Flappy Bird: Neuroevolution vs NEAT

Naud Ghebre, Ethan Jen, Matthew McBrien, Arihan Shah, Nahom Solomon

Project summary

Video games and mobile games are a large part of modern society in which multiple people can compete against each other. Video games have a predefined set of rules and a goal which make them fun and challenging. This also makes them a great playground for different reinforcement learning algorithms, machine learning algorithms which attempt to maximize some reward by training themselves as they explore the environment. Defining the rules and rewards of a real-world environment can be challenging, which makes video game a great place to learn about these algorithms. Deep Q learning was shown to be effective at learning strategies for classic Atari games in [1].

The goal of this project was to train a computer to play Flappy Bird using two different types of reinforcement learning algorithms. Flappy Bird is a game where the main character, a bird, flies in between 2 green pipes to score a point in which one pipe is coming down from the screen ceiling and the other pipe is coming up from the ground. The only command given to the bird is whether to flap its wings, giving an upward propulsion, or not, which causes a descent. The two types of reinforcement learning we will be using are Neuroevolution (NE) and Neuroevolution of Augmenting Topologies (NEAT). The project will compare how quickly each algorithm trains the computer to achieve certain scores and the overall ability of each trained model. In [2], NEAT was introduced and shown to perform better than traditional NE, so we expected similar results.

The project focused on utilizing data inputs from the game such as proximity to pipe center and vertical displacement. After initial testing, we also analyzed both algorithms when additional, red herring were supplied as neural net inputs. Both algorithms were able to teach the computer how to play Flappy Bird quickly when given good inputs. The difference in performance increased after adding the misleading inputs. The experiment resulted in NEAT out performing NE every time both experiments were run, and this was within expectations.

To further analyze the performance difference between the two algorithms, they could be applied to a game with a more complicated goal function, such as Pacman. A more challenging goal function would reveal more about the effectiveness of each algorithm. Additionally, NEAT could be compared against different reinforcement learning algorithms, such as Deep Q-Learning.

1 Detailed project description

1.1 Background Information

The first reinforcement learning technique utilized was NeuroEvolution (NE). In NE, the structure (or topology) of the neural net is selected by a human researcher before the experiment begins. Once the inputs, structure (typically a single, fully connected hidden layer), and outputs are specified, a genetic algorithm is run to tune the connection weights. The genetic algorithm scores each network in a generation and uses this to determine which networks are more likely to breed the next generation. This process repeats until some specified termination point, such as number of generations of goal score. The strongest networks survive and a neural network that is better at performing the task is evolved over the generations.

The second reinforcement learning technique utilized was NEAT. NEAT expands upon NE by also allowing for the topology of the neural net to mutate by means of the genetic algorithm. Some discussions, such as [3], have argued that the topology of the neural net affects how quickly it can learn a function, despite the fact that any fully-connected net with at least one hidden layer can approximate any continuous function. NEAT aims to improve upon NE by building a topology with the aim of improved speed over traditional NE techniques. NEAT can start with any number of hidden layers and any number of connections. Topology is evolved slowly by both adding nodes and adding connections between nodes, all while the weights between nodes are being mutated as in traditional NE. Altering a topology can significantly hamper performance in the short term, especially before the weights are properly tuned. This leads to a natural question of how to allow neural nets to alter their topology and tune the weights without killing off short term low performers. The answer to this is speciation and intra-species competition, which was introduced and discussed in [2].

NEAT has been shown to be effective in reinforcement learning problems as in [4–6]. Additionally, libraries exist for it in multiple languages, including python3, which will help us in our implementation of it.

1.2 Flappy Bird: Implementation

Flappy Bird was recreated in python3 and pygame for this project. Creating a custom version of Flappy Bird allowed the group to extract custom data from the game (such as x and y position of the bird) in real time and feed it back as input to the neural nets.

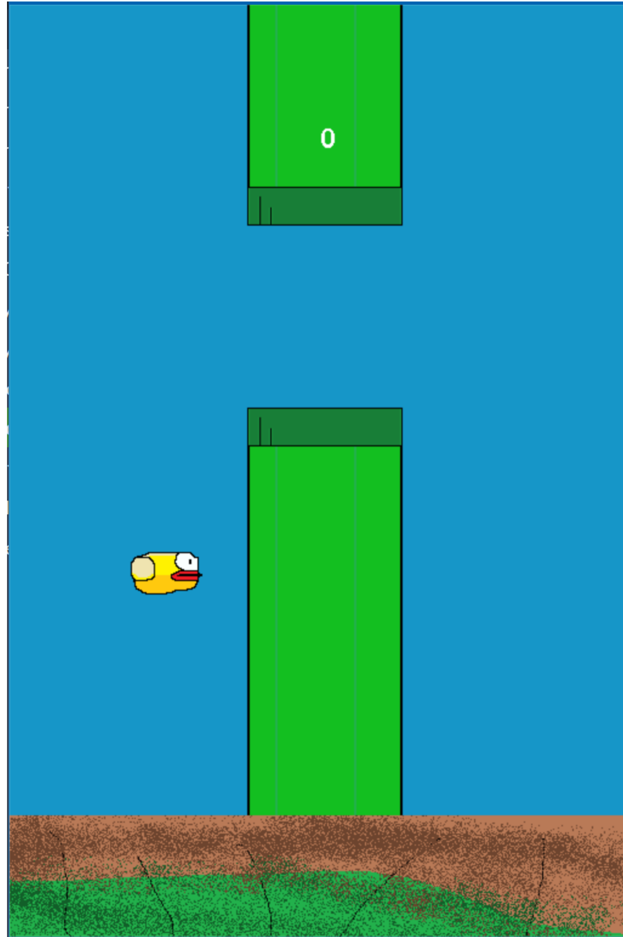


Figure 1: A screenshot of our version of Flappy Bird

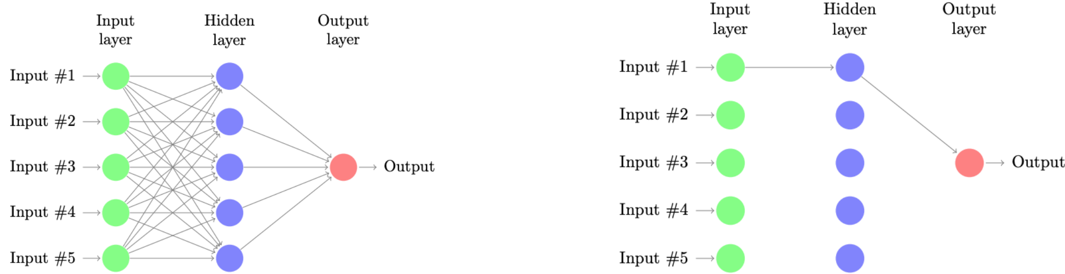
1.3 Neuroevolution: Implementation

NE was reimplemented by the group for this project. This required making three basic components: 1) a neural network, 2) a system for breeding and mutating neural nets, and 3) a system for creating new generations of neural nets.

For our implementation, creating a new neural net consisted of selecting new parents and then combining the weights. Parents were selected based on their score in the previous generation. If a particular neural net scored 100 points, and there were 500 points score total in the previous generation, that neural net would have a $\frac{100}{500} = 20\%$ chance of becoming a parent. Weights were then randomly selected from the two parents and assigned to the new neural net. Lastly, there was some small chance for each of the new neural nets to be mutated. Mutation would involve randomly adding to or subtracting from existing weights in the neural net.

Besides creating new neural nets, the new generation would also include the neural nets from the previous generation that performed the best, called the elite neural nets.

1.4 NEAT: Implementation



The above figures give a sense of how NEAT can work over time. At the start of the algorithm, each input is connected to each node in the hidden layer. With NEAT, multiple hidden layers can exist in which each node of the previous hidden layer is connected to each node of the next hidden layer. When the algorithm is running, connections and nodes have a probability of being added or removed after each generation.

NEAT implementation required the generation of a fitness function to evaluate fitness at each generation, a feed forward neural network, and a method for creating new generations after each member of a generation has failed.

The start of the algorithm takes in a configuration file with the base parameters such as number of inputs and size of the population and generates genomes (a specific version of the neural net) for each specific generation. The configuration file also contains specific details regarding genome creation and reproduction of the next generation. The NEAT implementation used typical configuration values for reproduction, stagnation, and genome provided by the python3 library. The algorithm then starts the action (which is the game in the experiment) by creating a feed forward neural net for each member of the population of that generation. The genome for each population member varies. When the action, the fitness is recorded and connections, nodes, and weights are altered based on fitness scores following the survival of the fittest model.

1.5 Deep Q Learning

Another reinforcement learning that was attempted was Deep Q Learning. Deep Q Learning expands upon the typical Q Learning method by evaluation of the Q function through a neural net. We built a reinforcement learner that acts upon a MLP neural net with 3 fully connected dense layers that are activated with Re-LU functions. Many of the previous research done trying to implement Deep Q Learning in video game settings have used convolutional neural nets that have taken images as input and we felt that in order to properly compare the performance of this as an extension to our current comparison, we must also use the same inputs. Thus, we shifted away from game images, and experimented with different bird-related inputs to train our net. The training time, and the time necessary to find the optimal neural net architecture proved to be the largest challenge. Our states were defined by an array of bird statistics at a given frame such as y position, x position, distance to pipe, and distance to ground and bird velocity. We implemented our neural net using Keras and our entire Deep Q Learner in python3. As of now, our DQN approach is shown to learn the game and shows the ability to make live decisions, however it seems as though some of the state action pairs that are learned tend not to make the most sense as we have run our algorithm for a little bit of time. We have researched that other Deep Q-Learning approaches take multiple hours to learn and require up to hundreds of thousands training iterations. We hoped to decrease the training time of our algorithm through changing the topology of the net, and inputs as well as using topology results from NEAT and NE to obtain a faster learning time.

2 Results

Flappy Bird is an extremely simple game to learn, the only reason it is challenging for people is because of the focus it requires. However, all that is required is to flap the bird's wings when you are below a certain point relative to the next pipe. If you are above this point, let yourself fall, otherwise, flap. This idea is illustrated in the following figure.

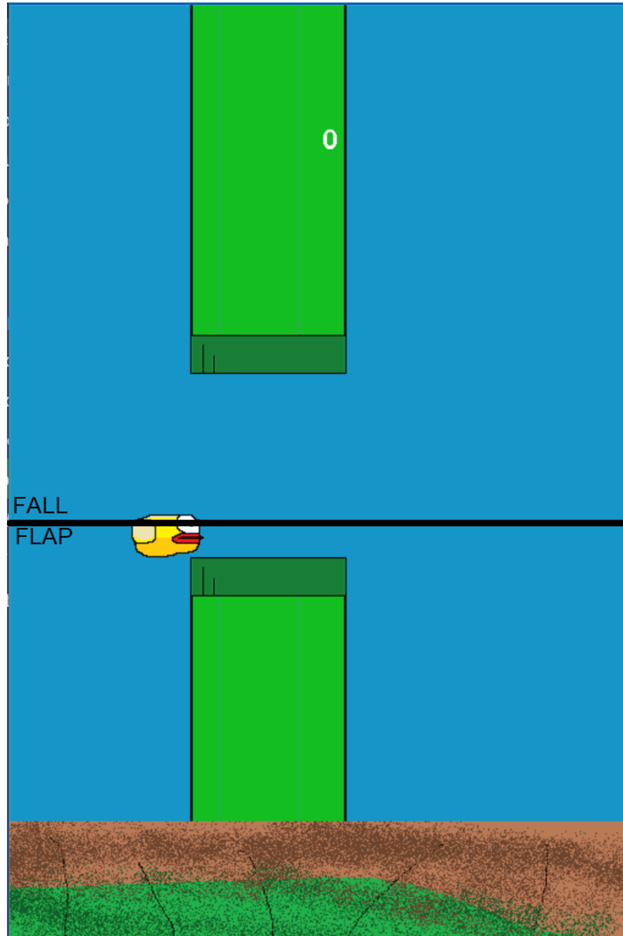
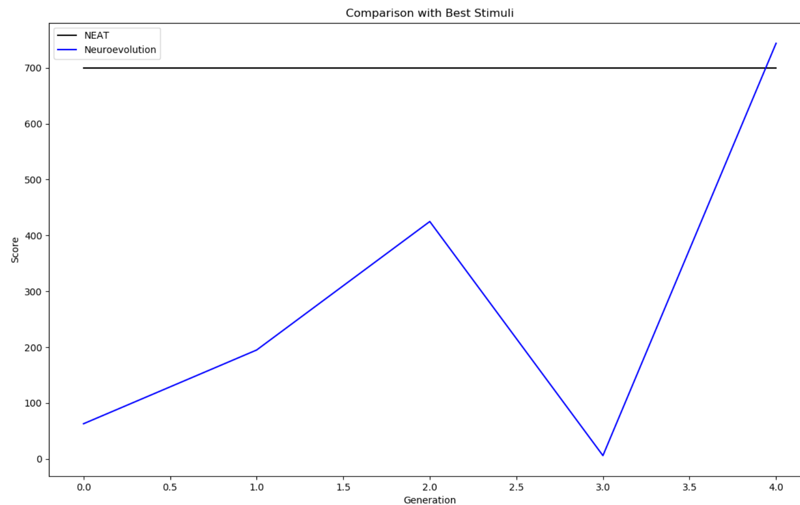


Figure 2: A visualization of the flap rule for Flappy Bird

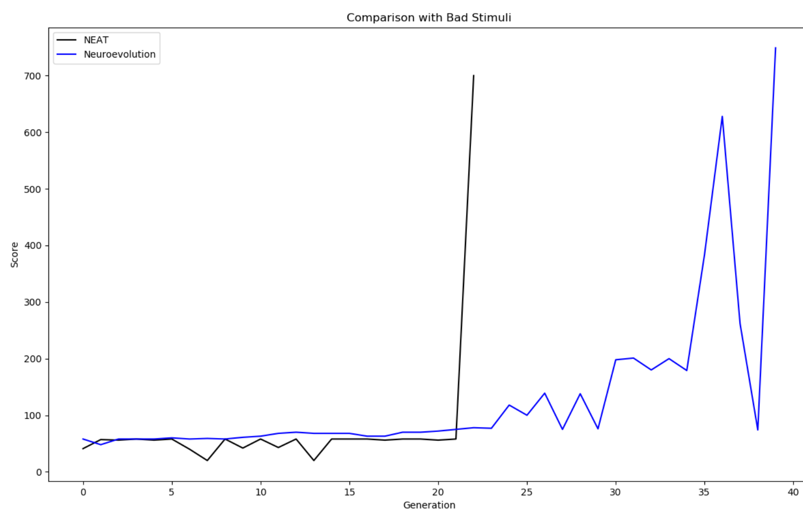
Following this rule guarantees that you won't fall too far and hit the bottom pipe and that you won't flap too early and hit the pipe above.

In our initial setup for experimentation, we provided the very best data as the input to the neural nets. This meant that the initial neural nets for both NE and NEAT consisted of a single input, the bird height relative to the height of the next pipe, and a single output, which was whether or not to flap. This means that the algorithms already have the ideal topology and only need to tune a single weight. This resulted in both algorithms learning the game very quickly, sometimes in the first generation, and leading to no interesting results about the difference between the two. A chart is shown below which shows score vs. generation. A typical example of both algorithms in this scenario is shown below. (Note that the algorithms were cut off once they achieved a true score of 10 pipes passed.)



This setup led to no learning about the differences between the two algorithms. It turns out that Flappy Bird is too easy to test these algorithms.

However, to stress the algorithms more, we changed our setup so that much more tuning had to occur to learn the correct function. Instead of a single input, we pulled five values from the game to serve as neural net inputs. These included (along with the good input as before): the bird's velocity, the bird's acceleration, the difference between the bird's height and the top of the pipe plus a constant, and the horizontal distance to the next pipe. Additionally, the initial topology of the nets was now two hidden layers each with five nodes and the nets were fully connected. This convoluted setup meant that the algorithms had no chance of guessing the function and had to learn it. Since the NE algorithm would have to iteratively tune out the bad inputs by altering the weights and the NEAT algorithm could completely drop connections and nodes, we expected to learn the game more quickly.



As expected, the NEAT algorithm learned the game much more quickly than the NE algorithm. We terminated this experiment when each algorithm had successfully passed 10 pipes. Repeating this experiment

leads to a high variance in the number of generations required for each algorithm, but the NE algorithm never outperformed the NEAT.

3 Responsibilities and Tasks

1. Project Proposal and Research
   - Nahom Solomon: Project summary, task distribution
   - Matthew McBrien: Neuroevolution and NEAT writeups
   - Naud Ghebre: Q-learning writeup
   - Arihan Shah: Deep-Q writeup
   - Ethan Jen: Deep-Q/Q-learning Research
2. Experiments
   - Nahom Solomon: NEAT implementation
   - Matthew McBrien: Neuroevolution implementation, game design
   - Naud Ghebre: Deep-Q Network Design/Implementation
   - Arihah Shah: Deep-Q Network Design/Implementation
   - Ethan Jen: Data Preprocessing
3. Project Poster and Report
   - Nahom Solomon: Poster and Report Design, Summaries
   - Matthew McBrien: Poster Graphics and Results, Report Results
   - Naud Ghebre: Poster Background Info, Detailed description in report
   - Arihan Shah: Poster Graphics, Detailed description in report, Report Results
   - Ethan Jen: Poster work

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[2] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[3] Byoung-Tak Zhang and Heinz Muhlenbein. Evolving optimal neural networks using genetic algorithms with occam's razor. *Complex systems*, 7(3):199–220, 1993.

[4] Marcus Gallagher and Mark Ledwich. Evolving pac-man players: Can we learn from raw input? In *2007 IEEE Symposium on Computational Intelligence and Games*, pages 282–287. IEEE, 2007.

[5] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366, 2014.

[6] Maximiliano Miranda, Antonio A Sánchez-Ruiz, and Federico Peinado. A neuroevolution approach to imitating human-like play in ms. pac-man video game. In *CoSECivi*, pages 113–124, 2016.