Adventure Game with a Neural Network Controlled Non-playing Character

Conference Paper · December 2017

DOI: 10.1109/ICMLA.2017.0-129

4 authors, including:

David Binnion
Georgia State University
2 PUBLICATIONS **6** CITATIONS
SEE PROFILE

Andre Kenneth Chase Randall
Carnegie Mellon University
3 PUBLICATIONS **6** CITATIONS
SEE PROFILE

Vibhuti Patel
Tata Institute of Social Sciences
181 PUBLICATIONS **274** CITATIONS
SEE PROFILE

2017 16th IEEE International Conference on Machine Learning and Applications

Adventure Game With A Neural Network Controlled Non-playing Character

Michael Weeks, David Binnion, Andre Chase Randall, and Vibhuti Patel
Department of Computer Science
Georgia State University
Atlanta, Georgia, USA
mweeks@ieee.org

***Abstract*—This paper presents a way to evolve a non-playing character by controlling it through a neural network, then using a genetic algorithm to alter the weights. Previous neural network data provides a population from which to select individuals, and the simulated results allow relative rankings that we use as a fitness function. Treating the selected sets of neural network weights as a binary sequence, the program uses cross-over and mutation to create a new set of weights, which are subsequently evaluated. Through this process, we find that the program creates better sets of neural network weights, and that the relative rankings allow the computer-controlled character to have a range of difficulty levels. We implement this experiment as part of a game.**

***Keywords*-Neural Networks; Genetic Algorithm; Web-based Games; Neuroevolution**

I. Introduction

This papers examines an experimental way to control a non-playing character (NPC) in a video game, without the involvement of a human opponent. Normally, the game allows a human-controlled character to move through a world, collect items, and fight NPCs. But how can the computer control the NPCs in a way that makes them complex, responsive, and somewhat unique? A neural network allows for this, where two instances of the same NPC could behave differently, according to their network weights and current inputs. While there are many ways to refine neural network weights, this paper explores the Genetic Algorithm to do this, an approach called neuroevolution.

To evolve the neural network weights, we implement a tournament where the computer moves two characters, an ogre controlled by a neural network, and a wizard controlled by a simple strategy. The neural network selects a behavior that moves the NPC, uses the shield, sets the direction, and varies the speed. Hereafter, we refer to both of these as characters, while we use the term NPC to refer specifically to the character being evolved. The computer controls the wizard character, too, for the purpose of the experiment, though it is a stand-in for a character normally controlled by a human player. Table I summarizes the possible moves, and Figure 1 shows a screen capture of the two characters fighting.

This game contains well-known attributes of adventure games, where the player must collect items, fight monsters, solve some puzzles, and locate a special item to win, similar to [1]. Under normal play, the user interactively navigates a character through different rooms, some with traps, or requiring keys. While navigating the course, the player can fight various monsters including an ogre with artificial intelligence. Ultimately, the player must locate the golden chalice and bring it back to the white castle to win the game.

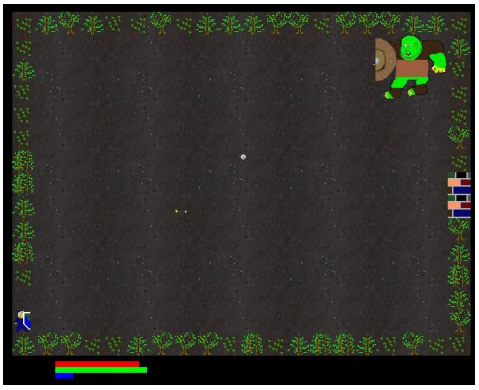

Figure 1. A screen capture of the two characters fighting. Normally, the human player controls the wizard, while the ogre (top right) is computer-controlled. In this experiment, both are computer controlled, with the ogre being controlled by a neural network.

No matter what action is taken, the program will generate data after each game is finished. This data records what actions the ogre performed, as well as the number of times the action repeated. An artificially intelligent NPC means that it evolves over time. In other words, it learns from its previous trials to become a more capable opponent. This could be used to have the NPC adapt as the human opponent progresses, though at the moment it evolves against an effective, simple strategy.

Neural networks are used to make the non-playing character, the ogre, more challenging to defeat. The ogre can learn from which techniques work the best, and use those techniques in the future to become a stronger opponent. In this case the ogre learns what works by analyzing its previous results, called the Neuroevolution approach.

The next section presents the background. Section III discusses the methodology and experimental set-up. Next, section IV presents the results, and section V concludes our paper.

0-7695-6321-X/17/31.00 ©2017 IEEE
DOI 10.1109/ICMLA.2017.0-129

Table I
POSSIBLE NPC MOVEMENTS IN THIS GAME, AND THEIR MEANINGS.

close	get closer at normal speed
charge	get closer at double speed
retreat	like a charge in the opposite direction
hold	use the shield, but hold the position
back-away	a slow retreat, with the shield. This is a step away, turning back to face the target, and using the shield
special attack	a slow but powerful strike. It takes several turns, but does double damage
check	use the shield, advancing into the opponent
ignore	move randomly

II. BACKGROUND AND RELATED WORKS

Role playing games involve a player that explores a world filled with other, non-playable characters to encounter. The NPCs can come in the form of less intelligent animals and humans such as shopkeepers and guards, or more intelligent ones such as those created to fight the player. The ultimate goal of NPC creation is to make a character that provides an engaging and entertaining experience, with believable behavior and to try to make it indistinguishable from a human player [2]. These non-player characters can be created using finite state machines. According to [3], in 2006, most non-playable game characters were made using a combination of finite state machines and scripting. These NPCs could be made with very little thought or code, when accounting for each individual character, but there would be a trade-off of increased rigidity [3]. With finite state machines, any situation that will be encountered must be accounted for, or the non-player character will not act according to how the player perceives it should act in the given situation. While this method of creation for non-player characters is acceptable, as computer games advance, players desire increased immersion in their games, accomplished in part by creating an adaptable non-player character. This can be done in a variety of ways, with the methods most similar to our approach being an altered finite state machine or a neural network. The use of the best model requires trial and error heuristic techniques. In fact, some have proposed an alternative solution to using a finite state machine to solve different general domain problems, such as [4].

The approach introduced by [3] is to use a system similar to a finite state machine, but with various artificial intelligence systems that are switched between as needed. This need is determined by a series of facts and rules that evaluate the situation based on the current environment and NPC state [3]. Our current system also evaluates the next action based on the current NPC state compared to the environment, but in this case the environment evaluation is limited to the state of the NPC's opponent.

In [5], the concept of using neural networks to create the non-player character is explored. With neural networks, there are sets of nodes organized into layers. Each node can be on or off, but the connections between that node and all the nodes in the next layer determine which nodes in the next layer are turned on. Neural networks are usually organized into two or three layers, with three layers usually being enough to solve most problems [6]. With neuro-evolution [5], a neural network is used, but instead of utilizing a backpropagation algorithm for training, an evolutionary algorithm is used. These, according to [7], are stochastic global search methods, able to search a large space very quickly. What they do is narrow down a set of possible measurements by taking the best of the group and using those to create the next set to test. This approach, combining neural networks and evolutionary algorithms, allows for generation of a neural network without the need for training against other users or a pre-determined difficulty rating. The problem is with the amount of time needed to create a new neural network for each NPC.

A similar approach to our work is used in [6]. In [6], a genetic algorithm is used to narrow the search space for the given data and the resulting data is passed to a neural network to evaluate errors and fitness. This result is then passed back to the genetic algorithm to create the next working set. This means [6] uses a neural network to finish off the genetic algorithm approach to optimizing the data, effectively using strengths of both approaches.

The paper by Ebrahimi and Akbarzadeh [7] uses a similar approach to [5], utilizing a neural network with evolution, but in this case, the data for the evolution step is taken using the interactive evolutionary computation method. This method uses human evaluation to replace the fitness function of the genetic algorithm. This causes conversion to the optimal solution within a smaller number of generations at the trade-off of user-fatigue [7]. It does provide a unique perspective, but this neuro-evolution set-up is used entirely for the movement-based game Pac-Man [7]. If adapted, there would need to be a large database to keep all the training data if the game is to adapt to the perceived abilities of the user, utilizing a weaker or stronger neural network as needed. Our current approach uses a method most similar to the one in [7]. An unsupervised genetic neural network is generated and tested against the opponent.

III. METHODOLOGY

The program initially populates a neural network with randomly assigned weights, then refines it with a genetic algorithm. The weights are 64-bit floating point values, however we do not use the full range. We restrict the values to a range of -1 to $+1$, and choose to limit the precision to $N = 10$ bits. Representing the weights this way, each one takes 10 bits, with an additional bit for the sign, for a total of 11 bits. Thus, each weight can be represented by a

binary string, or with three hexadecimal digits, though the first digit only ranges from 0 to 7.

We use 11 bits for each neural network weight. Let n be the number of different number possibilities for any given connection weight and s be the number of connection weights to compare for a given node. If we take n^s to be the number of different possibilities for one connection weight and take $(n! - (n - s)!)$ to be the number of different possible values for one connection weight, then using the function $n^s - (n! - (n - s)!)$, we can calculate the number of times one weight can have the same value as another. In this case, another equation is needed to compare the number of separate values created for all ending nodes. The equation would be $y^t - (y! - (y - t)!)$ where y is equal to $((n! - (n - s)!)$ and t is the number of ending nodes. The probability of having multiple copies of the same number in this case would be $\frac{y^t-(y!-(y-t)!)}{y^t}$.

A. How the neural network works in the game

The behaviors are enumerated as shown in Table I. Currently, we use seven inputs to map to the eight outputs, as shown in Figure 2. With seven weights per input feeding to seven nodes, we have 49 level-1 weights. We implemented the program with a ninth possible behavior, Choose, which re-evaluates the weights. Useful as a default state, it does not appear in Figure 2, since it should not be a possible decision outcome: it essentially means that the NPC would postpone the decision for another time step. The NPC uses it internally, as the way to select a new behavior once the previous behavior comes to an end. The seven level-1 nodes, with nine connections per input, mean we have 56 level-2 weights. Thus, a set of neural network data for one computer controlled NPC has 105 weights. With 11 bits per weight, we can represent the weight set as bit strings of 1155 zeros and ones. The program implements behaviors as a simple finite state machine, where the chosen outcomes repeat a set number of times as needed. For example, a charge should span several steps. Otherwise, the NPC could literally do 20 different things per second (i.e. with 50 ms between each step).

The tournament pits two characters against each other. Initially, we had the player-controlled character fighting a computer-controlled character. However, this is problematic with user fatigue and inconsistency between players. Instead, we have two computer-controlled characters fight. We define a *tournament* game as one fight between two characters. Both characters have a ranged weapon and a close-in weapon, and the program automatically switches between them according to the distance. Both characters have armor, and shields.

When the game loads, it processes the data from previous runs, ranks them, and retains data from the top ones. As shown in Figure 3, the data from previous runs includes the neural network weights, along with some statistics. To

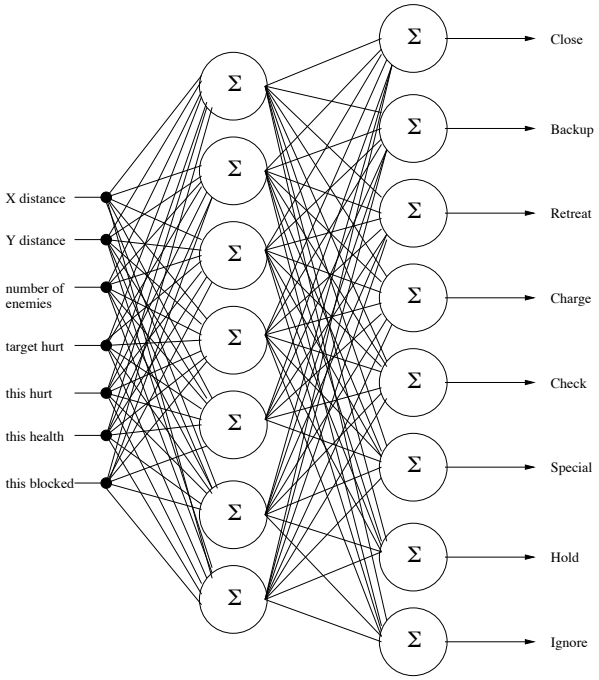

Figure 2. The neural network. To determine the ogre's next move, we use the inputs X distance, Y distance, number of enemies (currently limited to 1), whether the opposing character was recently hurt, whether this character was recently hurt, how much of this character's health remains, and whether this character recently blocked an attack. The outputs correspond to the possible character movements.

find the rank, which we use a fitnes function, the program divides the damage dealt by the damage taken. We define the damage dealt as the actual amount inflicted instead of the amount attempted, which could be reduced. For example, if a character is hit by a rock but has armor, the armor absorbs some of the damage that the rock would have done. If the character holds a shield between itself and the rock, the amount of damage may be nullified. Thus, the actual damage dealt (by the character throwing the rock) might be none. The actual damage taken comes down to the character's perspective: the actual damage dealt by a character is the actual damage taken by the opponent's character. Thus the rank for character i is defined as:

$$rank_i = floor\left(\frac{actual\ damage\ dealt \times 100}{actual\ damage\ taken + 1}\right). \qquad (1)$$

Scaling the fraction by 100 means that the *floor* function returns an integer value between 0 and the opponent's maximum health times 100. Positive, integer variable $actual\ \ damage\ \ taken$ can be zero, so the $+1$ in the denominator prevents a divide by zero problem. Thus, the higher the $rank_i$, the better. Armor does wear down over time, and the program keeps track of this. Also, each tournament game has a time limit, allowing the game to end in a tie. This prevents two defensive characters from having an infinitely long fight.

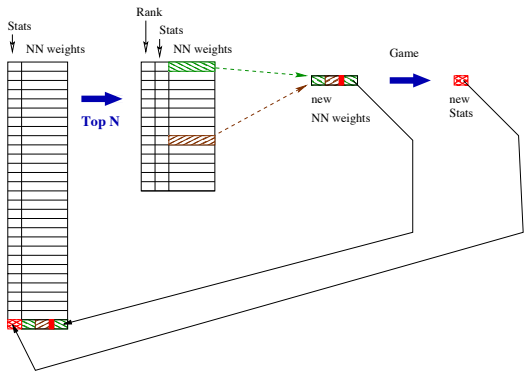

Figure 3. At the start, the program reads all previous sets of neural network weight, along with their statistics. It ranks them, and keeps the top ones. From this data, new weights are created, and statistics are generated.

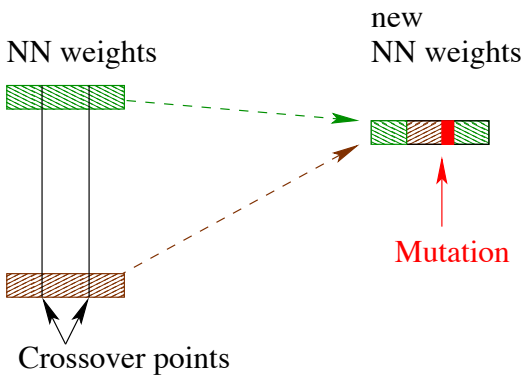

Figure 4. Two sets of weights are chosen, random crossover points are selected, random mutations are included, resulting in a new set of weights.

B. The Genetic Algorithm component

The program uses the genetic algorithm procedures of selection, crossover, and mutation. It forms a new neural network by taking the best data set from previous runs, selecting another top data set, and combining them together. The neural network weights for one set are converted to a long, binary strings, and the same is done for the other set. Figure 4 shows how two sets of weights are combined to form a new set. At the moment, instead of selecting two crossover points, the program selects one, and randomly determines which of the two data sets to use first. It copies the binary string from the first set up to the crossover point, then copies the string from the second set. After this, it randomly makes mutations, where it flips some of the bits in the string. Once complete, it unpacks the binary string into the neural network weights guide the character. If there is no previous data, the program instead randomly populates the weights.

Once a tournament game completes, the program sends the results to the server. Then it resets the statistics, creates a new set of neural network weights, and starts the fight over. This repeats a set number of times, until the tournament ends. When this occurs, the program will repeat the set of tournaments, with the difference that it first re-ranks the statistics of all previous runs. If one or more of the new sets of data score high enough, they will displace some of the old sets. This comes from the genetic algorithm, where selection, cross-over, and mutation happen at every generation. The game program only evaluates one set of neural network weights at a time, so one generation represents the selection, cross-over, and mutation for a new set of organisms.

IV. Results

The experiment ran for 15 generations. The program uses parameters as follows. It keeps five of the best, previous sets of neural network weights. Each generation (tournament) consists of ten games. We also refer to a game as a "run" in this section. These parameters are chosen partly from previous experience, as the following paragraph explains. There are 20 generations total per one complete experiment, and to get the total number of runs, we set up the computer to perform the experiment over and over. It reads in the `XML` file of all previous runs, so it has that data. The XML data notes the neural network weights, who won, and damages.

An earlier implementation remembered 50 best data sets, ran 100 games per tournament, and was set to repeat for 200 generations. The program is written in *javascript*, and the experiment displays the progress graphically. However, after running the experiment for over a week, we noticed that the response had slowed significantly, to the point where an arrow would take a couple of minutes to go from one part of the screen to the opposite, when it normally takes no more than a few seconds. Stopping the program by closing down the browser, then re-running the experiment gives a fast response. Thus, we set the program for a much smaller number of runs, and close down the browser once it completes.

Over the course of many generations, each behavior tended to come up more/less frequently. With the assumption that the later generations produce better adapted neural networks, a behavior that appears many times at the start and few times at the end indicates that that behavior is not well suited. On the other hand, a behavior that appears many times in the later runs indicates that it works well. Behaviors are represented as a percentage of that run, so that a run where behavior X occurs Y times gets the same value as in a run that takes twice as long, where behavior X occurs $2Y$ times. This gives an approximation of how well a behavior is suited for success, i.e. we can examine the how much a behavior is present in a weight set with a good fitness score.

The two characters start diagonally apart. The wizard (the defensive character that stands in for the human player) has bow for long-range attacks, while the ogre (the NPC to evolve) has a less-effective sling. While the player has sword, the ogre has hammer, better for short-range attacks.

Both characters have the same health at start (12). Each NPC has armor and a shield to deflect some damage. The armor automatically deflects some damage, but it wears down with use. The shield must be raised and correctly oriented to deflect damage, and only some behaviors use it. Behaviors allow the NPC to cover distance, perform a special attack, and use the shield. Attacks are automatic, assuming that the next attack should come as soon as the weapon allows. The wizard character uses a "back-up" movement pattern, identical to the behavior in Table I, except that this character only does this. Imagine being in the place of the characters; the relatively small wizard character would desire to stay far away from the much larger ogre, hide behind the shield, and launch projectiles. The ogre, meanwhile, would find that its projectiles are less effective than its opponent's. We "know" that the ogre should close the distance, but it must figure this out.

It is possible that two characters in a tournament could have a stalemate, where neither one wins. While this unlikely in our experiment due to the open layout of the room, it is easy to imagine, say if there is a wall between the two characters. To handle the possibility of a stalemate, tournaments are limited to 5000 ticks, with 50 ms between each "tick". This is set as an interval, with the *javascript* function `self.setInterval`. Thus, every 50 ms, the computer calls our `Tick()` function, which updates all sprite movements, detects for collisions, assesses damage, redraws the graphics, etc.

Table II shows the behaviors, their success, and their failure in this experiment. To compare these behaviors, we took all of the results, computed that run's percentages of each behavior, and summed it according to whether it produced a win or a loss for the ogre. In other words, adding all of the numbers in the "success" column to all of the numbers in the "failure" column results in the total number of games. The numbers reflect the fact that behaviors could be combined; perhaps the best set of weights would lead to multiple behaviors in one game. Interestingly, there is no clear winning behavior. The "ignore" behavior comes out as particularly bad, leading to the most lost games. As expected, some of the more defensive behaviors led to losses, while more aggressive behaviors tended to result in wins. The "chareg" behavior accounting for a relatively small number of successes is unexpected, however it also led to the least amount of losses, suggesting that it was under-utilized. The first generated set of neural network weights with a perfect score occurred in run 1206, that is, in the sixth generation.

Table II
NPC MOVEMENT RESULTS.

behavior	success	failure
hold	1.2	177.3
close	73.1	177.1
charge	31.4	67.1
back-away	0.8	76.4
retreat	1.2	134.3
check	122.8	75.6
special attack	1298.5	300.3
ignore	10.1	467.8

V. Conclusion and Future Work

Behaviors are aggressive, defensive, and neither. Behaviors allow the NPC to get closer, get away, or neither. We see in Table II that the behaviors leading the ogre to be aggressive and get close to the opponent, in general, tended to be successful. The results indicate no clear winning behavior, since even the favored "special attack" behavior led to a large amount of losses, too. What would a human do if he/she controlled the ogre? We expect that success is a mix of behaviors: some defense along the way is beneficial, that is, to use the shield as the ogre moves in close, then offense, such as charging, when the distance is small.

In this paper, we show a procedure for evolving behaviors of an NPC. It is a way to evaluate success of an individual set of neural network weights. It took 6 generations to get the first perfect score. Average fitness improved as the number of games increased, but later slowed. While running a longer experiment might result in additional good results, it is not clear that more is necessary.

When do we stop evolving the weights? The fitness function for a weight set is the amount of damage dealt divided by the amount received, as equation 1 shows. Thus, the fitness ranges from a minimum of zero to a maximum of 1200. Do we stop when one set of weights achieves a perfect score? This could be overly optimistic, since one run is not completely deterministic, especially since a human-controlled player will likely act differently. Do we stop when all of the top sets in the population achieved a perfect score? Or when randomly combined offspring consistently achieve this?

In a future update, we plan for the program to choose between previous sets of neural network weights to find a good match for the player, thus adapting. With a list of previous weights along with the relative ranks, the program can select a set with poor performance, then select sets with increasingly better performance according to how well a human player does against it. A good game should not be too easy nor too difficult, so we desire a game with several NPCs that progressively adapt based on the player. That is, the human-controlled player may need to fight several different ogres, and the program will select the $N + 1$ set of weights based on how the player does in round N. The game that we use for this experiment is fully functional; it is only when in tournament mode that the ogre fights the wizard in a room.

REFERENCES

[1] Michael Weeks, "Creating a Web-based, 2-D Action Game in JavaScript with HTML5", *Proceedings of the 52nd Annual Association for Computing Machinery Southeast Conference*, Kennesaw, GA, USA, March 28-29, 2014, pages 7:1–7:6. doi 10.1145/2638404.2638466, http://doi.acm.org/10.1145/2638404.2638466.

[2] J. M. L. Asensio and J. P. Donate and P. Cortez, "Evolving Artificial Neural Networks applied to generate virtual characters", *2014 IEEE Conference on Computational Intelligence and Games (CIG)*, August, 2014,ISSN 2325- 4270, pages 1-5, doi 10.1109/CIG.2014.6932862

[3] Teemu Heinimaki and Juha-Matti Vanhatupa, "Implementing artificial intelligence: a generic approach with software support", *Proceedings of the Estonian Academy of Sciences*, 2013, volume 62, pages 27-38 doi 10.3176/proc.2013.1.04

[4] J. Peralta, G. Gutierrez, and A. Sanchis, "Adann: automatic design of artificial neural networks", *GECCO '08 Proceedings of the 10th annual conference companion on Genetic and evolutionary computation (Companion)*, July 2008, pages 1863-1870, doi 10.1145/1388969.1388991

[5] S. Risi and J. Togelius, "Neuroevolution in Games: State of the Art and Open Challenges", *IEEE Transactions on Computational Intelligence and AI in Games*, 2015, volume PP, number 99, pages 1-1. doi 10.1109/TCIAIG.2015.2494596

[6] L. Hu, L. Qin, K. Mao, W. Chen and X. Fu, "Optimization of Neural Network by Genetic Algorithm for Flowrate Determination in Multipath Ultrasonic Gas Flowmeter", *IEEE Sensors Journal*, March 1, 2016, volume 16, no. 5, pages 1158-1167, doi 10.1109/JSEN.2015.2501427

[7] A. Ebrahimi and M. R. Akbarzadeh-T, "Dynamic difficulty adjustment in games by using an interactive self-organizing architecture", *2014 Iranian Conference on Intelligent Systems (ICIS)*, February, 2014, pages 1-6, doi 10.1109/Iranian-CIS.2014.6802557