Honours Project (CCIS)

INTERIM REPORT

2019-2020

Submitted for the Degree of: BSc Computer Games (Software Development)

Project Title: Assessing the suitability of NEAT in the implementation of a traditional AI found in Role Playing Games.

Name: Fotios Spinos

Programme: Computer Games Software Development

Matriculation Number: S1625069

Project Supervisor: Hamid Homatash

Second Marker: Mario Sofrano

Word Count:
(***excluding contents pages, figures, tables, references and Appendices***)

"Except where explicitly stated, all work in this report, is my own original work and has not been submitted elsewhere in fulfilment of the requirement of this or any other award"

Signed by Student: Fotios Spinos **Date: 22/04/2020**

Abstract

The industry standard for developing enemy AI in RPG games relies on arguably traditional approaches like finite state machines and behaviour trees. Breakthroughs in the field of Machine Learning (ML) made possible the development of programs through what is considered to be a new way of programming led by data instead of instructions. The NEAT algorithm combines two well established approaches in the field: Neural Networks (NN) and Genetic Algorithms (GA).

The hypothesis made by the research paper suggests that a common enemy AI found in Role Playing Games (RPG) can be implemented through neuroevolution. The final deliverable aims at producing an agent approaching and attacking a player object in the most effective way possible. The outcome of this project will produce data and assess the suitability of NEAT.

The benefit produced from using NEAT can potentially change the workflow of modern game creation as effort on explicitly programming enemy behaviours will not be required. To prove the hypothesis a training environment was developed in Unreal Engine 4 (UE4). The environment integrates the implementation of NEAT developed by the University of Texas at Austin. The creation of the environment has proven to be successful as data retrieved during training were stored and visualised.

Through observation and data gathered, the agents did not meet the expectations set by the research's hypothesis. The majority of tests showed that agents could not identify the best path leading to the player object. In addition, the population could not adapt to new player positions and showed no improvement throughout the training process. Nevertheless, agents could not attack the player in any of the conducted tests. Training produced agents orientating and moving towards the player while considering surrounding objects originating from the environment. Rays emitted from the agent have proven to be beneficial as agents respond ideally when identifying walls and the player object.

A severe problem during training would involve agents showing a behaviour exceeding the population's standards. These agents did not influence the population and were commonly extinct in consecutive generations. Changes in the configuration used by NEAT could produce better results. This process has proven to be time consuming and could admittedly add a layer of complexity as several variables affect the evolutionary process. Other machine learning techniques making use of fixed structure NN can remove this level of complexity and produce more suitable agents.

Acknowledgments

I would like to take this opportunity to thank my supervisor, Hamid Homatash for all the support he has provided throughout the research. The suggestions Mr. Homatash has made have undoubtedly shaped and increased the quality of this project.

I am grateful for my family and friends who supported me in this journey.

Contents

Table of figures

1.0 Introduction

1.1 Project Overview

1.1.1 Machine Learning Background

There is no doubt that the field of machine learning has gained a lot of recognition as a plethora of applications and projects have surfaced in recent years. Projects like Google's deep mind and other API's like Keras and Tensorflow have attained the attention of developers and users alike. Virtual personal assistants, videos surveillance, email spam, malware filtering, search engine result refining and product recommendations are just a few applications of the field.

Taking the mentioned examples into consideration, it would be reasonable to question why ML is necessary instead of traditional artificial intelligence techniques. Machine learning by definition is not explicitly programmed, instead learning and improving is possible through data fed to the agent. Arguably this field introduces a new way of programming driven by data.

Applications like the spam filter use labelled data or in this case examples of emails flagged as spammed or non-spammed. The AI uses the data without relying on instructions specified by the programmer. This makes the software reliable and expandable as long as the appropriate data set is used. The problem of classifying data such as spam emails can be very large in scope, as spammed emails do not follow a specified pattern. Without ML developers would have to identify patterns found in spam emails and explicitly program instructions for each one of them. The problem is that these explicitly programmed instructions will exclusively work on the given or similar data.

In conclusion, it takes an unreasonable amount of effort to provide the system with new instructions with the sole purpose of keeping the software up to date. In other words, this solution does not generalize the task, making its use limited and impractical.

1.1.2 Deep Learning and Neural Networks

The concept of Neural Networks and Deep Learning exist for over 50 years. Because of hardware limitations at the time, the amount of training time and data needed for the desired behaviour made DL insufficient to use. In recent years hardware advancements overcame these limitations leading to the massive attention NNs have today.

Deep learning is a subfield of ML which makes use of neural networks for its implementation. Neural networks are heavily inspired and aim to mimic the human brain by taking into consideration how neurons and neuron connections cooperate to provide us with our cognitive abilities. It is essential to clarify that brain neurons are way more complicated than NNs but some major concepts like the artificial neuron and neuron connections are present.

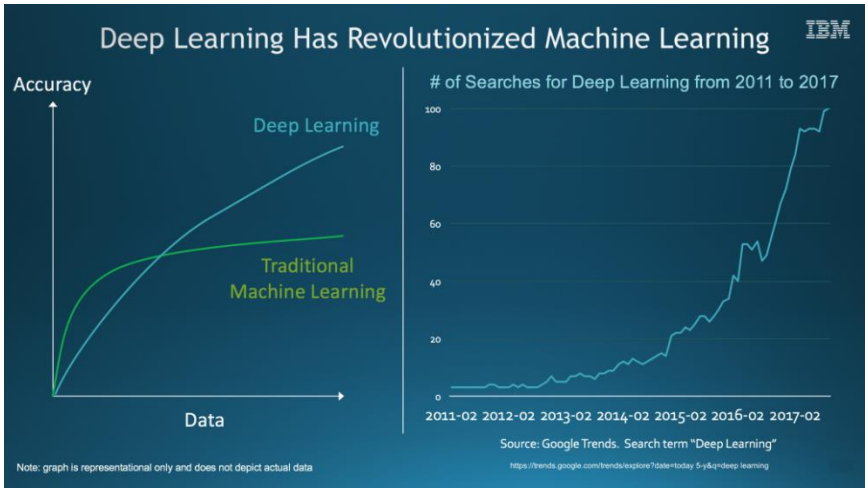


Figure 1The impact of deep learning (Sumit G., 2018)

1.1.3 Neuroevolution

The approach of Neural Evolution of Augmented Topologies (NEAT) is said to outperform fixed-topology neural networks making neuroevolutionary the preferred approach in a number of fields. The video game industry in no stranger to this approach as projects using this implementation have gained great recognition from the outcome of their projects. The most recognised projects involve the implementation of a neuroevolutionary networks which are Capable of learning to play a game. Google's self-learning AI Alpha Zero has learned to play several games outperforming human players.

What makes an Evolving Neural Networks unique is that they do not have a predefined structure. The network starts with a minimal structure. Changes called mutations happen during runtime to determine the best performing neural network.

1.1.4 Game Development and Machine Learning

Taking everything into consideration, machine learning is not the standard for AI implementation in games development. Most recognised games rely on more traditional approaches like finite state machines and behaviour trees for developing enemy AI. Explicitly writing the rules of a complex AI is undeniably a time-consuming process. Implementing enemy AI can be expensive for a game studio as many processes and employees from different principles need to ensure that the agent behaves as designed. This applies in most Role-Playing Games (RPGs) as the enemy AI behaviour is defined by gathered from detailed environment. As such, designing and implementing the decisions and behaviour of an NPC (Non-Playable Character) can be particularly challenging. Relying on ML techniques can potentially speed-up development as the developers will not have to explicitly program the rules followed by the agent. The agent can observe the environment and act based on retrieved data for the purpose of maximising its fitness.

1.2 Project Overview

1.2.1 Project Outline

The research question of this paper is as follows:

Is the application of neuro-evolution suitable for the implementation of a traditional enemy AI in Role Playing Games?

The focus is to produce an agent who chases and attacks a player object while avoiding collisions with walls observed in the environment. This suggests that the agent should execute the underlined behaviour in the most effective manner. It is not a consideration to optimise the agent's behaviour so that it suits the design of a game. The behaviour produced by the AI agent will not be altered in the context of making a game more enjoyable. Producing the best possible results is one of the main objectives of this paper. Building an agent who deliberately avoids attacking and chasing the player in the most effective way would add an additional layer of complexity.

The agent is meant to represent a character with the ability to navigate the environment by walking. With that being said, the agent will be affected by collisions with other objects in the scene such as walls or the player object. The agent can move forwards or backwards and rotate left or right. It will be up to the agent to choose the fastest way to approach the player object and conduct the greatest number of successful attacks possible. Elaborating on the research question, a suitable approach suggests that the described behaviours will be followed by the neuroevolutionary agent.

Other neuroevolutionary approaches can be proven to be more suitable for the presented task. Using and analysing the results produced by other approaches is not the focus of this research. The NEAT algorithm is well established and provides the base for other neuroevolutionary techniques.

1.2.2 Project Aims and Objectives

Literature Review Objectives

- Examine Neural Networks and Genetic Algorithms
  It is important to gain a general understanding on other neural network architypes and concepts introduced by GA before investigating neuroevolution. The NEAT algorithm aim to restructure a basic structured neural network to maximise fitness. Taking that into consideration, a range of terminology and concepts learned from this objective will support the implementation of the final product. It is also significant to research other implementations of NEAT applied in game development. This stage will involve relevant libraries and techniques supporting the process of building the agent along with algorithms used for producing the appropriate fitness level.

- Investigate suitable development environments
  The purpose of this objective is to support the implementation of the final product. It is at most significance to identify a stable and established environment to cover the needs of the project. Since the project is based on video game development, game engines like Unity3D and UE4 are strong candidates.

Primary Research Objectives

- Develop a traditional RPG based enemy AI using NEAT
  After most of the research phase has been completed, the development of the AI agent will be prioritised using the knowledge gained before or during the implementation process. Machine learning techniques can be unpredictable leading assessments and refinements to ensure that the final product meets the set criteria.

- Gather and analyse data obtainable from the final product
  Gathering and analysing data from the finished project is the key to conclude and answer the research question. These data will include the agent's behaviour, the time of trainings and the structure of the neural network.

- Assess the outcome of the project against traditional enemy AI behaviours
  Once all observations and data have been gathered, we will be able to draw out a conclusion. Assessing the outcome of the machine learning agent with traditional enemy AI behaviours found in RPG games will ensure that the conclusion is accurate and covers possible advantages, disadvantages and other differences.

1.3 Hypothesis

The creation of an enemy AI found in RPG games is possible through the use of NEAT as data gathered during training will produce the most effective behaviours for chasing and attacking the player object while avoiding collisions with walls originated from the environment.

2.0 Literature and Technology Review

2.1 Investigating Neural Networks and Genetic Algorithms

Back in 1957 Frank Rosenblatt developed the "Perceptron" which is known to be the first neural network. It consisted of one or multiple inputs, a processor and an output. This network follows the feed forward approach. As the name suggests, this type of network passes the data received from the left where the input neurons exist to the right where the output neurons are present.

In current neural network architectures, between the input and the output layers are one or more hidden layers. The idea produced by the Perceptron was so significant where neural networks are now present in every area of machine learning. The idea of a neuron and neuron connections were heavily influenced by brain neurons.

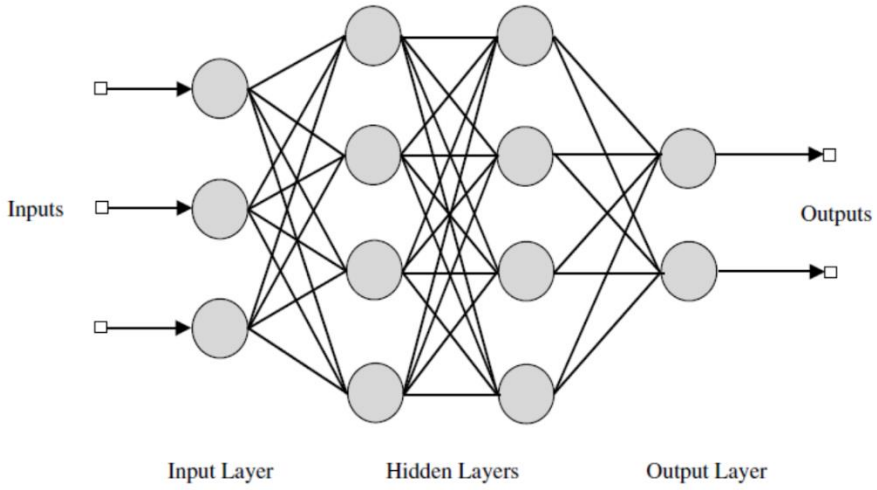


Figure 2 Representation of a feed forward neural network (Choudery H., 1018)

The nature of code written by Daniel Shiffman the director of The Processing Foundation provides an implementation and a detailed explanation on the "the Perceptron". Artificial neurons have a value and a weight for its corresponding connection. Feed forward neural networks follow the same principle introduced by Frank Rosenblatt. The value of a neuron is calculated by the sum of every connected node in the previous layer multiplied by the corresponding weight. This sum is passed to an activation function to ensure that the output received is within a certain range. Many architectures use the sigmoid function which has a range of [-1, 1]. The activation function has a great influence on the network leading to more opportunities for research.

In video game development the areas of supervised and unsupervised learning are not commonly used in the implementation of an enemy AI. This is because video games are "dynamic" and actions can change the state of the game. That is typically the reason why reinforcement learning researches are carried out in game environments. By nature, reinforcement learning makes observations of the environments and receives a reward or punishment depending on its output. The AI uses the observations made and attempts to maximise its reward.

In 2002 the MIT Press Journals released an article titled: "Evolving Neural Networks through Augmenting Topologies" (O. Stanley & Miikkulainen, 2002). This research paper presents a method which as claimed, can outperform fixed topology neural networks. Since performance is a great concern in the video game industry NEAT and other variations are mostly used by data scientists in an attempt to create game playing agents. It is often the case, that the agent can surpass even the most experienced players. Neuroevolution makes use of both NNs and GAs.

NNs can be thought of as function optimisers. The final product will involve agents observing the environment state (inputs) and choose to carry out an action based on it. Genetic algorithms on the other hand can be expressed as "naive" to mimic evolution. By naïve the paper suggests that changes happening to the agent's structure are selected in a random manner. The book Computational Statistics & Data Analysis suggests that GA are stochastic optimization tools that work on "Darwinian" models of population biology and are capable of solving for near-optimal solution for multivariable functions without the usual mathematical requirements of strict continuity, differentiability, convexity and other properties(Chatterjee S.& Laudato M. & A.Lynch L, 1996). The implementation of NEAT will use GA to alter the NN properties of each agent to produce a new population. On each generation, the best performing agents will influence the rest of the population through crossover.

The process of crossover produces a new offspring formed by combining the chromosomes of two selected organisms. Typically, agents using this approach contain a collection of numbers used to describe their behaviour. Combining these number can produce new organisms. The algorithm picks the best performing agents also referred to as organisms and “mates” them with a portion of the population. Influencing the population with genes resulting to high fitness improves the population but does not allow the exploration of new behaviours. Mutations are responsible for altering the agent’s genes in a random manner. The organisms produced by the NEAT algorithm use NNs to present the agent’s genes. Mutating the structure of a NN can include adjusting, adding and removing nodes and links. Most of these concepts depend on the implementation of NEAT along with configuring the variables used by the algorithm. These variables control the probability in which NEAT operations affect each agent in the population. For example, the NEAT implementation might allow the developer to set the probability an agent being mutated on each generation. A generation occurs when a new generation replaces the current one.

One of the most famous examples among the gaming community is Mario I/0 (Clinton Nguyen, 2015). This software uses NEAT to learn how to complete a level in Super Mario. Starting up from a very simplistic structure the AI tries to identify the controls and ends up staying idle. When the agent finds the button to move forward, Mario begins to walk resulting in a positive reinforcement. This will not be enough as the level consists of enemies and deadly barriers. After many mutations (changes in the structure of the neural network) the neural network formed can complete the level.

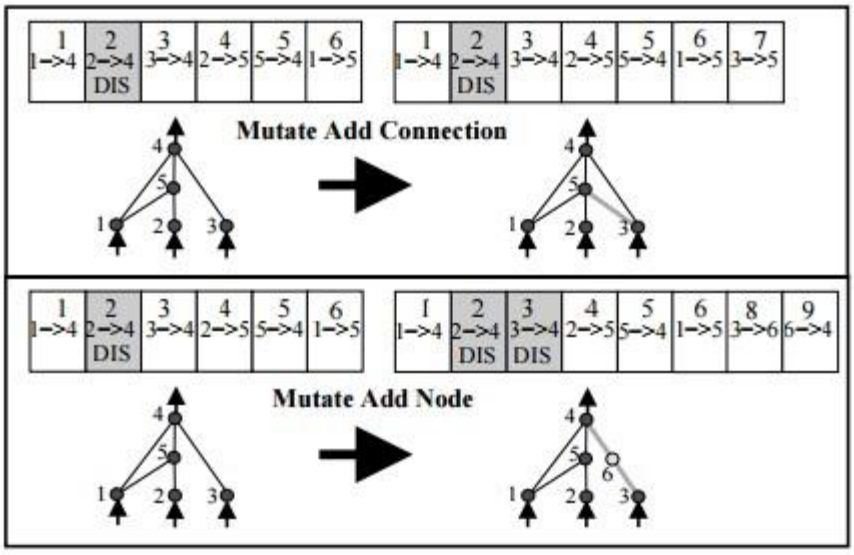


Figure 3 Mutations in evolutionary networks (Hunter H. , 2019)

A research paper: “Neuroevolution in Games: State of the Art and Open Challenges” highlights all the common approaches and applications of neuroevolution in video games (Risi & Togelius, 2015). The terminology and methodology in a plethora of projects goes through the calculation of fitness, input types, output along with how these networks evolve to give the desired results. Relevant to this research paper is the method of using angle sensors as input to the neural network. This method is mostly used in the creation of self-driving cars in virtual environments and presents the opportunity of using the agent in multiple environments under the assumption of the same conditions being preserved.

A self-driving car for instance maximises its reward by driving towards the destination point and receives a punishment when it hits a wall. Rays emitted by the car can return the distance to the wall

they collide with. This distance can be used as input to the NN making the AI capable of driving in different maps with no additional training. This approach can produce an expandable and stable agent as the behaviour relies on the data produced by the rays. Therefore, rays allow the agent to observe its surroundings. The AI agent of this research could use sensors used to detect if the object is approaching a wall in the virtual environment. The ideal outcome would produce an agent avoiding walls and moving towards the player. The agent should perform the desired behaviour on different environments after training.

A great amount of machine learning approaches requires that a fixed amount of NN inputs and outputs are defined before training. To test the agent's behaviour, the research will take a deep look on results gathered from different environments. These environments will differ from each other as object placement and amount will differ. Representing the environment is unsuitable as the number of objects on each scene will not be the same.

Figure 4 Footage of implemented agents used in the research: "Emergent Tool Use from traction" (OpenAI, 2019)

A Project where agents use rays to observe their surroundings comes from OpenAI. This project investigates how complex behaviours can arise from using multiple agents in a game of hide and seek (OpenAI, 2019). As seen from the image above the green rays are emitted in all angles around the agents. Each one of these rays hit the surfaces of objects. The information gathered from each ray include the distance and the type of object being hit by each ray. These data are represented by numerical values and are passed to the agent's NN.

2.2 Investigating available resources and techniques for the implementation of the project

Taking into consideration that the final product is related to game development, it is critical to use tools supporting the needs of our project and allow the extraction of data gathered in the training environment. Game engines like Unity3D and UE4 have a long history of development providing a stable environment for the creation of games. Both engines provide a variety of tools to support game development.

Unity3D was first launched in 2005 and from then a wide collection of games have released on console, PC and mobile devices. Unity's primary scripting language is C# but there is also some support for creating scripts using Javascript (Haas, J). UE4 supports blueprints and C++. It is worth mentioning that some plug-ins allow the creation of scripts using other programming languages like Python. This produces a very strong candidate for this project since Python provides a rich variety of modules. Most machine learning projects are implemented in Python using high level libraries like Keras, Scikit-Learn and Tensorflow.

It is very likely that information like the distance and the position of an object will be used as input to the NN. Both game engines have their own implementation of a transform and vector class as every GameObject or Anchor entity has a transform component which provides setters, getters and transformation functions for the scale, rotation and position of the object.

As mentioned in the previous section of the literature review the approach of using angle sensors can produce a robust agent unbound to changes in the environment. For the creation of the angle sensors, we can use a physics engine to conduct multiple raycastings to retrieve information on the objects surrounding the agent and the distance to the collision point of the ray.

Python is now one of the most famous programming languages providing a great collection of libraries for machine learning. A few game engines like PyGame provide support for the language. The great advantage produced from using PyGame is that a great collection of tools can automate many parts of the development process. Considering that most of the game engines supporting Python were built just a few years back, we can expect that they might not be as stable as the engines mentioned in the previous paragraphs. In addition, the standard programming language for game development is C++ which both Unreal and Unity use in their core. Adding to that, the most relevant sources found during the research phase include implementations of NEAT in C++.

Taking everything into consideration the machine learning agent will be built on UE4. This environment provides many tools for the implementation of the project. The physics engine supports recasting which is ideal for the development of angle sensors. The scene will include two types of objects: The player and the walls constructing a level. Using Unreal's tag system, object types can be defined from the editor. The distance between the player to each collided point will result as the first set of inputs retrieved by the rays. The second set of inputs will represent the type of object the ray collided with. There is a variety of online resources on how to use UE4 including a very detailed documentation which can be found in Epic's official website.

The book "AI for Game Developers" written by David Bourg and Glenn Seemann provides several implementations and examples of neural networks written in C++ (Bourg & Seemann, 2004). These authors are well recognised for their contribution in game development, making the use of the source reliable as a learning material and suitable for the development of the software.

To support development further the creation of scripts will be made in Visual Studio 2019. This software is an integrated development environment (IDE) which provides support for developing C++ files. Unreal engine introduced an extension for Visual Studio called UnrealVs which provides several features to make development convenient.

3.0 Execution

3.1 Research methods

A literature review was conducted, and a great variety of tools and methodologies were identified for approaching the development of the project. Achieving the desired outcome will involve a great number of attempts as we need to experiment with multiple fitness functions, agent observations and NEAT configurations. The research will follow a develop and test approach to identify how changes to the agent and environment influence the behaviours originating from the population. This method is the most suitable as exploring and comparing multiple approaches can lead to proving the researches' hypothesis. Machine learning algorithms are usually treated as a black box, since we cannot accurately explain why agent decisions are made. This is also the case with neuroevolution since changes occurring in each agent's NN are partially random. Therefore,

gathering data from multiple tests can provide a better understanding on agent behaviours and lead to a fully justified answer to the presented research question.

Observation provides the means to a subjective and justified answer based on the produced behaviour and evolutionary process resulted from each generation. Bob Faux (2001) defines observation as a combination of using the senses to experience an actual event followed by reflection of what was experienced. This method will be used to identify agent behaviours and changes in the population resulting from each generation. This is an appropriate research method as the agent is not affected by the observer. One of the challenges produced is to build an environment which allows us to gather data and observe the behaviour of individual agents in multiple tests. The author is the most appropriate observer due to his familiarity with the research. This includes the goals and set hypothesis. The author is also responsible for developing the training environment in UE4, and so issues with the functionality are likely to be identified because of this familiarity.

To gain a better understanding on how the population evolves, the research will rely on qualitative and quantitative data. The training environment will store quantitative data on each generation. Information on the average fitness of the entire population, the best recorded fitness and the number of successful attacks will help determine if the training process is successful. The fitness function will encourage agents to follow the behaviour set in the hypothesis. The mentioned values retrieved during training should increase in future generations.

While quantitative data provide information on the evolutionary process and agent behaviours, observation can help identify actions taken by the agents during runtime. It is important to observe agents during training to verify how agent behaviours change as generations occur. This research method allows to quickly and efficiently produce some conclusions on the population's behaviour driven by operations provided by the NEAT algorithm. Overall conducting observations is quick and provides the mean to explain quantitative data through behaviours.

3.2 Identified functional and non-functional requirements

Functional requirements

1. Implement the agent object
2. Construct a training environment
3. Provide visualisation of data gathered during training

The agent provides the necessary functionality for turning NN outputs into actions. The agent will have the ability to turn left/right, move forwards/backwards and attack. One of the main purposes of this object is to feed a NN with a fitness and observation values. The agent will assess its state resulted from the NN outputs and produce a fitness value.

The creation of the agent is one of the key components required for developing the training environment as the fitness resulted from NN inputs and outputs define the most optimal agents used to influence the population. The training environment requires two objects: the agent and player. Walls can be optionally added to assess the agent's ability to respond to surroundings originated from the environment. Since problems could arise from the implementation of NEAT, the software architecture will allow to conveniently add and use multiple libraries providing the algorithm. To debug agent actions, the architecture will allow to manually control agents based on inputs given by key codes. The training environment will use the state pattern so that changes and additions (like integrating other NEAT libraries) do not affect other parts of the software. The environment state will produce agent inputs based on retrieved fitness and observations values

Organising and visualising training data will provide insight on agent behaviours and evolutionary process. Graphs can clearly display how population fitness varies on each generation. The values appropriate for analysing the population are: Average population fitness, best recorded fitness and average number of attacks. These values can be graphed against the generation number to monitor behaviours and the overall training process. If training is successful, the values mentioned will increase as generations occur.

Non-functional requirements

1. Providing the capability of storing trained agents
2. Launching trained agents
3. Allow the user to manually control the agent

Storing and launching trained agents will deepen our understanding on agent behaviours. Additional tests can be conducted from agents launched to new environments. Agents adapting to a trained environment should showcase the same level of performance in other environments following the same characteristics. The agent will be isolated allowing to observe and gather data from individuals. A limitation of the presented functional requirements is that agents are analysed collectively. This feature is a non-functional requirement because data gathered during training along with observing agent behaviours can produce a justified answer to the research question.

Manually controlling the agent is not necessary for training and producing the agents. This feature is meant to aid the project in temps of debugging. Since the user controls the inputs given to the agent, the developer can observe and analyse a determined behaviour is followed by the agent.

4.0 Analysis

4.1 Requirements and considerations

Before considering strategies for approaching the fitness function and how to effectively use the NEAT implementation, we decided that the environment would allow the user to conveniently pass and retrieve data from the NN operated by each agent. Since the author has never used NEAT, we decided that the training environment would allow the developer to support multiple implementations of NEAT without having to make alterations to the rest of the software. This would allow to identify and try out multiple implementations conveniently if any problems arise during development.

It is therefore a consideration to identify a design pattern allowing to decouple functionality, simplify the software and take advantage of the engine's capabilities. It would be ideal to decide on the environment's behaviour before run time so that a different goal is met. The engine's editor for instance could provide a combo box underlying available environment behaviours chosen by the developer. In this case, the supported environment behaviours could represent the user's manual control and operating with the NEAT implementation. In manual control the user can operate the agent's through inputs captured by the game engine allowing identify any abnormalities caused by the agent. The design pattern allows to maintain functionality used by the environment without affecting other parts of the software. If issues with the implementation of NEAT are identified, the developer can produce a new state integrating another library.

4.2 Visualising training data

Visualised data can be analysed in a convenient manner. Training data and observations will lead to the conclusion of this research paper. Graphically displayed data can show the population performs on each consecutive generation. A rising slope in the population's average fitness vs generations graph suggests that the population improves. A best generation fitness vs generation graph can show how agents with a dominant behaviour influence the population. It is expected that agents with the highest fitness values will pass their behaviour to other agents increasing the population's average fitness. If this hypothesis is not met, that would suggest that the evolutionary process is not conducted successfully as optimal behaviours are lost during training. The average successful attacks vs generations graph can be used to identify how successful attacks corelate with the mean population fitness. The optimal agent would maximise the number of successful attacks to the player object. In addition, behaviours arising from running the program can be compared with other runs to identify if the same results are produced along with any changes in the number of generations occurring before the population shows clear signs of improvement.

5.0 Design

5.1 Software Planning

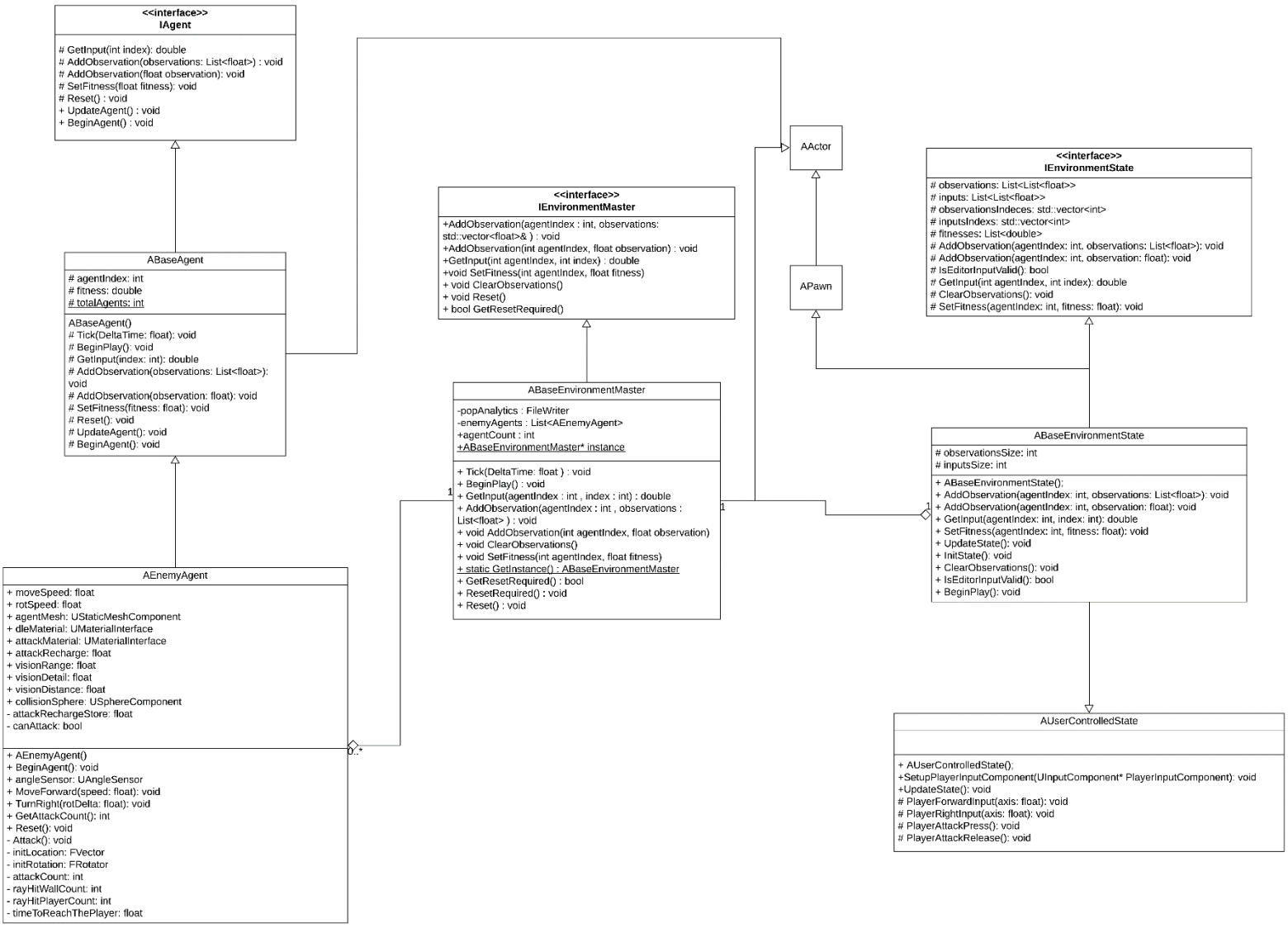


Figure 5 Environment scene UML diagram

The previous section provides a set of identified requirements arising from the conclusion of the literature and technology review. The purpose of this section is to identify a software structure to cover the underlined requirements.

The enemy agent should inherit from a base agent class providing the core functionality for retrieving and passing input to the neural network used by the agent. This approach makes the functionality reusable as the developer can inherit from the class to create custom agents. The enemy agent should only describe the enemy's behaviour and fitness function. Providing unique functionality can be achieved through overriding agent functions.

Environment states describe the desired functionality before running the software. We can expect to provide at least three environment states for manually controlling the agents, training the agents and launching trained agents. The state pattern allows to support multiple behaviours and integrate

additional implementations of NEAT if the need arises. Also, states allow to conveniently assign the desired functionality followed by the environment and decouple parts of the software. The environment master will manage its assigned state by initializing and updating the object on each frame.

5.2 Training Scene

The project will contain multiple training environments, each providing their own set of challenges. Each environment will be used to gather data and assess agent behaviours. The first environment will only involve the agent and the player object. This is going to be the simplest of the environments giving us a clear insight on how effectively the population figures out how to approach the agent object given their current position. The purpose of this scene is to assess how agents approach the task in a simple environment. This is considerably the simplest test and should provide some desired behaviour before moving on to other assessments. When the agents show the ability to move towards the player, we can confirm that the environment, fitness function and NEAT configuration can potentially produce the desired agent behaviours.

Fixed behaviour AI can conduct actions based on the player's state. AI based on state machine and decision trees produce behaviour through data received from the player. This can involve player actions like attacking, stealing, or talking to NPCs. This information is gathered in real time and AI agents should demonstrate adaptivity to a dynamic environment. The test mentioned in the previous paragraph does not provide any dynamic behaviour other than the agent's own actions. To test if the agents can adapt to new player states the project will include a test which teleports the agent to a new random position defined on each generation.

One of the goals of this project is to produce agents capable of adapting to a given environment. For that reason, one of the tests will surround the agents and the player with walls. This environment will be used to determine if the agent can avoid detected walls and reach the player. The last environment will be used to assess how the population responds to walls partially blocking the path leading to the player object. Arguably the most difficult test as paths are thinner and agents are expected to walk around the walls before they can reach the player object.

The number of agents spawned in each environment will influence the amount of behavioural variation contained within the population. While variation is important, the addition of multiple agents can have severe affects on performance. A good balance needs to be found between the available computing power and number of agents in the scene. A successful training environment would make use of the library conducting mutations and crossovers with each agent in the population. The desired outcome involves mutations assisting in exploring new behaviours and crossovers passing genes resulting to high fitness values to the rest of the population.

5.3 Storing and visualising training data

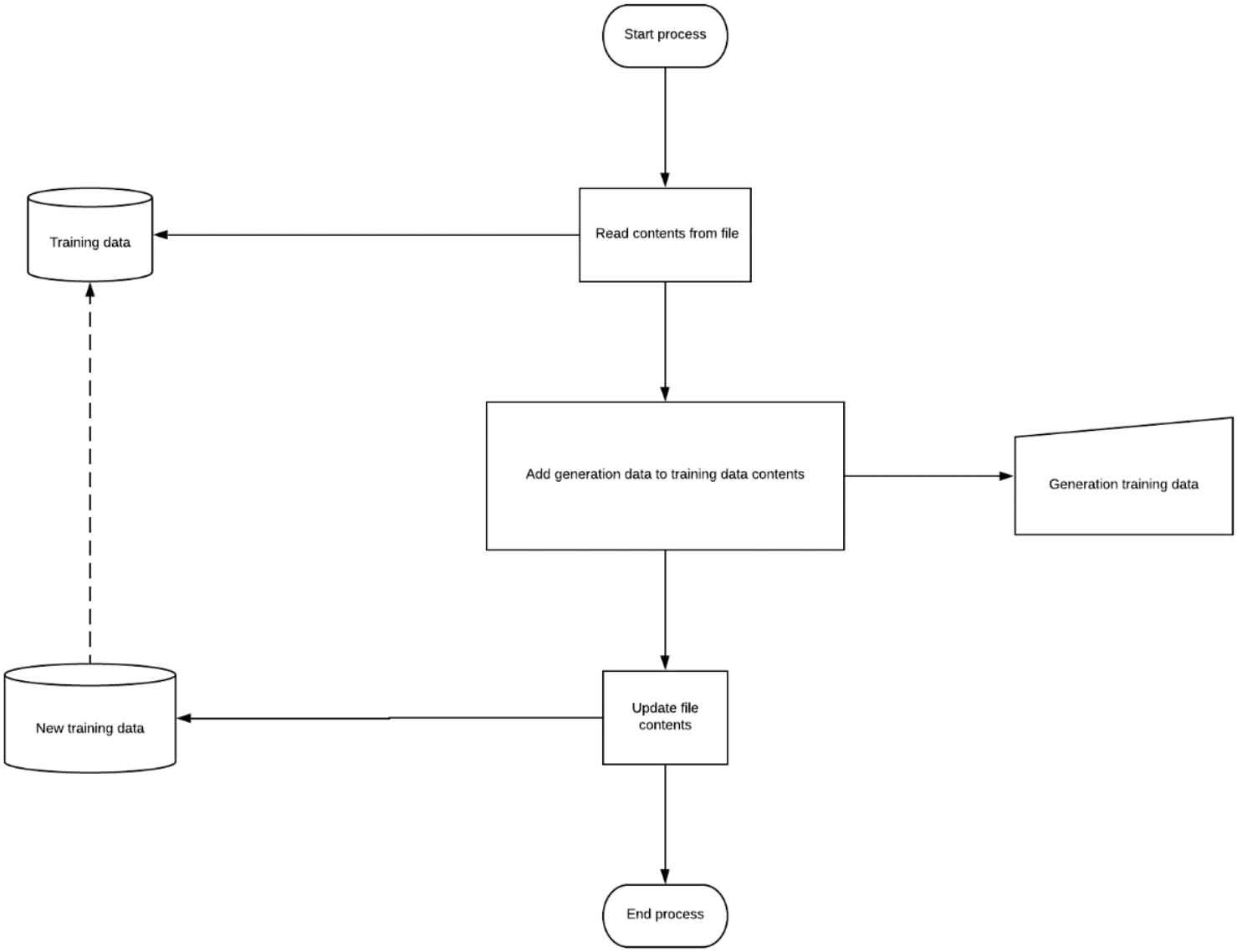


Figure 6 Flowchart for storing data gathered from each generation to file

The research paper has expressed the need to gather and analyse data gathered during training. The data will be stored in a txt file and processed by a Python script for visualisation purposes. Storing data to the txt file will occur on each generation. Data already stored in the file from previous generations should not get overwritten. For this purpose, the process of storing data will involve reading the already stored contents of the file, adding the data from the current generation and finally storing them to the file. The class responsible for writing data to a file should retrieve the generation data as input so that when a request for storing data occurs the already given data are added to the file.

Pass data from visualising data to here and say that it is easier to compare and analyse data organised in graphs than comparing fitness values produced by each generation.

To visualise the training data, the project will on Python and matplotlib. The Python script used to generate the graphs will be written in a Jupiter notebook. Therefore, the visualisation of data takes place in the browser. This approach allows to store the visualised data in an HTML file. This provides

a convenient solution for inspecting and analysing stored data. The Python module is easy to use and covers the project's needs.

5.4 Angle ray

The literature review identified the need for observing the agent's surroundings. Rays surrounding the agent object could provide information on distances and types of objects "hitting" the rays. The rays will be emitted from the agent and gathered information will be fed to the NN controlled by each agent as surrounding objects should influence its decisions.

To implement a tool providing us with data originated from the environment, we constructed a custom UE component. The benefit produced from this approach is that the engine makes all the necessary calls to actors and their components so that all objects update and initialize sequentially. The use of components provides an expandable solution as multiple components can be attached to actors for producing unique behaviours. These components can be reused in the implementation of other Actors.

6.0 Implementation

6.1 Angle Sensor Component

The angle ray is implemented in the UAngleSensor class. To implement an Unreal Engine component the UAngleSensor class needs to inherit from UActorComponent. This class provides a number of virtual functions allowing to provide custom functionality. The two overwritten functions are:

```
virtual void TickComponent(float DeltaTime, ELevelTick TickType,
FActorComponentTickFunction* ThisTickFunction) override;
virtual void BeginPlay() override;
```

The TickComponent function is called once on each frame. The BeginPlay function is called when the software executes. These or similar functions are present throughout the project. In most cases BeginPlay contains instructions for initialisation, or includes initial functionality while Tick provides the functionality of the actor or component executed on each frame. These functions are called by the engine.

6.1.1 Ray casting

To emit a ray and check if an object is "hit" by it, the method uses the following code fragment:

```
if (GetWorld()->LineTraceSingleByChannel(outHit, start, end, ECC_Visibility,
collisionParams))
```

Ray directions are defined at the EmitSensorRays method. The origin is a vector used to measure the angle formed between the agent's orientation and the origin's direction. To measure the formed angle, we used the dot product. This angle is essential as generated rays will depend on the agent's rotation. The other rays are determined by the agent's forward vector and a defined angle offset. This offset is a float representing the angle formed between each consecutive ray. Rays are spawned to the left and right side of the forward vector. The purpose is to add the offset to define the ray

laying at the right side of the forward vector and accordingly subtract the offset to find the angle laying at the left side of the forward vector.

To conduct a ray-cast, the software must provide a start and end point. These parameters are represented by vectors. Therefore, angles need to be described as vectors oriented by the agent actor. The x component of the end vector is defined by the cosine of the angle and the y component by its sinus.

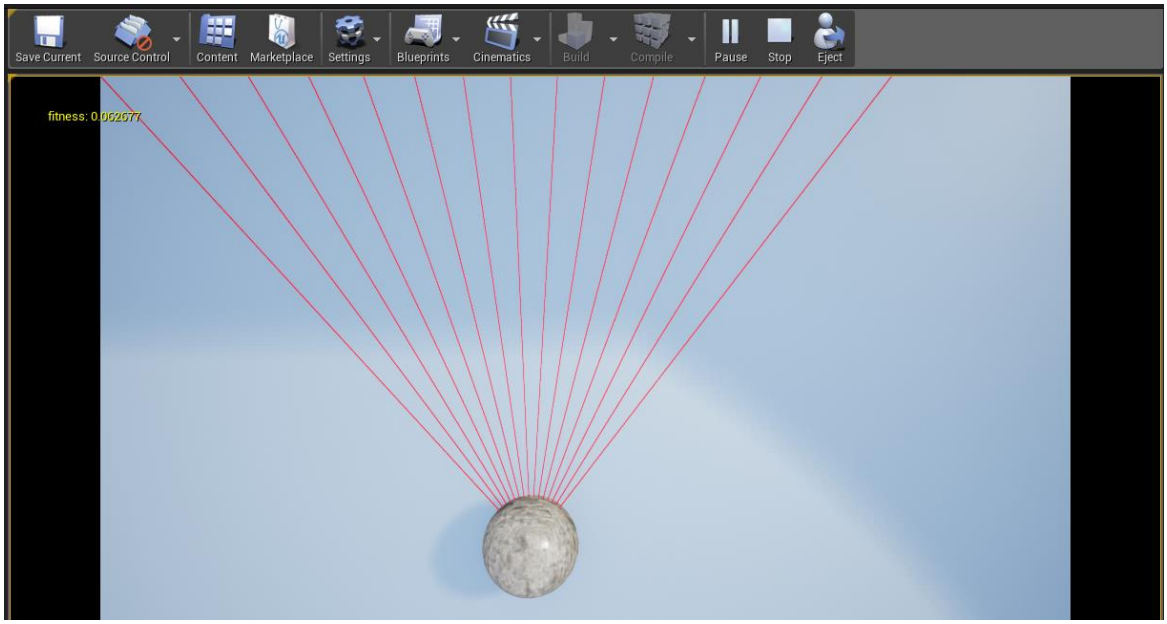


Figure 7 A close view at the angle sensors emitted from the agent

The result of this methodology provides a set of rays at the left and right side of the forward vector following the agent and adapting to its orientation. The code fragment used to calculate the angles of each ray is shown below:

```
FVector actorForwardVector = GetOwner()->GetActorForwardVector();
FVector origin(1.0f, 0.0f, 0.0f);  // The origin

	// the angle formed between the origin and the forward vector
	float originForwardAngle = atan2(actorForwardVector.Y - origin.Y,
actorForwardVector.X - origin.Y);

	// Emit a ray at the center
	EmitSingleRay(originForwardAngle, 0);

	// Emit rays
	for (unsigned int rayCount = 1; rayCount < raysNumber / 2 + 1; rayCount++)
	{
		// increase the offset for the new rays
		currentAngleOffset = offsetAngle * rayCount;

		// Create a ray for each one of the two sides (left and right)
		for (unsigned int i = 0; i < 2; i++)
		{
			if (i == 0) // positive rays
				sensorAngle = currentAngleOffset + originForwardAngle;
			else		// negative rays
				sensorAngle = -currentAngleOffset + originForwardAngle;

			EmitSingleRay(sensorAngle, rayCount * 2 - 1 + i);	//rayCount
```

```
            }
        }
```

The EmitSingleRay method conducts the ray cast and uses the mentioned trigonometric functions to define the vector resulting from the angle passed as a parameter. For debugging purposes and for monitoring the agent's behaviour during training the EmitSingleRay function calls DrawDebugLine to draw the emitted ray.

6.2 Recording training data

The need for storing data gathered during training was identified in earlier sections of the document. These data need to be labelled so that they can be processed and analysed. To meet these expectations the project provides the FileWriter class. This class allows to store, labelled or unlabelled data to a txt file. The data stored to file are represented by an FString. Multiple overloads of the RecordContents method allow to pass different data turned to FStrings and stored to a file. The FString is essentially Unreal's implementation of a string. Its use is ideal as engine modules make use of this type and because it provides methods for converting other types into FStrings.

Data passed for storing are added at the end of the string containing the contents of the file.

```
void FileWriter::RecordContents(FString description, FString value)
{
        fileData = fileData + "\n" + description + ": " + value;
}
```

Requests for storing data are made one each generation. If the training process is interrupted the txt file will contain the data gathered from all the previous generations. The SaveDataToFile function reads the data stored in the file and temporarily stores them to an FString. The data captured by the current generation are added to the recorded contents and passed to the file. To read and write to the file the software took advantage of the FFileHelper class provided by the engine.

The following code fragment is used to execute this process:

```
FString loadedContents;
FFileHelper::LoadFileToString(loadedContents, *directory);

fileData = loadedContents + fileData;

// save recorded data to file
FFileHelper::SaveStringToFile(fileData, *directory);

// clear file data
fileData = FString("");
```

6.2.1 Visualisation of data gathered during training

To visualise training data, the project relied on Python and matplotlib. The Python script used to generate the graphs was written in a Jupiter notebook. To store the data, the software reads through every line of the txt file and stores the labelled data in their appropriate collections. The three collections are:

fitness_items = []

```
best_fitness_items = []

average_attacks = []
```

To identify the data the software conducts a search on the provided label (or desc as mentioned in the script) likewise:

```
if desc == "population fitness":

        fitness_items.append(float(contents))
```

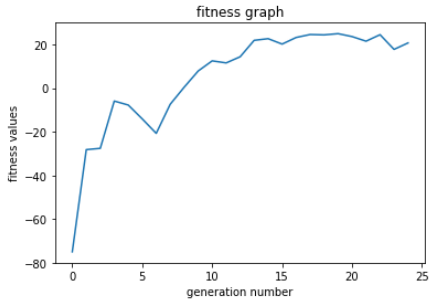
The while loop iterates through each line of the document until it reaches its end. The script stores data only if the ‘:’ character is present on the current line. This character is used to separate the label with the stored value.

```
if ":" in line:

    seperator_index = line.index(':')

    desc = line[0 : seperator_index]


    contents = line[seperator_index + 2 : len(line) - 1]


    if desc == "population fitness":

       fitness_items.append(float(contents))

    elif desc == "best generation fitness":

       best_fitness_items.append(float(contents))

    elif desc == 'avairage attacks' :

       avairage_attacks.append(float(contents))
```

Since data in the file are read as strings the float function is used to convert the read contents into floats which can be graft by the Pyplot module.

The function calls made to the Pyplot library and the resulted functionality is shown below:

```
In [56]: # display graph using matplotlib

         plt.plot(fitness_items)
         plt.title('fitness graph')
         plt.ylabel('fitness values')
         plt.xlabel('generation number')
         plt.show()
```




```
In [57]: # display best fitness graph

         plt.plot(best_fitness_items)
         plt.title('best fitness graph')
         plt.xlabel('generation number')
         plt.ylabel('best fitness values')
         plt.show()
```

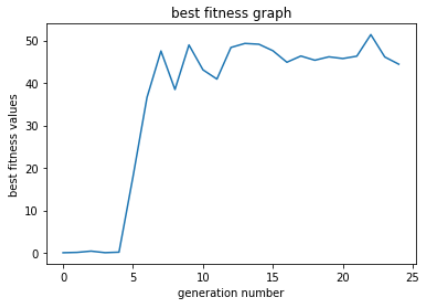


Figure 8 A screenshot of the Jupiter notebook used to graph the data

The data passed to the Pyplot module are a collection of floats. These data represent the y axis values. The developer can pass a second collection as a parameter representing the x-axis points. This is not required as each element is produced by a single generation. The module automatically makes a graph with the values and their indexes in the collection. The title, x label and y label functions are used to provide a description on the data and the graph.

6.3 Agent

The agent is represented by the AEnemyAgent class inheriting from ABaseAgent. The purpose of the agent is to retrieve inputs, provide observations and define a fitness value. The observations and fitness values are passed to the current environment state. The running state use the retrieved data and conduct the appropriate processes to produce an output. During training, observations are passed to the agent's NN and the produced output will be passed as input to the agent. The agent is responsible for translating these inputs into actions influencing its state.

The ABaseAgent class inherits from AActor and implements the Agent template. The functionality provided by this class is contained by the AddObservations, GetInput and SetFitness methods. The agent class does not provide any functionality for agent behaviours. It handles communication with the environment master which receives and provides data produced from the running state. The EnvironmentMaster is a singleton class allowing access to functions for providing and passing data from or to its environment state.

```
double ABaseAgent::GetInput(int index)
{
	if (ABaseEnvironmentMaster::GetInstance() == nullptr)
		return 0.0f;

	return ABaseEnvironmentMaster::GetInstance()->GetInput(agentIndex - 1, index);
}

void ABaseAgent::AddObservation(std::vector<float>& observations)
{
	if (ABaseEnvironmentMaster::GetInstance() == nullptr)
		return;

	ABaseEnvironmentMaster::GetInstance()->AddObservation((agentIndex - 1),
observations);
}
```

Each agent contains a unique agent Index passed as a parameter to the GetInput and AddObservations functions. The ABaseEnvironmentMaster makes calls to the running state to respond to agent requests. The agent index is used to define the collection of inputs and outputs stored for each agent.

The AEnemyAgent is a physics Actor inheriting from ABaseAgent. Collisions with walls and the player will affect the agent's motion. To simulate physics the software attaches a SphereComponent using the engine's editor.

Alternatively, a component can be attached by calling CreateDefaultSubobject in the Actors constructor. The editor will show all the attached components after compilation. The agent also contains a StaticMeshComponent and a UAngleSensor.

```
angleSensor = CreateDefaultSubobject<UAngleSensor>(TEXT("Angle Sensor"));
```

The EnemyAgent class contains functionality for moving the agent forward or backwards, rotate around the vertical axis and attack. The mentioned actions are handled from the following methods: MoveForward, TurnRight and Attack.

TheMoveForward function retrieves a float named amount. The parameter represents the amount of displacement and the direction of movement. The following line is used to move the agents:

```
void AEnemyAgent::MoveForward(float amount)
{
	SetActorLocation(GetActorLocation() + GetActorForwardVector() * amount *
moveSpeed * GetWorld()->DeltaTimeSeconds);
}
```

The same approach is used to rotate the agent:

```
void AEnemyAgent::TurnRight(float amount)
{
	AddActorLocalRotation(FRotator(0.0f, amount * rotSpeed * GetWorld()-
>DeltaTimeSeconds, 0.0f));
}
```

Rotations in Unreal Engine are represented by the FRotator structure. The AddActorLocalRotation applies the rotator passed as a parameter to the object's current rotation.

The UpdateAgent is called on each frame by the environment master. This method applies the data held by the environment state, calculates the fitness value recorded in the current frame and passes the observations and fitness to the operating environment state. The function makes us of the SetFitness, AddObservation and GetInput provided by its parent class.

6.4 Base Environment Master

The environment master inherits from Unreal's Actor and overrides the BeginPlay and Tick methods to initialize and update the agents and state on each consecutive frame. When the program executes, the BaseEnvironmentMaster finds and stores a reference of every enemy agents in the scene. The enemy agent pointers are held by a vector provided by the language's standard library.

To find and iterate through all AEnemyAgent objects in the scene the BeginPlay function uses the actor iterator provided by the EngineUtils header file.

```
for (TActorIterator<AEnemyAgent> ActorItr(GetWorld()); ActorItr; ++ActorItr)

{
	enemyAgents.push_back(*ActorItr);
	ActorItr->BeginAgent();
}
```

The same method initializes the FileWriter used to records the data gathered from training:

```
popAnalytics->RecordContents(FString("Title"), FString("This file contains population
data resulting from training"));
popAnalytics->RecordContents(FString("\n"));

popAnalytics->RecordContents(FString("Population size"), enemyAgents.size());
```

The Tick function iterates through the agents and calls their update method. As new observations are produced on each update, the function clears the previously stored observations by calling the ClearObservations method.

```
// Clear observations
environmentState->ClearObservations();

// Update all agents
for (int i = 0; i < enemyAgents.size(); i++)
{
	enemyAgents.at(i)->UpdateAgent();
}
```

This class provides functions for communicating with the assigned environment state. This includes assigning agent observations and fitness and retrieving inputs translated into actions by the EnemyAgent class.

6.5 Environment States

Environment states retrieve observations and fitness values from all agents in the scene. Each agent contains a unique ID required for passing and retrieving data from the state. The observations and fitness value are associated with the passed ID. Custom made states are expected to inherit from the BaseEnvironmentState class to provide their own functionality for producing agent inputs resulted

from given observations. The custom state associates the agent observations with the given IDs to provide the agents with the correct set of inputs. The custom states can use the functions implemented by their parent for passing and retrieving data from agents.

6.5.1 User controlled state

The UserControlledState inherits from the BaseEnvironmentState class. This class overrides the SetupPlayerInputComponent method provided by the APawn class for the purpose of binding axis and actions. The axis and actions are defined in the system settings of the project. The inputs retrieved from the axis and actions are stored and passed as inputs to the agents.

Binding axis and actions can be achieved with the following code fragments:

```
PlayerInputComponent->BindAxis("MoveForward", this,
&AUserControlledState::PlayerForwardInput);


PlayerInputComponent->BindAction("Attack", IE_Pressed, this,
&AUserControlledState::PlayerAttackPress);
```

6.5.2 NEAT environment state

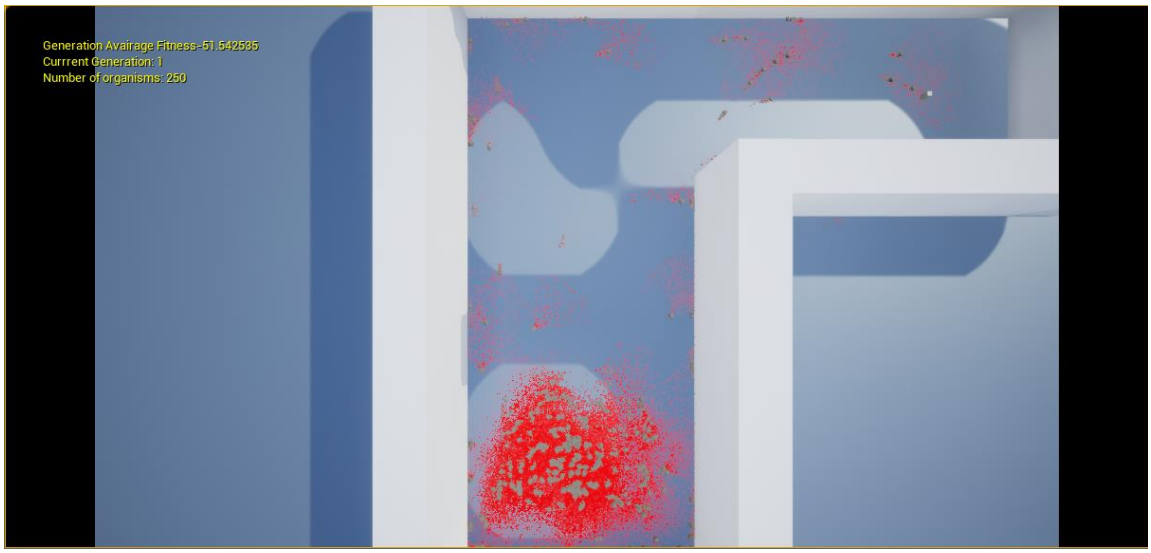


Figure 9 A screenshot of the training environment

The ANeatTrainingEnvironmentState provides all the necessary functionality to support training. The InitState method initializes a genome, the initial population and defines all the required NEAT variables. The library provides a Genome and Population class. A population can be generated with a start genome. The Genome constructor requires an id, the number of inputs, outputs of the NN along with the min and max number of layers included by the NN. The two final parameters determine if the NN is recurrent and the probability of adding a NN link on each generation.

```
genome = new NEAT::Genome(1, observationsSize, inputsSize, 5, 15, false, 0.2);


population = new NEAT::Population(genome, ABaseEnvironmentMaster::GetInstance()-
>GetAgentNumber() - 1);
```

To speed up training the method sets the time dilation to 1000 fast forward the software's execution.

```
GetWorldSettings()->SetTimeDilation(1000.0f);
```

On each frame the function iterates through the population and stores the retrieved observations.

```
for (int orgIndx = 0; orgIndx < population->organisms.size(); orgIndx++)
{
	population->organisms[orgIndx]->net->load_sensors(observations[orgIndx]);
}
```

Once the inputs are fed to the NN, the feed forward algorithm is used by the library to calculate the outputs. The program can retrieve the values of each output node of each organism's NN using the activation attribute. The value retrieved varies between 0 and 1 which suggests that the agents can only move forward and turn right. To change the range to -1 and 1 the software multiplies the value output by two and subtracts by 1.

```
for (int i = 0; i < population->organisms[orgIndx]->net->outputs.size(); i++)
{
```

```
        inputs[orgIndx][i] = population->organisms[orgIndx]->net->outputs[i]-
>activation * 2 - 1;
}
```

The class contains a timer attribute defining when the next generation will occur. The time elapsed from the previous frame is retrieved with the following code segment:

```
generationTimer -= GetWorld()->DeltaTimeSeconds;
```

The generation timer is reduced by the time passed on each frame. When the generationTimer is reduced to or below zero the next population is generated. The fitness attribute of each organism is assigned before the next population is generated. The method calculates and stores the population's average fitness, the highest recorded fitness and the generation number to a txt file. The population class provides the epoch function which makes all the necessary calls for generating the organisms making up the new population.

The functionality provided by the library is relied on the configuration variables. These variables are defined in the InitState function called at the start of execution. The configuration describes the probability of certain actions (like mutations and crossovers) occurring on each agent at each generation.

7.0 Evaluation and Discussion

7.1 Initial tests

The first tests were conducted after the implementation of the environment. The environment would consist of 100 agents and one player object. These tests made use of the NEAT configuration found in the examples provided by the used library.

Observed Behaviours

The configurations used did not encourage the population to mutate and crossover as the variables representing these probabilities are low numbers and changes on the population's behaviour would not be visible. It was observed that during training, the population's behaviour would remain almost identical in future generations. Changes in the NN structures would occur. These changes were not enough to influence the observed behaviours by much.

The behaviour produced in other runs during testing was very simplistic. Most agents would turn in one direction before the generation ends. This is because NNs start with their input, output and a single hidden layer. It was noticed that increasing the starting number of hidden layers would produce agents with more complex behaviours from the start of each training session. The agents produced in the following tests start with 5 hidden layers. As agents produced more complex behaviours from the start, fewer generations were required before the population improved.

Agent observations

- Information provided by the rays (a number representing the distance and another representing the type of object intersecting with each ray).
- Agent position
- Agent rotation
- Player position
- Agent and player distance

- Agent forward vector

NEAT configuration

trait_param_mut_prob	0.5
trait_mutation_power	1.0
linktrait_mut_sig	1.0
nodetrait_mut_sig	0.5
weigh_mut_power	2.5
recur_prob	0.00
disjoint_coeff	1.0
excess_coeff	1.0
mutdiff_coeff	0.4
compat_thresh	3.0
age_significance	1.0
survival_thresh	0.20
mutate_only_prob	0.25
mutate_random_trait_prob	0.1
mutate_link_trait_prob	0.1
mutate_node_trait_prob	0.1
mutate_link_weights_prob	0.2
mutate_toggle_enable_prob	0.00
mutate_gene_reenable_prob	0.000
mutate_add_node_prob	0.03
mutate_add_link_prob	0.05
interspecies_mate_rate	0.001
mate_multipoint_prob	0.3
mate_multipoint_avg_prob	0.4
mate_singlepoint_prob	0.0
mate_only_prob	0.15
recur_only_prob	0.1

dropoff_age	15
newlink_tries	20
babies_stolen	0

Fitness function

The fitness value relied on the distance between the agent and the distance between the agent and the player object. The purpose is to increase reward value as the agent approaches the player object. To meet this goal, the fitness was defined as: $fitness = \frac{1}{x}$ where x is the distance between the two objects.

7.2 Increased mutation probabilities

By considering that the training process did not produce new behaviours, we increased the probabilities affecting mutation types. The test changed the following NEAT variables:

mutate_add_node_prob = 0.33

mutate_add_link_prob = 0.45

mutate_link_trait_prob = 0.2

Observed behaviours

The introduced changes produced more variety in the population. After 5 to 10 generations (varying on each run), agents approaching the agent appeared. It was noticed that as training continued, agents would lose their successful behaviours. This phenomenon appeared to repeat periodically. After a certain amount of generations, the population showed what appeared to be random behaviour.

Agents managing to approach the agent in previous generations would wander around the map. As training continued, agents finding the optimal behaviour re-appeared. These agents displayed different behaviours from the ones discover in previous generations. The problem produced, stops the agents from improving their existing behaviour.

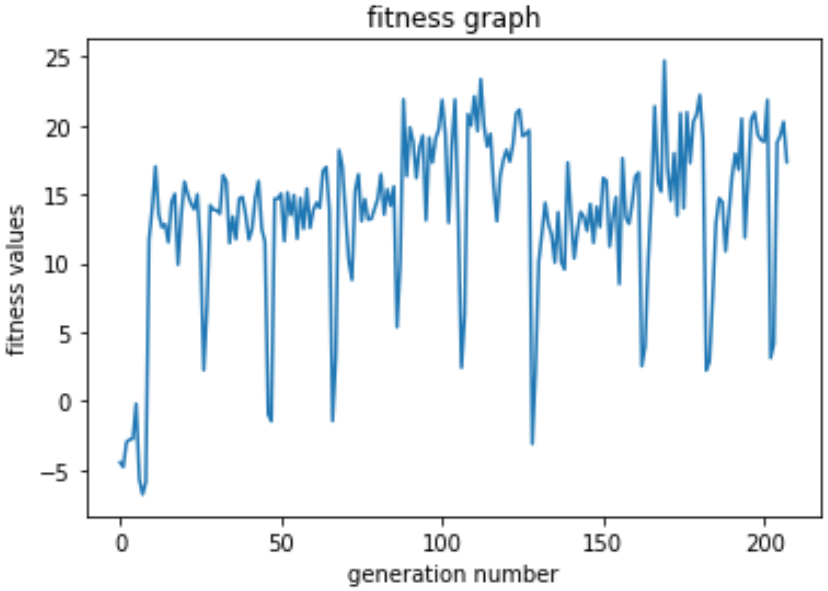


Figure 10 Average population fitness vs generations number graph. The data produced were taken from increasing the probability for mutations.

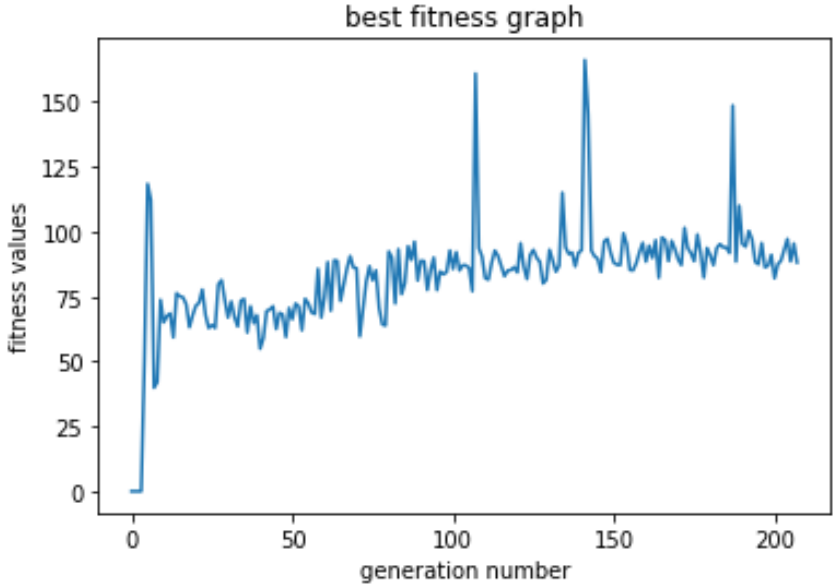


Figure 11 Best fitness vs generations number graph. The data produced were taken from increasing the probability for mutations.

By combining the results of the two graphs we can see that while a varying population average fitness could suggest that the NEAT algorithm is properly used to explore new behaviours, the best fitness graph clearly suggests that agents with optimal behaviours seem to disappear in next generations.

The agents were also tested in the other environments mentioned in the design section. Data originating from the agent's surroundings were not fed to the NN. As a result, agents colliding with walls influence the population.

Agents did not show the ability to find the fastest path to player object. In some runs the agents would circularly wander around the map before nearing the player object. An instability of behaviours was present as each run would produce different results.

It is difficult to suggest why these problems are present because we are not aware of the operations produced by the library. The algorithm is treated as a black box, but certain assumptions can be made. The variation of behaviours between each run could be caused from mutations occurring on each generation. The populations produced, discover different NN structures to produce agents approaching the player. Loosing optimal agents could be caused from low probabilities controlling crossover. As a result, highly performing agents are not likely to pass their genes to the rest of the population. Mutations occurring on these agents result in "losing" the agent's behaviour. The drop of age resets half of the population after several generations. It is possible that agents producing a good behaviour, were being reset and as a result the population lost its achieved progress.

7.3 Observing the environment

For this test, the data gathered from the ray angles were passed to the NN as inputs along with the observations described in the previous section. The ray angles feed two types of data: the distance with the object hitting the ray and a number used to identify the type of the object.

Fitness function

The fitness function is modified to give a reward for each ray recording a short distance between the agent and the player. Accordingly, rays recording short distances with walls produce a punishment. To ensure that these events influence the population on each generation, the fitness value is being reset at the end of each generation. The fitness function adds a reward every time a successful attack is conducted. The distance between the player and the agent is influencing the fitness value as agents reaching close to the agent without retrieving a reward originated from the rays could produce a dominant behaviour.

$$total\ fitness = \frac{1}{player\ distance} + attack\ count * attack\ reward + rayHitPlayerCount * rayHitPlayer\ Reward - rayHitWallCount * rayHitReward$$

Agent Observations

trait_param_mut_prob	0.5
trait_mutation_power	1.5
linktrait_mut_sig	1.0
nodetrait_mut_sig	0.5
weight_mut_power	3.5
recur_prob	0.1
disjoint_coeff	1.0

excess_coeff	1.0
mutdiff_coeff	0.4
compat_threshold	3.0
age_significance	1.0
survival_thresh	0.15
mutate_only_prob	0.2
mutate_random_trait_prob	0.8
mutate_link_trait_prob	0.1
mutate_node_trait_prob	0.1
mutate_link_weights_prob	0.1
mutate_toggle_enable_prob	0.15
mutate_gene_reenable_prob	0.15
mutate_add_node_prob	0.23
mutate_add_link_prob	0.25
interspecies_mate_rate	0.4
mate_multipoint_prob	0.4
mate_multipoint_avg_prob	0.4
mate_singlepoint_prob	0.3
mate_only_prob	0.1
recur_only_prob	0.2
dropoff_age	10
newlink_tries	20

Simple scene data

First Run

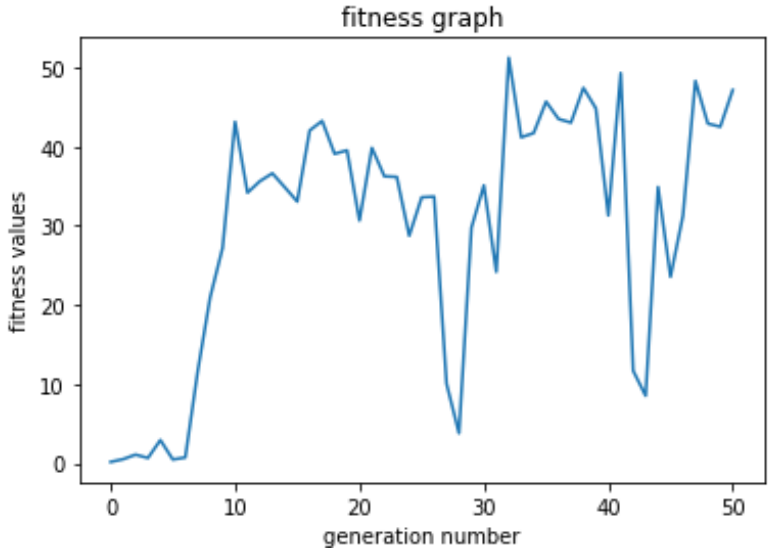


Figure 12 Average population fitness vs generations number graph. The data were produced from the 1st run. The environment contained the agents and player object.

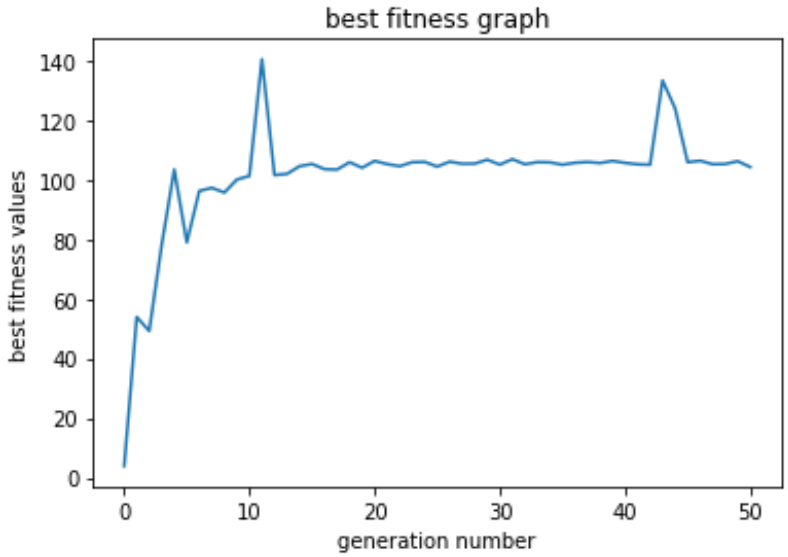


Figure 13 Highest agent fitness vs generations number graph. The data were produced from the 1st run. The environment contained the agents and player object.

Second Run

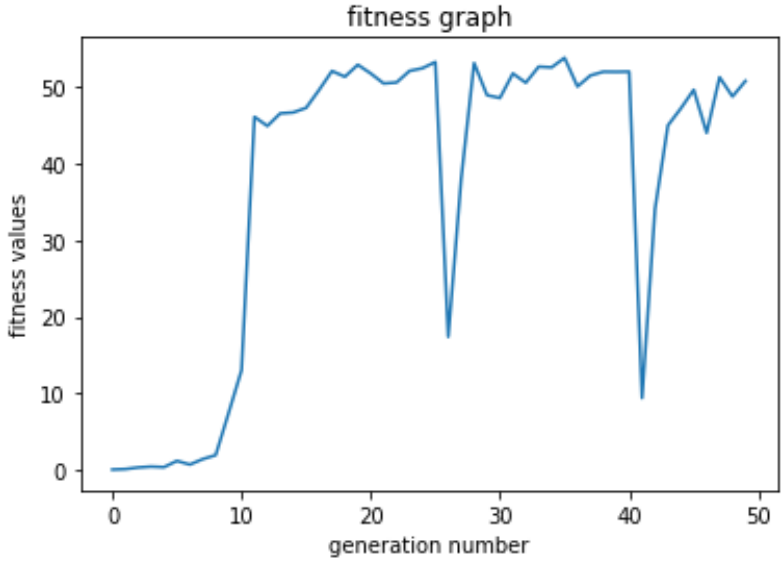


Figure 14 Average population fitness vs generations number graph. The data were produced from the 2nd run. The environment contained the agents and player object.

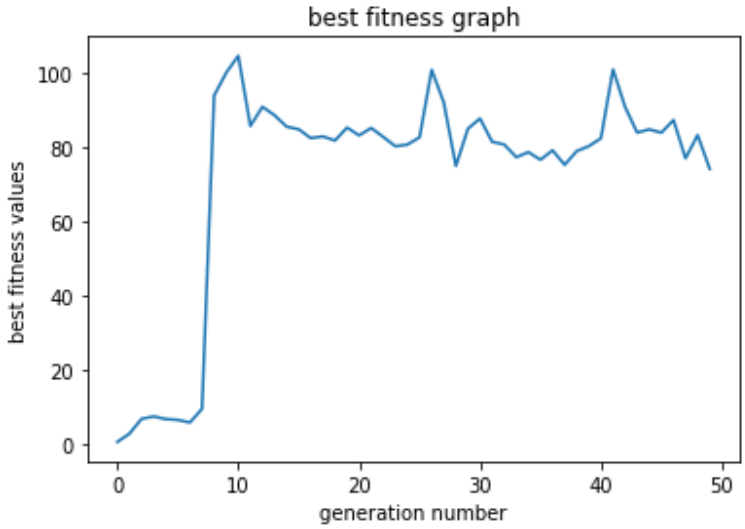


Figure 15 Highest agent fitness vs generations number graph. The data were produced from the 2nd run. The environment contained the agents and player object.

Third Run

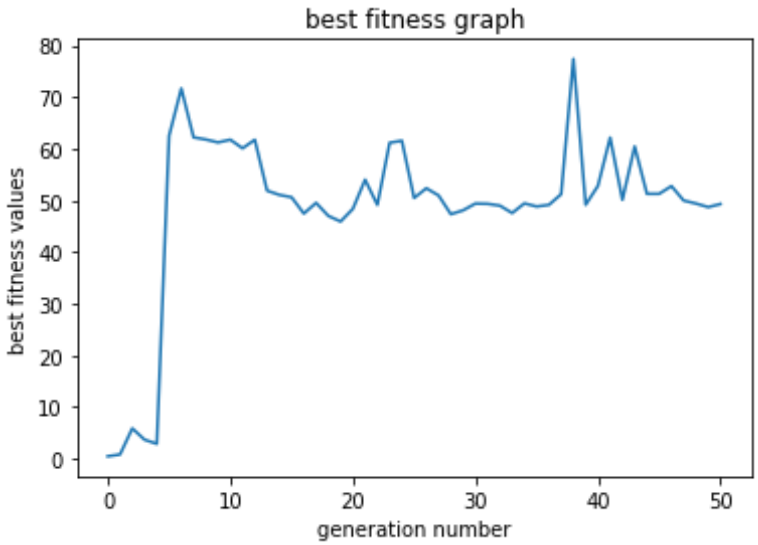


Figure 16 Average population fitness vs generations number graph. The data were produced from the third run in the simple environment.

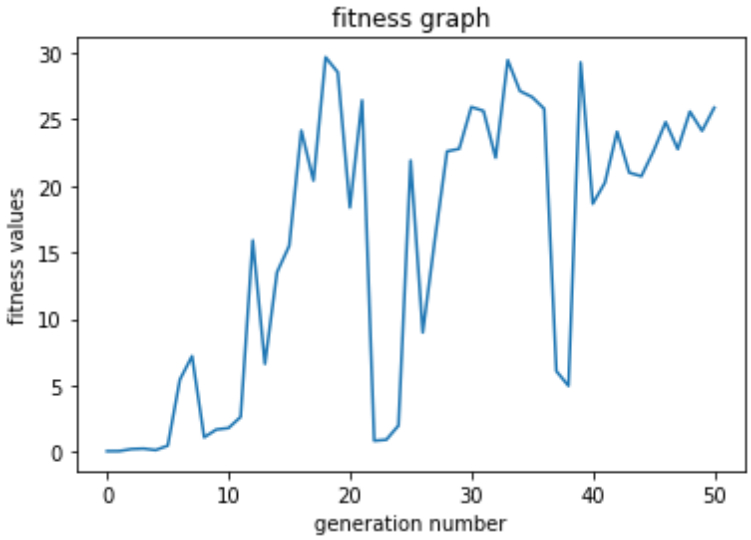


Figure 17 Highest agent fitness vs generations number graph. The data were produced from the 3rd run. The environment contained the agents and player object.

Observed behaviours and data analysis

All the tests produced agents who could approach and orientate towards the player object. The agents did not show the ability to find the best possible path leading to the player objet.

This test shows that the population originated from each run requires more generations before produsing improved agents. During the second run, the population showed almost no improvement before the 40th generation. Acrodingly, the third test produced a population with clear signs of improvement after the 20th generation. The behaviour also varied as some runs produced agents with higher fitness values.

Walls surrounding the player and agents

First Run

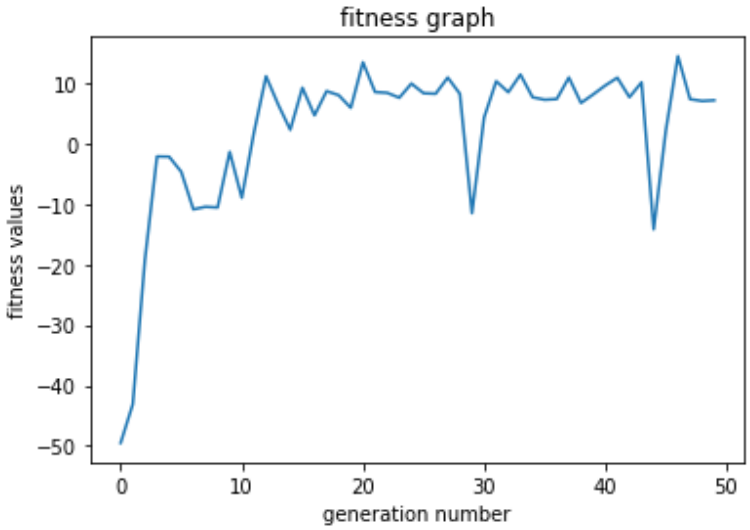


Figure 18 Average population fitness vs generations number graph. The data were produced from the 1st run. The training environment contained the agents, the player object and walls surrounding them.

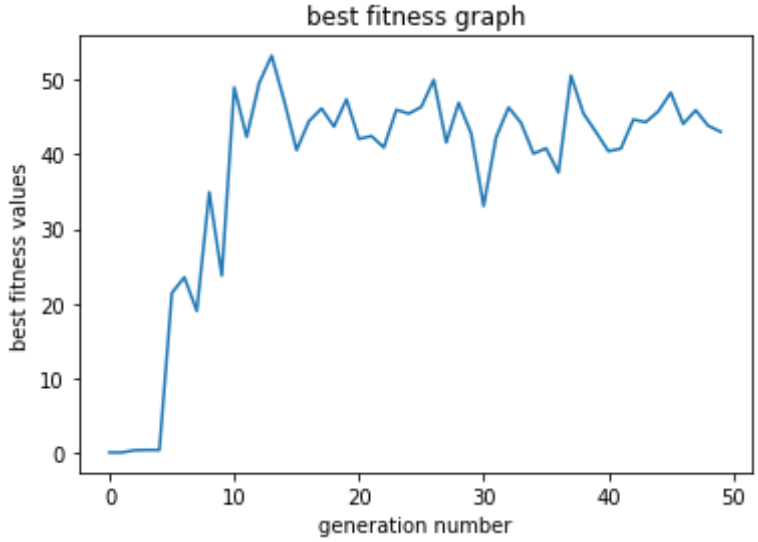


Figure 19Highest agent fitness vs generations number graph. The data were produced from the 1st run. The training environment contained the agents, the player object and walls surrounding them.

Second Run

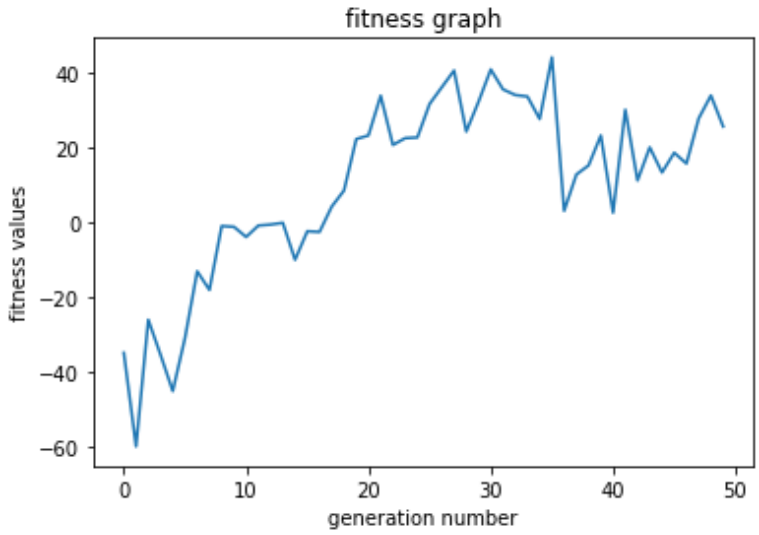


Figure Average population fitness vs generations number graph. The data were produced from the 2nd run. The training environment contained the agents, the player object and walls surrounding them.

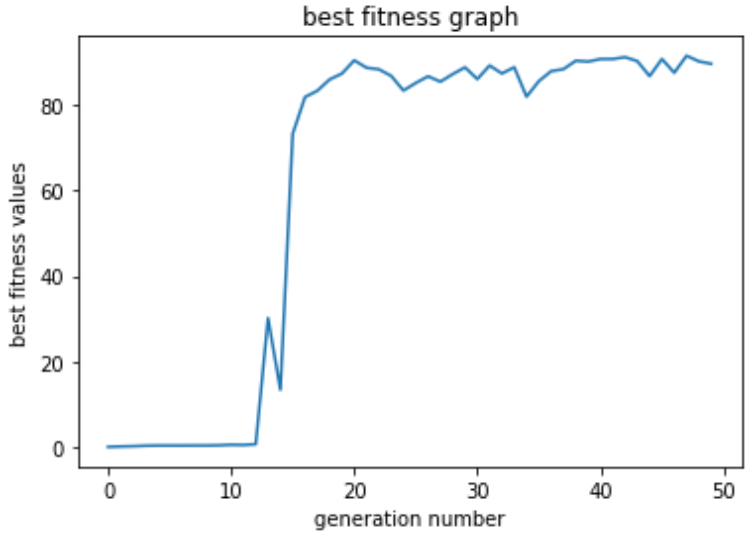


Figure 20 Highest fitness value vs generations number graph. The data were produced from the 2rd run. The training environment contained the agents, the player object and walls surrounding them.

Third Run

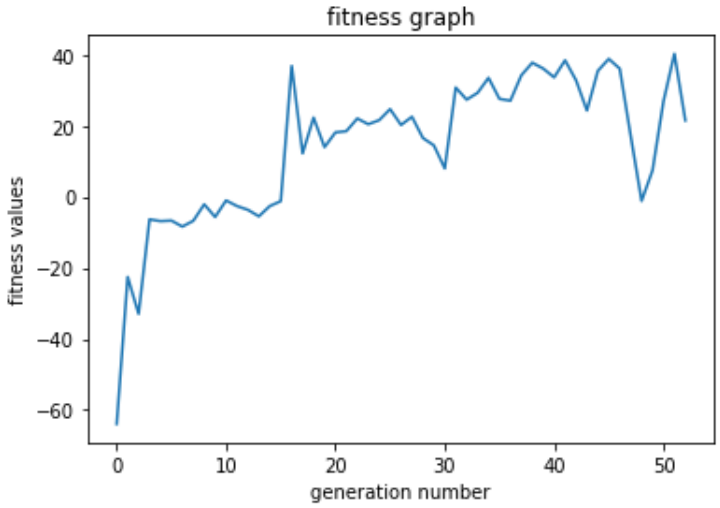


Figure 21 Average population fitness vs generations number graph. The data were produced from the 3rd run. The training environment contained the agents, the player object and walls surrounding them.

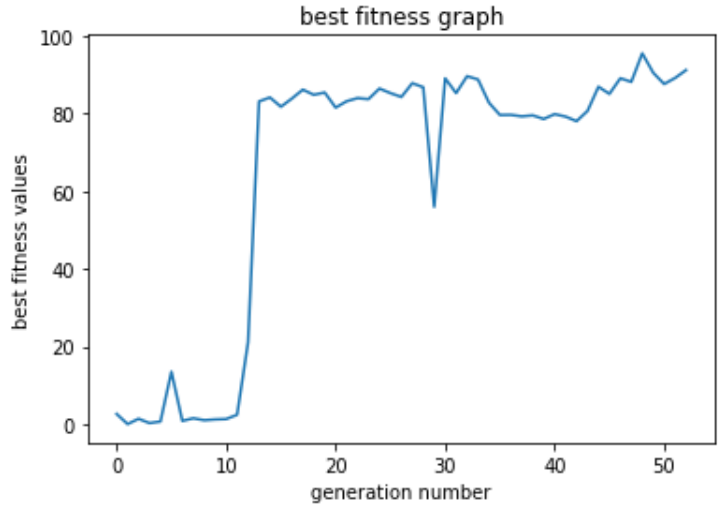


Figure 22 Highest fitness value vs generations number graph. The data were produced from the 3rd run. The training environment contained the agents, the player object and walls surrounding them.

Environment with walls blocking the agent's path

First Run

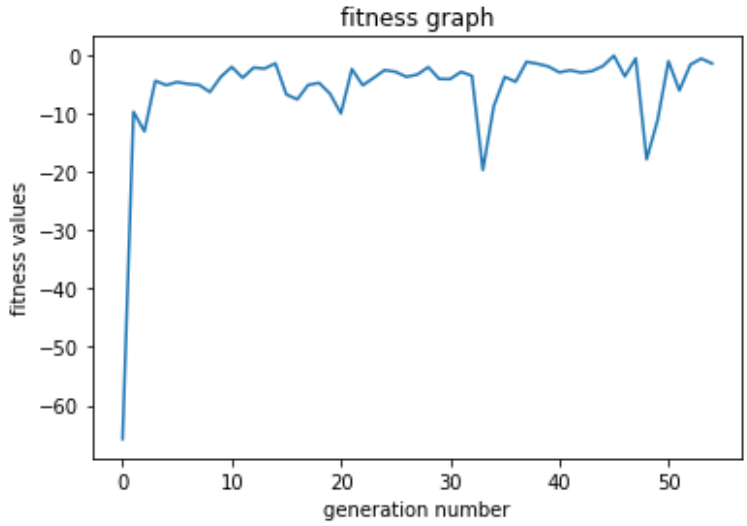


Figure 23 Average population fitness vs generations number graph. Average population fitness vs generations number graph. The data were produced from the 1st run. The training environment contained the agents, the player object and walls surrounding them. Additional walls block the path leading to the player object.

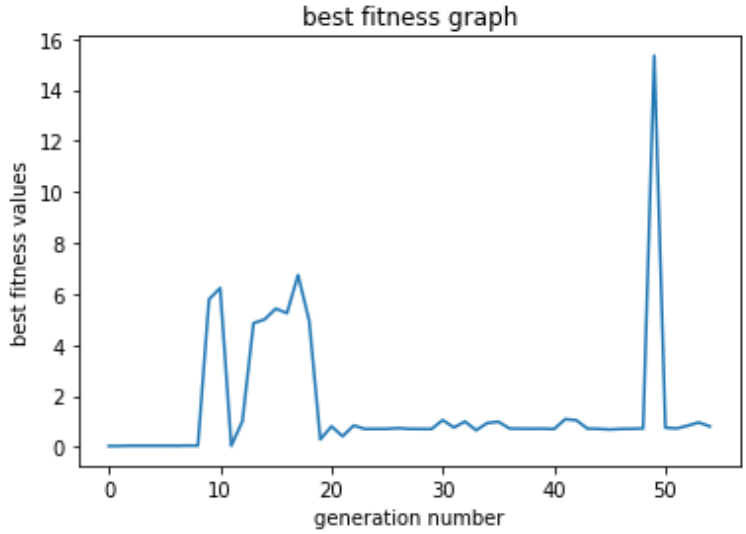


Figure 24 Highest fitness value vs generations number graph. The data were produced from the 1st run. The training environment contained the agents, the player object and walls surrounding them. Additional walls block the path leading to the player object.

Second Run

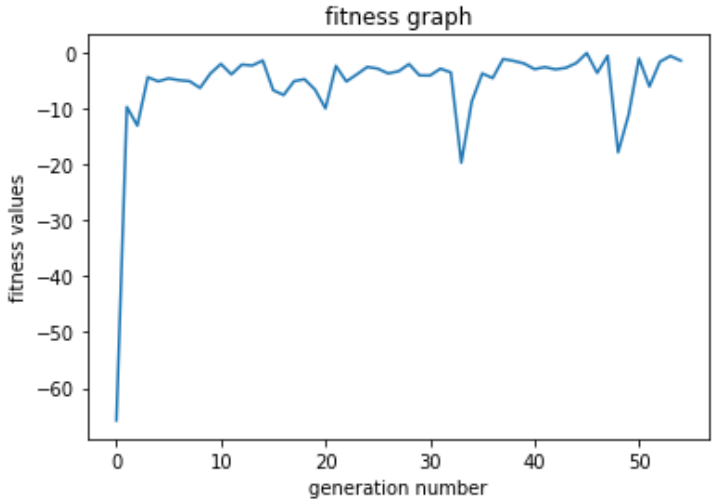


Figure 25 Average population fitness vs generations number graph. The data were produced from the 2nd run. The training environment contained the agents, the player object and walls surrounding them. Additional walls block the path leading to the player object.

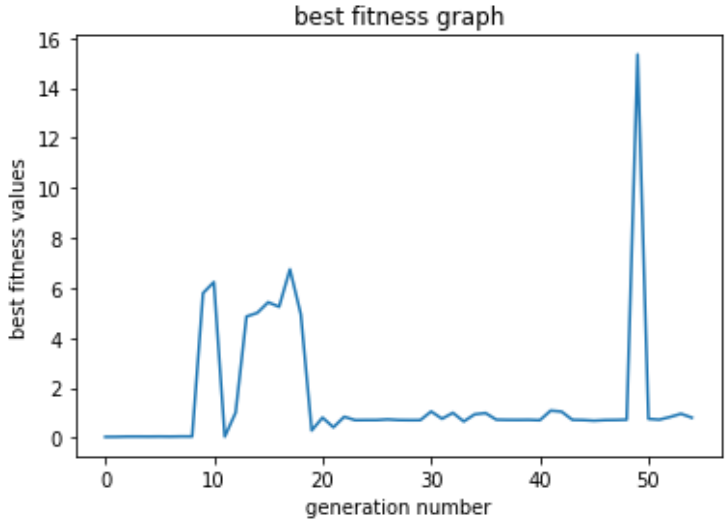


Figure 26 Highest agent fitness vs generations number graph. The data were produced from the 2nd run. The training environment contained the agents, the player object and walls surrounding them. Additional walls block the path leading to the player object.

Third Run

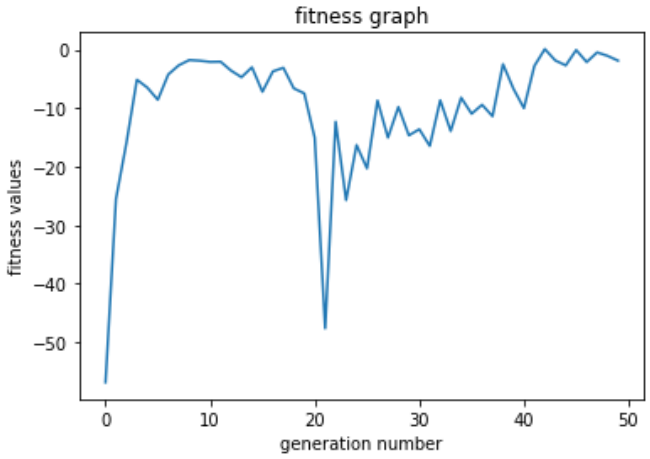


Figure 27 Average population fitness vs generations number graph. The data were produced from the 3rd run. The training environment contained the agents, the player object and walls surrounding them. Additional walls block the path leading to the player object.

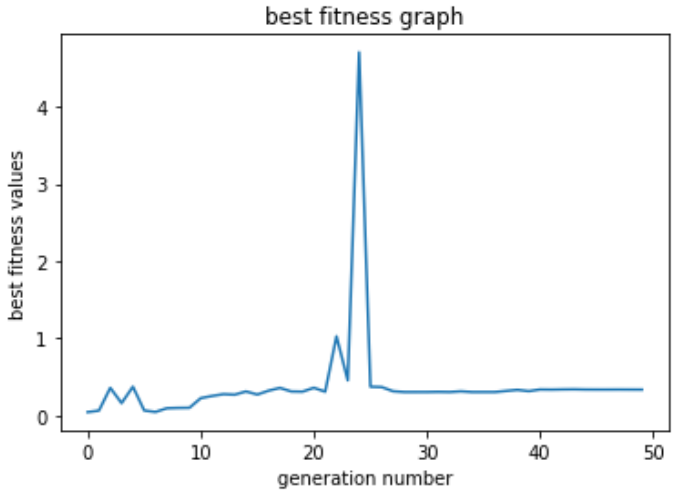


Figure 28 Highest agent fitness vs generations number graph. The data were produced from the 4th run. The training environment contained the agents, the player object and walls surrounding them. Additional walls block the path leading to the player object.

Fourth Run

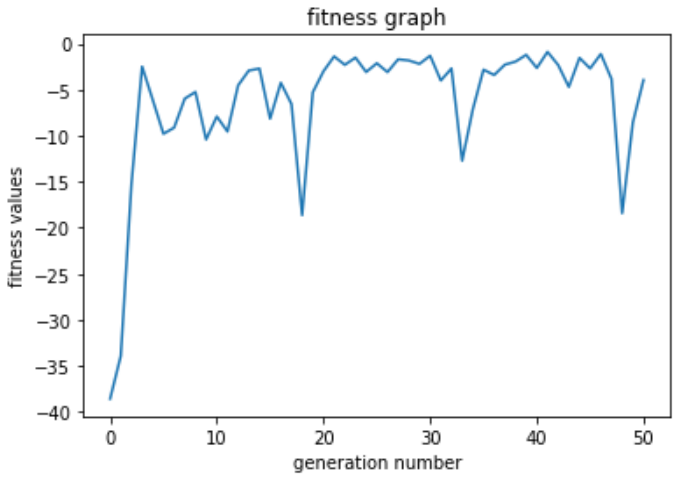


Figure 29 Average population fitness vs generations number graph. The data were produced from the 4th run. The training environment contained the agents, the player object and walls surrounding them. Additional walls block the path leading to the player object.

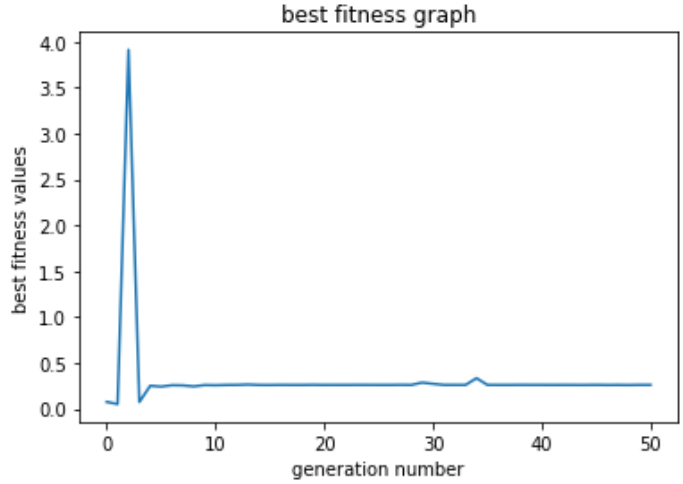


Figure 30 Highest agent fitness vs generations number graph. The data were produced from the 4th run. The training environment contained the agents, the player object and walls surrounding them. Additional walls block the path leading to the player object.

Observed behaviours and data analysis

The agents managed to increase their fitness as more generations occurred. During training, the population showed clear signs of improvement as agents showed the ability to approach the player object. While clear signs of improvement were shown, the agents could not find the most optimal behaviour maximising their fitness. The population seemed to settle on a behaviour after a certain number of generations. As most agents followed that behaviour, a small fraction of the population would show slight or rapid differences in that behaviour. It was also noticed that if new agents achieved a greater fitness value, several agents would be influenced and follow the new uncovered behaviour. The number of agents being influenced by the new behaviour increased after multiple generations. Agents following the previous behaviour were not extinct. The two behaviours compete within the population resulting in improved agents. Mutations in agents following these behaviours

can result in identifying the optimal path leading to the player or encourage the agents to conduct successful attacks.

Rapid changes in the population are most likely to occur after the current generation reached the set drop of age. The drop of age is a variable set before runtime and represents the number of generations required before half of the population “dies”. In this test, the drop of age was set to 10. The agents being affected by this event showed random behaviour and were most likely to be affected by other organisms(agents) in the population.

The test showed that best performing agents would not survive in upcoming generations. This assumption is made by both research methods. It was noticed that agents outperforming the rest of the population, did not show any signs of influence in next generations. This is also being suggested by the graphs, as the best fitness values were rapidly reduced. The best population fitness vs generations graph produced in the final test, shows that the best fitness recorded in the first generations exceeded a value of 4. A rapid drop in the best generation fitness is displayed in the consecutive generation since the best agent recorded did not exceed a fitness value of 0.3. The change between a single generation is massive as the best fitness on this generation is reduced by around 92.5%.

NEAT relies heavily on the configurations set before runtime. The phenomenon of losing “good” agents can be resulted from reaching the drop of age as half of the population is being reset. This is not likely as the data gathered show that the generation number where the optimal agent was lost do not match the drop of age. A more likely explanation is that crossover did not occur between the agent and other members of the population. In the next population the agent would instead undergo mutations resulting in losing the displayed behaviour. This suggestion is only assumed as all NEAT variables influence the population in a random manner. Other causes could lead to this result and thus a conclusion cannot be made.

The second environment where walls would surround the agents and player objects showed that some runs produced more successful populations than others. This was initially identified through observations since agents produced in the third run could find the fastest path leading to the player. This behaviour was not reached by the populations produced in other runs. The quantitative data provide additional evidence on this observation as the best achieved agent fitness of the third run exceeded the fitness values produced in other runs. Due to random changes occurring on each agent, the NN structures produced in each run differs. The NN structure influences the behaviour followed by the agents. It is possible that the first and second test could produce better agents through extensive training.

Overall, the training process produced agents reaching the player object. Most commonly, agents could not find the best path leading to the agent. Another frequently occurring behaviour, had the agents approaching the player without sustaining the position minimising its distance with the player. The agents would most likely approach the player, wander around the training area and approach the player once again just before the current generation ends.

7.4 Adapting to the agent’s position

Using the observations and fitness variables described in the previous section of the paper, we conducted a test to observe if the agent can adapt to multiple player positions. This test was conducted in the basic environment where the only objects present are the agents and the player object. On each generation the player spawns in a random location.

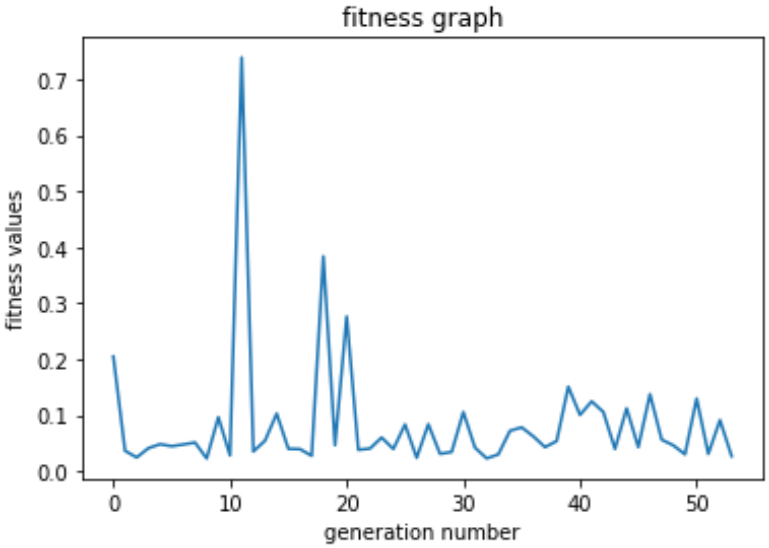


Figure 31 Average population fitness vs generations number graph. The data were produced from the 1st run. The training environment contained the agents, the player object. On each generation the player is spawned to a random location.

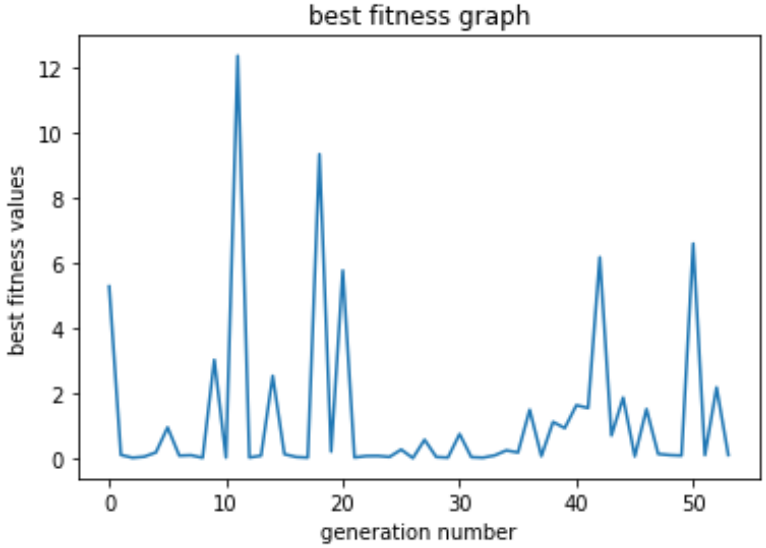


Figure 32 Highest agent fitness vs generations number graph. The data were produced from the 1st run. The training environment contained the agents, the player object. On each generation the player is spawned to a random location.

Fitness levels were scattered on each test. Rapid changes in the average population's fitness can be observed as different results were given on previous and next generations. The same can be said about the best fitness values recorded on each generation. Through observing the agents during training, it was noticed that high fitness values were not primarily dependent on agent behaviours. Instead high fitness values were given when the player's position was relatively close to the agent's spawn location. It was commonly noticed that agents appearing to move in a random manner would influence the population as more agents followed this behaviour, even if the player object were not near them. As a result, agents chosen to influence the next generation were not ideal.

8.0 Evaluation Conclusion

8.1 Project critique

8.1.1 Evaluation of results

The agents showed the ability to meet some of the set criteria while producing a range of limitations. When it comes to adaptivity to a given environment, the agents evolved to avoid observed wall objects while orientating and moving towards the player. This behaviour was present in every conducted test and suggests that the method of using ray angles for observing surroundings originated from the environment was ideal for the scope of this project.

The produced limitations concern the behaviour and evolutionary process. Multiple runs in the same training environment can produce changes in their behaviour as well as the number of generations required before the population shows clear signs of improvement. Some runs would improve the population and discover ideal behaviours faster than others. Since agents undergo random mutations, the produced behaviour varies on each run. This produces a certain level of uncertainty as we cannot know what behaviours will arise from running the application.

One of the most significant problems involved losing agents outperforming the rest of the population. The data gathered from graphs suggests that these agents achieved a high fitness value. These agents would not influence the population and their behaviour would disappear in the following generations. The most likely explanation would suggest that the highly performing agent would not mate with the rest of the population but instead mutate which could explain why the displayed behaviour was not present in future generations. Lowering the probabilities responsible for mutating the agents can provide a solution to this problem.

The population did not manage to adapt to the new agent's location as agents did not show any signs of improvement. The main causes are the NEAT configuration and environment. The population's performance relied heavily on the random location of the player object. Agents wandering near their spawn location could influence the population if the player spawned near them. The NEAT configuration encourages mating which produces agents who blindly follow the discovered behaviour which is not relied on the player's position. A beneficial improvement could involve spawning the agents at random positions so that more variant data can be fed to the NNs.

8.1.2 Project problems and improvements

Multiple problems disclosed during the research and development of this project. Analysing these problems can help identify and justify the suggested further work in the next section of this research paper. One of the main factors slowing down the progress of the research is the authors inexperience in both the engine and machine learning technique. Previous experience with the engine would be ideal as a reasonable amount of time was required before the training environment was built. It would be more ideal if a more familiar tool were chosen since the time spent building the environment could be used to conduct other tests to meet the criteria set in this research. In addition, the author had no previous experience with the used machine learning technique. Neuroevolution involves concepts taken from NNs and GAs which were proven to be overwhelming especially at the start of the project. A reasonable amount of time was also required before the knowledge gained could be used in the implementation of the final project.

The approach of neuroevolution is well established and has found use in many projects. Under the context of this project and the lack of experience possessed by the author, this approach is arguably

unsuitable. The random nature of the algorithm produced different results in the same environments on separate runs. Therefore, conclusions on the produced behaviours and possible improvements could not be easily identified. Another consideration is configuring the variables used by the algorithm. Changing the probabilities for events like mating and mutating can produce significant changes in the produced agents and as a result a great amount of time was required before a suitable configuration was identified. This is because the algorithm was treated as a black box. Improvements in the configuration were based on observed behaviours, which made the process of identifying limitations difficult as the observer is not aware of changes occurring on each NN. Solving the problem of losing well performing agents could rely on experimenting further with the NEAT configuration. Different ML approaches relying on fixed structured NN could remove a level of complexity and produce better agents.

8.2 Further work

The environment produced in this project is stable and the game engine used provides a variety of tools to support development. Using states to describe the functionality followed by the environment can support the inclusion of other ML techniques. It was discussed in the previous section that approaches with a fixed structure NN could help reduce some instabilities and remove a layer of complexity as the developer will not have to conduct multiple tests to identify a suitable NEAT configuration. With that being said, the final deliverable met some of the set criteria and so future work can involve the continuation of this research using NEAT on the same or different environment. Additional tests can be easily conducted on the produced project and so additional research aiming at improving the training environment, fitness method, agent observations and NEAT configuration could lead to producing agents with the desired behaviour. While the current data do not suggest that the approach can produce the desired agents, additional research can potentially prove the opposite.

Fitness values and observations cannot produce the desired agents without being affected by the NEAT configuration and thus Increasing the difficulty involved with creating the agents. Fixed structured NN could produce the desired agent through using or altering the project's fitness function and observations. If this is achieved, NEAT can be used to identify the most minimal NN structure for producing the desired behaviour. A minimal structure suggests a NN with the lowest number of links and nodes providing the desired behaviour. This results in improved performance as data travel through less NN nodes and links.

8.3 Conclusions

The research set several desired behaviours followed by traditional enemy AIs in RPGs and attempted to reproduce them using neuroevolution. The desired behaviour involves agents chasing and attacking the player in the most effective way while taking into account objects surrounding it. While some of the underlined behaviour is present in the final deliverable, the data gathered from the conducted tests do not suggest that NEAT is suitable for the tasks at hand.

The agents showed the ability to move and orientate towards the agent. Walls produced by the training environment were avoided by the agent suggesting that the use of angle sensors was suitable for producing agents aware of their surroundings. The agents showed a reasonable amount of limitations. In terms of behaviour, the agents could not reach the player object in the fastest way possible while successful attacks were not conducted to the player object in any of the conducted tests. Issues during training were also present. Agents with optimal behaviour would not influence

the population and disappear in consecutive generations. In addition, agent behaviours varied on each run conducted in the same training environment.

A reasonable amount of time taken in building the environment as the author did not have any previous experience with the engine. Neuroevolution makes use of many variables controlling the occurrence of crossovers and mutations. Further experimentation with the given configuration could improve the given results but this process was proven to be time consuming. Overall, some of the desired results were achieved but other ML techniques with a fixed NN topology could produce better agents.

9.0 References

1. Heidenreich, H. (2018). What are the types of machine learning?. Last accessed: 13th Oct. 2019. Available: https://towardsdatascience.com/what-are-the-types-of-machine-learning-e2b9e5d1756f
2. Géron, A. (2017) *Hands-on machine learning with Scikit-Learn and TensorFlow*.
3. Brownlee, J. (2016). Supervised and Unsupervised Machine Learning Algorithms. Last accessed: 20th Oct. 2019. Available: https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/
4. Stephenson, N., & Crash, S. (2019). Deep Learning. Last accessed: 20th Oct. 2019. Available: https://learning.oreilly.com/library/view/deep-learning/9781491924570/ch01.html#review_of_ml
5. Patterson J, & Gibson A. (2017) *Deep Learning*. 1st edition. O'Reilly Media, Inc., 2017. Print.
6. Karlik, B. & Vehbi Olgac, A. (2019). Performance Analysis of Various Activation Functions in Generalized MLP Architectures of Neural Networks. Available: http://www.cscjournals.org/manuscript/Journals/IJAE/Volume1/Issue4/IJAE-26.pdf
7. Shiffman, D., Fry, S., & Marsh, Z. *The nature of code* (1st ed.).
8. Samothrakis S. & Perez-Liebana D. & Simon M. Lucas & Fasli M (2015) Neuroevolution for General Video Game Playing.
9. Einar H. & Andreas Høgetveit W. (2019) Creatively Evolving Cooperative Behaviour with the 'NeuroEvolution of Augmenting Topologies' Algorithm. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2562096/17993_FULLTEXT.pdf?sequence=1&isAllowed=y
10. Jonathan Hastings E. & Ratan K. Guha & Kenneth O. Stanley (2009) Automatic Content Generation in the Galactic Arms Race Video Game. IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 1, NO. 4, DECEMBER 2009. Accessed: 1st Nov. 2019
11. Vurucu, M. (2019). How do we teach a machine to program itself ? — NEAT learning. Accessed: 1st Nov. 2019. Available from: https://towardsdatascience.com/how-do-we-teach-a-machine-to-program-itself-neat-learning-bb40c53a8aa6
12. isi, S., & Togelius, J. (2015). Neuroevolution in Games: State of the Art and Open Challenges Available from: https://arxiv.org/pdf/1410.7326.pdf
13. Bourg, D., & Seemann, G. (2004). *AI for game developers* (pp. 356-461). Beijing: O'Reilly.
14. Price W. & Schrum J. (2019). Neuroevolution of Multimodal Ms. Pac-Man Controllers Under Partially Observable Conditions Accessed 3thNov. 2019
15. Luo, J. (2019). An Exploration of Neural Networks Playing Video Games. Available from: https://towardsdatascience.com/an-exploration-of-neural-networks-playing-video-games-3910dcee8e4a Last Accessed: 21th Oct 2019
16. Stanley, K. (2019). Neuroevolution: A different kind of deep learning. Available from: https://www.oreilly.com/radar/neuroevolution-a-different-kind-of-deep-learning/
17. Delgado, C. (2019). Top 15: Best open source javascript game engines. Last Accessed: 23th Oct 2019. Available from: https://ourcodeworld.com/articles/read/308/top-15-best-open-source-javascript-game-engines

18. Python, R. (2019). PyGame: A Primer on Game Programming in Python – Real Python. Available From https://realpython.com/pygame-a-primer/ Last Accessed: 9th Nov 2019
19. Hausknecht, M., Lehman, J., Miikkulainen, R., & Stone, P. (2014). A Neuroevolution Approach to General Atari Game Playing. IEEE Transactions On Computational Intelligence And AI In Games, 6(4), 355-366. doi: 10.1109/tciaig.2013.2294713. Available from: https://www.cs.utexas.edu/~mhauskn/papers/atari.pdf
20. Miikkulainen, R., D. Bryant, B., Cornelius, R., V. Karpov, I., O. Stanley, K., & Han Yong, C. (2016). Computational Intelligence in Games Available From: http://nn.cs.utexas.edu/downloads/papers/miikkulainen.wcci06.pdf
21. Jordan, J. (2019). Convolutional neural networks. Available: https://www.jeremyjordan.me/convolutional-neural-networks/ .
22. Hansen, C. (2019). Neural Networks: Feedforward and Backpropagation Explained. Available from: https://mlfromscratch.com/neural-networks-explained/#/
23. Upadhyay, Y. (2019). Feed forward Neural Networks. Available from: https://towardsdatascience.com/feed-forward-neural-networks-c503faa46620
24. Heidenreich H. (2019) NEAT: An Awesome Approach to NeuroEvolution. Last accessed 10th Nov. 2019, from https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f
25. (2019). Physics Simulation. Unreal Engine Available from: https://docs.unrealengine.com/en-US/Engine/Physics/index.html
26. Bahceci, E., D'Silva, T., Karpov, I., & Stanley, K. (2019). NNRG Software - NEAT C++. Last accessed: 15th Oct 2019. Available from: http://nn.cs.utexas.edu/keyword?neat-c
27. https://www.sciencedirect.com/science/article/pii/S1364815296000308
28. Chatterjee S.& Laudato M. & A.Lynch L (1996). Computational Statistics & Data Analysis, 633-651.
29. Sauro, J., 2020. 4 Types Of Observational Research. [online] Measuringu.com. Available at: https://measuringu.com/observation-role/ Last accessed 19th Apr. 2020
30. Docs.unrealengine.com. 2020. *Unreal Engine 4 Documentation*. [online] Available at: https://docs.unrealengine.com/en-US/index.html Last accessed 19th Apr. 2020.
31. Lindsay, G., 2020. Recurrent Connections Improve Neural Network Models Of Vision. [online] Simons Foundation. Available at: https://www.simonsfoundation.org/2019/05/23/recurrent-connections-improve-neural-network-models-of-vision/ Accessed at: 19 April 2020
32. OpenAI. 2019. Emergent Tool Use From Multi-Agent Interaction. [online] Available at: https://openai.com/blog/emergent-tool-use/ Accessed 19 April 2020
33. Hunter H., (2019) NEAT: An Awesome Approach to NeuroEvolution [online photograph]. Last accessed: 10th Nov 2019. Available from: https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f
34. Choudery H., (2018) What are neural networks? (A non-mathematical explanation) [online photograph]. Last accessed: 10th Nov 2019. Available from: http://aiforanyone.org/all/blog/what-are-neural-networks/
35. Haas, J. A History of the Unity Game Engine. [online] Web.wpi.edu. Available at: https://web.wpi.edu/Pubs/E-project/Available/E-project-030614-143124/unrestricted/Haas_IQP_Final.pdf Last accessed: 21 April 2020.
36. Dutta, A., n.d. Crossover In Genetic Algorithm - Geeksforgeeks. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/crossover-in-genetic-algorithm/ Last accessed: 21 Apr. 2020.
37. Sumit G., (2018) Deep learning performance breakthrough [online photograph]. Available from: https://www.ibm.com/blogs/systems/deep-learning-performance-breakthrough/ Last accessed: 29th Oct 2019

38. Stanley, K.O., Clune, J., Lehman, J. Miikkulainen R. (2019) Designing neural networks through neuroevolution. Nat Mach Intell 1, 24–35. Available from: https://doi.org/10.1038/s42256-018-0006-z Last accessed: 25 Apr. 2020.

10.0 Bibliography

1. (2017). 9 Applications of Machine Learning from Day-to-Day Life. Last Accessed: 13th Oct 2019. Available from: https://medium.com/app-affairs/9-applications-of-machine-learning-from-day-to-day-life-112a47a429d0
2. maiti, t. (2019). Detecting a simple neural network architecture using NLP for email classification. Last Accessed: 13th Oct 2019. Available from: https://towardsdatascience.com/detecting-a-simple-neural-network-architecture-using-nlp-for-email-classification-f8e9e98742a7
3. ChessNetwork. (2017). Google's self-learning AI AlphaZero masters chess in 4 hours [Video]. YouTube. Viewed: 9th Nov 2019. Available From: https://www.youtube.com/watch?v=0g9SlVdv1PY
4. McClelland, S. (2019). Announcing Unity and Havok Physics for DOTS – Unity Blog. Last Accessed: 9th Nov 2019. Available from: https://blogs.unity3d.com/2019/03/19/announcing-unity-and-havok-physics-for-dots/
5. Maxion, R. (2019). Experimental Methods for Computer Science Research - IEEE Conference Publication. Available from: https://ieeexplore.ieee.org/document/5234306
6. Bahceci, E., D'Silva, T., Karpov, I., & Stanley, K. (2019). NNRG Software - NEAT C++. Last accessed: 15th Oct 2019. Available from: http://nn.cs.utexas.edu/keyword?neat-c
7. Faux, R. (2019). The Role of Observation in Computer Science Learning. Available from http://www.micsymposium.org/mics_2001/faux2.pdf
8. (UNIVERISTY OF TEXAS, University of Texas, 2010. NEAT C++ [library]. neural networks research group. [2nd Feb 2019]. Available from: http://nn.cs.utexas.edu/?neat-c
9. Mallawaarachchi, V., 2017. *Introduction To Genetic Algorithms — Including Example Code*. [online] Medium. Available at: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3 Accessed 19th Apr. 2020
10. Nguyen, C., 2015. This AI Used 'Neuroevolution' To Teach Itself How To Play 'Super Mario World'. [online] Vice. Available at: https://www.vice.com/en_us/article/kbz9mx/this-ai-used-neuroevolution-to-teach-itself-how-to-play-super-mario-world Accessed 19th Apr. 2020