AI Game Agents Based on Evolutionary Search and (Deep) Reinforcement Learning: A Practical Analysis with Flappy Bird

Leonardo Thurler, José Montes, Rodrigo Veloso, Aline Paes, Esteban Clua

AI Game Agents based on Evolutionary Search and (Deep) Reinforcement Learning: a Practical Analysis with Flappy Bird

Leonardo Thurler, José Montes, Rodrigo Veloso, Aline Paes, and Esteban Clua

Universidade Federal Fluminense, Niterói, Rio de Janeiro, Brazil
{lpthurler, joselucasbrandaomontes, rodrigoveloso}@id.uff.br
{alinepaes, esteban}@ic.uff.br

Abstract. Game agents are efficiently implemented through different AI techniques, such as neural network, reinforcement learning, and evolutionary search. Although there are many works for each approach, we present a critical analysis and comparison between them, suggesting a common benchmark and parameter configurations. The evolutionary strategy implements the NeuroEvolution of Augmenting Topologies algorithm, while the reinforcement learning agent leverages Q-Learning and Proximal Policy Optimization. We formulate and empirically compare this set of solutions using the Flappy Bird game as a test scenario. We also compare different representations of state and reward functions for each method. All methods were able to generate agents that can play the game, where the NEAT algorithm had the best results, reaching the goal of never losing.

Keywords: Artificial Intelligence · Reinforcement Learning · Deep Reinforcement Learning · Genetic algorithm · Q-Learning · NEAT · PPO · Ml-Agents · Flappy Bird · AI Game Agents · Game · Unity · Pygame

1 Introduction

The AI field started to be applied to games through agents that could play against humans or to make game characters more convincing [1–4]. All the solutions proposed towards those goals led to a state where the literature present numerous solutions for the same problem. The existence of different solutions is not an issue by itself but it makes it harder and more complex to understand which ones are more appropriate to handle a given problem [5].

Motivated by this issue, this work presents a methodology to create and evaluate the performance of an AI game agent based on distinct AI techniques. For this, we use three different and widely used AI techniques used for solving gameplay problems. The agents were developed using NEAT as the search algorithm, which is based on the genetic algorithm with neural networks. We also address the Q-Learning reinforcement learning algorithm and the Proximity Policy Optimization (PPO) with a deep reinforcement learning algorithm.

Our developments are based on the Flappy Bird game as a test case. The player's goal in this game is to cross between pipes that appear in random positions on the stage as much as possible. This randomness feature makes this

game a promising test environment for training agents who need to learn behaviors based on non-deterministic phenomena. For the development of the learning environment, two digital game engines were used: Unity [6] and Pygame [7]. Previous work investigated the performance of Flappy Bird agents based on genetic algorithms [8] and reinforcement learning [9]. While their focus is mostly restricted to analyse agents based on a single AI technique, here we take a step towards comparing these techniques plus deep reinforcement learning, showing a proper strategy for analysing which technique did better.

As a result, this work presents strategies to formulate, train and evaluate agents based on different AI methods. The results demonstrate that, when using a proper model, these techniques can create agents capable to play Flappy Bird and obtain high scores. It is also demonstrated how to analyze and compare results obtained by varying hyperparameters and modifying agents representations. Finally, by comparing the results of each agent, we present how to create metrics that allows the evaluation and comparison of agents that use different techniques, making it possible to define which is the best for a specific problem and situation.

2 Related Work

Previous works had implemented and compared Deep Learning, Reinforcement Learning and evolutionary algorithms as strategies for AI based player training. In [10] authors conclude that the Deep Q-learning was the best, beating all the others methods that were tested and even human expert performance in some of them. [11] reproduced existing research on deep reinforcement learning algorithms applied to games and compared the obtained results with the published ones with Breakout, an Atari 2600 game. In their work, the Asynchronous Advantage actor-critic algorithm proved to be much better than Deep Q-Learning. These studies reinforce the need for novels ways to analyze and compare the performance of different AI methods and demonstrate how this is not a trivial task for all the games. Thus, our work contributes to such studies by presenting a methodology to create and evaluate the performance of agents based on different AI techniques.

In [8], authors developed and studied an AI agent using a combination of neural networks and genetic algorithm to play Flappy Bird. They divided the gameplay into two difficulty levels and the trained agent could reach scores above 150. [9] applied various reinforcement learning algorithms to train an agent to play Flappy Bird, such as SARSA, Q-Learning, Q-value approximation via linear regression, and approximation via a neural network. From the experiments, they conclude that Q-learning had better performance with regular scores above 1400. The available literature shows that the Flappy Bird game presents some interesting challenges to AI game agents and can be used as a test bed scenario for this research topic. Although they demonstrate that these techniques are capable of training agents who can perform well in the game, as far as we know, there are no literature that compare and analyse which techniques are the most

suitable ones and which advantages and disadvantages they present. Our work makes a contribution in this direction by comparing the performance of these techniques in conjunction with deep reinforcement learning and demonstrates which one did better.

3 Theoretical Background

3.1 Genetic Algorithms

A genetic algorithm (GA) constitutes a mathematical model simulating the theory of Darwinian Evolution. It consists of searching on a set of possible solutions to a given problem, called the population, initially created with random patterns [12, 13]. In this work, we pose the task of learning the weights *and* the architecture of an ANN as a GA task. To achieve that, each individual of the GA is composed of a different ANN, including architectures and their weights. Then, the NeuroEvolution Algorithm for Increasing Topologies (NEAT) [14] searches for the best solution for the problem. The search process implemented by the NEAT algorithm involves the use of the classical genetic operators Selection, Crossover and Mutation, operations that manipulate individuals throughout the generations by adding neurons and hidden layers, connections and adjusting activation functions of the added neurons [14].

3.2 Reinforcement Learning and Deep Reinforcement Learning

Reinforcement learning (RL) works by building agents that interact with the environment. The environment provides a response about the efficiency of agent strategy through a reward function [13]. The agent must experience a lot of different strategies in order to find which is the best to achieve a goal. When the agent is trying different strategies it is actually searching over the space of all possible inputs and outputs (state/action) in order to earn a reward.

One of the most classical approaches for learning policies is the Q-Learning algorithm [4]. It tries to learn an action-value function iteratively. The idea is to create a table with values of Q for all possible state action pairs and use experiences to update them. When one needs to directly deal with large spaces of states and actions, or discretization or clustering is not an option, other strategies must be used. Typically, in these scenarios, one may approximate a function with different methods, such as deep neural network as the function approximator, which in this case corresponds to a Deep Reinforcement Learning (DRL) [15, 16]. Recently, DRL has achieved the state-of-the-art in several game related problems, such as AlphaGO [2]. An approach, based on DRL, that has achieved impressive results is the Proximal Policy Optimization (PPO) algorithm [17]. PPO is based on an architecture called Actor-Critic. This architecture uses two neural networks: the first is called critical and is used to assess the current state of the environment by generating an estimate of the Q value; The second serves to define the best action for the current state, which is called an actor [18].

4 Problem Description and Modeling

4.1 Modeling Flappy Bird as an AI agent

Flappy Bird [1] is a single player game that gives a point every time the player-controlled bird passes between an upper and a lower pipe. There are infinity pairs of pipes alongside the game environment and it the finishes when the bird hit the ground or any of the pipes.

The player has only two possible actions: jump, when the bird will have its vertical position (y) shifted upwards, and do nothing, where the bird will move downwards due to the gravity action. The horizontal position of the bird (x) is always the same, the horizontal position of the pipes (x_c) are modified considering a movement with constant speed. The vertical position of the end of the lower pipe (y_{ci}) and the upper pipe (y_{cs}) are defined at random, always maintaining a constant spacing between the them. When the bird passes through a pair of pipes, two new pipes are generated. Figure 1 illustrates the main elements of the game.

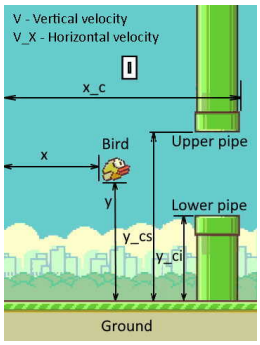

Fig. 1. Flappy Bird game illustration.

Based on the primary version aforementioned described, two versions of the game were implemented for this paper: one using Unity [6], and based on NEAT and PPO, and the other using Pygame [7], based on Q-Leaning.

Environment Modeling We model a state by defining the positions of the bird and pipes. To model the bird, we rely on a component of vertical velocity v and its position as y. The position of the pipes rely on a horizontal velocity component v_x (constant), and positions y_{cs}, y_{ci} (vertical) and x_c (horizontal). From those variables, the state is first defined as a 4-position vector $s = (y, y_{cs}, y_{ci}, x_c)$.

We also considered other variables that may help the agent to achieve its goal. Variables such as the vertical (d_y^{ci} and d_y^{cs}) and horizontal (d_x) distance between the bird and any of the pipes are also used. Furthermore, we add two binary variables, α and β to represent whether the bird is above or below one of the pipes. The variable α is equal to 0 if the bird is below the bottom pipe and

[1] http://flappybird.io/

1 otherwise and the variable β is equal to 0 if the bird is below the top pipe and 1 otherwise. The representations are summarized in Table 1.

Table 1. State Representation Parameters.

State	s_1	s_2	s_3	s_4
Vector of observable variables	$(d_y^{ci}, d_y^{cs}, d_x, v)$	(y, y_{cs}, y_{ci})	$(y - y_{ci}, v)$	$(d_y^{ci}, v, \alpha, \beta)$

Adjustments to run Q-Learning The variables used to define the state of the game are not discrete, as required by Q-Learning. Therefore, to use this method in this context, it is necessary to discretize the variables. The discretization strategy followed here is given by (1) where var_d is the variable after discretization, var is the original variable, N is an integer parameter that controls the reduction of the state space. Thus, if $N = 5$, any real number y between 0 and 4 will be represented by a discrete variable $y_d = 0$, considerably reducing the state space even when y is an integer variable. Reducing the space can increase the generalization capacity and reduce the Q-Learning training time because the larger the space the greater the probability of existing states that have not yet been visited by the agent.

$$var_d = \text{int}(var/N) \tag{1}$$

Transition Model and Goal The agent performs two possible actions: jump and do nothing. From these actions it is possible to define a transition model for the problem, as follows in (2) and (3). Where p is the displacement caused by the jump and g_t is the displacement caused by gravity. There are two other possible transitions that only occur when $x_c = x$, $y < y_{cs}$ and $y > y_{ci}$, that is, the moment the bird crosses the pipes, as shown in (4) and (5). Where y'_{cs} and y'_{ci} are the new pipe positions, randomly defined and x_{max} is the maximum horizontal position.

$$\text{r}((y, y_{cs}, y_{ci}, x_c), \text{jump}) = (y + p, y_{cs}, y_{ci}, x_c + v_x) \tag{2}$$

$$\text{r}((y, y_{cs}, y_{ci}, x_c), \text{do nothing}) = (y - g_t, y_{cs}, y_{ci}, x_c + v_x) \tag{3}$$

$$\text{r}((y, y_{cs}, y_{ci}, x_c), \text{jump}) = (y + p, y'_{cs}, y'_{ci}, x_{max}) \tag{4}$$

$$\text{r}((y, y_{cs}, y_{ci}, x_c), \text{do nothing}) = (y - g_t, y'_{cs}, y'_{ci}, x_{max}) \tag{5}$$

4.2 Reward function

The definition of a suitable reward function is crucial to ensure that the agent learns the desired function and it is one of the most complex phases of a RL process. Thus, we design and experiment different reward policies for the Flappy Bird to experimentally select the most suitable one. The simplest and most explicit reward function would be as follows in (6).

The problem with this function is that the reward will only be achieved if a very specific set of actions is chosen since at the beginning of the training

there is no prior information about the problem. Thus, this simple reward would require a great deal of training time and no guarantee of any learning. It is then possible to formulate more explicit functions, such those described in (7), where a and b are constant real numbers. The function r_t^2 was used in the preliminary tests and it makes the agent learning. Thus, even more explicit functions were elaborated to allow the analysis of which function would be the best to solve this problem. We defined the functions as r_t^3, r_t^4 and r_t^5 as detailed in (8), (9) and (10) where d is the vertical distance between the bird and the center of the tubes and δ is a real number that defines how close to the center the bird must be from the center to earn the reward; in r_t^3 the center refers to the center of the tubes.

$$r_t^1 = \begin{cases} 1, \text{ if made a point} \\ 0, \text{ if did not make a point} \end{cases} \tag{6}$$

$$r_t^2 = \begin{cases} a, \text{ if the bird is alive} \\ -b, \text{ if collided} \end{cases} \tag{7}$$

$$r_t^3 = \begin{cases} a, \text{ if the bird is alive and between the pipes} \\ -b, \text{ if collided} \end{cases} \tag{8}$$

$$r_t^4 = \begin{cases} a, \text{ is alive and between the pipes and } d < \delta \\ -b, \text{ if collided} \end{cases} \tag{9}$$

$$r_t^5 = \begin{cases} a, \text{ jumped when below the center} \\ a, \text{ didn't jump being above the center} \\ -b, \text{ if collided} \end{cases} \tag{10}$$

5 Experimental Methodology

Training a Flappy Bird agent consists of making it play the game several times until the total training time exceeds a limit (here established as 30 minutes) or until the average agent score is over a defined value (here, it is fixed as 100).

In this work, the main metrics used to compare agents and evaluate their performance are the accumulated reward, the average score, the maximum score and the probability of crossing two pipes. First, we rely on their accumulative reward, either when they have the same reward function or by converting their different functions. Furthermore, as a Flappy Bird objective is to go through the largest number of pipes as possible to obtain the highest possible score, it is natural to evaluate the performance of an agent through the achieved score. Finally, the performance of an agent can be assessed through its consistency in carrying out a predetermined task.

5.1 NEAT

NEAT uses the Fitness function of each individual for its evaluation. In this paper the function follows the score obtained by the agent during the game. The score for each agent is calculated when it touches any of the pipes or the ground, which does not happen with a properly trained agent. When any individual of the generation reaches this limit, it is considered converged and GA evaluates the next generation, up to the limit of 100 generations.

During the initial tests, several configurations of different neural network topologies and hyperparameters were verified, surpassing the simplest ones and those that used the ReLU function and topologies without hidden layer, using the configurations present in [19], with the adjustments shown in the Table 2.

Table 2. NEAT Settings

Parameter	Setting 1	Setting 2	Setting 3	Setting 4
weight_mutate_power	10	1	0.5	0.1

5.2 Q-Learning

The value of the parameters considered for running the game agents with Q-Learning is detailed in Table 3. The policy chosen in the training was ϵ-greedy, where the parameter ϵ is defined as a function of the number of iterations (i), if $i < 1000$ then $\epsilon = \epsilon_0$, if $i > 1000$ ϵ is decremented by 0.01 whenever the number of iterations is a multiple of 500. This function guarantees more exploration at the beginning of the training and more exploitation at the end.

Table 3. Default parameters of Q Learning.

Parameter	γ	μ	ϵ_0	N
Value	0.9	0.7	0.5	5

After preliminary tests it was possible to notice that the standard training strategy is inefficient. The problem is that the game time increases with the agent's performance and consequently increases the training time. The game time of an agent that scores 100 points is almost 100 times greater than the time of a game where the agent scores 1. These difficulties led to the development of a new training methodology. Instead of letting the agent play freely until reaching 100 points or die, the game ends when the agent gets 2 points. Since there is no distinction between the representation of a low-scoring state to a high-scoring state and the pipes are generated randomly, an agent who consistently can pass the first two pipes will also be able to pass the others with high probability. Applying this process, considering state representation s_1 and reward function r_4, an average score of 42.5 have been achieved and the best score was 279 points for an agent trained for 1 hour. Note that the average of 42.5 is much higher than the asymptotic performance obtained by the previous methodology.

After changing the training methodology, we find the best reward function and the best representation of states. In order to do so, the state was fixed and

different reward functions were evaluated, so that the reward function r_4 was the one that got the best result. To find the best state representation, we fixed the reward function r_4, testing different representations of state in order to analyse the result of each representation. The best representation achieved was s_1 as shown in Figure 2.

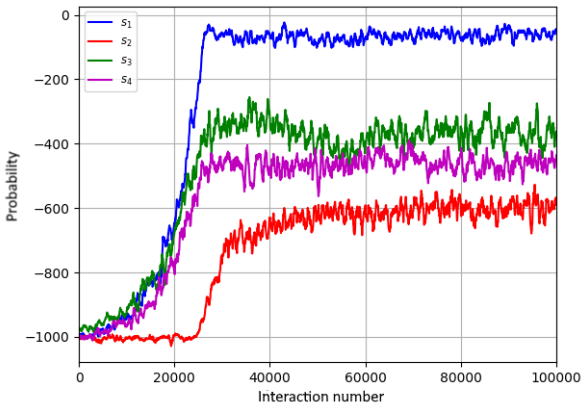

Fig. 2. Accumulated reward as a function of the number of iterations considering different state representations.

5.3 PPO

PPO allows multiple agents to train simultaneously in order to streamline the training process. The PPO training process resulted in the reproduction of the same problems founded in the Q-Learning standard training strategy, which took much longer time as the agent become better at passing through the pipes. Thus, the PPO training strategy was also adapted to the training strategy where the game ends when the agent obtains 2 points and the configuration of 30 agents in the training environment was maintained.

As with Q-Leaning, preliminary tests were carried out, using base values for the hyperparameters and the new training strategy in order to identify an optimal configuration of state representation and reward function to be used in the PPO learning algorithm. In [20], it is possible to find a detailed description and the base values of each hyperparameter.

Analyzing the preliminary tests results, *num_layers*, *hidden_units* and *beta* were selected. The choice of the parameters *num_layers* e *hidden_units* aimed to analyze different ANNs' architectures. The parameter *beta* allows the algorithm to fully explore the states, doing more tests before decreasing the entropy, resulting in a better generalization at the end of the training.

With the chosen parameters, we start a new test session in order to find an optimal configuration considering changes in these 3 hyperparameter. Each test run uses the r_t^5 reward function and goes through one million interactions. Table 4 shows the configurations used to train the agents to select the best

values for hyperparameters. The better representation was $Env1$, as it managed to quickly train the agent while maintaining a good generalization of the agent when compared to the others.

Table 4. PPO's Environments Hyperparamets Settings

Parameter	**Env 1**	**Env 2**	**Env 3**	**Env 4**
num_layers	2	1	2	1
hidden_units	64	32	64	32
beta	5.0e-3	5.0e-3	4e-1	4e-1

6 Results and Discussions

6.1 NEAT

Experiments were performed with different configurations for the initial population, increasing the amount of neurons to 10, 50 and 100 in the hidden layer, considering that no individual was able to reach the 100-point mark. The other varied NEAT parameter is the mutation rate, which influences the weight, inclusion, and removal of the neural network edges, as well as the inclusion and removal of neurons. The randomness coming from different values of the mutation rate, may be the responsible for the random behavior of individuals over generations. Figure 3 illustrates a partially random behavior from the scoring of the best individuals of each generation, in addition to the lack of consistency between the score of one generation and the score of the next generation. Thus, after experimenting mutation rates of 5% and 10%, we noticed that the behavior of the agents in the game was no longer at random and they all began to perform the same actions making the search very repetitive, with no evolution over the generations. Observing the average score for each one, it is possible to state that there is a large disparity between individuals within the same generation, since the maximum average score was 2 points, even in generations with individuals who achieved the maximum of 100 points.

6.2 Q-Learning

After defining the default parameters, trained the network and defined the reward function and state representation, we carried out tests for identifying the optimal set of parameters and understand how they impact on the learning process. In these tests, only the investigated parameter can change while the others are fixed to their default values.

The ϵ_0 parameter had a great impact on the agent's performance, both in the consistency to perform the task and in the accumulated reward. For ϵ_0 varying from 0.3 to 0.6 it is possible to observe a decrease in initial learning speed but the asymptotic performance remains the same. However, for $\epsilon_0 < 0.3$, despite maintaining the highest initial learning speed, the asymptotic performance is inversely proportional to ϵ_0. This result highlights the need for better initial exploration in the hyperparameters space to learn a more efficient solution.

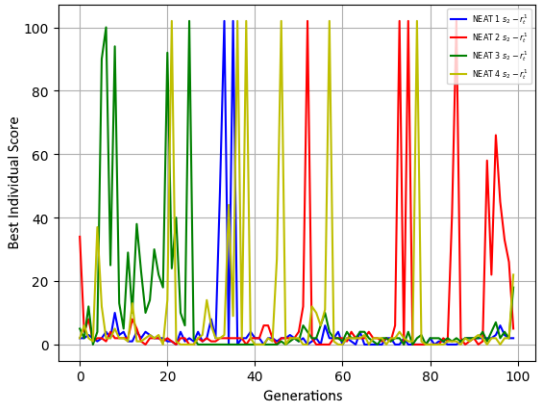

Fig. 3. Score of the best individual of each generation.

The parameters γ and μ have also been tested, but they did not present a great influence on performance. The discretization control parameter N had a notable impact on asymptotic performance and the best performance occurs for $N = 6$. Small values of N decreases the agent's generalization capabilities, while large N can oversimplify the state space and important information can be lost.

Table 5 shows the optimal hyperparameters according to the experiments. With those values and a greedy policy, the trained agent obtained an average score of 49.18 and the best score of 290 points considering 30 minutes of play.

Table 5. Optimal parameters found for Q Learning.

Parameter	γ	μ	ϵ_0	N
Value	0.9	0.7	0.5	6

6.3 PPO

Once we found the optimal settings for the hyperparameters, a final analysis step was started where each execution takes two million interactions. This step consisted of using the mentioned state representations s_1, s_2, s_3 and analyzing the performance of each of them when combining with the reward functions: r_t^1, r_t^4 and r_t^5. Figure 4 shows the combination that achieved the best result was s_3 with the reward r_t^5 as it was able to faster train the agent and still keep a good generalization of when compared to the others states and reward functions.

At the end of each training process, Unity's PPO algorithm generates a file that contains the model achieved during the training. To evaluate how the new agent performs, the file generated by the state representation s_3 was used with a reward r_t^5, since those were the configurations with the best results. When using this model in the game without any scoring restriction, after 30 minutes of play session we obtained an average score of 1.5 and a maximum of 10. This result is somewhat curious since as the agents managed to do well during training, it was expected that the generated model would do better.

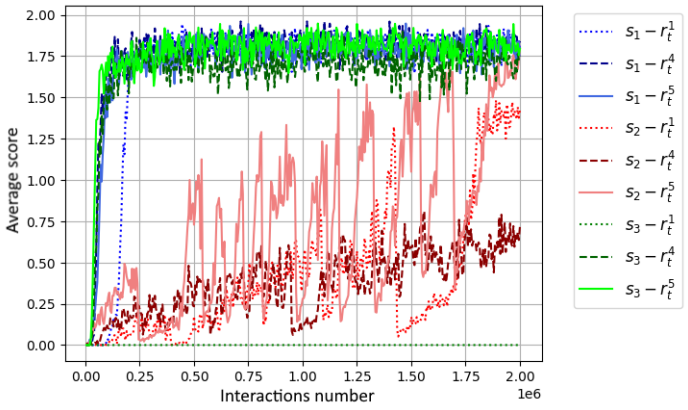

Fig. 4. Average score based on the number of interactions. Each curve represents a different state representation s_n, using the r_t^y reward functions.

7 Conclusion

In this paper, the Flappy Bird game was formulated as an AI problem including an AI-based state representation and a transition model between a pair of actions. The goal of the formulation is to train a Flappy Bird AI-agent that never loses or at least consistently perform well. We approach the problem using a genetic algorithm, reinforcement learning and deep reinforcement learning perspectives. To achieve that, reward and fitness functions were also developed to guide the agent to find a good solution for the problem.

We evaluated the NEAT algorithm in order to search for an effective configuration of a neural network, a Q-learning algorithm built upon discretized representations of states and actions and the PPO approach. The NEAT algorithm was the most effective to solve the Flappy Bird as its agent never loses when used on a play session, although we have observed some noisy issues during its operations due to the mutation rate. The PPO agent performed well during training, but when used on a play session it was not so effective as the training sessions. The Q-Learning agent performance was so consistent on training as on play sessions. All agents were able to play, but some were better than others, as shown in Table 6.

Table 6. Agent max score on 30 minutes of play session after training.

Method	NEAT	Q-Learning	PPO
Max Score	Never lose	290	10

As future work, we intend to apply these algorithms and configurations in other games that have different characteristics and challenges. We also intend to create agents that combine these techniques to train a single agent and evaluate the result of this combination.

References

1. G. N. Yannakakis and J. Togelius, *Artificial intelligence and games*. Springer, 2018, vol. 2.
2. D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, 2016.
3. O. Vinyals *et al.*, "AlphaStar: Mastering the Real-Time Strategy Game StarCraft II," https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.
4. M. A. Samsuden, N. M. Diah, and N. A. Rahman, "A review paper on implementing reinforcement learning technique in optimising games performance," in *2019 IEEE 9th International Conference on System Engineering and Technology (ICSET)*. IEEE, 2019, pp. 258–263.
5. M. Injadat, A. Moubayed, A. B. Nassif, and A. Shami, "Machine learning towards intelligent systems: applications, challenges, and opportunities," *Artificial Intelligence Review*, pp. 1–50, 2021.
6. Unity, Available: https://unity.com/ Accessed: 2021-06-19.
7. Pygame, Available: https://www.pygame.org/ Accessed: 2021-06-19.
8. Y. Mishra, V. Kumawat, and K. Selvakumar, "Performance analysis of flappy bird playing agent using neural network and genetic algorithm," in *International Conference on Information, Communication and Computing Technology*. Springer, 2019, pp. 253–265.
9. T. Vu and L. Tran, "Flapai bird: Training an agent to play flappy bird using reinforcement learning techniques," *arXiv preprint arXiv:2003.09579*, 2020.
10. I. Hosu and A. Urzica, "Comparative analysis of existing architectures for general game agents," in *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2015, pp. 257–260.
11. A. Jeerige, D. Bein, and A. Verma, "Comparison of deep reinforcement learning approaches for intelligent game playing," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, 2019, pp. 0366–0371.
12. S. Mirjalili, "Genetic algorithm," in *Evolutionary algorithms and neural networks*. Springer, 2019, pp. 43–55.
13. S. Marsland, *Machine Learning - An Algorithmic Perspective.*, ser. Chapman and Hall / CRC machine learning and pattern recognition series. CRC Press, 2009.
14. K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.
15. M. Lanham, *Learn Unity ML-Agents – Fundamentals of Unity Machine Learning: Incorporate new powerful ML algorithms such as Deep Reinforcement Learning for games*. Packt Publishing, 2018.
16. I. França, "Learning how to play bomberman with deep reinforcement and imitation learning," *Springer International Publishing*, pp. 121–133, 2019.
17. J. Schulman *et al.*, "Proximal policy optimization algorithms," 2017.
18. M. Pecenin, A. Maidl, and D. Weingaertner, "Optimization of halide image processing schedules with reinforcement learning," in *Anais do XX Simpósio em Sistemas Computacionais de Alto Desempenho*. Porto Alegre, RS, Brasil: SBC, 2019, pp. 37–48.
19. A. McIntyre *et al.*, "neat-python," https://github.com/CodeReclaimers/neat-python.
20. "Unity ML-Agents PPO hyperparameters configurations," https://github.com/Unity-Technologies/ml-agents/blob/release_15_docs/docs/Training-Configuration-File.md, accessed: 2021-06-19.