One type of project that my models particularly struggled with was multi-agent and more unpredictable simulations. This project is an opportunity to go back to my old models, improve them, and see if they can overcome the same challenges that I faced four years ago.

Find a link to my original models (particularly in model.py) here:

<https://github.com/SophieWalden/selfDrivingCarGeneticAlgo>

Engine Upgrades:

While the main topic of this project is the AI created, about half of the project was dedicated to engine improvements to create the best environment to visually watch the networks improve

Visual Upgrades:

The first and most visually appealing update is the change from a top-down board to an isometric view of the game board. To do this I drew sprites for all the tiles and units isometrically and rendered them at offsets to visually appear as a 3D game map. On top of this I added a ton of functionality to the display, showing up in isometricDisplay.py, that includes:

- Zoom and panning. You can drag the mouse to move the map around and zoom in using the mouse wheel to see small-scale battles.
- Window resizing. This is extremely difficult in pygame as even if you let the window resize your content will not resize accordingly. To fix this UI is rendered relative to window size and zooming / panning allows the map to be explored regardless of size. Try to use full screen to view the simulation at its best.
- Multiple UI elements to easily understand game state

- Turn counter in the top left
- Bar spanning the top of the screen denoting percentage of cities captured by each faction in their respective colors
- Mode selector dropdown. Select between evolution, versus, and endless game modes which will be discussed later
- Inspection panel:
 - By default shows all alive faction information including age, unit count, city count, and all collected materials
 - You can click on any unit to see a brief amount of information about stats, moves, and rank.
- Animation support, used to render:
 - Death animation, indicating when a unit dies on a tile
 - Structure building animations
- Tile outlines
 - A full tile outline indicates a city is owned on that tile
 - A smaller partial outline indicates who a structure is owned by

Game Upgrades:

Slight additions were added to create varied game positions and complexity in decision making, including:

- New terrain types: stone and water
 - Stones are defensive tiles which can be upgraded to mine stone material
 - Water is impassable and requires agents to plan and path around
- Structure types: Woodcutters and Miners
 - Can be placed on Forests and Stone tiles respectively
 - Passively mines resources which can be used to place buffs on created units
 - Can be taken over by pathing onto that cell the same as cities
- Defecting
 - A new mechanic mostly shown in longer games (Particularly in endless mode)
 - Generals (talked about later in army structure) have aptitudes for how much death they can see in their faction due to their commanders' decisions before defecting, becoming the commander of their own army
 - In endless mode when all factions become one the generals auto-defects to continue on the chaotic nature of endless evolution.
- Command shuffling updated to preserve command order within factions

Optimizations:

A major problem I faced when tackling neuroevolution in the past is that it is truly only as strong as how fast and how many models you can get learning.

- Rewrote a ton of the game functionality to reduce time complexity
- Cached:
 - o All rendered game tiles and units, using a LRU Cache to not needlessly fill up memories of renders at different zoom levels
 - o All rendered text
- Pathfinding for the units implemented using flow fields
 - o Flow fields are computed for a map when a general wants to path-find their soldiers to a specific point
 - o Flow fields are cached for performance
 - o A flow field queue is created which includes first all cities and then all possible terrain for structures. These are computed when time is left over before a clock tick
 - o With caching, any unit on the map can lookup how to pathfind to any city in $O(1)$ time
- Introduced multiple “logic frames” of turns in between draw calls for faster performance
- Added an option to run without visual displays, reaching up to 695 turns per second while running the GANN models



Figure 1: Final Iteration of Visual Display of the Game

Base AI System:

Before tackling the GANNs I had to create a base agent that could both be used to test functionality and as a benchmark to see agent improvement. During spring break when this assignment got released, I was in the middle of reading part of Orson Scott Card's Ender's Game series which has significant influence on this model.

Armies are made up of three different ranks of units: soldiers, generals, and commanders. There can only be one commander per army, but there can be an unlimited number of generals or soldiers.

They each have different responsibilities and capabilities:

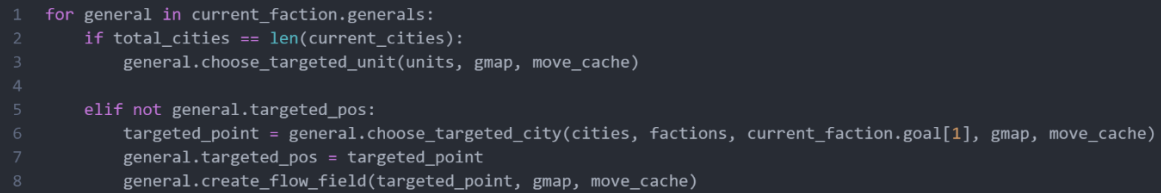
- Commander based on randomized unit aptitudes pick goals for the entire faction, such as "conquer closest city" or "gather wood"
- Generals (or toon leaders) then choose the appropriate target point to path towards based on both the commander's goals and the generals' positions.
- Soldiers have no active decision making, but follow the flow field creating by their assigned generals targeted point

All of this is condensed into the AI class which is used in all agents as a base. System classes are further defined to define both when/where generals target and when units/structures should be built.

At this point enough of a system is in place where the simplest benchmark system can be created: the aggression system. I tested with a ton of different strategies, the other one saved being a more balanced system, but found that the system that kept winning was the aggression system.

The aggression system truly does not care about what its commander's goals are. All it wants is for its generals to repeatedly path towards and take over the closest cities. This is hard to defend against since it gives no time to focus on a resource-based strategy and instead promotes needing a mix of defense and offense to counter.

This aggressive agent had a win rate of 90% over 50 games against the balanced agent mentioned previously. It is the final test for whether neuroevolution has properly been implemented in a way to beat an agent that I had not been able to create a system that gets a positive win rate against.



```

1 for general in current_faction.generals:
2     if total_cities == len(current_cities):
3         general.choose_targeted_unit(units, gmap, move_cache)
4
5     elif not general.targeted_pos:
6         targeted_point = general.choose_targeted_city(cities, factions, current_faction.goal[1], gmap, move_cache)
7         general.targeted_pos = targeted_point
8         general.create_flow_field(targeted_point, gmap, move_cache)

```

Figure 2: Aggression system's simple targeting system

Base Neural Network Models:

Due to being built completely from scratch the neural networks in my projects were never insanely complex. Early versions that tackled problems such as car driving never even had important concepts such as activation functions or node biases. A new version, created as the class Model in my neuralNetworks.py file, is an optimized version with a couple core capabilities that make genetic algorithms possible:

- Basic forward feeding capabilities. Given a list of inputs, it multiplies all layers of its weights and adds the biases to come to a result conclusion
- Crossover for generation of new models from multiple parents. This works by taking specific layers of weights and biases randomly from each parent to make a new model.
- Mutation, usually applied after crossover, which slightly tweaks the values to try to produce a model superior to either of its parents

Even with the basic functions being kept the same between my project years ago and this one now the major difference is the speedup I got through leveraging NumPy. For a year or two after I stopped experimenting with GANNs I would go back and try to rewrite it to tackle one key problem I found with it: it is incredibly reliant on getting as many training epochs as possible to reach valuable training conclusions.

I had a couple ways I was using NumPy when I implemented the model years ago, but those came from trying to make optimizations after I made the model. This time everything from the ground up was written using NumPy, from the handling of weights and biases to activation functions. This also made it incredibly easy to save as small as 10 KB pretrained models in .npz files, a way to store multiple arrays (weights and biases)

compressed. These optimizations caused benchmarked leaps by a factor of ~400 ahead of my previous models which made masses of them be able to be run without any problems.

Another key difference between my old implementation and my new one that simplified the model is the removal of the ability to create/remove new connections and nodes in mutation. This allowed my old models to effectively optimize down the size of the model and only use the necessary connections, but added complexity in terms of how much training would be needed to get to said optimized network. This AI was reduced down to as bare bones as possible to reduce training time even if it had a lower optimization ceiling.

Training:

For training I decided to use the networks to decide where a general should target. I gave it general inputs on game state as are listed later in this section and let it return general decisions such as “move to nearest ally city” or “go to furthest enemy city”. Furthermore, I removed the commander and allocated one general per faction, essentially allowing every faction loss and gains to be reflective of the general’s decision making.

The environment for which this model was trained on varied as I tried to nudge it towards its goal of beating the aggression system. The first one, which models how the network has been trained in other problems, can be explored in the endless game mode (which can be selected in the dropdown in the top right).

This mode created an endless sandbox for the models to repeatedly fight in. At first, you could see many models with completely randomized weights and biases standing still deciding to defend current cities while a few might venture out to get resources. A couple measures were put in place to encourage moving around. First, a 30-turn rule was implemented to kill any unit who stood still for too long and a 300-turn age limit for units in general to counter wiggling, a tactic which will be talked about later in the “notable observations” section.

Eventually in the endless mode a neural network will decide to take the conquerable approach and start to throw its units at other factions. A fitness function was created that scores the model based on metrics such as the cities gained, or soldiers lost. When the network starts to have a positive soldier kill or city capture ratio they jump to the top of the model leaderboard. All models that are created afterwards are a mutated baby of all-time top 4 models. If a faction can conquer all of the factions, then the generals will automatically defect and continue on the learning process.

Another metric added in the fitness function to encourage the agents to act is a negative punishment based on the general's age. If a general spends too long waiting without a result, they are scored way lower than one that can rush in and create change. This is to once again speed up the training process and encourage fighting to learn, but in the end makes chaotic agents which will be discussed below

The neural networks are passed 11 different inputs to make these decisions (All normalized to values between 0-1):

- Average soldier health
- Number of soldiers controlling
- Relative number of soldiers controlling compared to all units on map
- Relative number of cities controlling compared to all cities on map
- Distance to nearest ally city
- Distance to nearest enemy city
- Whether the network can build a structure this turn
- 4 Inputs based on number of enemies around a certain point in a 4 tile radius:
 - o Enemies surrounding general
 - o Enemies surrounding closest ally city
 - o Enemies surrounding closest enemy city
 - o Enemies surrounding furthest enemy city

Notable Observations:

A couple days were spent just tweaking the fitness function, adding in new metrics, and trying to encourage the agents in any way to become "smart" about their decisions. I keep a log of all my notes for design and have a couple interesting ones I decided to write about:

Not completely sure how they reliably recreated this since they had very limited points they could path towards, but at some point, they became aware of the 30-turn rule. They realized that if they waited long to still, they would have no chance for score improvement and invented "wiggling" to prevent it. The units would move back and forth between two tiles and when two armies did it in sync it could cause battles to stall for up to hundreds of thousands of turns. To counter this I implemented the turn and age limits, but it is still a tool used by a ton of the successful AIs to stall while building an army.

At first there was no penalty to losing cities since I wanted to encourage exploration and chaos, but this also got them stuck in weird ways. The most common thing was to see two generals who were able to wiggle back and forth in sync on the same city tile to make sure they were re-conquered in rapid succession. This cross-faction allying to generate more

score resulted in AIs topping the leaderboard with thousands of cities captured when they in fact are usually worse at capturing cities in aggressive one on one games.

To encourage a ton of different models at the start I changed faction count to high amounts to ensure a couple models would start learning quickly. This immediately created defensive models as the agents that lived the longest were the ones creating a defensive force and patrolling their agent cities. I found this interesting since it came entirely without a change to the fitness function or how new models were made. It was completely off the back of what models performed better in chaotic situations and was one of the few times I saw them default to defense tactics.

Endless Results:

The results from the endless were unsurprising. After doing multiple runs of training for 10,000,000 turns each I took the models saved and did tests against the aggression AI. After a large amount of test games, the top models ended up winning ~45-50% of the games against the aggression AI. Not quite an overwhelming winning force, but much better than the balanced systems attempt before. In looking at the networks and what they were predicting though I found that most of them were no better than the aggression AI opting to go for closest city aggression 90%+ of the time. This was shown furthermore by the top agents just being those who fought in most battles, not the ones who lasted the longest in said battle.

Another problem that stood out plaguing long training times is that it was hard to rate a general's performance just by looking at soldiers or cities lost / gained. This was because all these metrics are relative to how well the generals around you performed. To be the first general to figure out that conquering is good and taking all the cities, you could skyrocket if you are against agents who did not figure that out. Further than that your spot on the leaderboard might never drop if there is not a similar level of improvement that makes an AI generationally better than the others. This was a problem that I did not end up solving which made me look towards other means to solve my goals.

What I decided to do I had not done for any of my GANN projects prior: I decided to throw away the fitness function. The metric for which I've been testing the models on is whether they can beat the aggression AI, so why not directly pit them in battles and only promote the models that consistently win. That is where the main mode of the simulator was created: evolution.

Evolution:

In this mode there is a distinct lack of optimizations available due to switching maps so often, but one key upside which is the exploration rate of new models. Agents in endless mode were incentivized to optimize the fitness function which resulted in agents finding weird solutions, stalling the game, and creating long battles to up their score.

Evolution is the default mode that boots up when launching the simulator. To make it easy to follow along, the evolutionary AI model is always red and the aggression AI is always blue.

There are two major phases two evolution: exploration and refinement. In the exploration phase a completely new model is created every single game. This could be an awful model that prioritizes building log cabins over everything else or a crushing general that is able to take a game off their aggression system. In fact, that is the goal of this phase, to find a singular model that can take a game off the aggression system.

Once it finds this singular model it goes into a refinement phase. It starts to create models that are children of it, with mutated tweaks to see if they have better performance. If they can win as well they are added to the leaderboard tracking them. From the leaderboard their rankings are adjusted based on whenever a direct child of that model wins. At the start this means that the first agent chosen will get a ton of wins since most of the first-generation models will be based off of it, but at a certain point better agents show up and have higher success rates from their children as the system continues evolving

Evolution Results:

When training on evolution mode the first phase can take anywhere from 10 seconds to 10 minutes. All it matters is trying a ton of different approaches, so I cannot guarantee immediate results for anyone all you have to do is wait.

When it does win though you start to see wins after that trickle in steadily. Sometimes the mutation creates agents that get handily beat, but no longer do you go to tens or hundreds of games without winning anymore. In fact, when you start to win you see a noticeable shift in how the games are played out. Now there are longer matches, there is more back and forth, and a surprising number of matches the evolutionary AI is winning

After a threshold of the children of a model have won, I save them to later be evaluated. In a training course for 5 hours over 400 distinct models were saved. To evaluate them I wrote a simple function that tested them on an ever-increasing sample size until the models fell below the desired win rate and ranked them according to that. From that I took the best ranked model to include in my submission, under the name pretrained model

which you can watch it fight in the “versus” mode of the simulator. Ranking them all and evaluating the top model’s win rate over 1000 games it scored 635 wins against the aggression system.

Conclusion

The model, while not being perfect, hits my goal I’ve set for this entire project. It can handily beat the aggression model that so many of the systems have lost to and it does so almost two thirds of the time. Furthermore, this is just one session of long-form training in the evolution simulator, with even more there may be a stronger model produced that can completely shut down the aggressive system’s strategy.

A big part of the extra credit is to list and explain all my additions, and while I have already listed them above here’s a short system detailing why each category:

Visual Upgrades: tremendously helped in quickly finding bugs and the amount of information reported on screen made what systems were prioritizing, pathing towards, and understanding their thought process way easier.

Game Upgrades: Structures added more game depth potentially adding another way to win through defense and research as opposed to violence. The best models in evolution never went that route expressly, but in multi faction chaotic situations they pop up in the endless game mode

Optimizations: As talked about in multiple places in this write up the model is gated by how fast it can process and running the game as fast as possible was key to seeing quick changes in the evolution process

Systems: While not explicitly changing the game, the base model of army delegation calls back to previously explored topics of information webs I did in the soccer based hw2. Further then that I was able to explore previously tackled topics, GANNs, and overcome issues in an environment I have not been able to before.

I am not sure what matches the definition of “complete overhaul/reimaging of the game turning it into a nightmare of complexity and awesomeness”. I am hoping however that the significant amount of time I have spent redesigning as many aspects of this simulator as possible into what turned out to be my favorite project I have worked on in college certainly qualifies.