**A Project Report on**

**AI Playing Flappy Bird**

BY

**Pratik Rajesh Tiwari (1932377)**

FOR THE DEGREE OF

**BACHELOR OF COMPUTER APPLICATION (SCIENCE)**

UNDER THE GUIDANCE OF

**Mrs. Ishita Dixit Mr. Deepak Kumbhar**

(Project Guide) (Head of the Dept.)

DEPARTMENT OF BCA (SCIENCE)

Modern College of Arts Science and Commerce College,

Ganeshkhind, Pune – 411007

April - 2020

**Modern College of Arts Science and Commerce College, Pune.**

**DEPARTMENT OF BCA (SCIENCE)**

**CERTIFICATE**

This is to certify that \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ a student of class TYBCA (Science) has successfully completed the Project entitled: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_during the academic year of 2019 – 20.

Signature Signature

(Project Guide) (Head of the Dept)

**Table of Content**

|  |  |  |
| --- | --- | --- |
| **Chapter** | **Topics** | **Pg. No.** |
| **1** | **Abstract** | **4** |
| **2.** | **Introduction** | **5** |
| **3.** | **Software requirement Analysis** | **6** |
| **4.** | **Fundamental Concept** | **7** |
| **5.** | **Advantages** | **19** |
| **6.** | **Screenshots** | **20** |
| **7.** | **Conclusion** | **29** |
| **8.** | **References** | **30** |

**Abstract**

Genetic Algorithms and Reinforcement Learning are useful for finding solutions to problems for which there is no clear solution and the situations vary with time, i.e. for probabilistic situations. In this project, we have implemented an AI for the Flappy Bird which makes the computer play the game much more efficient as compared to what a normal human being can do. We have achieved these goals using Genetic Algorithms with an underlying Neural Network Architecture

The aim of this system is to make the machine understand the game so deeply that it plays the game without failing or causing any mistake.

The machine understands the pattern of the game on its own and creates an optimal pathway to tackle the problem and learns on its own. Ai in the field of gaming can help the gaming experience exponentially.

This project is based on neural networks

Neuroevolution, i.e. evolving artificial neural networks with genetic algorithms, has been highly effective in reinforcement learning tasks, particularly those with hidden state information. An important question in neuroevolution is how to gain an advantage from evolving neural network topologies along with weights. We present a method, NeuroEvolution of Augmenting Topologies (NEAT) that outperforms the best fixed-topology methods on a challenging benchmark reinforcement learning task.

**Introduction**

AI and Machine learning is an emerging field and it has many application in this modern era. Artificial Intelligence is an approach to make a computer, a robot, or a product to think how smart human think. AI is a study of how human brain think, learn, decide and work, when it tries to solve problems. And finally this study outputs intelligent software systems. The aim of AI is to improve computer functions which are related to human knowledge, for example, reasoning, learning, and problem-solving.

The application of ai has increased a lot

Example:

Gaming, Natural Language Processing , Expert Systems, Vision Systems , Handwriting Recognition, Intelligent Robots

Techniques used in AI game programming include decision trees and pathfinding. Some AI opponents in first-person shooter games can listen for player movements, look for footprints or even take cover when a human opponent fires on them. Artificial intelligence has long been used to simulate human players in board games.

This project mainly focusses on the gaming application and hence with using computer power plays the flappy bird game on its own and increases its accuracy to the level at which it can play the game without any flaws.

**Software Requirement Analysis**

* Operating System : Windows, Linux
* Front End Software : Python 3.6
* Back End Software : Python 3.6

**Fundamental Concept**

**Creating Game UI**

Flappy bird is a common and mostly used game by the kids. Most of the people love to play these sort of games. The game is particularly made with Unity or Unreal. You should have a knowledge on designing for the layout of the characters in a game.  
The game mostly consists of characters and the obstacles they are facing to clear the step. This kind of games is 2D games which have only got a single background and the continuous random obstacles. We use coding as well as the software to make the games but you don’t need them if you want a simple game.

**Pygame library used to create UI**

## Setting Up a Pygame Program

The first few lines of code in the Hello World program are lines that will begin almost every program you write that uses Pygame.

 1. import pygame, sys

Line 1 is a simple import statement that imports the pygame and sys modules so that our program can use the functions in them. All of the Pygame functions dealing with graphics, sound, and other features that Pygame provides are in the pygame module.

Note that when you import the pygame module you automatically import all the modules that are in the pygame module as well, such as pygame.images and pygame.mixer.music. There’s no need to import these modules-inside-modules with additional import statements.

 2. from pygame.locals import \*

Line 2 is also an import statement. However, instead of the import modulename format, it uses the from modulename import \* format. Normax`lly if you want to call a function that is in a module, you must use the modulename.functionname() format after importing the module. However, with from modulename import \*, you can skip the modulename. portion and simply use functionname() (just like Python’s built-in functions).

The reason we use this form of import statement for pygame.locals is because pygame.locals contains several constant variables that are easy to identify as being in the pygame.locals module without pygame.locals. in front of them. For all other modules, you generally want to use the regular import modulename format. (There is more information about why you want to do this at [http://invpy.com/namespaces](https://invpy.com/namespaces).)

 4. pygame.init()

Line 4 is the pygame.init() function call, which always needs to be called after importing the pygame module and before calling any other Pygame function. You don’t need to know what this function does, you just need to know that it needs to be called first in order for many Pygame functions to work. If you ever see an error message like pygame.error: font not initialized, check to see if you forgot to call pygame.init() at the start of your program.

 5. DISPLAYSURF = pygame.display.set\_mode((400, 300))

Line 5 is a call to the pygame.display.set\_mode() function, which returns the pygame.Surface object for the window. (Surface objects are described later in this chapter.) Notice that we pass a tuple value of two integers to the function: (400, 300). This tuple tells the set\_mode() function how wide and how high to make the window in pixels. (400, 300) will make a window with a width of 400 pixels and height of 300 pixels.

Remember to pass a tuple of two integers to set\_mode(), not just two integers themselves. The correct way to call the function is like this: pygame.display.set\_mode((400, 300)). A function call like pygame.display.set\_mode(400, 300) will cause an error that looks like this: TypeError: argument 1 must be 2-item sequence, not int.

The pygame.Surface object (we will just call them Surface objects for short) returned is stored in a variable named DISPLAYSURF.

 6. pygame.display.set\_caption('Hello World!')

Line 6 sets the caption text that will appear at the top of the window by calling the pygame.display.set\_caption() function. The string value 'Hello World!' is passed in this function call to make that text appear as the caption:

## Game Loops and Game States

 7. while True: # main game loop

 8.     for event in pygame.event.get():

Line 7 is a while loop that has a condition of simply the value True. This means that it never exits due to its condition evaluating to False. The only way the program execution will ever exit the loop is if a break statement is executed (which moves execution to the first line after the loop) or sys.exit() (which terminates the program). If a loop like this was inside a function, a return statement will also move execution out of the loop (as well as the function too).

## pygame.event.Event Object

The list of Event objects will be for each event that has happened since the last time the pygame.event.get() function was called. (Or, if pygame.event.get() has never been called, the events that have happened since the start of the program.)

 7. while True: # main game loop

 8.     for event in pygame.event.get():

Line 8 is a for loop that will iterate over the list of Event objects that was returned by pygame.event.get(). On each iteration through the for loop, a variable named event will be assigned the value of the next event object in this list. The list of Event objects returned from pygame.event.get() will be in the order that the events happened. If the user clicked the mouse and then pressed a keyboard key, the Event object for the mouse click would be the first item in the list and the Event object for the keyboard press would be second. If no events have happened, then pygame.event.get() will return a blank list.

## The QUIT Event and pygame.quit() Function

 9.         if event.type == QUIT:

10.             pygame.quit()

11.             sys.exit()

Event objects have a **member variable** (also called **attributes** or **properties**) named type which tells us what kind of event the object represents. Pygame has a constant variable for each of possible types in the pygame.locals modules. Line 9 checks if the Event object’s type is equal to the constant QUIT. Remember that since we used the from pygame.locals import \* form of the import statement, we only have to type QUIT instead of pygame.locals.QUIT.

12.     pygame.display.update()

Line 12 calls the pygame.display.update() function, which draws the Surface object returned by pygame.display.set\_mode() to the screen (remember we stored this object in the DISPLAYSURF variable). Since the Surface object hasn’t changed (for example, by some of the drawing functions that are explained later in this chapter), the same black image is redrawn to the screen each time pygame.display.update() is called.

There are many functions in pygame that can be used to create UI of a game

**NeuroEvolution of Augmenting Topologies (NEAT) Algorithm**

This is the algorithm used to understand the game and play on its own.

**INTRODUCTION**

Neuroevolution (NE), the artificial evolution of neural networks using genetic algorithms, has shown great promise in reinforcement learning tasks. NE outperforms standard reinforcement learning methods in many benchmark tasks [6, 10, 11]. Neural networks are a good class of decision making systems to evolve because they are capable of representing solutions to many different kinds of problems, and the mapping from genotype to phenotype is generally efficient.

NE is particularly well suited to reinforcement learning tasks because NE does not require supervision. A major question in NE is how to gain an advantage from evolving topology in addition to connection weights. On one hand, evolving topology might overcomplicate the search. On the other, it can also save time by finding the right number of hidden neurons for a particular problem automatically [7]. A previous study showed that fixed-topology NE can outperform a topology-evolving system on the benchmark double pole balancing task [6]. This finding is important because pole balancing has been a benchmark task in NE and reinforcement learning for over 30 years [1, 6, 7, 9], and double pole balancing is challenging to even the best of modern methods. Doing well at this important benchmark suggests that a method will do well in other tasks as well.

Whether Topology and Weight Evolving Artificial Neural Networks (TWEANNs) can enhance the performance of NE remains an open question. In this article, we aim to show that evolving topology can indeed increase performance. We present a new TWEANN, NeuroEvolution of Augmenting Topologies(NEAT), that significantly outperforms the fixed-topology NE method that currently takes the fewest evaluations on the double pole balancing task.

We identify three major challenges for TWEANNs and presentsolutions to each of them:

(1) Is there a genetic representation that allows disparate topologies to crossover in a meaningful way? Our solution is to use historical markings to line up genes with the same origin.

(2) How can topological innovation that needs a few generations to optimize be protected so that it does not disappear from the population prematurely? Our solution is to separate each innovation into a different species.

(3) How can topologies be minimized throughout evolution without the need for a specially contrived fitness function that measures complexity? Our solution is to start from a minimal structure and grow only when necessary. This paper establishes that each of our solutions is necessary by showing that NE performance significantly declines with the ablation of any of the major solution components. Working together in NEAT these components constitute a promising new approach to difficult reinforcement learning tasks. We begin by describing the NEAT method, including results showing that NEAT is significantly faster than other NE methods on the hardest pole balancing benchmark. We then present ablation studies designed to explain NEAT’s performance in terms of its components.

**NEUROEVOLUTION OF AUGMENTING TOPOLOGIES (NEAT)**

1. **Genetic Encoding**

NEAT is designed specifically to address the three challenges raised in the introduction. Each genome includes a list of connection genes, each of which refers to two node genes being connected (figure 1). Each connection gene specifies the in-node, the out-node, the weight of the connection, whether

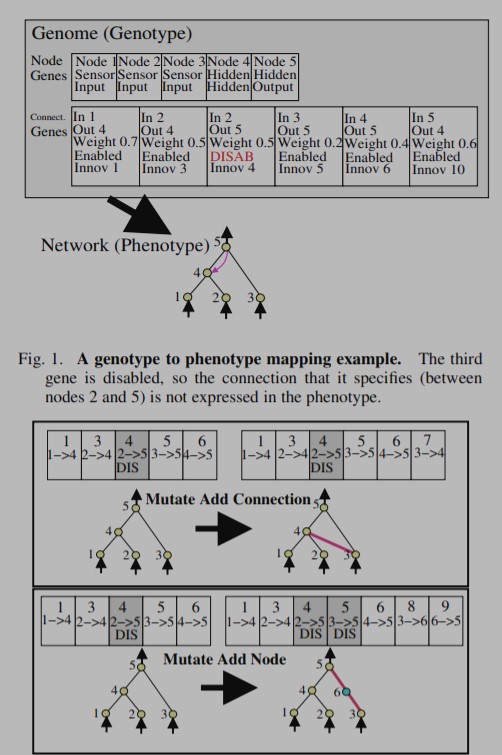
****

Fig. 2. The two types of structural mutation in NEAT. Both types, adding a connection and adding a node, are illustrated with the genes above their phenotypes. The top number in each genome is the innovation number of that gene. These numbers identify the original historical ancestor of each gene, making it possible to find matching genes during crossover. New genes are assigned new increasingly higher numbers. or not the connection gene is expressed (an enable bit), and an innovation number, which allows finding corresponding genes during crossover (as will be explained below). Although the experiments in this paper evolve networks with a single output, NEAT can evolve networks with any number of inputs or outputs.

1. **Tracking Genes**

through Historical Markings In order to perform crossover, the system must be able to tell which genes match up between any individuals in the population. The key observation is that two genes that have the same historical origin represent the same structure (although possibly with different weights), since they were both derived from the same ancestral gene from some point in the past. Thus, all a system needs to do to know which genes line up with which is to keep track of the historical origin of every gene in the system.

Tracking the historical origins requires very little computation. Whenever a new gene appears (through structural mutation), a global innovation number is incremented and assigned to that gene. The innovation numbers thus represent a chronology of every gene in the system. As an example, let us say the two mutations in figure 2 occurred one after another in the system. The new connection gene created in the first mutation is assigned the number , and the two new connection genes added during the new node mutation are assigned the numbers and . In the future, whenever these genomes crossover, the offspring will inherit the same innovation numbers on each gene; innovation numbers are never changed. Thus, the historical origin of every gene in the system is known throughout evolution.

1. **Protecting Innovation**

through Speciation Adding new structure to a network usually initially reduces fitness. However, NEAT speciates the population, so that individuals compete primarily within their own niches instead of with the population at large.

This way, topological innovations are protected and have time to optimize their structure before they have to compete with other niches in the population. Speciation is commonly used in multimodal function optimization and in coevolution of modular systems, where its main function is to preserve diversity [8, 12].

We bring the idea to TWEANNs, where its main task is to protect innovation. Historical markings make it possible for the system to divide the population into species based on topological similarity. The number of excess and disjoint genes between a pair of genomes is a natural measure of their compatibility.

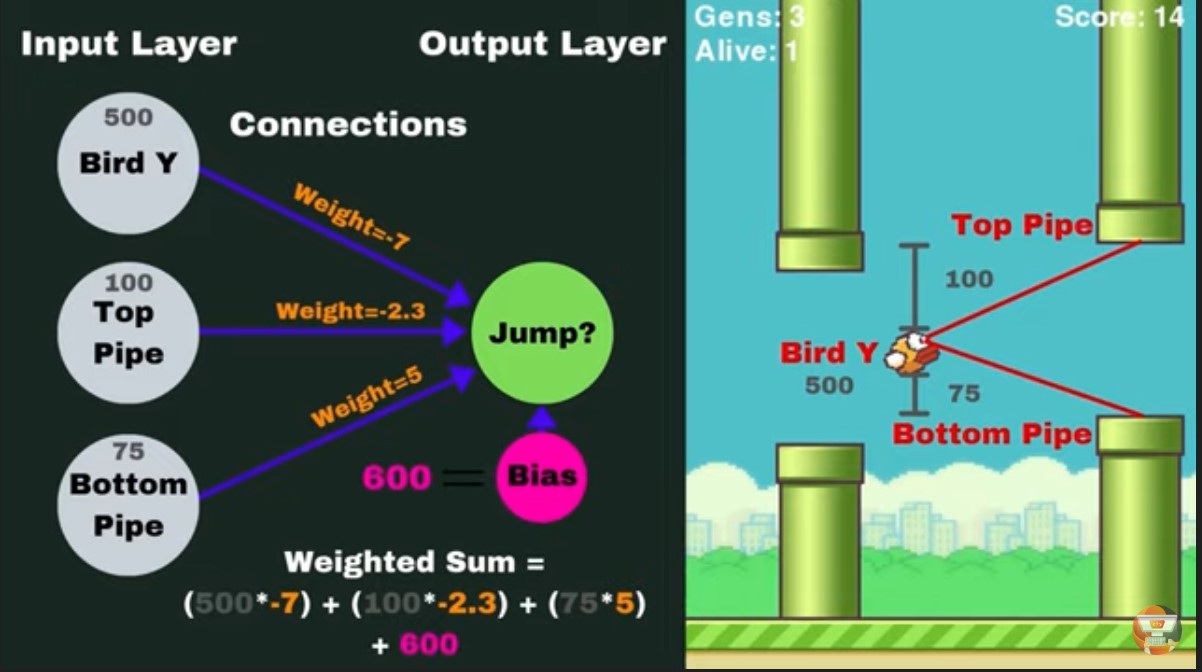
The more disjoint two genomes are, the less evolutionary history they share, and thus the less compatible they are. Therefore, we can measure the compatibility distance of different structures in NEAT as a simple linear combination of the number of excess () and disjoint () genes, as well as the average weight differences of matching genes

1. **Minimizing Dimensionality**

TWEANN algorithms typically start with an initial population of random topologies [2, 7, 21, 22]. Such topological diversity must be introduced from the start because new structure frequently does not survive in these methods, which do not protect innovation. However, it is not clear that such diversity is necessary or useful.

A population of random topologies has a great deal of structure that has not withstood a single fitness evaluation. Therefore, there is no way to know if any of such structure is necessary. It is costly though because the more connections a network contains, the higher the number of dimensions that need to be searched to optimize the network. Therefore, with random topologies the algorithm may waste a lot of effort by optimizing unnecessarily complex structures

**HOW NEAT IS USED IN FLAPPY BIRD**

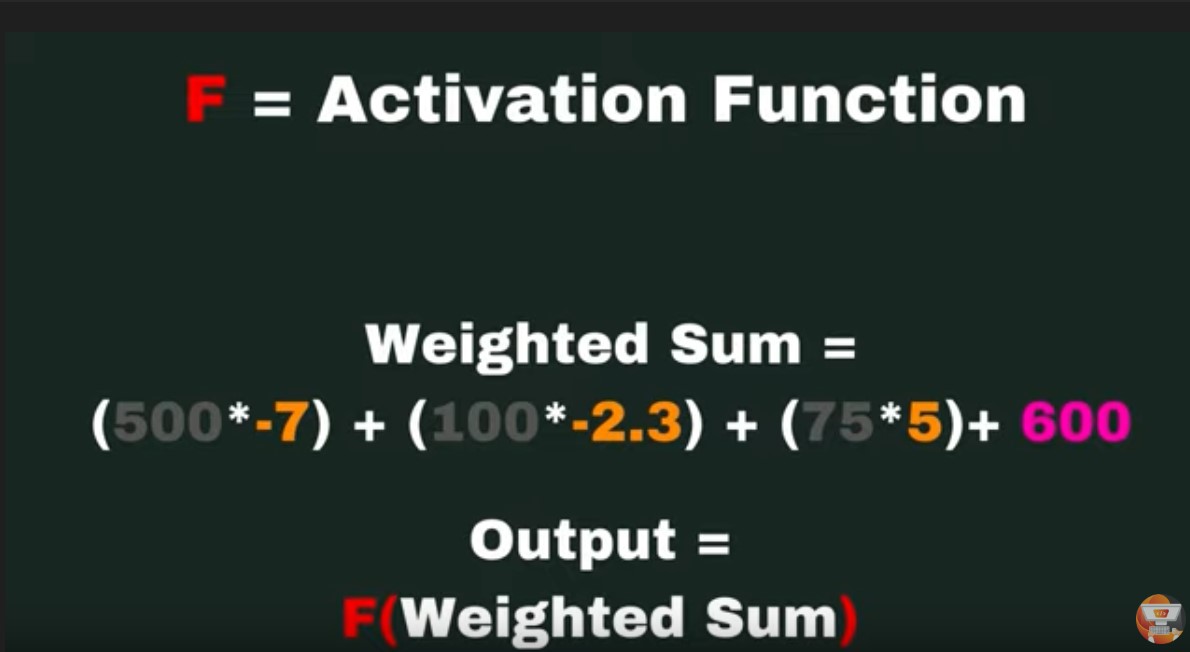
****

A Neural Network works here where it has a an input, bias, weights and connto calculate a output.

A **neural network** is trained by adjusting **neuron** input weights based on the **network's** performance on example inputs. If the **network** classifies an image correctly, weights contributing to the correct answer are increased, while other weights are decreased

**Weight** is the parameter within a **neural network** that transforms input data within the **network's** hidden layers. ... As an input enters the node, it gets multiplied by a **weight** value and the resulting output is either observed, or passed to the next layer in the **neural network**.

**Bias** unit is an "extra" **neuron** added to each pre-output layer that stores the value of 1. ... As you can see, a bias unit is just appended to the start/end of the input and each hidden layer, and isn't influenced by the values in the previous layer. In other words, these neurons don't have any incoming connections.

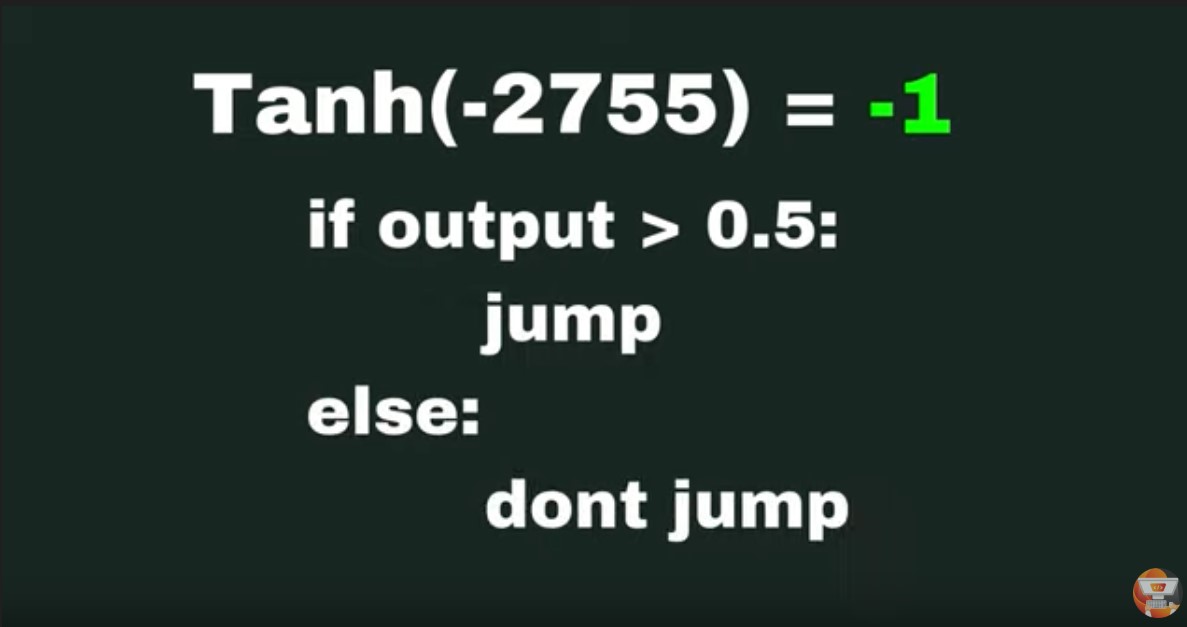


An activation function f is applied to the weighted sum to get a single values as an output.

**Activation functions** are mathematical equations that determine the output of a **neural network**. The **function** is attached to each **neuron** in the **network**, and determines whether it should be activated (“fired”) or not, based on whether each **neuron's** input is relevant for the model's prediction.

There are many activation functions to choose from

1. Sigmoid or Logistic
2. Tanh — Hyperbolic tangent
3. ReLu -Rectified linear units

**

After applying the activation function which in our case is tanh we get the value of the output

If the output is greater than 0.5 then the flappy bird jumps otherwise it doesn’t jumps.

And that is how the Neat algorithm decides if it has to jump or not.

# Requirements

The modulebelow only requires Python 2.6, Python 2.7, or Python 3. You may want to install these modules

* Numpy
* Pygame
* neat-python
* graphviz
* matplotlib

# Usage

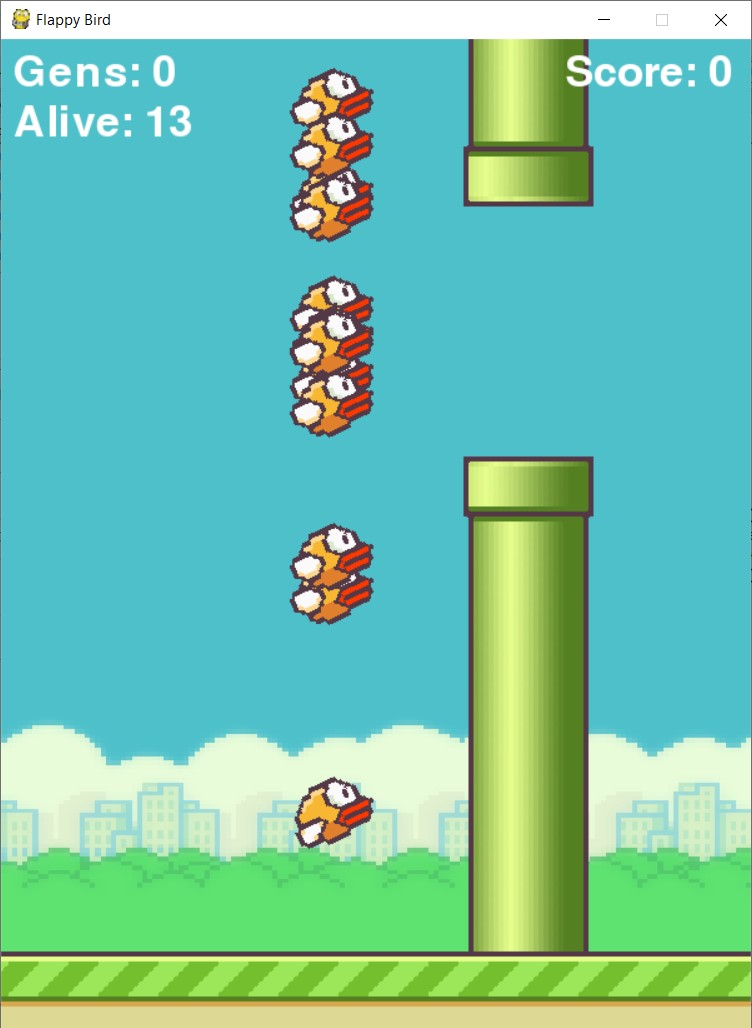
* Draw items on your screen
* Play sound effects and music
* Handle user input
* Implement event loops
* Describe how game programming differs from standard procedural Python programming
* Automate processes in games as well as in real life using neat algorithms etc

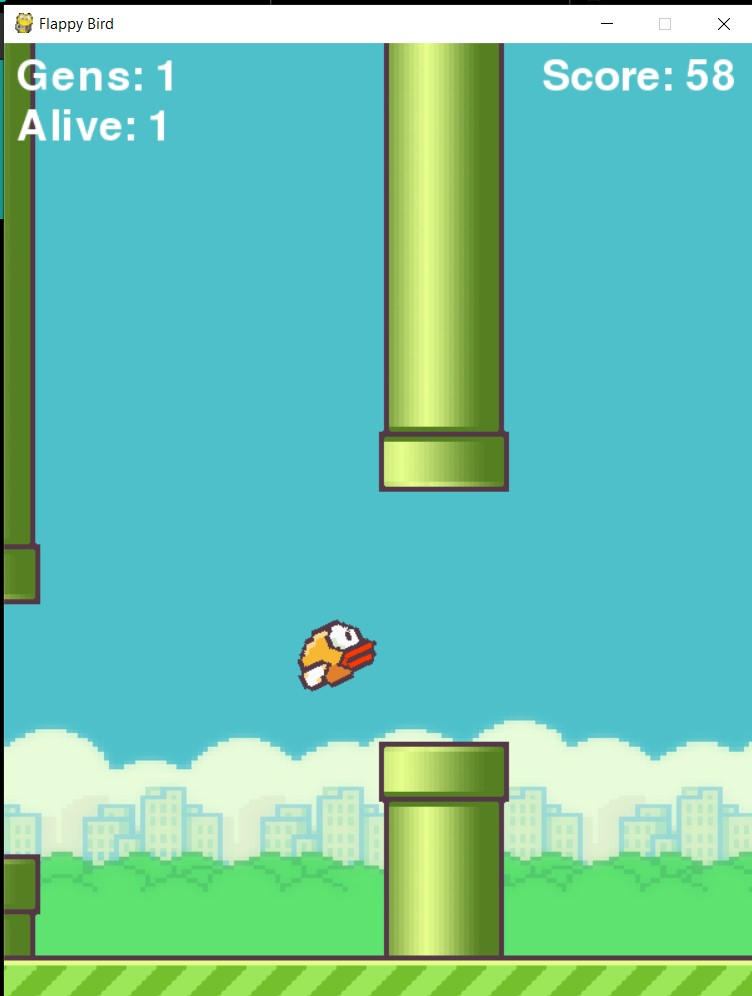
**Advantages**

* Storing information on the entire network: Information such as in traditional programming is stored on the entire network, not on a database. The disappearance of a few pieces of information in one place does not restrict the network from functioning.
* The ability to work with inadequate knowledge: After ANN training, the data may produce output even with incomplete information. The lack of performance here depends on the importance of the missing information.
* It has fault tolerance:  Corruption of one or more cells of ANN does not prevent it from generating output. This feature makes the networks fault-tolerant.
* Having a distributed memory: For ANN to be able to learn, it is necessary to determine the examples and to teach the network according to the desired output by showing these examples to the network. The network's progress is directly proportional to the selected instances, and if the event can not be shown to the network in all its aspects, the network can produce incorrect output
* Gradual corruption:  A network slows over time and undergoes relative degradation. The network problem does not immediately corrode.
* Ability to train machine: Artificial neural networks learn events and make decisions by commenting on similar events.
* Parallel processing ability:  Artificial neural networks have numerical strength that can perform more than one job at the same time.

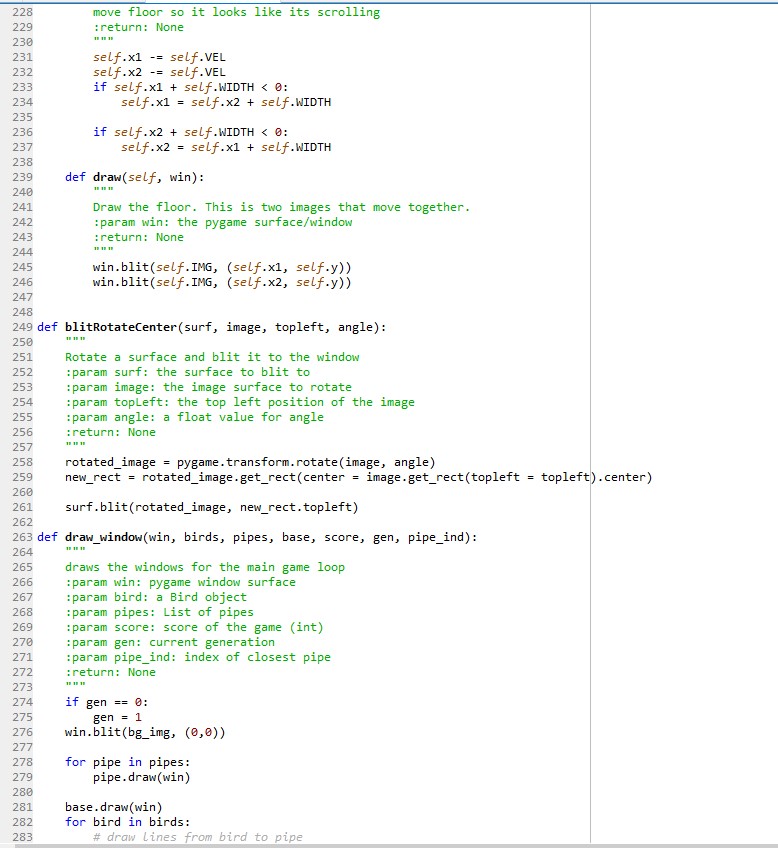
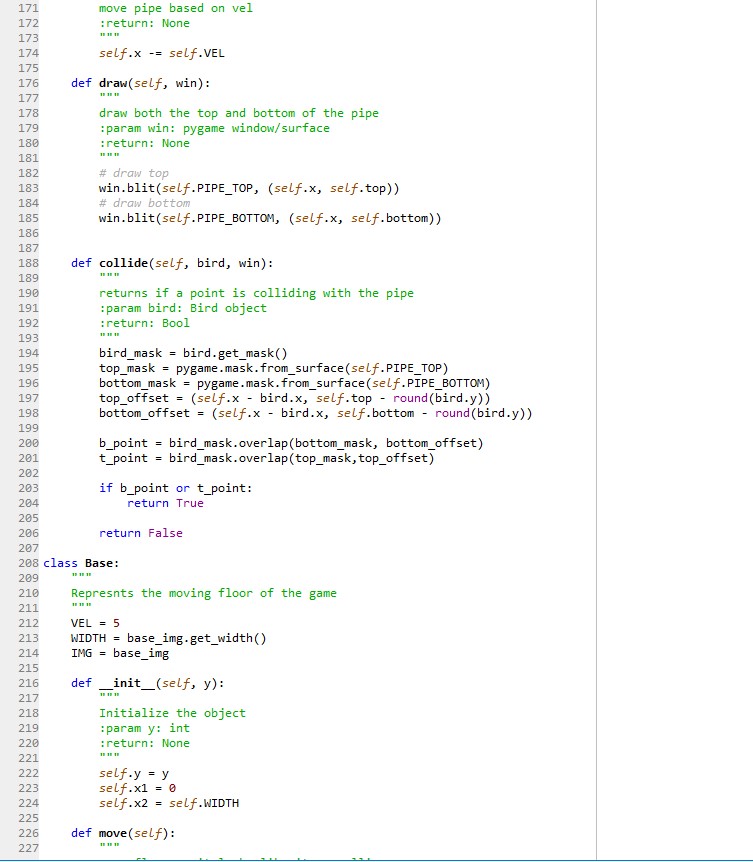
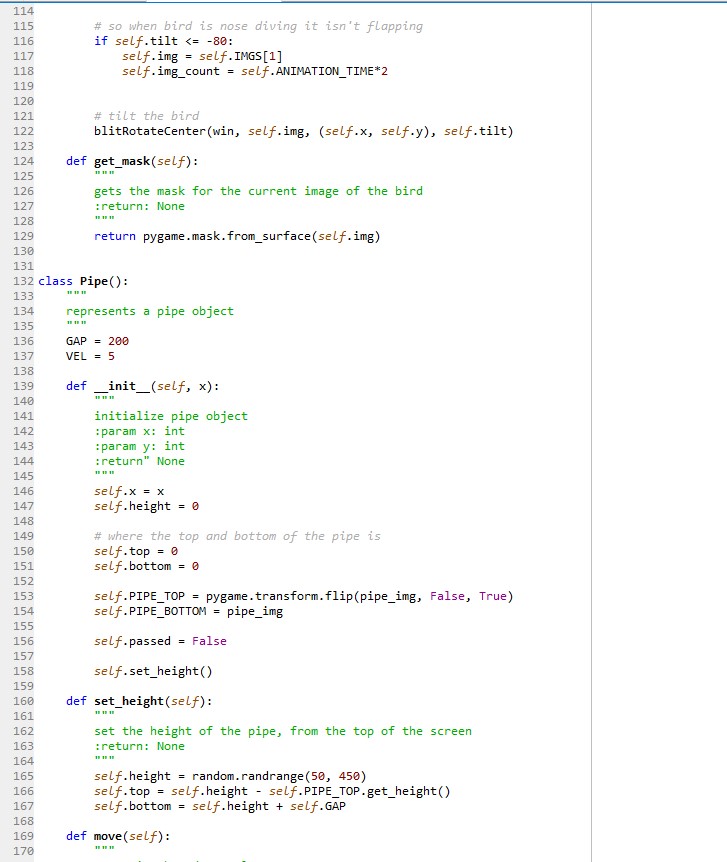
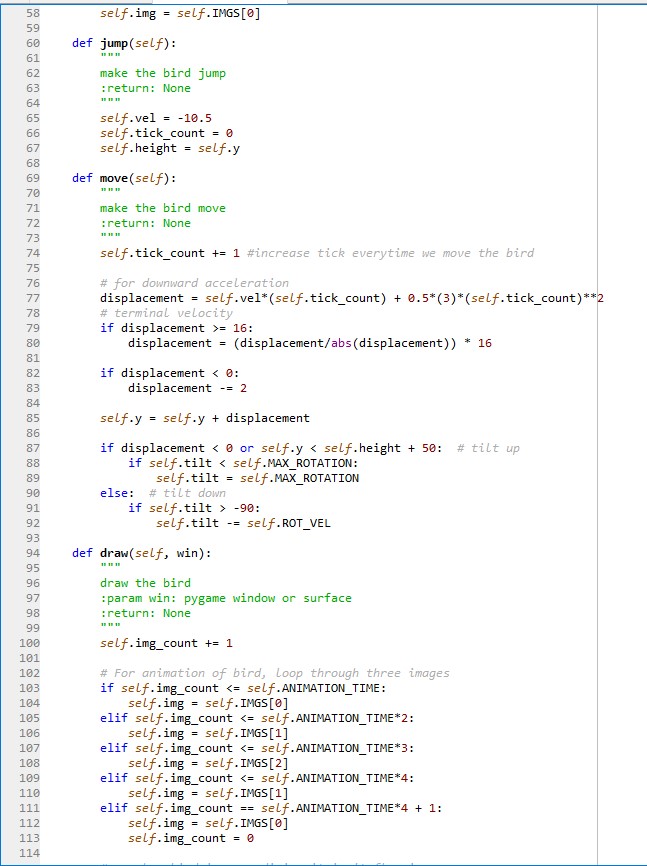
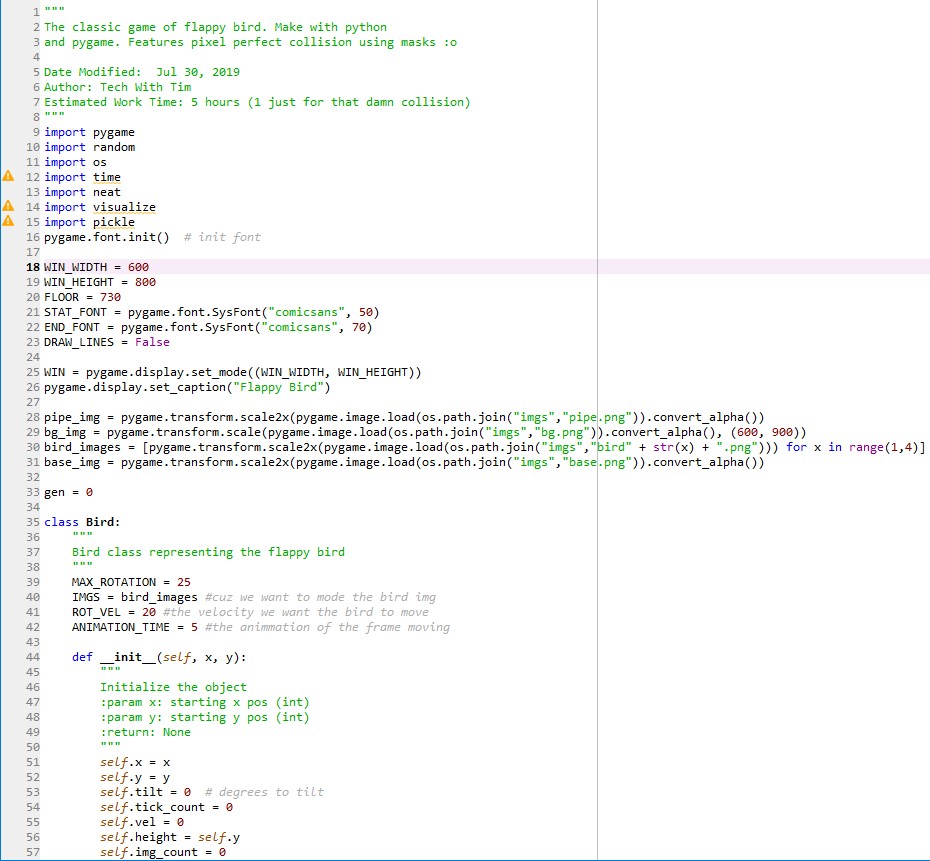
**Screenshots**

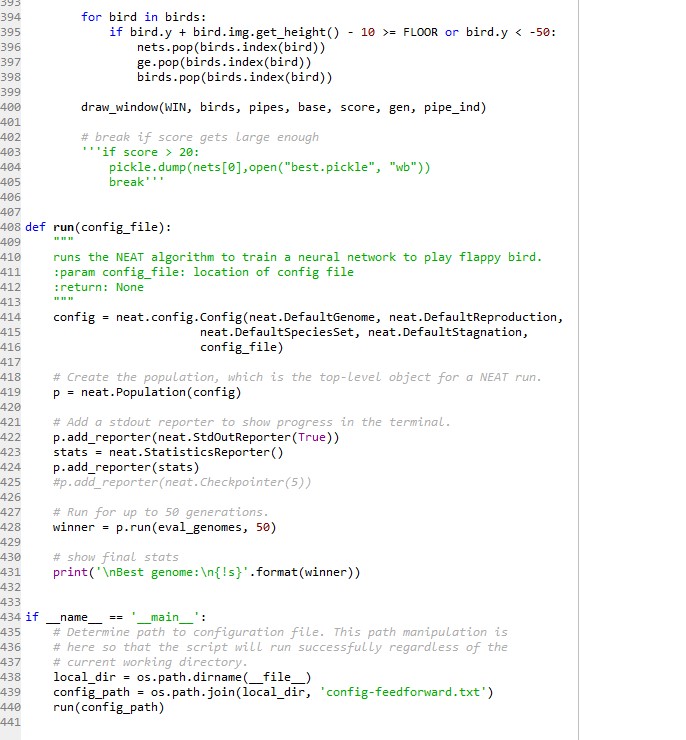
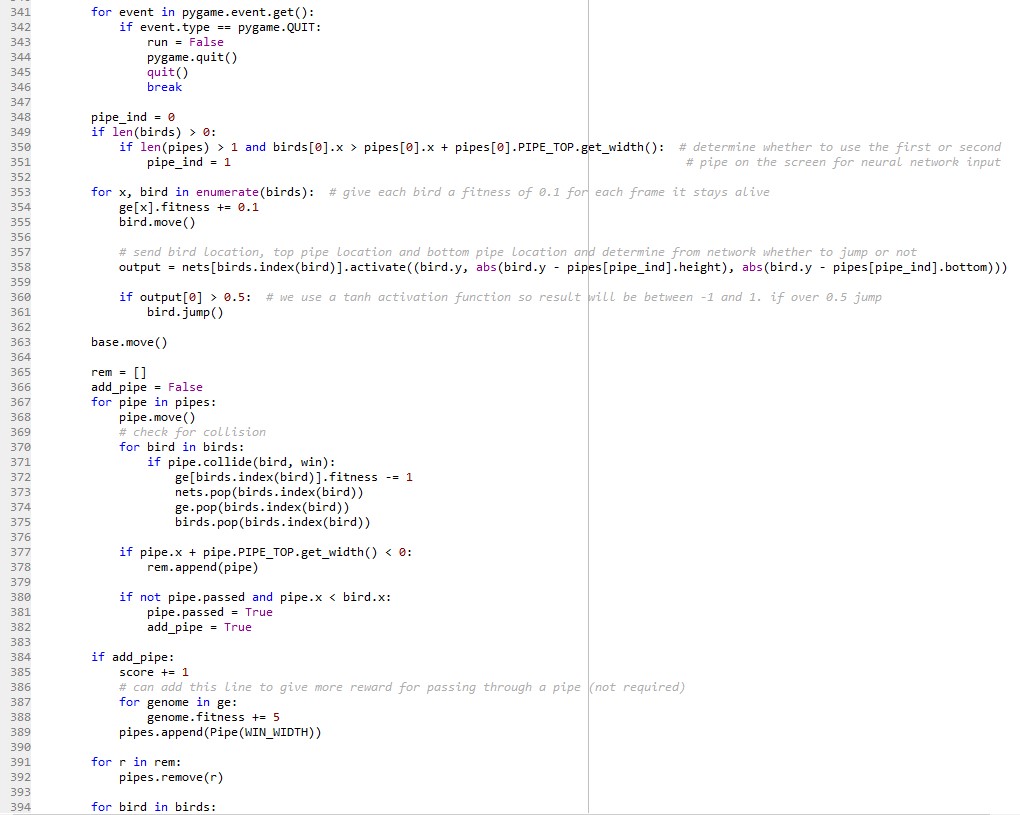
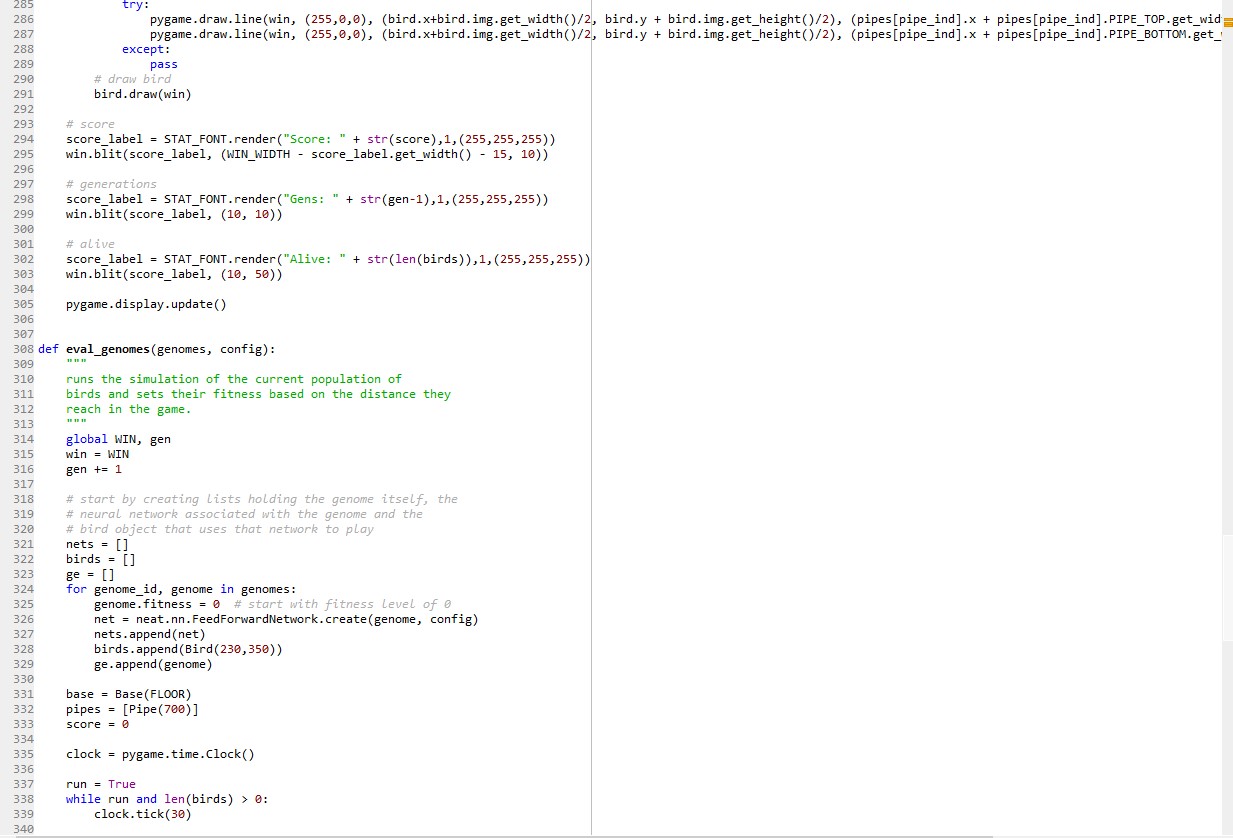
Flappy bird screenshots:





Source Code:





**Conclusion**

I felt great while working on this project. I was quite fascinated by the fact that such simple ”random” updates can develop such undefeatable game players. Also, the idea of evolution behind the Genetic Algorithms is also quite interesting.

Artificial Intelligence is now being widely used in a variety of businesses and games. Artificial Intelligence and the technology are one side of the life that always interest and surprise us with the new ideas, topics, innovations, products …etc. AI is still not implemented as the films representing it(i.e. intelligent robots), however there are many important tries to reach the level and to compete in market, like sometimes the robots that they show in TV. Nevertheless, the hidden projects and the development in industrial companies.

At the end, we’ve been in this research through the AI definitions, brief history, applications of AI in public, applications of AI in military, ethics of AI, and the three rules of robotics. This is not the end of AI, there is more to come from it, who knows what the AI can do for us in the future, maybe it will be a whole society of robots.

**References**

<http://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf>

<https://www.w3schools.com/python/>

<https://www.pygame.org/docs/>

<https://neat-python.readthedocs.io/en/latest/neat_overview.html>

<https://www.geeksforgeeks.org/game-playing-in-artificial-intelligence/>