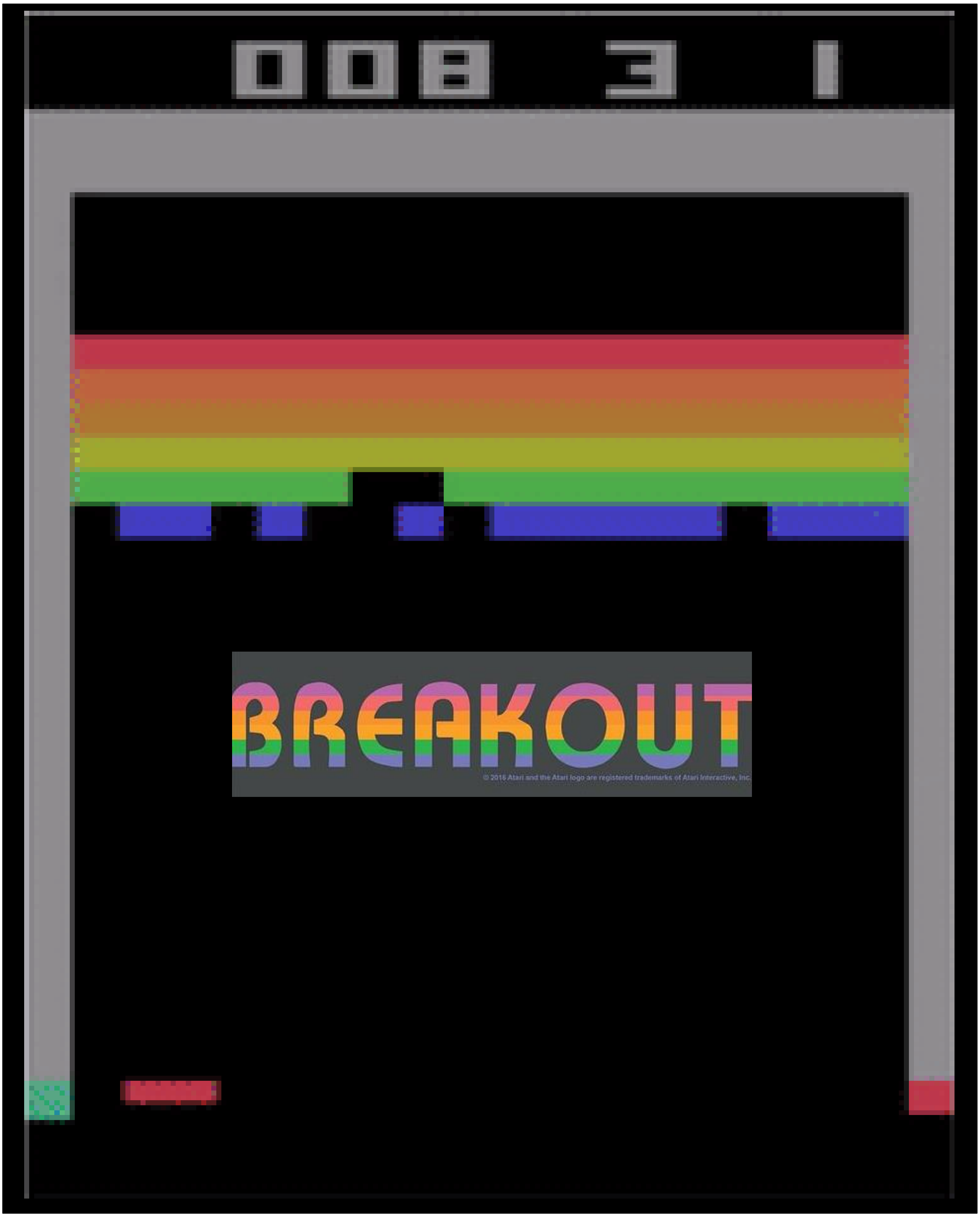
A project applying the NeuroEvolution of Augmenting Topologies (NEAT) Algorithm onto the OpenAI game:


008 3 1
BREAKOUT
© 2016 Atari and the Atari logo are registered trademarks of Atari Interactive, Inc.

Our team: Oliver Morgan, Ismail Hendryx, Anas Jehani, Naomi Jack

Link to Gitlab Project

Table of Contents

Abstract

In this project, we set out to create an AI that can achieve the highest score possible at Atari's 'Breakout'. We used RAM provided by OpenAI Gym, to train a NEAT algorithm. In the final model, it scores 25+ points per game consistently

Introduction

What is Breakout?

Atari's Breakout is a classic arcade game involving a paddle, ball, and a wall of bricks to break through. The objective of the game is to break all the bricks in the limited lives the player has.

The gameplay is as follows, the ball is initialised to drop down from, like gravity in the real world, and the player must prevent the ball from falling below while also aiming at the blocks in order to score points. If you fail to hit a block, the ball will just bounce according to physics, but if you fail to prevent the ball from falling into the bottom of the screen, the player will lose a life.

A detailed breakdown of the score is:

Red - 7 points	Orange - 7 points	Yellow - 4 points
Green - 4 points	Aqua - 1 point	Blue - 1 point

What is the environment?

OpenAI's environment provides us with data to make the machine learning easier. The first kind of data is a 210x160 RGB image of the game and the second is 128 bytes of RAM. Both are valid options; however, it does change which algorithms are best suited for the game. The RAM array contains all the game state, including the reward system, the position of the paddle, position of the ball etc.

Rationale:

We opted for the 128 bytes of RAM and the NeuroEvolution of Augmenting Topologies (NEAT) algorithm. The most common approach would be using the RGB array, such as in Deepmind's famous paper, which is usually coupled with Deep Q-learning(DQL). We explored DQL but opted for NEAT because we were more familiar with this algorithm, although there are a few more reasons covered in the 'alternatives' section.

NEAT is a type of genetic algorithm that is used to train artificial neural networks (NN), in our case feed forward neural networks. The algorithm finds an optimal neural network structure for a given problem, without requiring any prior knowledge.

The algorithm starts with a population of random neural networks, known as genomes, and then uses genetic operators such as mutation and crossover to create new networks. The new networks are then tested on a task, and their fitness is evaluated. The fittest networks are then selected for the next generation, while the less fit networks are discarded. In this case fitness corresponds to the score achieved by the NNs, but what it exactly is defined as is further explored in the next section. The process can be summarised in three stages: Initialisation, selection and reproduction.

Architectural Design:

NEAT has many genetic algorithm techniques, but we settled for five key ones: speciation, mutation, crossover, elitism, and survival threshold. We experimented with other techniques as well, mentioned in the "Implementation" section.

Each genome is a feed forward neural network (FFNN) with thirteen inputs and 3 outputs (RIGHT, LEFT and NOOP). Each genome will have its own random configuration of hidden layer(s), with randomized weights and biases, and an output layer with its' own random weights and biases. These weights would determine the specific directions in which the paddle moved. Every genome plays several episodes (in final version set to five), which are the number of rounds it plays of the game and earns a reward level depending on how much score it acquired.

As mentioned, the score or the reward system, was used in order to contribute to the fitness function. In the final version, the fitness function was the sum of the reward of the population over all the episodes and summed with the penalty function. This determined what 'successful' meant, to select the best to reproduce and carry on to the next generation. This results in a better set of genomes, known as the population.

We opted for the FFNN because it was a relatively simple problem, and fitted well with the RAM array we are given. If we had opted for using the RGB values, we would not have used this neural network.

A configuration file was also used to determine the hyperparameters, such as: population number, generation number, stagnation level etc. These were adjusted by trial and error, a conclusion we came to as other optimisation algorithms were not necessary for the task.

Other design alternatives:

The main design alternative we explored was Deep Q-learning with convolutional neural networks (CNNs). As CNNs were used, we inputted in the RGB value instead of the RAM array. This model was slower than NEAT, and we could not get it running in time. It involved converting to grayscale, as RGB was redundant. In spite of this reduction in information, it was still 84x84 size. After a few more variables are taken into account, the total parameters were about 1,700,000 (Eliuseev, 2022), which made the coding process difficult to debug. However, it is an effective algorithm and would have converged sooner than NEAT(ANDERSSON, 2012). An explanation for why it is so effective is that it uses a Q-table, the net benefit for a given decision, and chooses the highest value out of the table. It steadily learns using backpropagation and gradient descent, to optimise for the best values in the table. This reduces the redundancy found in NEAT, where even at advanced stages in the training it can have unfit genomes in the population.

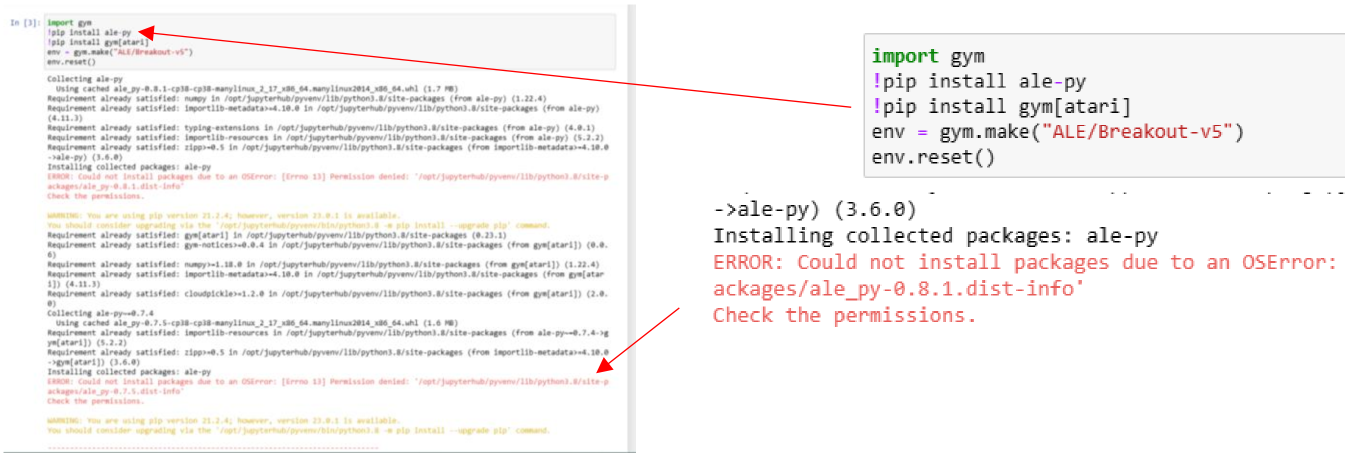
A consideration for neural network design, as an alternative to FFNN, was a Recurrent Neural Network (RNN).This is a sequential type of neural network, which seemed to fit with our problem as the agent takes actions over time. However the input to the neural network is a fixed subset of the RAM memory of the Atari emulator at each time step. This limited its usefulness and so we decided not to use it and stuck with the FFNN.

Implementations:

We had four versions of implementing an AI Agent for "Breakout." Before we go through all of them, an early roadblock was encountered.

Unfortunately, our preferred environments did not support the ALE-py import needed initialise a Breakout instance.

Issues using Jupyter Notebooks (using the Kent Servers):



```
In [3]: import gym
        !pip install ale-py
        !pip install gym[atari]
        env = gym.make("ALE/Breakout-v5")
        env.reset()

Collecting ale-py
  Using cached ale_py-0.8.1-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.7 MB)
Requirement already satisfied: numpy in /opt/jupyterhub/pyvenv/lib/python3.8/site-packages (from ale-py) (1.22.4)
Requirement already satisfied: importlib-metadata>=4.10.0 in /opt/jupyterhub/pyvenv/lib/python3.8/site-packages (from ale-py)
(4.11.3)
Requirement already satisfied: typing-extensions in /opt/jupyterhub/pyvenv/lib/python3.8/site-packages (from ale-py) (4.0.1)
Requirement already satisfied: importlib-resources in /opt/jupyterhub/pyvenv/lib/python3.8/site-packages (from ale-py) (5.2.2)
Requirement already satisfied: zipp>=0.5 in /opt/jupyterhub/pyvenv/lib/python3.8/site-packages (from importlib-metadata>=4.10.0
->ale-py) (3.6.0)
Installing collected packages: ale-py
ERROR: Could not install packages due to an OSError: [Errno 13] Permission denied: '/opt/jupyterhub/pyvenv/lib/python3.8/site-p
ackages/ale_py-0.8.1.dist-info'
Check the permissions.

WARNING: You are using pip version 21.2.4; however, version 23.0.1 is available.
You should consider upgrading via the '/opt/jupyterhub/pyvenv/bin/python3.8 -m pip install --upgrade pip' command.
Requirement already satisfied: gym[atari] in /opt/jupyterhub/pyvenv/lib/python3.8/site-packages (0.23.1)
Requirement already satisfied: gym-notices>=0.0.4 in /opt/jupyterhub/pyvenv/lib/python3.8/site-packages (from gym[atari]) (0.0.
6)
Requirement already satisfied: numpy>=1.18.0 in /opt/jupyterhub/pyvenv/lib/python3.8/site-packages (from gym[atari]) (1.22.4)
Requirement already satisfied: importlib-metadata>=4.10.0 in /opt/jupyterhub/pyvenv/lib/python3.8/site-packages (from gym[atar
i]) (4.11.3)
Requirement already satisfied: cloudpickle>=1.2.0 in /opt/jupyterhub/pyvenv/lib/python3.8/site-packages (from gym[atari]) (2.0.
0)
Collecting ale-py~=0.7.4
  Using cached ale_py-0.7.5-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.6 MB)
Requirement already satisfied: importlib-resources in /opt/jupyterhub/pyvenv/lib/python3.8/site-packages (from ale-py~=0.7.4->g
ym[atari]) (5.2.2)
Requirement already satisfied: zipp>=0.5 in /opt/jupyterhub/pyvenv/lib/python3.8/site-packages (from importlib-metadata>=4.10.0
->gym[atari]) (3.6.0)
Installing collected packages: ale-py
ERROR: Could not install packages due to an OSError: [Errno 13] Permission denied: '/opt/jupyterhub/pyvenv/lib/python3.8/site-p
ackages/ale_py-0.7.5.dist-info'
Check the permissions.

WARNING: You are using pip version 21.2.4; however, version 23.0.1 is available.
You should consider upgrading via the '/opt/jupyterhub/pyvenv/bin/python3.8 -m pip install --upgrade pip' command.
```

```
import gym
!pip install ale-py
!pip install gym[atari]
env = gym.make("ALE/Breakout-v5")
env.reset()
```

```
->ale-py) (3.6.0)
Installing collected packages: ale-py
ERROR: Could not install packages due to an OSError:
ackages/ale_py-0.8.1.dist-info'
Check the permissions.
```

Similar issues were faced when using Virtual Studio Code and other environments such as PyCharm and IntelliJ. Furthermore, the proposition of running a Neural Network on our local machines, which may require hours of training, appeared to be an unsuitable approach.

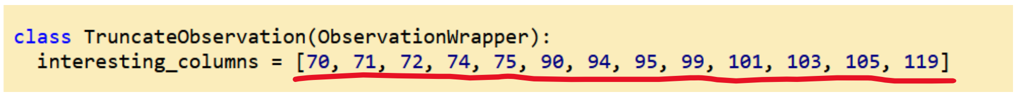
For these reasons, we used the Google Collaboratory platform for our project. All training and runtime operations were executed by Google's servers and the platform supported various imports including ALE-py.

Breakout: Version one

Before we could use NEAT, as concluded in the rationale, we need to determine whether we would use the RAM array or the RGB values. We investigated the RGB approach at first. After printing the RGB values, we knew immediately this approach would be slower. There was a great deal of noise in the data, mainly arrays of [0,0,0], indicating the black background. Although we were familiar with this kind of computer vision problem given our DQL experience, it did not make sense for NEAT. The NEAT algorithm contains hundreds of neural networks, meaning the architecture could not be so complex to take an 84x84 input, as a reminder this was the figure we found in the rationale.

Now knowing we needed to use the RAM array, our task was to find which part of the 128-byte RAM array was most useful. Finding the x-axis position of the paddle, and the X and Y position of the ball was identified as the most important. Our main goal for this version was to understand the RAM array as best as possible, so the paddle movement was kept as random with no learning taking place for this version.

We found the most important values, 13 total, after delving deeper into documentation (CodeProject, 2020).



```
class TruncateObservation(ObservationWrapper):
  interesting_columns = [70, 71, 72, 74, 75, 90, 94, 95, 99, 101, 103, 105, 119]
```

However, we still had to know what these values were responsible for. This was done by trial and error. We would print out which RAM position was used, and observe the corresponding movement using the matplot to plot the state of the game after each step. For example, we observed that when the paddle moves left and right, the 72nd position of the RAM array would be used. In other words, the 72nd position in the RAM had 255 values that determined the X-position of the paddle.

After this we normalised all the RAM values, which we would need for the next stage. The score was predictably low, never reaching higher than 3.

Breakout: Version two

At this stage added the NEAT algorithm. A small removal we incorporated was that by default there are four actions a paddle can take, NOOP, fire, left and right. However, we removed the fire operation, as it just added complexity but did not change anything. We then forced a fire action before every step of the game, ensuring a ball was always fired after the agent lost a life. As mentioned, there are three central stages in NEAT, and GA at large, each stage encompasses its own techniques. The genomes go through the three stages, which are explained below. The values of each of these techniques were based on TechWithTims NEAT Python Flappy Bird (Tim, 2019)/

Initialisation:

Set of genomes was generated all with their own **random** weights and biases. Every genome would play one round of the game, known as the number of episodes, which were all created by a seed, the seed is the random state that each game was initially created as. Therefore, every genome would have its own random variation of the game.

Selection:

The largest determinant to a genome being selected was the **fitness function.** This was set to the scoring system mentioned in the rationale, so if a blue block was broken it was gaining a score of 1. Every genome would play the game once, either beating the game or losing all five of its lives.

Elitism was also a crucial addition. This selects a certain number of the best performing genomes, and carries them onto the next generation.

This encouraged stability in the algorithm, as we guaranteed some good genomes in each generation.

Survival Threshold: This is similar to elitism, accounts for the number of genomes that will carry on to the next generation. However, it is overridden by elitism and meant that it was often a secondary hyperparameter that we adjusted.

Reproduction:

At this stage, the algorithms that were selected must combine their attributes to create a fitter population for the next generation. There are two main ways of doing this.

Crossover: During crossover, two parent genomes are randomly selected from the population, and a new offspring genome is created by combining their structure and weights. The offspring genome inherits nodes and connections from both parents, with excess and disjoint genes being inherited from the more fit parent, and matching genes being inherited randomly from either parent. The weights of the matching genes are averaged to obtain the weight of the corresponding gene in the offspring genome. This was handled by *DefaultReproduction* class in the NEAT library.

This was chosen as it will allow the algorithm to build from one generation to the next, meaning if it performs the correct action, it should exploit this in future generations and similarly poor actions should be disregarded.

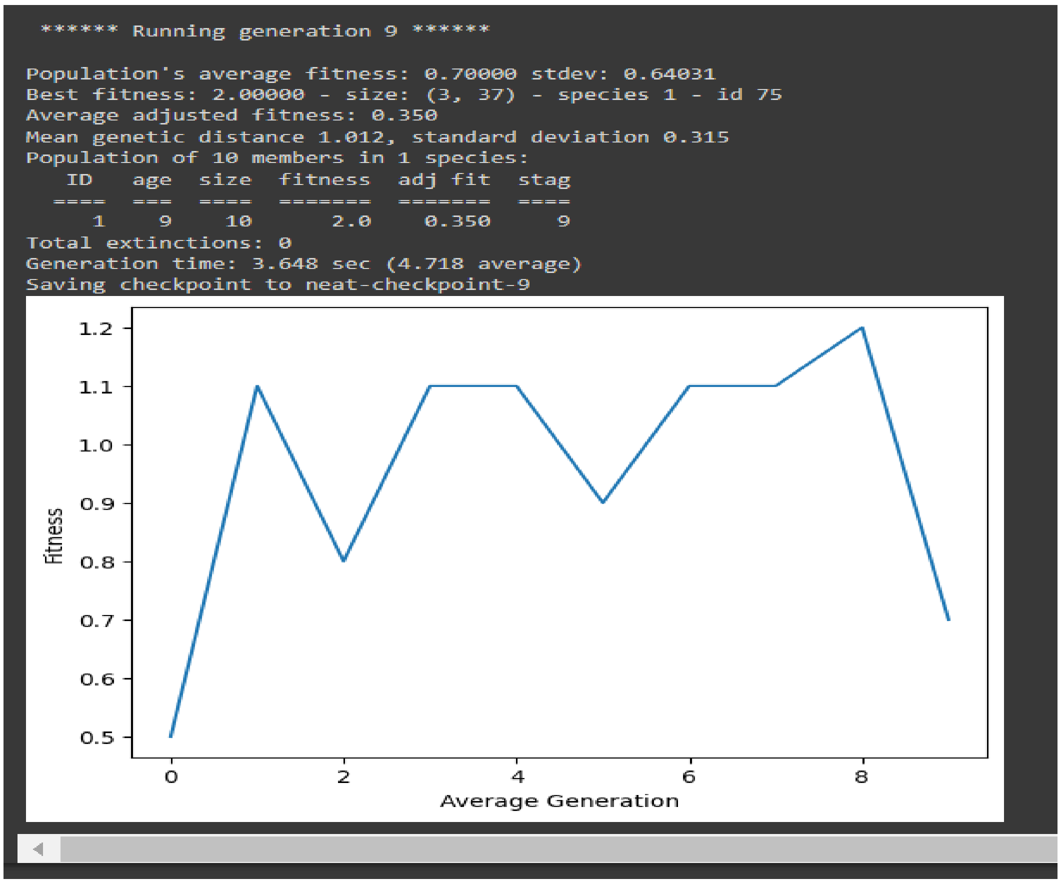
Mutation: Mutation is applied to the offspring genome by adding new nodes or connections or modifying the weights of existing connections. This was handled by *DefaultReproduction* class in the NEAT library.

It was chosen as we did not want to only be limited by what the parents had, but this value had to remain low throughout the process to not have too much randomness.

Speciation: Calculates the genetic distance between each genome and the representative genome of each species. If the distance is less than the specified threshold, the genome is added to that species. Otherwise, a new species is created for that genome.

Now the model was set to learn from its games, as opposed to random movement in the first stage. The fitness function will inform the algorithm what to improve, which at this stage is equal to the reward system as outlined in the rationale. The genetic algorithms used by NEAT would allow the algorithm to how to improve, and as such we expected a much higher score for this version.

Performance Evaluation:



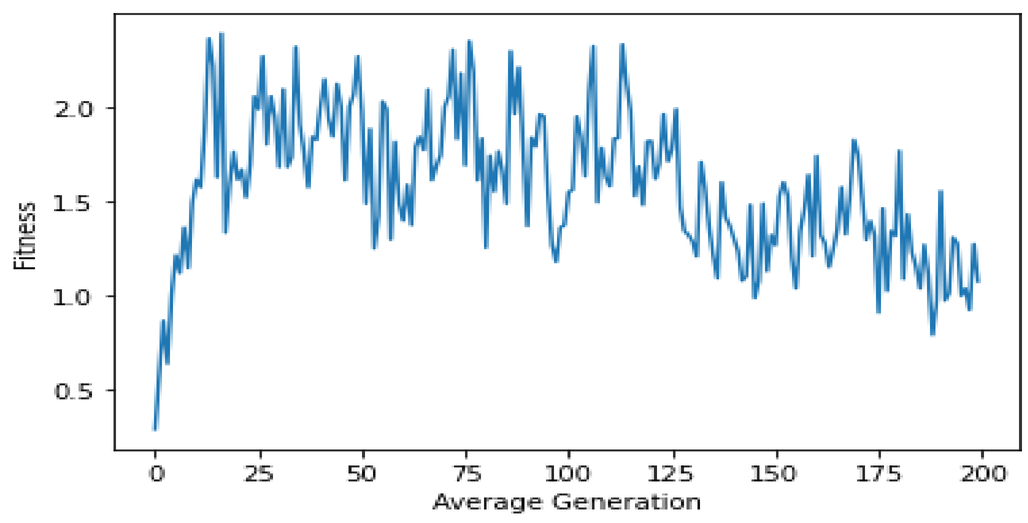
```
 ****** Running generation 9 ******

Population's average fitness: 0.70000 stdev: 0.64031
Best fitness: 2.00000 - size: (3, 37) - species 1 - id 75
Average adjusted fitness: 0.350
Mean genetic distance 1.012, standard deviation 0.315
Population of 10 members in 1 species:
   ID   age  size  fitness  adj fit  stag
  ====  ===  ====  =======  =======  ====
     1    9    10      2.0    0.350     9
Total extinctions: 0
Generation time: 3.648 sec (4.718 average)
Saving checkpoint to neat-checkpoint-9
```

The average fitness peaked around 1.2 meaning the ball was hit around an average of 1.2 times for the best generation. The score was also only slightly increased to four. This was far lower than we expected so we turned to the visualiser, that we set up in the first version, to understand what was happening.

Getting stuck at local minima:

Having trained this version over 10 generations with a population of 10, we noticed in our visualiser that the paddle would get stuck in either far right corner or the far left. To tackle this, our first instinct was to increase the amount of data. For this population size was increased and number of generations was increased, to 30 and 200, respectively.



Although a slightly higher average fitness was achieved during some generations, this was an unsuccessful attempt as for most of the game, the paddle stayed in either of the corners.

The average fitness peaked around 2.4 meaning the ball was hit around an average of 2.4 times for the best generation. The score also remained at the low of four. For the next version we knew we needed a more complex fitness function.

Breakout: Version three

There were several areas we identified as needing improvements, which we will break down for each stage.

Initialisation improvements: The seed used for each genome was different and therefore unfair. A good genome could get a lower score only because the game state was initialised in a disadvantageous manner. We solved this by making the random seed constant for each of the genomes.

Selection improvements:

There were three main problems.

The first problem was that a genome could get lucky as it only played one episode of the game. Episodes is the number of games the genomes get to play, and was now set to be a value of five. The fitness function was changed to take the average of a genomes score between the five episodes, a better indication of a genome's true performance. Now the genome could not just get one lucky episode, it had to perform well in all five, making the algorithm more reliable.

The second problem, partially caused by increasing the episode count, was that the seeds for each episode were all random. It meant that no two genomes could truly be compared, leading to a less accurate selection process. Thus, we kept the seeds constant for each episode, I.e. each episode had a seed that was used for the entire population. This is known as **incremental learning**. This solution meant that each genome could experience five unique game environments, making the selection process better, but also each genome could be fairly compared with each other, inacontrast to before where they all had different seeds.

The third problem was more complicated, that of being stuck at a local minima. As our initial idea of putting more data to the problem did not resolve anything, we knew we had to have a penalty function. This would take successive moves that in either of the farthest X-positions of the paddle, that being the farthest left position by the left wall or being the farthest right position by the right wall. This penalty function would deduct from the reward obtained by a genome in a given episode. We thought that this needed to be a high value at first, to emphasise to the algorithm so it would not get stuck in the local minimum. However, we discovered that any penalty even that of 0.1 would prevent those genomes from being selected.

In total the fitness function was now an average of the reward over the five episodes, and a summation with the penalty function. This now means that fitness is no longer exactly indicative of the score the agent will get from playing, but they are closely correlated.

To be clear fitness function used to be:

fitness function = reward function

And now is:

fitness function = (Σ (reward function)/numberOfEpisodes) - penalty function

No changes were needed for the reproduction stage, although we did tweak the hyperparameters. The generation size, population size and number of elitisms were all slightly increased.

Performance evaluation:

We finally achieved a good score in this version, a score of thirteen. After inspecting the visualiser, we felt that the algorithm was learning well and performing the correct moves. However, it felt like we needed to train it more, a problem that we struggled with as the runtime took multiple hours. Each generation was an average of 91 seconds, and we had a total of 250 generations.

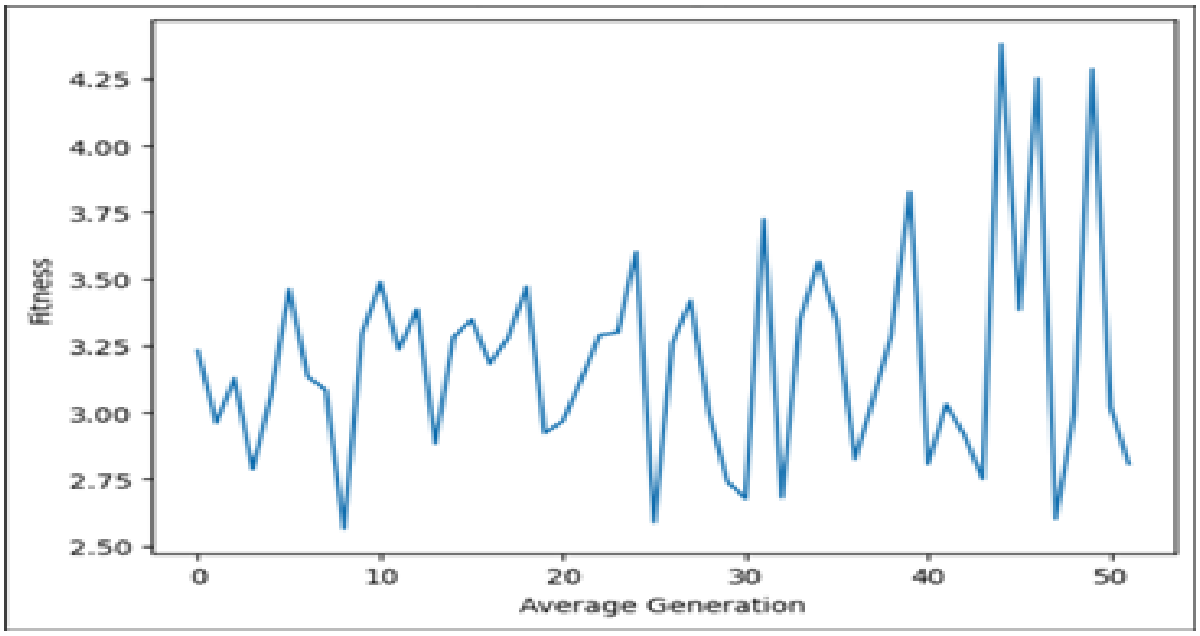
Breakout: Final version

With the Artificially trained game now earning respectable counter scores, the next step was to improve & fine tune the model to achieve even better scores. This is the stage we introduced parallelism, known as parallel NEAT (P-NEAT).

We were reluctant to use parallelisation at first as NEAT is interdependent between generations, one generation influences the next. However, we realised that within each generation, we could parallelise the genomes so that they could be trained concurrently. This would solve our speed problem, as the time taken to complete one generation would be the same for completing any number of generations, constrained by our system's CPU and RAM of course. This was done using the *multiprocessing* package, using a Pool where we define the number of processes, we can run multiple instances of our *eval_genomes* function at once. We settled for 450 generations, previously 250.

We also made final changes to the values of the hyperparameters based on kb100's NEAT_Breakout configuration file (kb100, 2022).

Performance Evaluation



This graph shows the fitness for each generation from Generation 400-450 as we had used checkpoints to help us run the code for extended periods of time over multiple days (which is why the image above only goes up to 50).

A significant difference in average fitness during the majority of generations, with the paddle score reaching 25+ in most games. This version was by far our most successful attempt with the paddle playing efficiently.

An overview of the final approach is as follows:

1. Define the NEAT configuration settings, including the fitness criterion, population size, and genome options.
2. Initialize the population of genomes with random weights and architectures.
3. Evaluate the fitness of each genome by running an episode of the Breakout game using the genome's neural network as the controller. The fitness is calculated based on the score achieved in the episode.
4. Select the fittest genomes to reproduce and create new offspring genomes using crossover and mutation.
5. Repeat steps 3-4 for multiple generations, with each generation containing a new population of genomes.
6. If the maximum fitness threshold is reached, stop the algorithm, and return the best performing genome. Otherwise, continue to the next generation

A summary of the values of the most important hyperparameters is displayed in the table below.

	Version1	Version 2	Version 3	Final Version
Population	10	25	25	50
Generation	30	30	250	450
Mutation rate	0.01	0.01	0.01	0.05
Elitism	2	2	2	3
Connection Add Rate	0.5	0.5	0.5	0.9
Connection Remove Rate	0.5	0.5	0.5	0.05
Survival Threshold	0.2	0.2	0.2	0.05

Potential Improvements:

For the NEAT model there a few things we could have added. One such idea included finding the ball's x position and providing additional reward if the paddle's x position matched the ball's or vice versa. Another idea was to make the fitness function even more complex, by penalising lives lost so that the best genomes would achieve a high score and not lose lives in the process.

The other improvement we would have liked to make was to be able to get the DQL model fully functional. We explored it quite well, but due to bugs we were unable to complete the programming in time. It could have been a good contrast to NEAT, observe how and why the models are so different in architecture and how that affects its performance.

Another improvement that we could have made is to not tune the hyperparameters using trial and error but rather an algorithmic technique such as Bayesian hyperparameter optimisation. This was not required for this project, but it could have made the solution more elegant and save us time, although we still would have needed to tweak the 'prior.' The 'prior' is the initial hyperparameters that it will optimise, a good prior saves time overall and thus would have had some element of trial and error to obtain the best one possible.

Bibliography:

CodeProject. (2020). *Learning Breakout From RAM – Part 1*. [online] Available at: https://www.codeproject.com/Articles/5271949/Learning-Breakout-From-RAM-Part-1 [Accessed 3 Apr. 2023].

Eliuseev, D. (2022). *Teaching a Neural Network to play the Breakout game*. [online] Medium. Available at: https://blog.devgenius.io/teaching-a-neural-network-to-play-the-breakout-game-793ad7d1b20e [Accessed 3 Apr. 2023].

ANDERSSON, M. (2012). *How does the performance of NEAT compare to Reinforcement Learning?* [Pdf] p.76. Available at: https://www.diva-portal.org/smash/get/diva2:1643563/FULLTEXT01.pdf [Accessed 3 Apr. 2023].

Ruscica, T. (2019). *techwithtim/NEAT-Flappy-Bird*. [online] GitHub. Available at: https://github.com/techwithtim/NEAT-Flappy-Bird/blob/master/config-feedforward.txt.

kb100 (2022). *NEAT Breakout*. [online] GitHub. Available at: https://github.com/kb100/NEAT_Breakout/blob/master/train_config [Accessed 3 Apr. 2023].