# A Minimal Training Strategy to Play Flappy Bird Indefinitely with NEAT

Matheus Cordeiro
*Teleinformatics Engineering Dept. (DETI)*
*Federal University of Ceará (UFC)*
*Fortaleza, Brazil*
*matheuscordeiro@alu.ufc.br*

Paulo Serafim
*Instituto Atlântico*
*Fortaleza, Brazil*
*paulo_serafim@atlantico.com.br*

Yuri Nogueira, Creto Vidal, Joaquim Neto
*Department of Computing (DC)*
*Federal University of Ceará (UFC)*
*Fortaleza, Brazil*
{*yuri, cvidal, joaquimb*}*@dc.ufc.br*

***Abstract*—A large number of algorithms to generate behaviors of game agents have been developed in recent years. Most of them are based on artificial intelligence techniques that need a training stage. In this context, this paper proposes a minimal training strategy to develop autonomous virtual players using the NEAT neuroevolutionary algorithm to evolve an agent capable of playing the Flappy Bird game. NEAT was used to find the simplest neural network architecture that can perfectly play the game. The modeling of the scenarios and the fitness function were set to ensure adequate representation of the problem compared to the real game. The fitness function is a weighted average based on multiple scenarios and scenario-specific components. Coupling the minimal training strategy, a representative fitness and NEAT, the algorithm had a short convergence time (around 20 generations), with a low complexity network and achieved the perfect behavior in the game.**

***Keywords*-artificial intelligence; autonomous agents; neuroevolution; flappy bird;**

## I. Introduction

The Neuroevolution technique is presented as a powerful strategy to evolve Artificial Neural Networks (ANN) in unsupervised learning problems (problems where there is no input and output table). It offers an alternative way to find the best configuration for an ANN without depending on a correct output value, which is commonly used to generate an error to optimize the networks settings through Descending Gradient algorithms like Backpropagation. On the other hand, the game Flappy Bird [1] shows itself as a promising virtual testing environment to optimize agents whose goal is to learn the behavior of nondeterministic phenomena. It is a popular game that was initially developed for mobile devices. Its goal is simply to keep the bird, the player, alive as long as possible by passing through a gap between pair of pipes without colliding with them (Fig. 1). When the screen is touched, the bird will perform a jump, on all other moments, it will fall gradually as a result of gravity.

In this type of problem, the agent interacts with the environment and then it responds in such a way that the agent's sensors receive information about the current state of the environment, through which it is calculated what actions the actuators must perform, that is, there is no information about what action to take, or whether the action chosen is good or bad, emphasizing a fairly probabilistic communication between the agent and the environment. In this context, the use of Flappy Bird as an environment for Neuroevolution algorithms tests is an excellent study, since the obstacle-transpose challenge finds application both in game development and in the field of Mobile Robotics (Smart Navigation). Neuroevolution can choose the best configuration of an ANN without depending on a set of correct actions within the Flappy Bird, since this strategy requires only a value that translates agent performance at the end of its lifetime, and through the choice of the best performances and the combination of the best settings the technique finds the ANN that solves the problem.

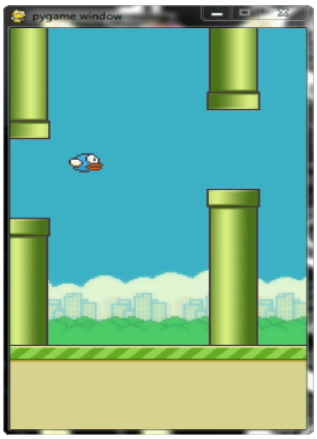


Figure 1. Screenshot of the Flappy Bird game.

In this work we apply a Neuroevolution algorithm known as Neuroevolution of Augmenting Topologies (NEAT) [14] in the Flappy Bird environment, using a minimal strategy that in addition to finding a simple agent to play the game, finds it quickly, i.e., few generations and playing short scenarios. The choice of NEAT is related to the fact that it starts from simpler configuration agents (topology and weights) and complicates this configuration over the generations, generally increasing the topology, so it is possible to guarantee that the solution found is the simplest for the problem.

[1] D. Nguyen, "Flappy bird," Apple App Store, 2013.

In Section II, we analyze works that make use of Neuroevolution in other games, works that use the Flappy Bird for other purposes, and also those that couple Neuroevolution and Flappy Bird. In the Methodology Section we detail the proposed strategy, and also the tools used. In the Results section the scores and fitness maximization charts, with the best topology found and the algorithm speciation chart, are presented in order to evaluate our technique.

## II. Related Work

The application of Neuroevolution in games has been used for some time with satisfactory results in the creation of intelligent agents capable of human and even super-human level of playing [10]. Within the scope of Neuroevolution, NEAT is one of the most popular and promising techniques in the context of games.

The uses of NEAT ranges from finding a Go player agent independent on the board size [15], to use in real-time environments such as Neuro-Evolving Robotic Operative (NERO). In the latter case, it is called real-time Neuroevolution of Augmenting Topologies (rt-NEAT) [13]. In a NERO context, NEAT is employed in the training of groups of virtual robots capable of playing against other teams. A very distant NEAT variation is content-generating Neuro-Evolution of Augmenting Topologies (cg-NEAT) [4], in which NEAT is used to generate the contents of a game called Galatic Arms Race (GAR). It allows the game to change during execution, increasing players' immersion and retaining their attention. We also call attention to NEAT's use in the development of playing agents for Fighting Games, where building a consistent form of measuring fitness is quite relevant [6].

NEAT can also be applied to the multiobjective paradigm in certain games, where NPCs must perform more than one task, such as Ms. Pac-Man [11]. In this game, a variation known as Modular Multiobjective NEAT is used in which the search types will always look for an agent with multimodal behavior [12].

FlappyBird is a popular game originally developed for mobile platforms. Currently, it has rereadings in different programming languages and has become an esteemed testing environment for different fields of Artificial Intelligence. The literature contains references to its use in the creation of a T2FuzzyLogicControl (T2-FLC) [9] as it is a classic obstacle avoidance problem with gravity in its implementation, causing the game to be comparable to an autonomous flight drone problem. Even more, FlappyBird is closely associated with temporal differences and can be used as a video training dataset (the frames) to construct a framework that makes temporal predictions in the presence of uncertainties [5]. It is also important to highlight the application of FlappyBird in Control studies, such as constructing an artificial player based on a Model Predictive Control (MPC) capable of controlling the bird's flight [17].

However, FlappyBird is mostly used as an environment for Reinforcement Learning problems, where it can help testing the use of Multiagent Reinforcement Learning with variations - using Epsilon Sinusoidal Function to reduce time [8] - or even investigating Transfer Learning [3]. In this context, it is widely used in works involving Deep Q-Learning, which is a paradigm that makes use of Q-Learning [16], a traditional Reinforcement Learning algorithm, to optimize Deep Neural Networks. This kind of work goes from a direct application of Deep Q-Learning to FlappyBird [2], to comparative studies with other approaches, for instance, employing different memory scalability such as Scale-Invariant Temporal History (SITH) [1], a way to compress the past. It is also worth highlighting a work that use Deep Q-Learning with FlappyBird emphasizing how the AI communicates with the Learning Environment through WebSocket [19].

## III. Methodology

Three components are essential to this work: fitness function calculation, presented in Section III-A; how to expose the scenario to the agent, presented in Section III-B; and phenotype settings, presented in Section III-C.

### A. Fitness

To compute the fitness of the agent three components are used, called Scenario Fitness Components (SFC):

- Traveled Distance (TD): a counter that increases in each interaction of the agent with the environment;
- Score: the number of pairs of pipes already transposed;
- Y Factor ($\Delta Y$): the value obtained when the agent fails on any part of the scenario, which is calculated by the difference between the y coordinate of the agent and the y coordinate of the midpoint of the passage between the following pipes, defined as:

$$\Delta Y = y_{agent} - y_{passage} \quad (1)$$

Y Factor plays a crucial role since it enables to penalize the performance of the agent in the fitness function with a value that expresses how far the agent was from the objective, the passage between pipes, before failing. In Fig. 2 three Y Factors are highlighted, $\Delta Y_1$, $\Delta Y_2$ and $\Delta Y_3$, each of them corresponding to a different agent of the same population. These values will only be considered when the agent fails, defined by the collision with any part of the scenario. When this occurs the interaction with the environment ceases and the agent's performance is measured.

The Y Factor is used to transmit a quality value to the fitness calculation, since it differentiates agents that failed in the same pair of pipes but in different heights. The goal is to ensure that agents closer to the passage are considered better than others further away, something that would not be possible if the performance of the agent was based only in TD and score.

As shown in Fig. 2, if all agents failed at the exact same moment, they would have the same TD and score, however the Y Factors would be different, such that $\Delta Y_1 < \Delta Y_2 < \Delta Y_3$, showing that agent 1 is closer to the passage. Also, the value of $\Delta Y$ is absolute since it aims to penalize the performance of the agent based on the distance from the passage, regardless if it was above or below the agent.

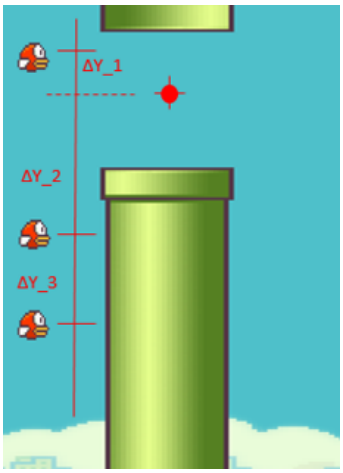


Figure 2. Y factor difference between three agents with the same TD and score.

These three SFCs were combined in a single expression, called scenario fitness function (SFF):

$$SFF = \alpha \times TD_S + \beta \times Scores_S - \gamma \times \Delta Y_S \quad (2)$$

This expression shows that the scenario fitness of an agent is a linear combination of SFCs and three constant weights, $\alpha$, $\beta$, and $\gamma$, which regulate the importance of each SFC in fitness. The goal of the agent is then to travel the maximum possible distance, obtaining the highest score and stay closer to the passages. The $S$ subscript corresponds to the standardized version of the component:

- $TD_S = TD/TD_{max}$, where $TD_{max}$ is the TD when transposing the maximum number of pairs of pipes in a scenario.
- $Scores_S = Scores/Scores_{max}$, where $Scores_{max}$ is the maximum score possible in a scenario.
- $\Delta Y_S = \Delta Y/W_h$, where $W_h$ is the height of the game window, that is, the largest value of the y coordinate.

Obtaining SFFs from the game and combining them into an agent fitness function (AFF) evaluates how fitted an agent is:

$$AFF = \frac{\sum_{i=1}^{MS}(k_i \times SFF_i)}{\sum_{i=1}^{MS} k_i} \quad (3)$$

Thus, (3) shows that the fitness of an agent in a specific generation is given by a weighted average based on the fitness that this agent obtained on every scenario, which means that some scenarios will have more importance than others. $MS$ is the total number of scenarios, which have different aspects.

### B. Scenarios

The main objective regarding the way the scenario is exposed to the agent during training is to reduce the number of pipes while still preserving the ability of the algorithm to converge in a short time. Three scenarios with a specific variation of the gap between the passages were presented to the agent as shown in the example of Fig. 3. Therefore, the total number of scenarios is only three, i.e. $Ms = 3$.

The gaps were chosen strategically as 0 pixels, and approximately 80 pixels and 160 pixels. These values provide a great gap versatility since 0 and 160 pixels are close to the vertical limits of the screen and 80 pixels being the midpoint of them. An agent that dominates its performance completely in scenarios with these gaps will be able to handle any kind of passage variations, since it learned how to transpose the small gaps, medium gaps, and large gaps, mastering the game's gaps domain.

Since the number of pipe pairs in each scenario equals three, $Scores_{max} = 3$ and $TD_{max} = 195$, such that 195 is the TD for the agent that transposes three pairs of pipes. The reason why there are three scenarios and not two ("smaller" and "larger") is supported by the fact that a second scenario allows a faster and smoother convergence, since, while the agent is solving the intermediate challenge, he is getting parameters to solve the more complex challenge ("larger"), leading to less abrupt and less time-consuming convergence.

In the context of the constituents of each scenario, three pairs of pipes were chosen as shown in Fig. 3. Items (a) and (b) show a flat gap; (c) and (e) present fall gaps; (d) and (f) show climb gaps. These types of gaps comprise the summary of the movements that an agent could execute in a large scenario, composed of thousands of pipe pairs with different gaps between them.

### C. Network Phenotype, Parameters and Tools

The agent's phenotype is represented by an ANN that has three input neurons, one output neuron and with an internal structure that will only be defined after the training by the evolutionary algorithm, as shown in Fig. 4. The output of the network is the probability of performing a jump. Fig. 5 shows all inputs in a frame, which are:

- Ay: Agent's y coordinate y;
- By: Y coordinate of the tip of the bottom pipe ahead;
- Cy: Y coordinate of the center of the passage between the pair of pipes ahead.

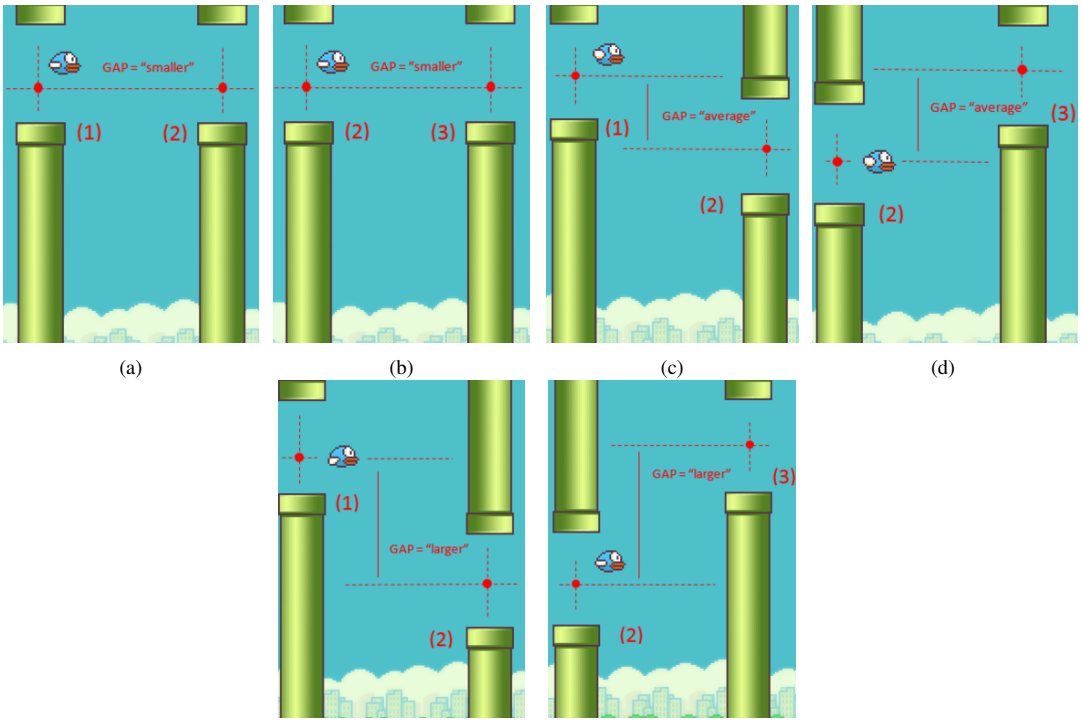


Figure 3. The three types of scenarios used in learning.

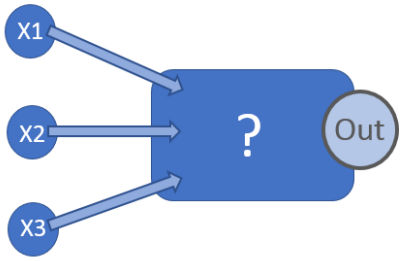


Figure 4. Network architecture with an initially unknown internal structure.

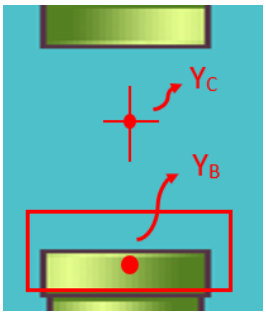


Figure 5. Inputs.

When the agent receives the information given by the environment (Ay, By and Cy), it is processed by the network, generating a probability of executing a jump. This probability will determine the action to execute according to the rule: *Action* = *Jump* **if** (Out $\geq b$) **else** *None*, where $b$ was 0.4.

In the configuration of the NEAT parameters, the following values were established, which can then be used to reproduce this work:

- Population: a population of 30 individuals is used. It is not a big population, which enables a faster execution, and it is large enough to allow a genomic diversity.
- Elitism: an elitism of 18 individuals was chosen, a value that compared to the previous parameter is equivalent to 60% of the population. This value allows the preservation of innovations and it is small enough to not incur in stagnation or the absence of innovations.
- Compatibility threshold: the value given to this term is 3.0, which is large enough to not create many species initially and is small enough to not completely prevent the formation of species within the population. It is interesting to note that a large number of species in a population can generate intersections between species which can lead to problems in optimization especially when the population is small.
- Mutation rate: this parameter has a value of 0.05, which causes the connections to not activate immediately as the topology increases, they are activated gradually.

- Weight and Bias: the average initial generation of weights and bias were 0 and 0.01, respectively, with a standard deviation of 1.3 in both. These values allow a slightly more varied distribution when the weights are generated, thus getting closer to the topologies of better performance.
- Probabilities to add or remove connections: the likelihood of adding connections and the likelihood of removing connections were set to 0.7 and 0.2, respectively. These values symbolize a greater affection for a robust topology through connection creation, a proposal aimed at solving complex problems. If the topology does not increase, but the optimization finds good individuals it is because a simpler topology was sufficient for the problem, since NEAT starts from less complex to more complex topologies.
- Probabilities to add or remove nodes: the probability of adding nodes was set to 0.7 and the probability of removing nodes was set to 0.2, as in the above parameters, and they have a similar explanation.

The algorithm ran for 100 generations using as the activation function a Sigmoid. Given the simplicity of the agent's decision and since Descending Gradient is not part of the process, this function uses all of its logistical power without drawbacks.

With respect to the parameters used in the calculations of the SFFs and the AFF the following values were used:

- SFF ($\alpha = 1.0$, $\beta = 1.0$, $\gamma = 0.08$): Note that the weight of the Y Factor is very low compared to the others. This was established for two reasons: (i) the Y Factor before being normalized gets a small value when compared to the other parameters during the generations and when the normalization is done this distance is lost, so a small value of $\gamma$ allows a reduction of the $DeltaY$ magnitude; and (ii) at the beginning of the generations the Y Factor has very high values, which is the opposite of its primary function of being a differentiation of similar fitness when performing fine adjustments in their scores, which can harm the convergence speed.
- AFF ($k_1 = 1.0$, $k_2 = 2.0$, $k_3 = 6.0$): SFFs weights for more complex scenarios were given a higher value to strengthen good performances in more difficult environments. This is the reason why Scenario 2 has a slightly greater weight than Scenario 1 (easy) and Scenario 3 (hard) has a much greater weight than Scenario 2. This strategy allows a faster convergence of the algorithm since the agent finds the best performance faster.

This work was constructed using NEAT-Python [7] and PyGame Learning Environment (PLE) [18]. PLE is the library of the agent's interaction environment, which is triggered when calculating its fitness is needed, making it communicate with the environment through his sensors and actuators. This relationship is best explained by Fig. 6.

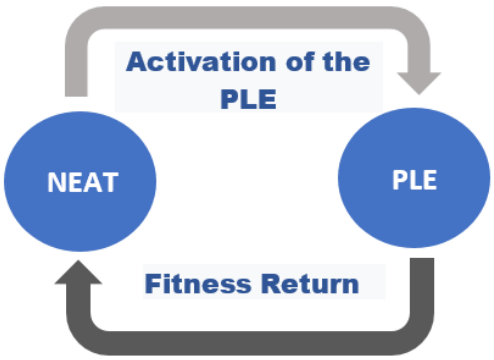


Figure 6. Relationship between NEAT and PLE.

To execute the steps discussed in Sections III-A and III-B, some modifications were made to the PLE to allow the definition of the structure of the scenario before the beginning of the optimization. The changes are needed to construct the three types of scenarios with different gaps between the pairs of pipes. These modifications in the code and a video showing how the best phenotype works are available online [2]

## IV. Results

The results of fitness and scores are presented in Section IV-A. The speciation chart is shown in Section IV-B. Finally, the final network topology is presented in Section IV-C.

### A. Fitness and Scores

Fig. 7 presents the fitness results. The x-axis of the chart corresponds to generations, from the beginning going up to 100, while the y-axis corresponds to the average fitness (blue line) and the best fitness (red line) on every generation. It can be seen that in about generation 20 the fitness stabilizes until the end of the tests. The algorithm is able to achieve an optimal score since the first generations, showing the robustness of the applied strategy.

The score chart (Fig. 8) is very similar. The blue line is the average score and the red line is the best score achieved on every generation. The stabilization of the scores occurs again around the 20th generation, agreeing with the fitness.

In both figures the mean values stabilize in values that represent around $2/3$ of the maximum values in each chart. The maximum score in the fitness chart is equal to 2.0, the value when all SFFs are equal to 2.0, which occurs when $TD = 195$, $scores = 3$ and $\Delta Y = 0$. This means that the agent was able to pass through all three pairs of pipes and did not shock into anything. This implies that the $SFF = \alpha + \beta = 1.0 + 1.0 = 2.0$, leading to a $AFF = [(1.0 \times 2.0 + 2.0 \times 2.0 + 6.0 \times 2.0)/(1.0 + 2.0 + 6.0)] = (18.0/9.0) = 2.0$, since $k_1 = 1.0$, $k_2 = 2.0$ and $k_3 = 6.0$. On the other

[2]https://github.com/matheus123deimos/Papers

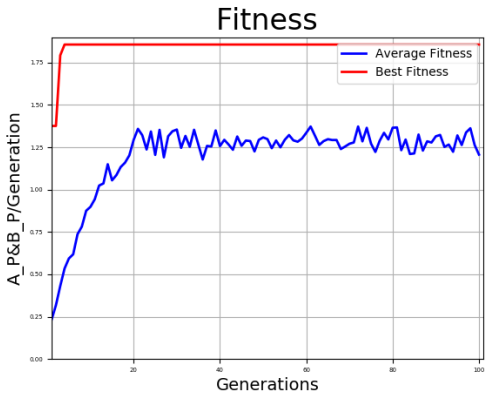


Figure 7. Agent fitness chart.

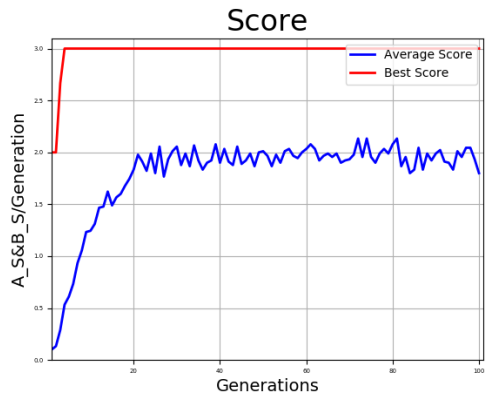


Figure 8. Agent scores chart.

hand, the maximum score possible is equal to three, which corresponds to transpose three pairs of pipes.

When put to play the real game with random types of obstacles, an agent is able to transpose all pipes indefinitely. This can be seen in Fig. 9, in which an agent achieved a score greater than ten thousand and is still playing the game. This shows that the minimal training strategy was very successfully to generate agents with optimal behaviors.

### B. *Speciation*

Fig. 10 shows the Speciation Chart, in which the x-axis informs the generations and the y-axis informs the size of the species that is at most 30. Since only a single species was enough to converge and solve the game, only a single color is shown.

### C. *Topology*

The phenotype of the best agent of the last generation is shown in Fig. 11. It is interesting to note that this phenotype is a perceptron, a very simple model of an artificial neural network. The network weights computed for each input are:

- **W_Ay:** 0.6285.


Figure 9. Example of an agent playing the full game, achieving a score of more than ten thousand and still alive.

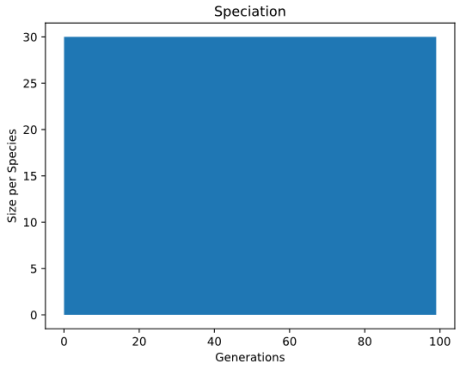


Figure 10. Speciation Chart.

- **W_By:** -1.5107.
- **W_Cy:** 1.6638.
- **Bias_Node:** -1.0770.

The weight of the input Ay ensures that when the agent is with a small value of the y coordinate, i.e. the agent is far above the passage, the sigmoid function will result in a very low jump probability. This implies that the agent will tend to go down by the action of gravity. However, when the agent has a high value of the y coordinate, i.e. the agent is far below the passage, the sigmoid function will tend result in a very big jump probability, which will lead the agent goes up.

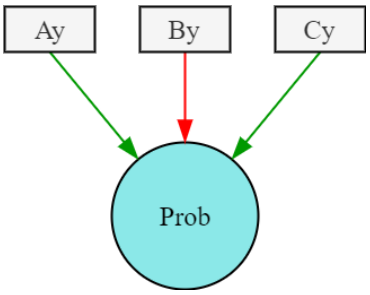


Figure 11. Topology found by NEAT.

Evaluating all of the three results together it can be seen that, our methodology turned the game into a problem simple to solve by the algorithm. Those results show that training agents using a very restricted training environment, only three types of obstacles, can lead to optimal behaviors.

## V. Conclusion

In this work, the authors propose a minimal training strategy to generate agents capable of achieving optimal scores in the game Flappy Bird. By using scenarios with only three different types of obstacles to train the agents, they evolved a neural network to stay alive indefinitely in an environment with endless pair of pipes with random heights. The fitness calculation used ensured that the objective of an agent in a reduced environment represented its goal in a real run of the game.

Because the obstacles used have gaps near the environment borders and an intermediate one, when the agent manages to maximize its result in these three cases it then masters all possible variations within these limits. Considering that NEAT always searches for the simplest solution to a problem and that the fitness presented together with the division into three scenarios help the NEAT find a perceptron network architecture using a single species, this solution is the simplest. The techniques discussed in this work helped the algorithm to find this solution in a short time, thus proving its effectiveness.

Using a population of only 30 individuals, the evolutionary algorithm was able to converge to an optimal behavior after about twenty generations. At this point, an agent is able to play the game indefinitely. This shows that this strategy can find optimal solutions in a short number of generations. As future works, this minimal training strategy can be tested in simple platform games that show some kind of action repetition through their stages, and also with different types of learning algorithms.

## Acknowledgments

The authors would like to thank the Brazilian National Council for Scientific and Technological Development (CNPq), the Instituto Atlântico, the Teleinformatics Engineering Department (DETI), and the Department of Computing (DC), both at the Federal University of Ceará (UFC), for their support.

## References

[1] R. Chuchro, "Game playing with deep q-learning using openai gym," *Semantic Scholar*, 2017.

[2] M. Ebeling-Rump and Z. Hervieux-Moore, "Applying q-learning to flappy bird," *Semantic Scholar*, 2016.

[3] W. Guofang, F. Zhou, L. Ping, and L. Bo, "Shaping in reinforcement learning via knowledge transferred from human-demonstrations," in *2015 34th Chinese Control Conference (CCC)*, July 2015, pp. 3033–3038.

[4] E. J. Hastings, R. K. Guha, and K. O. Stanley, "Automatic content generation in the galactic arms race video game," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 4, pp. 245–263, Dec 2009.

[5] M. Henaff, J. J. Zhao, and Y. LeCun, "Prediction under uncertainty with error-encoding networks," *CoRR*, vol. abs/1711.04994, 2017. [Online]. Available: http://arxiv.org/abs/1711.04994

[6] T. Kristo and N. U. Maulidevi, "Deduction of fighting game countermeasures using neuroevolution of augmenting topologies," in *2016 International Conference on Data and Software Engineering (ICoDSE)*, Oct 2016, pp. 1–6.

[7] A. McIntyre, M. Kallada, C. G. Miguel, and C. F. da Silva, "neat-python," https://github.com/CodeReclaimers/neat-python.

[8] C. Rosset, C. Cevallos, and I. Mukherjee, "Cooperative multi-agent reinforcement learning for flappy bird *," *Semantic Scholar*, 2016.

[9] A. Sahin, E. Atici, and T. Kumbasar, "Type-2 fuzzified flappy bird control system," in *2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, July 2016, pp. 1578–1583.

[10] S. Samothrakis, D. Perez-Liebana, S. M. Lucas, and M. Fasli, "Neuroevolution for General Video Game Playing," *2015 IEEE Conference on Computational Intelligence and Games, CIG 2015 - Proceedings*, pp. 200–207, 2015.

[11] J. Schrum and R. Miikkulainen, "Discovering multimodal behavior in ms. pac-man through evolution of modular neural networks," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 1, pp. 67–81, March 2016.

[12] J. Schrum and R. Miikkulainen, "Constructing complex npc behavior via multi-objective neuroevolution," in *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'08. AAAI Press, 2008, pp. 108–113. [Online]. Available: http://dl.acm.org/citation.cfm?id=3022539.3022558

[13] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Real-time neuroevolution in the NERO video game," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 653–668, 2005.

[14] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002. [Online]. Available: http://nn.cs.utexas.edu/?stanley:ec02

[15] K. O. Stanley and R. Miikkulainen, "Evolving a roving eye for go," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004)*. Berlin: Springer Verlag, 2004.

[16] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[17] B. Takács, J. Števek, R. Valo, and M. Kvasnica, "Python code generation for explicit mpc in mpt," in *2016 European Control Conference (ECC)*, June 2016, pp. 1328–1333.

[18] N. Tasfi, "Pygame learning environment," https://github.com/ntasfi/PyGame-Learning-Environment, 2016.

[19] H. Zhang, L. Duan, X. Zeng, and X. Y. Tang, "Browser/server based experimental environment for reinforcement learning," *2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, pp. 571–575, 2018.