Neuroevolution for Flappy Bird Game

Bc. Alexandra Smolová

Slovak University of Technology in Bratislava,
Faculty of Informatics and Information Technologies

Abstract. In this paper, we focused on the process of neuroevolution of a neural network with a simple genetic algorithm. In individual experiments, we tried to evolve the neural network and a neural network with Q-learning to learn how to play a very popular mobile game - Flappy Bird.

Keywords: Machine Learning · Neural Network · Genetic Algorithm · Reinforcement Learning · Q-learning.

1 Introduction

The field of neural networks and reinforcement learning has become a very popular topic of the last decade due to its wide applicability. With its ability to learn and search for complex patterns in data, it finds use in the field of computer vision and knowledge discovery. A system starting as completely random will learn to solve the given task by gradually improving its weights based on gained memories of state, action and reward values. The problem of reinforcement learning (RL) remains the time-consuming and often inefficient learning process, which we can further optimize by using evolutionary algorithms in a process called neuroevolution.

In our work, we used a neural network in combination with a genetic algorithm to speed up the agent's learning process to play a mobile game - Flappy Bird, which consists of a precisely timed "flap" of a bird and subsequent free fall without touching the obstacles. The task of the neural network is to time this flap correctly and thus achieve a high score in the game.

2 Related work

In this section, we will discuss two articles that focus on the use of machine learning in the game Flappy Bird:

- Neuroevolution - Machine Learning for Flappy Bird using Neural Network and Genetic Algorithm
- Q-learning - Use reinforcement learning to train a flappy bird NEVER to die

2.1 Machine Learning for Flappy Bird using Neural Network and Genetic Algorithm

In search of related work, we found a project that used neuroevolution in Flappy Bird, programmed in the HTML5 markup language using the Phaser framework and the Synaptic Neural Network library.

The aim of the project was to create a population of ten agents, whose neural network is developed through a genetic algorithm based on the natural selection of the best individuals.

The Synaptic Neural Network library was used to create the neural network. The structure of the neural network consisted of two inputs, one output and one hidden layer with six neurons, see Fig. 1.

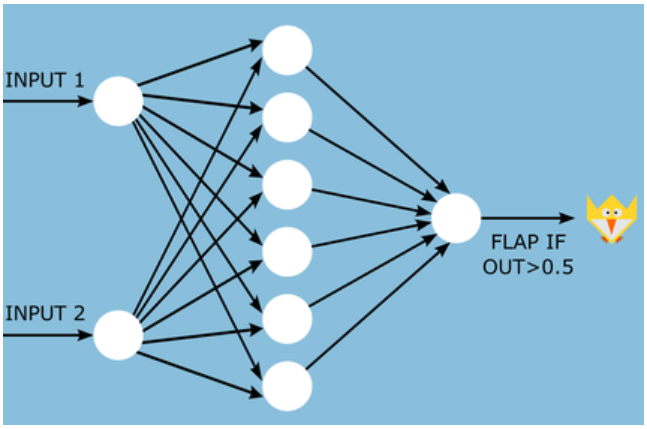

Fig. 1: The structure of neural network

The input layer with 2 neurons represents what a bird sees - the horizontal and vertical distance to the nearest gap. Output above 0.5 determines the execution of the flap.

In the genetic algorithm, they used a fitness function that rates agents by success, defining this function as the distance traveled penalized by the distance to the nearest gap to distinguish the success of agents who traveled the same distance.

When creating a new population, they used this replacement strategy [1]:

- select the top 4 units (winners) and pass them directly on to the next population,
- create 1 offspring as a crossover product of two best winners,
- create 3 offsprings as a crossover products of two random winners,
- create 2 offsprings as a direct copy of two random winners,
- apply random mutations on each offspring to add some variations.

In the 43rd generation, the main winner was created, who learned to play the Flappy Bird game with a score bigger than 450.

2.2 Use reinforcement learning to train a flappy bird NEVER to die

This article focused on troubleshooting and performance tuning of bird's neural network.

The input parameters for the agent were again horizontal and vertical distance to the gap, but also the velocity of the bird in the vertical direction. Later, a parameter was added indicating the vertical difference between the current and the next obstacle so that the agent would be one step ahead in the decision making process.

An interesting fact was the change in the reward for not dying from 1 to 0. "It forces the bird to focus on the long term alive, to keep away from any action causing death. It will get penalty of -1000 rewards on death no matter how many successful sessions the bird ran in the past. [2]"

This particular work trained neural network for more than ten hours. Tuned neural network using Q-learning for the game Flappy Bird gives excellent stable results, but its training takes too much time. Therefore, we would like to know whether it is possible and more efficient to evolve neural network with Q-learning using genetic algorithm.

3 Used algorithm

3.1 Neural network with Q-learning

First, we created a bird's brain - neural network with Q-learning, which learned to flap through obstacles based on reward and punishment. After each run, we retrained the neural network based on actions and rewards in individual states.

To adjust the Q values representing the quality of the action in the state, we used Bellman's equation, which says that the maximum future reward for this state and the action is the immediate reward r plus the maximum future reward for the next state. We will reduce the maximum future reward for the next state by the discount value y.

$$Q(s,a) = r + y * max(Q(s',a')) \tag{1}$$

1.1: Bellman equation

We chose neural network with Q-learning - reinforcement learning, because we wanted to achieve independent learning without the need for a training dataset. We did not want to explicitly show the neural network what to do in which situation, but to let the neural network react naturally to both positive and negative rewards. From previous experience with neural networks, we

have encountered the problem of falling into the local optimum, which solves the problem, but does not provide the best possible solution compared to the global optimum. For this reason, we used a value representing the rate of exploration, so that if the neural network falls into the local optimum, it can come out of it by randomly choosing the future action, and thus examining most of the state space.

3.2 Neuroevolution

After successful implementation the neural network, we found that despite the visible progress of the bird, the neural network trains for a very long time. In one thousand experiments, the bird achieved a very low score (around 40-50). Discovering the state space as well as updating the weights after each run was not as effective. Therefore, we decided to try the process of neuroevolution.

Neuroevolution rests in the random initialization of a neural network and its evolution. To optimize learning time and the evolution of the neural network, we chose a simple genetic algorithm that creates each population of x agents (birds), each with its own brain - a neural network. These agents go through obstacles and after their failure they are evaluated by the fitness function. The best agents are then chosen to create the next generation.

Through the natural process of selection, we can achieve a high score, and thus optimize the neural network in a relatively short time.

4 Experimets

Fig. 2: Design of the game

4.1 Environment and programming language

For this project we decided to use the Unity 2019.2.6f1 editor and the C# programming language. The game mechanics of the Flappy Bird game itself, as well as the game assets, see fig. 2, are taken from the tutorial on site learn.unity.com. Of course, these scripts have been extended and further adapted to the needs of the current version of the project.

4.2 Neural network with Q-learning

We use three main scripts to create a neural network from scratch - Neuron.cs, Layer.cs and ANN.cs. Therefore, we do not use any additional libraries. A neural network of the bird is created in the Brain.cs script, and after the agent dies, it is retrained by updating weights based on bird's memories.

When creating the neural network, we played with several input parameters as well as the structure of the neural network itself. The fastest learning progress was observed with the following parameters:

- one input layer with two neurons,
- one output layer with two neurons,
- one hidden layer with six neurons,
- alfa value = 0.9.

The structure of our neural network, see fig. 3 is similar to the one mentioned in the section 2.1. The difference is number of neurons in the output layer. Because we use Q-learning, we select an action based on the maximum quality value of the action in a given state. Since we have two actions (flap or not), we use two outputs.

We use the same inputs as in the article from the section 2.1:

- the horizontal (x) distance to the nearest gap,
- the vertical (y) distance to the nearest gap.

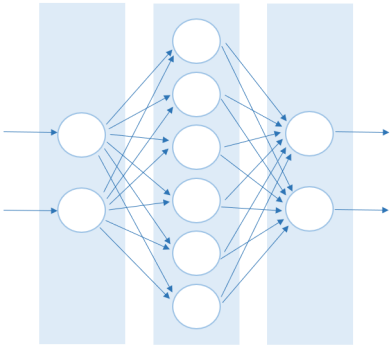

Fig. 3: Structure of our neural network

We also tried to experiment with other input parameters, like the ones used in the article from the section 2.2, which significantly extended the training time. For simplicity, we continued to follow with only two inputs.

One hidden layer with six neurons gave the best results in training runs. We tried to reduce the number of neurons as well as increase them. We also tried to extend the neural network by another layer to create the so-called deep neural network using Q-learning - *DQN*. However, the best results were obtained by the simple neural network structure chosen above.

An important part of reinforcement learning is the way in which the agent is punished or rewarded. In our work, we did not choose as high a penalty for death as in a article 2.2, because it prevented the learning of agents and led to unstable training. Unity in its ML-Agent extension recommends using values in the range $< -1.1 >$. Although we didn't use this extension, we've decided to follow the recommendations. We penalize the death of the agent with a value of -1 and we reward not dying with a value of 0.1. This simple reward system teaches the agent to live as long as possible and of course to avoid death.

None of the articles mentioned whether the data had been pre-processed in any way before being fed to the neural network. However, in our work we normalized the values of the horizontal distance (interval $< 0, 1 >$) and mapped the values of the vertical distance to the interval $< -1, 1 >$, as the vertical distance from the center of the gap could also become negative (in the case that the bird is above the center) and positive (when the bird is below the center of the gap). We further rounded these values to two decimal places. Without rounding, the neural network did not have enough generalized data to train. The last adjustment was to subtract the rounded normalized value of the horizontal distance from 1. This adjustment was made because the closer we are to the obstacle, the more we want the neuron to fire the signal as well as properly update the weights in the learning process. The natural distance from the obstacle, if the bird is right next to it, approaches zero. If we multiply the zero input to the neuron by the corresponding weight, nothing changes and the signal transmission only depends on the bias value. We modified this value by subtracting it from 1, so when the bird is closest to the gap, value will approach one and neuron will most likely fire.

In our project we use two types of activation functions - TanH (since we work with values from -1 to 1) and Sigmoid for the output layer.

4.3 Genetic algorithm

The entire genetic algorithm is implemented in the PopulationManager.cs script. Script creates a primary population - of one hundred agents with the same neural network structure and initializes it to random values. After the extinction of the generation, it selects twenty percent of the best individuals, based on the value of their fitness function. Fitness function is calculated from the time of death combined with the achieved score. Originally, we wanted to rank the population only on the achieved score, but we needed to make a difference between birds who died between obstacles or didn't score at all.

When creating a new population, we use elitism - the best 20% of individuals are automatically copied to the new generation.

To restore the population to the original size, we first tried roulette selection technique, which gives better individuals a higher probability of reproduction. After a few generations, however, we observed the predominance of one type of neural network - all birds flew in the same way and in the same place. Despite 2% mutations, such population was not sufficiently diverse, which led to a fall into the local optimum, and thus did not reached the high score.

Therefore, we decided to create the rest of the population from the elite by breeding each bird from the top 20% twice with the next best one and twice with a random bird from the elite. The generation created this way had good enough, but also diverse enough agents to reach a higher score.

```
 // Best 20% of population
for (int i = sortedList.Count - 1; i > (int)(4 * sortedList.Count / 5.0f)
    ↪  - 1; i--)
{
    // Elitism - Make a copy of best 20% of birds
    population.Add(Breed(sortedList[i], sortedList[i]));

    // Breed bird with next best one to produce 2 offsprings
    population.Add(Breed(sortedList[i], sortedList[i - 1]));
    population.Add(Breed(sortedList[i - 1], sortedList[i]));

    // Breed bird with random bird from best 20%
    int j = (int)Random.Range((int)(4 * sortedList.Count / 5.0f),
        ↪ sortedList.Count);
    population.Add(Breed(sortedList[i], sortedList[j]));
    population.Add(Breed(sortedList[j], sortedList[i]));
}
```

Listing 1.1: PopulationManager.cs - snippet of function BreedNewPopulation()

Breeding itself consisted of randomly selecting the position of a neuron up to which all weights and biases are copied from one parent and the rest of the weights and biases are copied from the other. Finally, mutations are applied in 2% to individuals of the new population. Mutation consist of random selection of one of the neurons in the neural network and setting its bias to a random value from the interval.

This way, the population of 100 birds scores above 450 up to the 10th generation, while the related work 2.1 did not succeed with population of 10 birds until the 43rd generation.

5 Evaluation - Neuroevolution vs Neuroevolution with Q-learning

The original reason for creating a neural network with Q-learning was the intention to optimize this neural network with an evolutionary algorithm. However,

at first this approach did not give the results we expected. The initial hypothesis was that neural network optimized with genetic algorithm would learn faster. Based on the performed experiments, we found that the birds did not achieve a high score and the learning itself was not that visible and rather random. The reason why we did not observe an improvement is the bird's very exploration of state space.

We can see in the graph 4 that the achieved score is relatively low - up to 350. We can also observe from the data that a huge number of generations did not score at all.

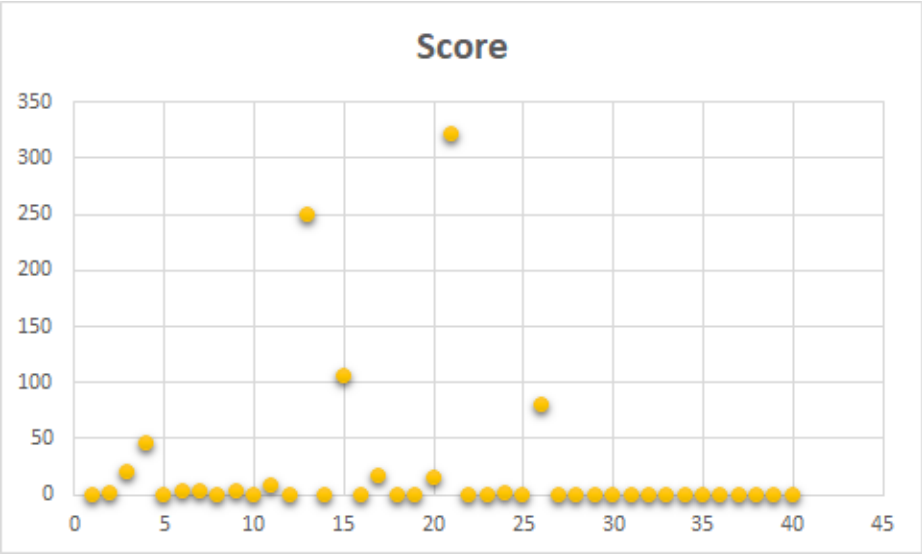

Fig. 4: Maximum score of individual generations using a genetic algorithm and a neural network with Q-learning.

After examining the data, we found that the neural network with Q-learning did not sufficiently explore the state space. So we introduced the *exploreRate* variable, which represents the rate of exploration. We set it to 20%. This value decreases periodically by $exploreDecay = 0.0001$. After this adjustment, we see a huge progress in the score itself - 2045, see 5. However, the early generations (up to the 18th generation) do not score. This is for a high percentage of exploration, and thus a high percentage of selecting a random action. Action can be selected up to two times per frame. If we randomly choose to flap or not, it is more than likely that the bird will often flap and hit the top collider situated on the top of game window. Until the value of *exploreRate* decreases sufficiently, the score does not increase, but the neural network still learns from the memory of state, action and given reward.

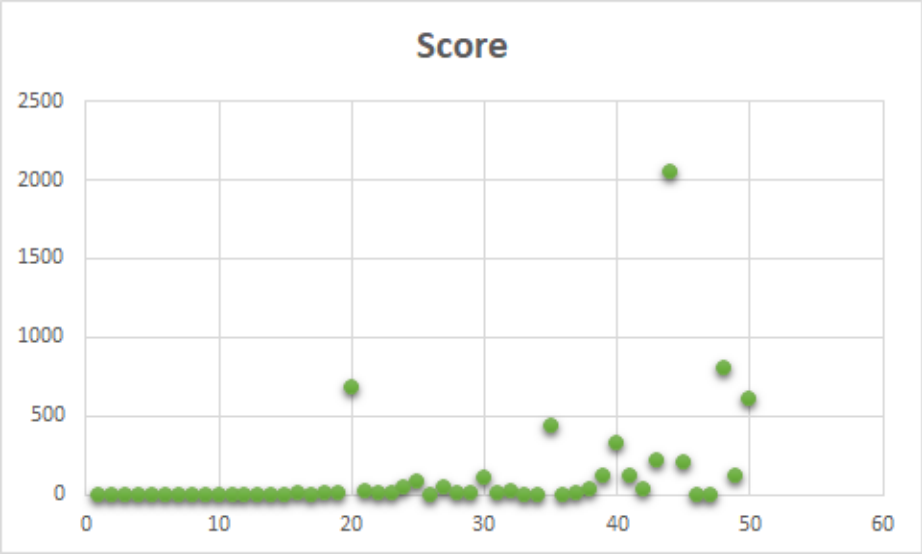

Fig. 5: Maximum score of individual generations using a genetic algorithm and a neural network with Q-learning and a 20% exploration rate.

A possible problem in neuroevolution with Q-learning is the very fact that our network is trained only after the bird is dead. The genetic algorithm thus evaluates the success by calculating the fitness function of the original neural networks, but in the meantime the weights are updated from the bird's memories. This fact can suppressed the process of evolution itself, as the best rated birds already have updated weights in the neural network, and thus may not be the best in the population as a result.

Therefore, we decided for the last experiment to skip the training and leave the optimization of the neural network purely to the genetic algorithm and natural selection. In contrast to the two previous experiments, we observe almost exponential progress in the reached score, see fig. 6.

All of the experiments above were running for the maximum of one hour and fifty generations.

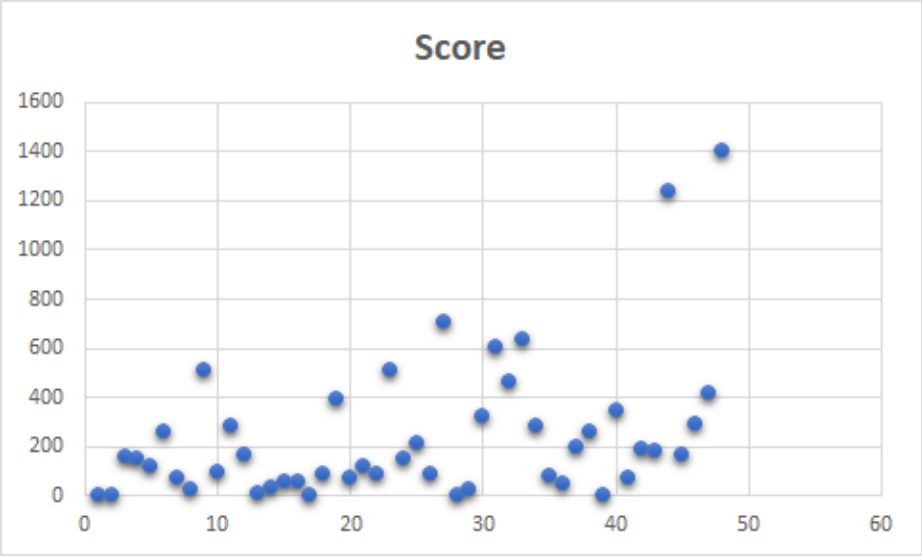

Fig. 6: Maximum score of individual generations using a genetic algorithm and a neural network.

6 Conclusion

Experiments have shown that within 50 generations, a genetic algorithm in conjunction with an ordinary neural network (GA + ANN) has proven to be a very good optimization technique. It is also very comparable to more complex version which involves genetic algorithm and neural network with Q-learning and high exploration rate (GA + ANN + RL + exploration). The score was the highest for GA + ANN + RL + exploration version, but the learning curve wasn't that nice because of high exploration rate at the beginning.

"One might expect GAs to perform far worse than other methods because they are so simple and do not follow gradients. Surprisingly, we found that GAs turn out to be a competitive algorithm for RL – performing better on some domains and worse on others, and roughly as well overall as A3C, DQN, and ES – adding a new family of algorithms to the toolbox for deep RL problems. [3]"

We expect future work to tune parameters of Q-learning neural network in the context of neuroevolution, to let the population train for much longer time and more generations and also try more selection techniques for genetic algorithm.

References

1. SRDJAN. Machine learning algorithm for flappy bird using neural network and genetic algorithm, 2017. cit: 03.05.2020. Dostupné z https://www.askforgametask.com/tutorial/machine-learning-algorithm-flappy-bird/.
2. Tony Xu. Use reinforcement learning to train a flappy bird never to die, 2017. cit: 03.05.2020. Dostupné z https://towardsdatascience.com/use-reinforcement-learning-to-train-a-flappy-bird-never-to-die-35b9625aaecc.
3. Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning, 2017.