Generation Next with Neuroevolution

Nick Monroe and Elizabeth Hagnauer

1. **Introduction and Problem**

Flappy Bird is a mobile game that was released in May of 2013, and by January of 2014 it had over 50 million downloads [1]. It was a free app that reached the top of the charts in both Google Play and Apple’s App Store for months straight. The game became “so addictive and overplayed”, as spoken by the developer of the game, that it was removed from both app stores in February of 2014. Even though the official version of the game was removed, many copycat and knockoff versions of the game have circulated ever since. The version within this project is Flappy Plane, which is our own copycat version of the original Flappy Bird. Within our game, the bird is replaced by a plane and the pipes are replaced by jagged rocks. The background and ground are also slightly changed to fit the scenery better. The rest of the game’s process and rules are identical to the original Flappy Bird idea.

Within this game, the plane will be continuously flying/flapping in the right direction. As the plane moves to the right, it will navigate between rocks that are attached to both the ceiling and the ground of the screen. These rocks will have a small opening that allows for the plane fly in between them, if done precisely enough. If the flying plane touches any part of the rock, then it dies and the game is over. The process of flying is done by tapping the screen which will make the plane jump upwards. The process of going down is accomplished by a system of gravity that will let the plane start to free fall if the user is not tapping the screen. Every time a user navigates through a set of rocks, the score will increment by one. Although the score will increase, in our version of the game the plane will not speed up, nor will the space between the rocks decrease in size. This means that throughout the entire game, no matter how far you get, the game will remain constant. The rocks opening position, however, will be randomly chosen throughout the game so that the game does not just operate on too basic of a loop.

In order to truly understand the game that this project is built around, you should visit this website [2] which allows you to play the original version of Flappy Bird. While this game does not sound difficult to play, when you attempt to play it yourself, it is hard to get past the score of 10, much less reach a score of 1,000 or 1,000,000. The purpose of this project is to develop a system that will be able to learn how to play Flappy Plane and get it to a user unreachable score. If the system created can successfully get past 10,000 points, then by our standards it will be considered to meet the “perfect system” criteria. However, we plan to create a system that would be able to get to 1,000,000 if it was left running for a long enough time. This can be done through a variety of different artificial intelligence methodologies. Our choices within this project were to use a genetic algorithm or a neural network to accomplish this specific task. We used neuroevolution [3] (which is a mixture of both artificial neural networks and evolutionary algorithms) because it would increase the overall efficiency of our system. If only a genetic algorithm or a neural network had been used, the results would be nowhere near as good as what is expected when using a mixture of the two (neuroevolution). Our approach to this problem is not one-dimensional, but rather two-dimensional in both complexity and difficulty.

**Approach**

The first part to this project is developing the user interface used to play the game by the neuroevolutionary algorithm. We decided to also make the game playable by a user to start off with, so that we could better understand the mechanics before we jumped into creating a neuroevolutionary algorithm to play the game. This was one of the pieces that took the most time within the project because it required a complete interface using the library “PyGame” that allowed the algorithm to run on. We started with just creating a plane that would move up when tapped or fall when not tapped. Once the basic plane movement was implemented, we decided to add a real-life system where the plane moves in accordance with upward acceleration and downward gravity that had been calculated using kinematic equations. Once the proper “flapping” of the plane had been completed, we were able to add rocks to the screen that would slowly move from right to left. Every time a set of rocks went off the screen, we would delete them from the system and add a new set to the end of the queue so the memory would not overflow with rocks that had been passed previously. With proper movements of the plane and the rocks in place, we could start on the implementation of the collision occurrence between the plane and any singular rock. If the plane collided with any pixel of the rock, then we wanted it to crash and the game to end. This was done by creating a “mask” within PyGame that uses 1 bit per-pixel to store which parts collide. This mask allowed us to know the exact moment when a single pixel of the plane touched a single pixel of any of the rocks. We then added in some extra fun to the game so that when the plane did collide with rocks, it would be set on fire and fall to the ground burning to make the death of the plane more realistic.

We also had to implement a collision detection with the ground and the ceiling of the game window. This was so that when the neuroevolutionary algorithm was introduced, planes wouldn’t be able to fly straight up over the rocks and survive without actually having went between the pipes in the proper way. After this we could implement the scoring system where the player’s points would increment by one whenever a set of rocks had been successfully passed without collision. This gave us the idea to leverage the “pickle” library that allowed us to store a high score value for the user. This is important because it is able to tell us later on how far our trained planes are able to fly in the game in a visual representation. Once we had the entire game implemented where the user was able to play it with the intended rules, we spent some time playing the game so that we could better understand the different inputs and outputs that would be needed within our artificial neural network. Below is the basic interface that we used within the system. On the title page, there are four buttons: Play Game which is where the user will play by themselves, Watch AI Learn, Watch Trained AI, and Play Against AI. The latter three had not been implemented at this point but were rather included on the screen as a “goal” of ours as an end result. We were successful in implementing all of the pieces by the end. There are also four planes that are circling the screen which was added purely for cool visual effects. The second picture is what the game currently looks like as the user is playing Flappy Plane.

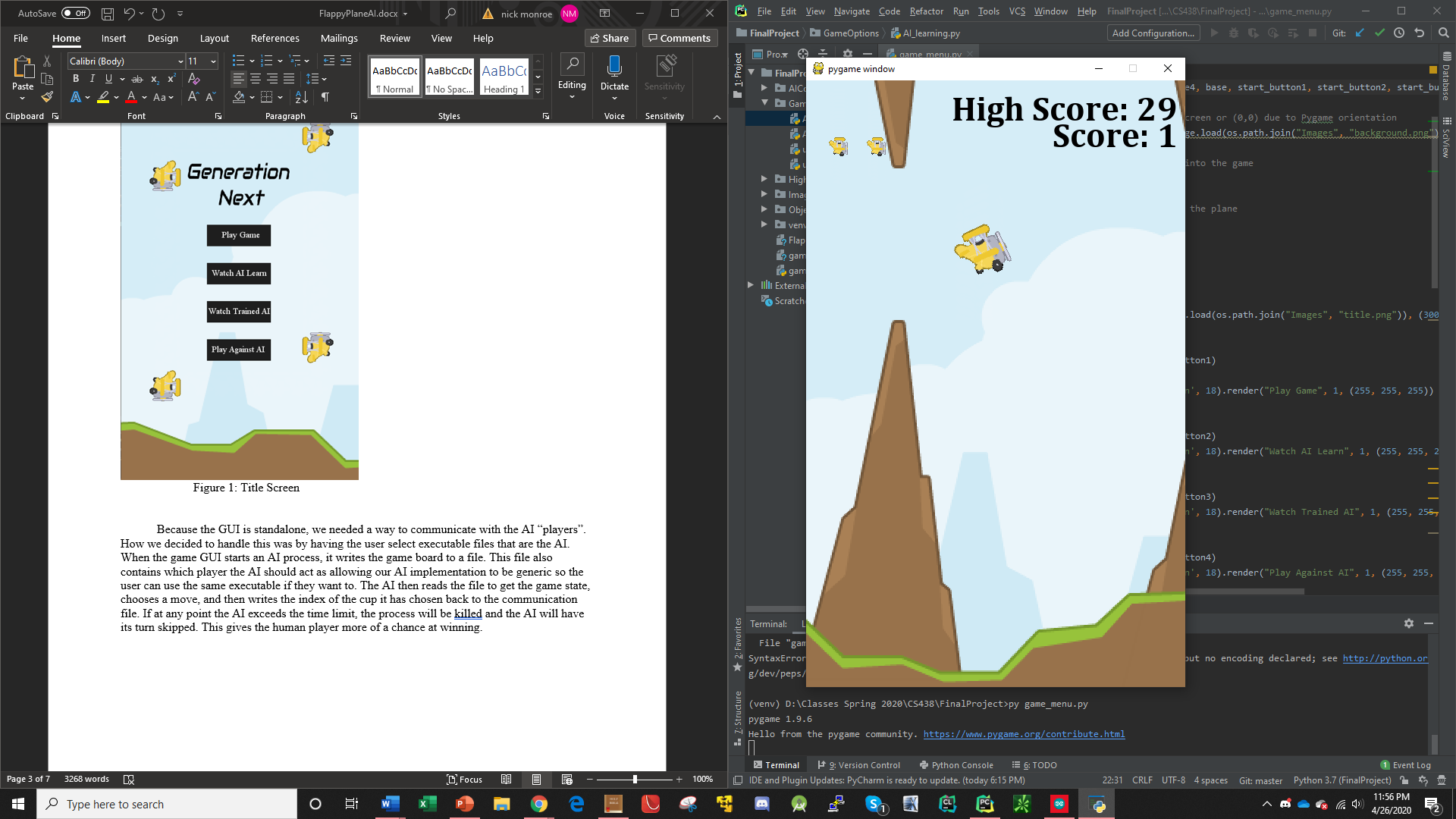


Figure 1: Title Screen Figure 2: User Playing

Once this heavy part of the project had been finished, we moved on to the implementation of the actual neuroevolutionary algorithm into our system. Or in other words, the basic requirement for completion of this project. For this, we decided to use a popular library called “NEAT” which stands for Neuroevolution of Augmenting Topologies [3]. NEAT is considered a mixture of the project requirements because it leverages both artificial neural networks as well as evolutionary algorithms. When implemented within our game, this means that there will be a population of genomes that will run a fitness function and evolve each generation based upon the output fitness value. Inside of the fitness function, a neural network will be used to learn this process and run each of the genomes individually. For the overall neuroevolutionary system, there is a multitude of parameters that need to be set for NEAT to run properly. We learned that if we just used the default values, then the system would perform poorly and not accomplish the task that was before us. That is because some of the parameters are very specific to the problem that they are being used to solve. This was the most time-consuming part of the implementation of the neuroevolutionary algorithm because it required a bunch of plug and play random parameter value choices to see if our learning process increased in efficiency. After a lot of random testing, we decided look into different research papers and textbooks that could give us insight into what each value the parameters should have. The following sources are all places where we were able to find some of this insight: [4], [5], and [6]. Without these, it would have been a nightmare to figure out which values for each of the parameters would create a proper system for the task at hand because of how many there were. With the neuroevolutionary algorithms parameters initialized, we could move on the fitness function that would be used to teach the generations.

Looking on an inside out view of the neuroevolutionary algorithm, the neural network is the innermost piece that we had to approach first. The goal behind the neural network is to determine when the plane should jump whenever it is about to encounter a set of rocks. The neural network will take three inputs that determine whether or not this should occur: the y-coordinate of the specific plane it is currently evaluating, the distance from the tip of the top rock, and the distance from the tip of the bottom rock. If the neural network gives an output of high enough value, then the plane will fly upwards and if the value is too low, then the plane will free fall. The main thing that needs to be recognized within the NEAT library implementation is that the neural network is not constant, but rather dynamic. This means that the number of hidden layers can change with each generation that occurs. The only thing that remains constant is the three inputs and the single output.

With the neural network in place, we had to implement the genetic algorithm that would hold it. We needed something to go in line with the neural network that would encourage planes to want to go further than the other planes in their population set. This required us to give a reward for planes whenever they make it further than other planes in the level. The best rewarding system that we found was to give each of the planes an additional fitness score of 0.1 and then 10 additional points for each pipe that was passed. This was to encourage the planes to want to pass through rocks instead of suiciding directly into the ground or the ceiling the moment the game starts. Once all of the planes have died, the fitness function should end and then the next generation will be reproduced and mutated using the highest performing plane within the last generation. As a result, every generation should be smarter and better than its parents. The nice thing about the NEAT library is that it allows for you to designate the size of each population set and the number of generations that should run, so in our implementation we let the user choose those two values. This is significant because it lets the user see how changing the number of generations and increasing the population size can either increase the performance at the end of all iterations or decrease it.

With our project requirements finished, we decided to add in the Trained AI and User Against AI pieces to the project. The Trained AI was done by running a training of 50 planes for 20 generations and killing the program once it hit 10,000 (so 10,000 is not the best possible score of our Trained AI, but rather just a point at which we decided to terminate the program). Whenever the program hit 10,000 points, it “pickled” up the data of the highest performing plane and saved it to a configuration file. This configuration file is considered the “perfect” AI for this game. So, for Trained AI, instead of initializing the system with our earlier made configuration, we initialized it with the parameters that the best-found plane ended on. This allowed for no learning to need to occur because the plane was already at an amazing generational performance state, hence the ability to call it Trained AI. The final piece, User Against AI, is simply just taking the original user playing version and doing an overlay of Trained AI so it appears as if the user is competing. If either the AI, or the user, hits a rock then the game will end. After multiple attempts on our part, the AI won every time. Feel free to try and see if you can beat it yourself.

1. **Discussion**

Overall, our project went a lot better than we were expecting when we came up with this idea. Our original mindset was to just get the learning process of the neuroevolutionary algorithm implemented and that was it. The ability for the user to play the game, watching the trained AI, and for the user to be able to play against the AI were not within our original mindset, but rather were considered possible future implementation additions once this semester was over. The fact that we were able to get all four pieces working within the time frame came as somewhat of a surprise to both of us. And, even better, is that all of the pieces of our implementation work amazingly. The Trained AI can play up to 10,000 consistently which was the highlight milestone when we were implementing this project. The ability for the user to play against the AI was just a fun little addition that we decided to add in because one of the possible project choices was for the user to play against an AI in some game. Adding that last piece made it so that our project contained: a neural network, an evolutionary algorithm, and the ability for the user to play against the AI. So, instead of just doing one of the project requirements, we decided to go the extra mile and give our project the dimensionality of all three possible requirements. The hardest part of this project was the choosing of the configuration of the neuroevolutionary algorithm within NEAT. It took a lot of testing various combinations of parameters until we got to a place where we were satisfied with how our AI learned how to play the game. Even though it works amazingly, we both still see that it could be improved by testing and changing around these parameters. Most of our insight for these parameters came from different research papers and textbooks. Without those, the process of configuration would have been a nightmare to figure out by ourselves. While this project took a decent amount of work to get to our finished product, it was by far the most interesting project that we have done while at SIUE so far. It was fascinating to see how some code, that really isn’t extremely complex, can learn how to play our game “perfectly” with ease, while it takes humans a lot of effort to just get a couple of points. It really gave us insight into the possibilities that AI has in different assets of our life that may seem complex to us, but with proper computational ability, become easy. In finality, if we were to restart this project right now and do it again, there isn’t anything in particular that we would change. Overall, our process to get to the finished product can be seen as a success considering the inability to meet with one another while working through each stage of the project.

1. **Future Work**

While our neuroevolutionary algorithm will eventually come to a “perfect” specimen after a certain number of iterations and a decent population size, it could still be improved. One piece of possible future work would be to make this learning process more efficient and streamlined so that it takes fewer generations with a smaller population size. This would require playing around with the configuration parameters, fitness evaluation, and possibly introducing more hidden layers within the artificial neural network. The reason this was not accomplished within our project was due to the short time given to implement the project. It could take months of testing various combinations to create a more efficient system in how it learns. Another possible future endeavor would be to add extra obstacles to the game. Right now, the plane will only try to avoid the ground, the sky, and the rocks that are suspended from the ceiling and attached to the ground. This is a fairly simplistic system, however, neuroevolution has the capability of learning how to accomplish tasks in much more enhanced systems. For example, these are papers where neuroevolution was used within game playing: Mrs. Pac-Man [7], Reversi [8], and Super Mario Brothers [9]. As you can see, all of these are a lot more complex than a plane that is moving across a screen attempting to fly through two rocks. A possible enhancement to this game could be made by adding planes that fly in the opposite direction that you have to dodge. An even harder task would be to have planes flying behind your plane shooting at you. Then, the neuroevolutionary algorithm would need to not only learn how to dodge the original obstacles in the game, but also the bullets that are coming at it from the opposite direction. This requires extra input variables to be passed into the neural network so that the plane can decide based upon the bullets velocity when it needs to jump or fall to avoid it. A third and final possible future work idea would be to add difficulty to the game as the score increases. Right now, the game remains the same whether the player is at a score of 0 or at a score of 1,000,000. The only randomness currently within the game is the location on the screen where the opening between the two rocks will be at. The game could be made more difficult by decreasing the opening between the rocks on the screen (until the space becomes equal to a set minimum size that the plane can fly through so that the game doesn’t become impossible), or by speeding up the plane so that the player needs to react faster to navigate through the rocks. These additions would require the neuroevolutionary algorithm to not only pay attention to the distance from the rocks, but also to the speed at which it is approaching the rocks and the size of the gap that is between the rocks. A third and final addition that can be made to the game is the adding of targets that your plane must shoot while still playing the regular game. This would require the neuroevolutionary algorithm to calculate where it would need to be positioned to shoot a bullet and hit the target. All while still making sure that it can fly up and fall down fast enough to make it to the opening of the rocks in time. Any of these future work ideas could easily be turned into a project based upon the added complexity that they would bring to the current system that we have in place.

1. **References**

[1] Artyom Dogtiev, “Flappy Bird Revenue – How much did Flappy Bird make?,” Business of Apps, 02-May-2018. [Online]. Available: https://www.businessofapps.com/data/flappy-bird-revenue/. [Accessed: 27-Apr-2020].

[2] M. McDonnell, “Flappy Bird,” Play Flappy Bird, 16-Apr-2014. [Online]. Available: https://flappybird.io/. [Accessed: 27-Apr-2020].

[3] K. O. Stanley and R. Miikkulainen, "Efficient evolution of neural network topologies," Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600), Honolulu, HI, USA, 2002, pp. 1757-1762 vol.2.

[4] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen, “Designing neural networks through neuroevolution,” Nature Machine Intelligence, vol. 1, no. 1, pp. 24–35, 2019.

[5] Omelianenko, Iaroslav. (2019). The efficiency comparison of NEAT libraries when looking for Double Pole-Balancing controller.

[6] Stanley, Kenneth & Miikkulainen, Risto. (2002). Efficient Reinforcement Learning through Evolving Neural Network Topologies.

[7] P. Burrow and S. M. Lucas. Evolution versus temporal difference learning for learning to play Ms. Pac-Man. In Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on, pages 53–60. IEEE, 2009.

[8] S. Y. Chong, M. K. Tan, and J. D. White. Observing the evolution of neural networks learning to play the game of Othello. Evolutionary Computation, IEEE Transactions on, 9(3):240–251, 2005.

[9] J. Togelius, S. Karakovskiy, J. Koutn´ık, and J. Schmidhuber. Super Mario evolution. In Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on, pages 156–161. IEEE, 2009.