CS 378: Computational Intelligence in Game Design

Evolving Boss Behavior

Ali Rashed
Jason Palacios
Scott Munro
Josh Montgomery

Introduction

Game design tends to be an involved process. Creating fun, interesting strategies for enemy agents, that are also balanced, so they aren't too challenging or too easy, can be a daunting and time consuming task. Our goal with this project was to evolve a boss agent in a basic game environment. Ideally, once the basic architecture for evolving a boss's behavior is implemented, a developer would be able to evolve multiple strategies for an opposing boss with relative ease, without having to design specific behaviors themselves. Evolving components of a game separately from the final game could make designing and implementing a game easier, and lead to more interesting behaviors that a developer might not have thought of, or would have had difficulty coding. At the very least, this method of generating boss-behavior can be used to explore specific ways a boss can act to counter certain habits of its enemies.

With this project, we explored basic behaviors and actions, so there is room for further research in this area, such as experimenting with complex actions and its effect on the evolution of the boss agent. It is also worth noting that there is room for exploration and experimentation in regards to evolving other aspects of a game separately.

Approach

Genetic Algorithm

We considered multiple approaches to solving this problem. We decided that pitting two bosses against each other wasn't a smart idea, so we opted against a co-evolutionary approach. We considered using NEAT or a similar algorithm to evolve the structure of the neural network, but that idea proved infeasible to implement with the time that we were given. We settled on using a genetic algorithm. Breeding the top quarter of performing boss agents together in order to produce the next population seemed like it'd have the most success, and would scale well with more complex bosses and game mechanics. At the end of evolution, a human player was to go up against the evolved boss.

Game Environment

We created a basic game environment in pygame to serve as the base for our evolution and experimentation. The "player" was near the maximum of the playing field's y-axis, and the boss was near the minimum y-axis. Both agents could only move along the x-axis, and could fire up to a maximum of 4 bullets on the screen at any one time. The boss agent was twice as big as the player.

From technical view, many game entities were object-oriented. The boss and bullets had their own class, while the different NPC AI were the subject of inheritance from a NPC base class with abstract methods. Another class controlled the NPCs and would also serve as a hook for human input. Classes were made to represent a single neuron and a neural network. The neuron class interfaced with the neural network while the neural network interfaced with the boss class which was very similar to the class that controlled NPCs.

Fitness

Our first challenge, after creating the game environment, was determining the fitness of the boss character. We experimented a lot with various methods of determining fitness and testing the boss's overall capabilities, in order to best optimize the evolutionary process. Using a naive fitness, such as whether or not the boss won, or the NPC's final health, didn't lead to any interesting behaviors and the final agents ended up only learning how to do simple things.

In an attempt to produce priority-based strategies and/or teach the boss important concepts such as dodging and tracking, shaping of fitness was involved. Basically, there were multiple scenarios where fitness would be decided differently although all of them would produce similar scoring.

Types of Fitness:

- If the game played for the max number of frames, fitness would designed so that it would punish the boss for taking so long and not killing the NPC while keeping in mind how well it had done against the strategy by measuring the boss' health. The logic defining this can be seen below:
    - `(-1 * MAX_FRAMES) + (npc.health) * -20 + (boss.health) * 20`
- If the boss had managed to kill the NPC, fitness would be decided by how fast it completed the task and how well it had done against the strategy by, again, measuring the boss' health. The logic defining this can be seen below:
    - `(MAX_FRAMES - frame_counter) + boss.health * 20`
- If the boss had died against the NPC, it was to be punished for how fast it had died while keeping in mind how well it had done against the NPC by measuring the amount of damage the boss gave to the NPC. The logic defining this can be seen below:
    - `-1 * (MAX_FRAMES - frame_counter) + (100 - npc.health) * 20`

NPCs

We hard-coded a set of NPC AI to represent different strategies a player might use. These agents were stand-ins for the player agent. Interactive evolution would've been prohibitively long and arduous, and while it might have led to a boss that was well-evolved for a certain player, user fatigue and the actual evolution time to determine fitness would have taken very long. Our solution was to code a set of behaviors to 'represent' a wide range of 'typical' player behaviors and actions. We could then cycle through the behaviors and have the boss learn how to defeat the different strategies, in hopes that, at the end, it'd evolve well enough to handle a human player. We settled upon the following strategies.

No.	**Strategy Type**
1.	Do not move and do not shoot
2.	Shoot while standing still in the center
3.	Move all the way to the right and shoot
4.	Move all the way to the left and shoot
5.	Move back and forth from wall to wall and shoot

6.	Move back and forth randomly and shoot
7.	Move back and forth randomly and do not shoot
8.	Retreat away from boss and shoot when in range
9.	Approach the boss and shoot when in range

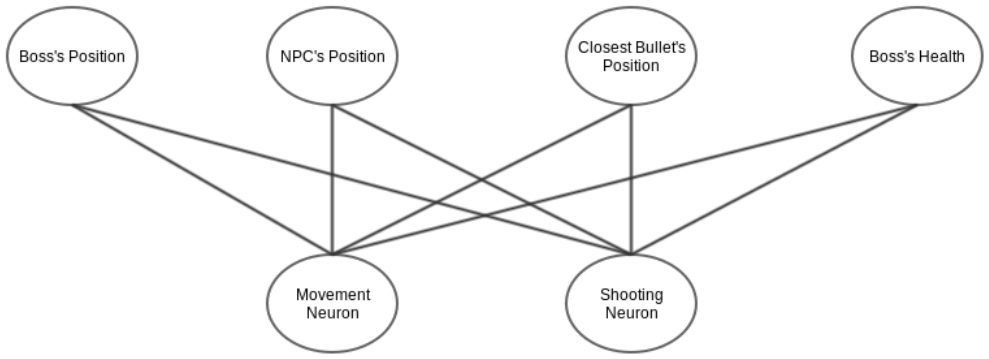
NPC Cycling

Another challenge was deciding how to cycle through the strategies. Playing every individual in the population, against every strategy, for every generation, was prohibitively long. We experimented with two different methods of cycling: random and sequential selection. The former involved randomly choosing a strategy out of the ‘pool’ of strategies and running the boss against it for 20 generations. The latter sequentially progressed through the strategies in the above order. Although the resulting boss of random selection was typically not as well-rounded as one might hope, unique, priority-based strategies tended to appear (i.e. health vs shooting). With sequential selection, the boss could learn how to best react and adapt to basic behaviors before being thrown against harder and more complicated strategies -- this usually produced well-rounded bosses.

Networks

We worked with two different networks. Since we weren’t using NEAT to evolve the structure of the artificial neural network (ANN), we had to hand-design and implement networks that we thought would have some level of success. Therefore, it is important to note that there are other networks that could possibly have better performance, or reach an optimal boss sooner than ours could. In the end, we designed two different networks: the “simple” network and the “S-RA-LR” network.

The Simple Network

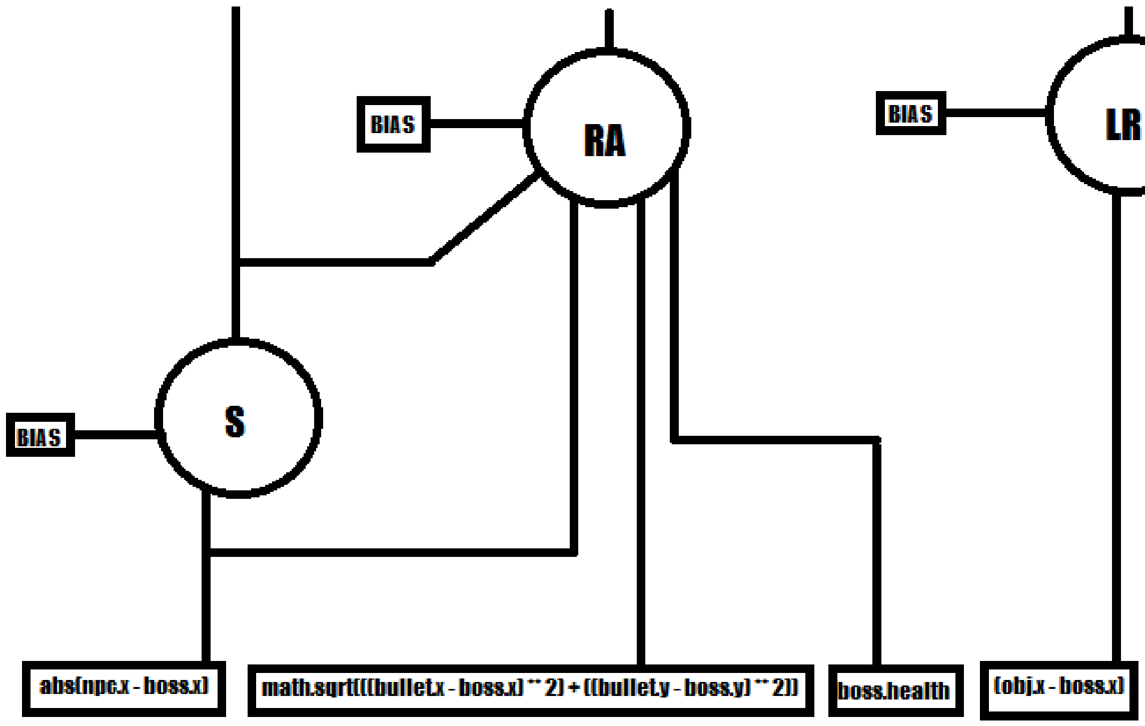


Because the game environment was relatively simple, a simple network consisting of 4 input nodes that fed directly into the two output nodes was implemented. At any moment during the game the Boss agent only had to keep track of the NPC agent and the closest incoming bullet, so it would be unnecessary for the genetic algorithm to have to evolve a multitude of weights for hidden nodes that did not actually serve to produce better agents. Initially we had tried to develop a network that did include hidden layers and that network performed less effectively than the simple network and usually failed to evolve anything significant at all.

The network's input nodes were decided based off of what kind of information a Boss character generally has in a videogame. It has access to its own position so that it can navigate its environment intelligently (i.e. not running into walls). It can see where the player would be positioned at any given moment (the NPC's position) so that it can make decisions about where to move in relation to that player. It has the closest bullet's position so that it can react to oncoming attacks. Finally it has access to its own health so that it can implement different strategies based on how much health it has; a boss with more health could implement a more aggressive strategy and a boss with less health could implement a more defensive strategy.

After a handful of trial runs, the network wasn't yielding any results that were more significant than what our original failing network had developed. It's simple structure consistently led it to the local optimum of a hugging one of the walls and firing as much as possible because doing so kept it out of the line of fire. To diversify this network's evolution, the genetic algorithm was restructured so that it kept track of populations that had reached local optimum, and if after 10 generations the network was not showing any improvement then the population was replaced with a fresh population of random children. The algorithm was also edited to keep track of these optima in a separate list so that at the end of evolution all of these optima could be compared to one another. The end result was surprising. By resetting the population whenever the network had reached a local optima, the network tended to find much more successful and diverse results.

<u>The S-RA-LR Network</u>


BIAS
RA
BIAS
LR
BIAS
S
abs(npc.x - boss.x)
math.sqrt(((bullet.x - boss.x) ** 2) + ((bullet.y - boss.y) ** 2))
boss.health
(obj.x - boss.x)

The "S-RA-LR" network was designed to be simple while still having the potential to evolve interesting strategies. Just to clarify, the "S" stands for "shoot", the "RA" stands for "retreat-approach", and the "LR" stands for "left-right". When combined, each term describes the basic functionality that a boss should possess; the boss should be able to shoot, the boss should know whether to retreat or approach the closest object, and the boss the should be aware of whether the closest objects lies on its left or right side.

As one might have guessed, each term also represents a single neuron in a network and they also represent the information that the network outputs.

The inputs of the “S” neuron were decided such that the network might be able to tell whether or not the NPC was in range to shoot. By usage of the BIAS and appropriately selected value for the weight of absolute value of the relative x-coordinate of the NPC in relation to the boss, the network might be able to manipulate the range checks that the AI would use. If correct weights were selected, this might produce the “scatter shot” strategy described later in the results section.

The inputs of the “RA” neuron were decided such that the network would know whether or not to retreat or approach the closest object. Manipulating the weight of the bias would provide the beginnings of evolving a priority based on either retreating or approaching. Feeding in the boss’ health and the output of the “S” neuron could potentially lead to priorities based on “health vs NPC killing”. Lastly, providing the basic inputs of distances between closest bullet and the absolute value of the relative x-coordinate of the NPC in relation to the boss would eventually help to distinguish a relationship of knowing to retreat from bullets and to approach the NPC. If set up correctly along with the “LR” neuron, this would eventually lead to the “weaving” strategy defined later in the results section.

The inputs of the “LR” neuron were decided such that the network would know which side the closest object lies on in relation to the boss. The closest object was decided by comparing the distance of the closest bullet to the boss and the absolute value of the relative x-coordinate of the NPC in relation to the boss. When used correctly with the “RA” neuron, this could produce basic relationships of knowing what to direction to move in.

Results

Initially, our program produced very bland results in boss-behavior and evolution. Upon close examination of our code, we recognized very crucial bits of code that were incorrect. Upon fixing the bugs within our code, the program began to produce more fruitful outcomes -- our results then starting becoming quite interesting. We managed to evolve multiple strategies successfully, and there was enough variation for the resulting evolved agents to be ‘different’ enough to be distinguishable. As we optimized our fitness function, NPC cycling, and network, we began to see more complex and challenging behaviors.

Boss Behaviors

We successfully evolved a range of strategies that were different enough to be significant. One of the behaviors that emerged, albeit rarely, was a ‘weaving’ strategy. The boss would learn to follow the player, but move out of the way of bullets. This was a human-like strategy, in which the boss would prioritize avoiding damage, followed by pursuing it’s opponent. Bosses that learned to weave were usually the most interesting to play against.

One of the other strategies that emerged was hugging the wall. The boss learned to stick to one side of the game environment, and only moved when the player came within its range, or to move out of the way of a bullet. These bosses were successful against our training NPC’s but were not as fun to play against.

We also evolved strategies where the boss followed the player, shooting as fast as it could, and neglected the negative effects of being shot. These agents made shooting their opponent the top priority, doing more damage in the short-term, but sacrificing their survivability.

Another rare occurrence was bullet management. We limited the boss to 4 bullets on screen at a time, and the most common strategies had the boss firing all 4 bullets at once, whenever it could. However, every so often, an agent would evolve to learn how to better manage its bullets, shooting in bursts in order to better hit its opponent.

It is important to note that the way we cycled through the NPC strategies had an effect on the evolution of boss behaviors. Random-selection of NPC's led to bizarre priority-based strategies and "hugging", while sequential cycling was more prone to producing bosses that could weave and manage their bullets. It is also of note that all the agents except those that learned to "weave" and/or produce "scatter shot" performed poorly against NPC strategies that involved randomization.

Frequency

Strategy	**Development**
Weaving	Initially could emerge as early as 30 generations, but became precise around 50 generations.
Scatter Shot	Initially could emerge as early as 50 generations, but usually was seen around 110 generations.
Hugging	Would always appear after going against an NPC that fires bullets (i.e. 30 generations in, when using sequential cycling)

Many of the agents were not able to beat every NPC strategy that we had created. This, however, was ideal for bosses that were being evolved to ultimately play against humans. Humans are inconsistent and have neither the reaction-time nor the accuracy of an NPC. By evolving agents that could outperform some, but not all, of the NPC strategies, we were able to produce agents that were challenging for human players, but not impossible to beat. This is important, because an impossibly hard boss would not fit well in an actual game.

Conclusion

We were successfully able to evolve agents that performed well as bosses in the game environment we created. While we did see positive results with this progress, the game systems that we used were bare. Our results show that there is promise in using this method to develop enemy AI in game development, but there is still more work to be done. The boss-behaviors evolved using our program are functional, but could be far more efficient in optimizing against patterns. Our program never generated a boss with excessively impressive ability.

In regards to evolving the agents, it is worth noting that the topology we used was fixed. The most challenging part of evolving suitable agents was finding a topology that led to interesting behaviors. An algorithm, like NEAT, might be better equipped at evolving the optimal topology, along with the correct weights. We also simplified our game environment as much as possible in order to create the basic framework for this project. Building upon the game (i.e. different weapons, alternate types of movement, power-ups, etc.) could lead to interesting results. One could evolve multiple agents, in the same game, that behave in the ways best suited to the arsenal they are given.

Ideally the best results might be seen during interactive evolution, and a boss could be tailored to be challenging against a wide range of players. Humans are random and imprecise, and that behavior is hard to replicate. Furthermore, our game environment was purposefully limiting, there was only a handful of actions a player could perform. As the player's freedom increases, the difficulty with hardcoding set NPC patterns increases as well. While we were able to create 'good' bosses, the 'perfect' boss might not be attainable using hardcoded NPC's, especially in a more complex game. One solution is to evolve a group of bosses, and a group of NPC's, and have them engage in coevolution whilst playing against each other. This might lead to boss agents that are better at handling unexpected behaviors. However, it should be noted that this would slow down evolution-time considerably.

Future work on this project itself would include the implementation of complex systems such as: rotational-axis shooting, differing weapons (e.g. shotgun spread bullet, diagonal bullets), shielding that is more effective against certain types of bullets, interactive evolution in which the user chooses how to train the boss by selecting different behaviors among the NPCs it battles against, and objects floating in the game-screen that may interfere with bullet path such as asteroids, fixed walls, or mirrors.