Solving Yellow-Spaceship game using GP and NEAT

Abstract

Game playing was an area of research in AI from its inception. Video games offer an amazingly interesting testbed for AI research and new ideas. The aim of this report is to show the potential of Genetic Programming and Neuroevolution of augmenting topologies in solving the Yellow-Spaceship game and compare the two strategies.

1. Introduction

Computer games have been linked with artificial intelligence (AI) from its inception. Computer games implement rich and complex environments that usually require a real-time response by the player and a dynamic strategy with an adaptive behavior in order to reach a good score or win depending on the kind of game.

Artificial Neural Networks (ANNs) are computational learning systems inspired by biological neural networks that use a network of functions to understand and translate some input data into a desired output. An ANN is based on a collection of connected units or nodes called artificial neurons, each neuron may be in principle connected to all the others. We can differentiate the neurons in 3 categories:

- input neurons: which receives the input data from the environment;
- hidden neurons: which elaborates the input data;
- output neurons: which outputs the desired output.

They are being deployed in a variety of tasks including playing computer games, but a core problem with the standard approach for training them is that we do not know a priori the best structure and we may have to try a lot of different architectures before finding one that provides good results. NeuroEvolution of Augmenting Topologies (NEAT) [2] is a genetic algorithm which attempts to simultaneously learn weight values and an appropriate topology for a neural network. In our implementation [7], NEAT is used to evolve an ANN that is evaluated to produce the output that encodes what the agent has to do.

Genetic Programming (GP) [3] is a method used to generate computer programs. Starting from an initial population of programs, the GP algorithm will evolve them in order to solve predescribed automatic programming and machine learning problems. In our implementation [7], GP evolves computer programs represented as tree structures, that are evaluated recursively to produce an output that says what the agent has to do.

2. Scenario and Problem

Yellow-Spaceship [1] is a space shooter game implemented using Pygame [6] in which the player, a spaceship, fights a number of enemies by shooting at them while dodging their fire: enemies can be aliens or spaceships. The player and the enemies move horizontally and can fire: actions can be defined as “move left”, “move right”, “do not move”, “fire”; moving and firing can be performed simultaneously so in the end we have in total 6 possible combinations of actions. The levels are structured in a way that there are always 2 aliens for each level except in levels multiple of 5 where there is an enemy spaceship. The spaceship and the enemy spaceship have an initial health of 50 life points whereas the health of enemy aliens increases as the level increases. Initially, the alien health is set to 10 life points and gradually increases by 8 life points every 10 levels. The laser damage for the spaceship is fixed to 1 life point whereas the one of enemies is initially set to 2 life points and gradually increases by 6 life points every 10 levels. As the game proceeds, it becomes more difficult and takes longer to beat the enemies and their lasers become more lethal. Fig. 2.1 shows a screenshot of the game.

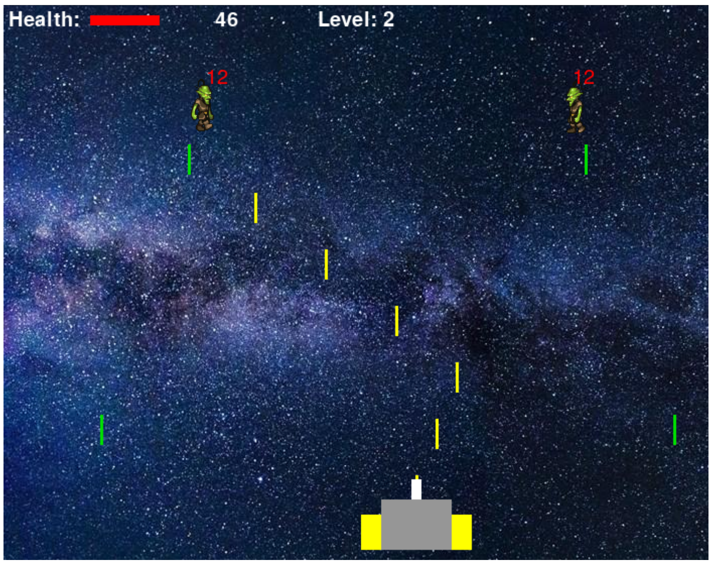

Fig. 2.1: Screenshot of the Yellow-Spaceship game [1].

The number of levels is not bound and the aim of the game is to kill as many enemies as possible and reach the highest level without losing the whole life.

3. Methodologies and Implementation

Starting from the Yellow-Spaceship game [1], we have adapted the code to our needs. We have restructured the whole code and in particular we modified the part that reads the action from the user keyboard and applies the actions to the game in order to be able to substitute the user input with the program computed action. The controlling action can be computed by an ANN trained using NEAT, or by a tree-based program evolved using GP, passing them the appropriate information about the state of the game.

The inputs from the environment for both NEAT and GP individuals are the following:

- the x coordinate of the battleship;
- the velocity of the battleship;
- the x coordinate of the first and second alien (if any, otherwise they are set to 0);
- the x and y coordinates of the first and second closest lasers (if any, otherwise they are set to 0);
- the x coordinate of the enemy spaceship (if any, otherwise it is set to 0).

In this way, there are a total of 9 arguments of type float passed to the individuals as input in order to decide the action to perform. Based on the actions the individuals perform, it is possible to evaluate their performance based on how much they moved on in the game.

The fitness function for the individuals of both the algorithms that we decided to use can be formulated as follows:

```
fitness = alien_kills * 10 + spaceship_kills * 50 +
  sum(battleship_healths_per_level) / 50 +
  sum([((current_alien_health - alien.health) / current_alien_health)
        for alien in aliens]) * 10 +
  sum([50 - enemy_spaceship.health
        for enemy_spaceship in enemy_spaceships])
```

The fitness function should be maximized and takes into account the number of aliens and enemy spaceships killed, the battleship health at each level and the health of aliens and enemy spaceships at the last level reached. There is no intermediate reward and the overall fitness is returned at the end of simulation.

In order to handle cases in which the simulation takes too much time because the spaceship is not killed, we introduced a maximum threshold on the number of frames: the game will be stopped after 1,000,000 frames. The threshold permits to prevent the execution from getting stuck and also incentivizes the NEAT and GP algorithms to learn how to reach higher levels with the same number of frames. We released this limit when showing the best individual execution to obtain actual fitness when the game is not interrupted.

To speed up the evolution process, individuals are evaluated only every 10 frames obtaining the action to perform: the previous action is applied for the rest of 9 frames. This value should be sufficient to permit the individual to have enough control over the spaceship actions and not be killed by enemies. In this way the evaluation process proceeds about 10 times faster and the spaceship piloting looks smoother.

3.1 NEAT

The implementation is based on the NEAT-Python library [4], which is a pure Python implementation of NEAT, with no dependencies other than the Python standard library.
We decided to evolve a feed forward neural network with the possibility of learning skip connections instead of a recurrent neural network since the actual state of the game is sufficient to train the neural network and no more information is required. The network has 6 output nodes and each of them encodes a particular action that the agent has to perform:

- 0: go left
- 1: go left and fire
- 2: stay still
- 3: stay still and fire
- 4: go right
- 5: go right and fire

For deciding the action to perform we take the one with maximum output value.

We decided to adopt the following relevant parameters:

- num_runs = 10, num_generations = 200 and no_fitness_termination = True
- pop_size = 100 and reset_on_extinction = True
- activation = sigmoid and aggregation = sum
- conn_add_prob = 0.5 and conn_delete_prob = 0.5
- enabled_default = True and enabled_mutate_rate = 0.01
- feed_forward = True and initial_connection = full_nodirect
- node_add_prob = 0.2 and node_delete_prob = 0.2
- num_inputs = 9, num_hidden = 9 and num_outputs = 6
- species_fitness_func = max, max_stagnation = 20 and species_elitism = 2
- elitism = 2, survival_threshold = 0.2 and min_species_size = 2

All the configuration parameters used by the algorithm can be modified in the configuration file named ‘configNEAT.txt’ present in the root directory of the project.

3.2 GP

The implementation is based on the DEAP library [5], which is a Python evolutionary computation framework which provides the main blocks for building Genetic Programming algorithms. Strongly typed genetic programming (STGP) is an enhanced version of genetic programming which enforces data type constraints.

A tree-based program has been evolved using STGP. The output of the program is one of the 6 encoded actions that the agent can perform. To encode the actions, we defined one class for each of them: A, B, C, D, E, F. The classes encoding is the following:

- A: go left
- B: go left and fire
- C: stay still
- D: stay still and fire
- E: go right
- F: go right and fire

The terminal set is composed of the action classes, some float values (5, 10, 15, 20, 25, 30) and the boolean values (“true”, “false”). The function set, or primitive set, is composed of the boolean operators “greater than”, “equal”, “and”, “or” and “negation”, the arithmetic operators “add”, “subtract” and “multiply” and the “if-then-else” function that can outputs a float value or an action class. Since the “if-then-else” function is the only function that outputs an action, it should always be at the root of the three.

We decided to adopt the following relevant parameters:

- num_runs = 10 and num_generations = 200
- pop_size = 100
- crossover_prob = 0.5
- mutation_prob = 0.5
- tournament_size = 7
- hof_size = 4
- max_tree_size = 500
- max_tree_height = 10

All the configuration parameters used by the algorithm can be modified in the configuration file named ‘configGP.txt’ present in the root directory of the project.

4. Results

In this section we will present the results obtained with the two approaches and a comparison between them and randomly piloted spaceships.

The evolution process stores at each run the best ANN or program generated with the relative graphs in the folder ‘runs’ according to the specified approach. In this way it is possible to relaunch them later and see through the graphical interface of the game how they behave.

4.1 NEAT

The NEAT algorithm managed to evolve a ANN that can reach level 42 in the game with a fitness of 1114 after about 160 generations. The ANN is represented in Fig. 4.1. As we can see the network is quite simple, with a small number of nodes and connections: it has just 10 hidden nodes and 63 enabled connections.

In Fig. 4.2, we can observe that the best fitness increases significantly in just a few particular generations, whereas the average fitness does not grow significantly and stays under 200. With this fitness trend, we can also notice that the speciation decreases significantly after 20 generations due to stagnation of a lot of species and just the 2 elitism species will survive after about 60 generations as shown in Fig. 4.3.

A weak point of NEAT is that the generated ANN is not really interpretable and we cannot understand why the network produces certain output values.

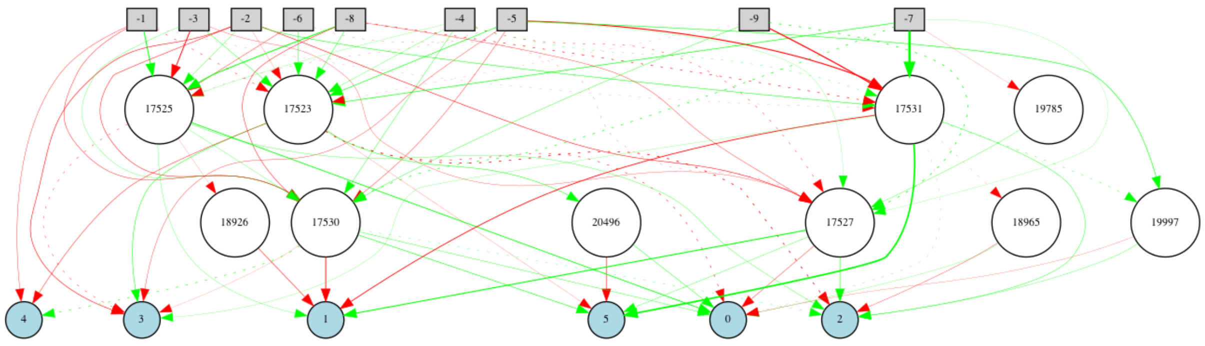

Fig. 4.1: Best ANN generated by NEAT.

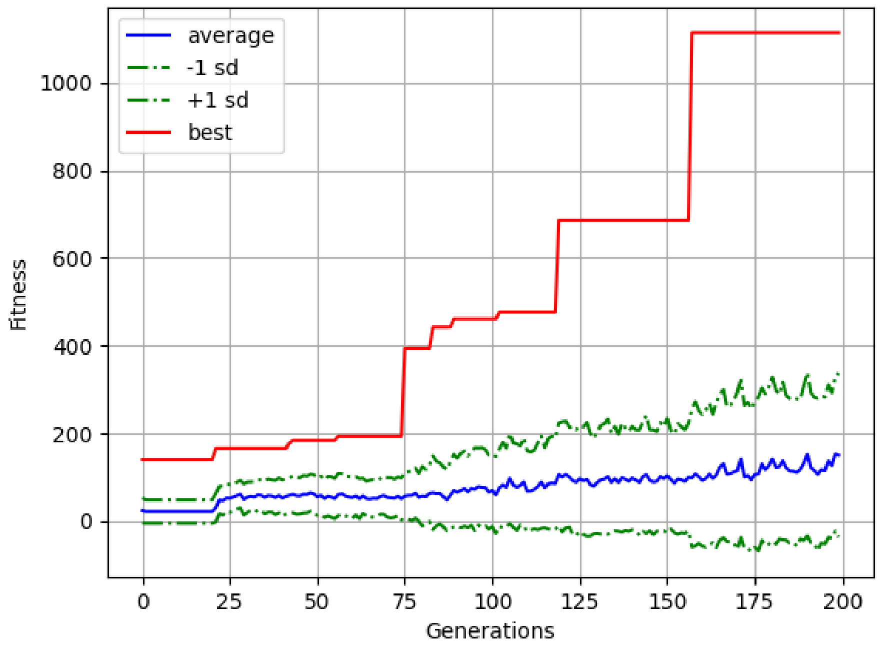

Fig. 4.2: Fitness trend of the best ANN generated by NEAT.

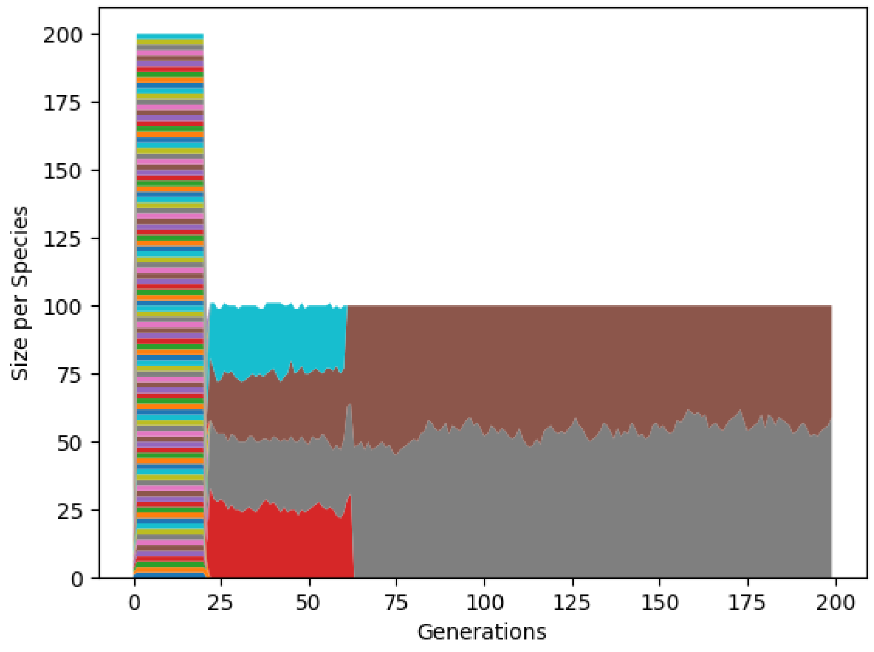

Fig. 4.3: NEAT speciation trend of the generations that produces the best ANN.

4.2 GP

The GP algorithm managed to evolve a tree-based program that can reach level 151 in the game with a fitness of 4061 during the evolution process and level 194 with a fitness of 5212 without limiting the number of frames. The tree of the program is represented in Fig. 4.4. As we can see the tree is quite complex, with 259 nodes and a depth of 10 (the maximum depth allowed). In Fig. 4.5, we can observe that the best fitness improves a lot in the first 60 generations reaching the frame threshold; in the remaining generations, the best fitness trend increases slightly, probably also due to this limitation to the number of frames. The average fitness trend is smoother and follows the best trend with a continuous increase: it reaches about 3000 fitness with a large variance (over 1000).

A weak point of GP is that the generated tree is a bit complex and can be simplified a lot, e.g. simplifying some if_then_else statements with boolean values directly taken from the terminal set (‘true’ or ‘false’) as conditions.

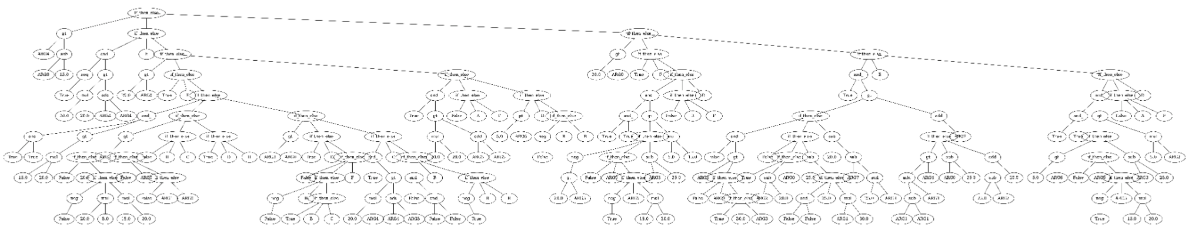

Fig. 4.4: Best program generated by GP.

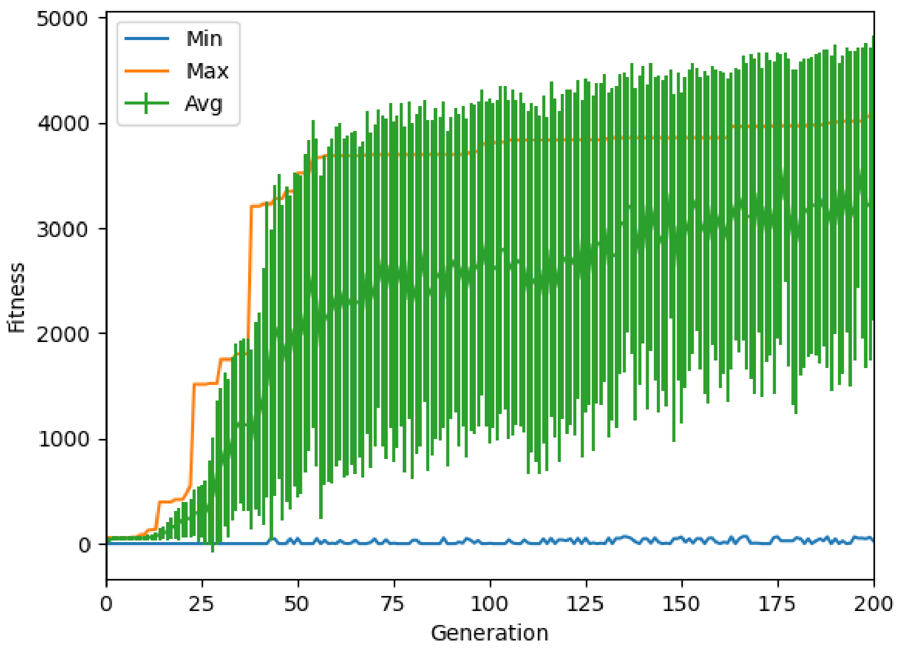

Fig. 4.5: Fitness trend of the best program generated by GP.

4.2 Comparison

With both techniques, we managed to find a program that is able to play the game well, learning to fire almost always to hit enemies while dodging their lasers.

GP demonstrated to have a bigger potential considering our scenario: thanks to the tree-structure and the functions provided, it is able to learn more complex strategies reaching level 194, much more than the best evolved ANN. NEAT, instead, struggles to evolve the network reaching in the best only level 42. NEAT is probably more suitable for different tasks when a program would not be the best option or even it would be impossible to build.

Moreover, a tree with a limited size can be more interpretable than a ANN and we can easily provide an explanation for what the agent is doing.

Observing the agent in action we can say that: both move quite smoother, even though neither of them is human-like, GP looks smarter with respect to NEAT. In fact, GP has a tendency to stay in the middle of the playing area dodging all the lasers, while NEAT tends to remain in the corners.

Across multiple runs, NEAT seems to obtain more stable results than GP that only a few times manages to reach very good results as shown in Fig. 4.6. Both the techniques are pretty good and much better than a randomly piloted spaceship that on average is just able to reach level 3.

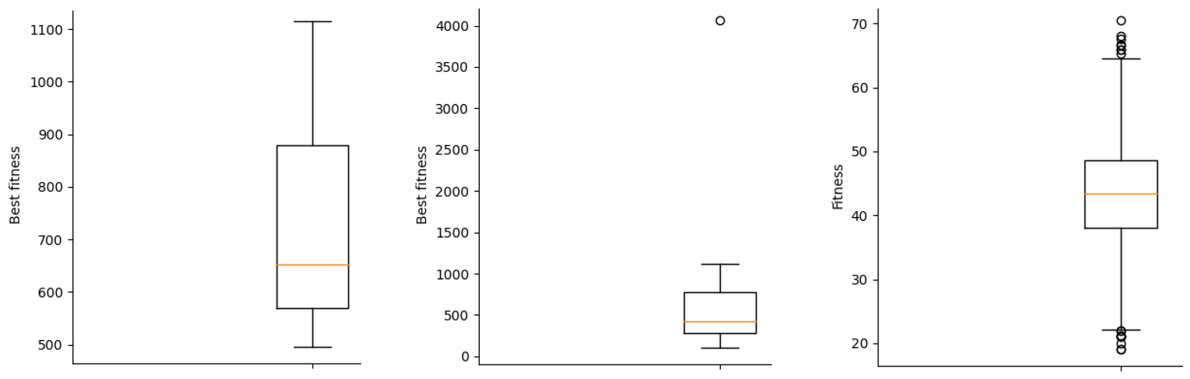

Fig. 4.6: Comparison between boxplots of NEAT, GP and randomly piloted spaceships.

5. Conclusions

Both are very good approaches and managed to find a way to play the game well. Even though, across multiple runs, NEAT seems to obtain more stable results, GP has demonstrated to have a bigger potential thanks to the tree-structured individuals and overall manages to reach very good results.

One of the issues that we encountered using GP is how to manage the different input types: one may implement functions taking into account inputs of different types or, alternatively, one can use a strongly typed primitive set as we do.

Another issue we have is that the execution of the algorithms takes a lot of time and computational resources, especially when the frame threshold is reached. Due to this, the number of runs and individuals should be limited.

In the end, we learned a lot about this field from this project, in particular how it is possible to apply bio-inspired techniques to real-world problems and how to benefit from it. They have proved to be a valid alternative to classic back-propagation algorithms for ANN and in this case we have finally learned new ways to do a kind of reinforcement learning considering the fitness as a reward.

References

[1] Original implementation of the game: https://github.com/ph3nix-cpu/Yellow-Spaceship

[2] Original implementation of NEAT:
https://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf

[3] Original implementation of GP:
http://www0.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/surveyRN76.pdf

Library used in the project:
[4] NEAT-Python library: https://neat-python.readthedocs.io/en/latest/index.html
[5] DEAP libraty: https://deap.readthedocs.io/en/master/index.html
[6] PyGame: https://www.pygame.org/docs/
[7] Project repository: https://github.com/samuelbortolin/Bio-Inspired-Spaceship