GLITCHED BACONS

FlapANN
Self-winning Flappy Bird – Final Report

Silesian University of Technology
Faculty of Automatic Control, Electronics and Computer Science

Authors:	Dawid Grobert
	Julia Boczkowska
Created:	June 18, 2022
instructor	dr inż. Grzegorz Baron
year	2021/2022

Introduction

Flappy Bird is one of the most popular games of the year 2013/2014. Its simplicity is a perfect example for a short project, although it gives many possibilities for implementation.

Gameplay

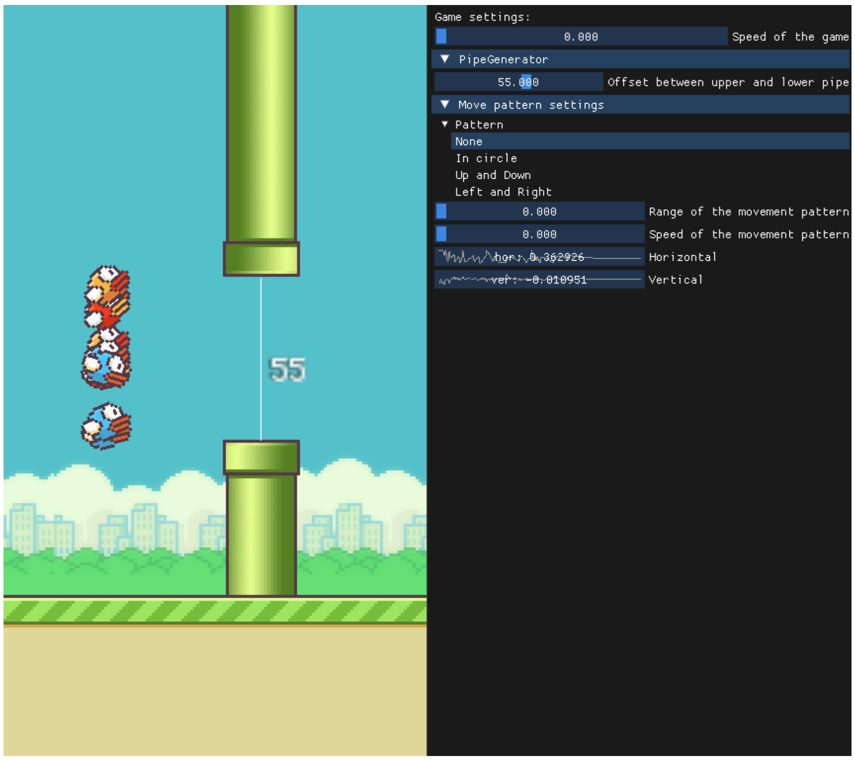
Flappy bird is a so-called *side-scroller* (see info), in which the player plays the role of a flying bird. The main goal is to avoid pipes protruding from the bottom and top of the screen by "jumping" the bird. The more pipes we jump through, the better the result.

> **Side-scroller** is usually a 2D game with a character moving to the left or right of the screen. Action is viewed from a side-view camera angle, that follows the player. Some side-scrollers allows moving only one side of the screen (such as Flappy Bird). Most often, this genre is associated with arcade games.

The bird must not fall or hit any of the pipes. The difficulty is that our bird does not maintain a stable movement and constantly falls and the impulsive "jumping" suddenly and with great power tosses it up, which requires the appropriate precision in execution in order not to hit any of the above-mentioned pipes.

1 About the project

`FlapANN` is a self-winning game. This time, it's an artificial intelligence that take on the challenge of playing the game that has prompted tens of thousands of people to destroy their smartphones. Additionally, from within the app the user is able to make life difficult for the AI by dynamically changing the spacing of the pipes, or even their speed of movement. It is possible to set different patterns of pipe movement and speed. Player can even change the speed of the game itself.



```
Game settings:
0.000        Speed of the game
▼ PipeGenerator
55.000       Offset between upper and lower pipe
▼ Move pattern settings
▼ Pattern
None
In circle
Up and Down
Left and Right
0.000        Range of the movement pattern
0.000        Speed of the movement pattern
hor: 0.362926    Horizontal
ver: -0.010951   Vertical
```

Figure 1: Screenshot from FlapANN

2 Analysis of the task

2.1 Used libraries

The vast majority of ANN's programs are developed in python, but we opted for C++. It is a language in which we program smoothly. Moreover, it offers quite a few libraries that were suitable for our needs. The libraries we used in the project are as follows:

- SFML – for creating a window and displaying the game
- ImGui – to display "debug" windows inside the game containing any kind of checkboxes or sliders.
- FANN - Fast Artificial Neural Network Library which implements artificial neural networks in C

Fast Artificial Neural Network Library was one of the many libraries available, we chose it for several reasons. It was not as huge as other libraries, and it has well described documentation.

2.1.1 Other available libraries

The choice among other libraries implementing ANN for C++ is not so small either, there are frameworks like:

- TensorFlow
- mlpack
- DyNet
- Shogun
- SHARK

Still, many of them were too large or too complex for our needs. Some even seemed to disallow interference with ANN in the way we wanted. In contrast, FANN gave us everything we needed without being a huge artillery piece for our needs.

2.2 Approach to the problem

2.2.1 Choosing training algorithm

At first, we did some research on what we should use to achieve a self-learning AI. We ended up torn apart between two options, Reinforcement learning or Genetic algorithm. Both of these algorithms are inspired by nature. Due to the simplicity of our problem and the ease of choosing fitness functions to measure the performance of individuals, we chose Genetic Algorithm. Reinforcement learning is more suitable for more complex problems, and also training time can be longer than in case of Genetic Algorithm.

Furthermore, it seems to us that gradient descent is the most commonly used algorithm in training artificial neural networks. Out of sheer curiosity, we decided to abandon the trotted out scheme and replace it with a genetic algorithm. We were not disappointed, the results we obtained were satisfactory.

This space intentionally left blank

2.2.2 ANN combined with Genetic Algorithm

At the end, in our project we decided to use an evolutionary algorithm, namely genetic algorithm (GA), to train artificial neural networks (ANNs). In a nutshell, each bird has its own neural network used as its AI brain for playing the game. ANNs can be changed very easily in our code, although at the time of handing in the project the default values of our ANN are 3 input neurons, 8 neurons in the hidden layer, and 1 output neuron.

Input neurons

The input neurons for our ANN are:

- horizontal distance of the bird to the closest pipe's gap
- vertical bird position
- vertical distance between bird and the closest pipe's gap

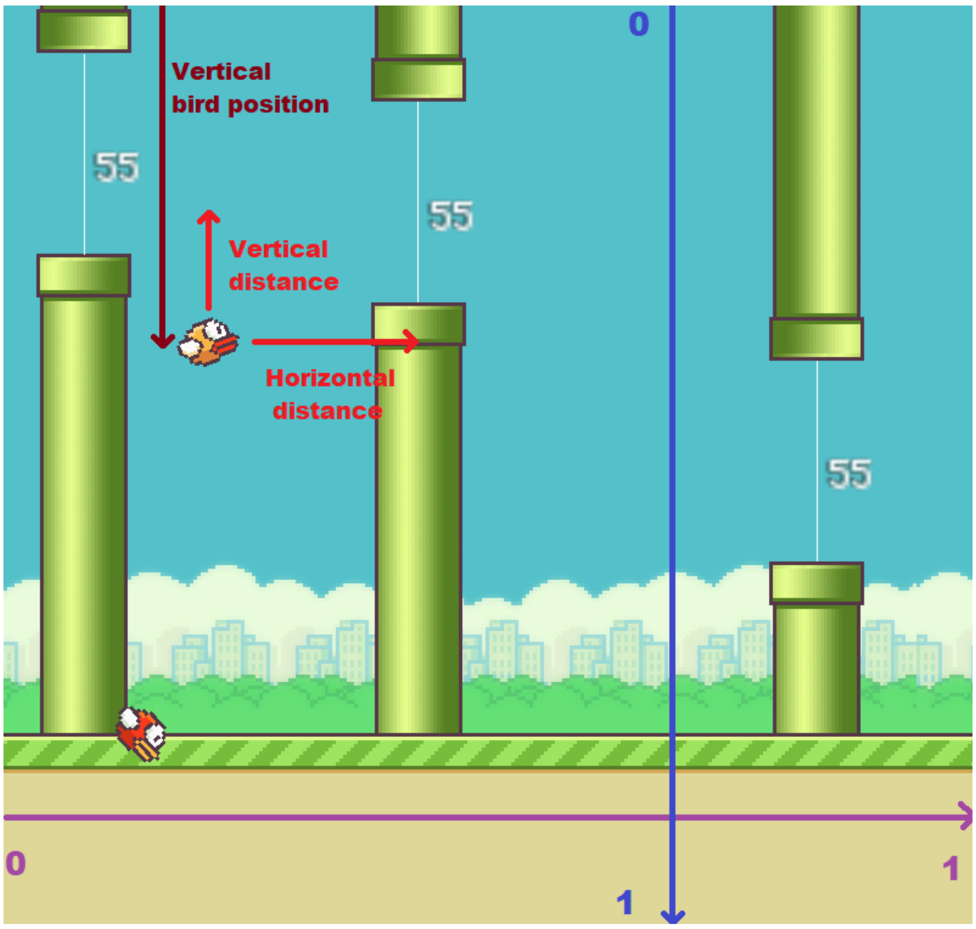


Figure 2: Image showing input neurons

All values were closed between 0 and 1. This was not required, but we hoped it would help the neural network a bit.

Output neuron

Output layer indicates whether to make a jump or not.

3 External specification

The program is created using a graphical library, so everything the user need to start the program comes down to running the executable file (.exe). Before getting into specifics, it can be seen how the AI handles the game in this short video. The video shows an example of training a bird population, as well as the use of various additional settings available in the program, such as reducing the offset between the pipes or changing their movement to up-down or left-right. A sample video of AI training is available under the QR code. The recording can also be opened simply by clicking on it.

Video

A short demonstration of the game

3.1 Mechanics

In each passing frame of the program, the AI can:

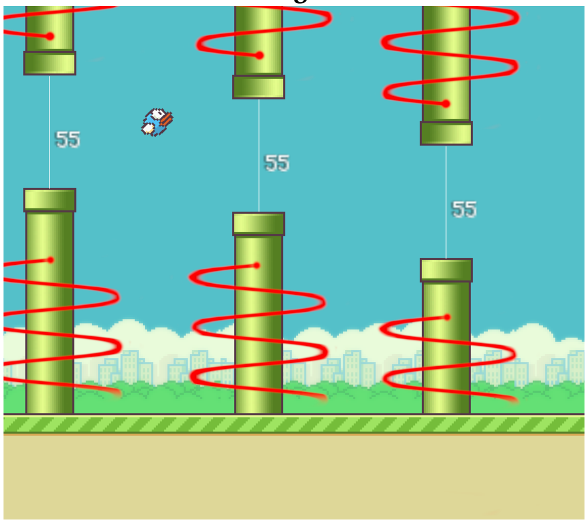
- **jump** - which causes the bird to rise
- **do nothing** - gravity will act on the bird and pull it down

The bird dies if it touches any part of the pipe. AI in learning is as lazy as students, so in addition to wanting to encourage them to develop properly, we started to consider going beyond the top area of the game as death. Touching the ground is also naturally death.

3.1.1 Move Patterns

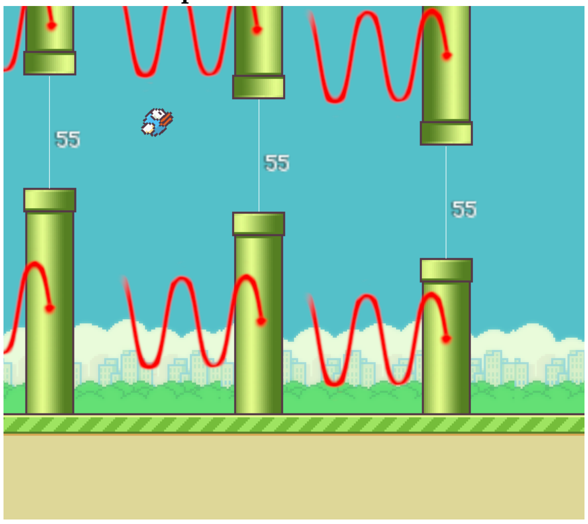
Move Patterns are an additional complication for AI. They make pipes move in a specific way. The player can adjust the speed of the movement as well as its range.

Left and Right Pattern



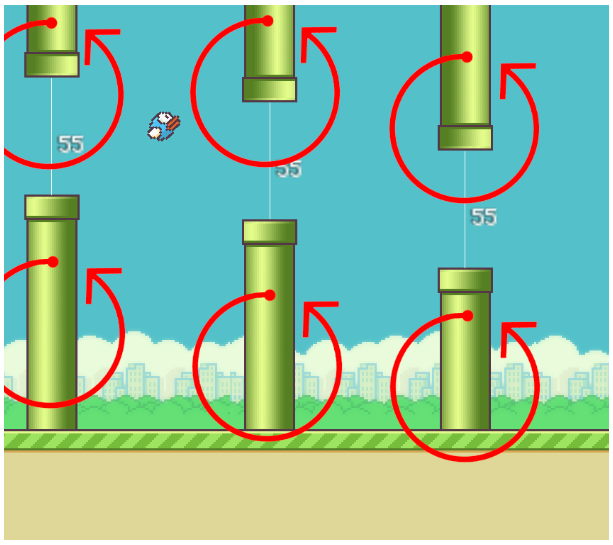
Pipes moving left-right. Thanks to the sine wave, they move smoothly and calmly

Up and Down Pattern



Pipes moving up-down. Sinusoidal motion gives smooth up-and-down move

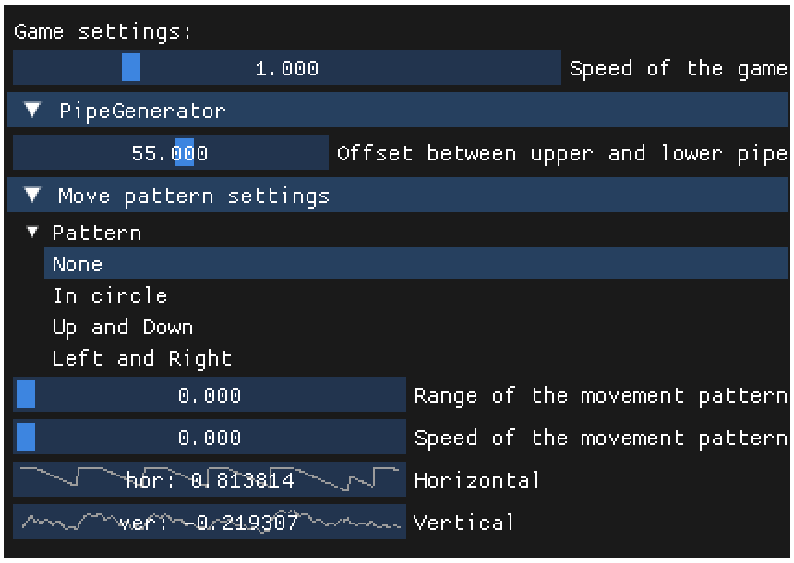
In Circle Pattern



Pipes move around the circumference of the circle. Here the player can determine their diagonal and speed of rotation too.

3.2 GUI

The ImGui library was used to create the graphical user interface. The buttons and sliders seen in the screenshot below are interactive. This means that when the user changes their value, we can expect an immediate reaction in the form of, for example, slowing down or speeding up game time, or changing the behavior of newly created pipes.



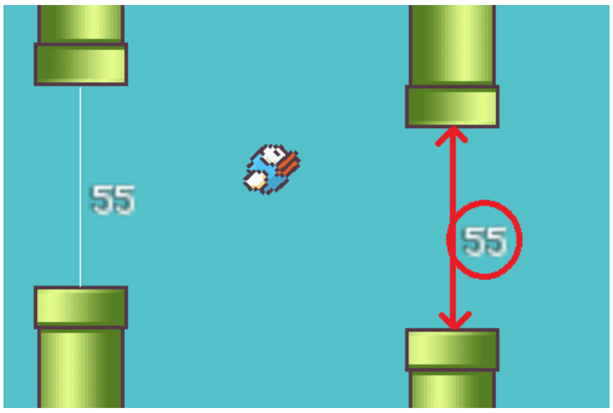
```
Game settings:
1.000                      Speed of the game
▼ PipeGenerator
55.000       Offset between upper and lower pipe
▼ Move pattern settings
 ▼ Pattern
  None
  In circle
  Up and Down
  Left and Right
0.000              Range of the movement pattern
0.000              Speed of the movement pattern
hor: 0.813814      Horizontal
ver: -0.219307     Vertical
```

Figure 3: ImGui user interface window

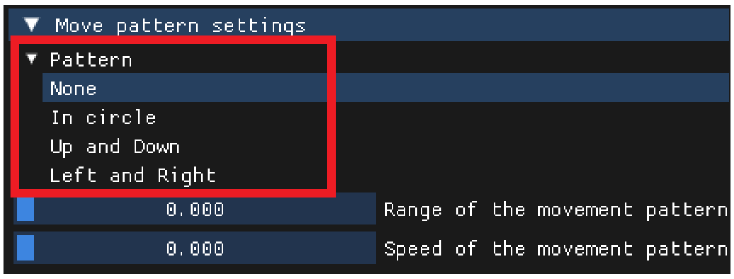
Below, there is a description of the various functionalities added to the program to make the training of AI more fun and unpredictable.

- **Speed of the game** - population training can be tedious, so a speed booster slider was added to the program. In this way, birds learn faster from our perspective. We can also slow down time to look at their efforts more closely.

- **Offset between upper and lower pipe** - the offset between pipes can be changed in real time from very small to large. For every two pipes between them, a text is drawn with a number corresponding to the distance between the two pipes

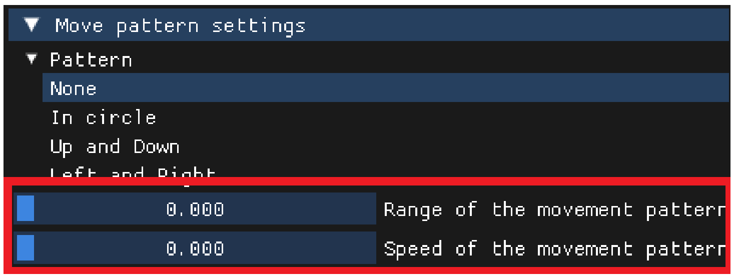


- **Move pattern settings** - by default, pipes do not move, but we can change this with just a mouse click. Four moves are available:
  - in circle
  - up and down
  - left to right.



▼ Move pattern settings
▼ Pattern
None
In circle
Up and Down
Left and Right
0.000 Range of the movement pattern
0.000 Speed of the movement pattern

- **Range of the movement pattern** - this is the maximum distance the pipes can travel with movement enabled
- **Speed of the movement pattern** - this indicates how fast the pipes will be moving



▼ Move pattern settings
▼ Pattern
None
In circle
Up and Down
Left and Right
0.000 Range of the movement pattern
0.000 Speed of the movement pattern

4 Internal specification

4.1 Genetic algorithm implementation

The **FANN** library provided the neural network, but writing the training part of the algorithm was done by us. Our implementation of `Genetic Algorithm` is as follows:

1. Create a population of given number of birds (150 by default), each with its own neural network filled with random weights and connections.

```
void GeneticAlgorithm::createPopulation()
{
    for (int i = 0; i < maxUnits(); ++i) {
        auto ann = fann_create_standard_array(mLayers.size(), mLayers.data());
        Unit unit = {ann, i, 0};
        mPopulation.push_back(unit);
    }
    for(auto& unit : mPopulation) {
        fann_randomize_weights(unit.ann, -1.f, 1.f);
    }
}
```

Figure 4: Implementation of `createPopulation()` function

2. As long as the birds are alive, we let them play the game. We constantly feed them with inputs that are then used to determine the output - jump or no jump

```
void GameManager::updateANN()
{
  if(mBirds.size() != mGeneticAlgorithm.population().size()) {
      throw std::runtime_error("Number of birds is not equal to number of 'brains'");
  }

  auto birdNumber = 0;
  for(auto& currentBird : mBirds)
  {
      const auto& nearestPipe = *mPipesGenerator.sortedByDistancePipesetsInfrontOfBird(
        currentBird.getPosition()).front();
      const auto& [horizontalDistance, verticalDistance] =
        normalizedDistancesBetweenBirdAndPipeset(currentBird, nearestPipe);
      const auto& birdPositionY = normalizedVerticalBirdPosition(currentBird);
      const auto& distanceToGap = distance(horizontalDistance, verticalDistance);

      auto& currentGenome = mGeneticAlgorithm.at(birdNumber);
      currentGenome.fitness = calculateBirdFitnessScore(currentBird, distanceToGap);
      currentGenome.performOnPredictedOutput(
                { horizontalDistance, verticalDistance, birdPositionY },
                [&currentBird](fann_type* output)
                {
                    if(output[0] > 0.5f)
                    {
                        currentBird.flap();
                    }
                });
    ++birdNumber;
  }
}
```

Figure 5: Implementation of function updating ANN every frame

3. When all birds are dead, we evolve the current population. We do this by selecting the best individuals and mixing their genes. The **crossover** process will be described in more details in the upcoming sections.

```
void GeneticAlgorithm::evolve()
{
    resetIfTheBestUnitIsTooWeak();
    evolveWeakUnits();
}

void GeneticAlgorithm::evolveWeakUnits()
{
        reassignPopulation(replaceWeakBirdsWithCrossovers(sortByFitness(mPopulation)));
        ++mCurrentGeneration;
}

void GeneticAlgorithm::resetIfTheBestUnitIsTooWeak()
{
    if(doesBestUnitFailed())
    {
        clearPopulation();
        createPopulation();
    }
}

std::vector<GeneticAlgorithm::Unit> GeneticAlgorithm::replaceWeakBirdsWithCrossovers(
                                    std::vector<GeneticAlgorithm::Unit>&& sortedPopulationByFitness)
{
        const auto firstWeakUnitIndex = mTopUnits;
        const auto& populationSizeWithoutTopUnits = mPopulation.size() - mTopUnits;

        for(int i = 0; i < populationSizeWithoutTopUnits; ++i)
        {
                auto offspring = std::unique_ptr<Unit>();

                if(i == 0)
                {
                        offspring = crossoverTwoBestUnits();
                }
                else if (i < populationSizeWithoutTopUnits - 2)
                {
                        offspring = crossoverTwoRandomBestUnits();
                }
                else
                {
                        offspring = crossoverTwoRandomUnits();
                }

                offspring->mutate();
                sortedPopulationByFitness[firstWeakUnitIndex + i] = *offspring.release();
        }
    return sortedPopulationByFitness;
}
```

Figure 6: Implementation of the evolve function

4. Go back to step two and repeat until you reach maximum fitness

4.2 Fitness function

In order to choose and later crossover only the best individuals, `GA` uses a fitness function. Our function is very simple, birds are scored based on the distance they have flown, the greater the distance, the better the score. Based on this value, the birds are sorted and ranked, ensuring that only the best units are selected.

Additionally, we have a small addition in the form of distance to the gap. The closer the bird was to the gap at the time of death the better the score. This is to encourage birds to try to get closer to the gap in the first stages of learning.

4.2.1 Code implementation

```
float GameManager::calculateBirdFitnessScore(const Bird& currentBird, const float& distanceToGap)
{
        return currentBird.birdScore() - distanceToGap / 10.f;
}
```

Figure 7: Implementation of fitness function

4.3 Replacing weak birds with crossovers

For an algorithm to work, only the genes of the fittest individuals must survive. For this reason, in our code, we use a function that replace weak birds as a mixture of stronger birds. Indirectly, we use here the crossover mechanism about which more will be said in section `4.4. Crossover`. The flow of the function is as follows:

1. Sort the population by fitness
2. Take x number of top units and preserve their genes to next generation
3. Replace the remaining population (weak birds) with new offspring created in different ways
   - As crossover product of two random top units
   - As crossover product of two best units
   - As crossover product of two random units
4. At the end, connections and neurons of each of the offspring are randomly selected – a crossover mechanism is used here.
5. There is also a 20% chance of gene mutation by default. Each weight is checked for this and if a mutation occurs then the weight is changed accordingly.

$$gene\ weight = gene\ weight \times (1 + ((x - 0.5) \times 3 + (y - 0.5))$$

   where x, y are random values between $(0, 1)$

Info: It is worth noting here that the best units are not changed in any way. The whole operation is carried out only on weaker units. How many units do we consider to be the best? Depends on the settings, but by default we assume 5 for the whole population (default population size is 150).

4.3.1 Code implementation

Goal of the function below is to select the best individuals and then cross their genes in various ways. More about the crossover will be in the section `4.4. Crossover`.

```
std::vector<GeneticAlgorithm::Unit> GeneticAlgorithm::replaceWeakBirdsWithCrossovers(
                                    std::vector<GeneticAlgorithm::Unit>&& sortedPopulationByFitness)
{
        const auto firstWeakUnitIndex = mTopUnits;
        const auto& populationSizeWithoutTopUnits = mPopulation.size() - mTopUnits;

        for(int i = 0; i < populationSizeWithoutTopUnits; ++i)
        {
                auto offspring = std::unique_ptr<Unit>();

                if(i == 0)
                {
                        offspring = crossoverTwoBestUnits();
                }
                else if (i < populationSizeWithoutTopUnits - 2)
                {
                        offspring = crossoverTwoRandomBestUnits();
                }
                else
                {
                        offspring = crossoverTwoRandomUnits();
                }

                offspring->mutate();
                sortedPopulationByFitness[firstWeakUnitIndex + i] = *offspring.release();
        }

    return sortedPopulationByFitness;
}
```

Figure 8: Implementation of function responsible for choosing units for crossover

This space intentionally left blank

4.4 Crossover

Crossover is a very simple mechanism of combining two units (parents) and a new unit (child) is created based on their weights. In the figure below (see fig. 9), you can see two arrays. One from parent A and the other from parent B. Both represent an array of weights. Then a completely random point is took from the range of `0` to the `size of the array - 1`, and this is the cross-section point.

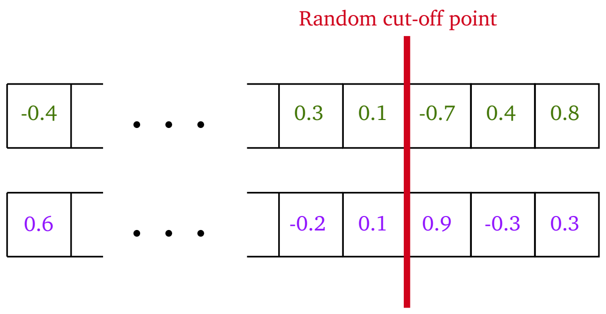
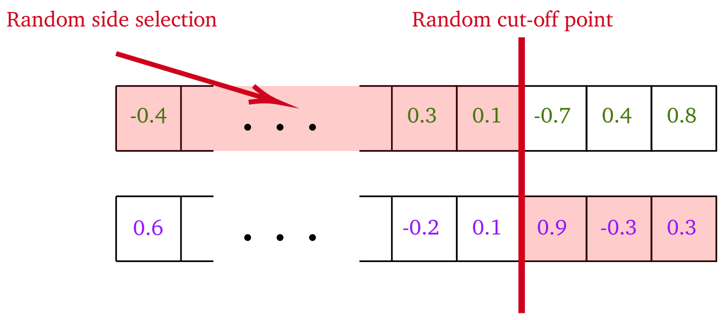
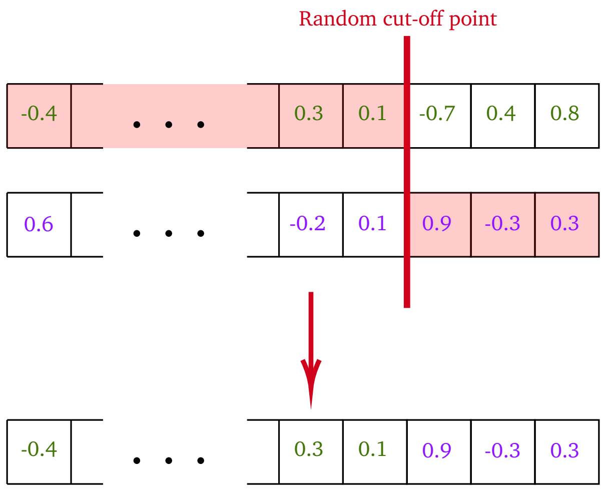


Figure 9: Arrays of parent weights A and B crossed by a random cross-section point

The next step is to choose a side. For example, we can give the child the weights from the left side of the intersection, taking them from either parent A or parent B. Here we have chosen the weights from the left side from parent A. In another iteration they may be from parent B. It is all a matter of luck.



Since we chose to rewrite the left side from the intersection from parent A, we take the right side from parent B. In this way, we have created a new array of weights for the resulting child.



As it can be seen, the child inherits the behavior of both parents in part. Each time a little differently.

4.4.1 Code implementation

Below the crossover function can be seen itself. The two parents are taken, and their genes are mixed as described in section `4.4. Crossover`.

```
std::unique_ptr<GeneticAlgorithm::Unit> GeneticAlgorithm::crossover(
                                        const Unit& parentA,
                                        const Unit& parentB) const
{
    static std::random_device rd;
    static std::mt19937 gen(rd());
    const std::uniform_int_distribution<> distr(0, parentA.ann->total_connections-1);
    const std::bernoulli_distribution trueOrFalse;

    auto cutPoint = distr(gen);
    for (int i = cutPoint; i < parentA.ann->total_connections; ++i)
    {
        auto biasFromA = parentA.ann->weights[i];
        parentA.ann->weights[i] = parentB.ann->weights[i];
        parentB.ann->weights[i] = biasFromA;
    }

    return (trueOrFalse(gen) ? std::make_unique<Unit>(parentA) : std::make_unique<Unit>(parentB));
}
```

Figure 10: Implementation of crossover function

5 Experiments

During development, we experimented with different values to improve AI training to be as quick as possible. Some of these experiments are described below.

5.1 Fitness function

One change that proved to be a small improvement was the enhancement to the fitness function. Not only the distance was calculated, but also extra points were given when the bird died closer to the gap between pipes. In this way, we tried to direct and encourage them to pass through. With this functionality, we have noticed that birds learn slightly faster than it was seen before.

5.2 When the bird should jump

Determining the most optimal output level was somewhat ambiguous. The way we call out whether a bird should do a flap or not is very simple:

```
...
currentGenome.performOnPredictedOutput({ horizontalDistance, verticalDistance, birdPositionY },
[&currentBird](fann_type* output)
{
    if(output[0] > 0.5f)
    {
        currentBird.flap();
    }
});
...
```

The output values are returned to the range of 0 to 1. Based on this, our task was to determine the optimal threshold from which the birds should jump.

By setting a value as small as 0.1, we could be sure that most birds would start jumping like crazy. In such a case the chance of jumping is very high, it could be simplified to some shorthand and say even 90%.

This didn't make it easy to find a bird that was better than the others, as they would all simultaneously kill themselves indiscriminately over the top edge of the window. This occurrence may have extinguished the evolution of the population for a time.

The same phenomenon occurred in reverse when the threshold value was too high. If we considered the chance of jumping too low, then the birds did not want to jump at all and the very first generations died at the very beginning simply by not jumping and dying.

Therefore, we decided to set a neutral and optimal threshold, which was 0.5. This preserved the appropriate randomness of the population as well as the relatively fast visible effects of training.

5.3 Changing number of inputs

To get the best results obtained by the neural network, we used three inputs, which are the vertical position of the bird and the vertical and horizontal distance from the pipe. But before knowing these input values, we had to test a couple of values. Below, we have listed how the AI behaved when we provided it with different types of input data.

Different combinations of input data:

- **distance to middle of the gap** - Distance to gap was calculated using `Pythagoras theorem`. The bird had too little information to learn properly how to jump through even a single pipe. From this, we can conclude that in order to correctly determine the centre of the pipe, AI needs the horizontal and vertical distance to the pipe as two separate inputs.
- **distance to middle of the gap and current bird's vertical position** - With this combination of inputs, the AI seemed also not to learn at all. The vertical position of the bird is only helpful in determining how high or low it is. It can thus only learn that touching any edge of the screen results in death.
- **horizontal and vertical distance** - These two values seem to be sufficient to train the bird population. The disadvantage of this solution is that there is no indication of the current vertical position of the bird. The birds have too little information to determine at what level they might die from touching one edge of the window.
- **horizontal and vertical distance and current bird's vertical position** - This is the optimal configuration we are using in our program.

6 Summary

The course, and the project, was a great opportunity to explore the science of artificial intelligence inspired by evolutionary algorithms. The result we eventually got was not easy to achieve. While writing the genetic algorithm, we encountered many errors that took us a considerable amount of time to solve. For a long time, the only thing was the sight of constantly falling birds that did not evolve in the right way due to trivial errors in the code. However, by taking another approaches and rethinking of the problem finally proved successful.

The project is complete, and we do not plan to develop it further. Although, a tempting option would be to implement an `ANN` that improves itself using reinforcement learning. Comparing the speed of their training time would be very interesting.

7 References

[1] Laurent Gomila and many others. Sfml. `https://github.com/SFML/SFML`.

[2] Omar Cornut. Dear imgui. `https://github.com/ocornut/imgui`.

[3] Jakub Zelenka. Fann. `https://github.com/libfann/fann`.

[4] Tensorflow. `https://github.com/tensorflow`.

[5] Artidoro Pagnoni. Dynet. `https://github.com/clab/dynet`.

[6] Heiko Strathmann, Viktor Gal, Sergey Lisitsyn, Soeren Sonnenburg, Gunnar Raetsch, and Fernando Iglesias Garcia. Shogun. `https://github.com/shogun-toolbox/shogun`.

[7] Oswin Krause. Shark. `https://github.com/Shark-ML/Shark`.

[8] Konrad Budek Błażej Osiński. What is reinforcement learning, the complete guide. `https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/`.

[9] Wikipedia contributors. Genetic algorithm — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Genetic_algorithm&oldid=1092570674`, 2022. [Online; accessed 18-June-2022].