Final Year Project Report

Learning to Play Wolfenstein 3-D

Gearóid Mac Ghiolla Coinnig

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science

Supervisor: Prof. Arthur Cater

UCD School of Computer Science
University College Dublin

March 16, 2019

Project Specification

General Information:

The goal is to develop a program able to play some early stages of the first person shooter video game "Wolfenstein", by learning to associate possible actions (moving in various ways, shooting) with what is perceptible in the game at any moment. A technique that proved successful in learning to play a Mario Bros level is to be applied to the new game. An existing open-source reimplementation of Wolfenstein will need to be adapted to provide the learner with information about what it perceives, where it is, how much ammunition it has, and how much time has elapsed.

The technique to be used for learning to play is a combination of neural network and genetic programming. Input neurons correspond to the presence of visible or measurable features (walls, enemies, power-ups, clock), and output neurons correspond to controller buttons (go left/right/forward/back, turn left/right/upward/downward, jump, shoot). By starting with a minimal network, adding random links from inputs to hidden nodes and onward to output nodes or perhaps other hidden nodes, and randomly adding new hidden nodes, different behaviours are obtained. By applying ideas of genetic algorithms, many individuals can be created and rated in terms of how much progress they make before dying and how soon they die. Mutation of, and crossover among, the more successful individuals of a generation leads over time to general improvement and ultimately, it is hoped, a really excellent player.

This technique succeeded in a Mario Bros game level, using inter-neuron links that had simple weights: excitatory or inhibitory. The Wolfenstein game has several similar characteristics, being deterministic and possessing something that can be used as a measure of progress (in Mario, a combination of distance from start and time taken was used but coins gathered were ignored).

Mandatory:

- Install the open-source reimplementation of Wolfenstein.
- Identify and implement code changes necessary to determine whether the hero has died, and if so, at what time and distance from starting.
- Identify and implement code changes necessary to allow a program rather than a human to control the character's actions.
- Identify and implement code changes necessary to allow a program to detect what is visible to the character at any moment in play.
- Design and implement a system for linking measurements of what can be detected in several of the floors of the first stage of Wolfenstein to activation of the player controls, using a small randomly generated system of neurons (nodes) and links that are either excitatory or inhibitory. (Such a system is virtually certain to die quickly.) Only small numbers of links in to or out from any node should be permitted at this stage, a maximum of six.
- Develop a way to combine parts of one random network with parts of another.

Discretionary:

- Design and develop a metric for comparing the degree of success of two networks, in terms of (large) distance travelled and (fast) time taken.
- Design and develop an evolutionary mechanism for taking a generation of several individual networks, picking the best few, performing crossovers and occasional mutations (new hidden nodes, wholly new links) in order to create a new generation of individuals.
- Apply this mechanism for at least 20 generations each consisting of at least 12 individuals. Measure the performances of the best, worst and median individuals in each generation.
- Apply the entire system to a third of the levels in the first Stage of Wolfenstein.

Exceptional:

- Apply this mechanism to substantially more generations, or substantially larger generations, or with more generous limits on in-degree and out-degree of nodes. Measure performance.
- Enrich the measure of performance, for example to reward the kills of enemies and the low use of ammunition.
- Apply the entire system to half or more of the levels in the first Stage of Wolfenstein.

Abstract

Wolfenstein3D is a first person shooter MS-DOS game that was released in 1992. The goal of the video game is to escape Castle Wolfenstein, a Nazi prison. Its creators, ID Software, released the source code for the game in 1995, meaning it is now possible to edit the source code for our own purpose.

The aim of the *Learning to Play Wolfenstein 3-D* project is to replace a human player with a computer that progressively learns to play Wolfenstein using two Machine Learning techniques, Genetic Algorithms and Neural Networks. These algorithms are implemented according to an algorithm called *NeuroEvolution of Augmenting Topologies (NEAT)* which is based on a paper by Kenneth Stanley and Risto Miikkulainen written in 2002 [1].

Part of the NEAT algorithm is dedicated to describing how a genetic population ought to be represented. A Genotype represents an individual from a genetic population. Each Genotype has a list of Genes which describes connections between Neurons in what is known as a Phenotype or Neural Network. In this document, Genotypes will be referred to as *individuals* and Phenotypes will be referred to as an individual's *network*. NEATDoop is the name given to the AI that was created as a result of implementing the software for this project. NEATDoop contains a population of individuals which will be used to learn to play Wolfenstein. Doop stands for *Developing Object-Oriented Program* and is sometimes referred to in this document as *the AI*.

NEATDoop's learning will be aided by a fitness function that measures the success of an individual's network when it was used to play Wolfenstein, which is a requirement for any project of this nature. By using previous individuals a new, hopefully better, individual will be generated who's network will then be used to play Wolfenstein. The end goal of this project is that NEATDoop is a fully functioning AI that is capable of learning how to play some levels of the Wolfenstein campaign.

Acknowledgments

I wish to express my sincere gratitude to the various community forum pages that have aided me in setting up such a strong foundation for my project. These include, but are not limited, to the DoomWorld forums, DRD team forums, StackOverflow, Wolfenstein3D Dome and the Wolf3D haven forums.

I would like to personally thank Iona Chera for providing me with information and feedback on my initial project concepts. I would also like to thank him for outlining various ways in which I could work with the source code for Wolfenstein.

I want to thank my fellow classmates Joe Duffin and James Keating for always finding the time to help me with problems I have had with my project. Without their help I fear that this project may not have been as complete as it is to date.

Finally I would like to thank my supervisor Prof. Arthur Cater. Over the past year he has continually provided me with support and feedback on my project and I cannot thank him enough for his time and patience.

Table of Contents

Chapter 1: **Introduction**

The inspiration for this project came from a YouTube video that was published over a year ago by a content creator named *Seth / SethBling*. The algorithm that Seth used to implement his AI was based on a paper called *Evolving Neural Networks through Augmented Topologies*. [1] Seth's video describes how his AI learns to play and complete a level of *Super Mario World* using a combination of two Machine Learning techniques; Neural Networks and Genetic Algorithms.

Wolfenstein3D is a first person shooter MS-DOS game that was released in 1992. The goal of the video game is to escape Castle Wolfenstein, a Nazi prison. Its creators, ID Software, released the source code for the game in 1995, meaning it is now possible to edit the source code for one's own purpose.

The aim of the *Learning to Play Wolfenstein* project is to replace a human player with a computer that progressively learns to play Wolfenstein using two Machine Learning techniques, Genetic Algorithms and Neural Networks. These algorithms are implemented according to an algorithm called *NeuroEvolution of Augmenting Topologies (NEAT)* which is based on a paper by Kenneth Stanley and Risto Miikkulainen written in 2002 [1].

Part of the NEAT algorithm is dedicated to describing how a genetic population ought to be represented. A Genotype represents an individual from a genetic population. Each Genotype has a list of Genes which describes connections between Neurons in what is known as a Phenotype or Neural Network. In this document, Genotypes will be referred to as *individuals* and Phenotypes will be referred to as an individuals *network*. NEATDoop is the name given to the AI that learns to play Wolfenstein and was created as a result of implementing the software for this project. NEATDoop contains a population of individuals which will be used to learn to play Wolfenstein. Doop stands for *Developing Object-Oriented Program* and is sometimes referred to in this document as *the AI*.

NEATDoop's learning will be aided by a fitness function that measures the success of an individual's network when it was used to play Wolfenstein, which is a requirement for any project of this nature. By using previous individuals a new, hopefully better, individual will be generated who's network will then be used to play Wolfenstein. The end goal of this project is that NEATDoop is a fully functioning AI that is capable of learning how to play some levels of the Wolfenstein campaign.

Super Mario World is a mostly linear game, but by no means simple in terms of gameplay. You travel left and right on the screen to get to a particular point or objective on the map. There are points in Super Mario World's gameplay where the player needs to kill enemies, jump over obstacles or go down tubes to get to other parts of the map. The plan for NEATDoop is to extend the approach taken for Seth's AI and to introduce its concepts to a more complex game. Doing this will involve creating a more complex fitness function than was used for Seth's AI to better model the requirements of Wolfenstein.

This document details the approach taken in order to complete the *Learning to Play Wolfenstein* project specifications. Each chapter of this document introduces several topics, each of which are discussed in detail.

Chapter 2 will outline all the background research that was done for this project. Here, the process of selecting the game for this project will be talked about. An in-depth overview of the NEAT algorithm will be provided as well as definitions for Neural Networks and Genetic Algorithms in

general, and finally Wolfenstein's game mechanics will be briefly discussed.

Chapter 3 provides information on how the project was initially approached. This involved deciding what language to use for this project by selecting from different language reimplementations of the original Wolfenstein source code. Various important source files that were identified in Wolfenstein's source code will then be documented and other NEAT algorithm implementations that were used to aid in programming NEATDoop will be explained. It will also argue why the NEAT algorithm is well suited for this type of project.

Chapter 4 will explain how the NEATDoop's Neural Network works and concludes with an explanation of the interaction between the NEAT algorithm and the Wolfenstein source code.

Chapter 5 provides in-depth documentation and explanations for how core components of NEATDoop were implemented. It describes how the NEATDoop's surroundings are represented, how it learns to play the game, attempts that were made to try and speed up gameplay and what stopping conditions are used to stop networks from playing Wolfenstein when they do nothing useful. It will also explain how distances are calculated for the project's fitness function, what the final fitness function is and finally shows how learnt networks are stored and reloaded so that they can be replayed.

Chapter 6 is dedicated to testing and evaluations that were done for the *Learning to Play Wolfenstein* project. It will provide an analysis of the fitness functions used over the course of the project time frame, the testing that was done with NEATDoop and ends with an evaluation of NEATDoop.

The last chapter, Chapter 7, will provide some future work that could be done with this project and how certain components that were implemented might be changed. It will also provide a final conclusion for the *Learning to Play Wolfenstein* project.

1.1 Why This Project Was Chosen

Using Neural Networks to aid learning in systems has been of interest for a long time although, in practice, they have only recently become feasible (in the past twenty years or so) to use primarily because of how much computational time they require. Modern advancements in technology have made the use of Neural Networks to solve complex problems more and more possible.

At first, I thought that the best approach for developing an AI capable of playing a game would be to implement a rule-based system where the AI reacts deterministically to its surroundings. However, using an algorithm like NEAT allows an AI to learn this behaviour itself and is far more interesting both to implement and to watch.

Having the opportunity to use machine learning techniques that have revolutionised problem solving in the industry as well as being able to incorporate it into an area of computer science that I am very fond of was particularly appealing and is the reason why I proposed this project.

1.2 Specification Changes

Initially, part of this project's specifications was to create an AI capable of completing dozens of levels within the Wolfenstein campaign. Two parts of the initial specification have now changed with the agreement of my project supervisor, to the below.

- **Discretionary:**
  Old: Apply the entire system to all floors of the second stage of Wolfenstein
  New: Apply the entire system to a third of the levels in the first Stage of Wolfenstein.
- **Exceptional:**
  Old: Apply the entire system to some or all the floors of the second, and perhaps third, stage of Wolfenstein
  New: Apply the entire system to half or more of the levels in the first Stage of Wolfenstein.

The changes to the specifications were primarily due to the fact that it takes a considerable amount of time for NEATDoop to learn how to play just parts of a level and so letting it complete several levels was deemed impractical. As things stand, however, neither of the modified goals have been achieved, since the learning time required is even greater than had been feared.

Chapter 2: **Background Research**

This chapter will outline in detail the discoveries that were made whilst researching numerous relevant topics of interest to the *Learning to Play Wolfenstein* project. The first item that will be discussed will describe how the game for this project was selected; what caused Wolfenstein to be chosen over previously considered open-source games, such as Doom and Duke Nukem. However, the main topic of discussion in this chapter will be the *NeuroEvolution of Augmenting Topologies(NEAT)* algorithm; what it is and why it will be of major importance to NEATDoop's ability to learn.

2.1 Game Selection

In order for an AI to be able to learn to play Wolfenstein it will need to be able to make decisions based on what it can see at any point in time. This would not easily be done with a game whose source code was not open source since information could not be read directly from the source code to determine its surroundings.

The first open source game that was considered for this project was *Duke Nukem 3D*. It was released early 1996 and was developed by *3D Realms*. The game was discovered having read an analysis of the game's source code on a web-blog early into the specification for this project. [2]

The author of the web-blog recommended that I do not use *Duke Nukem 3D* due to its rotten codebase. [3] From inspection of an open source reimplementation of the source code called *Chocolate Duke Nukem 3D* the author's recommendation seemed well founded. The source code contains approximately 70,000 lines of code and is very, very sparingly documented. By comparing the game.c source file from *Duke Nukem 3D* and the WL_GAME.c source file from Wolfenstein it is now very clear that not choosing *Duke Nukem 3D* was a solid decision. Both of these source files implement logic that sets up levels, constructs in game text, displays player health statistics and includes logic that is fundamental to the main game loop. WL_GAME.c from *Wolfenstein's* source code contains approximately 1,600 lines of code whereas game.c from *Duke Nukem 3D's* source code alone roughly contains 11,000 lines of code with no substantial documentation indicating how the source file works. [4][5]

The Original Doom was then considered as it was recommended by the author of the previously mentioned web-blog. A software developer who created a reimplementation of the Doom source port called *AutoDoom* recommended that *Wolfenstein 3D* be considered for this project as opposed to Doom due to the fact that Doom's level design is a lot more complex than Wolfenstein's. This is evident from any gameplay showcasing the games. [6]

Doom allows for players to move in the vertical axis. This extra dimension that a player can move in would have been a huge roadblock in NEATDoop's ability to learn. *Wolfenstein's* map design is completely flat i.e. the player cannot move in the vertical axis. A worst case scenario for NEATDoop playing Wolfenstein is that it gets stuck in a corner or runs in a circle. Not only would this be a concern in Doom but there exists a probability that NEATDoop would get stuck behind a staircase or similar. The fitness function for Doom would also have to take into account the extra axis which would have resulted in an even more complex measure of fitness.

As a result of the outlined problems, Wolfenstein was chosen for this project because the source code is well documented. As well as this, the source code, whilst still containing approximately 45,000 lines of code, is easy to navigate and understand. The source code that was chosen is actually not the original one that was written in C but a reimplementation written in C++ called *Wolf4SDL*. [7] The main reason behind this choice was so any code written to interact with the Wolfenstein source code could be object oriented. Further reasons will be discussed later in Chapter 3.

2.2 NEAT Algorithm

NEAT stands for *Neuro-Evolution of Augmenting Topologies* and is based on an paper by Kenneth Stanley and Risto Miikkulainen [1]. The paper demonstrates the ability for the NEAT algorithm to solve problems in quicker succession than other types of topology evolving algorithms, such as *Topology and Weight Evolving Artificial Neural Networks (TWEANN)* algorithms [8]. The NEAT algorithm consists of two very important Machine Learning techniques, Neural Networks and Genetic Algorithms.

2.2.1 NEAT Neural Networks

Neural Networks(NN) are a Machine Learning technique that roughly simulate the behaviour of a brain. They consist of artificial Neurons that are connected using artificial Synapses. The synapses in a NN typically have a weight associated with them that describes how strong a connection any two Neurons have with each other. [9]

In traditional Neuro-evolution techniques, a topology is chosen before any experimentation begins. This topology is normally maximally connected, meaning that every Neuron in the input layer is connected to every Neuron in the output layer. If the topology has an intermediate layer, known as a hidden layer, then the input layer is instead maximally connected to it, and it is then maximally connected to the output layer. This network then receives inputs to be processed which are propagated through the network starting from the input layer until, ultimately, reaching the output layer where a result is produced.

This network is then modified by means of mutating the weights on the links of the network using evolutionary techniques such as genetic algorithms. The goal of this type of neuro-evolution is therefore to optimise the weight matrix associated with the network.

In order to determine what Neurons are active at any point during network analysis, an *activation function* is used. As stated in NEAT [1], a modified Sigmoidal transfer function is used, $\varphi(x) = \frac{1}{1+e^{-4.9x}}$. This is not the standard Sigmoidal function used in traditional neuro-evolution techniques due to its use of a coefficient. This coefficient causes activations to be close to linear at the Sigmoid's steepest ascent between -0.5 and 0.5 resulting in more possibilities for fine-tuning at the extremes of the modified sigmoid function, since it does not plateau as fast as a standard sigmoid.

The weights between Neurons are not the only aspect of a Neural Network that contribute to their behaviour. The structure of a neural network also affects its functionality. The NEAT algorithm is primarily focused on this aspect of neuroevolution. It extends and tries to improve

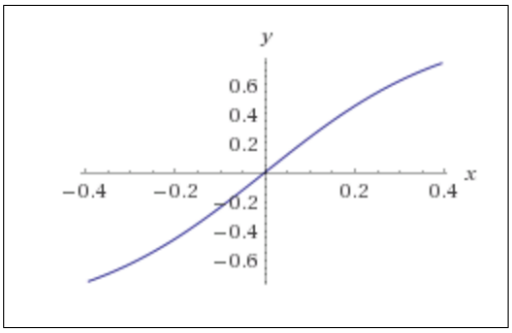

Figure 2.1: Modified Sigmoidal activation function used for NEAT algorithm

on some popular techniques utilised by some TWEANNs. The two main ideas it introduces are the *Speciation* of a population and using *Innovation Numbers* on network encodings so that the historical origins of each network can be tracked. These concepts try to counter some of the common issues with typical TWEANNs. [10]

NEAT Encoding

Typical neural networks consist of Neurons (nodes) connected using Synapses (links). NEAT describes its networks using a type of direct encoding whereby each network is represented using a series of *Genes*. Each Gene indicates two Neurons that are connected, whether or not the link is enabled, a weight and an *innovation number* that is unique to each Gene. This innovation number is used to calculate similarities between networks.

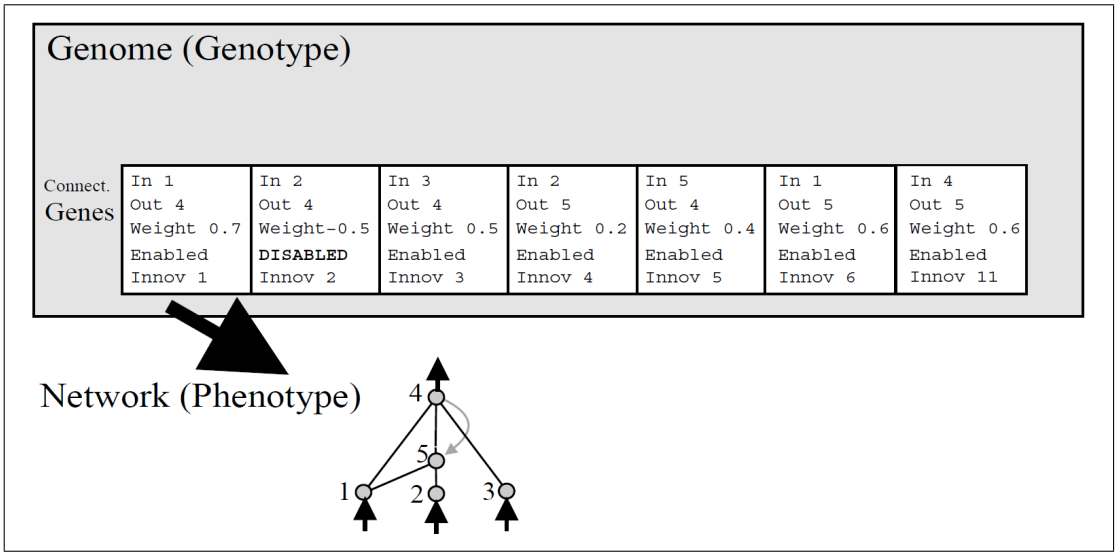

Figure 2.2: Shows a mapping from an individual (Genotype) to a network (Phenotype).

Fig 2.2 is taken directly from the NEAT paper [1] and indicates how an individual's Genes map to a network. Notice that the innovation numbers associated with each Gene in this network are not increasing uniformly. This is an example where Genes would have been added to another individual before the last Gene in this individual was added.

NEAT Speciation

Many problems arise when experimenting with networks that involve modifying the structure of the network as well as the weights on links in order to produce better offspring. One such problem

is that, in many cases, modifying a network causes an initial decrease in an individual's fitness. As a result, the topological innovation is very unlikely to make it through to the next generation where it has the potential to be improved.

In order to counter the above problem, the NEAT algorithm uses *Speciation* on its population. This is done by grouping individuals by their networks if they share similar enough genetic history.

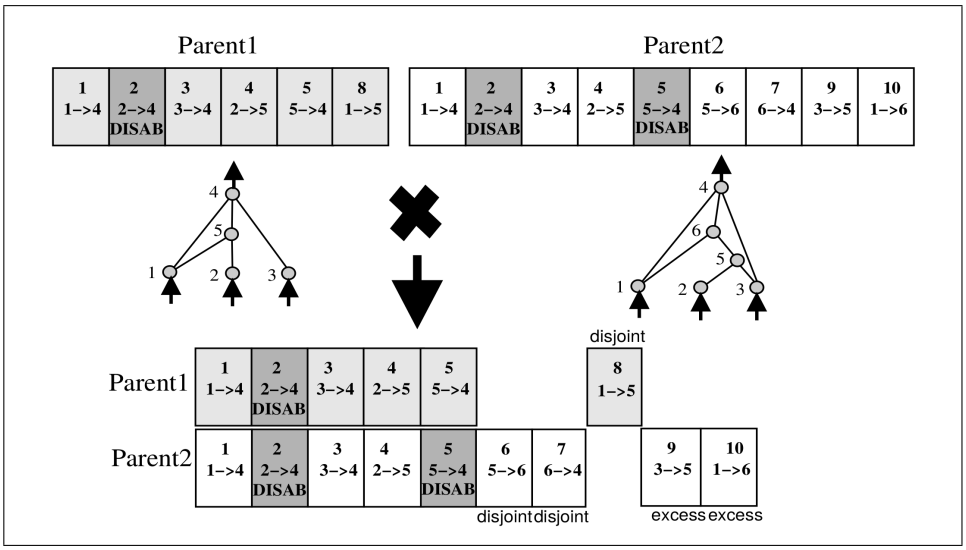

Figure 2.3: Shows a comparison of the Genes in two different individuals.

The bottom portion of Figure 2.3 shows how comparing genetic history between two individuals might work. The Genes that appear in both individuals are lined up. These Genes are referred to as *Matching Genes*. Any Genes that do not match are considered to either be *Disjoint* or *Excess* depending on whether the mismatch appears in the middle of the comparison or at the end. Calculating Gene similarities between two individuals will be important when performing crossovers, which will be discussed in the next section.

There are a number of implications of Speciation. Firstly, structural innovations have a better chance of making it through to the next generation where they can be further improved. Secondly, it reduces the chances of a single individual dominating the entire population.

Speciation is a concept that most TWEANNs do not employ, as indicated in the NEAT paper. Innovative structures in TWEANNs tend to have more connections and as such take far longer to improve than simpler ones. The result of this is that innovative structures in TWEANNs cannot compete with simpler ones.

2.2.2 NEAT Genetic Algorithms

Genetic Algorithms(GA) are a set of rules that try to describe how simulated evolution might work with an artificial population. The population consists of a set of individuals that are to be subject to these rules of evolution. The individuals are evaluated and their relative success measured according to some *Fitness Function*. Crossovers are then performed by combining aspects of two parent individuals to create a new child individual. Individuals are then mutated by making small random changes to them in order to add genetic diversity to the population. Depending on the implementation, a number of individuals are then moved forward to the next generation where the same rules are applied to the new, modified population. This process is repeated until a terminal condition is met. In the case of this project the terminal condition will ultimately be NEATDoop completing a level.

The GA used for NEAT is a variation of these standard rules, since the population is *speciated* / sub-divided the rules are altered slightly.

1. **Initialisation:** As was mentioned in Subsection 2.2.1, the population in the NEAT algorithm is Speciated / sub-divided so that clusters of similar individuals are created. This is how the population in NEAT is created. Each individual starts with a network that is minimally connected, meaning that there is no hidden layer initially; only an input and output layer, which may have some connections as a mutation is applied to each initial network.
2. **Evaluation:** This step involves taking each network in the population and feeding it inputs in order to produce some output. The fitness or success of the network is then calculated according to some fitness function which is used as a measure for comparison between networks.

   As stated in NEAT [1], *explicit fitness sharing* is used between the individuals of a particular niche. This value is assigned the fitness of the highest performing individual in that niche. It does this so that no single niche or *Species*, as they are also called, can take over the entire population even if all of its members are high performing.
3. **Selection:** Once all the individuals in the population have been evaluated, niches / species reproduce by first eliminating the lowest performing individuals and then the entire population is replaced by the offspring of the remaining individuals in each niche / species.

   In the implementation for this project, only the best individual per species is actually brought forward to the next generation. Since all individuals in a species are either equal in performance or worse than the current best performer, removing the weak individuals creates a higher chance that new ones generated will perform better than the current best.
4. **Crossover:** Crossover produces new offspring for the next generation by taking two parent individuals from the same species and combining aspects of them to produce a child. The parent individuals always come from the same species since the networks of each are similar enough for this crossover to work properly.

   If the networks were not similar then too many disjoint and excess Genes would be used in the child causing the genetic history shared to become tainted.
5. **Mutation:** In order to introduce genetic diversity within the population, mutations are performed on the population. Types of mutations include:
   - NEAT topological modifications (Discussed in the next section)
   - Enabling / disabling existing links in a network
   - Modifying an individual's mutation rates
6. **Repeat process until terminal condition met**

2.2.3 NEAT Topological Mutations

NEAT specifies two types of topological / network mutations [1], *Node insertion mutations* and *Link insertion mutations*. Both of these mutations add new connection Genes to an individual and each new Gene is assigned a new, unique Innovation Number.

- **Link insertion mutation:** In a link insertion mutation a new connection Gene is added to the individual, specifying an in-neuron and out-neuron. Both of these neurons are chosen randomly from the individual's pool of existing neurons. If a link already exists between these two neurons then no new link is added. Fig 2.4 shows that the neurons selected were 3 and 5 in this case.

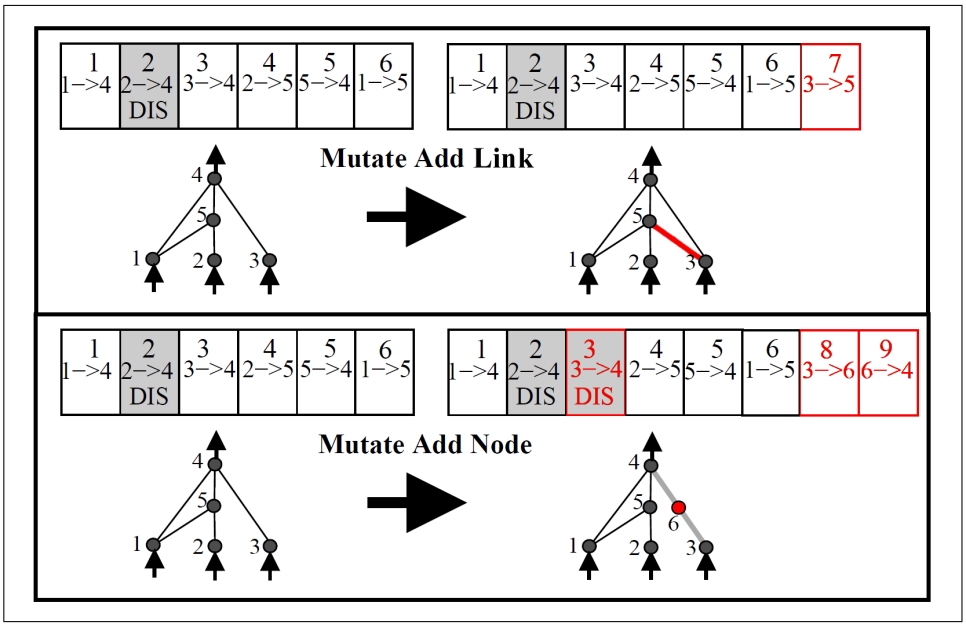

Figure 2.4: Shows an individual's network before and after NEAT Link and Node insertion mutations.

- **Node insertion mutation:** A node insertion mutation involves randomly selecting a Gene from an individual. This Gene is then set to be disabled, meaning there is no longer a direct connection between the Gene's in-neuron and out-neuron. Fig 2.4 above shows the Gene selected has 3 as its in-neuron and 4 as its out-neuron.

  Once the selected Gene is disabled, two new connection Genes are created and added to the individual. Notice that the numbers at the top of the displayed diagrams are actually the innovation numbers on each connection Gene. The innovation numbers assigned to the new Genes in the node insertion mutation assume that the link insertion mutation happened first to some other network, which is why the first Gene here has an innovation of eight and not seven.

  - The first Gene has the old in-neuron as its in-neuron and the newly added neuron as its out-neuron (3 -> 6)
  - The second Gene has the newly added neuron as its in-neuron and the old out-neuron as its out-neuron. (6 -> 4)

2.2.4 NEAT Individual Mutations

NEAT also mutates various aspects of individuals. Below, a brief description of these mutations are given:

- **Enable / Disable Mutation:** In this mutation, there is a chance that a random enabled / disabled Gene from an individual is chosen and its state flipped, meaning if the Gene selected is currently enabled it is disabled, and vice versa.
- **Weight Mutation:** This mutation causes a random Gene from an individual to be selected and its weight mutated.
- **Mutation Rate Alterations:** Every individual in the population has a certain chance of performing any of the previously mentioned mutations. These mutation rates can also be mutated.

2.3 Wolfenstein 3-D Game Mechanics

By the very nature of this project it is of high importance to be able to understand the source code of the game in order to determine what information can and cannot be used for NEATDoop's learning. This section will introduce how Wolfenstein's gameplay works and also give a brief description of some parts of the source code where attention was focused early on. The purpose of reading the source code at this stage was to understand how the Wolfenstein map is represented and how objects, such as enemies and pickups, are tracked throughout gameplay. These components will form an important base for the NEAT algorithm which is discussed after this section.

2.3.1 Wolfenstein's Gameplay

Wolfenstein is a First-Person Shooter(FPS) and the goal of each level is to get to an elevator tile that brings the character to the next level. A player reaches the end of each level by moving through a series of rooms in the game, potentially killing enemies and picking up items such as health, ammo, treasure or new guns as it does so.

Depending on the difficulty selected, a variety of enemies will spawn in the game. These enemies will attempt to kill the player on sight. The player is equipped with a low-power pistol and a knife by default, although it is possible to pick up a sub-machine gun or a chain gun in various levels that offers a higher damage output than the pistol.

Damage is done on a distance basis. The closer you are to an enemy when you hit them, the more damage that is inflicted. This also applies to enemies, the closer they are to you the more damage they do when they hit you.

Enemies react to sound. On hearing a gunshot, an enemy will move in the direction of the gunshot. This will be the cause of a substantial amount of deaths in NEATDoop's learning where it shoots randomly and attracts enemies that ultimately, kill it.

Wolfenstein is a mostly deterministic game. Enemies and items that can be picked up, always spawn in the same location. However, enemy movements are not deterministic and can cause network replays to behave slightly differently.

2.3.2 Map Representation

Every map in Wolfenstein is represented as a 64x64 grid of tiles. It holds *Byte* values indicating the type of structure located at that (x,y) co-ordinate. If the *Byte* value at a particular (x,y) co-ordinate is greater than 0, then it can either be a wall, push-wall, door or an elevator tile (which is the tile associated with the end of the current level). If the value at any (x,y) co-ordinate is 0 then it represents a plain tile that a player can walk on.

This particular information is vitally important for determining the structural surroundings of NEATDoop during gameplay. From this *2 dimensional array of bytes* it is possible to get the spawn location in each level, the end location in each level and all wall, door and push-wall data that will be used primarily as input to the neural networks of the population.

The *elevator tile* is a particularly important tile. On this tile a button is located that, when pressed by the player, ends the current level. Reaching the elevator tile indicates that the AI (NEATDoop) will have managed to learn to play a level of Wolfenstein. The co-ordinates of this

tile position will be used later in the fitness function as will the spawn co-ordinates.

2.3.3 Enemy/Item Representation

Both enemies and items that can be picked up are represented as *structs* within the source code.

So called *Actors*, represent enemies and the player of the game. The *struct* storing their information is named *objstruct* and contains information about the position of the Actor in the map, the health the Actor has, the sprite image assigned to it and references to the next and previous *objstruct* etc. A list of objstructs is maintained with the name *objlist* which stores all Actors currently alive on the map.

Items that can be picked up include things like guns, ammo, health and treasure. These are considered to be *static items* in the Wolfenstein Engine since they do not move. However, other items such as chairs, tables and statues are also considered to be static. The *struct* that represents these static objects is called *statstruct* and a list maintaining references to all static items currently on the map is called *statobjlist*.

Identifying static items of interest is simple since each static item has a label associated with it. For instance, ammo has a *bo_clip* label associated with it so it is possible to iterate through the *statobjlist* array and find items that would be considered useful for NEATDoop to pickup.

This chapter provided an overview of the work that was done early into the project timeline. The NEAT algorithm was introduced which will be core to NEATDoop's ability to learn. Finally, having come to a decision on which game to use, the next natural step was to gain some insight into how the most important information is represented as well as understanding how the mechanics for the game work so that some factors from it could be used later on when designing the fitness function.

Chapter 3: **Project Approach**

The previous chapter introduced the game that was used for this project and outlined the NEAT algorithm, which is at the very core of this project. This chapter will introduce the reimplementation of the original Wolfenstein source code used throughout the project, describe the methods employed in order to identify and understand the most important source files from the Wolfenstein source code and the importance of reading other adaptations of the NEAT algorithm.

3.1 Selecting the Source Port

Wolfenstein 3-D was originally written in a combination of the C and Assembly languages. Very early on in this project it was desired to find another adaptation of the Wolfenstein source so that integrating the NEAT algorithm into the source code would not involve writing in either of these languages. This was primarily because the NEAT algorithm is much easier to understand if it is implemented in an Object-Oriented language such as C++. As such, a source code adaptation implemented in C++ was sought after.

Initially, the original source code was used. This involved using a compiler known as *Borland C++ v3.1*, which is an old, outdated piece of software, in order to compile the source code. The *Borland* compiler is a 16-bit compiler requiring it to be run within *DosBox*, since it is not possible to run the compiler natively on a 64-bit machine. It was decided eventually that this means of compiling and executing the source code was not ideal and so alternatives were considered.

Since it was desired that a C++ implementation of the source code be used, *Wolf4SDL* was a promising candidate. This particular adaptation is essentially the same as the original source code but it is commented more generously and allows the possibility for any integrated code to be written in an object oriented language. Compiling the source code was still not ideal with this adaptation, as it requires the use of an old version of C++, but it at least can run natively in a 64-bit Windows environment.

3.2 Understanding the Source Code

Once the *Wolf4SDL* source code was selected for this project the core source files needed to be identified. This was a troublesome task. No adequate documentation for the source code exists online and even though the *Wolf4SDL* source code is essentially identical to the original source, only that it is written in C++, no in-depth documentation exists even for the original source.

Understanding the source code involved reading through certain source files that appeared important. *wl_main* was first explored. It seemed a natural starting point since it contains the main game loop for the game and at that point it was assumed that any other immediately important source files would be referenced from here. In order to quickly visualise the source files that were directly referenced by *wl_main* a program known as *Doxygen* was used to model dependencies

between the C++ source files of Wolf4SDL. [11]

This was predominantly how the rest of the important source files were found. Below, a list is provided describing some of the source files. These source files are considered important because they either contain necessary information for NEATDoop to utilise or because they tie directly in with the life-cycle of the game and were modified so that NEATDoop could take the place of a human player.

- **wl_def.h:** This is a sort of *god source file*. ID (Wolfenstein's authors) designed this source file to contain function, variable and macro definitions for use by all other source files. This source file is where most of NEATDoop variables are defined. This makes it easy for both the NEAT source files and the existing Wolfenstein source files to access them without having to make modifications to many of the Wolfenstein source files (since all relevant Wolfenstein source files include wl_def).

- **wl_main.cpp:** This source file contains the main loop for the game. It checks command line arguments that were set for the game, such as *–tedlevel*. This command is used to start the game on a particular level, which is used when replaying individuals. The use of this command is necessary since the main menu does not allow selecting individual levels within each stage.

  The main use of this source file will be to set up the NEAT population at the start of the main game loop.

- **wl_draw.cpp:** As is pretty obvious from the source file's name, this source file draws what is visible on the screen. It scans in an area around the player's current location searching for enemies, pickups, walls, doors etc. Once it locates all of the items that the player can see it uses ray-casting in order to generate a *frame*, which is a graphical representation of the Wolfenstein world that is displayed on a computer screen. Frames are generated in Wolfenstein at a rate of seventy per second.

  In order to give the *illusion* that the game is 3D, logic is used to change the scale of sprites for walls, enemies and pickups according to the distance of the object from the player's current location. The function *AsmRefresh()* implements this logic.

  It was not necessary for this project to understand the exact way in which the code works for this source file. Once it was discovered that the function *DrawScaleds()* finds all visible items in close proximity to the player it was just a matter of incorporating the NEATDoop logic into it.

- **wl_play.cpp:** This source file contains a function called *PlayLoop()* which contains logic that is executed repeatedly while a level is being played. It checks the keyboard for buttons presses every frame of the game and checks the state of the player (have they died, have they finished the level). It updates frames by making calls to the *AsmRefresh()* from *wl_draw.cpp*.

  At this point it was understood how instead of polling the keyboard for controls, it would be possible to set the keyboard buttons (which maintain a true or false value) to the boolean values outputted by the individual's network that is currently playing the game.

- **wl_game.cpp:** If the state of the player changes during an iteration of the *PlayLoop()* in the *wl_play* source file, the loop exits and returns to the logic defined in wl_game. wl_game's logic switches on the player state to see what further logic is to be executed. In the case that the player died, the player state is set to *ex_died*. The *wl_game* source file then handles this case, amongst others in a function called *GameLoop()*. This is really all that is necessary to understand in this source file.

- **wl_agent.cpp:** When the player picks an item up, such as a new gun or ammo, functions in this source file are called. These functions add the ammo to the player ammo pool etc.

- **wl_act1.cpp:** This source file is similar to the above source file but handles door / push wall interactions. When the player opens a door / push wall, functions in this source file are called.

3.3 Why NEAT?

As introduced in Section 2.2.2 the population for this project consists of a host of individuals whose networks will be used to play Wolfenstein. Evaluating all of these individuals is very time and resource consuming. As such it is vital that each individual's network start as minimally connected as possible and only modified when necessary. This very requirement immediately rules out TWEANN algorithms because in order to ensure diversity in the population they often heavily randomise the topologies of each network in the beginning. This introduces problems where it is required to search through the population and remove poorly performing members. This is very time consuming and as it turns out, avoidable.

3.3.1 The Problem with TWEANNs

Whilst TWEANN solutions are in effect, very similar to the NEAT solution, they have trouble both with efficiently introducing genetic diversity to their population and with protecting structural innovations that are introduced as a result of mutations. Since TWEANN solutions generally create individuals with random topologies when setting up the initial population, a lot of unnecessary computational effort is spent trying to find individuals whose structural innovations are worth the trade-off of extra computational evaluation time. [16]

It may be the case that two individuals have the same functionality but one has a topology that is inherently more complex than the other; In which case, the TWEANN solution would have just wasted computational time for no valid reason. NEAT avoids this by starting every network with the simplest topology possible, keeping structural innovations only when they provide an improvement. This has the effect of reducing the number of parameters in each topology that need to be explored in order to produce an output from the network.

Some TWEANNs attempt to protect structural innovations by adding non-functional structure to networks in the hope that the new innovation will be used at some point in the future. [15] Unfortunately, in the event that no useful connections are incorporated with the new innovation it results in extra parameters being added to the search space, further reducing the performance of the network.

3.3.2 The Effectiveness of NEAT

NEAT is particularly attractive in a scenario where efficient learning of a complex input space is required. If this was important for Seth's project then it is even more important for this project since Wolfenstein's gameplay is more complicated than Mario's. As will be seen in Chapter 6 a significant amount of time is necessary for NEATDoop to learn even the simplest game mechanic's in Wolfenstein.

Run-time is the main reason why NEAT was selected. Since NEAT avoids many problems that TWEANNs have, it runs much faster. Seth, the author of the previously mentioned YouTube

video, had to let his AI learn for twenty-four hours before it was able to complete the first level of Mario. Since Wolfenstein's gameplay is more complex than Mario's, it is not unreasonable to assume that it would take longer than this in order for NEATDoop to learn a substantial portion of gameplay for a single level. Had TWEANNs been used instead of NEAT the training duration would have had to be even longer.

3.4 Examining Existing NEAT Implementations

This project was inspired by a video that was published on YouTube about a year and a half ago. In it, the author described how he had designed and implemented an AI capable of playing and completing the first level of Mario. The source that he wrote in order to accomplish this is open-source and served as an initial reference to understand the way in which the NEAT algorithm should be implemented.

3.4.1 MarIO

MarI/O is the AI that was created by author *SethBling / Seth* as a result of him implementing the NEAT algorithm to play Mario. He wrote the NEAT algorithm in Lua as a plug-in for an emulator called *BizHawk*. The implementation of the NEAT algorithm that he wrote does not strictly follow the rules of NEAT, but it is sufficient enough to closely follow its guidelines and produce similar results. [12]

The implementation of NEAT that was used in this project is based off Seth's implementation. [13] The Lua script that was written for his project was translated to C++ to be used for Wolfenstein. The main difference between the implementations is that the one used for NEATDoop is object-oriented and not a script.

Translating the Lua script was not a straight forward process. Since the version of C++ used for this project is outdated many data structures and built in functions could not be used and alternatives needed to be used. Simple things such as trying to convert a *double* value to a *std::string* value are not trivial in C++98. In C++11, this is an easy task. The function *std::to_string(doubleVal)* will do this converting, however this is not available in C++98.

3.4.2 NEATFlappyBird

NEATFlappyBird is another implementation of NEAT that was explored in order to understand how the algorithm works. This version of NEAT was not actually used for NEATDoop's implementation directly, but the design principles for the objects were referred to since this version of NEAT was written in *Java*. It was created by a French student named Alex and was found on his *GitHub* from a simple search. [14]

This Chapter focused heavily on the actual approach that was taken having finished in-depth background research on the NEAT algorithm and the Wolfenstein game. It included a discussion on why the *Wolf4SDL* source code was used for this project, gave a brief description on the core source files in the Wolfenstein source code and finally discussed what existing implementations of the NEAT algorithm were referred to in order to design and implement the NEAT algorithm for the *Learning to Play Wolfenstein* project.

Chapter 4: **Design Aspects**

Chapter 3 focused on some of the tasks that were completed early on in the project time frame in order to build a strong foundation for the actual implementation of the *Learning to Play Wolfenstein* project. This chapter describes how a NEATDoop neural network works and demonstrates the interaction between the Wolfenstein gameplay and the NEAT algorithm.

4.1 Understanding the NEATDoop Neural Network

The previous section details why it is important that each network used to play Wolfenstein be efficient. This section focuses on understanding how different networks topologies cause different gameplay interactions with Wolfenstein. Here, it is important to note that the way networks are built and manipulated is directly affected by the fitness function used for evaluating the success of each network.

As was outlined in Sub-section 2.2.3, NEAT mutates a network's topology using two types of mutations. There are a few ways these can affect NEATDoop's networks:

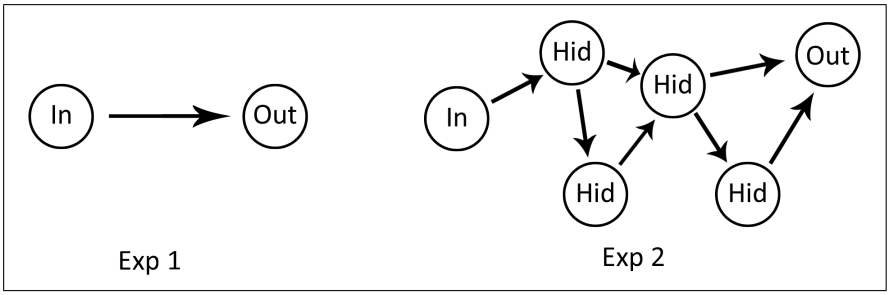

Figure 4.1: Shows two types of Network / Phenotype topologies.

- The inputs for this project, in its current state, are represented as eleven 5x5 matrices representing the space around the player. Each of these matrices represents a specific thing that NEATDoop can see e.g enemies, ammo, health. Node insertion mutations can cause a previously unused input to be utilised. This will attach a Neuron to a cell in one of the eleven matrices. Depending on whether or not the Neuron link is enabled or disabled, when the cell reads a positive value the Neuron may be activated. Node insertion mutations can also create new neurons within the hidden layer of a network, which are either connected internally within the hidden layer or to the output layer.
- Node insertion mutations can cause previously used inputs to be deactivated. This simply happens when the Gene that links the Input Neuron to some other Neuron in the network is disabled.
- Network behaviour can either be very simple or rather complex. Consider, for example, scenarios as depicted in Fig 4.1. In the first network, an input Neuron is connected directly to an output Neuron. When this input Neuron is activated there exists a chance that the output Neuron will also be activated. This behaviour is predominantly seen in early stages of

training since each network initially only consists of input and output layers. The activation of a Neuron is calculated according to the below formula.

$$\rho = \varphi(\sum_{i=0}^{n} input\_weight_i \cdot \rho_i)$$

This formula denotes that the activation on any Neuron is the sigmoid (the modified sigmoid which was introduced in Subsection 2.2.1) of the sum of the weights of each input link to the Neuron multiplied by the activation of the other Neuron on that link. If ρ, the activation, is greater than zero for a particular Neuron, then it is considered to be activated.

The second network might be seen after some amount of training. In this case, if the input Neuron is active, the output Neuron has either a greater or smaller chance of being activated depending on the activations of the numerous Neurons in the hidden layer.

- The networks in the *Learning to Play Wolfenstein*project each have nine outputs. Each is given an output Neuron in the output layer of each network. These outputs represent the nine buttons that NEATDoop can press during gameplay e.g move forward, turn left, shoot etc.

The fitness function can easily bias the way future individuals are generated. By rewarding NEATDoop for completing certain tasks, subsequent generations may modify networks to perform in unexpected ways. This exact behaviour was seen in early testing where NEATDoop was allowed to keep playing Wolfenstein so long as it kept moving. It biased some of the networks to just keep running in a circle, forever.

4.2 Wolfenstein and NEAT Interaction

Now that a comprehensive description of the NEAT algorithm has been given the next step is to understand its interaction with Wolfenstein's gameplay. This will be the focus of this section.

Fig 4.2 shows a dependency diagram showing the interaction between NEATDoop source files and some components of the Wolfenstein source.

wl_def.h creates an instance of the NEATDoop object. Since this source file is included by most Wolfenstein source files, they have access to the NEATDoop object.

Circular dependencies between *Genome.h, NEATDoop.h* and *wl_def.h* were avoided by moving some constant definitions to a new source file, *Def.h*.

Source file *Genus.h* is comparable to the *Pool* defined in the Seth's project. A Genus consists of a group of *Species* and is implemented as a static class in this project, since only one Genus will exist.

Neurons have a list of *Genes*, which are all the Genes whose out-neurons connect to a specific Neuron.

In this section, high level definitions were given for some of the essential design aspects of the *Learning to Play Wolfenstein* project. These included the design principles behind using the NEAT algorithm instead of other types of Topological altering algorithms, how NEATDoop's Neural Network evolves and the source interaction between the NEAT algorithm and the main Wolfenstein source files. The next Chapter, Detailed Design and Implementation, will give a more technical description as to how several components of NEATDoop were implemented.

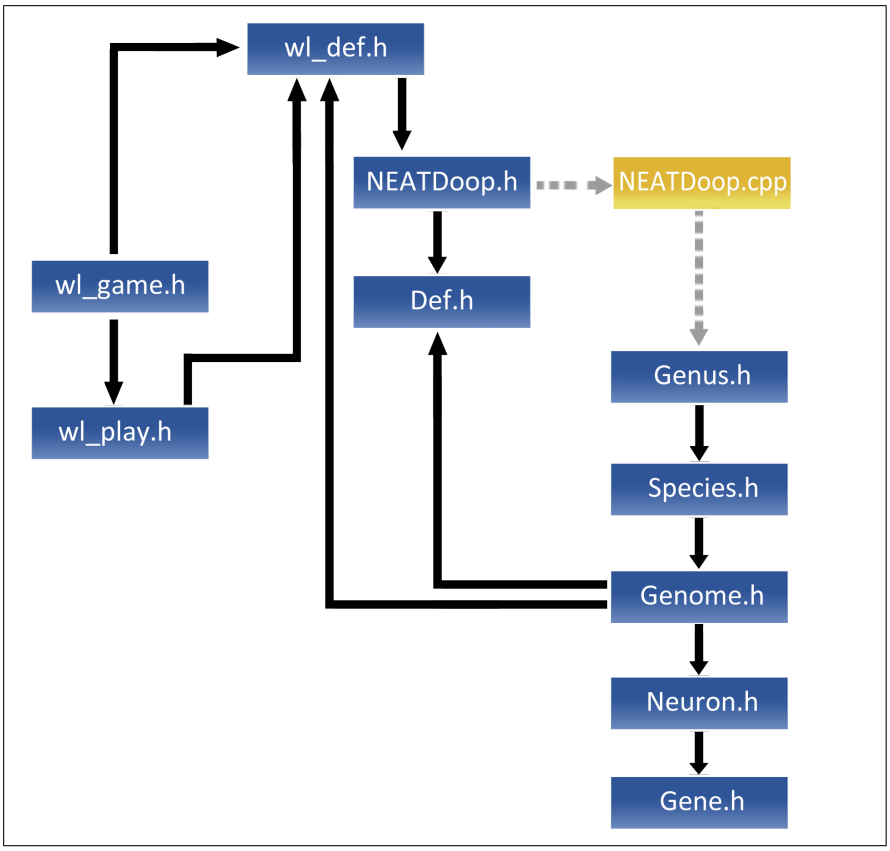

Figure 4.2: Shows a dependency diagram of a subset of important interactions between some of the Wolfenstein source files and the NEAT algorithm. The dulled line indicates NEATDoop.h using the static class Genus in its source file

Chapter 5: **Detailed Design and Implementation**

Chapter 4 gave a high level description for some of the design aspects of the *Learning to Play Wolfenstein* project. These aspects included the reasoning behind the use of the NEAT algorithm and the interaction between Wolfenstein's gameplay and NEAT algorithm. The main focus of this chapter is to describe how NEATDoop interacts with components of the source code of Wolfenstein and how, as a result of this interaction, it is able to identify objects during gameplay and make decisions based on what it can see. This will also lead to discussions of how NEATDoop's learning process is aided by assigning every attempt a measure of success and of how an individual is able to be replayed.

5.1 Giving NEATDoop Game Vision

In order for NEATDoop to make decisions during gameplay it needs to be aware of its surroundings. In order to do this it is necessary to identify how and where objects in the game are represented. As was mentioned in Subsection 2.3.3 there are *structs* in the Wolfenstein source that describe enemies and pick-ups, and *arrays* that represent walls, doors and walkable space. By accessing these data structures it is possible to provide NEATDoop with so called *Game Vision*.

There is a special struct pointer in the Wolfenstein source that contains information about the player. This pointer is used in the *wl_draw.cpp* source file in order to identify what the inputs to an individual's network ought to be. Specifically, a special function called *GetInputs()* made specifically for this project, is called and uses the player location in order to scan an area around their position and identify important information.

Depending on what lies in the tiles surrounding the player, specific cells in specific arrays are set to 1 which indicates something is there (otherwise they remain 0). Recall that the network inputs are eleven 5x5 grids. Below is a small example as to how some of the arrays of the input might be set for the given tile map on the left side of Fig. 5.1. This demonstrates something similar to how the inputs are actually set within the game, frame by frame.

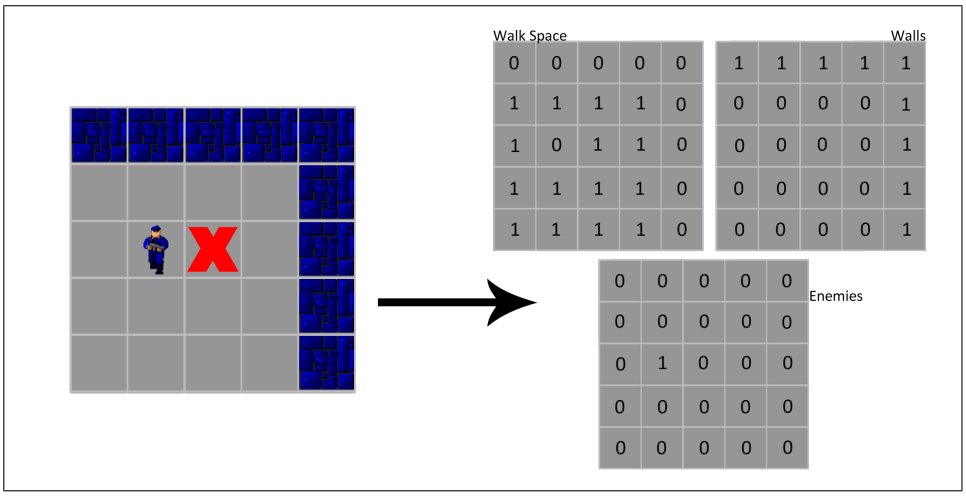

Figure 5.1: Shows a mapping from a sample Wolfenstein map state to network input.

The above is a simple example. The way certain inputs are set can change during gameplay. For instance, when a door is spotted the cell associated with the door's position in the door array is set to 1. However, when the player opens the door that same cell is set to 0 and the same position in the *walk-space* array is set to 1. This indicates that for the duration of time where the door is open, the character can move through it. Similarly, when items such as ammo are picked up, the cell associated with that position gets set to 0, indicating that the item was picked up and no longer exists on the map.

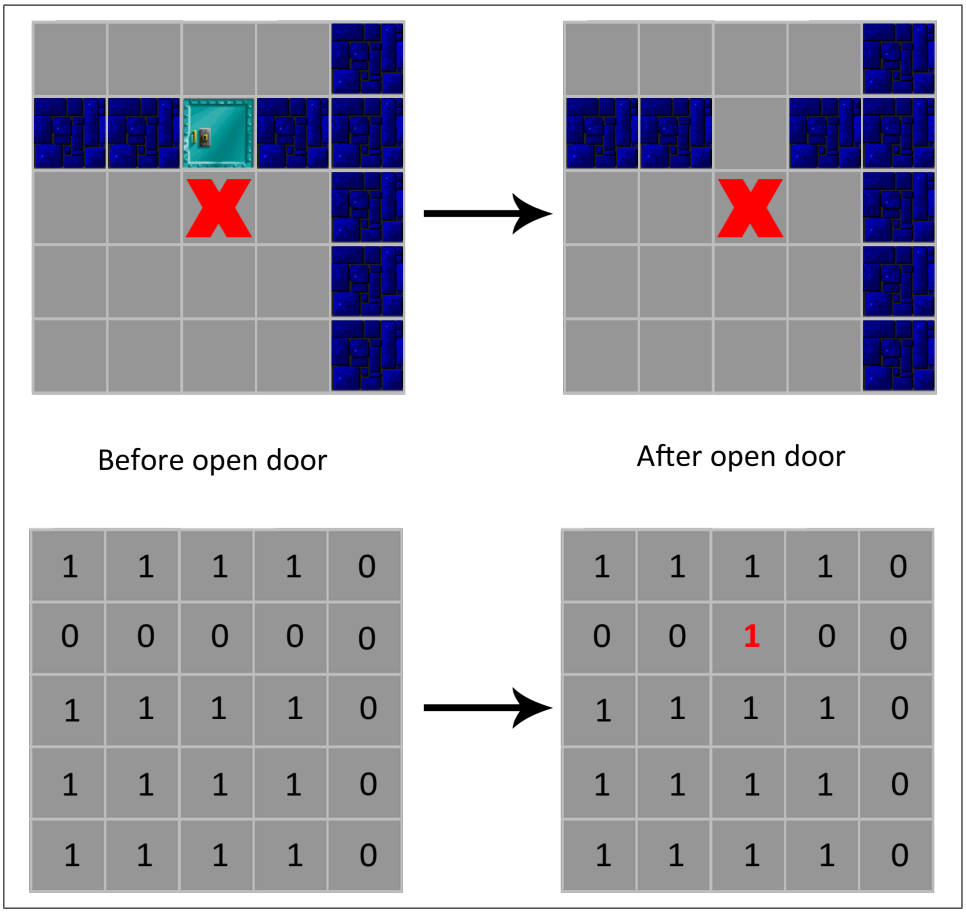

Figure 5.2: Shows the contents of the array associated with walkable space once a door is opened.

The function that was mentioned earlier, *GetInputs()*, updates the inputs once per frame. This is done by calling the function directly from Wolfenstein's *DrawScaleds()* function which, as mentioned in Section 3.2, already detects what objects surround the player's current location. The contents are analysed by the current network every five frames, which is similar to the approach taken in Seth's Mario project. This frame delay is fixed to five since increasing the frame delay by too much can cause NEATDoop to temporarily stop pressing buttons before the next set of network inputs are processed.

5.2 Playing Wolfenstein with NEATDoop

As was previously discussed, all individuals start out initially with just an input and output layer in their networks. Most of these individuals accomplish nothing in the first generation. Eventually however, one or more individuals will be found (maybe in a subsequent generation)that will have

a Neuron active that causes the forward key to be pressed. This Neuron could be taking its value from any of the eleven 5x5 grids but it is hoped that it will take its value from the walk-space grid, as that would make the most sense.

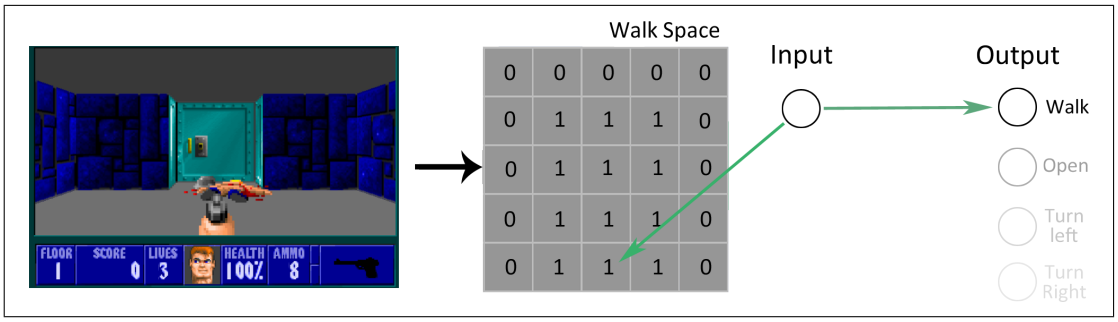

Figure 5.3: Walk-space input array at start of Wolfenstein Stage 1 level 1.

Fig 5.3 shows what the Walk-space input array looks like at the start of Stage 1 level 1 in Wolfenstein. On the right side of the array is a simple network with one input Neuron that reads from a cell in the walk-space array and is attached to the *walk* output Neuron. The input Neuron shown will always point to the same cell in the walk-space array and so long as it reads a 1 in the cell it points to, the *Walk* Neuron has to potential to activate which would cause NEATDoop to move forward.

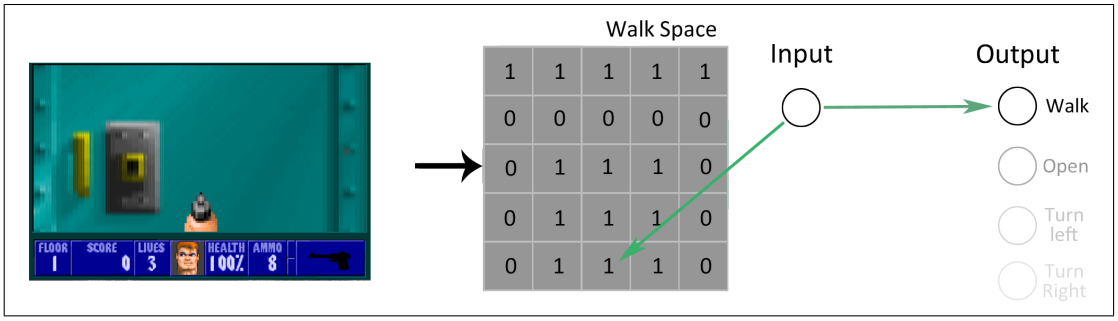

Figure 5.4: Shows the state of the walk-space array once NEATDoop has hit the door and cannot move any further.

Eventually, NEATDoop will walk into the door and not be able to move any further. Fig. 5.4 shows this scenario. In order to solve this problem, it is necessary to make the network larger, adding another Neuron that will read from another input array, the *Door* array. Fig. 5.5 shows the network solution to this problem. Once solved NEATDoop will open the door in front of it and has the potential to continue walking so long as the first Neuron keeps reading a 1.

In general, the behaviour of each network is substantially more complex than the above example but after several hours of training and is dependant on the level that is being learnt. For instance, after approximately eight hours of training on the first level of Wolfenstein, the best individual's network had about one-hundred hidden Neurons.

These examples are intended to give an important insight into how NEATDoop might play the game with a more complex network. It is also important to note that the solution provided in Fig. 5.5 is an efficient one, where the Neurons are reading from 5x5 grids that make sense. In reality a network structure can evolve to read values in unexpected ways. An example of this would be that a network that evolves to read from the Ammo pickup 5x5 grid and use 1 values in it to try to open doors.

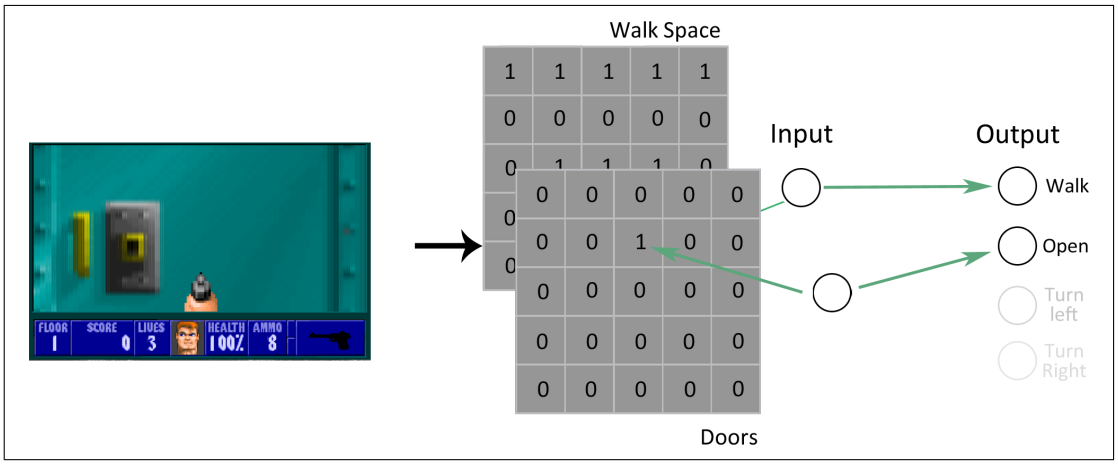

Figure 5.5: Shows the solution to the previous problem where NEATDoop could not walk any further.

5.3 Speeding Up Learning

One of the main reasons for using NEAT is its efficiency. Because every individual's network starts with a minimal topology (only has an input and output layer), all initial topologies perform just as bad as each other with no significant outliers. In TWEANNs, since initial topologies can be very complex and varied, some networks will perform much worse than others and so time is spent finding and removing these networks. However, this does not mean that the learning process for NEAT is swift. Seth's AI in Super Mario World took twenty-four hours of continuous learning to complete the first level. He programmed his AI to play the game at normal speed which considerably slows down the learning process. Since Wolfenstein's gameplay is considerably more complex than Mario's it should be obvious that in order to learn a considerable amount of play it will take longer than twenty-four hours. It was therefore deemed necessary to try and speed up Wolfenstein's gameplay to hasten the learning process.

Speeding up the gameplay involves changing a *TickBase* variable that handles the speed of enemy, player and door animations. Having experimented with this variable, it seems as though making the gameplay any more than five times normal speed causes severe time scaling problems. It appears that the time it takes for certain animations to play does not scale properly with the timing variable. This became very apparent when replaying certain individuals. In experiments done at ten times normal speed for a particular individual, NEATDoop was capable of making it through doors for instance, but when the same individual was replayed at normal speed it would get stuck in a corner beside the door.

It has been concluded that, while it is possible to speed up the gameplay and thus the learning, it is not possible to reliably replay the generated solution at normal speed, it must be replayed at the same speed it was learnt at.

5.4 Attempt Termination

The terminal condition for NEATDoop is when it completes a level in Wolfenstein. This raises an interesting question though, *How do we move to the next network if NEATDoop does nothing useful?* It turns out that it is possible to use a similar approach to how *Seth* handled this in Mario. By assigning a *timeout* to every individual it is guaranteed that eventually the

current individual will be forced to stop playing.

The *timeout* for Wolfenstein is initially set to approximately two seconds (150 frames) for every individual. If the individual's network does nothing useful in those two seconds it will time out and the next individual will be assessed. However, the problem now remains as to how we stop the individual timing out when it is in fact doing something useful.

5.4.1 Letting NEATDoop Continue to Play

In order to solve the previously mentioned problem it is necessary to adjust the *timeout* variable during the actual play. If NEATDoop's co-ordinates have changed, it has picked up an item, has opened a door or has killed an enemy then approximately two seconds (150 frames) are added to the timeout variable.

However, this actually introduces another problem. If NEATDoop learns to press the move forward button and the turn left / right button at the same time then it can cause NEATDoop to learn to move in a circle indefinitely.

The base case of this problem is easily solved. If the move forward and turn left / right buttons are active at the same time, a *circletimeout* variable is activated. If these buttons are activated for too long (approximately the time it takes to make two full circles) then the current individual times out and the individual's fitness is deducted 500 (which is substantial) to discourage this type of behaviour.

If the current map contains a more complex loop, such as a figure of 8, and an individual's network learns to run around it the above solution will not work. A possible solution to this problem is to remember what parts of the map the individual's network had already explored and time out the individual if the same place was visited too often. This was not implemented however.

The variables that allow the timeout to be modified during gameplay are kept up to date, frame to frame. The code that updates these variables was added to the relevant parts of the Wolfenstein source. The *pickups* variable, for instance, is updated in several places of the source file *wl_agent.cpp*. This is because pickups include health, ammo, keys and guns and so the variable is updated in functions *HealSelf(), GiveWeapon(), GiveKey()* and *GiveAmmo()*.

5.5 Calculating Distances

Initially, when updating the timeout in each frame a distance calculation was used to identify if NEATDoop had moved from its (x,y) position in the previous frame. Originally the *Euclidean* distance formula was used, but this proved to be too computationally expensive and so the *Manhattan distance* formula was used instead.

In the end, neither of these formulas ended up being needed to alter the timeout variable during gameplay. Instead the current and previous (x,y) co-ordinates are compared to see if there is a difference. The Manhattan distance is used however in the fitness function as will be seen later.

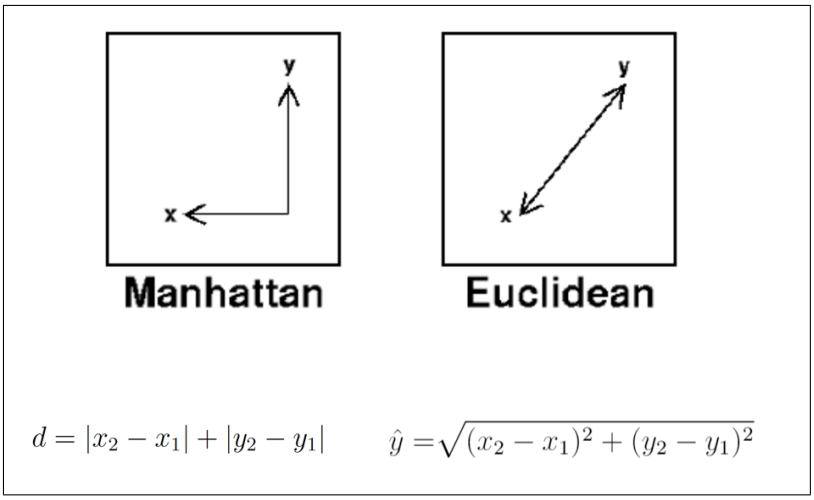

Figure 5.6: Shows the formulas used to calculate distances between two sets of (x,y) co-ordinates

5.6 Aiding NEATDoop's Learning

In order for NEATDoop to learn how to play Wolfenstein it is necessary for each network used to play the game to be assigned some sort of score that denotes the success of that network. This is done using a fitness function which rewards the network for specific tasks it completes during gameplay. This section is going to outline this function and show how it can bias learning.

The below fitness function was used to analyse networks that had timed out or were killed during gameplay. This fitness function was developed on a trial and error basis, some of the other fitness functions that were tried will be discussed in Chapter 6.

$$\begin{aligned} F = (MAX\_DISTANCE - distanceFromEnd) \cdot 10 \quad &+ \\ distanceFromSpawn^2 \cdot 10 \quad &+ \\ pickups \cdot 100 \quad &+ \\ accuracy \cdot 10 \quad &+ \\ kills \cdot 50 \quad &+ \\ inputsused \quad &+ \\ levelDoneReward \end{aligned}$$

- The constant MAX_DISTANCE is the largest Manhattan distance that can be achieved in a 64x64 grid map (the size of each Wolfenstein map).
- distanceFromEnd is the Manhattan distance between NEATDoop's position when it either timed out or died and the end of the level.
- distanceFromSpawn is a similar variable to distanceFromEnd. It is the Manhattan distance from the spawn location in the map and the position of NEATDoop when it timed out / died.
- The coefficients that were used for the fitness function were found through a trial and error process.
- The above fitness function is a simplified version of the actual code written to assign an individual's network a fitness measure. The levelDoneReward is a reward given when NEATDoop makes it to the end of a level, which it did not in any of the short training sessions done. In the actual code, unless NEATDoop reaches the end of the current level this variable is not added.

Coefficients for the fitness function were modified when they either rewarded NEATDoop too little or too much. The most desired behaviour for the game is movement, so it makes the most sense for this to be weighted highly. By making this measure a combination of the distance from the end and the distance from the start, NEATDoop hopefully learns to move away from the spawn point but in the general direction of the end of level tile.

5.6.1 Fitness Biasing

If only one distance measure was used, say for instance distance from spawn, then it would have resulted in NEATDoop learning to get as far away from the spawn as possible and not learning to try and finish the game. This is an example of fitness bias.

Earlier versions of the fitness function were particularly vulnerable to fitness biasing. One of the first networks that successfully learnt to get through several rooms in the first level was killed as a result of this. What had happened was one of the networks had learnt to pick up numerous items of ammo while another learnt to move through the several rooms. However, the network that had picked up the ammo had been given a higher fitness and so the other individual and its network were killed off and not brought forward to the next generation. This simple example demonstrates how modifications were made to the coefficients to help avoid this problem.

5.7 Saving NEATDoop Attempts

An important aspect of this project is making it possible for an individual's network to be replayed. This is considered valuable since it takes a considerable amount of time for NEATDoop to learn to do something worth replaying. For instance, it took approximately eight hours for NEATDoop to learn to exit the first room and turn left towards the level end at normal gameplay speed. In order for a particular network to be replayed a representation of its network needs to be saved and reloaded. This section outlines how this is done.

5.7.1 Saving the Network Encoding

There is a very important consequence of NEAT using direct encoding, which was stated earlier in Subsection**??**. It allows the entire network to be described in the Genes of an individual, meaning that the entire network can be rebuilt using the Genes of an individual alone.

This means that the only information about the network that needs to be remembered in order to rebuild the network for a previous attempt at playing Wolfenstein is stored in the Genes.

Fig 5.7 shows how an individual for this project is stored. The first four lines always contain the same information. All other lines are Gene encodings. All output Neurons (which are connected to buttons such as move forward, shoot etc..) have a integer value in the range $10,000 \leq x \leq 10,009$. All input Neurons have an integer value in the range $0 \leq x \leq 274$.

1. **Fitness:** The fitness of the encoded individual
2. **Global Rank:** A global rank assigned to every individual across all niches / species. A global rank of 0 indicates that this is the best fitness across the entire population for some

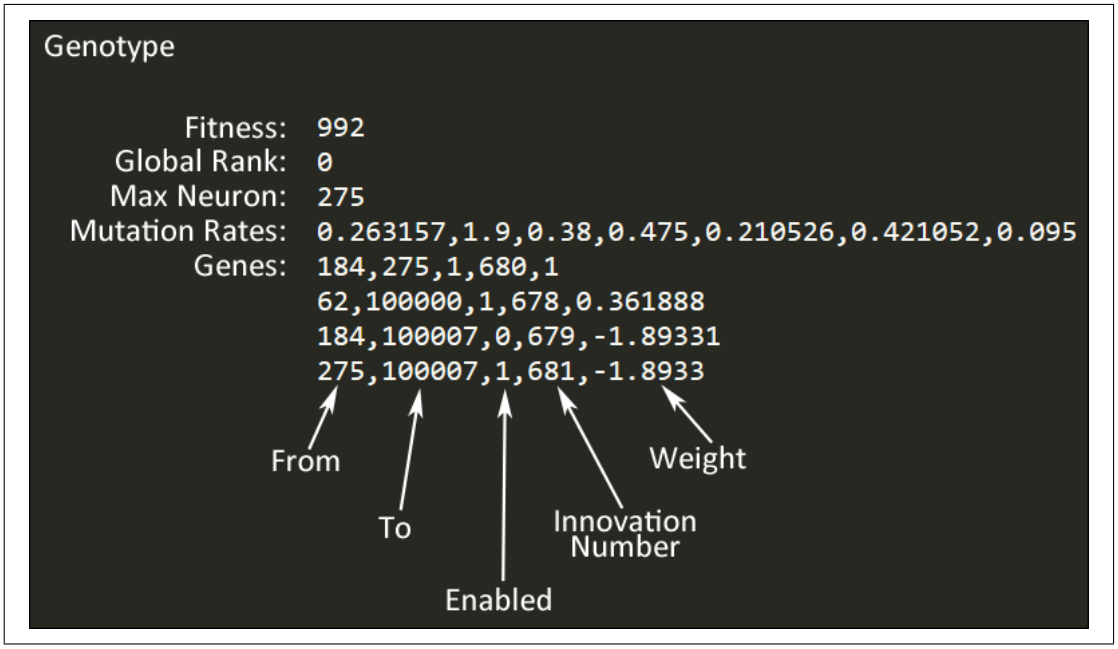

Figure 5.7: Shows a sample encoding of an individual

generation

3. **Max Neuron:** The number of Input + Hidden Neurons that there are. A Max Neuron of 274 means that there are no hidden neurons and so in Fig. 5.7 there is one hidden Neuron. The number of Hidden Neurons will always be $MaxNeuron - 274$
4. **Mutation Rates:** Every individual has a certain chance of performing certain mutations. These need to be saved with each individual encoding. A brief description for each is given below:
   (a) **Connection:** Chance to mutate the weight of a Gene.
   (b) **Link:** Chance to perform a NEAT link insertion mutation between two random Neurons in an individual's network
   (c) **Bias:** Chance to perform a biased NEAT link insertion mutation between a random Neuron and the last input Neuron, 274
   (d) **Node:** Chance to select a random Gene from an individual and perform a NEAT node insertion mutation on it.
   (e) **Enable:** Chance to enable a random disabled Gene from an individual
   (f) **Disable:** Chance to disable a random enabled Gene from an individual
   (g) **Step:** Chance to mutate the mutation rates.

   These mutations are performed by generating a random double in the range $0 \leq x \leq 1$, if the double is less than a local copy p of a mutation rate, the mutation is performed. p then becomes $p - 1$ and the process is repeated until $p \leq 0$. This is done for each of the mutation rates individually.

A problem with the current integer representations of the Neurons is that the size of the hidden layer is limited. If all hidden neurons were assigned negative integer values instead of being in the range $275 \leq x < 10,000$, then the hidden layer could grow indefinitely if needed. However, no network even came close to having 9,725 hidden neurons so the current representation is sufficient.

5.7.2 Reloading the Individual's Encoding

As it stands, reloading an individual from a file requires the file to be of the correct format. It also must contain only one individual encoding. The consequence of this is that only one individual can be replayed at a time.

Since the first four lines of the individual encoding file will always contain the same information these can be read directly into their corresponding individual variables. When it comes to reading in the Genes for the individual, it loops until the end of the file reading in as many Genes as are present in the file. From these Genes, the entire network can be perfectly rebuilt assuming the encoding is correct.

The only necessary information to be reloaded in order for an individual to be replayed are the Genes, but if it was required that learning be continued from a point in time then the extra information would be required. However, the ability to reload a particular point in a training session and continue learning was not implemented.

This Chapter gave a very in-depth discussion on the design and implementation of the *Learning to Play Wolfenstein* project. Without going into too much detail about the underlying source code, the way in which NEATDoop gets its inputs and how it plays Wolfenstein were described in great detail giving some basic examples as to how more complex behaviour might work. This chapter also outlined the distance formulae that were used as well as what fitness function ended up being used. This chapter concluded with details about how individuals were encoded and reloaded from files in order to replay the best individuals from each generation.

The next chapter will provide testing and evaluation documentation for the *Learning to Play Wolfenstein* project.

Chapter 6: **Testing/Evaluation**

Chapter 5 presented the techniques used to provide NEATDoop with the necessary functionality to learn how to play Wolfenstein. This chapter will give an analysis of the various fitness functions that were used over the course of this project as well as describing the testing that was done with NEATDoop. The chapter will conclude with an evaluation of NEATDoop. This will be aided by results that were obtained from running the software that was created as a result of combining the NEAT algorithm with Wolfenstein's source code.

The way in which NEATDoop learns heavily depends on the fitness function used to assign a score to the networks used to play Wolfenstein. Various different fitness functions were experimented with in order to find the most suitable one that allows NEATDoop to learn to play Wolfenstein in the way it is expected to. The expected way for NEATDoop to learn is to try to continually move toward the end of the level.

Over time, the fitness function used for this project was changed significantly. Originally, a fitness function identical to Seth's was used. This was just a measure of how far from the spawn NEATDoop got and how fast it got there. This was done primarily out of curiosity. It was obvious at the time that this would not be a viable fitness function to accurately measure the performance of each network since it does not take into account various aspects of the game that are essential to gameplay, such as collecting ammo and extra health. Below, *frames* is divided by seventy (since Wolfenstein plays at seventy frames per second) to get the number of seconds elapsed.

$$F = distanceFromSpawn - \frac{frames}{70}$$

The next fitness function used did take into account several important aspects of Wolfenstein's gameplay such as rewarding NEATDoop for collecting extra ammo and health. However, this particular fitness function rewarded killing enemies and collecting items too heavily causing several networks that made it through several rooms of the first level to be killed off in favour of collecting items in the game such as ammo. As you can see, this is because the coefficients used in the fitness function for pickups and kills are far too large relative to the distance travelled coefficients.

$$\begin{aligned} F = (MAX\_DISTANCE - distanceFromEnd) \cdot 10 \quad & + \\ distanceFromSpawn^2 \cdot 10 \quad & + \\ pickups \cdot 500 \quad & + \\ kills \cdot 200 \quad & + \\ levelDoneReward & \end{aligned}$$

The final fitness function developed was very similar to the above one and was shown in 5.6. The changes made were done to the coefficients of the fitness function. Two new measures were also introduced, one which awards the use of inputs in the network and the other rewarding NEATDoop for its accuracy, which is calculated by:

$$accuracy = \frac{shots\ on\ target}{shots\ taken}$$

.

6.1 Testing NEATDoop

In order to test NEATDoop's ability to learn, the program was run on several levels of the first stage of Wolfenstein for several hours. For the most part, this was done overnight. Since one of the exceptional specifications for this project was to get NEATDoop learning on half or more of the levels in the first stage of Wolfenstein, training sessions were done on six levels of the first stage, in which there are ten levels overall. None of these training sessions resulted in a level being completed but NEATDoop was still able to learn fundamental aspects of Wolfenstein gameplay, such as picking up ammo, opening doors etc.

In early development, debugging the program involved running training sessions for several hours in order to find crashes and bugs in the code. This particular process took a lot of time. In order to easily reproduce bugs when debugging, the same random seed was used so that any crashes would occur at the same point in time. This, however, could still involve waiting several hours for the bug to reappear.

All final training sessions lasted between ten and eighteen hours. The main reason for not training any longer is that NEATDoop had problems distinguishing similar parts of maps from each other and would get stuck trying to learn how to treat them differently for too long. Training sessions were terminated when it was thought that NEATDoop would not be able to get any further. Regardless of this issue, NEATDoop still shows that it is capable of learning the fundamentals of the Wolfenstein gameplay even if it cannot fully complete a level.

6.2 Evaluating NEATDoop

From the results shown in this document, it is clear that NEATDoop successfully learns to play some aspects of Wolfenstein very well. From the various training sessions that were done, it was proven that NEATDoop could learn the fundamentals of Wolfenstein's gameplay. It was capable of learning to open doors, shoot enemies, pick up ammo / health and even find secret rooms with extra treasure and weapons. Usually, after approximately ten hours NEATDoop would struggle to progress for a considerably long amount of time. All training sessions were terminated at around this time simply because NEATDoop would fail to distinguish similar sections of the map from each other. In these scenarios it would thrash by adding large amounts of new hidden Neurons and by creating new Neural connections but it would still struggle to progress.

NEATDoop's struggle to distinguish similar looking parts of the map was the main cause for many of the training session's best fitnesses plateauing for so long. This exact problem was present in Seth's Mario project. In his video he describes how his AI had lots of problems with learning to do the opposite to what it had just learnt. In Wolfenstein's first level of Stage 1, NEATDoop needs to learn to open the first door and turn left. Because it has just learnt to open doors, when it sees the second door it can, instead of turning left, walk forward and open the second door in front of it. This scenario is shown in Fig. 6.1.

6.2.1 Some Results

Since only the individuals with the highest, lowest and median fitnesses from each generation are encoded and saved to external files so that they can be replayed, a lot of NEATDoop's learnt behaviour is not seen when replaying these. From observing NEATDoop's training sessions it is

Figure 6.1: Shows the second door at the start of Stage 1 - level 1 that, if NEATDoop learns to open, causes NEATDoop's learning to plateau for a long period of time during training.

clear that movement is rewarded more than other tasks such as collecting ammo. While this is intentional it does cause more interesting behaviour (such as killing enemies) to rarely be seen when replaying individuals.

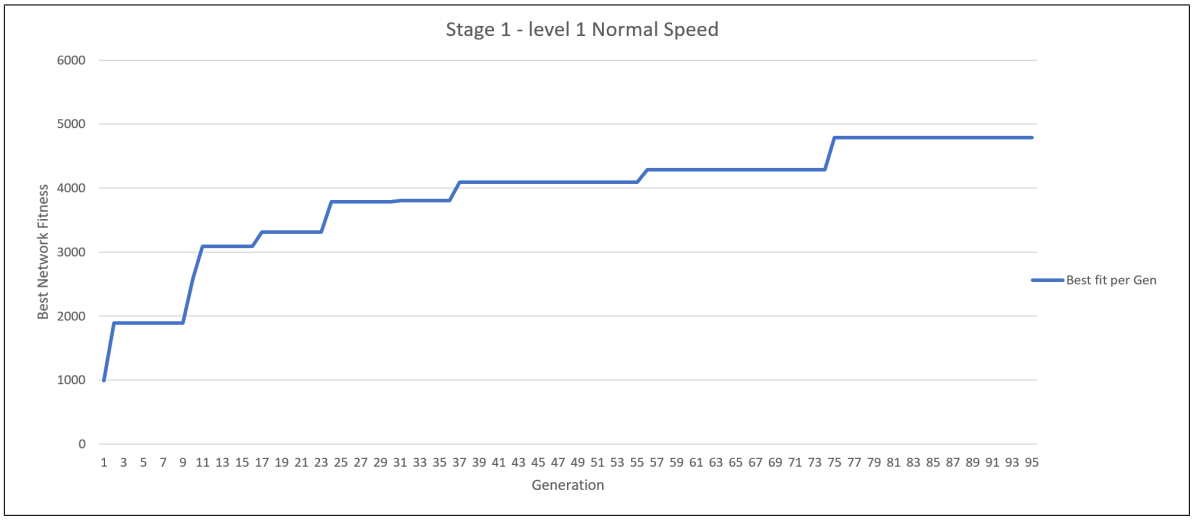

Figure 6.2: Shows the best network fitness per Generation for the first level of Stage 1 in Wolfenstein.

Fig. 6.2 shows learning behaviour that was seen across all training sessions. Generally, learning was slow with a lot of plateaus. Most of these plateaus were scenarios where NEATDoop had to learn to treat similar parts of the level differently. While it was able to solve some of these scenarios and progress it would eventually get stuck and not be able to get any further. This particular example shows the best networks from a training session done on the first level of Wolfenstein. This particular session was run over the course of about thirteen hours and was terminated because NEATDoop had not learnt a significant amount in a very long period of time. While the graph shows an increase in fitness, this was nothing significant in terms of Wolfenstein gameplay.

The training session that was done for level three of the first stage of Wolfenstein was the most successful, although this may be due to it being run for the longest period of time (approximately eighteen hours, see Fig. 6.3). This particular training session was an interesting one because it was actually unable to learn very much at all for the first eight or so hours. In this time period,

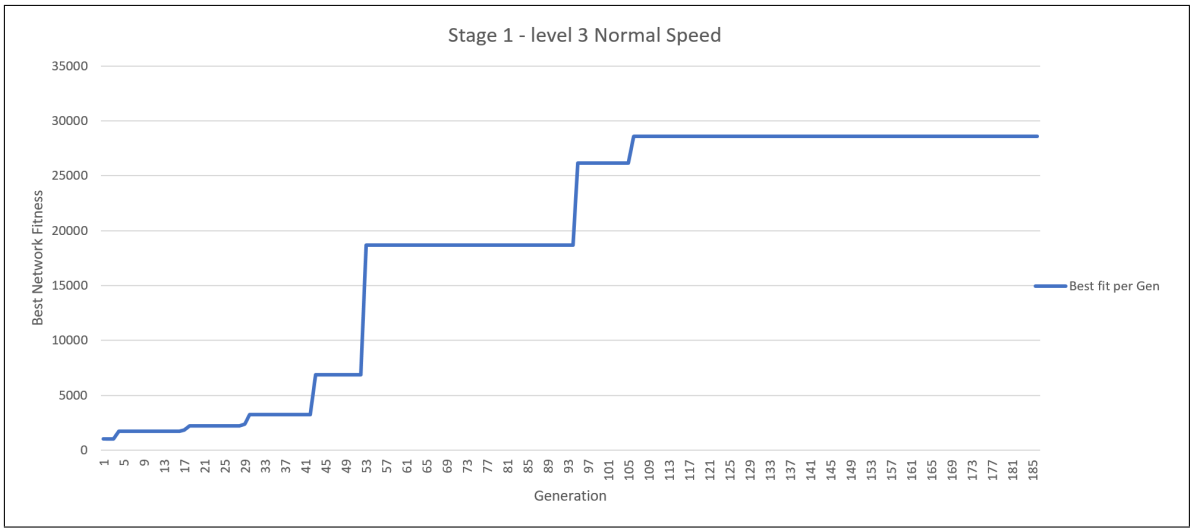

Figure 6.3: Shows the best network fitness per Generation for level three of the first Stage in Wolfenstein

it would just run straight and go through the first door it saw. Eventually, at around generation fifty it learned to turn left and made a significant amount of progress. It then had a lot of issues with making it past an enemy that is behind the door it opens when it turns left. At around generation one hundred it learnt to just run around the enemy but then ended up getting stuck for too long and so the training session was terminated.

Chapter 7: **Conclusions and Future Work**

The previous chapter explained the results that NEATDoop produced over several training sessions and described the different fitness functions that were experimented with. The chapter concluded with an evaluation of NEATDoop, presenting its ability to learn how to play Wolfenstein. This chapter will make some final remarks about the *Learning to Play Wolfenstein* project and outline some future work that could be done with it.

7.1 Extending NEATDoop

Extending NEATDoop's functionality would primarily revolve around changing the inputs to the network and the fitness function. In its current state, NEATDoop has issues with distinguishing similar parts of the map from each other. This is something that could potentially be fixed by using NEATDoop's previous movements on the map as extra inputs to the network. This is something that was conceived too close to the end of the project deadline to implement but it could result in significantly better learning as NEATDoop would have a means to distinguish its current surroundings from similar surroundings that were seen previously.

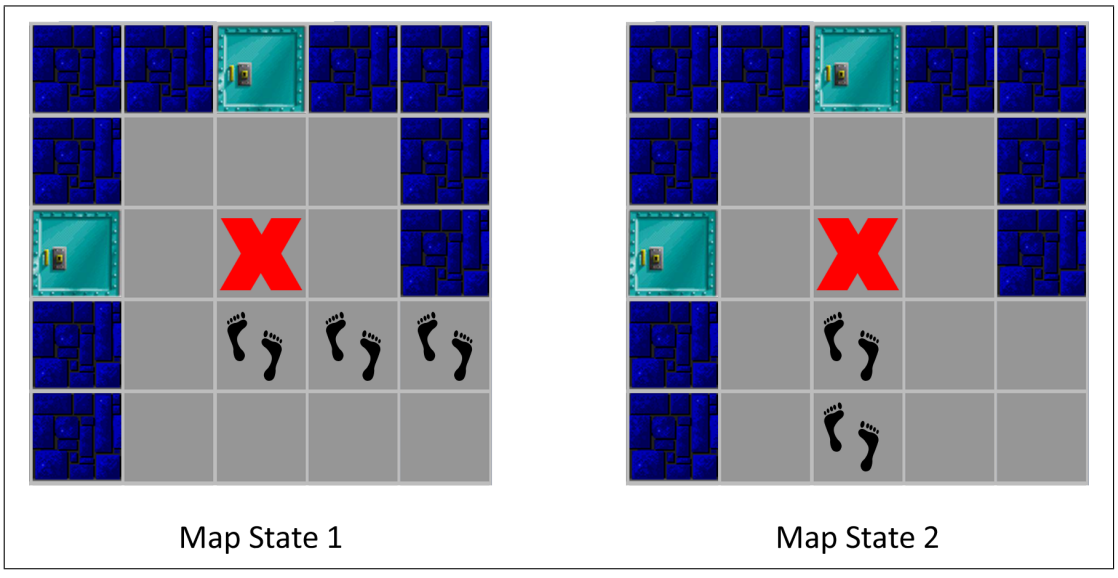

Figure 7.1: Shows how two similar map states might be distinguished from each other.

This could be implemented by determining what directions (North, East, South, West) NEATDoop had moved in from the previous N frames and by then using the data as extra input to the networks. The set of directions would then serve as an almost unique stamp on the current map inputs that are being seen by NEATDoop. The thought process here is that if NEATDoop comes to a portion of the map similar to one it has seen before, as shown in Fig. 7.1, then it will be able to hopefully make a different decision to the one it made before because it got there in a different way. These extra inputs could also be used to solve the complex circular path problem discussed in Subsection 5.4.1. There still exists the chance that NEATDoop reaches two similar looking parts of a map in the same way so these directional inputs may still not uniquely differentiate them.

The fitness function is another component of this project that could be improved. As it stands, the fitness awarded for the distance from the end of the map is very basic, this could be altered to better reward NEATDoop for getting closer to the end of the level. The coefficients used for assigning fitness based on how much ammo was picked up etc. could also be improved. There needs to be a balance in the fitness function between awarding NEATDoop enough so that movement is encouraged but also so that it enables NEATDoop to learn to pick up vital items such as keys, ammo and health.

Finally, longer training sessions could be run with NEATDoop. Even though it was found that learning progress was slow after around ten hours of training, if NEATDoop was let learn for a longer period time then it would be able to learn far more Wolfenstein gameplay.

7.2 Final Conclusion

The *Learning to Play Wolfenstein* project was inspired by a video that was seen over a year ago where a popular YouTube content creator, Seth, successfully created an AI capable of playing Mario using a machine learning algorithm called *NeuroEvolution of Augmenting Topologies (NEAT)*.

The goal for this project was to do similar, but for a more complex game called *Wolfenstein 3-D*. The AI created in this project, named NEATDoop, was successful in learning the fundamental skills required to play some parts of various levels within Wolfenstein but struggled with learning to distinguish similar parts of maps from each other, resulting in the AI's learning progress getting stuck for long periods of time. This exact same behaviour was seen in Seth's Mario AI but it was able to solve this problem both because these scenarios happened much less frequently but also because the gameplay in Mario is much simpler to learn.

Overall, this project was successful in creating an AI capable of learning to play parts of various levels within Wolfenstein 3-D, although NEATDoop was unable to learn to actually finish any of these levels. This was primarily due to the complex gameplay of Wolfenstein 3-D and also the necessity for very long training periods in order for NEATDoop to learn a substantial amount about a particular level.

References

[1] Kenneth Stanley & Risto Miikkulainen, *'NeuroEvolution of Augmenting Topologies'*, Evolutionary Computation, vol. 10, no. 2, 2002 , 30 pp.

[2] Fabien Sanglard, *'Duke Nukem 3D Code Review'*, [web blog], 13 February 2013, http://fabiensanglard.net/duke3d/index.php, (accessed July 2016)

[3] Gary Mac Elhinney, *'Duke Nukem3D project'*, [email to Fabien Sanglard], 10 April 2016.

[4] Fabien Sanglard, *'Chocolate Duke Nukem'*, [website], https://github.com/fabiensanglard/chocolate_duke3D, 2013, (accessed June 2016)

[5] ID Software, *'Wolf3d'*, [website], https://github.com/id-Software/wolf3d, 2012, (accessed August 2016)

[6] Iona Chera, *'AutoDoom'*, [website], https://github.com/ioan-chera/AutoDoom, 2009, (accessed August 2016)

[7] Moritz Kroll, *'Wolf4SDL'*, [website], https://github.com/mozzwald/wolf4sdl, 2013, (accessed October 2016)

[8] Vittorio Maniezzo, *'Genetic Evolution of the Topology and Weight Distribution of Neural Networks'*, Transactions on Neural Networks, vol. 5, no. 1, 1994, 15 pp.

[9] B. Müller, J. Reinhardt & M.T Strickland, *'Neural Networks An Introduction'*, 1995, pp. 325

[10] Gene Sher *'DXNN:Evolving Complex Organisms in Complex Environments Using a Novel TWEANN System'*, 2011, pp. 2

[11] Dimitri van Heesch, Doxygen, [website], http://www.stack.nl/~dimitri/doxygen/index.html, (accessed November 2016)

[12] Seth / SethBling, *'MarI/O - Machine Learning For Video Games'*, https://www.youtube.com/watch?v=qv6UVOQ0F44, (accessed March 2016)

[13] Seth / SethBling, *NEATEvolve*, [website], https://pastebin.com/ZZmSNaHX, 2015, (accessed March 2016)

[14] Alex. A, *'NEATFlappyBird'*, [website], https://github.com/NeatMonster/NEATFlappyBird, 2015, (accessed December 2016)

[15] Gregory M. Saunders, Peter J. Angeline & Jordan B. Pollack, *'Structural and Behavioural Evolution of Recurrent Networks'*, 1994, 8 pp.

[16] Gene Sher, *'Discover & eXplore Neural Network (DXNN) Platform, a Modular TWEANN.'*, 2010, p. 2.