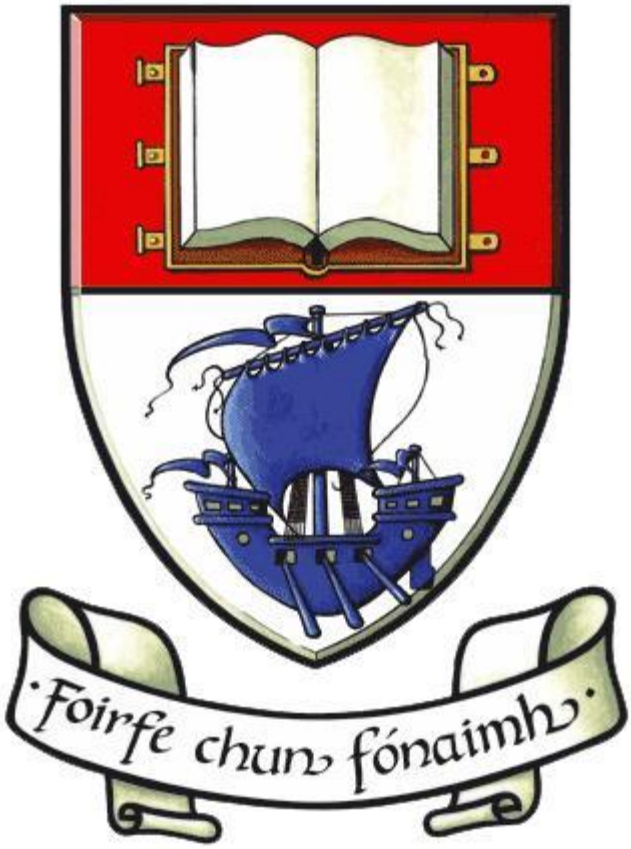


Using Genetic Algorithms to Play Atari Games

FINAL YEAR PROJECT REPORT

Padraig O Neill | Student Number 20063053 | BSc. (Hons) in Entertainment Systems
Supervisor: Dr. Kieran Murphy | Waterford Institute of Technology

Declaration of Authenticity

"Except where explicitly stated, this report contains work that I have done myself. Except where explicitly stated, I have not submitted the work represented in this report in any other course of study leading to an academic award."

Signed: _____

Padraig O Neill

Table of Contents

Abstract 4
1. Acknowledgments 5
2. Project Overview 6
2.1. Document Introduction 6
2.2. Project Scope 6
2.3. Aim 6
2.4. Previous Work in this Area 6
2.4.1. Learnfun & Playfun *7*
2.4.2. DeepMind *7*
2.4.3. OpenAI - Evolution Strategies as a Scalable Alternative to Reinforcement Learning *7*
2.4.4. HyperNEAT-GGP: A HyperNEAT-based Atari General Game Player *7*
2.5. Tools & Technologies 8
2.5.1. Development Environment *8*
2.5.2. Python *8*
2.5.3. Emulator *8*
2.5.4. OpenAI Gym *8*
2.5.5. AI Techniques *9*
2.6. Constraints 11
2.7. Development Iterations 11
3. Implementation 12
3.1. Methodology 12
3.1.1. Agile *12*
3.2. Project Iterations 12
3.2.1. Iteration 1. Hard coded AI *12*
3.2.2. Iteration 2. Using an Existing NEAT Implementation *13*
3.2.3. Iteration 3. Customized Implementation *17*
3.3. Results 19
3.3.1. Random Player *19*
3.3.2. Max Reward Threshold *19*
3.3.3. Fitness Score *20*
3.3.4. Fitness between different games is not universal *20*

3.3.5. Fitness score in relation to each game ... *20*
3.3.6. Experimental Parameters ... *20*
3.3.7. Results Evaluation ... *21*
4. Time & Risk Management ... 23
4.1. Time Management ... 23
4.2. Risk Management ... 23
4.2.1. Emulator & AI Interaction ... *23*
4.2.2. Successful Research ... *23*
5. Project Evaluation ... 24
5.1. Problems ... 24
5.2. Learning Outcomes ... 24
5.2.1. Research Skills ... *24*
5.2.2. Time management ... *24*
5.2.3. Software Cycle ... *25*
5.2.4. Technological Skills ... *25*
5.3. Future Development ... 25
5.3.1. Graphical User Interface ... *25*
5.3.2. HyperNEAT ... *25*
5.3.3. Amazon EC2 ... *25*
6. Conclusion ... 26
7. References ... 27
8. Appendices ... 29
8.1. Glossary ... 29
8.2. Fitness Graphs for NEAT and Custom GA Method ... 31
8.2.1. Custom GA ... *31*
8.2.2. NEAT ... *32*
8.3. Code Overview ... 33
8.3.1. Hard Coded AI ... *33*
8.3.2. NEAT Code ... *34*
8.3.3. Custom AI Code ... *35*

Abstract

In recent years, the rise of AI has been quite prominent both in academic and commercial ventures, with its range of creative uses continuing to grow from its use in mobile phones to self-driving cars. One aspect that has been explored in the use of AI is using computer games as a method to both test and train AI. This project has explored this area, creating an AI engine for playing classic Atari games, without the use of human input or prior learning.

In contrast with some of the previous research that has concentrated on reinforcement learning algorithms and Deep-Q learning, we have used genetic algorithms to approach this task. We present our research into the implementation of this AI engine and its results from playing Atari games, incorporating methods using both neural networks and our own customized approach. Our AI has been tested in use with the Arcade Learning Environment through the interface provided by OpenAI Gym, allowing for comparison with other approaches through their website.

1. Acknowledgments

I would like to thank Dr. Kieran Murphy my project supervisor, for sharing his expertise and guidance during this project, and continuing to push me and give instruction where needed to achieve my best always during its development.

2. Project Overview

2.1. Document Introduction

Presented in this document is the final report for the project "Using Genetic Algorithms to Play Atari Games", containing details regarding its research, development and implementation. Technologies used, the management of time and risk and a self-evaluation of the project will also be included.

The document shall follow the following structure:

- Project Overview
- Implementation
- Time & Risk Management
- Project Evaluation
- Conclusion

2.2. Project Scope

The scope of this project was to research and implement genetic algorithms to play Atari games. This included using both previously developed methods such as NEAT and the creation of my own implementation using genetic algorithms to play Atari games. Due to issues during development, the completed project goals and functionality were refined during the implementation of the project.

It was hoped that we would only work with screen data and score, however in the implementations of the NEAT algorithm, RAM data was used rather than screen data. It was also originally an aim to capture screen data with the use of OpenCV, however this proved ambitious, and instead the interface provided by the emulator was used to gather data from the games.

2.3. Aim

The aim of this project was to research and develop an AI Engine that could be used to play an arbitrary number of Atari games. Part of this aim was to use no prior learning or human input when implementing the AI. The reason for this was so that the AI could be used to play multiple games, rather than being limited to and tailored to specific games. Tailoring the AI to a specific game would simplify the development of the AI but the resulting implementation would then be tied to that game. With our knowledge of the game being played, we can exploit it and use specific strategies and algorithms to optimise the AI. However, these exploits will not carry over from game to game, meaning each game would need its own customised AI. Instead, when implementing a generic AI, the AI could only use general purpose methods. Specific game rules, strategies or heuristics would not be accounted for in the design of the AI.

2.4. Previous Work in this Area

There have been other endeavours into this area, which were come across in research and have served as inspiration and a reference point while working on this project that will briefly be

discussed here. While previously discussed in the first semester report, two more projects have been added to this section.

2.4.1. Learnfun & Playfun

Learnfun and Playfun analyses the bytes in RAM to play NES games. They use lexicographic ordering to watch for bytes in memory that rise as the program sends input to the emulator (Murphy 2013). It was designed mainly for use in playing the game Super Mario Bros on the NES platform. The emulator that was used for this project was FCEUX. The main objective here was to only use values that could be found in RAM and watch them to see if they changed in value. It assumed that a rising value meant the program was "winning". It didn't use any data from the video screen or audio output. The program was written in C++, so that it would be compatible with FCEUX. It was provided some previous input from a human play through to analyse before playing the games itself to "teach" the program.

2.4.2. DeepMind

DeepMind Technologies is a company now owned by Google that came to prominence in 2016 when it pitched its AI program AlphaGo against Lee Sodel, a professional 9 dan rank Go player, and emerged victorious winning 4 out of 5 games (Silver 2016). They used general purpose AI methods, with deep neural networks and reinforcement learning to create this AI. Previous to this they used a version of this program with a deep Q agent to play Atari 2600 games. The only inputs they gave their program was the game score and screen pixel data with high success, beating human player's high scores in some games (Mnih et al 2015).

2.4.3. OpenAI - Evolution Strategies as a Scalable Alternative to Reinforcement Learning

OpenAI have recently applied evolution strategies to use with both the MuJoCo physics engine and Atari games in comparison with previously used RL frameworks. Using a computer cluster of 80 machines with a combined 1,440 CPU cores, they could train a 3D MuJoCo humanoid character to walk in only 10 minutes. In their testing of their implementation with Atari games, on 720 CPUs, they brought the training time down to an hour per game. They found that as they had little interaction between workers it made it possible to implement parallel processing for their training, so more CPUS were used than in comparable RL approaches, but the time needed to train was shorter (Salimans et al 2017).

2.4.4. HyperNEAT-GGP: A HyperNEAT-based Atari General Game Player

HyperNEAT, an extension to the original NEAT algorithm, uses a form of indirect encoding called a Compositional Pattern Producing Network (CPPN). In its encoding of an artificial network it also contains data regarding geometric relationships in its domain. This gives it an advantage over NEAT as it can work with geometric regularities in the domain it's being used, such as Atari 2600 games. In their experiments they used a population size of 100, run for 250 generations. In the game Asterix, one of the games tested in this project using a condor computing cluster, the average and best scores achieved were 870 and 1000 respectively (Hausknecht et al 2012).

2.5. Tools & Technologies

2.5.1. Development Environment

My development environment consisted of a MacBook Pro running OS X Sierra, with Python being the language used for development.

2.5.2. Python

Python was the language used for development in this project. Python is a diverse, quick scripting language with a multitude of libraries available. As of writing, there exists two branches of the Python language, 2.7.13 and 3.6.1. While the 3.x branch is newer and will be the future of Python (Python.Org 2016), not all of the Python libraries that may be used in this project have been updated. Python 2.7.x was the branch used for this project. Python has become one of the main languages currently used for machine learning due to library support from packages such as Tensorflow.

2.5.3. Emulator

To do this project, a way to be able to play a large set of Atari games with a common interface with which to access them was needed. This is where using an emulator came in. As time for this project was limited, an emulator saved time by providing a common interface rather than development time being used on communication protocols to automate key input. Last semester, as detailed in the first report, several emulators were researched and evaluated. Options researched were those previously used by DeepMind, the Arcade Learning Environment (Mnih et al 2015) and Learnfun which used FCEUX (Murphy 2013). On evaluation, I found that the Arcade Learning Environment would be the most suitable for use for this project, in terms of its available features and support.

2.5.3.1. Arcade Learning Environment

The Arcade Learning Environment is not itself directly an emulator. It is a framework built on top of the Stella emulator meaning it provides emulation for the Atari 2600 system (Bellemare et al 2013). It creates an abstraction from the emulator itself allowing the developer to focus on player agent design. Developed in C++, it also allows the creation of agents in Python and Java. It comes with its own documentation, which provides installation details and a brief guide on its python interface and is available on OS x and Linux with some versions compatible with Windows also. On testing input for the Arcade Learning Environment, it was found that using player agents and defined action sets input could be sent successfully to the emulator. It also contains built in methods for grabbing the emulator's screen data back as NumPy arrays and saving the screen outputs as PNG files, which could be used for image manipulation, allowing for the location of objects in frames.

2.5.4. OpenAI Gym

OpenAI Gym, recently released as of April 2016, is a software library currently in beta, aimed at providing a resource for researchers who are making endeavours into reinforcement learning (Brockman 2016). It consists of a library of environments which can be used for researchers to test their own algorithms such as the Arcade Learning Environment, and they can all be accessed

through a common interface. It operates in tandem with a website where results may be shared and algorithms discussed (OpenAI 2016a). It provides tasks for algorithms to solve in the form of environments. These environments include Atari video games, classic RL problems such as cart pole balancing and board games like Go (OpenAI 2016b). Each attempt at a task until termination is called an episode, such as a playthrough of a game of space invaders until all lives are lost. Episodes consist of steps. Each step, the user can send an action to the environment, while receiving an observation, reward, done and info variable back in response. The observation is a representation of the current environment's state. Depending on the environment being used, this may be screen data or RAM dumps in the case of Atari games. The reward is a float value representing whether a successful action was taken in the last step. Done represents a Boolean value which indicates whether the episode has been terminated. The terms for whether an episode is done or not varies from environment to environment. It could be that the last life has been lost in the game or that the timer has run out. Depending on the environment there are different measurements for the problem being solved. In the CartPole environment, it is considered solved if an average reward is met. In Atari environments, there is no reward threshold at which these are considered solved, making them unsolved environments. Although OpenAI's Gym main intention is to be used for reinforcement learning in conjunction with libraries such as Theano or TensorFlow, it also provides a good toolkit for this project and its website allowed the drawing of comparisons with other methods for approaching the task of creating an AI to play Atari games.

2.5.5. AI Techniques

Genetic algorithms were implanted into the approach taken to my AI development, while using previous success in this area as a reference.

2.5.5.1. Genetic Algorithms

A genetic algorithm, is an implementation of Darwinian natural selection, and its theory of survival of the fittest in programming. The initial step spawns a random population of individuals. Individuals can be represented in arrays of bytes. These bytes can encode a representative of behaviours for the AI representing virtual genes. Data in these bytes can be representative things such as simple Integer numbers or values indicating what a chess playing AI puts on taking an enemy piece versus protecting their own pieces (Omid et al 2014). New populations are created iteratively from the fittest of the previous population. Fitness is evaluated using an objective function. In the case of game playing AI, the fitness could be considered as the length that it survives in a game or what the value of the end score is. This process is called Selection. New populations can be created asexually by directly carrying the fittest individuals to the next population. To prevent a mere copy of the previous fit population, other processes are used to introduce diversity such as introducing new randomly generated individuals, crossover and mutation.

1. **Crossover**
   This involves taking two "parent" individuals and merging them to create a single child. By combining two successful parents, the characteristics of each are passed onto the child, with the aim that two successful parent individuals breed a superior child.

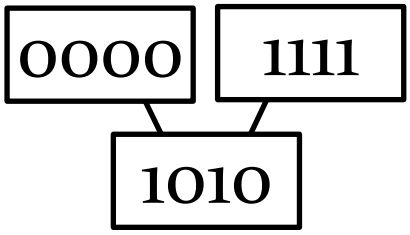


Figure 1 - Crossover

2. **Mutation**
   This is introduced to bring in limited randomness to the population. To establish this, a mutation chance is implemented giving a small chance that each byte in an individual will mutate randomly creating new behaviours. Mutation must be limited. Too high a mutation chance will lead to the population being too random.

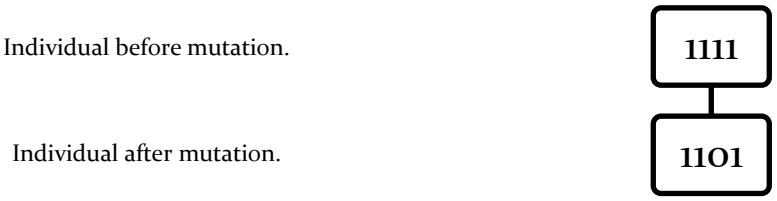


Figure 2 - Mutation

2.5.5.2. Neural Networks

Artificial Neural Networks (ANN) consist of a system of highly connected nodes that receive data via an input layer, passing it through to hidden layers of interconnected nodes where the processing of information is done. The connections between nodes are weighted, and through learning functions, these weights will change in value. Nodes, like human neurons, will perform a function on received input, forwarding its output as input for other nodes. Neural networks have been used with neural evolution in order to create AI for Atari games, however, evolved behaviour was shown to sometimes be different to the intended gameplay, creating exploitive traits to optimise it's score (Hausknecht et al 2014).

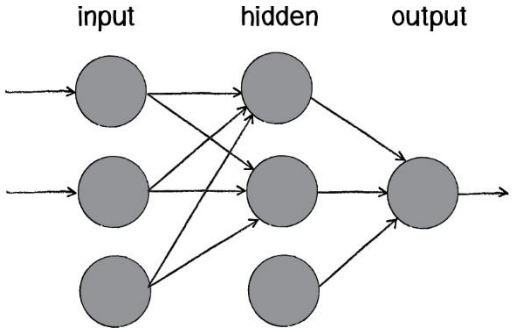


Figure 3 – Neural Networks (Shiffman 2016).

2.5.5.3. RL Learning

Reinforcement Learning (RL) is a form of unsupervised deep learning using neural networks. RL networks see their environment as a state, and will perform actions based on this state. It learns through a reward system, with interaction with its environment providing a numerical reward. Rewards can be positive or negative. As reward comes from its environment, little to no human interaction should be needed. The perfect RL agent will not only choose an action that gives it the best reward available, but will chose actions based on the best possible accumulated reward to receive in the future. In DeepMind's work, they combined the use of reinforcement learning with deep learning neural networks to create an AI capable of playing Atari video games with success, surpassing human player scores with no human input during the process (Mnih et al 2015).

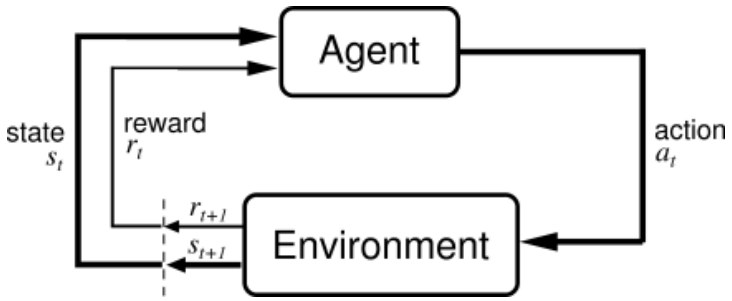


Figure 4 – Reinforcement Learning (University of Alberta 2005).

2.6. Constraints

During the planning and development of this project, several constraints were identified. The main constraint identified was regarding the amount of CPU power that was available. In the HyperNEAT tests, a Condor computing cluster was used respectively (Hausknecht et al 2012) whilst OpenAI used 720 CPUs through Amazon EC2 (Salimans et al 2017) for their testing of evolution strategies. This constraint influenced the experiments run during development by increasing the time taken to run experiments, whilst also reducing the amount of finished data we would have. When performing simulations in this project, the time to run experiments could be multiple hours long, even with smaller parameters then other tests (smaller population, less steps, less episodes). This reduced the amount of data that could be accrued.

2.7. Development Iterations

When implementing the development of the project, there were three major iterations.

- **Iteration 1**: A hard coded AI was developed to test out the tools being used for this project, and to act as a proof of concept.
- **Iteration 2**: Using the NEAT method to play Atari games. This allowed for research into how a previously developed method of using genetic algorithms to perform tasks was implemented, whilst also exploring the use of neural networks.
- **Iteration 3**: Creating an original implementation of genetic algorithms, to address the task of creating an AI player for Atari games.

3. Implementation

3.1. Methodology

3.1.1. Agile

Due to the nature of a single person project, it was decided that an agile approach would be the methodology used. This allowed for the flexibility that would be needed for the project. There were weekly meetings with the supervisor to build the project up iteratively with weekly goals similar to the use of sprints from the Scrum methodology. The Lean Development principle of eliminating waste was also used as an inspiration for approaching the project, as time was limited and avenues that provided little reward would have to be abandoned.

3.2. Project Iterations

3.2.1. Iteration 1. Hard coded AI

For the first iteration of the project, it was decided to implement a hard-coded AI to become familiar with the interface that would be used. The game that was used for this iteration was Pong. While implementing this iteration, the ALE interface was directly dealt with, rather than through, OpenAI. After identifying the relevant correct action set in Pong - moving up, down and doing nothing – a simple AI using these moves was developed. In this case, the paddle would move in relation to the balls height value, if it was above or below the players paddle, the paddle would move in response. Initially, OpenCV was going to be used to identify and track the ball, but later decided that the Python Imaging Library (PIL) library would fulfill what was required.

To track the ball, first some basic preprocessing of the image was done. The part of the frame that was needed was cut out into a region, removing the score and top wall.

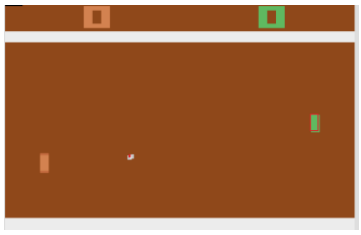


Figure 5: Original frame.

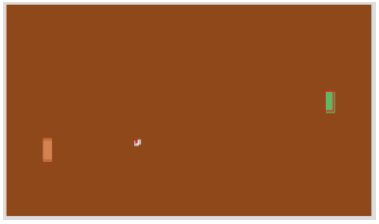


Figure 6: cropped frame.

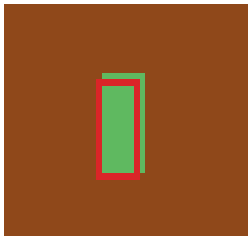


Figure 7: Paddle pixels identified

Once the region was gathered, the relevant pixel colours for both the paddle and ball would be searched for. Using the value for the objects height that was derived from the region, if the ball fell outside of a range for the paddles y value then the paddle would move in response.

This approach to the hardcoded AI proved successful in playing the game. However due to the constant image processing it was a bit intensive. As an upgrade to this, rather than track just the ball's y value, the x value could be tracked also and used to raycast where the ball is going. As this hard-coded AI was not the emphasis of this project, time was not spent on improving this AI.

3.2.2. Iteration 2. Using an Existing NEAT Implementation

During research, one of the most successful and prominent use of genetic algorithms to crop up repeatedly, was the NEAT method. Here, it will be discussed it in detail as it served as the biggest point of reference for the project and understanding of the method was critical to its use.

3.2.2.1. What is NeuroEvolution of Augmenting Topologies (NEAT)?

NEAT is a method developed for evolving artificial neural networks (O Stanley and Miikkulainen 2002). NEAT uses feed forward neural networks. Its goal is to have a minimal population and to minimize the networks that are generated.

Each Genome in NEAT contains a list of connection genes, each of which refers to two node genes to which it is connected. Node genes provide a list of inputs, hidden nodes & outputs that can be connected. Connection genes provide the in-node, the out-node, the weight of the connection, whether connection gene is expressed and the innovation number for finding corresponding genes.

Mutation can change both connection weights and network structures. Structural mutations occur in two ways. Each mutation expands the size of the genome by adding genes. These can be a connection or node mutations. Connection mutations add a single new connection gene with a random weight connecting two previously unconnected nodes. Node mutations split an existing connection, with the new node placed where the old connection used to be. The old connection is disabled and two new connections are added to the genome. The new connection leading into the new node receives a weight of 1, and the new connection leading out receives the same weight as the old connection. Mutation causes the genomes to gradually get larger resulting in varying genome sizes, sometimes with different connections at the same positions.

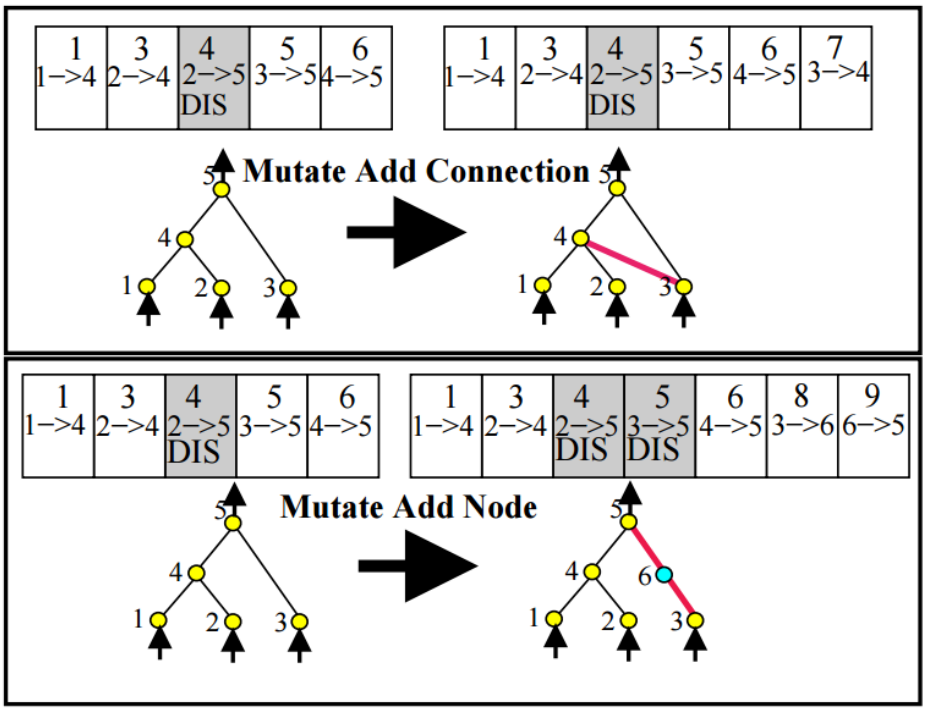


Figure 8: Structural mutation in NEAT. Gene innovation numbers are at the top of each gene. Mutate Add Connection: New connection gene is added to the end of the genome, with the next innovation number. Mutate Add Node: The connection gene being split is disabled, with two new connection genes added to the genome at the end. The created node lies between two new connections (O Stanley and Miikkulainen 2002).

Whenever a new gene appears due to mutation, a global innovation number is incremented and assigned to that gene. Innovation numbers represent a chronology of the appearance of genes. Whenever genomes mate, offspring will inherit the same innovation numbers on each gene. This means the origin of each gene is known, allowing the system to know which genes match. By keeping a list of the innovations that occurred in the current generation, when the same structure arises more than once through independent mutations in the same generation, each identical mutation is assigned the same innovation number. This solves Competing Conventions/Permutations Problem - having more than one way to express a solution to a weight optimization problem with a neural network.

When crossing over, genes in both genomes with the same innovation numbers are lined up. These genes are called matching genes. Genes that do not match are either disjoint or excess, depending on whether they occur within or outside the range of the other parent's innovation numbers. They show structure that is not present in the other genome. In creating the new offspring, genes are randomly chosen from either parent at matching genes. Excess or disjoint genes are always included from the more fit parent. Smaller structures optimize faster than larger structures and adding nodes and connections typically decreases the fitness of the network. New structures have little hope of surviving more than one generation, thus speciation is used to combat this. Dividing the population into species allows genomes to compete and optimize within their own species.

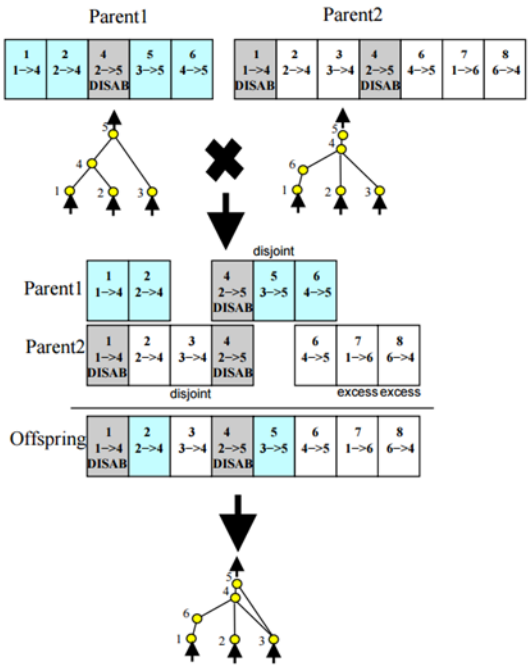


Figure 9: Matching up genomes for using innovation numbers. Parent 1's and Parent 2's innovation numbers show which genes match up. Matching genes are inherited randomly, whereas disjoint genes (those that do not match in the middle) and excess genes (those that do not match in the end) are inherited from the more fit parent. Shown above are genomes with equal finesses, so disjoint and excess genes are inherited randomly. Genes that are disabled may be enabled again in future generations, as there is chance that an inherited gene is enabled if it is disabled in either parent (O Stanley and Miikkulainen 2002).

The number of excess and disjoint genes between a pair of genomes is used to measure compatibility distance. The more disjoint, the less compatible they are. A compatibility threshold is set, and if the distance falls within the threshold, genomes are in the same species. Species are represented by a random genome inside the species from the previous generation. A genome in the current generation is placed in the first species in which it is compatible with the representative genome of that species. If the genome is not compatible with any existing species, a new species is initiated with the genome as it's representative. Individuals within the same species share their fitness payoff, called fitness sharing. The number of networks that can exist in the population on a fitness peak is limited by its size. Larger species with some successful genomes must share the success with other genomes in the species, reducing threat of one species taking over. Species reproduce by first eliminating the lowest performing members from the population. Population is then replaced by the offspring of the remaining organisms in each species. If fitness of the entire population does not improve for more than 20 generations, only the top two species can reproduce.

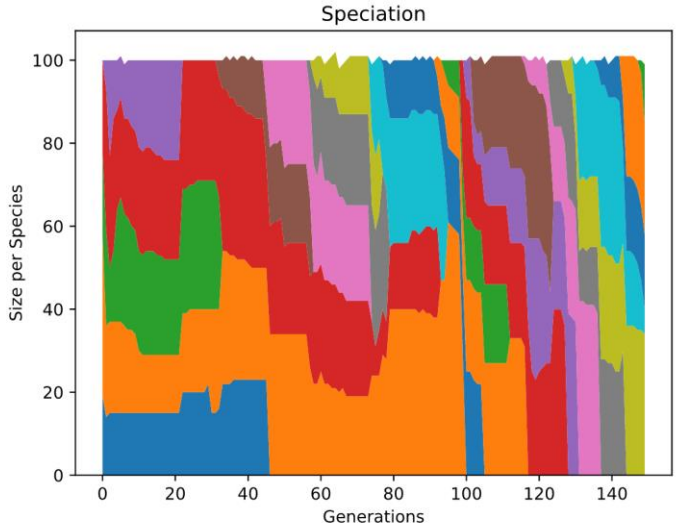


Figure 10 – Speciation Graph for MsPacman played by NEAT .

Size of each species has been graphed against the number of generations for the NEAT player. Each colour represents a distinct species. When all genomes in a species have been removed, it is removed on the graph such as the lower left blue block. New coloured blocks appearances indicate a new species appearing, as can be seen no species ever takes over, due to fitness sharing.

3.2.2.2. Using the NEAT-python Implementation

For the implementation of NEAT in this project, the recently released neat-python library was used (NEAT-Python's Documentation 2017). To change the parameters used to create our population using NEAT, a configuration file is prepared. In this configuration file, it is possible to modify and tweak individual parameters, such as our mutation rate for our connections or our nodes and specify the number of input and outputs we will have in our neural network and the size of the population to be generated. To use our NEAT player, we provide it our evaluation function for the genomes, so that it will know what fitness score the genome has.

Within our created evaluate function we create a feed forward neural network – the phenotype - using our genome and our configuration file parameters. This network represents the phenotype. With our neural network prepared, we provide the environment through OpenAI Gym. On each step, the observation data is sent to the neural network. In the case of this project the observation data is the RAM. The output received from the neural network will be used to decide on the action that will be taken in this step. This process continues until the episode is terminated. The fitness is calculated as an average of the rewards received from every step. When using multiple episodes this fitness is then averaged among the number of episodes that have been simulated.

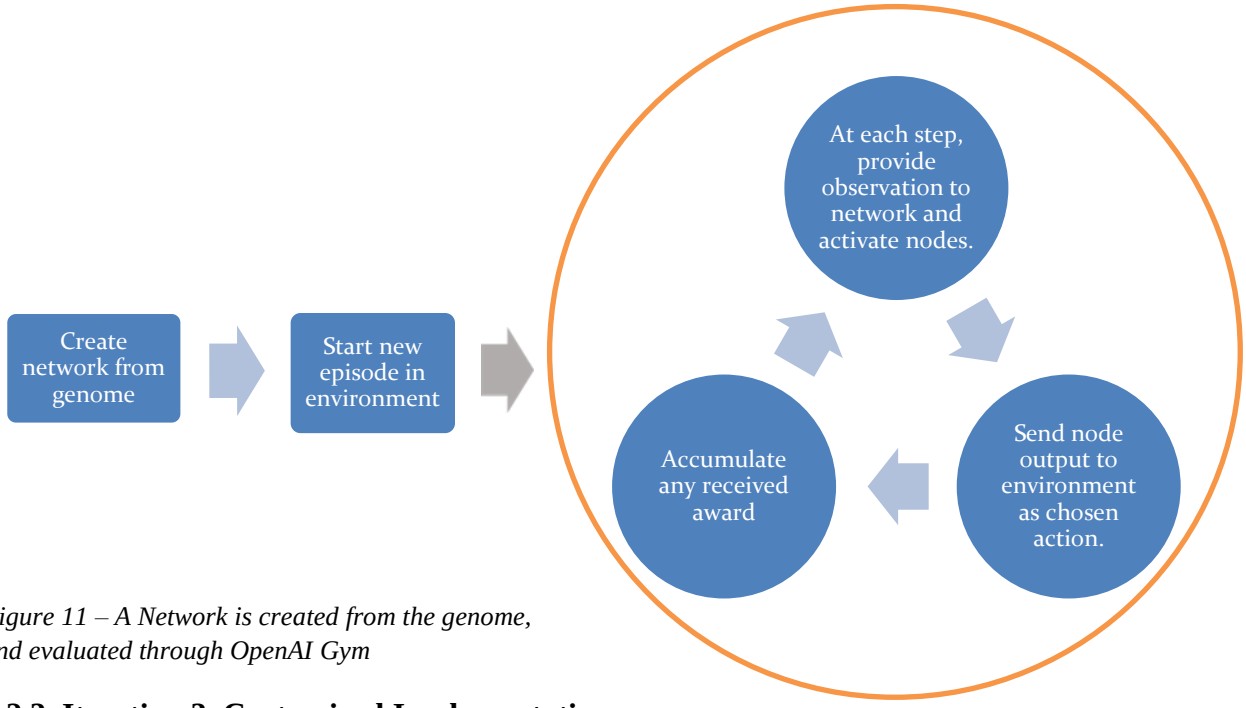


Figure 11 – A Network is created from the genome, and evaluated through OpenAI Gym

3.2.3. Iteration 3. Customized Implementation

In the genetic algorithms method created in this implementation, neural networks were not used. This was costly due to having limited hardware. In designing the approach, a quicker way of using genetic algorithms was an aim. To this end, each genome in this method consisted of an array of controller input. In the OpenAI Gym interface, controller input is represented as an integer value from a set of legal actions. Every gene (Array element) thus represented a valid individual controller input. When creating the mating pool for the creation of a new population, I used roulette wheel selection to give a proportionate chance to each genome to be selected for crossover, while also spawning a certain amount of brand new genomes.

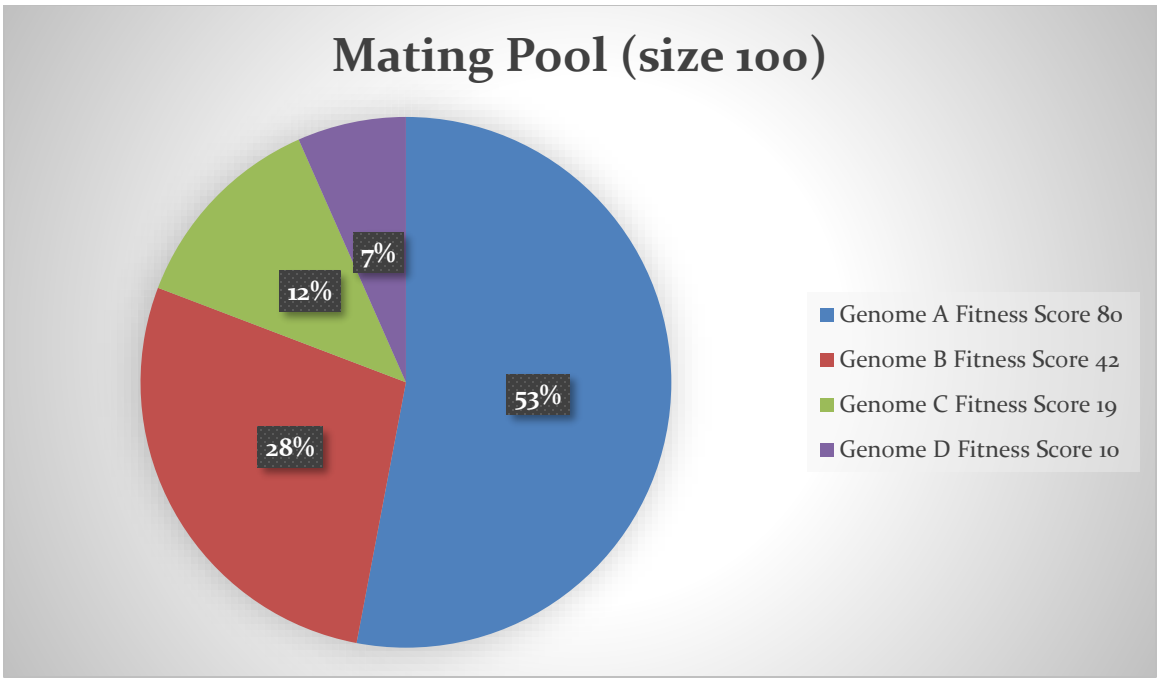


Figure 12 – Genomes are added to the mating pool in proportion to their fitness

During crossover, when creating a new genome, two genomes are taken from a mating pool with each gene having a 50% chance of being inherited by the child from either parent. This child genome is then subject to mutation, with each gene having a chance to change to a random value for legal actions.

Initial implementation of this approach did not see much of an improvement over time. Thus, two new factors were introduced to the creation of new populations. Elitism and Culling. Whenever a new population was to be generated, a percentage of the current population would be left out of the mating pool to avoid the worst performers. Elitism was integrated by directly carrying over a certain amount of the best genomes in each generation. This was to combat the best players being lost every generation.

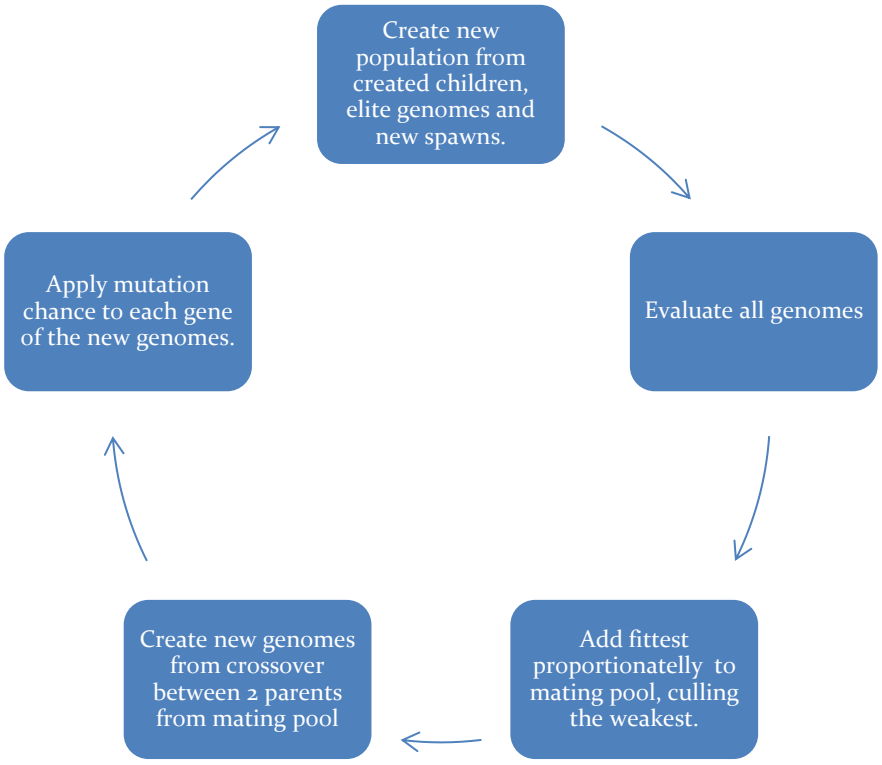


Figure 13 – The cycle of from being evaluated, to the creation of a new population.

In terms of efficiency, by running the custom implementation, time to generate, and evaluate a generation was cut by almost half, without the use of any parallel processing for evaluating the genomes. In the case of the best fitness scores of the genomes, in three of five games, the best player in the custom approach scored better.

3.3. Results.

There were several areas that have to be considered when evaluating the results in order to clarify their meaning, which shall be discussed here.

3.3.1. Random Player

To create a random player score for comparison, each environment was evaluated by playing each environment 1000 times, for a length of 5000 steps, taking a random possible action at each step.

3.3.2. Max Reward Threshold

For the Atari environments, there are no maximum reward thresholds. They are all considered unsolved environments. This means it is not solved upon reaching any specific reward value. Thus, when testing with Atari games, there was no score at which a player would be considered to have “solved” the game.

3.3.3. Fitness Score

The Fitness score has been derived from the total reward received during an episode. We accumulate the total value of rewards received in each episode at each step. The reward in most steps will be 0. In experiments where numerous episodes have been played for evaluation such as in NEAT, the total reward will be divided by the number of episodes, thus finding the mean value.

3.3.4. Fitness between different games is not universal

The scoring for fitness between environments is not universal. Some environments have more opportunities for reward and receive larger reward values when they do receive rewards. A less skilled player in one environment may receive a high fitness score, while a highly-skilled player in another may not receive a high score due to these variances from game to game. In the game Breakout, the opportunity to receive rewards and amounts that rewards are worth is considerably less and slower than in a faster paced game such as MsPacman. As each fitness score is only relevant to its environment, fitness scores between the different approaches should only be compared in the same environment.

3.3.5. Fitness score in relation to each game

Game	**Rewards received when:**
AirRaid	Enemy is destroyed, rewarding from 25 upwards to 100.
Asterix	Collectible picked up, rewarding of 50 is received.
Breakout	A block is destroyed a reward of 1 is received, with higher blocks worth more.
Ms Pacman	A pill is eaten, rewarding a value of 10, destroying ghosts is worth a value of 200.
Space Invaders	A Score of 5 each time an Invader is destroyed. Each row going upwards increments an addition 5 to this score. E.g. The invaders on the third row are worth 15, enemies on the 6th row are worth 30. Destroying bonus invaders worth more.

3.3.6. Experimental Parameters

For the simulation using the NEAT player the same configuration file was used for all games. Each game with NEAT was run for 150 generations with a population size of 100, each evaluated in 3 episodes for 10000 steps, while the custom method used the same parameters. However, each

genome in the custom method was only evaluated for one episode, as the seed value for the environment was specified during evaluation, leading fitness scores by the same genome in the same environment to be identical. Both the NEAT method and custom method were compared across five games.

3.3.7. Results Evaluation

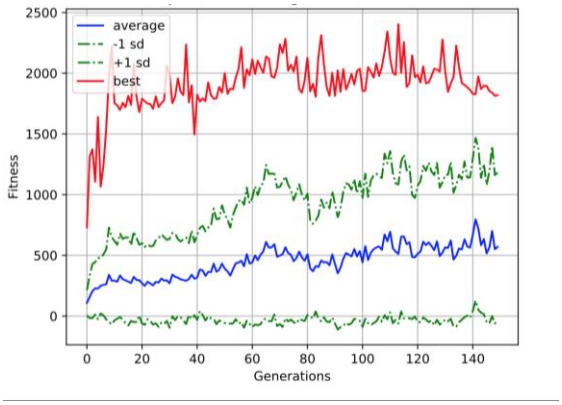
Game	Random Player Average Fitness	NEAT average generation time.	Custom GA average generation time.	Best NEAT Genome Fitness Score	Best Custom GA Genome Fitness Score
Airraid	591.9	150.02	44.89	5525	3475
Asterix	375.3	152.07	17.55	2000	1200
Breakout	1.3	249.18	20.24	12.0	26
MsPacman	226.2	61.78	37.15	2283	5110
Space Invaders	141.1	103.06	31.62	781.66	865.0

The best scores from both NEAT and the custom method scored significantly better than a random player on all games showing that the approaches used have at least beat a random player.

Best results in comparison with the random player occurred for both NEAT and the custom method showed in MsPacman, where it scored 10.09 times better than the random player, while the custom methods best player 20 times better. In contrast, the worst game for performance was also the same – Asterix. NEAT only managed to accumulate a score 5.33 times better than random, and our custom method only scored 3.2 times better.

In relation to time spent used for training the NEAT player, the average time used on the custom GA method, was nearly 5 times less. Even considering that the NEAT method, which had each genome evaluated for 3 episodes this leaves the custom GA method faster without any use of parallel processing for evaluation which NEAT did use.

On three of the games tested, the custom GA method provided a better scoring genome than NEAT. In MsPacman and Breakout the best custom GA genome scored just over twice as well as the NEAT genome. The custom GA method also beat it in Space Invaders, but by a far smaller margin.



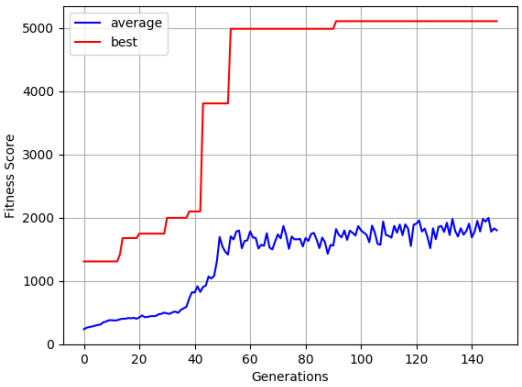


Figure 14 - NEAT Populations Best & Average Fitness MsPacman Figure 15 – Custom Methods Populations Best & Average Fitness in MsPacman

Looking at the graphing of both methods performance in MsPacman, we can draw some conclusions. In both methods, there are fluctuations in the average. This is due to the constant flow of new genomes, with occasions where a group have performed badly leading to the decline of the average. This is also the case for large jumps in the populations fitness but with the new genomes performing considerably above average. However, looking at the graphs in whole, both methods show a steady incline over time.

With NEAT, there are also fluctuations in the best scores. This is due to use of stagnation. When a genome hasn't innovated in a certain amount of time it is removed. In the custom GA implementation, a form of stagnation was not implemented. The elitism that is used to carry over a certain amount of the best player's directly to the next population results in the plateaus we see in the best values in Figure 14. Until the best score is beaten, this score remains the same.

In terms of time spent training, our custom approach has performed better than NEAT due to better scores in three out of five games. However, drawing a conclusion from the MsPacman graphs, and the graphs contained in the appendix it appears that the average for our custom approach is more susceptible to being influenced by the best score. In Figure 14, when there is a large jump after generation 40, we also see this trend in the average. While not seen here, this may indicate that a stagnant value for our best score for extended periods, will result in stagnant scores in the average as well. In NEAT, towards the end of the 150 generations, the best score started to decline, however its average continued to rise suggesting there is less of an effect from the best score in NEAT in relation to the average.

In conclusion, looking at the resulting performances of both algorithms, we can see that looking at the scores achieved suggest that genetic algorithms can perform better than a random player, and to a level that is comparable to an amateur human player unaware of the game rules. Whilst the results achieved in this project, do not match the success of DeepMinds, they do show some success.

4. Time & Risk Management

4.1. Time Management

As with all large projects spanning a long amount of time, time management was an area that had to be considered, lest it became an issue. As this project had an emphasis on research, time had to be allocated for reading white papers, blogs and documentation. A significant amount of time would be spent on this research and comprehension of the topics.

As stated in the methodology section, an approach had to be taken to research and development where features that were not adding anything meaningful to the project would be disregarded and not developed further. This didn't necessarily mean they would not add value, but for the prospective value it would add was not worth the allocation of time.

4.2. Risk Management

4.2.1. Emulator & AI Interaction

A risk that was addressed for early on in development was whether the AI would respond fast enough to send interaction to the emulator the game was playing on. During development, this was combatted by sending input on a per step basis in each game. This resulted in avoiding the problem of having the AI send an action too quickly to the emulator, however with the caveat that the slower the AI was to decide on an action, the slower the game would be and longer it would take to evaluate the genome.

4.2.2. Successful Research

As this project was research and experiment based, there was no guarantee that the approaches that were to be implemented would be efficient or successful. This however is the nature of a research based project, and as the aim of the project was to test using genetic algorithms in the test case of playing Atari game, results even if they shined negatively on this approach are still results and would indicate it is an avenue not worth further exploring.

5. Project Evaluation

5.1. Problems

As is to be expected with a large project, there were problems and difficulties encountered during development. The largest of these, was the initial setting up and tweaking of tools being used. When initially using OpenCV to do my image manipulation for the hard-coded AI, OpenCV proved awkward to initially get up and running. Due to a change in design however for the hard-coded AI which now uses PIL, this problem was combatted.

The ALE framework provided the same issue and while it was installed correctly eventually, it did take a while. Upon doing some research into OpenAI Gym, it was decided to make the switch to using it. The installation process was streamlined, easing the setup of experiments whilst also providing the features of ALE and multiple other environments for testing with.

Time required to do experiments proved to be an issue. In the case of the NEAT implementation, experiments could take up to 8-10 hours in the case of some games, with the parameters used. These often had to be left run unsupervised at night, which meant that if an error occurred or parameters such as population size, mutation rate etcetera needed to be tweaked then the experiment needed to rerun which was quite time costly, and made the discovery of bugs a lengthy process.

5.2. Learning Outcomes

During both planning and developing for this project, there were multiple areas where my skills and experience were improved.

5.2.1. Research Skills

One of these areas was the reading and researching of academic papers. While this has been part of some previous projects, it was never to the extent of this project and on topics that were as scientific as these were. Initially, reading of these papers proved difficult as the concepts and way of writing were sometimes foreign to me, especially in the case of neural networks which had never come up during my college course, but with time I became accustomed to this way of writing and have become much more adept at reading and comprehending scientific academic papers.

5.2.2. Time management

Time management skills were substantially developed as this was the largest project that I have worked on to date. It is difficult to plan without having completed a project of such length before. On completion of this project if I were to go back to the start of the project I would organize my time much more efficiently. One specific I have learned is that it is not necessarily about the hours put in, but how those hours are spent. It is not the quantity, but the quality of work that matters, and I feel that I would be much more confident in this area if taking on a new project. Working on a project of this scale has made evident the importance of using time wisely, and achieving as much as possible in a limited time frame,

5.2.3. Software Cycle

This project allowed me to work upon something significantly large from its initial conception to fruition, allowing me to see the full software cycle and become familiar with the planning, research and development that it requires.

5.2.4. Technological Skills

Due to being able to work on one specific project has allowed my programing skills to improve significantly, largely due to the amount of self-learning it required. Learning about technologies that were used, often required the reading of documentation and papers rather than being taught in, forcing me to stop and comprehend what I was trying to use. Experience was accrued with technologies, that I previously would never have used during my course like OpenCV and neural networks.

5.3. Future Development

As is the case with many projects, there was a limited time frame for this. Thus, some features that are felt would add to the project have not been implemented. If further development were to happen with the project, some of these features that would be added will be discussed here.

5.3.1. Graphical User Interface

As the project currently runs as scripts with information provided though the terminal with the game screen being the only visuals that could be rendered, a feature that would add to the project would be a graphical user interface. Through this, adding the ability to make it possible to change the parameters of the experiments without configuration files and providing some visual feedback as the experiments runs would be planned such as graphing the data live. Due to previous experience with the PyQt framework, it is the graphical library planned for use.

5.3.2. HyperNEAT

One of the other projects in this area used an extension of the NEAT method – HyperNEAT. As an extra comparison for the NEAT and customized genetic algorithms method, it would be considered beneficial to add an implementation of HyperNEAT to this project. While also adding an extra comparison, further research into HyperNEAT, may provide some useful features that could be used in the customized method.

5.3.3. Amazon EC2

In the recent OpenAI paper Evolution Strategies as a Scalable Alternative to Reinforcement Learning, they made use of Amazon Elastic Compute Cloud to achieve quick training speeds for their project. This would be something interesting to consider, as one thing found during the project was that experiments were quite time consuming. If it were possible to speed this up using Amazon Elastic Compute Cloud, then it would allow for quicker innovation and to work with larger data sets.

6. Conclusion

On review of this project, it proved to be a fantastic experience, allowing me to learn about many interesting technologies I wouldn't normally have got to use. While it proved a tough challenge and not always easy, much knowledge was accrued and skills developed including my research, programming and time management skills. I now feel confident in taking on another project of this scale, and it has heavily inspired a love of research.

The ability to say that I have taken a project of this size from conception to fruition, makes me feel considerably more prepared for entering the industry and confident in my abilities.

Overall regarding the finished project, while I do feel there are many areas that could be improved and explored in future development I feel happy with the results of this project. Machine learning and artificial intelligence are becoming ever prominent in both industry and academics, and to have experience in this area I feel is a great boon and is an area that I plan to continue research into.

7. References

Bellemare, M., Naddaf, Y., Veness, J. and Bowling, M. (2013) 'The Arcade Learning Environment:' An Evaluation Platform for General Agents' Journal of Artificial Intelligence Research, 47(1), available: http://www.arcadelearningenvironment.org/wp-content/uploads/2012/07/bellemare13arcade.pdf [accessed 10 December 2016].

Brockman, G. & Schulman,J. (2016) 'OpenAI Gym Beta', OpenAI Blog [online], 27 Apr, available: https://blog.openai.com/openai-gym-beta/ [accessed 25 Mar 2017].

Hausknecht, M., Khandelwal, P., Miikkulainen, R. and Stone, P. (2013) 'HyperNEAT-GGP: A HyperNEAT-based Atari General Game Player', available: http://nn.cs.utexas.edu/downloads/papers/hausknecht.gecco12.pdf [accessed 21 April 2017].

Hausknecht, M., Lehman, J., Miikkulainen, R. and Stone, P. (2014) 'A Neuroevolution Approach to General Atari Game Playing', IEEE Transactions on Evolutionary Computation, 6(4), available: http://www.cs.utexas.edu/~mhauskn/papers/atari.pdf [accessed 10 Dec 2016].

NEAT-Python's Documentation (2017) NEAT-Python's Documentation [online], available: http://neat-python.readthedocs.io/en/latest/# [accessed 21 Apr 2017].

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves,A., Riedmiller, M., Fidjeland, A. , Ostrovski, G., Petersen , S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015) 'Human-level control through deep reinforcement learning', Nature, 518(7540), available: https://storage.googleapis.com/deepmind-data/assets/papers/DeepMindNature14236Paper.pdf [accessed 9 Dec 2016].

Murphy, T. (2013) 'The First Level of Super Mario Bros. is Easy with Lexicographic, Orderings and Time Travel . . . after that it gets a little tricky.', 1 April, available: http://www.cs.cmu.edu/~tom7/mario/mario.pdf [accessed Dec 7 2016].

Omid, D., Henrik, H., Koppel, M. and Nethanyahu, N. (2014) 'Genetic Algorithms for Evolving Computer Chess Programs', IEEE Transactions on Evolutionary Computation, 18(5), available: http://www.genetic-programming.org/hc2014/David-Paper.pdf [accessed 10 Dec 2016].

OpenAI (2016a) OpenAI Gym[online], available: https://gym.openai.com/ [accessed 25 Mar 2017].

OpenAI (2016b) OpenAI Gym Environments[online], available: https://gym.openai.com/envs [accessed 25 Mar 2017].

O Stanley, K. and Miikkulainen, R. (2002) 'Evolving neural networks through augmenting topologies', Evolutionary Computation, 10, 99-127 available: http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf [accessed 21 Apr 2017].

Python.org (2016) 'Python 2 or Python 3', available: https://wiki.python.org/moin/Python2orPython3 [accessed 9 Dec 2016].

Salimans, T., Ho, J.,Chen, X., & Sutskever, I. (2017) 'Evolution Strategies as a Scalable Alternative to Reinforcement Learning', ARXIV, available: https://arxiv.org/abs/1703.03864 [accessed 21 April 2017].

Shiffman, D. (2016) Figure 10.14[image online], available: http://natureofcode.com/book/chapter-10-neural-networks/ [accessed 9 Dec 2016].

Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham. J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Graepal, T., Kavukcuoglu, K. and Hassabis, D. (2016) 'Mastering the Game of Go with Deep Neural Networks and Tree Search', Nature, 529(7587), available: http://airesearch.com/wp-content/uploads/2016/01/deepmind-mastering-go.pdf [accessed 9 Dec 2016].

University of Alberta (2005) Figure 3.1: The agent-environment interaction in reinforcement learning. [image online], available: https://webdocs.cs.ualberta.ca/~sutton/book/ebook/node28.html [accessed 10 December 2016].

8. Appendices

8.1. Glossary

Connection - A connection is the link between two nodes.

Environment - An environment is a task to solve. In the case of our OpenAI Gym this can be an Atari game, or a board game such as Go.

Episode - An episode is a simulation of an environment until termination, such as all lives being lost, or the time running out.

Fitness score - The fitness score is a measure of how successful a genome has been at solving the task it has been given. How fitness is measured depends on the problem. In the case of Atari games, it uses rewards gathered from the environment.

Game score - The game score is the score as it is defined within the respective game. While it is not necessarily the same as the fitness score, they are correlated.

Genome - Also sometimes referred to as a chromosome or genotype. The genome is a set of parameters or variables which define a proposed solution to the current problem. In NEAT, the genome includes a list of connection genes, each of which refers to two node genes being connected.

Mutation - Mutation is used to maintain diversity from one generation of a population of to the next. If it is set too high, the population will be too random, and may never come close to a solution.

Neat - NeuroEvolution of Augmenting Topologies. NEAT is a genetic algorithm for the generation of evolving artificial neural networks. It alters both the weighting parameters and structures of networks.

Neural Network - An artificial neural network is an interconnected group of nodes, based on the biological neurons in a brain. Nodes are used to represent artificial neuron and connections allow the output of one neuron to be the input of another.

Node - Layers are made up of several interconnected nodes. There are 3 groups of nodes, input nodes, hidden nodes and output nodes.

Phenotype - A Phenotype is the representation of the genome. The genome is a rawer representation of the data. In the case of NEAT, the neural net created based on the genome would be considered the phenotype.

Population - A population is a collection of genomes. Each generation a new population is created from the previous population.

Reward - A value received due to the previous action. The scale varies between environments, but the aim is always to increase your total reward

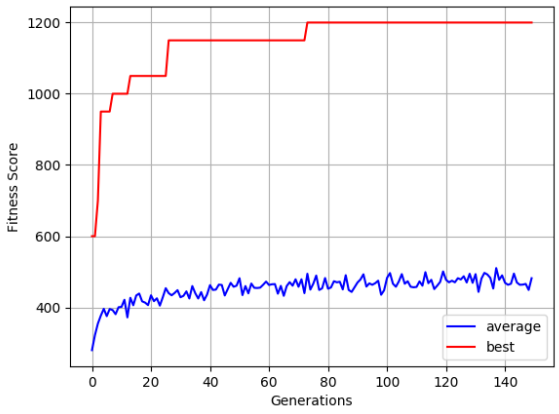
Species - Speciation in genetic algorithms is defined by using a function to compare the similarities between two genomes, and using it to group similar genomes into species.

Step - A time step in an environment. In the case of Atari games a step is a frame. In OpenAI's Gym, it is also a function through which an action is passed to through, and returns our observation, reward, done, and info data.

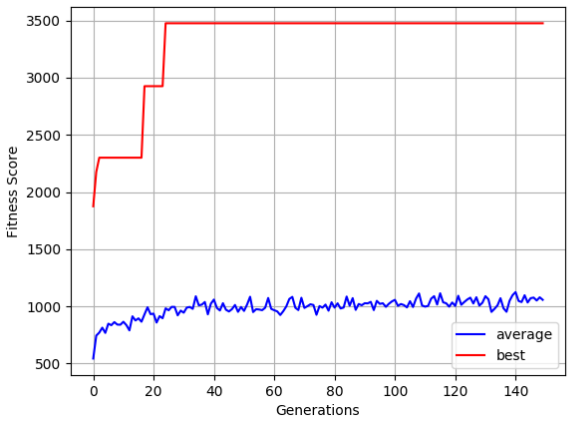
8.2. Fitness Graphs for NEAT and Custom GA Method

8.2.1. Custom GA

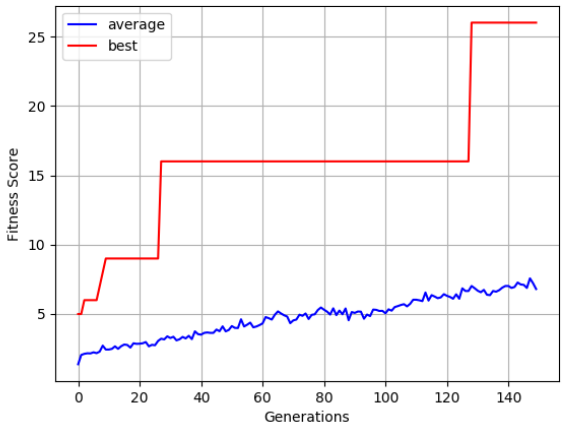
Airraid



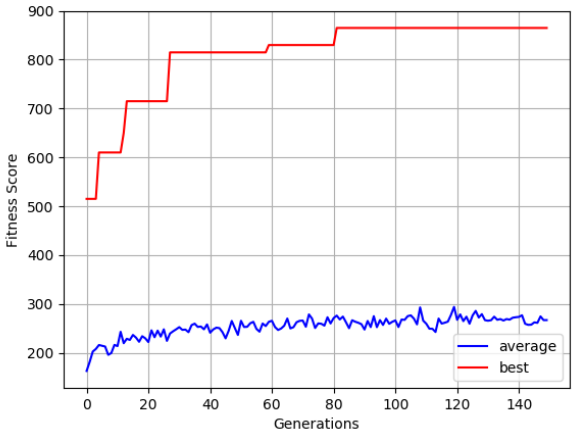
Asterix



Breakout

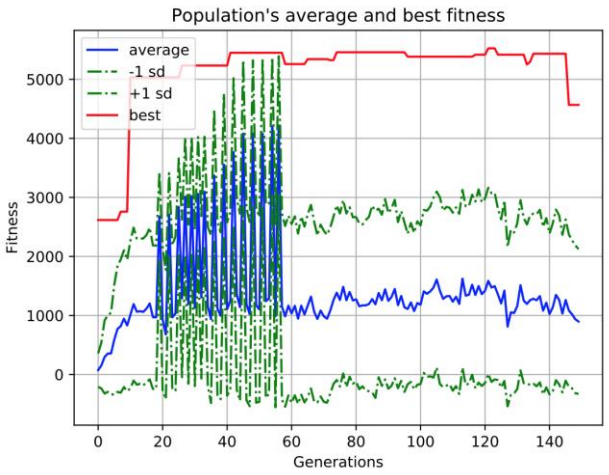


Space Invaders

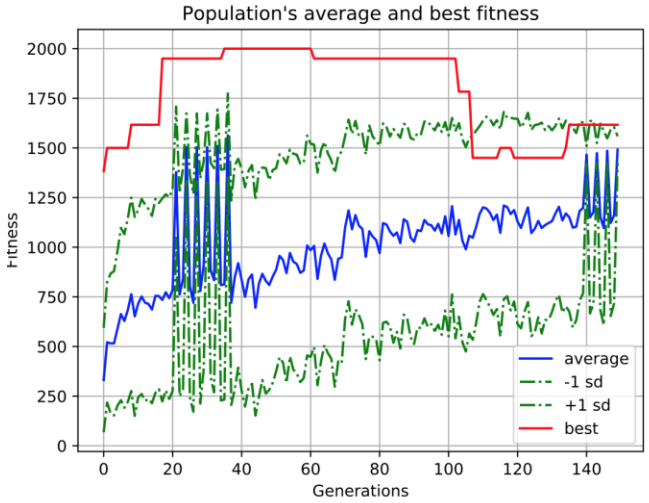


8.2.2. NEAT

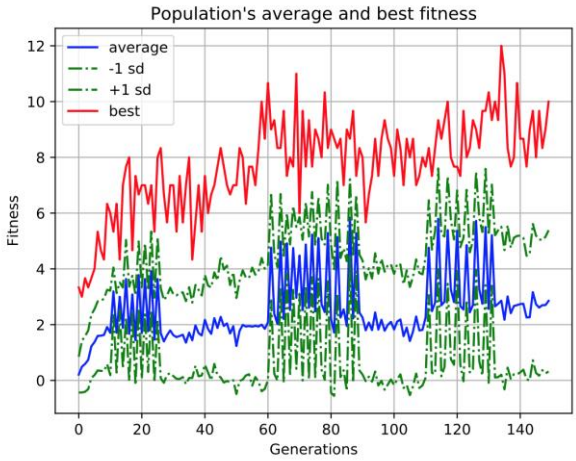
Airraid



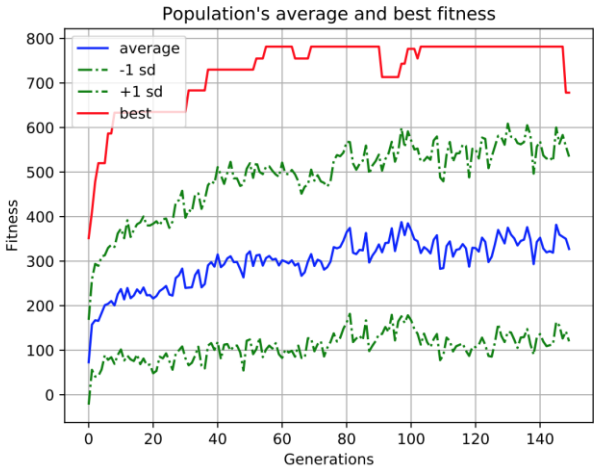
Asterix



Breakout



Space Invaders



8.3. Code Overview

Code used for the experiments shall be included here although in a shortened fashion, with non - essential or self-explanatory parts left out. For full reference please consult the source code.

8.3.1. Hard Coded AI

```
rom = "../roms/pong.bin"
total_reward = 0
ball_y = 0
player_y = 0
(screen_width, screen_height) = ale.getScreenDims()
while not ale.game_over():
    ale.saveScreenPNG("frames/frame-%07d.png" % k)
    img_rgb = Image.open("frames/frame-%07d.png" % k)
    box = (0, 34, 320, 194)
    # Crop out unnecessary objects from frame.
    region = img_rgb.crop(box)

    action = 0
    p = region.load()
    width, height = region.size
    for x in range(0, width, 1):
        for y in range(0, height, 1):
            if find_pixels(p, 92, 186, 92, x, y):
                player_y = y

            if find_pixels(p, 236, 236, 236, x, y):
                ball_y = y

    # Decide action based on height of ball in relation to player
    if (ball_y - player_y) > -3:
        action = move_down()
    elif (ball_y - player_y) < -12:
        action = move_up()
    else:
        action = do_nothing()
    reward = ale.act(action)
    total_reward += reward

print("Episode ended with score:", total_reward)
```

8.3.2. NEAT Code

```
game_name = 'AirRaid-ram-v0'
def evaluate_fitness(net, env, episodes=1, steps=5000, render=False):
    fitnesses = []
    for episode in range(episodes):
        inputs = game_env.reset()  # Receive observation from OpenAI as Inputs
        total_reward = 0.0
        for j in range(steps):
            outputs = net.activate(inputs)
            action = np.argmax(outputs)
            inputs, reward, done, info = env.step(action)
            if done:
                break
            total_reward += reward
        fitnesses.append(total_reward)
    fitness = np.array(fitnesses).mean()
    return fitness
def evaluate_genome(g, conf):
    net = neat.nn.FeedForwardNetwork.create(g, conf)
    return evaluate_fitness(net, game_env, args.episodes, args.max_steps, render=args.render)
def run_neat(env):
    def eval_genomes(genomes, conf):
        for g in genomes:
            fitness = evaluate_genome(g, conf)
            g.fitness = fitness
    config_path = os.path.join(local_dir, 'openAI_neat_config')
    conf = neat.config.Config(neat.DefaultGenome, neat.DefaultReproduction,
                              neat.DefaultSpeciesSet, neat.DefaultStagnation, config_path)
    pop = neat.population.Population(conf)
    parallel_evaluator = neat.parallel.ParallelEvaluator(args.numCores, evaluate_genome)
    pop.run(parallel_evaluator.evaluate, args.generations)
    best_genome = pop.best_genome
    env.close()
game_env = gym.make(game_name)
run_neat(game_env)
```

8.3.3. Custom AI Code

```
# Configuration of GA parameters below
maximum_population = 100
mutation_rate = 0.02
start_time = time.time()
steps = 10000
generations = 150
best_genomes = []
average_fitnesses = []
elitism = 20  # 10
spawn_rate = .25
culling_rate = .5
hall_of_fame = []
env = gym.make(game_name)
pop = population.Population(mutation_rate, maximum_population, steps, env.action_space.n, elitism,
spawn_rate,culling_rate)
def evaluate_pop(pop):
    for genome in range(maximum_population):
        env.seed(0)
        total_reward = 0
        for t in range(steps):  # Length of game
            action = pop.genomes[genome].genes[t]
            observation, reward, done, info = env.step(action)
            total_reward += reward
            if done:
                break
        pop.genomes[genome].set_fitness(total_reward)
def run(p):
    evaluate_pop(p)
    best_genomes.append(p.get_best().get_fitness())
    average_fitnesses.append(p.get_average_fitness())
    hall_of_fame.append(p.get_best())
    p.natural_selection()
    p.generate()
for gen in range(generations):
    run(pop)
```