Extended Essay in Computer Science

Personal Code: ***kdz007***

Session: ***May 2023***

Topic:

The ability of intelligent agents to operate in unfamiliar scenarios

Research Question:

"To what extent is the ability of an intelligent agent to operate in unfamiliar scenarios affected by the number of hidden-layer neurons in its neural network and the number of generations it is trained for?"

CS EE World
https://cseeworld.wixsite.com/home
May 2023
30/34
A

Submitter Info:
Name: Sachin Ramanathan
School: UC San Diego Class of 2027
Email: saramanathan [at] ucsd [dot] edu

Word Count: 3,983 words

Table of Contents

List Of Figures

1 Introduction

Genetic Algorithms are extremely useful in narrowing down a vast search space and finding near-optimal solutions quickly. They can be used even when there is a lack of mathematical representation for the problem being solved, which is why they are used in a variety of reinforcement learning applications. One such application is in creating intelligent agents to play games autonomously.

This helps us in our broader goals for AI as games are closed and controlled environments, free of real-world constraints, thus providing the perfect sandbox to gain a deep understanding of the behaviour of machine learning algorithms. They are quantifiable and provide us with numerical data (scores) to evaluate how our algorithm performs, which might be difficult or infeasible to gather in the real world.

They could also be used by game developers to understand their game's difficulty level and find bugs that could be exploited by players. For instance, an evolutionary algorithm was able to find two bugs in the game Qbert which allowed users to gain endless points (*Chrabaszcz et al. 1423*).

A neural network can be used as the "brain" of an intelligent agent, to evaluate the actions to be taken by the player based on its inputs. However, the parameters of neural networks must be trained for them to function effectively. Genetic algorithms can be used to optimally tune these parameters to create an efficient intelligent agent.

However, an agent that can only function in situations similar to the training environment has minimal real-life applications. Machine learning algorithms are often trained on a small subset of all the scenarios they encounter and are expected to generalise their learning to tackle new situations. This behaviour is known as transfer learning. The architecture of the neural network

and the training parameters govern how it learns and therefore can be varied to extend its capabilities.

Flappy Bird was chosen to be the game to evaluate this research as it is simplistic, allowing the configuration of the game to be easily changed. It circumvents the additional complexities associated with various inputs, numerous possible actions by the player, and complex relationships between them.

Therefore, the research question this paper seeks to answer is "***To what extent is the ability of an intelligent agent to operate in unfamiliar scenarios affected by the number of hidden-layer neurons in its neural network and the number of generations it is trained for?***"

2 Theoretical Background

2.1 Genetic Algorithms (GAs)

Genetic Algorithms are heuristic search algorithms which use the principles of natural evolution to identify optimal solutions for the given problem. They are particularly useful in solving problems with a large search space, lack of mathematical representation, or having a large number of parameters. They yield competent solutions in short durations of time, although usually not the best possible solution (*Wirsansky 9, 19*).

The generic pseudocode for GAs is as follows (*Mallawaarachchi*):

```
Generate the initial population
Compute fitness
REPEAT
   Selection
   Crossover
   Mutation
   Compute fitness
UNTIL population has converged
```

These function by maintaining a fixed-size population of individuals which are evaluated at every iteration. These individuals represent candidate solutions as **chromosomes**, which are arrays of **genes**. Each gene can be a binary digit, integer, or real number (*Refer to Figure 1*).

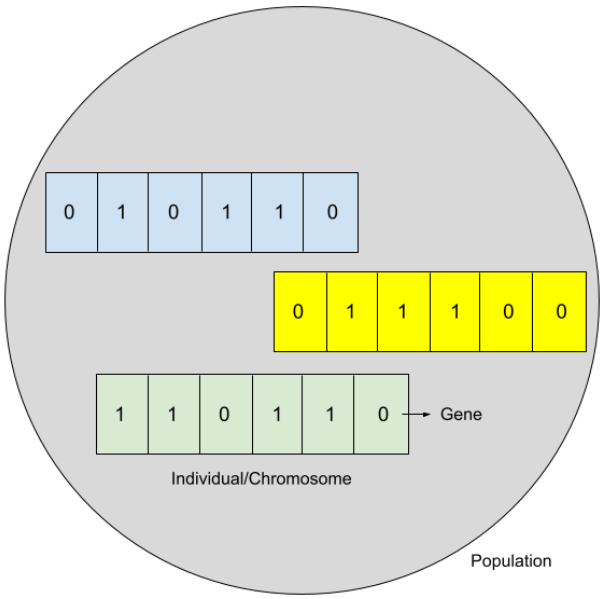

Figure 1. Population of individuals with binary-coded chromosomes.

(Made by Candidate using Google Drawings)

The **initial population** is generated by randomising the genes of each individual and is likely to contain inefficient solutions. Once the population is generated, a **fitness score**—a measure of the effectiveness of a candidate solution—is evaluated for each individual.

Weighted by their fitness scores, some individuals are chosen to advance to the next generation with no modification, as part of a process known as **selection**. The genes of a few individuals are **crossed over** or swapped between chromosomes to create new individuals inserted into the next generation. Finally, certain chromosomes are randomly reassigned genes in a process named **mutation** to introduce new genes into the population, thus preventing a homogenous aggregation of individuals. The chances that mutation or crossover occur in individuals are controlled by the specified **mutation rate** and **crossover rate**.

Once these three operations are conducted, the fitness of the population is computed again, and the process repeats until the population satisfies a predetermined stopping condition (*Wirsansky 11-13*).

2.2 Artificial Neural Networks (ANNs)

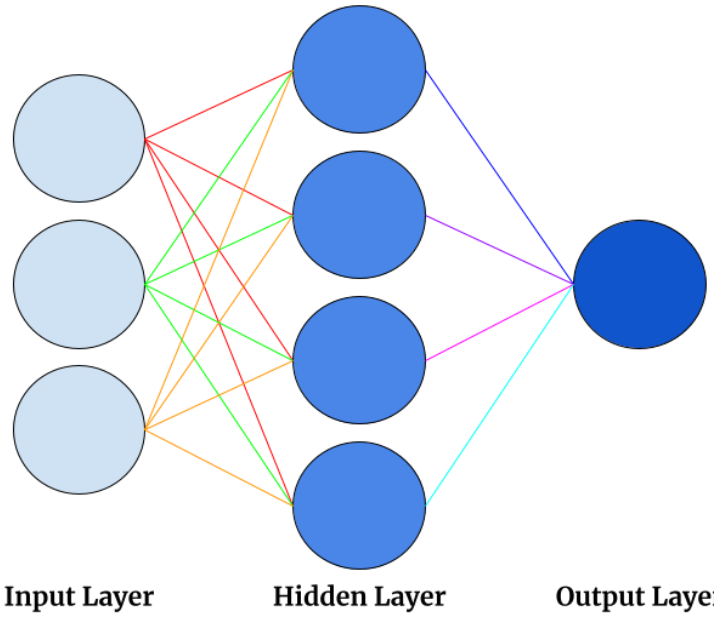

Figure 2. Simple ANN architecture with three inputs and one output, having a single hidden layer. (Made by Candidate using Google Drawings)

Artificial Neural Networks, shown in *Figure 2*, consist of three primary layers- the **input layer,** which takes in the data, the **output layer**, which forms the final output, and the intermediary **hidden layers** which can be as many as required. Each layer consists of multiple **neurons**, and each neuron is connected to every neuron in the next layer. In other words, the outputs of each neuron in an n^{th} layer are the inputs to every neuron in the $(n+1)^{th}$ layer (*Mueller and Massaron 274-275*).

The function of each neuron is to compute a weighted sum of its inputs using the **weights and biases** assigned to the neuron for each input during training, pass it through an **activation function** which transforms the output into a specific range, and output to the next layer (*Mueller and Massaron 272-273*).

The weights and biases of each neuron are known as the **parameters** of the ANN and determine the magnitude to which each input influences the output. These are tuned during the training process through methods such as stochastic gradient descent or using genetic algorithms.

The activation function, the number of neurons in each layer, and the network architecture are known as **hyperparameters** of the ANN, whose values are determined based on the application of the ANN.

2.3 Flappy Bird

Flappy Bird is a popular side-scroller arcade game, where players have to control a bird using flapping actions to fly through pipes without hitting the pipes, ground or ceiling. Therefore, the flaps have to be timed correctly so that the bird passes through the pipe since flapping is the only available method to control the vertical position of the bird. The game requires a sense of when to flap, which is acquired as players play the game. An intelligent agent will be used to do this artificially on a clone of the game.

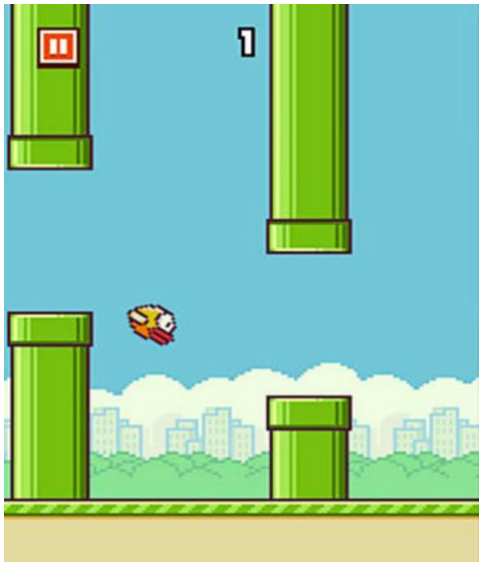

Figure 3. Screenshot of the original Flappy Bird game (Brustein).

3 Experimental Methodology

3.1 Tuning the parameters of an Artificial Neural Network using a Genetic Algorithm

For the ANN to produce an effective output, the weights and biases of each neuron has to be optimally tuned. Since we do not have a differentiable fitness function, which is a requirement to use stochastic gradient descent, we use a GA to tune the parameters instead (*Kwiatkowski*).

Each gene would represent either the weight or bias of a neuron and a chromosome would represent all the weights and biases of the network. The GA would be used to determine the optimal weights and biases for the ANN, and the ANN would be used to play the game once trained. The outcome of the game played using each individual's genes will enable the GA to evaluate its fitness score and judge its effectiveness relative to other individuals.

The GA is enumerated through numerous generations until the ANN reaches a predetermined condition of accuracy and consequently, the ANN is considered trained and can be used as an intelligent agent to play the game.

3.2 Architecture of the Artificial Neural Network controlling the bird

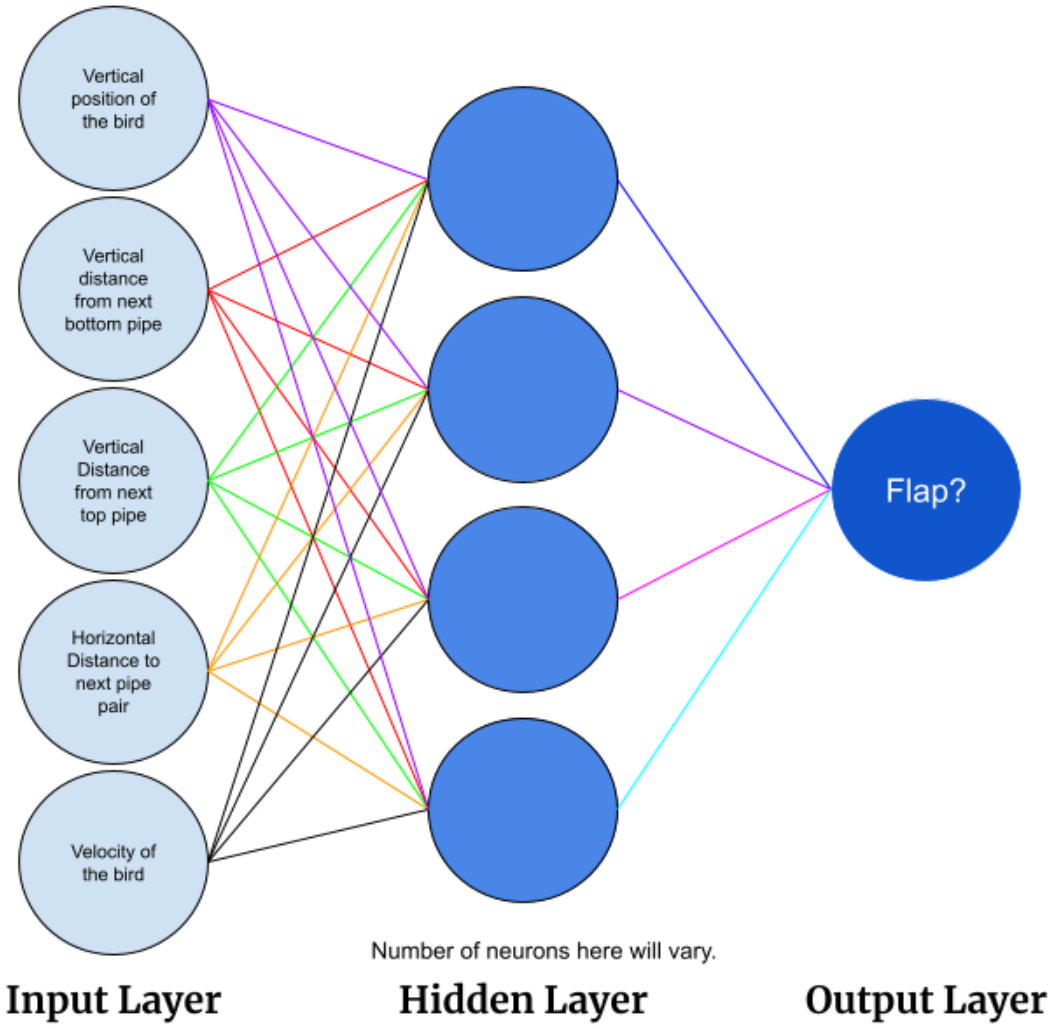

Figure 4. Architecture of the ANN used to control the actions of the bird (Candidate).

The ANN controlling the actions of the bird will take five inputs as shown in *Figure 4*, which correspond to the distances labelled in *Figure 5*.

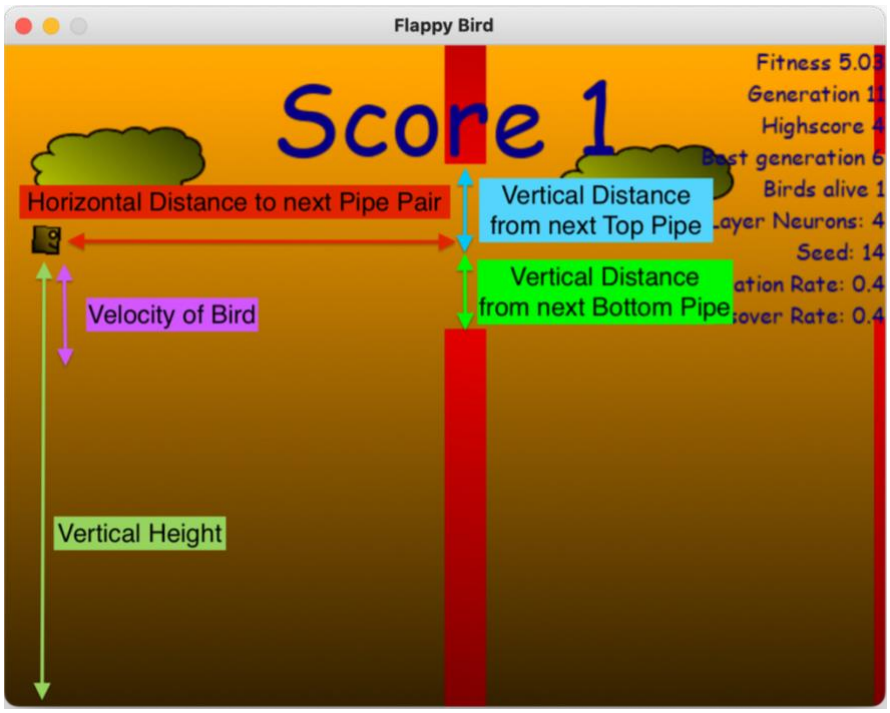

Figure 5. Annotated screenshot representing the inputs of the Artificial Neural Network (Candidate).

The sole output of the ANN will be a floating-point value between 0 and 1. If the $output > 0.5$, the bird will flap, and if $output <= 0.5$, no action will be taken. To prevent extremely large outputs, we use the sigmoid activation function, which takes in the output of the neuron and manipulates it to satisfy the constraint $0 < output < 1$.

Overfitting is when a machine learning model "memorises" how to react to the training data and is unable to replicate its ability during testing despite performing exceptionally with the training data because its output is completely based on its memorization (*Mueller and Massaron 161*). The network will only have one hidden layer to avoid overfitting since the scenario is not complex enough to justify the use of multiple hidden layers, which would also increase the time and resources required to train and execute the network.

3.3 Parameters of the Genetic Algorithm

The game score, which increases when an agent successfully passes through a pipe pair, will be weighted the highest in the fitness equation to reward the agent as maximising this value is our primary goal. However, the distance travelled by the bird will also be factored into the equation, although weighted lesser, to differentiate between birds with the same score but reaching a higher distance before colliding with an object. Hitting a pipe, the ground, or the ceiling result in an immediate termination of a game. Hence, we impose a penalty on these to reduce the fitness value to disincentivize the agent from repeating actions that lead to this state.

Therefore, the fitness of the bird is given by the equation:

$$Fitness = (3 \times Score) + (0.1 \times Distance\ travelled\ by\ bird) - Penalties$$

We will be having a **constant population size** of **90 individuals** for every generation, to avoid lag on the system as the individuals will be evaluated concurrently. **Elitism** will be implemented to preserve the best individuals in the population, although only one individual will remain unchanged across generations to allow for greater variance within the population.

The selection method being used will be **Rank-Based Selection**, where individuals are ordered by their fitness values, and the probabilities of each individual being selected are calculated based on their rank, instead of their raw fitness values. This selection method will ensure that some individuals with abnormally large fitness values will not overshadow the other individuals and dominate the population, instead providing a substantial opportunity for all individuals to be selected.

A normally distributed, also known as **Gaussian mutation**, will be used since the genes representing the weights and biases of the neurons are real numbers and Gaussian mutation is compatible with real-encoded genes. It generates a pseudorandom number following a normal

distribution (*Wirsansky 46*). An advantage of this method is that the generated genes are close to the original gene, preventing the pre-existing gene from being completely eliminated from the population. The **mutation rate** is set to a relatively high value here, **0.4**, to allow new genes to enter the population, thus preventing convergence at less-efficient local maxima.

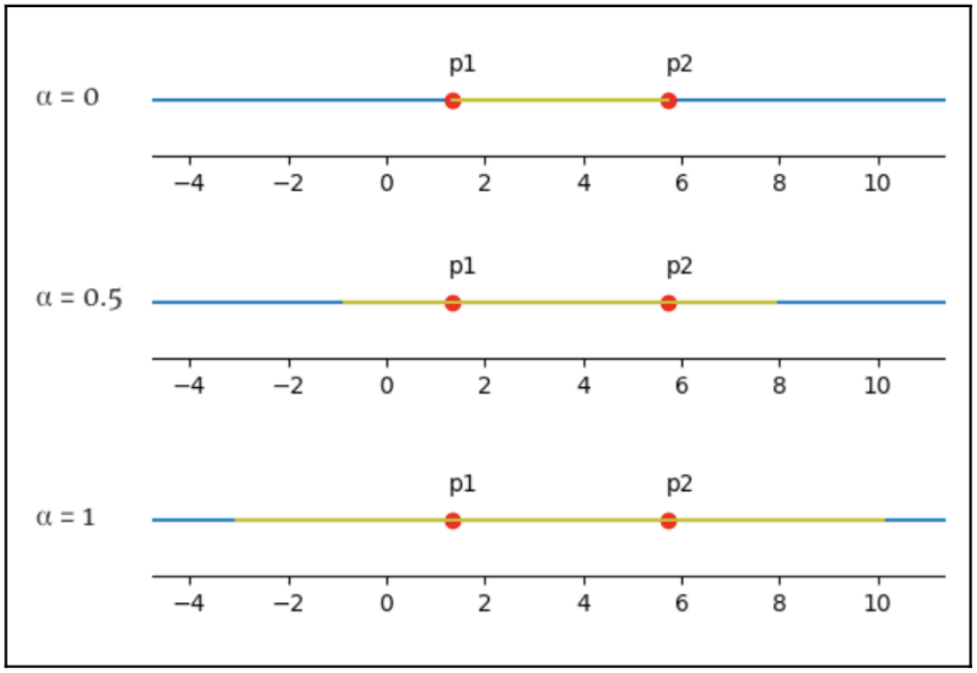

Figure 6. Visual representation of Blend Crossover (Wirsansky 43)

Blend Crossover will be used as the crossover method. Using this method, the resultant gene will be randomly generated from an interval derived from its parents' genes. The alpha (α) value is used to control how wide this interval is. The higher the alpha value, the wider the interval. *Figure 6* represents this, where p1 and p2 are the genes of the two parents. The green region represents the interval which the gene will be randomly chosen from. We will be using $\alpha = \mathbf{0.5}$ to allow for greater variance in the population (*Wirsansky 42)*. The **crossover rate** will also be set at **0.4** for the same reason, as it allows for more variance across generations.

3.4 Experimental Design

3 Trials will be conducted and averaged for each experiment with different **random seeds** (namely **14**, **4**, and **26**) to prevent the state of the randomly-generated initial population from skewing the results. The same 3 random seeds will be used for all the experiments for uniformity, since repeating the experiment specifying the same seed will yield identical results. A **cap of score 100** will be kept for each generation as without it, some excellent candidates will fly almost infinitely, blocking the program from advancing to the next generation.

3.5 Experimental Procedure

3.5.1 Experimental Setup

The algorithm will be trained on the original version of the game, and then tested on both the original version and altered version where the **height of the gaps between the pipes** is reduced from **120 units** which it was trained on to **110 and 105 units**, to examine how the intelligent agent functions. The **number of generations the GA is allowed to run for** and the **number of hidden-layer neurons** will be varied separately to observe their impact on its performance in the altered version of the game.

3.5.2 Experiment 1: Varying the number of neurons in the hidden layer

The neural network saved at generation 200 of each trial will be used to compare. The neural networks used will be from the same generation for all the trials. The 200^{th} generation was selected to ensure the individual being evaluated is already trained so the results are not skewed by a partially-trained agent.

3.5.3 Experiment 2: Varying the number of generations trained

Although the number of hidden-layer neurons is kept constant within an experiment, we will conduct three distinct experiments with different numbers of hidden-layer neurons—**one low (2)**, **one medium (15)**, and **one high (30)**—so that we can observe trends, if any, between the different neural network architectures in the effect of varying the number of generations trained. We start at the 150^{th} generation as all different hidden neuron configurations chosen have converged by this point.

4 Hypothesis

From my theoretical knowledge, I hypothesize that a higher number of hidden-layer neurons should result in the agent gaining a deeper learning of the game. Therefore, the agent is more likely to perform better when placed in different game settings. It will be able to model more complex relationships and may be more accurate in its outputs. However, the performance of the agent in both the original and the modified game settings may be adversely affected if the number of hidden-layer neurons is too high since overfitting may occur.

Similarly, training for more generations should provide more individuals the opportunity to gain optimal genes through mutation and crossover, and therefore make them more likely to be high-performing individuals. Hence, these agents should perform better in the modified game settings. Due to the implementation of elitism, it is unlikely that optimal genes will inadvertently be eliminated in future generations.

5 Experimental Results and Analysis

5.1 Experiment 1: The effect of varying the number of hidden-layer neurons

5.1.1 Experimental Data

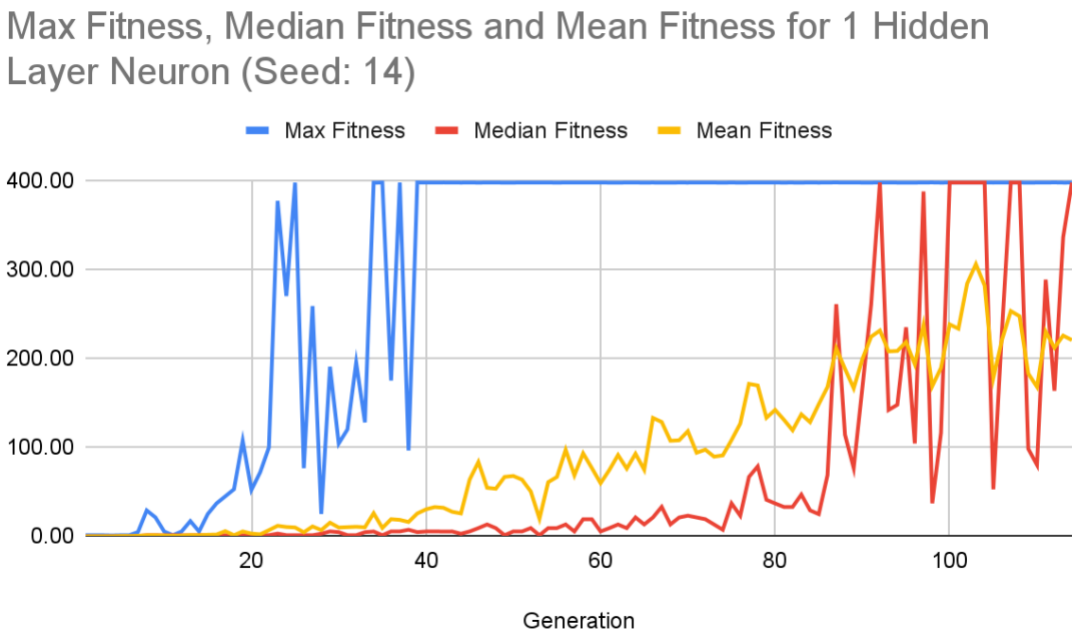

Figure 7. Variation of the fitness values of the population over generations (1 hidden layer neuron)[1]

In *Figure 7*, the max fitness shows the fitness of the best-performing individual in the population, and the median and mean fitness can be used to identify what proportion of the population is trained. Initially, the fitness values are clustered around zero. As mutation occurs, new genes are introduced into the population through a few individuals, shown by the observation that only the max fitness rises. This value fluctuates across generations, eventually reaches high

[1] Refer to Appendix 9.3.2 (page 52)

levels, signifying the tuning of the weights of the neural networks until it converges to a fitness value of approximately 400 which corresponds to a score of 100, which we limited the training until. Since we use elitism in the genetic algorithm to retain the best-performing bird without mutation, these genes are not eliminated and hence the max fitness curve remains constant from now.

The mean and median curves show slow but steady increases, which illustrates the rest of the population being optimised through random mutation and crossover with the top-performing individuals. The median fitness reaching 400 represents the top 50% of the population being trained. However, this curve fluctuates slightly in later generations as mutation and crossover can cause the genes of optimal individuals to be overwritten, thus explaining the need to preserve optimal genes through elitism.

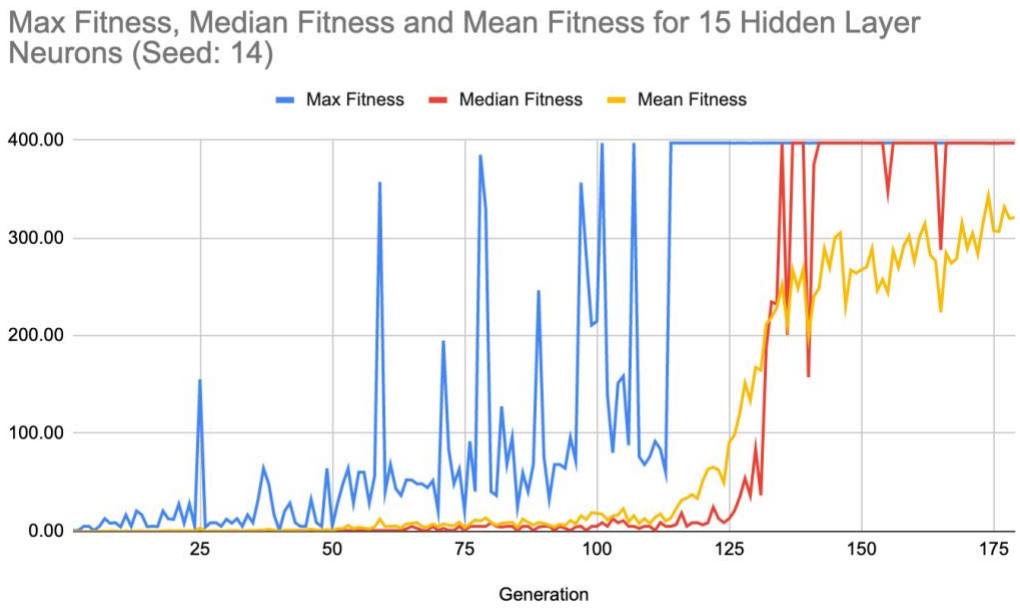

Figure 8. Variation of the fitness values of the population over generations (15 hidden-layer neurons)[2]

[2] Refer to Appendix 9.3.3 (page 57)

Comparing *Figures 7* and *8*, it is immediately noticeable that it takes more generations for the max fitness to converge with 15 hidden-layer neurons (114), compared to with 1 hidden layer neuron (39). The max fitness in *Figure 8* fluctuates considerably, with a generally increasing trend, until a sudden fluctuation causes it to converge. This could be because of a new set of genes introduced in the population through mutation, or through the crossover of two individuals creating an optimal individual. Similar to *Figure 7*, the mean and median fitnesses slowly approach the max fitness curve as the optimal genes spread across the population through crossover.

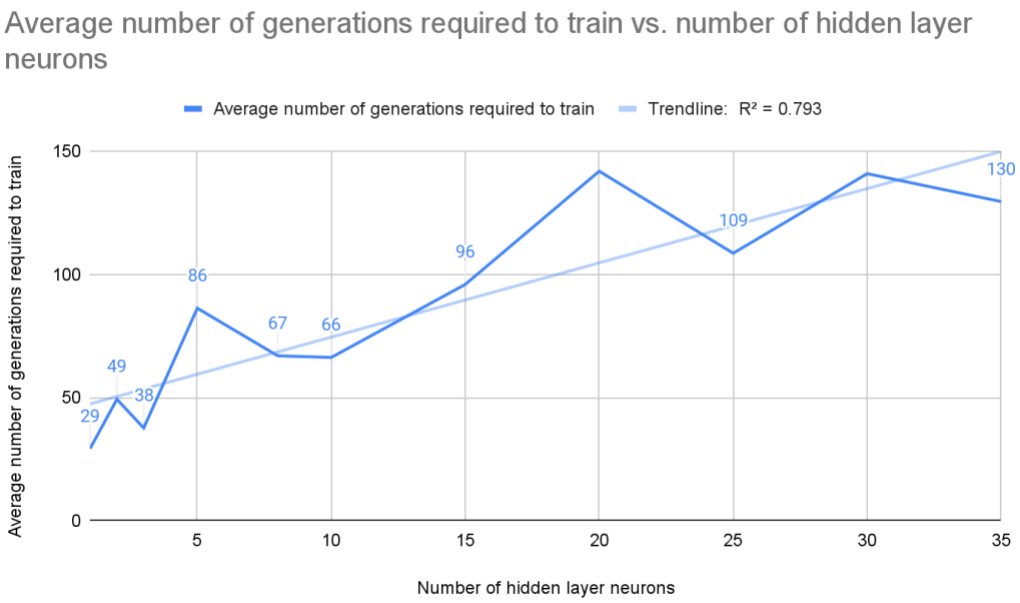

Figure 9. Average number of generations required to train the ANN against the number of hidden-layer neurons.[3]

Figure 9 represents the trend of how the average number of generations required to train the neural network is correlated with the number of hidden-layer neurons. The y-axis values are

[3] Refer to Appendix 9.3.1 (page 51)

obtained by observing when the max fitness curve converges to 400 for each number of hidden-layer neurons. There is evidently a broader trend where neural networks with more hidden-layer neurons take longer to train since there are more genes that need to be optimised, as shown by the trendline in *Figure 9*.

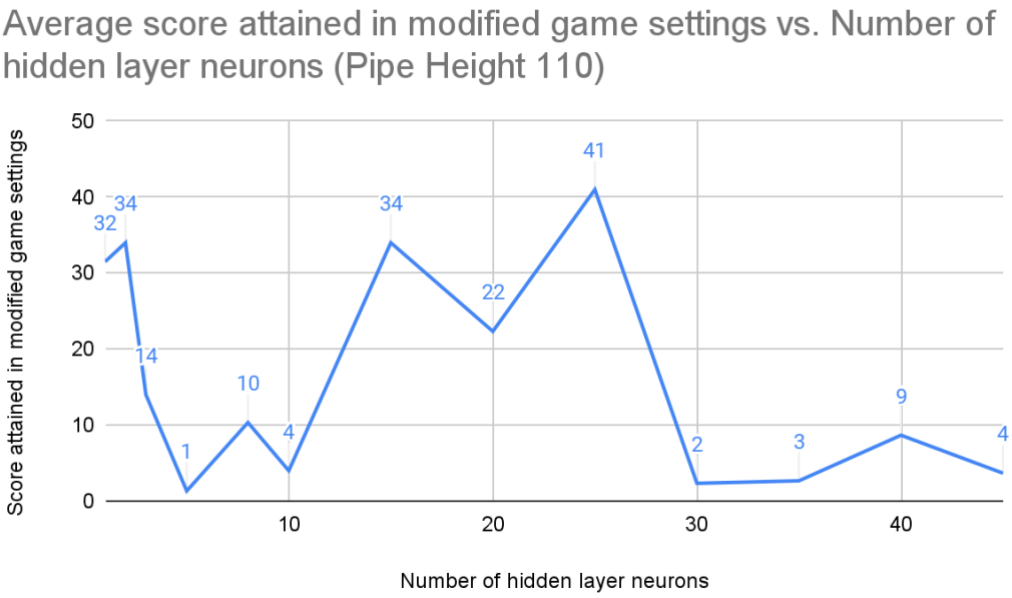

Figure 10. The correlation between the number of neurons in the hidden layer and the average score the agent attained in the modified game with a smaller pipe height of 110 units[4]

In *Figure 10*, we see that the score initially begins high, but suddenly drops and gradually increases again as the number of hidden-layer neurons increases, reaching a peak at 25 hidden-layer neurons, consequently falling rapidly again. This confirms our hypothesis that the score will increase as the number of hidden-layer neurons increases since the agent achieves deeper learning, but the scores remain low beyond 30 neurons.

[4] Refer to Appendix 9.3.4 (page 65)

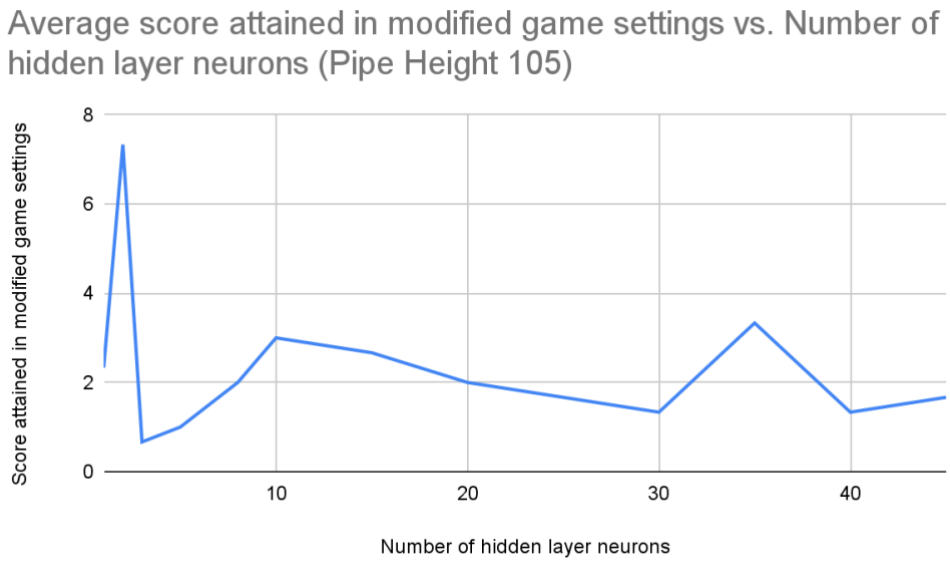

Figure 11. The correlation between the number of neurons in the hidden layer and the average score the agent attained in the modified game with a smaller pipe height of 105 units[5]

Figure 11 depicts the performance of the bird with the pipe height reduced further than in *Figure 10*. However, in the graph, we see no clear trend in scores. Therefore, the average scores are uniformly lower, reaching a maximum score of only 7, while in *Figure 10*, they reached a maximum average score of 41.

5.1.2 Result Analysis

It is clear from the collected data that the number of hidden-layer neurons corresponds to the number of generations required to train the network. This is because the number of possible combinations of neuron weights significantly rises with the number of neurons, thus complicating the process of finding an optimal configuration.

[5] Refer to Appendix 9.3.5 (page 65)

The initially high score with pipe height 110 occurs because the relationship between the inputs and the output is simple, allowing networks with lesser neurons to function effectively. However, this advantage disappears when the number of neurons increases. Following this, the upward trend validates our hypothesis, but the scores fall beyond 30 neurons as a result of overfitting—these networks memorised the original environment during training and therefore were unable to perform when the game settings are modified.

However, there is no clear trend in scores with pipe height 105 since the testing environment is significantly modified from the training environment and thus is too difficult for the agent to operate in. We observe that the greater the extent to which the environment is modified, the lower the effectiveness of the agent is.

5.1.3 Evaluation

The data from at least two of the three trials were often similar for each case, thus enabling us to validate the reliability of the data. The data from pipe height 105 enabled us to realise the limits of transfer learning while the data from pipe height 110 illustrated the optimal range of neuron configurations to utilise.

5.2 Experiment 2: The effect of varying the number of generations trained

5.2.1 Experimental Data

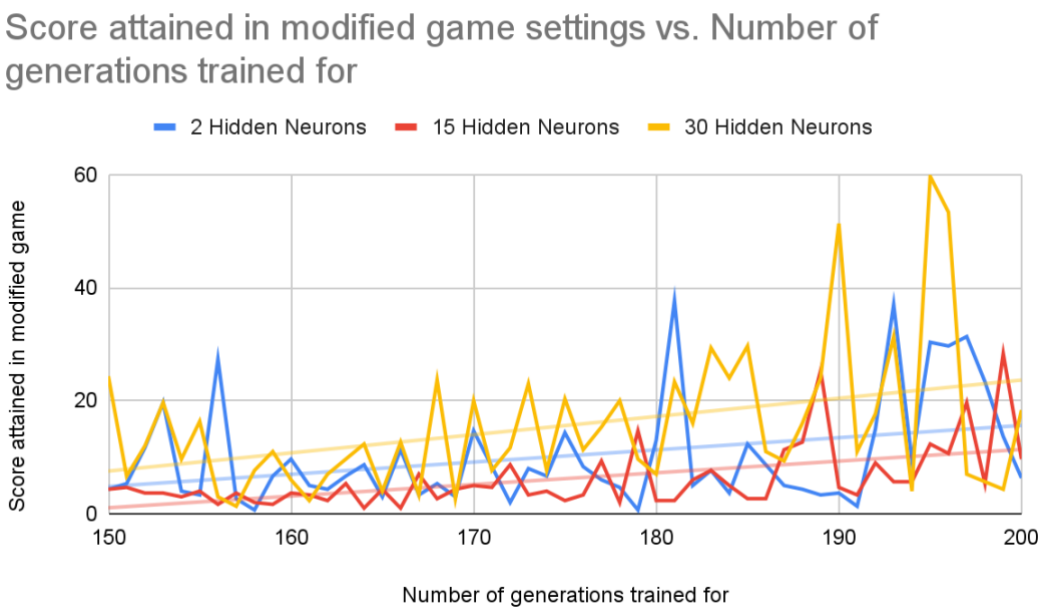

Figure 12. Graph of the scores attained in the modified game settings when the number of generations is varied from 150 to 200, with three different hidden neuron configurations[6]

In *Figure 12*, we see from the linear trendlines plotted that there is a universal tendency for the score to increase as we increase the number of generations trained, although the extent to which it varies differs. However, there are fluctuations due to large amounts of randomness propagating, despite the graph being plotted from an average of experiments using three different seeds to minimise this.

[6] Refer to Appendices 9.3.6, 9.3.7, 9.3.8 (pages 66, 68, and 71 respectively)

Although all three curves follow an increasing trend, there is a greater degree of fluctuation when 30 hidden-layer neurons are used, and the peaks are the tallest here. There are moderate fluctuations with 2 hidden-layer neurons and minimal fluctuations with 15 hidden-layer neurons.

5.2.2 Result Analysis

The score increases with the number of generations due to two reasons. Firstly, the optimal neurons crossover with other individuals to potentially form better individuals. Secondly, there is a higher chance that a mutation would provide optimal genes to a random individual to overtake the previous best individual.

The greater degree of fluctuation with 30 hidden-layer neurons suggests that there is greater instability in their genes, which is expected as there are more possible combinations of weights and biases. There is a moderate degree of fluctuation with 2 hidden-layer neurons due to the greater relative importance of each weight, once again causing instability. The network with 15 hidden-layer neurons shows minimal fluctuations due to striking a balance between these two behaviours.

5.2.3 Evaluation

Conducting the experiment with three different neuron configurations and attaining similar results indicating a direct correlation between the number of generations and the score validates our hypothesis for most ANNs. It also allows us to realise which situations the trade-off in training time is worth the gain in score resulting from better-optimised ANNs.

6 Limitations and Potential Improvements

In this investigation, only three trials were used and their results averaged, which meant that extremely-large or extremely-small anomalies skewed the resultant data significantly. To counter this, a higher number of trials could be conducted and their median used instead of their mean to curb the effect of outliers and ensure a more accurate result.

While the neural network saved at the 200^{th} generation from each trial was compared in this investigation, this provides a slight advantage to networks with lesser hidden-layer neurons as their scores converge to 100 earlier, thus allowing them more opportunities to enhance their performance. Furthermore, the score cap of 100 for each generation could be increased since the agent attains this by chance rather than by skill in some trials.

The mutation rate can be set lower to prevent useful genes from inadvertently being eliminated from the population. This, including increasing the number of individuals retained through elitism, would reduce the rapid fluctuations of the fitness in the graph, although the increase in score would take longer.

7 Conclusion

This investigation successfully explored how varying the number of hidden-layer neurons and the number of generations trained affects the ability of an agent to adapt to a modified environment.

Our hypothesis that the effectiveness of the agent in the modified environment improves as the number of hidden-layer neurons increases was true to some extent. Although this trend was upheld for small numbers, upon exceeding a certain limit, presumably determined by the complexity of the relation between the inputs and outputs, the effectiveness dropped due to overfitting. Furthermore, we observed that when the environment is modified to a greater extent, there is no clear trend since the effectiveness is low for all hidden-layer configurations.

However, we also observed that as the number of hidden-layer neurons increases, the training time also increases multifold. This limitation could offset any potential gain provided by the marginal increase in effectiveness for some use cases.

The second experiment validated our hypothesis that as the number of generations we train the neural network increases, its effectiveness in the modified environment improves. Yet, we noticed that the extent to which this matters is non-constant and varies based on the neuron configuration.

However, one significant limitation of this investigation was that only one game was used and the environment was varied only by switching the height of the pipe gaps. In the case of Flappy Bird, even one hidden-layer neuron is sufficient for the neural network to operate as the relationship between the inputs and the output is simple, which means that increasing the number of hidden-layer neurons makes it more complex and could lead to worse performance as well. However, this may not be the case for games with greater complexity, and therefore further

experimentation is required before this finding can be generalised. Using more methods to vary the environment would also allow us to be more certain of our results.

In this investigation, our inputs were limited to 5, and all the hidden-layer neurons were placed in one layer. Future exploration into how the quantity and nature of the inputs to the neural network or the number of hidden layers in the neural network affect its ability to perform in novel environments can prove beneficial.

Therefore, the research question "***To what extent is the ability of an intelligent agent to operate in unfamiliar scenarios affected by the number of hidden-layer neurons in its neural network and the number of generations it is trained for***" can be concluded with this investigation, which exhibits that the hidden-layer neurons and the number of generations do affect the effectiveness of a neural network in new environments, although the degree of correlation varies in certain cases.

8 Bibliography

Works Cited

Brustein, Joshua. “The Mysteries of Apps: Flappy Bird Shows That Dumb Luck Matters.” *Bloomberg.com*, 7 February 2014, https://www.bloomberg.com/news/articles/2014-02-07/the-mysteries-of-apps-flappy-bird-shows-that-dumb-luck-matters. Accessed 14 September 2022.

Chrabaszcz, Patryk, et al. “Back to Basics: Benchmarking Canonical Evolution Strategies for Playing Atari.” 2018, pp. 1419–1426, https://www.ijcai.org/proceedings/2018/0197.pdf. Accessed 20 June 2022.

Kwiatkowski, Robert. “Gradient Descent Algorithm — a deep dive.” *Towards Data Science*, 22 May 2021, https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21. Accessed 18 January 2023.

Mallawaarachchi, Vijini. “Introduction to Genetic Algorithms — Including Example Code.” *Towards Data Science*, 2017, https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3. Accessed 7 June 2022.

Mueller, John Paul, and Luca Massaron. *Machine Learning For Dummies*. Wiley, 2021. Accessed 12 June 2022.

Wirsansky, Eyal. *Hands-on Genetic Algorithms with Python: Applying Genetic Algorithms to Solve Real-world Deep Learning and Artificial Intelligence Problems*. Packt Publishing Limited, 2020, https://www.packtpub.com/product/hands-on-genetic-algorithms-with-python/9781838557744. Accessed 26 May 2022.

9 Appendix

9.1 Screenshot of conducting the experiment

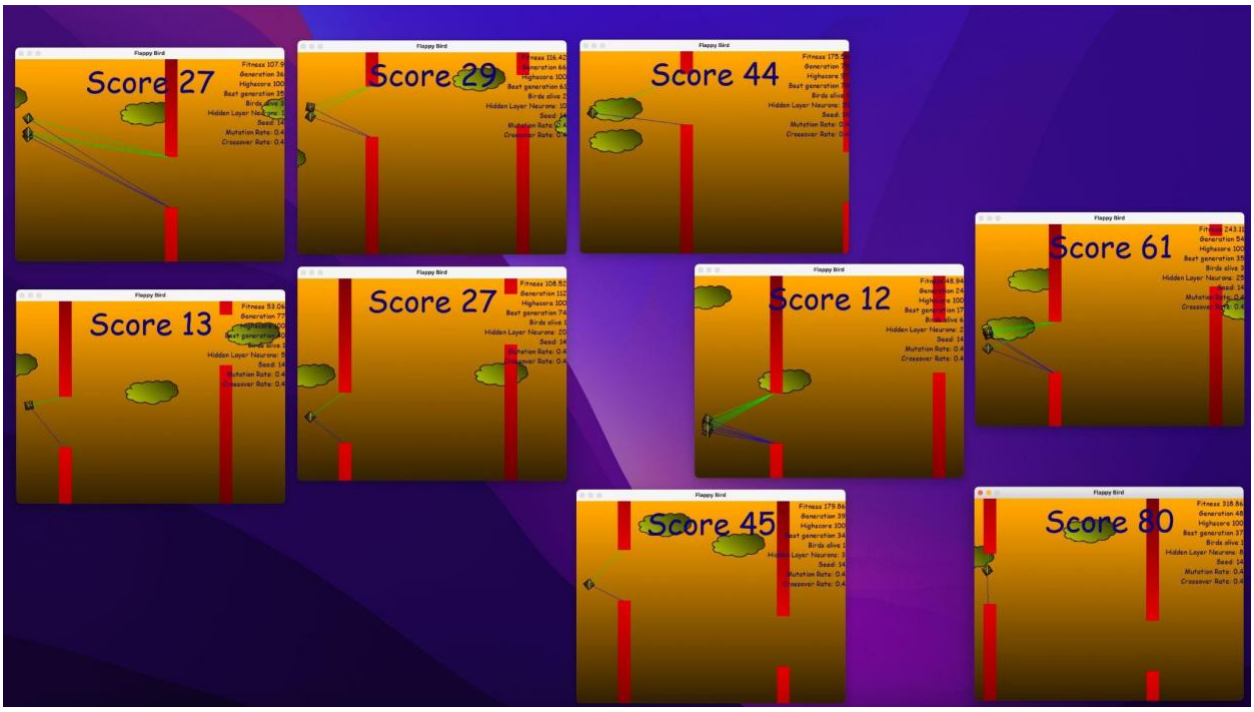

Figure 13. Screenshot taken while conducting the experiment (Candidate).

9.2 Codebase used for experiment

(Adapted from https://github.com/cr00nkz/Sloppy-Block under the GPL3 licence by modifying the genetic algorithm and neural network, modifying the game code, and including logging of readings)

main.py

```
# System / External Library Imports
import csv
import sys
```

```
import copy
import pygame
import random
import numpy as np
from time import time

# Local imports
import bird
from pipe import Pipe
import cloud

# Initialize constants
WIDTH = 640  # Game window width
HEIGHT = 480  # Game window height
BLOCKSIZE = 20  # Size for bounding box
VELOCITYGAIN = -13  # Downward acceleration of bird

#  Game settings
birdView = False  # Set to false, if you don't want to see what the birds see
HIGHDETAILS = True  # Set to false for efficiently training.
FPSSES = 1200  # Frames per second
PIPEHEIGHT = 120  # Initially 120 (trained on this) but changed to 120 for
new environment
SCORE_CAP = 100  # Usually after a score of 100 the agent will go on playing,
so we stop the execution here.
GENERATION_LIMIT = 200  # Usually after 200 generations the fitness values
stagnate and further progress is minimal.

# Genetic algorithm parameters
```

```
BIRDS = 90  # Initial birds population size
MUTATION_RATE = 0.4
CROSSOVER_RATE = 0.4
ELITISM = True
SEED = None  # None (to prompt for the seed) or an integer (for preset seed)


# Execution mode
EVAL_MODE = False  # Whether you are evaluating a neural network architecture
rather than training it
GEN_MODE = False  # Whether number of generations experiment or number of
hidden layers experiment is taking place


# Setting Seeds for Random
if not SEED:  # Random seed
   SEED = int(input("Please enter seed: "))  # 14,4,26
random.seed(SEED)
np.random.seed(SEED)
bird.random.seed(SEED)
bird.np.random.seed(SEED)


# Create csv files and enter headers
run_time = int(time())
if EVAL_MODE:
   with open(f"evaldata{run_time}.csv", "a") as f:
       writer = csv.writer(f, delimiter=',')
       writer.writerow(
           ["Time", "Seed", "Number of hidden layer neurons", "Generation",
"Pipe Height", "Score", "Best Fitness",
            "Median Fitness", "Mean Fitness"])
```

```
elif GEN_MODE:
    with open(f"evaldata{run_time}.csv", "a") as f:
        writer = csv.writer(f, delimiter=',')
        writer.writerow(
                ["Time", "Seed", "Number of hidden layer neurons",
"Generation", "Pipe Height", "Score", "Best Fitness",
                 "Median Fitness", "Mean Fitness"])
else:
    with open(f"data{run_time}.csv", "w") as f:
        writer = csv.writer(f, delimiter=',')
        writer.writerow(
            ["Time", "Seed", "Number of hidden layer neurons", "Generation",
"Score", "Best Fitness", "Median Fitness",
             "Mean Fitness"])
    with open(f"players{run_time}.csv", "w") as f:
        writer = csv.writer(f, delimiter=',')
        writer.writerow(["Time", "Seed", "Number of hidden layer neurons",
"Generation", "Score", "Best Input Weights",
                         "Best Hidden Weights"])


# pygame initialization
pygame.init()
fps = pygame.time.Clock()
window = pygame.display.set_mode((WIDTH, HEIGHT), 0, 32)
pygame.display.set_caption('Flappy Bird')
blockPic = pygame.image.load("./img/block.png")
upperPipePic = pygame.image.load("./img/upperPipe.png")
lowerPipePic = pygame.image.load("./img/lowerPipe.png")
backgroundPic = pygame.image.load("./img/background.png")
```

```
cloudPic = pygame.image.load("./img/cloud.png")
pygame.display.set_icon(blockPic)  # set Icon


# GlobalVariable Setup


player = None
multiPlayer = []
pipes = []
clouds = []
score = 0
running = True
font = pygame.font.SysFont("comicsansms", 72)
littlefont = pygame.font.SysFont("comicsansms", 16)
generation = 1
birdsToBreed = []
highscore = 0
highgen = 0
allTimeBestBird = None
maxscore = 0
singlePlayer = None
globalFitness = 0.0
respawn = False



def init():
    """ Initialise the game. """
    global player, running, score, multiPlayer, singlePlayer, respawn


    # Initialize Pipes
```

```
    while len(pipes) > 0:  # Reset existing pipes
        pipes.pop(0)
    init_pipe()
    init_pipe(w=WIDTH + WIDTH / 2)


    # Initialize clouds
    while len(clouds) > 0:  # Reset existing clouds
        clouds.pop(0)
    init_cloud(w=0)
    init_cloud(w=WIDTH / 2)
    init_cloud(w=WIDTH)


    # Reset some global variables
    score = 0
    running = True


    if EVAL_MODE or GEN_MODE:
        agent = bird.Bird(HEIGHT, num=len(evalHiddenWeights))
        agent.setWeights(evalInputWeights, evalHiddenWeights)
        multiPlayer = [agent]
        return
    else:
        singlePlayer = bird.Bird(HEIGHT, num=num)
        if len(birdsToBreed) == 0:  # This is the first init.
            for _ in range(BIRDS):
                multiPlayer.append(bird.Bird(HEIGHT, num=num))
        else:
            multiPlayer = []
```

```
            if ELITISM:  # Keep the best bird of generation without mutation
                _ = bird.Bird(HEIGHT, num=num)
                _.setWeights(birdsToBreed[0].inputWeights,
                             birdsToBreed[0].hiddenWeights)
                multiPlayer.append(_)

            # Rank-Based Selection
            sortedPlayers = sorted(birdsToBreed, key=lambda player:
player.fitness)
            totRanks = len(sortedPlayers) * (len(sortedPlayers) + 1) / 2  #
Sum of first N natural numbers formula
            wheel = {"players": [], "probabilities": []}

            for rank, player in enumerate(sortedPlayers, start=1):
                probability = rank / totRanks  # Proportion of the wheel
represented by this player.
                wheel['players'].append(player)
                wheel['probabilities'].append(probability)
            selected_players = np.random.choice(wheel['players'], BIRDS -
len(multiPlayer), p=wheel['probabilities'])
            new_players = []
            for player in selected_players:
                # Blend Crossover
                if random.random() > CROSSOVER_RATE:
                    player = bird.Bird(HEIGHT, player,
np.random.choice(selected_players), num=num)
                else:
                    player = bird.Bird(HEIGHT, player, num=num)
                    # Gaussian Mutation
```

```
                    if random.random() > MUTATION_RATE:
                        player.mutate()
                new_players.append(player)

            multiPlayer.extend(new_players)



def init_pipe(w=WIDTH):
    """Initializes a pipe, which will scroll in from the right side of the
screen. """
    dist = 0
    for pipe in pipes:
        dist = pipe.x
    return pipes.append(Pipe(w, HEIGHT, dist, pipeheight=PIPEHEIGHT))



def init_cloud(w=WIDTH):
    """ Initializes a cloud, which will scroll in from the right side of the
screen. """
    return clouds.append(cloud.Cloud(w, HEIGHT))



def draw(window):
    """ Drawws the game output """

    if HIGHDETAILS:
        # print background
        window.blit(backgroundPic, (0, 0))
```

```
        # print clouds
        for c in clouds:
            window.blit(cloudPic, (c.x, c.y))

        # print pipes
        for pipe in pipes:
            window.blit(upperPipePic, (pipe.x, pipe.upper_y - HEIGHT - 160))
            window.blit(lowerPipePic, (pipe.x, pipe.lower_y))
    else:  # Low Detail mode.
        pygame.draw.rect(window, (0, 0, 0), (0, 0, WIDTH, HEIGHT))
        for pipe in pipes:
            pygame.draw.rect(window, (255, 0, 0), (pipe.x, 0, 30,
pipe.upper_y))
            pygame.draw.rect(window, (255, 0, 0), (pipe.x, pipe.lower_y, 30,
HEIGHT))

    drewBird = False
    for player in multiPlayer:
        if player.alive:
            if HIGHDETAILS:
                topleft = (BLOCKSIZE, player.y)
                rot = player.velocity * -5
                if rot < -90:
                    rot = -90
                rotated_block = pygame.transform.rotate(blockPic, rot)
                new_rect = rotated_block.get_rect(
                    center=blockPic.get_rect(topleft=topleft).center)

                window.blit(rotated_block, new_rect.topleft)
```

```
            if birdView:  # Draw what the birds can see
                pygame.draw.line(window, (0, 255, 0),
                                 (20 + BLOCKSIZE / 2, player.y + BLOCKSIZE
/ 2),
                                 (BLOCKSIZE / 2 + player.distanceX,
                                  player.y + player.distanceTop))
                pygame.draw.line(window, (0, 0, 255),
                                 (20 + BLOCKSIZE / 2, player.y + BLOCKSIZE
/ 2),
                                 (BLOCKSIZE / 2 + player.distanceX,
                                  player.y + player.distanceBot))
                pygame.draw.line(window, (255, 255, 255),
                                 (20 + BLOCKSIZE / 2, player.y + BLOCKSIZE
/ 2),
                                 (20 + BLOCKSIZE / 2,
                                  player.y + BLOCKSIZE / 2 +
player.velocity))
                pygame.draw.line(window, (255, 255, 255),
                                 (20 + BLOCKSIZE / 2, player.y + BLOCKSIZE
/ 2),
                                 (20 + BLOCKSIZE / 2,
                                  player.y + BLOCKSIZE / 2 -
player.velocity))
            elif (not HIGHDETAILS) and (not drewBird):
                # Low detail mode - just one bird to draw
                pygame.draw.rect(window, (0, 255, 0),
                                 (20, player.y, BLOCKSIZE, BLOCKSIZE))
                drewBird = True
```

```
def draw_text(alive, score, highscore, fitness=None, gen=None, maxGen=None,
              noAlive=None, FPS=None):
    """Draw text on the screen. """

    textColor = (0, 0, 128)
    if not HIGHDETAILS:
        textColor = (0, 128, 0)

    if alive:
        text = font.render("Score {}".format(score), True, textColor)
        window.blit(text, (WIDTH / 2 - text.get_width() // 2, 0))
        text = littlefont.render("Fitness {}".format(round(fitness, 2)), True,
                                 textColor)
        window.blit(text, (WIDTH - text.get_width(), 0))
        text = littlefont.render("Generation {}".format(gen), True,
                                 textColor)
        window.blit(text, (WIDTH - text.get_width(), text.get_height()))
        text = littlefont.render("Highscore {}".format(round(maxscore, 2)),
                                 True, textColor)
        window.blit(text, (WIDTH - text.get_width(), text.get_height() * 2))
        text = littlefont.render("Best generation {}".format(maxGen), True,
                                 textColor)
        window.blit(text, (WIDTH - text.get_width(), text.get_height() * 3))
        text = littlefont.render("Birds alive {}".format(noAlive), True,
                                 textColor)
        window.blit(text, (WIDTH - text.get_width(), text.get_height() * 4))
        text = littlefont.render("Hidden Layer Neurons: {}"
```

```
                                .format(num), True, textColor)
        window.blit(text, (WIDTH - text.get_width(), text.get_height() * 5))
        text = littlefont.render("Seed: {}".format(SEED), True, textColor)
        window.blit(text, (WIDTH - text.get_width(), text.get_height() * 6))
        text = littlefont.render("Mutation Rate: {}".format(MUTATION_RATE),
True, textColor)
        window.blit(text, (WIDTH - text.get_width(), text.get_height() * 7))
        text = littlefont.render("Crossover Rate: {}".format(CROSSOVER_RATE),
True, textColor)
        window.blit(text, (WIDTH - text.get_width(), text.get_height() * 8))
    else:
        text = font.render("Game over.", True, (128, 0, 0))
        window.blit(text, (WIDTH / 2 - text.get_width() // 2,
                           HEIGHT / 2 - text.get_height() // 2))
        text = font.render("Score {}".format(score), True, (128, 0, 0))
        window.blit(text, (WIDTH / 2 - text.get_width() // 2, 0))
        text = littlefont.render("Highscore {}".format(round(highscore, 2)),
                                 True, (128, 0, 0))
        window.blit(text, (WIDTH - text.get_width(), text.get_height() * 2))



if EVAL_MODE:
    filename = input("Enter path to players CSV file")  # CSV file where
player details are stored.


    with open(filename, "r") as f:
        reader = list(csv.reader(f, delimiter=','))
        num = reader[1][2]
        for row in reader[1:]:
```

```
            if str(row[3]) == "200":  # 200th generation
                evalInputWeights = eval(row[5].replace("array", "np.array"))
                evalHiddenWeights = eval(row[6].replace("array", "np.array"))
                break
elif GEN_MODE:
    filename = input("Enter path to players CSV file")  # CSV file where
player details are stored.
    eval_individuals = []

    hundreds_in_a_row = 0
    with open(filename, "r") as f:
        reader = list(csv.reader(f, delimiter=','))
        num = reader[1][2]
        for row in reader[1:]:
            if hundreds_in_a_row >= 3:  # Make sure population has converged
before evaluating it.
                generation = int(row[3])
                evalInputWeights = eval(row[5].replace("array", "np.array"))
                evalHiddenWeights = eval(row[6].replace("array", "np.array"))
                eval_individuals.append((generation,evalInputWeights,
evalHiddenWeights))
            if str(row[4]) == "100":
                hundreds_in_a_row += 1


else:
    num = int(input("Enter number of neurons in hidden layer:"))  # Number of
neurons in hidden layer
```

```
if not GEN_MODE:
   eval_individuals = [None]


for individual in eval_individuals:
   if individual:
       generation, evalInputWeights, evalHiddenWeights = individual


   init()
   while True:  # the game loop.
       draw(window)  # Draw the fancy things.
       currentfitness = 0.0


       for event in pygame.event.get():
           if event.type == pygame.QUIT:
               sys.exit()


       if running:  # At least one bird is alive


           # Move clouds
           for c in clouds:
               c.move_left()
               if c.x < -140:  # Cloud out of screen. Spawn new cloud.
                   init_cloud()
                   clouds.pop(0)


           # Pipe Handling including collision check
           for p in pipes:
               if p.x < -30:  # Pipe out of screen. Spawn new pipe.
                   pipes.pop(0)
```

```
                    init_pipe()
                    score += 1  # We passed a pipe
                    for player in multiPlayer:  # Reward living birds
                        if player.alive:
                            player.fitness += 3

                p.move_left()  # Move the pipe to the left

            noAlive = 0
            p = pipes[0]  # Closest pipe

            for player in multiPlayer:
                if player.alive:
                    if not (EVAL_MODE or GEN_MODE):
                        if score >= SCORE_CAP:
                            player.alive = False
                    player.velocity += 1

                    player.handleCollision(HEIGHT, BLOCKSIZE, p)  # Did the
bird collide with anything?
                    if player.alive:
                        player.y += player.velocity
                        noAlive += 1

                    # Update what the bird sees to make decisions
                    player.processBrain(p.upper_y, p.lower_y, p.x)
                    currentfitness = player.fitness
                    globalFitness = player.fitness
```

```
                    # Jump or not?
                    if player.thinkIfJump():
                        player.velocity = VELOCITYGAIN

            if noAlive == 0:
                running = False  # Game over.

            # Report new high score if previous high score is exceeded
            for i in range(len(multiPlayer)):
                player = multiPlayer[i]
                if (player.fitness > highscore) and (not player.bestReported):
                    player.bestReported = True

            # Draw score and information
            draw_text(alive=True, fitness=currentfitness, gen=generation,
                      maxGen=highgen, noAlive=noAlive, FPS=FPSSES,
score=score,
                      highscore=maxscore)

        else:  # Player is dead
            if EVAL_MODE or GEN_MODE:
                time_ = int(time())
                fitness_values = [player.fitness for player in multiPlayer]
                with open(f"evaldata{run_time}.csv", "a") as f:
                    writer = csv.writer(f, delimiter=',')
                    writer.writerow([time_, SEED, num, generation, PIPEHEIGHT,
score, round(max(fitness_values), 2),
                                     round(np.median(fitness_values), 2),
round(np.mean(fitness_values), 2)])
```

```
                break
            if (score > 0) or (maxscore > 0) or (globalFitness > 0.2):
                # Only if at least one bird made it through one pipe
                birdsToBreed = []
                for h in range(2):  # Best two birds are taken
                    bestBird = -1
                    bestFitness = -10
                    for i in range(len(multiPlayer)):  # Find the best bird
                        player = multiPlayer[i]
                        if player.fitness > bestFitness:
                            bestFitness = player.fitness
                            bestBird = i
                            if bestFitness >= highscore:
                                highscore = bestFitness
                    if (h == 1) and (bestFitness >= highscore):  # New best
performing bird
                        allTimeBestBird = multiPlayer[bestBird]
                        bestInputWeights = copy.deepcopy(
                            multiPlayer[bestBird].inputWeights)
                        bestHiddenWeights = copy.deepcopy(
                            multiPlayer[bestBird].hiddenWeights)
                        highscore = bestFitness
                        highgen = generation
                        maxscore = score

                for j in range(len(multiPlayer)):
                    birdsToBreed.append(copy.deepcopy(multiPlayer[j]))

            time_ = int(time())
```

```
            fitness_values = [player.fitness for player in multiPlayer]
            with open(f"data{run_time}.csv", "a") as f:
                writer = csv.writer(f, delimiter=',')
                writer.writerow([time_, SEED, num, generation, score,
max(fitness_values), np.median(fitness_values),
                                 np.mean(fitness_values)])
            with open(f"players{run_time}.csv", "a") as f:
                writer = csv.writer(f, delimiter=',')
                try:
                    writer.writerow([time_, SEED, num, generation, score,
repr(bestInputWeights), repr(bestHiddenWeights)])
                except:
                    writer.writerow([time_, SEED, num, generation, score,
repr(multiPlayer[0].inputWeights), repr(multiPlayer[0].hiddenWeights)])

            if generation == GENERATION_LIMIT:
                print("Completed.")
                break
            generation += 1
            init()  # Here we go again

        # pygame updates
        pygame.display.update()
        fps.tick(FPSSES)
```

bird.py

```
import numpy as np
import random
```

```
class Bird:
    """ Models a bird/player, its attributes and its methods """

    def __init__(self, height, parent1=None, parent2=None, num=3):
        """ Constructs a bird, with num as the number of hidden layer neurons.
"""

        self.bestReported = False
        self.y = height / 2
        self.velocity = 0
        self.distanceBot = 0
        self.distanceTop = 0
        self.distanceX = 0
        self.distanceGround = 0
        self.distanceCeil = 0
        self.fitness = 0
        self.alive = True

        if parent1 == None and parent2 == None:  # New Bird, no parents
            self.inputWeights = np.random.normal(0, scale=0.1, size=(5, num))
            self.hiddenWeights = np.random.normal(0, scale=0.1, size=(num, 1))
        elif parent1 != None and parent2 == None:  # Duplicate the single
parent
            self.inputWeights = parent1.inputWeights
            self.hiddenWeights = parent1.hiddenWeights
        else:  # Two parents - Crossover.
            self.inputWeights = np.random.normal(0, scale=0.1, size=(5, num))
```

```
            self.hiddenWeights = np.random.normal(0, scale=0.1, size=(num, 1))
            self.crossover(parent1, parent2)

    def processBrain(self, pipeUpperY, pipeLowerY, pipeDistance):
        """ Updates what the bird sees. """
        self.distanceTop = pipeUpperY - self.y
        self.distanceBot = pipeLowerY - self.y
        self.distanceX = pipeDistance
        self.fitness += 0.01

    def handleCollision(self, HEIGHT, BLOCKSIZE, pipe):
        """ Checks if the bird hits the upper bounds, lower bounds or a pipe.
"""
        # Check if player collided with upper or lower pipe
        if ((pipe.x >= 20) and (pipe.x <= 20 + BLOCKSIZE)) or ((pipe.x + 20 >=
20) and (pipe.x + 20 <= 20 + BLOCKSIZE)):
            if self.alive and ((self.y <= pipe.upper_y) or (self.y + BLOCKSIZE
>= pipe.lower_y)):
                self.alive = False
                self.fitness -= 1

        # Check if bird collided with the ground/ceiling
        if self.y + self.velocity > HEIGHT - BLOCKSIZE:  # LowerBounds
            self.y = HEIGHT - BLOCKSIZE
            self.alive = False
            self.fitness -= 1
        elif self.y + self.velocity < 1:  # UpperBounds
            self.y = 0
            self.velocity = 0
```

```
            self.alive = False
            self.fitness -= 1


    def thinkIfJump(self):
        """Forward pass through neural network, giving the decision if the
bird should jump. """
        X = [self.y, self.distanceBot, self.distanceTop, self.distanceX,
             self.velocity]
        hidden_layer_in = np.dot(X, self.inputWeights)
        hidden_layer_out = self.sigmoid(hidden_layer_in)
        output_layer_in = np.dot(hidden_layer_out, self.hiddenWeights)
        prediction = self.sigmoid(output_layer_in)

        if prediction > 0.5:
            return True
        else:
            return False


    def setWeights(self, inputWeights, hiddenWeights):
        """ Overwrites the current weights of the bird's neural network. """
        self.inputWeights = inputWeights
        self.hiddenWeights = hiddenWeights


    def crossover(self, male, female, alpha=0.5):
        """Generate a new neural network from two parent birds using Blend
Crossover """

        for i in range(len(self.inputWeights)):
            for j in range(len(self.inputWeights[i])):
```

```
                range_ = abs(female.inputWeights[i][j] -
male.inputWeights[i][j])
                lower = min(female.inputWeights[i][j],
male.inputWeights[i][j]) - alpha * range_
                upper = max(female.inputWeights[i][j],
male.inputWeights[i][j]) + alpha * range_
                self.inputWeights[i][j] = lower + random.random() * (upper -
lower)

        for i in range(len(self.hiddenWeights)):
            for j in range(len(self.hiddenWeights[i])):
                range_ = abs(female.hiddenWeights[i][j] -
male.hiddenWeights[i][j])
                lower = min(female.hiddenWeights[i][j],
male.hiddenWeights[i][j]) - alpha * range_
                upper = max(female.hiddenWeights[i][j],
male.hiddenWeights[i][j]) + alpha * range_
                self.hiddenWeights[i][j] = lower + random.random() * (upper -
lower)

    def mutate(self):
        """ Mutate the neural network by randomly changing the individual
weights """

        for i in range(len(self.inputWeights)):
            for j in range(len(self.inputWeights[i])):
                self.inputWeights[i][j] =
self.getMutatedGene(self.inputWeights[i][j])
        for i in range(len(self.hiddenWeights)):
```

```
            for j in range(len(self.hiddenWeights[i])):
                self.hiddenWeights[i][j] =
self.getMutatedGene(self.hiddenWeights[i][j])


    @staticmethod
    def sigmoid(x):
        """ The sigmoid activation function for the neural network """
        return 1 / (1 + np.exp(-x))


    @staticmethod
    def getMutatedGene(weight):
        """ Apply Gaussian Mutation to a single weight """


        learning_rate = random.randint(0, 25) * 0.005
        mutatedWeight = random.gauss(weight, learning_rate)


        return mutatedWeight
```

pipe.py

```
import random



class Pipe:
  """ Models an in-game pipe pair. """


  def __init__(self, screen_width, screen_height, distanceToOldPipe,
pipeheight):
```

```
        """The constructor of a pipe pair (the obstacle the bird has to fly
through). """
        top = random.randint(0, screen_height - 100)
        self.upper_y = random.randint(0, screen_height - 140)
        self.lower_y = self.upper_y + pipeheight

        # Randomize distance of pipes so the bird can learn better
        self.x = distanceToOldPipe / 4 + screen_width + random.randint(0, 15)

    def move_left(self):
        """ Move the pipe to the left when a frame is processed. """
        self.x -= 4
```

cloud.py

```
import random



class Cloud:
    """ Models an in-game cloud. """

    def __init__(self, WIDTH, HEIGHT):
        """ The constructor of a cloud """

        self.x = WIDTH + 140 + random.randint(0, 140)
        self.y = random.randint(0, int(HEIGHT/2))
        self.TICKLIMIT = 5
        self.moveTick = 0
```

```
def move_left(self):
    """ Move the cloud every five frames from right to left """
    if self.moveTick > self.TICKLIMIT:
        self.x -= 1


    self.moveTick += 1
```

9.3 Data Tables

9.3.1 Hidden-Layer Neurons vs Training Time

Number of hidden layer neurons	Average number of generations required to train		Seed 1: 4	Seed 2: 26	Seed 3: 30
1	29		26	25	37
2	49		44	45	59
3	38		26	46	41
5	86		77	68	114
8	67		67	81	53
10	66		46	53	100
15	96		79	66	143
20	142		109	161	156
25	109		75	150	101
30	141		162	94	167
35	130		167	111	111

9.3.2 Max, Median and Mean Fitness for 1 Hidden Layer Neuron (Seed: 14)

Generation	Max Fitness	Median Fitness	Mean Fitness
1	0.52	-0.79	-0.71
2	0.51	-0.79	-0.67
3	0.52	-0.78	-0.62
4	0.28	-0.78	-0.65
5	0.52	-0.78	-0.40
6	0.54	-0.78	-0.45
7	4.12	-0.29	-0.13
8	28.45	0.51	0.52
9	20.62	0.54	0.51
10	4.75	0.40	0.24
11	0.56	0.53	-0.04
12	4.81	0.36	0.14
13	16.45	0.50	0.74
14	4.73	0.51	0.48
15	24.48	0.51	1.04
16	36.37	0.50	1.30
17	44.28	0.53	5.13
18	52.26	-0.79	0.54
19	106.94	0.57	4.79
20	51.52	0.54	2.38
21	71.34	0.39	1.53

22	99.23	0.58	6.23
23	376.71	2.35	11.15
24	269.61	0.61	9.80
25	397.24	0.54	9.25
26	75.95	0.59	3.84
27	258.39	0.51	10.41
28	24.50	2.26	6.30
29	190.14	4.82	14.66
30	103.70	3.96	9.19
31	119.60	0.62	9.62
32	194.89	0.55	10.04
33	127.53	3.96	9.44
34	397.12	4.73	25.30
35	397.29	0.53	8.70
36	174.57	4.81	18.59
37	397.31	4.79	17.81
38	95.92	6.68	15.43
39	397.39	4.01	25.02
40	397.23	4.75	29.66
41	397.26	4.76	32.24
42	397.35	4.73	31.41
43	397.31	4.75	26.88
44	397.20	2.21	25.04

45	397.28	4.78	63.13
46	397.18	8.55	82.85
47	397.26	12.65	53.88
48	397.30	8.51	52.87
49	397.13	0.55	66.14
50	397.20	4.81	67.21
51	397.25	4.78	63.07
52	397.30	8.58	49.82
53	397.29	0.54	19.16
54	397.29	8.54	60.23
55	397.13	8.59	66.11
56	397.29	12.63	96.62
57	397.26	4.76	67.88
58	397.22	18.52	92.63
59	397.25	18.55	76.09
60	397.25	4.76	59.07
61	397.20	8.52	73.86
62	397.11	12.61	90.74
63	397.29	8.51	75.43
64	397.26	20.57	92.18
65	397.16	12.64	74.19
66	397.23	20.55	132.49
67	397.10	32.46	127.96

68	397.17	12.70	106.83
69	397.23	20.50	107.34
70	397.21	22.52	117.55
71	397.26	20.55	93.34
72	397.30	18.56	96.79
73	397.25	12.65	88.94
74	397.12	6.67	90.38
75	397.28	36.44	107.56
76	397.12	22.55	126.08
77	397.23	66.15	170.81
78	397.23	77.96	169.01
79	397.28	40.42	132.66
80	397.22	36.32	141.56
81	397.21	32.37	130.49
82	397.22	32.37	118.59
83	397.15	46.30	136.56
84	397.32	28.51	127.76
85	397.19	24.28	148.00
86	397.32	68.13	167.34
87	397.37	260.45	210.76
88	397.28	113.66	187.74
89	397.23	76.01	165.83
90	397.21	167.14	198.99

91	397.20	260.29	223.86
92	397.26	397.26	230.83
93	397.26	141.45	207.38
94	397.21	147.32	207.85
95	397.13	234.42	217.92
96	397.18	103.78	192.86
97	397.29	387.31	237.15
98	397.37	36.45	168.21
99	397.18	115.66	188.81
100	397.27	397.27	237.73
101	397.22	397.22	232.89
102	397.34	397.34	283.69
103	397.25	397.25	305.54
104	397.28	397.28	281.48
105	397.21	52.22	177.42
106	397.31	232.53	220.67
107	397.31	397.31	252.73
108	397.28	397.28	246.93
109	397.12	97.77	182.50
110	397.24	79.71	167.45
111	397.25	288.07	229.68
112	397.40	163.04	211.53
113	397.21	335.72	225.13

114	397.22	397.22	220.14

9.3.3 Max, Median and Mean Fitness for 15 Hidden Layer Neurons (Seed: 14)

Generation	Max Fitness	Median Fitness	Mean Fitness
1	0.52	-0.78	-0.65
2	0.51	-0.78	-0.64
3	4.73	-0.78	-0.54
4	4.82	-0.78	-0.50
5	0.51	-0.78	-0.52
6	4.90	-0.78	-0.38
7	12.64	-0.78	-0.28
8	7.99	-0.78	-0.15
9	8.56	-0.78	-0.17
10	4.15	-0.78	-0.52
11	16.42	-0.78	-0.08
12	4.73	-0.78	-0.46
13	20.57	-0.78	-0.02
14	16.57	-0.78	-0.03
15	4.53	-0.78	-0.37
16	4.73	-0.78	-0.18
17	4.63	-0.78	-0.24
18	20.49	-0.78	-0.01

19	12.50	-0.60	0.28
20	11.83	-0.78	0.05
21	27.64	-0.16	0.48
22	7.97	-0.78	-0.02
23	27.80	0.29	0.32
24	4.78	-0.78	-0.11
25	155.25	-0.78	3.04
26	3.91	-0.78	-0.30
27	8.55	-0.77	0.02
28	8.54	-0.76	0.03
29	4.83	-0.21	-0.09
30	12.07	-0.54	0.05
31	8.02	0.03	0.07
32	12.67	-0.13	0.41
33	4.79	-0.77	-0.19
34	16.51	-0.19	0.44
35	8.50	-0.02	0.44
36	32.43	0.53	0.61
37	64.10	0.54	0.85
38	48.23	0.53	1.80
39	16.57	0.53	0.68
40	0.59	-0.51	-0.17
41	20.54	0.51	0.74

42	28.49	0.51	0.70
43	8.56	0.43	1.00
44	4.75	0.51	0.65
45	4.74	0.53	0.87
46	32.41	0.52	1.88
47	8.52	0.54	0.78
48	4.75	0.53	0.75
49	64.16	0.50	1.51
50	4.75	0.51	0.39
51	28.49	0.52	2.43
52	48.26	0.52	2.37
53	64.08	0.51	5.86
54	28.48	0.52	2.40
55	60.14	0.54	3.76
56	60.15	0.52	3.04
57	28.51	0.54	2.17
58	56.22	0.59	4.19
59	357.39	0.53	12.03
60	40.31	0.54	4.66
61	68.05	0.57	4.75
62	43.33	0.58	5.15
63	36.39	0.50	3.02
64	52.33	2.22	7.36

65	52.24	4.74	7.71
66	48.28	2.73	8.73
67	48.20	0.60	4.35
68	44.39	0.52	3.90
69	51.50	4.75	7.20
70	20.51	0.63	4.50
71	194.94	2.66	7.33
72	83.86	0.61	5.91
73	48.30	0.61	5.37
74	63.39	4.77	9.08
75	20.60	0.56	3.53
76	91.87	4.74	7.21
77	40.41	4.83	11.18
78	385.08	4.77	10.39
79	329.66	4.74	13.28
80	40.44	8.52	9.08
81	36.36	4.78	5.80
82	127.66	4.02	7.87
83	68.07	4.78	8.53
84	95.86	4.76	8.81
85	24.53	0.62	3.70
86	59.42	4.82	12.29
87	40.32	4.79	8.93

88	68.08	0.60	6.10
89	246.47	4.11	8.77
90	76.01	4.75	7.72
91	32.47	4.77	5.98
92	68.13	0.55	4.14
93	68.10	4.76	7.01
94	64.20	2.69	5.58
95	95.83	0.60	10.86
96	72.00	4.12	7.36
97	356.78	4.81	15.53
98	281.47	0.63	11.75
99	210.76	4.82	18.86
100	214.64	4.79	18.01
101	397.18	8.52	17.27
102	139.36	4.76	11.35
103	80.03	12.62	15.43
104	151.32	8.50	16.55
105	158.52	10.61	22.80
106	87.83	4.78	9.33
107	397.29	4.79	15.68
108	76.08	2.66	7.86
109	68.05	4.81	12.68
110	76.03	4.74	7.83

111	91.79	0.59	13.79
112	83.95	8.55	17.56
113	60.11	4.76	10.23
114	397.23	4.80	13.49
115	397.29	6.66	23.09511111
116	397.2	18.56	31.514
117	397.19	4.84	33.77911111
118	397.23	8.49	37.30044444
119	397.19	8.53	33.41055556
120	397.2	6.35	51.43011111
121	397.32	8.525	63.45177778
122	397.17	24.5	65.30333333
123	397.28	12.685	62.49033333
124	397.23	8.6	49.53544444
125	397.21	12.64	90.668
126	397.12	20.56	98.02044444
127	397.27	34.48	120.0878889
128	397.24	53.82	151.6648889
129	397.12	36.38	133.5595556
130	397.3	84.04	167.3923333
131	397.14	36.44	164.6716667
132	397.31	187.035	211.6618889
133	397.27	234.54	218.6645556

134	397.23	232.19	228.5846667
135	397.21	397.21	251.8702222
136	397.18	200.815	210.685
137	397.34	397.34	267.2154444
138	397.32	397.32	248.1885556
139	397.25	397.25	268.2436667
140	397.16	157.265	199.856
141	397.19	374.82	240.1583333
142	397.32	397.32	248.7101111
143	397.29	397.29	289.1445556
144	397.28	397.28	269.571
145	397.21	397.21	300.5576667
146	397.18	397.18	305.3174444
147	397.21	397.21	230.4615556
148	397.3	397.3	267.1468889
149	397.27	397.27	264.0756667
150	397.22	397.22	267.5594444
151	397.22	397.22	270.3647778
152	397.15	397.15	289.0781111
153	397.18	397.18	246.3966667
154	397.18	397.18	257.4742222
155	397.39	347.76	244.0626667
156	397.26	397.26	286.964

157	397.21	397.21	269.8262222
158	397.25	397.25	291.7905556
159	397.2	397.2	302.0175556
160	397.2	397.2	275.9645556
161	397.27	397.27	301.334
162	397.2	397.2	314.7288889
163	397.25	397.25	282.5782222
164	397.21	397.21	276.6954444
165	397.24	287.72	223.9163333
166	397.15	397.15	284.5931111
167	397.27	397.27	274.158
168	397.27	397.27	278.7238889
169	397.26	397.26	315.345
170	397.26	397.26	289.3608889
171	397.24	397.24	304.7337778
172	397.31	397.31	283.4957778
173	397.24	397.24	315.8652222
174	397.16	397.16	343.1457778
175	397.13	397.13	307.2334444
176	397.13	397.13	306.6426667
177	397.28	397.28	331.5072222
178	397.2	397.2	319.8943333
179	397.24	397.24	321.0216667

9.3.4 Score attained in modified game settings when varying hidden layer neurons (Pipe Height 110)

Number of hidden layer neurons	Score attained in modified game settings		Seed 1: 4	Seed 2: 26	Seed 3: 30
1	32		16	464 (Anomaly)	47
2	34		5	67	30
3	14		6	0	36
5	1		3	0	1
8	10		9	21	1
10	4		5	3	4
15	34		4	34	64
20	22		22	24	21
25	41		78	42	3
30	2		2	0	5
35	3		8	0	0
40	9		23	0	3
45	4		8	2	1

9.3.5 Score attained in modified game settings when varying hidden layer neurons (Pipe Height 105)

Number of hidden layer neurons	Score attained in modified game settings		Seed 1: 4	Seed 2: 26	Seed 3: 30
1	2		2	1	4
2	7		2	1	19

3	1	0	0	2
5	1	2	0	1
8	2	4	1	1
10	3	4	3	2
15	3	4	0	4
20	2	1	0	5
25	2	5	0	0
30	1	3	0	1
35	3	3	0	7
40	1	3	0	1
45	2	3	1	1

9.3.6 Score attained in modified game settings when varying generations (2 Hidden Layer Neurons)

Number of generations trained for	Score attained in modified game settings		**Seed 1: 4**	**Seed 2: 26**	**Seed 3: 30**
150	4		1	12	0
151	5		1	7	8
152	12		11	20	4
153	20		10	48	1
154	4		3	8	1
155	3		2	8	0
156	27		0	63	19
157	3		0	1	7
158	1		1	1	0

159	7		0	14	6
160	10		8	15	6
161	5		2	5	8
162	4		2	5	6
163	7		2	14	4
164	9		0	26	0
165	3		1	8	0
166	11		1	29	4
167	3		3	4	3
168	5		10	0	6
169	3		2	7	0
170	15		2	20	22
171	8		1	23	1
172	2		0	5	1
173	8		2	17	5
174	7		3	16	1
175	14		2	37	4
176	8		0	24	1
177	6		1	9	8
178	5		7	3	4
179	1		2	0	0
180	13		3	34	2
181	38		4	105	4
182	5		3	9	3
183	8		0	22	1

184	4		3	6	2
185	12		3	33	1
186	9		11	14	1
187	5		2	12	1
188	4		1	10	2
189	3		4	2	4
190	4		4	5	2
191	1		1	3	0
192	15		0	29	16
193	37		0	5	106
194	9		8	5	15
195	30		3	25	63
196	30		5	21	63
197	31		0	11	83
198	23		1	1	68
199	14		2	9	30
200	6		3	2	14

9.3.7 Score attained in modified game settings when varying generations (15 Hidden Layer Neurons)

Number of generations trained for	Score attained in modified game settings		**Seed 1: 4**	**Seed 2: 26**	**Seed 3: 30**
150	4		4	2	7
151	5		7	4	3
152	4		4	4	3

153	4		4	0	7
154	3		2	7	0
155	4		6	5	1
156	2		2	3	0
157	4		1	1	9
158	2		6	0	0
159	2		2	1	2
160	4		7	3	1
161	3		1	9	0
162	2		2	4	1
163	5		2	14	0
164	1		1	0	2
165	4		12	1	0
166	1		0	3	0
167	7		12	4	5
168	3		1	4	3
169	4		11	0	2
170	5		5	3	7
171	5		1	11	2
172	9		8	17	1
173	3		0	0	10
174	4		4	6	2
175	2		1	1	5
176	3		3	7	0
177	9		7	1	20

178	2	0	0	6
179	15	0	0	44
180	2	4	0	3
181	2	5	1	1
182	6	1	7	10
183	8	1	2	20
184	5	1	3	11
185	3	1	1	6
186	3	1	0	7
187	11	11	3	20
188	13	9	8	21
189	25	20	28	28
190	5	6	0	8
191	3	1	8	1
192	9	1	2	24
193	6	2	3	12
194	6	4	3	10
195	12	1	12	24
196	11	0	13	19
197	20	4	4	51
198	5	1	1	14
199	28	13	2	70
200	10	1	1	27

9.3.8 Score attained in modified game settings when varying generations (30 Hidden Layer Neurons)

Number of generations trained for	Score attained in modified game settings		Seed 1: 4	Seed 2: 26	Seed 3: 30
150	24		13	3	57
151	7		1	4	15
152	12		0	6	30
153	20		0	8	51
154	10		0	2	27
155	16		0	13	36
156	3		0	6	3
157	1		0	3	1
158	8		12	2	9
159	11		0	25	8
160	6		0	2	16
161	2		0	6	1
162	7		0	11	10
163	10		5	13	11
164	12		2	1	34
165	4		6	5	1
166	13		6	2	30
167	3		6	1	3
168	24		3	12	56
169	3		4	3	1
170	20		27	2	31

171	8		5	9	9
172	12		0	9	26
173	23		16	2	51
174	8		12	1	10
175	20		54	4	3
176	11		24	4	6
177	15		30	5	11
178	20		9	27	24
179	10		23	2	4
180	7		11	2	8
181	23		52	2	16
182	16		29	3	16
183	29		44	19	25
184	24		46	6	20
185	30		63	6	20
186	11		18	12	3
187	9		3	4	21
188	16		16	1	31
189	24		54	9	9
190	51		135	11	8
191	11		20	7	6
192	18		29	12	12
193	31		53	32	9
194	4		2	4	6
195	60		156	5	18

196	53		150	5	5
197	7		16	1	4
198	6		8	1	8
199	4		1	0	12
200	18		14	7	34