RL-Based Obstacle Game (Flappy Bird)

Written by,

Aadesh Mallya, Abhishek Kumar, Satvik Khetan
Khoury College of Computer Science
Northeastern University
Boston, MA 02115
mallya.aa@northeastern.edu, kumar.abhis@northeastern.edu, khetan.s@northeastern.edu

Abstract

Robotics and artificial intelligence have been prevalent for more than 10 years, and testing the effectiveness of new pathfinding or search optimization algorithms is a common task. These algorithms often need to be tested in simulated or artificial environments, which can be created specifically for this purpose. Games can also serve as a testing ground for these algorithms, allowing for the performance of the algorithms to be compared using artificial agents that follow the prescribed algorithm within the game environment. This paper explores the potential of using a genetic algorithm called NEAT, to play the Flappy Bird game without explicit human intervention.

Introduction

An intelligent agent is a software or hardware entity that has the ability to perceive its environment, make decisions, and act independently in order to achieve its goals. It can also learn from experience or use stored knowledge to improve its performance on tasks. The environment in which the agent operates is the context in which it receives input through sensors and sends output through actuators.

An approach to machine learning known as genetic algorithms, functions similarly to evolution. In a genetic algorithm, a population of potential solutions (called "individuals" or "chromosomes") is initially created. These solutions are then evaluated based on some measure of their fitness or performance, and the best-performing individuals are selected to "breed" and create a new generation of solutions. This process of selection and breeding is repeated over multiple generations, with each generation representing a small improvement over the previous one.

The key idea behind a genetic algorithm is that good solutions tend to produce even better solutions when combined, just as traits inherited from one's parents can be advantageous. By continually improving the solutions through this process of selection and breeding, the genetic algorithm can eventually find very good or even optimal solutions to the problem at hand.

In the context of RL, a genetic algorithm can be used to search for the best policies or strategies for an agent to follow in order to maximize its reward. The policies or strategies can be represented as chromosomes in the genetic algorithm, and the performance or fitness of each policy can be evaluated by running it in the environment and measuring the reward it receives. The best-performing policies can then be selected to breed and produce a new generation of policies, and the process can be repeated until a satisfactory solution is found.

In this paper, we explore the concept of the Neat algorithm (Neural Evolution of Augmenting Topologies), which belongs to the category of genetic algorithms known as Evolutionary Neural Networks (ENNs). ENNs are a type of genetic algorithm that is specifically designed to evolve and optimize artificial neural networks (ANNs).

We utilize the NEAT Algorithm to train a model to play the Flappy Bird game without specific human instructions.

NEAT Algorithm

The Neat algorithm (Neural Evolution of Augmenting Topologies) is a computational model that aims to simulate the processes of natural evolution in order to optimize the performance of artificial neural networks (ANNs). It was developed as a response to the limitations of traditional approaches to ANN training, which often require extensive human intervention and are prone to getting stuck in local minima.

The Neat algorithm utilizes a genetic algorithm to evolve and optimize the structure of the ANN over time. It begins with a simple network and gradually adds and modifies connections and nodes as needed in order to improve performance. The structure of the network is represented as a directed graph, with nodes representing neurons and connections representing the connections between neurons.

The Neat algorithm operates in generations, with each generation representing a small improvement over the previous one. At the start of each generation, a population of potential solutions (called "individuals" or "chromosomes") is created. These solutions are then evaluated based on their performance on a given task, and the best-performing solutions are selected to "breed" and create a new generation of

solutions.

The Neat algorithm has several key features that make it well-suited for optimizing ANNs:

- **Automatic feature selection:** The Neat algorithm can handle a large and varied number of inputs without requiring manual feature selection or engineering. This makes it particularly well-suited for problems in areas such as image recognition, natural language processing, and robotics, where the input data may be complex and varied.
- **Modular structure:** The Neat algorithm allows for the evolution of modular structures, with each module representing a subnetwork that is specialized for a particular task or input. This allows the algorithm to optimize the structure of the ANN in a more efficient and effective way.
- **Connection innovation:** The Neat algorithm introduces new connections to the network only when they are needed, rather than starting with a fully connected network and pruning connections as in traditional approaches. This allows the algorithm to discover new and potentially useful connections that may not have been found otherwise.

Overall, the Neat algorithm is a powerful tool for optimizing the performance of artificial neural networks and has been widely used in a variety of applications.

Related Work

While RL-algorithms are also suited to train video games, some specific difference between a genetic algorithm like NEAT provides over other RL algorithms are as follows -

- Search space: GAs search for solutions by generating and evaluating a large number of possible solutions, known as ”chromosomes,” and selecting the best ones to pass on to the next generation. Other RL algorithms, such as Q-learning and SARSA, use a more systematic approach to search for solutions, such as updating estimates of the value of each action based on the rewards received.
- Exploitation versus exploration: GAs tend to focus more on exploitation, or using the best solutions found so far, rather than exploration, or trying out new solutions. Other RL algorithms, such as Monte Carlo methods and temporal difference learning, often use exploration techniques to improve the chances of finding a good solution.
- Computational efficiency: GAs can be computationally intensive, as they require evaluating a large number of solutions and performing many iterations of the algorithm. Other RL algorithms, such as Q-learning and SARSA, can be more efficient, as they only update the estimates of the value of each action based on the rewards received.
- Applicability: GAs are well-suited for problems with a large number of possible solutions, such as optimization problems or tasks with a large number of states and actions. Other RL algorithms may be more appropriate for tasks with a smaller or more structured search space.
- Interpretability: GAs can be less interpretable than other RL algorithms, as the solution process is based on a combination of many different chromosomes rather than a set of explicit rules. Other RL algorithms, such as Q-learning, often produce solutions that are more easily understood by humans.

A major benefit of using NEAT over RL-algorithms is the fact that the NEAT Algorithm using Augmented Topology instead of fixed topology like in most RL-based ANN models, and this heavily impacted our decision to use this approach. Some advantages of using augmented topologies are as follows -

- **Increased flexibility:** Augmented topologies allow the network to adapt to the specific requirements of the task it is being trained on, rather than being constrained by a fixed number of connections. This can make the network more powerful and able to learn more complex patterns.
- **Better generalization:** By allowing the network to adapt to the specific characteristics of the training data, augmented topologies can help the network generalize better to new, unseen data.
- **More efficient training:** Fixed topologies may require a larger number of connections or layers in order to learn complex patterns, which can lead to longer training times. Augmented topologies can add or remove connections as needed, which can make the training process more efficient.
- **Improved performance:** In some cases, augmented topologies have been shown to perform better than fixed topologies on a variety of tasks, including image classification and language translation.

Overall, the use of augmented topologies in neural networks can offer a number of benefits, including increased flexibility, better generalization, more efficient training, and improved performance. Hence, we decided to go with this approach for the project.

Approach

Creating Flappy Bird game

We started with creating a baseline Flappy Bird game using the pygame environment. Following steps were included in the game -

- Created a Pygame window and set the title, size, and background color.
- Load the images that we needed for the game, such as the bird, pipes, and background.
- Defined the classes for the bird, pipes, and background objects. The bird class includes properties such as the bird’s position, velocity, and acceleration. The pipe class includes properties such as the pipe’s position and width. The background class includes the image for the background and a method for scrolling the background.
- Wrote the code to handle the game mechanics, such as the bird’s movement, the spawning and movement of the pipes, and the scoring system.

- Set up a game loop that updates the game objects and handles user input. The game loop also check for collisions and end the game if the bird hits a pipe or the ground.

  The baseline game appears like this -

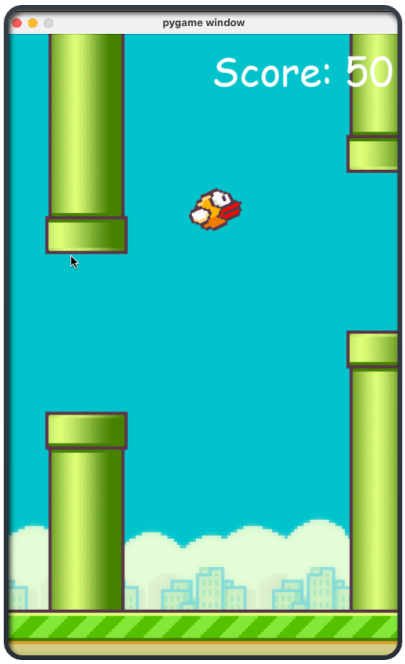

Fig 2. Initial game state

Defining the NEAT Parameters

Once the baseline game was ready, we proceeded with integrating a NEAT-algorithm based model with the game. The challenge here would be to devise an algorithm that should be able to detect the obstacle, find the distance between the obstacles relative to the current position of the bird, analyze the time to reach the obstacle, and then give an output by which the bird should successfully be able to void the obstacles. For this, calculating a valid fitness function was crucial.

The data that we considered while training the algorithm are: -

1. Score
2. Energy
3. Distance travelles
4. Vertical distance from pipe opening

Fitness function: The fitness function is used by the NEAT algorithm to determine which networks should be selected to produce offspring and which should be discarded. Networks with a higher fitness are more likely to be selected for reproduction, while those with a lower fitness are more likely to be discarded. The offspring produced by these selected networks are then evaluated using the fitness function, and the process continues until the desired level of performance is achieved.
Overall, the fitness function plays a critical role in the NEAT algorithm, as it guides the evolution of the neural networks and helps the algorithm find solutions to complex tasks. The fitness function we decided to use was as follows -

$$Fitness = (pos_{bird} * weight_{bird}) + (pos_{top} * weight_{top}) + (pos_{bottom} * weight_{bottom})$$

$$Fitness = distance - (1.5 * energy)$$

NEAT Hyperparameters Some important hyperparameters that we had to consider while training the model were as follows -

- **Population size:** This hyperparameter determines the number of neural networks in the population being trained by the NEAT algorithm. A larger population size may increase the chances of finding a good solution, but it may also increase the computational cost of training.
- **Mutation rate:** This hyperparameter controls the probability that a given neural network will undergo a mutation, which is a random change to one of its connections or neurons. A higher mutation rate may lead to more diverse solutions, but it may also increase the chances of introducing unintended changes that negatively affect the network's performance.
- **Crossover rate:** This hyperparameter controls the probability that two neural networks will produce offspring through crossover, which is the process of combining the connections and neurons of two parent networks to create a new, hybrid network. A higher crossover rate may increase the diversity of the population and improve the chances of finding a good solution, but it may also reduce the influence of individual networks on the population.
- **Complexity threshold:** This hyperparameter determines the minimum complexity of a neural network that is allowed to reproduce. Networks with a complexity below the threshold will not be considered for reproduction, which can help prevent the population from becoming dominated by overly simple solutions.
- **Add connection probability:** This hyperparameter controls the probability that a new connection will be added to a neural network during a mutation. A higher add connection probability may allow the network to discover more complex patterns, but it may also increase the chances of introducing unnecessary or detrimental connections.

After training the model multiple times, we observed that the optimal values for the hyperparameters were as follows -

```
NEAT Base parameters ->
fitness_criterion     = max
fitness_threshold     = 100
pop_size              = 15
reset_on_extinction   = False
activation_default      = tanh
activation_mutate_rate  = 0.0
activation_options      = tanh
num_hidden              = 0
num_inputs              = 3
num_outputs             = 1
```

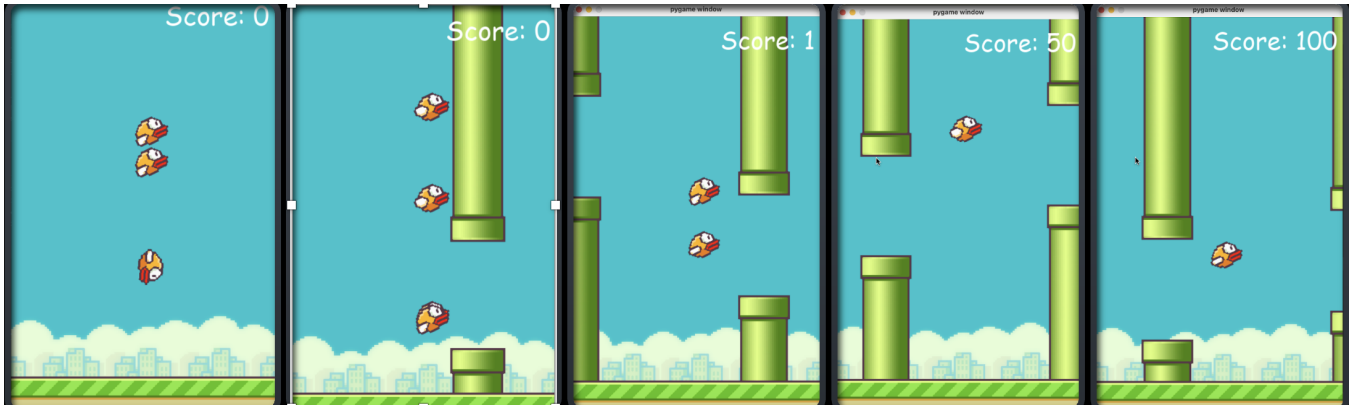

Figure 1: The game starts with multiple genomes/species of the bird proceeding towards the pipe. With the first collision, many of the genomes die off and the remaining genomes create new child genomes / neural networks which proceed towards the next pipe. As this process continues, the final trained genome/model is obtained which continues playing the game.

Experiments and Results

The algorithm does not require any past data or any specific dataset to be used. It relies on the real-time sensory data gathered by the artificial agent as it moves through its environment. The inputs to the algorithm are the y position of the agent, the distance of the agent from the top pipe, and the distance of the agent from the bottom pipe. Based on these inputs, the algorithm determines the appropriate action for the agent to take, such as jumping or falling due to gravity. The algorithm was implemented by starting with different initial populations. As shown in fig 1., the game proceeds reaching the max score once it is trained.

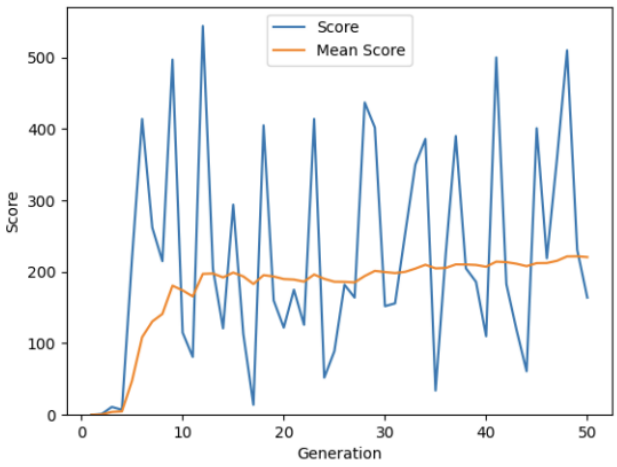

Fig 3. Scores over generations

Fig 3. shows the mean scores and the overall scores obtained by the agents over 50 generations and over a certain time limit.

As observed, the model trains over the initial generations and obtains score higher than 500 within 50 generations.

Conclusion and Future Scope

The flappy bird game can be used to effectively evaluate the performance of different algorithms. By creating an environment rather than relying on simulation, we have more control over the conditions under which the algorithm is tested. By altering the initial generation size, we have observed that the average score achieved by the agent can improve. The initial generation size also affects the speed at which the model is trained; a larger initial generation /species size leads to faster training.
We have found that the highest average scores are obtained when the model is run for more than 50 generations. In general, we have found that the performance of the algorithm improves as the number of generations and species increases.

In future scope, we can extend the model to accommodate other genetic algorithms like -

1. Deep Q-Learning
2. Proximal Policy Optimization (PPO)
3. Trust Region Policy Optimization (TRPO)
4. Distributional Reinforcement Learning with Quantile Regression (QR-DQN)
5. Hindsight Experience Replay (HER)

Credits:

- Abhishek Kumar: Collecting resources of the game, setting the layout of the game, model implementation and fine-tuning.
- Aadesh Mallya: Defining classes for the game, handling game mechanics (bird movement, spawning, scoring system), NEAT algorithm implementation.
- Satvik Khetan: Creating game environment on pygame and NEAT model integration and fine-tuning hyperparameters for the model.

Acknowledgments

We are grateful to Prof. Robert Platt for his guidance throughtout the semester and the TAs for their constant support. This project has been brought to fruition through the combined effort of all as a team. Thank you!