Evolving Agents using NEAT to Achieve Human-like Play in FPS Games

Zach Laster

Helsinki May 6, 2014

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Zach Laster

Työn nimi — Arbetets titel — Title

Evolving Agents using NEAT to Achieve Human-like Play in FPS Games

Oppiaine — Läroämne — Subject
Artificial Intelligence

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	May 6, 2014	69 pages + 0 appendices

Tiivistelmä — Referat — Abstract

Artificial agents are commonly used in games to simulate human opponents. This allows players to enjoy games without requiring them to play online or with other players locally. Basic approaches tend to suffer from being unable to adapt strategies and often perform tasks in ways very few human players could ever achieve. This detracts from the immersion or realism of the gameplay. In order to achieve more human-like play more advanced approaches are employed in order to either adapt to the player's ability level or to cause the agent to play more like a human player can or would.

Utilizing Artificial Neural Networks (ANNs) evolved using the NeuroEvolution of Augmenting Topologies (NEAT) methodology, we attempt to produce agents to play a First-Person Shooter (FPS)-style game. The goal is to see if the approach produces well-playing agents with potentially human-like behaviors. We provide a large number of sensors and motors to the neural networks of a small population learning through co-evolution.

Ultimately we find that the approach has limitations and is generally too slow for practical application, but holds promise for future developments. Many extensions are presented which could improve the results and reduce training times. The agents learned to perform some basic tasks at a very rough level of skill, but were not competitive at even a beginner level.

ACM Computing Classification System (CCS):
Computing methodologies: Neural networks
Computing methodologies: Continuous space search
Applied computing: Computer games

Avainsanat — Nyckelord — Keywords
game, artificial intelligence, NEAT

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Artificial Intelligence (AI) has been used in games for decades. Ranging from the very simple behaviors of the Pac-Man ghosts to the artificial life simulations of the Creatures series, AI in games allows us to create opponents, allies, pets, and game mechanics which help to engage the player. These applications have at best produced interesting gameplay for players, but often fall short of their desired results.

Most games where antagonistic play exists include some form of artificial opponents, often called bots or agents, which take on the roles of other players. However, in practice, players tend to prefer playing against other human players. Generally, this is due to dissatisfaction with current artificial players [Spronck et al., 2004].

Many games rely on simple scripts in order to control their artificial agents. While this is a cheap method of producing artificial intelligence, it lacks any capacity for learning. Often, players will discover failures in the AI, such as predictable behaviors under specific circumstances, which they can take advantage of. While the goals of the game often encourage taking advantage of these failings, a lot of the enjoyment is lost, resulting in a less entertaining experience overall.

In order to improve the player's experience and enjoyment, we seek to create artificial agents which exhibit human-like behaviors. It is believed that human-like agents which behave like living things are both more entertaining and engaging to the player, thereby improving the player's enjoyment in the game [Spronck et al., 2004].

We shall apply state-of-the-art technologies to creating agents in a popular genre of computer game, the First-Person Shooter (FPS). We will produce genetically evolved neural networks using NeuroEvolution of Augmenting Topologies (NEAT) which will be used to control artificial agents. These agents will then be evaluated to assess if the system can produce agents which play well and also exhibit human-like behaviors. The goal of the thesis is to evaluate the agents produced by NEAT to see if they are well-playing or human-like; ideally, they would be both. We further limit this by assessing if it is practical to use this architecture in a realistic environment, such as commercial game development.

First we will present some background information on AI in games, focusing on game balance, the connection to player immersion, and how artificial agents directly affect both of these things. Then we will cover the necessary information for creating artificial intelligence using genetic algorithms and neural networks, building into using NEAT. Next we will present FPS games as a game genre and research framework.

We will also cover some related research to creating agents for FPS-style games. After having introduced all of the literature, we will present the project of the thesis and its implementation, including a precise definition of the project question and its analysis. Finally the results will be presented and analyzed. The paper concludes with a repetition of the core results and points and by presenting a number of avenues to move forward.

2 AI and machine learning in games

Machine learning has been applied to games for decades, initially targeting boardgames. The first application of machine learning to games was in the late 50's, when Samuel [Samuel, 1959] applied a method similar to *temporal difference learning* [Sutton, 1988] to the game of checkers. Due to the success of this approach, other common boardgames have been targeted such as tic-tac-toe, chess, and Go. Go remains a very popular boardgame for AI research, as no solution to the game is known except for on small boards [van der Werf, 2005].

Machine learning in games can be broken into two categories: *online* and *offline learning*. Online learning implies that the system adapts and learns while the game is running, typically to adapt to the player in some way. Offline learning occurs outside of the game, generally during the development of the game. Online methods are required to be fast enough to operate within the game context without slowing the game down and should reliably produce good results [Bakkes et al., 2009]. Offline learning does benefit from both of these properties, primarily because they are useful in reducing the time and effort required during game development. The results of offline learning can easily be adjusted by a designer to improve gameplay.

Commercial video games do not often see large amounts of machine learning applied to them [Woodcock et al., 2000]. Applying machine learning techniques can take control of the agents away from designers and developers and give it to the game itself [Hunicke and Chapman, 2004], especially if online learning is used. This means that the development of agents using online learning is uncontrolled and often difficult to predict, possibly even completely unpredictable. It is hard to say if an agent will learn correct or useful behaviors, as well as difficult to prevent them from learning useless or problematic ones. Thus, while machine learning techniques are occasionally applied in an offline environment to aid in development and design of agents, online learning is still not used frequently in commercial games.

There are also complexity issues with applying machine learning to games. Most games have very complex and large state spaces, involving lots of information and possible actions. This makes it difficult to apply machine learning techniques to them. Furthermore, agents need to adapt very quickly. Rapid and reliable adaptation of agents is a frequently encountered issue in AI for games research [Bakkes et al., 2009].

Instead of applying complex, difficult, and unpredictable machine learning methods to game AI, most commercial games utilize some form of scripted behaviors [Arrabales et al., 2009, Tozour, 2002]. This allows designers to control exactly how an agent will behave in a given set of circumstances. Unfortunately, this leaves these scripted agents as very static actors, lacking any ability to adapt or learn. Their behaviors are often easy to predict and may exhibit poor responses or leave exploits available to the player. In modern, highly realistic games, people often expect the agents to behave like real people, so when they fall short of this the players are often disappointed. If the agents could learn, using machine learning techniques, then they could be much more convincing [Fogel et al., 2004].

Even though it is not prevalent, video games continue to see more applications of machine learning. While most applications are limited to research interests, some commercial games have used machine learning techniques in their mechanics. Examples of such games include the Creatures series (Creature Labs) and the Black and White series (Lionhead Studios).

The first game of the Black and White series, simply titled Black and White, was highly successful and praised for its use of machine learning [Johnson and Wiles, 2001]. Black and White is a strategy god game where the player is cast as a divine being with a group of followers to grow and nurture as the player sees fit. The machine learning aspect of the game revolves around the creature which the player selects at the beginning of the game to be their companion and to carry out their wishes [Johnson and Wiles, 2001]. The developers of Black and White wished for the creature to learn and behave in a natural way, so they applied a belief-desire-intention architecture to them, which is a form of the standard artificial-life model [Johnson and Wiles, 2001].

This application of machine learning greatly improved the enjoyment and interest in the game for many players. Notably, it didn't do so in a way that most AI affects gameplay, which is to increase or decrease difficulty. Instead, it was used as a mechanic within the game.

2.1 Game balance and player enjoyment

In order to apply machine learning to games, we must understand the basics of how and why artificial agents are used in computer games. We should particularly consider the impact of artificial agents on gameplay and what the goals of applying them are.

One of the primary aspects which we can affect through game AI is game balance. Game balance is important to player enjoyment [Hunicke and Chapman, 2004]. If a game is too easy then it quickly becomes boring; if there is no challenge then the player lacks a feeling of achievement. However, if the game is too hard then it quickly gets frustrating, causing players to stop playing or otherwise react negatively towards the game. In either case, the fun of the game is lost, and players will usually stop playing the game in favor of something else. This means that games need to strike a balance in how difficult the game is throughout, making sure the player never gets too bored or frustrated.

There is an ideal area where the player is having the most fun playing the game, called the *flowchannel* [Hunicke and Chapman, 2004]. Essentially, this is a proportional relationship between game difficulty and player skill. Outside of this area, or pathway, players will generally enjoy the game less. While a player lacks skill or abilities in playing a game, their ideal challenge is low. As a player's skill level rises, they require a higher level of challenge.

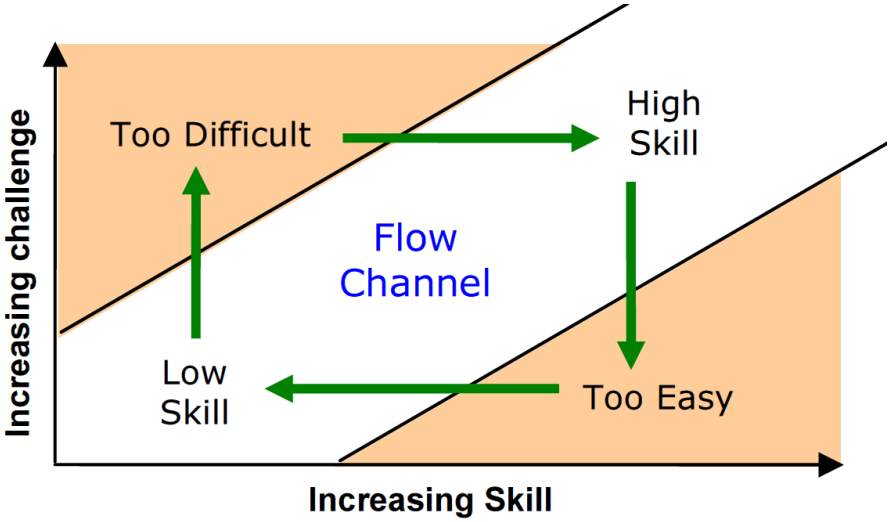


Figure 1: The *flowchannel*, in which the difficulty of the game best matches the player's skill level. [Hunicke and Chapman, 2004]

One concept presented by the flowchannel is that not all players are at the same skill level when playing the game. Some players will be highly experienced in the game's style of play, whereas others may have never played any digital games before. Additionally, a player's ability changes over time; while a player engages in the game they learn and improve the skills required to play the game. Conversely, if a player goes for a long time without playing, their skills may degrade.

In order to attempt to satisfy both expert and beginner players, games will often have a difficulty setting. This allows both novices and experts to enjoy the game by better matching their own skill level. However, this setting is rarely fine grained, often resulting in players being caught between two ill-fitting difficulty settings. In this case a player may be forced to choose a difficulty setting which is slightly too difficult or slightly too easy. Furthermore, players do not know their own best setting when first playing, and they may quickly regret their selection once in-game. This poses an annoyance factor in many games, as sometimes the difficulty cannot be changed once the game is started. For some games this may even require redoing a lengthy and tedious setup procedure. Lastly, difficulty settings do not necessarily scale against a particular play-style and they do not generally affect AI tactics [Bakkes et al., 2009], meaning that the setting will not improve the game experience for many players.

2.2 Dynamic difficulty and artificial players

In order to better manage game difficulty, machine learning can be applied to make the challenge scale dynamically or to have the agents adapt to the skill of the player [Spronck et al., 2004]. Additionally machine learning can be applied to make agents more interesting [Fogel et al., 2004]. Being able to maintain a consistent challenge and appearance for agents can positively affect the player's immersion. There is significant research applying machine learning to games in order to achieve dynamic and adaptive difficulty scaling or agent learning, especially in an online context [Bakkes et al., 2009].

2.2.1 Dynamic difficulty

Dynamic Difficulty Scaling (DDS) can be used to produce a more *even game*; a game balanced against the currently active player [Spronck et al., 2004]. This means the difficulty of the game is adaptive, and will therefore adjust over time in order to keep

pace with the skill of the player. Since the difficulty is adjusted to the player at an individual level, players are kept more interested and immersed [Andrade et al., 2005].

DDS is an online mechanic. This means that it is utilized during gameplay. This is often achieved by adjusting the difficulty between individual sessions or rounds of the game, though it is readily possible to adapt during a session.

There are two primary methods for DDS: *environmental manipulation* and *adaptive AI*. Environmental manipulation involves tweaking values or adding or removing entities from the game environment [Hunicke and Chapman, 2004]. This can be used to manipulate the player's inventory or resources indirectly by controlling what is available in the world to pick up. Adaptive AIs instead modify the behaviors of the existing entities to match the player's ability level, making the AI play better or worse. This means that the player is always playing against an opponent of reasonably even difficulty or skill.

An example of a DDS system is Hamlet [Hunicke and Chapman, 2004], a system written for Half-Life. Hamlet uses environmental manipulation to either aid or further challenge a player, depending on how well the player is performing. If the player is doing badly then the game can aid the player by increasing the number or effectiveness of health packs, making enemies drop more ammo, or reducing the number of enemies in the map. Alternatively, it could decrease the number of enemies fought at once, distributing the load on the player across more areas. Obviously, if the player is doing too well for the desired difficulty then Hamlet can do the reverse; make enemies stronger, decrease the number of dropped items, add enemies, or move enemies around so they attack in higher concentrations.

A method to achieve adaptive AI is to utilize *dynamic scripting*. Dynamic scripting was originally designed to create strong playing AIs [Spronck et al., 2004]. It is an unsupervised and online method for learning AIs that is "fast, effective, robust, and efficient" [Spronck et al., 2004]. It operates by storing a collection of weighted rules which are predefined by experts. In order to generate an opponent using this method a set of rules is selected from the collection based on the weight of each rule.

In order to use dynamic scripting to create Adaptive AIs, the rule selection must be changed to select rules such that the result plays at the player's level. Specifically, the resultant AI must avoid being too difficult. Three methods for adapting dynamic scripting to player skill level have been presented and compared [Spronck et al., 2004], top culling, high-fitness penalizing, and weight clipping, with top culling

being found to be the most effective.

It should be noted that sometimes players actually want a super easy or an extremely difficult challenge, with no adaptive scaling. Allowing the player to select a base difficulty setting can be used to constrain the adaptations or adjust the targets that the adaptation is attempting to reach [Spronck et al., 2004]. This allows the player to indicate how they wish to be challenged. Lastly, allowing players to turn off adaptive play enables them to try out new ideas or playstyles without affecting the difficulty they are used to playing at while allowing them to set the game to a much easier setting.

2.2.2 Artificial players

As a part of general game balance, artificial players affect the player enjoyment. The challenge presented to the player by artificial opponents is critical to game balance and game difficulty. If such an agent is too difficult then it will easily outmatch a human opponent. Conversely, if the agent is too stupid or predictable then the human will easily overcome any challenges it presents. Because of the impact on game balance and difficulty, the performance and style of artificial agents is important to player enjoyment and immersion.

We can apply machine learning to artificial agents in order to produce agents which better match player skill levels, particularly through the use of online learning. In the context of game AI, online learning means that the agent learns as the game is played, often changing in real-time or between play sessions in order to adapt to the player. Offline learning occurs outside of play, meaning that the agents do not adapt or change while the game is played. While not directly useful for adaptive agents, offline learning can be used in conjunction with online learning [Andrade et al., 2005] in order to produce effective agents before the game is played. These agents can then be improved during gameplay using online learning. This allows for agents to be well-performing when the player first starts playing and to adapt to the player over time. Offline learning by itself can be used by designers to produce agents which play well without having to worry about the agent adapting badly once in the hands of the user [Stanley et al., 2005].

In addition to the difficulty of an agent, it is important that the agents be interesting [Andrade et al., 2005, Fogel et al., 2004]. This implies that they should not be strictly predicable and should employ tactics which at least appear to be intelligent

and human-like. This improves the immersion of the player and makes the agents into more than simple mechanical opponents. If the agents are made to act in a human-like way, the game world feels similarly more alive.

In order to achieve more human-like behavior, agents are often made to learn and adapt to changes in the environment and how other players act [Bakkes et al., 2009]. This requires that agents utilize some form of adaptive AI. In addition, the agents should not be too challenging; instead they should present a challenge appropriate to human abilities. Because of this, many approaches attempt to adapt agents to perform at the level of the current player or players [Spronck et al., 2004].

However, simply providing an even challenge does not make agents seem more human-like [Bakkes et al., 2009]. In the simplest case, agents adapted to a very poor player may make numerous and unrealistic mistakes, such as endangering themselves or attempting to force the player to make good moves. In the opposite case, agents adapted to very good players may simply react very quickly or perform tasks with inhuman accuracy and timing, rather than developing more advanced strategies.

The same is true for the reverse case; making agents human-like certainly does not make them well-performing. It is possible for agents to exhibit human- or animal-like behaviors and not perform the target task specifically because of the behaviors. Behaviors such as running away or avoiding the task, as well as behaviors which cause the agent to fail the task, would sometimes appear very human-like. However, the agent is obviously not performing the task well in these cases.

Another factor in which game AI affects player enjoyment is whether or not the AI cheats [Laird and VanLent, 2001]. In many games, especially traditionally, AI agents have been made to cheat in order to gain an edge on the player. Examples of cheating include the AI knowing things or being able to perform tasks which the player is not able to do. In most cases, such as in the game Starcraft, this was to enable the AI to play at a level which would challenge the player. While some simple cheating enables the AI to provide a more even game, it also does not feel like a fair opponent, frustrating players who become aware of the cheating.

An example of an architecture for producing adaptive agents was created by Bakkes et al. using a case-based learning [Bakkes et al., 2009]. They applied this architecture to a real-time strategy game called Spring and achieved good results. The architecture was shown to adapt reliably and quickly, and to be usable in an online environment. Most impressively, it is capable of learning immediately, without review, allowing it to adapt within a single play session.

Traish et al. applied NeuroEvolution of Augmenting Topologies (NEAT) to real-time strategy games [Traish and Tulip, Sept], producing a collection of agents capable of playing Wargus. In their experiments, they found that NEAT was very proficient at producing effective strategies. However, it required some particular constraints in order to force NEAT to produce more complex strategies, rather than simply optimizing a single strategy.

2.3 The illusion of intelligence

One frequently observed phenomenon is that the illusion of intelligence is indistinguishable from actual intelligence [Buckland, 2005]. The goal of game AI is simply to create that illusion. As long as the illusion is maintained, the player will believe that the agent is actually intelligent, but once the illusion slips it is difficult to reclaim player immersion [Andrade et al., 2005, Buckland, 2005].

The developers of Halo (Bungie, 2001) invested a great deal of time and effort into balancing the AI of the enemies and allies. They generally found that an even challenge comes across as more intelligent [Champandard, 2007]. There is a direct relationship between agent difficulty and apparent intelligence: increasing one increases the other. They also found that agents can be made more human-like by giving them reactive behaviors, such as running away or charging when angered. It is also noted that biological-like sensor models improves believability.

One of the elements which really contributed to how human-like the Halo AI felt was the reactions of the agents [Champandard, 2007]. The enemies were given a breaking point behavior, which is initiated upon a certain set of conditions. For example, a Grunt, a small, fairly easy enemy, will run away screaming if they witness the player kill an Elite, one of the harder enemies. This presented a very convincing show of the Grunt being terrified.

The Elites also exhibited a breaking point behavior. If an Elite was reduced to not having any shields and low health, they would go berserk. This behavior consisted of the Elite roaring, switching to a sword-like weapon, and charging the player, as pictured in Figure 2. This was not a smart behavior, as the roar was done while stationary and triggered by being near death. Often, the Elite would die before they got through the entire animation. However, in terms of human-like behavior, it was extremely convincing and entertaining for the Elite to scream defiance at the player.

Originally, these breaking point behaviors did not happen every time [Champandard,

Figure 2: An Elite from the game Halo, which has reached its breaking point. It howls defiance at the player in an amazingly human-like display.

2007]; the agents would only break some percentage of the time. However, it was found that players did not always notice the behaviors or may not correctly attribute the behavior to its cause. Because of this, majority of the agents were changed to execute the behavior every time the conditions occurred. While some players would still miss the connection, this greatly improved the human-like appearance of the enemies.

Half-Life (Valve, 1998) used an interesting approach to give their opponents the impression of being a well trained, cooperative fighting force [Laursen and Nielsen, 2005]. They forced the number of active combatants to a maximum of two, cycling out the attackers so they they seemed to work together. This meant that, out of a large group on opponents, only two opponents would every be truly active in the combat at a time, either moving or firing, and everyone else would be hiding or preparing to fire. Once an enemy became ready to attack, one of the active attackers would hide behind cover and the now ready enemy would become an active attacker. This effectively produces the illusion of an intelligent group tactic which is highly coordinated.

Hingston [Hingston, 2009] created the BotPrize [Hingston, 2013] competition in 2009 as a testing environment for developing human-like bots in *UnrealTournament2004* (UT2004). The contest is designed like a Turing test, originally proposed in 1950 by Alan Turing [Turing, 1950]. In the contest, humans and bots are put against each other randomly such that the combatants do not know who is human. Each contestant evaluates the *humanness* of all the other contestants, indicating whether

the contestant thinks each opponent is human or not, resulting in a humanness score for each player. Both human and artificial players get a score. The bots also judge the other contestants, but these results are not used when calculating the averages. In order to win the BotPrize, an AI must achieve a rating of at least 50% human.

In 2012 the BotPrize contest was won by two entrants. Mihai Polceanu produced the MirrorBot [Polceanu, 2013], which emulated human behavior via online imitation. The second place winner, UT^2, was produced by a team [Schrum et al., 2011, Schrum et al., 2012, Karpov et al., 2012]. They received humanness scores of 52.2% and 51.9% respectively. As UT^2 employed neural-based learning, we shall discuss it more later.

3 Building AI for gameplay

In order to understand how NEAT functions, we need to look at how the underlying systems is uses work and have been applied to games individually. NEAT utilizes neural networks which it evolves using genetic algorithms. Genetic algorithms are in turn a form of search-based learning.

3.1 Search-based learning

For many learning tasks we may not have a proper training set or ideal solution. This poses challenges to training-based machine learning, such as traditional neural networks or supervised machine learning techniques [Harman and Jones, 2001]. In such cases it is more feasible to search from all possible solutions to find the best possible solution for the task [Harman, 2007, Togelius et al., 2011]. An example of such a problem is to find the best actions to perform in the current circumstances when playing a game or the best recognizer of a particular image. *Search-based learning* is a process of building solutions and evaluating their relative *fitness*, meaning how well they solve a given problem. Generally, this approach can be used to find good solutions, though it does not always find the best solution.

The solutions produced through search based learning are not simple answers, but models for producing answers from input information. Rather than find the right answer for a particular problem, such as the sum of two numbers, the target solution is a model for producing the right answer to a summation problem. This means that once a good solution is found it can be reused in other forms of the same problem.

It is possible for search-based algorithms to be trapped in a local optimum, where the current solutions are good and no immediate variations seem better [Montana and Davis, 1989, Priesterjahn et al., 2006]. This can cause search-based algorithms to fail to find potentially better solutions in a different part of the search-space.

3.1.1 Representation

In order to identify or build these solutions, it is necessary to be able to represent them in some fashion that allows us to map out the *search-space* [Harman, 2007]. The search-space is the set of all possible solutions. Ideally, every solution is unique in its representation and all solutions are possible to represent [Togelius et al., 2011]. While these are not technically strict requirements, it is generally wasteful to have multiple representations for a solution, and any solutions without a representation are either missed or must be evaluated separately.

The exact method of representation for a solution depends on the structure of the solutions themselves and how the search will be carried out. The representation can either be the solution itself or provide all of the necessary information to build the solution [Togelius et al., 2011]. Typically, building the solution itself is accomplished through some form of mapping from representation to solution.

3.1.2 Evaluation

The second challenge to search-based approaches is determining a method of evaluating fitness [Harman and Jones, 2001]. The fitness of a solution should indicate how well and consistently the solution solves instances of the problem. In order to measure the fitness of a solution, we build a *fitness function* which evaluates each solution's performance or output and produces a fitness value. We can then use these fitness scores to compare solutions to each other.

In some cases, a solution may be evaluated many times. This occurs most often when the environment changes between solutions, or when a test cannot cover every possible scenario [Togelius et al., 2011]. When this happens, the fitness score of a solution is often some form of composite of the values produced by the fitness function, generally a form of averaging.

The fitness function needs to reflect, for example, if near correct answers are valuable, as well as if a single completely wrong answer amongst otherwise perfect answers is tolerable [Harman and Jones, 2001]. Measuring fitness is simplified if the correct

answer for an instance of the problem is known, though simply being able to recognize a correct or good answer is sufficient. Note that it is often possible to recognize a correct answer to a problem even without an algorithm to produce that answer in advance.

3.1.3 Search

The number of possible solutions in the search-space can be extremely large, sometimes even infinitely large, implying that an exhaustive search is infeasible. In order to limit the number of solutions assessed, we use a search algorithm which selects solutions to evaluate based on the current best solutions [Harman and Jones, 2001]. Typically this is accomplished by moving from well-performing solutions to solutions which are "nearby" in the search-space. This produces a controlled movement through the search-space, limiting the search to only solutions which are likely to perform well. Furthermore, we may only want to move from solutions which are well-performing, to see if the nearby solutions are better.

In order to move through the search-space, we need a starting point. In practice, we tend to generate a random *population* of solutions [Harman, 2007]. Ideally, this population is distributed throughout the search space. We then evaluate this population and perform search from the best performing ones, building a new, similarly sized population.

One drawback to using search-based learning is that it is possible to miss good solutions in other areas of the search-space. This often occurs if the given population falls into some form of local maxima, where the current population has no neighbors which perform better, though there are still better performing solutions elsewhere in the space [Harman, 2007, Togelius et al., 2011]. Basic search algorithms can prevent us from discovering solutions which perform even better on the other side of these neighbors. Despite this, a good solution is typically found even if it is not the best solution.

Some example strategies to handling search are hill climbing, simulated annealing, and genetic algorithms [Harman, 2007]. Hill climbing uses a random starting population and evaluates the fitness of each solution and their neighbors in the search-space. For each solution, we then move to its neighbor which most improves fitness, if it has one. This method is particularly susceptible to getting trapped in a locally optimal solution, as it cannot climb "down" from a peak fitness value.

Simulated annealing differs from the hill climbing approach in that it is allowed to move to less fit solutions. It uses a probabilistic function to decide if it can move to a less fit solution, rather than moving to a more fit one. This function decreases over time, resulting in a greater exploration of the search-space initially and an eventual transition to a more pure hill climbing approach.

We shall focus on how genetic algorithms perform search and how they represent solutions.

3.2 Genetic algorithms

Genetic algorithms are a subset of search-based algorithms which utilize evolutionary algorithms to accomplish search [Harman, 2007]. The approach uses genetic representations of solutions which are evolved over many iterations in order to produce well-performing solutions. This implies that the search is limited; rather than evaluating all possible solutions, solutions similar to ill-performing ones are ignored. This makes evolutionary algorithms useful when the search-space if very large.

3.2.1 Representation of solutions

The representation of solutions in genetic algorithms are handled in an abstract way that is converted to the working model, roughly reflecting the concepts of *genotypes* and *phenotypes* from biological genetics [Togelius et al., 2011]. The actual genetic representation is a genotype of the solution, while the solution itself is a phenotype. It is possible for a genetic representation of a solution to directly equate to the solution, meaning the representation effectively is the solution. More commonly, however, the genetic representation is used to construct the solution, in a form of *genotype to phenotype mapping*. All genetic representations can be seen as using a genotype to phenotype mapping, even if the mapping is to itself.

The mappings from genotype to phenotype and be either indirect or direct [Togelius et al., 2011]. A direct mapping is one with a fairly simple system for converting the genotype to the phenotype. Often the elements in the genotype map directly to elements in the phenotype. In an indirect mapping, the mechanics for producing the phenotype can be much more complex, often utilizing some form of language. Gruau et al. presented a cellular-based system for handling indirect mappings [Gruau et al., 1996] for producing neural networks from genetic sequences, which was demonstrated to work well. However, it has been shown that direct encodings can be just as ef-

fective in the same context [Gomez and Miikkulainen, 1999] and it has been stated that such indirect encodings require more detailed knowledge of how neural and genetic mechanisms function in order to develop [Braun and Weisbrod, 1993]. Stanley et al. argues that this might bias the search in unpredictable ways [Stanley and Miikkulainen, 2002].

3.2.2 Genetic evolution

The strength of genetic algorithms is to be able to search only such parts of the solution space that seem promising. This allows for good solutions to be found more quickly compared to more basic search approaches. While it is still possible for genetic algorithms to be caught in locally optimal solutions, they tend to be less affected by them [Montana and Davis, 1989]. However, a sufficiently large local optima may trap them [Priesterjahn et al., 2006].

As a search-based approach, the first task of genetic algorithms is to produce a starting population of solutions. The exact nature of the starting population varies depending on the representation and applications, though typically the initial population is a randomly generated or selected collection of solution representations. We then evaluate the fitness of each solution, enabling us to rank the solutions relatively. From this we can generate a new population.

When creating a new population, we must decide if we wish to create an entirely new population or if we wish to keep individuals of the current population. Some approaches retain parts of previous populations when generating a new one, a process called *elitist reinsertion* or elitism [Cole et al., 2004, Eshelman, 1990, Westra, 2007]. In this model, the best performing of either the previous population or all previous populations are retained when a new population is generated. This is often used in cases where a small population is in use, so that newly developed features or behaviors are not lost. At this point, the worst solutions may be removed from the population or kept for their genetic diversity.

The simpler model is to generate an entirely new population, discarding the previous population and only keeping the offspring. It is also possible to occasionally add randomly generated or expert-crafted solutions to the current population, increasing diversity and possibly avoiding getting trapped in local optima.

In the simplest case, a genetic representation is just an array of values. Each cell of the array holds a gene which can only be selected from a fixed set of values. We can

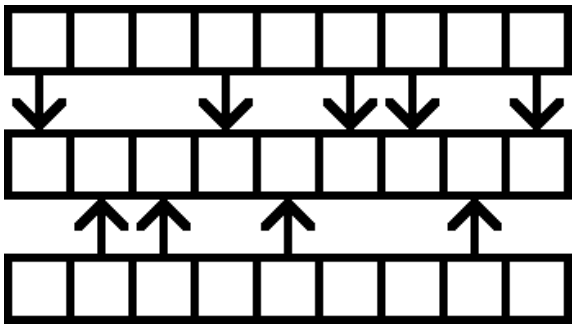


Figure 3: Two gene sequences are combined to form a child. The child's genes are each randomly selected from one of its parents.

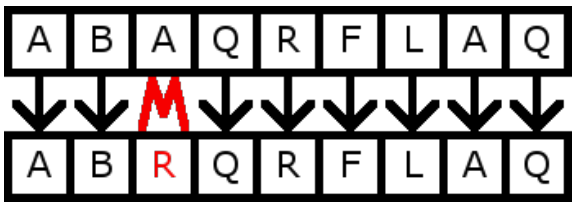


Figure 4: A single sequence is mutated to produce a new sequence. Here the genes are represented by letters.

combine these sequences in a process called *crossover* to create new offspring based on the previous population. When performing crossover on two or more genetic parents, we simply randomly choose which parent provides each gene in the array, as demonstrated by Figure 3. In practice, crossover is performed using two parents.

When producing offspring, it is also possible to alter the genes randomly using *mutation* [Fogel et al., 1990]. Mutation is simply altering the genes in a representation slightly as shown in Figure 4. This produces new solutions which are slightly different from existing ones. Mutation can be applied in addition to crossover or by itself, allowing for the possibility of completely excluding any form of crossover from the algorithm. A simple approach is to apply mutation for each offspring after the crossover step. The rate of mutation can be adjusted to better suit a given problem, though it is typically kept at some low value or degrades to a low value from an initially high one. This facilitates random search, without causing offspring to lose too much from the solution they are based on.

3.2.3 The permutation problem

With some representations it is possible that the offspring of two well-performing solutions will be damaged and perform badly [Montana and Davis, 1989, Schaffer et al., 1992, Radcliffe, 1993]. This can occur when the well-performing solutions model similar solutions in very different ways. This results in the offspring not representing either solution, possibly having what is effectively the same half of a solution twice. This problem is referred to as the *permutation problem* [Stanley and Miikkulainen, 2002, Radcliffe, 1993]. It can be visualized as a representation having three genes and a genome of three elements, A, B, and C. A representation with one of each gene will perform well, but those lacking one of the genes will have a lower fitness. In this example, the two solutions ABC and CBA are well-performing, but their offspring could be ABA, which would not perform as well as its parents.

Handling the permutation problem is often challenging, and it must be either solved or avoided [Stanley and Miikkulainen, 2002]. It is possible to design the solution representation to avoid the permutation problem, though this is typically difficult. Alternatively, we can attempt to separate solutions in such a way as to prevent differing solutions from producing offspring, such as preventing ABC and CBA from combining. How to accomplish this is far from trivial, however.

There is some debate as to whether or not even biology solves or avoids the permutation problem [Stanley and Miikkulainen, 2002]. Arguably, avoiding the problem is a solution, in that the problem does not arise. Therefore, there is not a strong difference between the two, so accomplishing either method is acceptable.

3.2.4 Genetic algorithms in games

Genetic algorithms are frequently applied to game AI in academic research as the evolutionary component of search [Agogino et al., 1999]. However, genetic algorithms can be used for more than just learning in games. The Galactic Arms Race game [Stanley, 2007] used compositional pattern producing networks (CPPNs) and a variant of NEAT to procedurally generate weapons in a multiplayer game environment. A CPPN is a network of function nodes which produces outputs by running input values through those functions. Each weapon had a CPPN which was used to describe the movement and color of the particles fired from the weapon. The number and power of the shots was fixed across all weapons.

Figure 5: Examples of procedurally generated weapons from GAR [Stanley, 2007].

3.3 Artificial neural networks

Artificial Neural Networks (ANNs) are an architecture of artificial intelligence commonly used as recognizers [Lippmann, 1987]. They take a collection of input signals, calculate the activity of the network, and produce a collection of output signals. Once a network is trained, it can be used as a black-box system to perform multi-valued calculations; ANNs are a form of function approximation for functions which handle many inputs and outputs.

3.3.1 Structure

The basic element of a network is the *neuron* [Lippmann, 1987]. Much like biological neurons, neurons in ANNs produce a signal based on their inputs. Artificial neurons have weighted input edges, or connections, leading from other neurons. The weighting of these edges allows the neurons to prioritize some inputs over others, or to treat some input signals as inhibitors. Neurons then perform a basic arithmetic operation upon these weighted signals and emit a signal along any edges leading from them.

A neuron is capable of handling multiple inputs, each with their own weighting. A neuron will then produce a single signal which is then weighted individually via the connections to other neurons. Exactly what signal a neuron produces and how it calculates that signal are up to the implementation of the neurons. In some cases, the produced signal may be on or off, or it might be a floating point value.

Figure 6 shows the structure of a neuron and the basic approach for handling inputs and output. Each neuron is comprised of three parts: summation, a signal function, and the output [Lippmann, 1987]. The neuron receives inputs from other neurons, each multiplied by its respective weight. All of the inputs are summed together to produce a single value. Finally the neuron runs this weighted sum through a function to determine if and at what strength the neuron fires. The output signal

is carried to any neurons with connectors from this neuron.

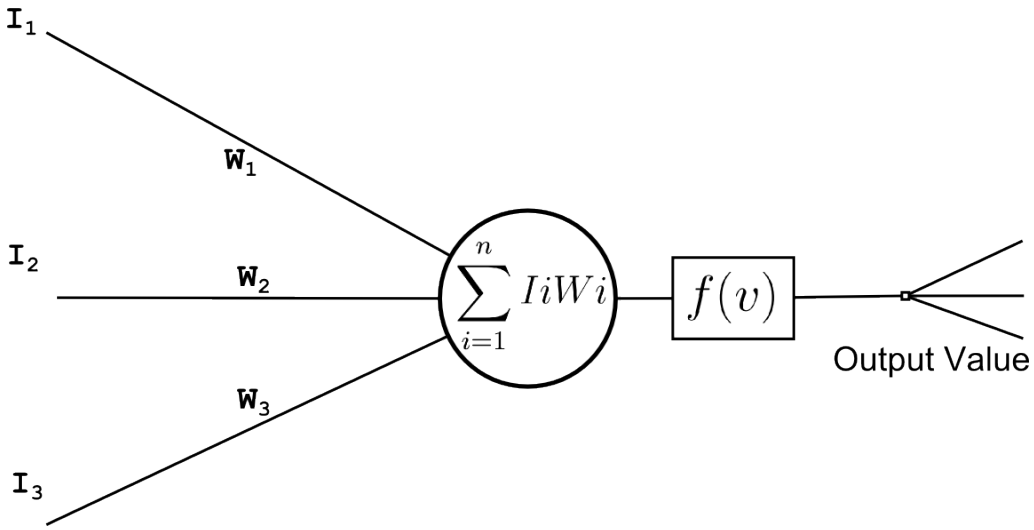


Figure 6: A basic neuron.

There are many functions which can be applied to a neuron's summed input in order to produce an output value [Lippmann, 1987]. Hard limiters, linear gradient functions, and sigmoidal functions are fairly common. The function applied to produce a neuron's signal tends to be identical across the network; all of the neurons use the same function. However, it is possible to create networks with varying functions. Compositional pattern producing networks [Stanley, 2007], which are highly similar to neural networks, use this mechanic.

In addition to weighted input signals, neurons often have a bias value used in their calculations [Lippmann, 1987]. This value is commonly shared with all neurons, but can be made unique to each neuron as well, allowing more flexibility in the network at the cost of increased complexity. This value can be used as a threshold, limit, offset, or bias, depending on the function used in the neuron itself.

Since ANNs are a form of function approximation, adjusting the network can change which function they represent. This can be done readily in two ways, either by altering the weights of the connections or by changing the structure of the network [Lippmann, 1987]. Neurons with individual biases can have these biases altered along with their connection weights. Thus neural networks can be used to better approximate desired functions by iteratively adjusting the network to better match values produced by those functions. This is how learning, or training, is achieved using ANNs.

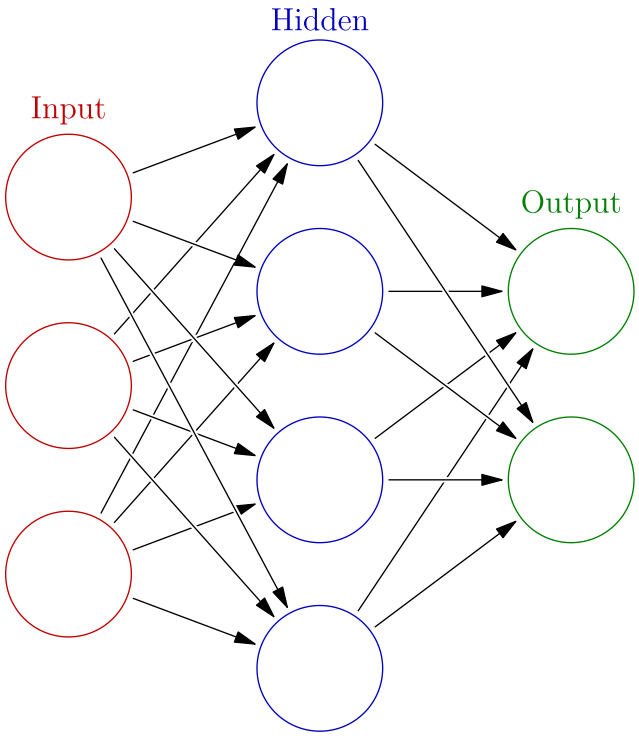


Figure 7: A standard neural network with a single hidden layer of four neurons. The network takes three input values and produces two output values [Glosser, 2013].

3.3.2 Forms of neural networks

In arguably the simplest form of network, traditionally a form of *Hopfield Net*, the incoming signals are simply true or false and only basic inversion weights are applied [Lippmann, 1987]. In such a network, a neuron typically uses some form of hard limiter or threshold function to process its inputs, producing only an 'on' or 'off' signal.

Perhaps the most commonly used form of an ANN consists of a layer of *input neurons* and a layer of *output neurons* connected via some number of layers of internal *hidden neurons*. In this architecture, traditionally called a *multi-layer perceptron* [Lippmann, 1987], every neuron in one layer receives inputs from every neuron in the previous layer. The input neurons provide externally provided signals along their connections to the hidden neurons. The hidden neurons then perform transformations on these signals, possibly utilizing many layers of hidden neurons, and send weighted signals to the output neurons. The output neurons perform one last operation upon their inputs, at which point the output neurons contain the output signals for the whole network.

These more complex networks typically use floating point values for produced signals in addition to floating value weights. In these cases, the weights are usually bounded in the range of $[-1, 1]$, though they do not technically need to be bounded at all. Networks that use floating point values will often have some form of function for the neurons to perform on their inputs, such as a Sigmoid or Gaussian function. Such non-linear functions are generally more flexible than simple summation with neuron-specific biases, as a series of linear functions can always be simplified to a single linear function. This means that multiple hidden layers do not meaningfully increase the logic powers of the network in a linear function network, but can do so in a network utilizing a non-linear output function.

3.3.3 Applications

One of the most typical and referenced uses of ANNs is for recognition, particularly image recognition [Lippmann, 1987]. This involves training the network to recognize whether an image matches other images, such as recognizing characters or faces. Once a network has been trained it can be used to predict if an image or character matches a target element. This is useful in security or general image matching tasks.

ANNs can be applied to machine learning for games in many ways. They have been applied as army controllers for strategy games [Traish and Tulip, Sept], as well as allowing more direct lower-level control, such as in Creatures or Black and White. Generally, neural networks can be used as controllers for individual agents or as higher-level controllers for teams of agents.

4 NeuroEvolution of neural networks

Utilizing genetic algorithms over neural networks allows us to generate neural networks to solve tasks. However, first we must be able to represent an ANN so that we can map the search-space. We must also handle the permutation problem. When doing these things, it is important to consider what changes in the network when evolving it. The simplest approach is for the network's topology to be fixed, and the weights between the neurons to change. However, it is more powerful for the topology of the network to be evolved as well, even though this is much more complex.

NEAT [Stanley and Miikkulainen, 2002] evolves the topology of the network, rather than simply evolving the weights. This allows it to start from a minimal topology and

produce minimal topologies for networks, keeping the networks faster and simpler than what would be possible if only the weights were changed.

The NeuroEvolution part of NEAT simply refers to evolving Neural Networks. Usually this is achieved, as in NEAT, by applying genetics. Augmenting topologies is obviously in reference to the fact that the topology of the networks is changed.

Augmenting topology has some difficulties that must be addressed. Typical problems facing evolving ANNs are the difficulty of representation and the permutation problem. When evolving structure, the representation gets even more challenging. Since the network is not fixed, we require a method of describing connections between neurons and the existence of neurons. The permutation problem becomes more significant as well, producing scenarios of duplicated connections or mismatched networks. In order to overcome these issues, NEAT uses a genetic encoding of connections, historical markings on the genes, and speciation.

4.1 Representation through genetics

NEAT encodes the connections between the neurons genetically. Each *connection gene* indicates the neurons to connect, the weighting for the connection, and includes an expression bit to allow genes to be turned off. Each connection gene is also marked with an *innovation number*, which indicates when the gene was added to the genome. The genetic encoding also contains a list of *node genes*, which express the neurons in the network which can be connected.

Mutation in NEAT can generate new connections as well as new topology in the network. In order to add a new neuron to the topology, an existing connection is split, adding the new neuron in the middle of it. The original connection is disabled, and two new connection genes are added to describe the new connections. To better handle the change in weighting, the connection weight to the new node is set to 1, while the connection from that neuron receives the original weighting. Adding a connection is simpler; a connection gene is created which connects two previously unconnected neurons together with a random weight.

4.2 Historical markings

Because NEAT allows the genomes to grow without any limitations, genomes of various lengths will result, meaning the gene sequences of solutions can be very

different lengths. Furthermore, some of the genes may be identical to others in different locations, producing the same connection and a different gene. This takes the permutation problem to an extreme, meaning that two very different genomes often cannot perform crossover without producing highly damaged offspring. NEAT solves this using the historical markings on the genes.

The historical markings on genes allows NEAT to handle crossover between genomes of differing lengths. When the gene is first created it is given a globally unique numerical identifier which functions as the historical indicator of the origin of the gene. The identifier is taken from a global counter, which is then incremented. This means that any two genomes with similar ancestors can correctly align their genomes, placing later genes further out and matching identical genes correctly. NEAT also handles identical genes being created within the same generation such that they get the same historical marking identifier, so as to not duplicate genes.

In order to perform crossover between two gene sequences of differing lengths, NEAT emulates the natural process of synapsis. Synapsis is the process of aligning homologous genes, or genes which function on the same trait, for crossover. This means that genes do not get accidentally randomly inserted and that the resultant sequence does not have multiple genes for the same trait. NEAT performs *artificial synapsis* by comparing the historical markings of the genes and aligning genes with the same historical markings. Once this is done, crossover can function normally, selecting genes from each parent sequence randomly and taking the disjoint and excess genes from the more fit parent. If the parents have equal fitness, then the excess and disjoint genes are inherited randomly.

4.3 Speciation

Typically, as new topology is added, the fitness of a solution drops slightly. This makes it difficult to evolve more complex topologies, as the solution might be removed or ignored in future evolution. Therefore it is important to protect innovation as new topology is formed. NEAT separates the genomes into species, a process called *speciation*, to protect innovation. This allows the species to compete within their own *niche* rather than against the global population, meaning the offshoots have a chance to optimize.

In order to determine the species of a network, NEAT actually looks at the gene sequence. By using artificial synapsis as in crossover, it is possible to determine the

genetic distance between two sequences, using the number of disjoint, excess, and slightly altered genes. NEAT uses the calculation

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 * \bar{W}$$

to determine the genetic distance between two sequences. Here $\bar{W}$ is the average weight differences between matching genes, and E and D are the numbers of excess and disjoin genes respectively. explain terms. The coefficients allow the formula to be tuned to control which factors are most important and N, which is the number of genes in the largest genome, helps to normalize the values. A fixed threshold δ_t is compared against the resultant value to determine if two sequences are in the same species. NEAT then keeps an ordered list of all the species, placing a sequence in the first species it matches.

At first glance, sorting ANNs into species seems it would be a topology matching problem. However, the historical markings make matching by genetics fairly trivial. Furthermore, matching by topology would produce similar errors as matching biological species by appearance does. This would result in incorrectly matching two networks which actually have very different historical backgrounds and genetic makeup. This would actually trigger the permutation problem, rather than avoid it.

In order to keep the species from growing too large, and thus crowding out other species, each organism in a species is forced to share fitness with its entire species. This is called *explicit fitness sharing*. Each organism is given an *adjusted fitness score* which factors in the fitness of the species, using the genetic distance between two organisms to determine species. This simplifies to only considering the organisms withing the same species when reproducing. Species then remove their lowest performing members and the remainder reproduce. Once the offspring are produced the existing population is discarded, leaving only the offspring of the well-performing members of each species.

5 AI research in FPS games

When testing basic, comparatively simple artificial agents and architectures, First-Person Shooter (FPS) games are frequently used due to their comparatively simple and straight forward interactions as well as fairly monotonous state information. The goals within the game are very simple; to survive and to defeat other opponents.

5.1 FPS games

FPS games get their name from the rendering perspective involved in the game. The player is presented with a 3-dimensional view of the game world as if they were looking through the eyes of the character they control. Players move around the environment, eliminating opponents and attempting to survive for as long as they are able. While some games of the genre have storylines or more complex objectives, this simple arena-style combat model is the baseline and heart of the genre.

Since the game is a shooter, weapons are obviously at the heart of the game. Most such games have a large variety of weapons which can be used. Players must regulate their ammunition and utilize these weapons to defeat their opponents. While most weapons are generally some form of gun or launcher, for example a rifle, machine gun, or the ever popular rocket launcher, shooter games occasionally have some close range weapons as well, such as Doom's chainsaw.

The environments in which the players move, simply referred to as maps, are closed environments where the different combatants are *spawned* at various locations. Spawning is a term in many games for the creation or resurrection of an object, such as the player's avatar. Various objects which can be collected, called *pickups*, are scattered around the map, encouraging traffic in certain areas with powerful or popular pickups. Pickups commonly include weapons, ammo, and various power ups such as temporary invisibility or additional damage.

Depending on the game mode, the players may be on teams or they may fight in a Free-For-All (FFA) fashion. In team combat, points are shared with the entire team, though individuals within the team may compete for best score. In FFA-style play everyone has their own score. The game usually ends when either a certain score is reached or the timer runs out.

During the game, each combatant moves around the map seeking better weapons, strategic control points, and enemies to *frag*. The term frag is used in many games to mean kill, though as a softer term, since the death is usually temporary. The origin of the word is not clear, though probably refers to the abrupt fragmentation of the deceased. When a combatant dies, they are quickly respawned elsewhere on the map to resume the match. Usually, weapons and other power-ups do not carry over through death.

FPS games are very common, and there are many very popular ones. Two of the earliest popular FPS games are *Quake* (id Software, 1996) and *Doom* (id Software,

1993), both of which were developed by id Software and have developed successful series after the original games. Quake introduced a large number of the primary concepts involved in FPS games, particularly the multiplayer competitive aspects, whereas Doom was a very story-driven game. Other games expanded upon these. *Unreal* (Epic & Digital Extremes, 1998) was built following a similar style to Doom, though with a superior engine because of advances in technology. The engine of Unreal has been used and adapted over the years to produce the Unreal Tournament series, which focused on multiplayer gameplay. *Quake III Arena* (1999), commonly just referred to as Quake 3, also developed in this direction. Common games for AI research in FPS games include Quake 3, UT2004, and *Halo* (Bungie, 2001).

These arena-style combat games require multiple combatants in order to occur, meaning a human player requires opponents to play against. While the game is primarily designed for multiplayer interaction, most games of the style include some artificial agents for people to be able to play independently or with very few players. These artificial players are called *bots* in the game. Bots can usually be configured to have varying levels of difficulty, some ranging well beyond human abilities in their timing, aim, and perception. It is possible to write bots for these games through various means, allowing for interesting new bots to be created. This is useful for research purposes, particularly when attempting to evaluate new artificial intelligence approaches, especially for adaptive or general game AI.

5.2 Working environments

In order to test an agent in a FPS environment, we first need the environment itself. While it would be possible to develop a game environment specifically for testing [Stanley et al., 2005], it is far easier and more efficient to use an existing environment.

Two of the most common options are UnrealTournament and Quake. UnrealTournament includes a scripting language, UnrealScript, which allows the game to be modified. This includes the ability to write new bots for the game. Quake III is now open source and written in C++, allowing the game to be modified and bots to be controlled to a very extensive level.

When building agents for games, it is often helpful to utilize frameworks which externalize the agent from the game. This can allow the agent to be applied to multiple games with fewer changes, as well as allow the agent to play as if it were

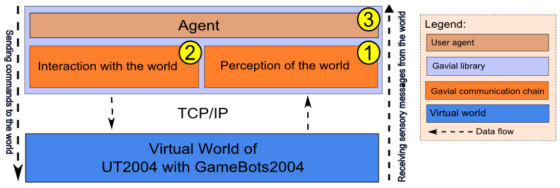


Figure 8: Pogamut communicates between the game and the agent through GameBots2004 over TCP [Gemrot et al., 2009]. The agent can perceive and interact with the world using GaviaLib.

a discrete client. This can also allow agents to be written in different languages, improving performance or enabling the use of existing code libraries.

5.2.1 Pogamut

Pogamut [Gemrot et al., 2009] is a toolkit that allows developers and researchers to write artificial agents for various games using Java. It handles the interaction and communication with the game, allowing for development to focus on the agent itself. The library also includes basic elements for sensors and pathfinding so that higher-level agents can be designed and tested without the need for implementing these lower-level logic elements. Additionally, Pogamut includes helpful tools for debugging and visualizing the execution of the agents.

Pogamut is targeted primarily at UT2004 and utilizes GameBots2004 [Gemrot et al., 2009], which is a version of GameBots designed to communicate information from UT2004. GameBots2004 provides the information from the running game server over TCP/IP, allowing bots to connect without needing to fully run an instance of the game themselves. Pogamut then uses the GaviaLib library [Gemrot et al., 2009] to connect to the environment provided by the GameBots2004 server, which handles bot perception and interaction with the world.

Pogamut includes a number of additional systems, such as a planning controller and an emotional state simulation, which have been developed externally. These are integrated into Pogamut to allow for agents to be developed using them with relative ease.

It is also possible to animate the controlled character through Pogamut. This could be utilized to make the agents seem more believable and human-like. These animations do not affect gameplay, however, so they are only useful in self expression. This kind of functionality could be used to produce behaviors similar to the breaking

behaviors of Halo.

5.3 NeuroEvolving Robotic Operatives

NeuroEvolving Robotic Operatives (NERO) [Stanley et al., 2005] is a *machine learning game*. This means that at least some of the gameplay is centered on applying some form of machine learning in order to train agents in-game. In NERO, the player is attempting to evolve a squad of agents to compete with other similar squads.

The game of NERO occurs in a simplified FPS, environment. The game has many of the same elements of an FPS if considered from the perspective of the bots, but lacks the more advanced terrain and navigation challenges such as lifts, grappling points, or vertical space.

The evolution system applied in NERO is a real-time version of NEAT which allows for the agents to be improved as the game progresses. It accomplishes this by replacing the weakest intelligences with offspring of the strongest at various intervals. The easiest timing for this is when an operative is killed, though it is possible for an agent to be replaced while it is still alive.

Agents are trained by creating goal conditions and behavior rewards and then evolving the populations until they succeed in these goals. Once the agents have learned to perform their tasks or to behave in a desired way, their networks are frozen and individuals from the population can be selected to be added to a team. This team is then put against other teams in a competitive multiplayer environment. This makes NERO something of a meta-, or two-level game. At the basic level it has evolved agents playing the shooter game, and at a higher level it has human players building teams of agents to pit against others.

The networks of NERO agents have very direct control over the actions of the agents, being able to instruct the agent to move in specific directions, rotate, and to fire. It does not use predefined activities such as “fire upon nearest enemy” or “move to flag”. This allows for more advanced behaviors to be developed than would be possible using selectable scripted behaviors.

The sensor system used in NERO is robust. It has four types of sensors, enemy sensors, object range finders, line-of-fire sensors, and an on-target sensor. The enemy sensors detect enemies in arced areas of the agent’s view, activating in strength proportional to the proximity of enemies in that area. The object range finders utilize rays to detect terrain or other obstacles. The line-of-fire sensors use rays

fired from each enemy and determine the intersections on several rays fired from the agent, allowing the agent to know how close they are to any enemy's line-of-fire in various directions. Finally, the on-target indicates whether the agent is currently looking directly at an enemy. Similar sensors could be added for detecting other information, such as friendly-fire lines and proximity to important items.

5.4 Producing neural network-based agents

Westra [Westra, 2007] has applied evolutionary neural networks to FPS games. In his tests, Quake III was used as the platform for testing the agents. While NEAT was not used, evolutionary algorithms were applied to train the networks. The networks used fixed topology and evolved the weights between the neurons.

There were some limitations to the maximum population size in Westra's tests. The tests used an evolving population of only 6 agents, implying it is possible to evolve reasonable solutions even with very small populations. To counter the size of the population, elitist reinsertion was used to retain the top performing agents.

Westra's ANNs used less direct control over the agents than NERO's agents, instead having the network choose a destination in the existing navigation map. However, firing was under direct control.

The agents produced generally performed better than the native bots, though occasionally exhibited strange behaviors such as staring at walls. Some agents succeeded in learning to seek ammo or health when needed, but no agents learned to seek both.

Gamez et al. produced a NeuroBot system [Gamez et al., 2011] which played UT2004. NeuroBot competed in the BotPrize competition in 2011, scoring very well and coming in second, suggesting that a neurally-based agent may be capable of human-like behavior. NeuroBot uses a form of sensory salience to determine the selected actions of the agents. In some ways, this is a fusion between direct control of the agent and controller systems. Rather than selecting specific actions, the NeuroBot determines how and if it wishes to move, particularly moving in certain patterns. In some cases certain systems may operate without input from the network, however; for example the jumping module seems to become more active if an enemy is near.

The UT^2 architecture uses NeuroEvolution in order to train an agent [Schrum et al., 2011, Schrum et al., 2012]. It trains using pre-existing human traces in order to mimic human behaviors. This allowed the agent to seem so human-like it won

the BotPrize in 2012.

UT^2 does not use direct control over the agent, unlike NERO. Instead, the architecture has a collection of priority-ordered modules which dictate the behaviors the agent exhibits.

6 Using NeuroEvolutionary agents in an FPS game

Through NEAT we can create neural network-based agents which can play games. What we wish to find out is if this can produce human-like agents which are interesting to play against. While we are not using a truly online mechanic, once fully trained the system could be used as an adaptive architecture. Such an agent would theoretically be able to keep the player well within the flowchannel for most of play.

We can build a system to train agents to play UT2004 using Pogamut and NEAT-based agents. Much of the work on the system presented here will be on the sensors and motors required to make it function.

The work here steps away from the existing work, either in complexity or application. It's important to look at those differences to see what can be used and what kinds of issues we may encounter.

6.1 Project hypothesis

The project was designed to answer the question *"Can NEAT be applied to FPS-style games in order to generate well-playing agents with human-like behaviors?"*. In this section we will assess this question and compare it to other research. To do this, we will look at what the question means, whether or not it has been previously answered, and what makes it a worthwhile question.

It is important to understand the intent of the question when trying to answer it, otherwise an answer becomes difficult to interpret. One important note at the very beginning is that when we ask if it can we done, we really want to know if it is practical to do. If it is possible but would take over a year of training time, then it is far from practical. This raises the important point that this is not a question with a binary answer. A simple yes or no leaves out extremely critical information, such as how long the training takes, or if it produces other interesting results. Thus it would be reasonable to understand the question as "What is the result of applying

NEAT to FPS-style games?". The question is phrased as it is in order to focus on the performance and possibility of human-like behaviors, as that is the desired information.

Well-playing is used here to mean an agent which can perform at a reasonable level of performance. We do not care for an even game, so much as we do for the bot to be able to accomplish basic game tasks. This means that a bot is well-playing if it can perform the goals of the game. This means that both an extremely difficult bot and a very easy bot could both be well-playing, despite being imbalanced opponents for most players; a difficult opponent is likely to be unfairly built, relying on superhuman abilities, and an easy bot opponent may not hold up well against an experienced player. Human-like refers to any behaviors which would either come from being controlled by a human player or behaviors which would be appropriate if the avatar of the bot were alive. This accepts a fairly general definition of human-like in this context.

It was decided to use NEAT for this project because of its success in other areas. It has been shown to be effective in RTS style games and in FPS-like games. We apply NEAT to a complete FPS environment in order to see if it can perform well in a real game environment. While NERO showed that NEAT is capable of functioning in a simplified FPS-like game, we wish to see how it will fare in a larger, more demanding environment.

A real FPS was used, rather than an environment only slightly more complex than that of NERO, primarily because stepping up the complexity iteratively is a detailed and difficult process. It would take significant work to recreate a system which would be able to range between NERO's level and that of a real FPS, and most of those steps would not reveal anything of interest. Additionally, iteratively increasing complexity in increments is more useful when determining where a system breaks down, rather than if it works at a more complex level.

Ultimately, the goal of the project is to experiment with using this architecture to develop interesting opponents to play against. We seek to evaluate if this architecture is feasible for producing agents which will improve the immersion of the player. As we have established, player immersion is a key element in the enjoyment in the game; thus it is something which both players and producers wish to maximize. Ideally, the architecture would provide interesting bots. However, establishing ways to modify the architecture for future research is also valuable.

Westra [Westra, 2007] produced a very similar system to the one presented here,

with some key differences. Westra actively decides against using NEAT, instead focusing on standard neural networks. Westra expresses an opinion that NEAT would not add anything to the work, expecting that normal methods would serve equally well. Additionally, Westra uses some predefined behaviors which the network selects from, rather than giving direct control to the agent. This impairs the agents ability to produce more novel behaviors, as it is limited in how it can move and behave. However, it reduces the complexity that the network needs to manage, making it much simpler for agents to learn to play well.

NERO [Stanley et al., 2005] is far more similar to the work presented here, using NEAT to directly control agents which traverse a landscape and attempt to destroy enemies. The primary differentiating factor between the bots in NERO and the bots presented here is their level of complexity. Looking at NERO as a FPS-style game, it is extremely simplified; it has an apparently flat landscape with only simple obstructions, no limits on ammo, and only one weapon. However, NERO does use direct control over the agent, rather than scripted behaviors. NERO shows that directly controlled bots can be well-performing and interesting in this simplified environment, but that does not necessarily extend to the more complex environment of a real FPS game.

The NeuroBot produced by Gamez et al.. [Gamez et al., 2011] used salience-based controllers to determine what behaviors were desired by the neural-based controller. Rather than give the network direct control, the network produces signals which motivate certain actions. The controllers themselves then workout how to translate this into activity. This is not a purely neural-driven system, as other components can motivate actions, such as direct responses to enemy presence. This means that the network is only part of the overall control of the agent. We seek to evaluate direct control agents.

UT^2 [Schrum et al., 2011, Schrum et al., 2012] was built using aspects and concepts from NEAT, but did not utilize it directly. It also had a number of underlying systems and controllers, meaning it did not utilize direct control via the network. Additionally, UT^2 was trained using human traces, so it did not discover new behaviors, but was attempting to emulate human ones.

In comparison to these works, the most distinct properties of this work are that it uses NEAT to produce agents with direct, neuronal control and that we seek to see what behaviors the agents will exhibit, rather than seeking to produce specific behaviors. We hope that the agents will learn well-performing behaviors, which

we encourage through fitness scoring, but we do not build any behaviors into the system. The network must learn to move towards goals and define good actions.

6.2 Design of the agent architecture

When building the system, there are a number of design decisions to make. For example, we must play attention to how we communicate with the neural-based agents. This involves defining the formulation of the inputs for the network and creating outputs which can translate signal values to meaning.

We must also define our testing environment as well as how training will occur. Importantly, we must decide how results will be evaluated and what things will be assessed and how.

6.2.1 Core framework

The system was designed using Pogamut and GameBots2004 to play UnrealTournament2004. The goal was to apply NEAT to the bots to test how well-playing and human-like the result is. This means we needed a system which would allow for NEAT-based agents to control the bots via Pogamut and be able to evolve. We needed systems to initialize, control, and save and restore the genetics in order for the testing to be viable.

Because Pogamut is designed such that bots are created using the Java programming language, a Java implementation of NEAT was necessary. NEAT itself was originally written in C++. Fortunately, there are freely available implementations of NEAT for several languages. jNEAT is a Java implementation created from the original C++ implementation of NEAT and includes a few tests of the system as well as a basic GUI for the test environment. The core NEAT implementation from jNEAT was used as well as the saving and loading systems. The file operations needed to be slightly modified to allow for non-contiguous testing and generation tracking, but otherwise the implementation served. There were a few compilation errors which needed to be fixed at the very beginning, but these mostly originated from non-ASCII text in the code.

In order to organize the NEAT variable assignments and the motor and sensor initialization, a data loader class is used. This class sets all relevant NEAT variables at start up, including the population size. It also creates lists of all of the motors

and sensors which the bots will use. The sensors and motors will be discussed in more detail in the following section.

Since Pogamut uses GameBots for network communication to the controlled agents, the bots have a series of events which they implement. They receive events for initialization, have update events, and can register listeners for gameplay events, such as receiving damage or killing something. This allows the bots to be responsive to events around them and to execute logic at a fixed interval.

Pogamut uses a 250ms update tick, meaning a bot gets a chance to execute logic every 250ms. The sensors and motors are updated in this update tick. This means that the agents driven by this system actually react very slowly, as they do not respond to events directly. This speed is still fast enough to match a decent player's reaction time, but it would be very slow compared to a highly skilled player.

The update logic starts by calculating the sensor inputs, which it then provides to the agent's neural network. The neural network is then simulated to produce the output signals. These output signals map to the motor controls for the agent, and are passed to the motors. Some of the motors activate actions directly, though most of the systems, such as moving and shooting, are executed after the motors have set the information they need in order to work appropriately.

There were a few options on how to instantiate the bots themselves. Pogamut provides a system for spawning each agent as a separate thread. The example agents do this in their main functions. These agents will respawn on death, without disrupting the thread. However, they do not persist through a map change, so they need to be restarted if the game restarts.

In the system, the bots are started by a server controller, which is actually the primary thread started when testing begins. When the controller first starts, it initializes the data loader class, which sets up the sensors and motors and initializes the NEAT configuration settings. NEAT is informed at this point how many sensors and motors are in use. The controller then establishes a connection to a running game instance. Once this connection is established, it sets the current map if we are not on the correct map. Once the game is connected and the correct map is active, the controller starts a number of bots equal to the NEAT population size, each in their own thread.

The server controller keeps a running timer for the game session. This is used to force reset the game session after a fixed time limit. The game's native systems for

time limits were too soft and would not change the session if there was a tie, so an alternative method needed to be found. Instead, the controller restarts the game by telling the server to change the map. However, only one map selection is ever used, so the map is only reloaded.

When the server controller resets the game, all of the current population is saved to a file, along with their fitness scores. The file's name indicates the population size, the numbers of sensors and motors, and what generation the agents were. A clone of this file is saved to a file which only indicates the population size and the numbers of sensors and motors without the generation number, thereby indicating it was the most recent population.

When the controller starts up the agents, either from a restart or when testing begins, it attempts to load the most recent population which has the correct population size and number of sensors and motors. If this fails, generally meaning there is not a previous population with the appropriate properties to load, a new population is created. Once the population is loaded the NEAT evolution process is activated, combining and mutating on the previous population. This produces a population of equal size which is then used as the population controlling the bots for this session.

Since the number of bots spawned is equal to the population size which NEAT provides, every NEAT brain has a body. This was chosen to give the entire population approximately the same amount of time to play. It would have been possible to swap out which agents were controlling which bots at defined intervals, such as when they die, but this was not deemed necessary.

6.2.2 Sensors and motors

In order to drive the agents using neural networks, the networks need some form of inputs and outputs. These inputs and outputs are generally handled as signal values. In this case, the signal values are real numbers between 0 and 1.

For this attempt, the agents were given direct control over the bots, as seen in NERO. This means that the agents had outputs which cause direct responses from the bot, rather than triggering predefined complex behaviors. Essentially, this means that the agents move the bots using controls very similar to what a human player would have.

One of the philosophies applied to the design of the sensors and motors was to provide enough information through them that a human player could learn to play

using them. While a human player would have difficulty processing and parsing the information as fast as a computer, if the information were sufficient for a human to accomplish basic tasks in the game environment then a neural network-based agent could theoretically be able to learn to perform these tasks as well.

At the end of development, the networks had a total of one-hundred-fifty-one (151) sensors. The sensors were often duplications of each other, rather than each being unique, meaning that there are far fewer kinds of sensors than sensor instances. The types of sensors include fixed values, raycast distance measures, arc-partitioned enemy detectors, direction and distance detectors for pickups, and health and armor sensors.

There were three fixed values sensors, each emitting a different value; 0, 0.5, and 1. These were provided so the network could guarantee those values if necessary. These function similarly to a ground value. Three were provided to allow for fixed values at different levels of activation. The signals only range from 0 to 1, so 0.5 is the middle value to the range, and the zero and one provide the extremes.

The only internal sensors were health and armor sensors. The health sensor returned a percentage out of 200, meaning at the starting value of health (100) the bot would receive a signal of 0.5 from this sensor. The maximum amount of health in the game is actually 199, so the sensor never quite reports 1. Using 200 was chosen to keep the numbers more rounded. The two armor sensors each covered one of the two kinds of armor, high and low. The types of armor come from slightly different pickups and stack differently. Low armor maxes at 50, while high armor caps at 100. The two sensors give a percentage value out of their respective maximum. Bots start without any armor, so these signals start at 0.

The majority of the sensors designed and provided to the agents are dedicated to information from the external environment. There is a lot of world for the agent to keep track of, and, as it has no systems for memory, it needs to be informed of everything constantly. This means that the locations of all items, enemies, and terrain needs to be fed to it each frame.

The raycast sensors provide a signal based on the proximity to terrain, where a high signal is close proximity and a low signal is far. Each sensor uses a single raycast to determine if there is terrain in the direction that the individual sensor instance looks relative to the bot. This means that, as the bot turns, the raycasts adjust accordingly; the forward pointing cast is always directly forward relative to the bot. There are 27 instances of this kind of sensor, each pointing in a different direction.

Type	Number
Fixed Value	3
Terrain Raycasts	27
Enemy Sensors	30
Pickup Direction	26
Pickup Distance	26
Active Weapon	12
Weapon Primary Ammo	12
Weapon Secondary Ammo	12
Health	1
Armor	1
Shield	1

Table 1: Numbers of types of sensors used in the system.

See figure 9 for more information on the distribution. It is worth noting that this is a weak method of providing terrain information, as it has many blindspots and gives no information on what the terrain might be like; the terrain might be deadly or move.

More rays were not used due to computation time concerns, as each bot was already using twenty seven rays. However, additional rays might have been beneficial. An alternative method for providing solid terrain information was considered which used areacasting, but a cheap method could not be found in the framework.

The enemy sensors are based on the areacast sensors in NERO as seen in Figure 10. However, Pogamut does not provide a system for areacasting; it is likely that this functionality is missing from UT2004, which would make providing it through GameBots or Pogamut expensive. Instead, we define a higher-level tracker object which handles defined arcs and calculates the signal each arc should provide when informed of relative enemy locations.

Each enemy sensor defines an area which it is interested in. This area is defined as two pairs of angles and a distance. The angles define a pyramid which originates from the bot. The distance is the maximum distance away in which we care about enemies being visible.

A loop over all visible enemies is performed each logic tick before the sensors are handled, wherein the squared distance and the relative angles to each enemy are

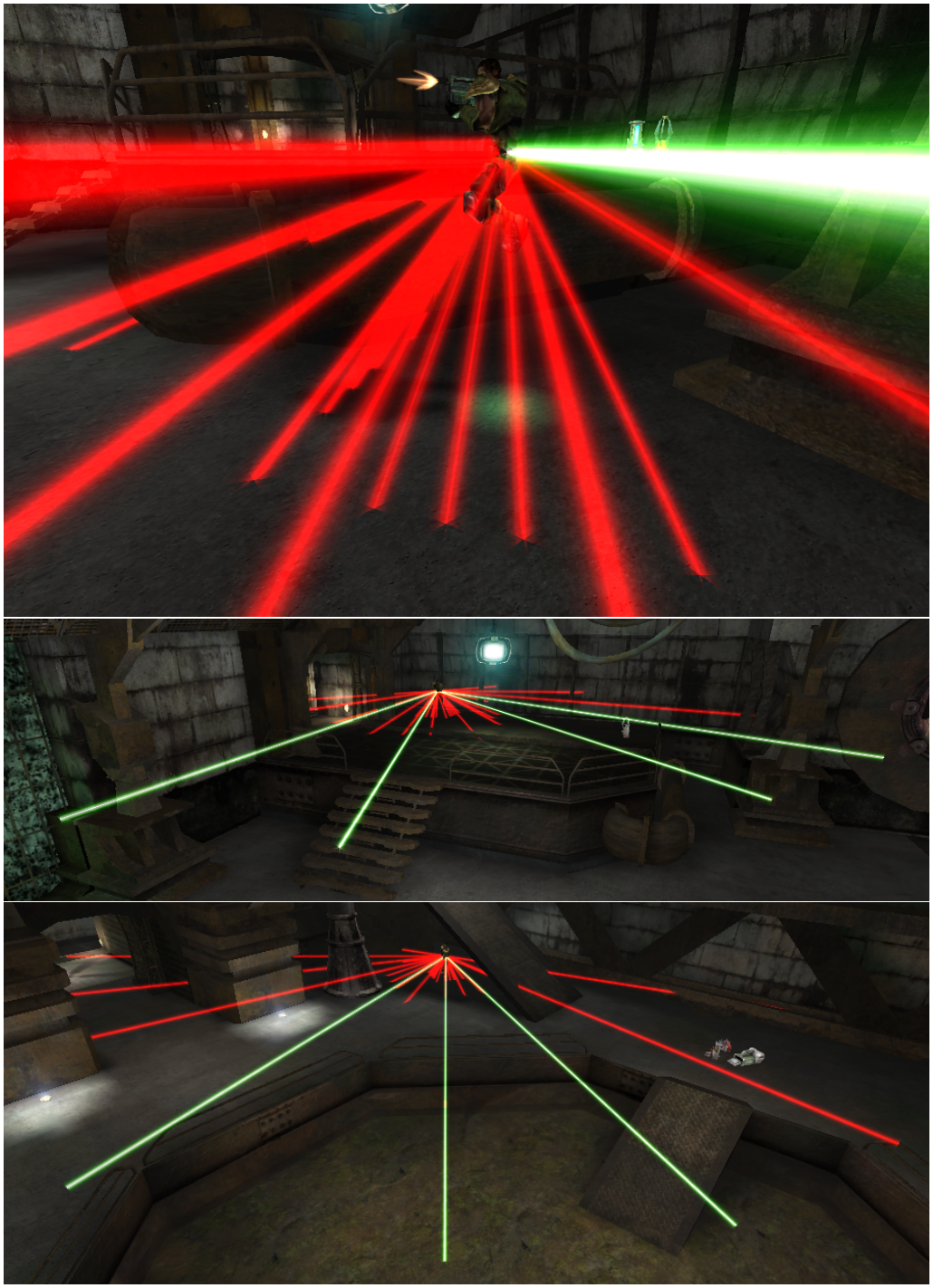


Figure 9: The raycasts form a fan of lines away from the bot, giving it forward vision of the terrain. Green lines are not currently colliding with terrain; red lines are. Note that there are multiple layers of rays, allowing the agent to see downward at different angles. There are two primary types of gaps in the agent's vision here; it does not see terrain upward and there are blindspots between the rays.

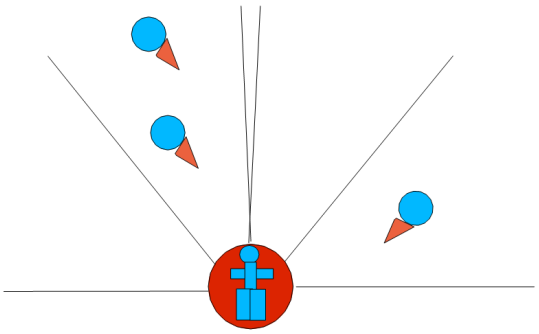
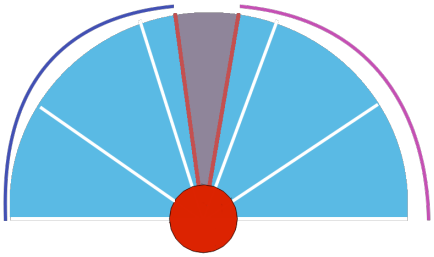


Figure 10: The sensors used for providing enemy proximity information in NERO. The arcs in NERO only worked laterally, meaning the sensors did not consider if an enemy was above or below the bot. Additionally, if multiple enemies were in the same arc the activations from each one were summed. In this project, only the closest enemy in the arc was used. Image Source: NERO [Stanley et al., 2005]

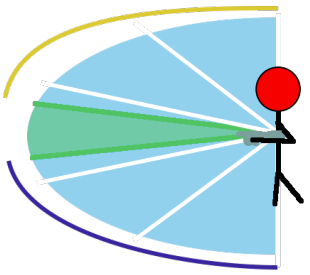
calculated and passed to the tracker object. The tracker object informs all of its stored arcs of the enemy information passed to it, and each arc handles if that enemy is within its area and updates accordingly. The arcs are responsible for calculating the signal value they should provide. The signal produced is calculated using the inverse distance activation, such that a higher value means closer proximity. For the calculation the squared distances are used.

The enemy sensors themselves do very little work. On bot initialization the sensors register the arc area they are interested in with the high-level tracker. Then, when queried, they return the signal of their matching arc. The sensors return the signal exactly as it is calculated, so high values indicate close enemy proximity. This provides the network with information on visible enemies with no artificial blindspots. This system also allows for arcs to overlap, which is used to provide an additional targeting arc at the bot's look focus, as well as left and right sweep arcs to help it adjust its aim. See figure 11.

The agent is also provided with a number of sensors to aid it in locating pickups in the game. The pickups include basic items, like ammo or health, which often come in many forms. There are also pickups for each weapon available on the map. In order to provide the information of where a pickup is relative to the agent, we provide two signals per pickup. The first signal is a direction to move in order to approach the pickup and the second is the total distance along the path. In order to produce these signals, we need to build a viable path to reach the item. We



(a) The environment in front of the bot is partitioned into five primary sections; FarLeft, Left, Center, Right, and FarRight. There is also a tight focus partition and vertically narrow left and right partitions.



(b) The five primary partitions are further partitioned into smaller areas by splitting them into Down, Low, Mid, High, and Up partitions. The tight focus partition is only as tall as it is wide. There are also narrow Up and Down partitions.

Figure 11: Each arc partition provides a signal based on the inverse distance to the nearest enemy in the arc's range. If an enemy is very close a large signal is provided. If there are no visible enemies in the arc then it returns 0. Most of the arcs overlap only at their edges, but a few are designed to overlay the others, providing additional information to aid in targeting. The Focus, NarrowUp, NarrowDown, NarrowLeft, and NarrowRight partitions form an elongated crosshair over the other primary partitions.

do this by having a manager object attached to the bot which calculates paths to all registered pickups using the Floyd-Warshall map provided through Pogamut. The sensors then use the calculated path to provide their information. The distance sensor simply returns the inverse distance to the pickup over the total path, meaning proximity has a higher signal value. This uses a large maximum value after which 0 is returned. The direction sensor returns a unit circle value which indicates in what direction the next navigation node on the path is; the signal value is distributed around a circle, so that each discrete value is a slightly different angle.

As an example, lets say there is a weapon pickup on the other side of a wall from the agent, as in Figure 12. If we were to give the agent a direct line to the pickup, it would run directly into that wall. Instead we look at the built-in navigation map and determine the shortest path to the pickup. From this, we can determine which way the agent should move in order to get there, and approximately how far away the pickup is.

Our distance sensor can simply provide an inverse distance using the squared values. We actually use the equation

$$1 - distance^2 / MaxDist^2$$

in order to determine the signal strength. The direction sensor is a little more surprising, as we must consider the second node on the path rather than the first. If we used the first node in the path, we would always be pointing at the nearest node in the navigation map. This is because we must first determine where we are in relation to that map, so the first node is the closest node to the bot. The only time this is the desired direction is if the path only contains that node, meaning the pickup we are looking at is there. Instead, we provide the direction to the second node on the path.

In addition to the pickup sensors, each weapon has three additional sensors. The first is a true or false value as to whether the weapon is active. This provides the bot with the information of which weapon is currently active. The other two weapon-specific sensors are ammo sensors for the primary and secondary ammo for the weapon. The ammo sensors return a percentage value out of the maximum for that kind of ammo. This allows the agents to track how much ammo they have in each weapon.

There were twenty two motors provided to the agents which allowed for direct control over the actions of the bot. They include basic motor function such as forward,

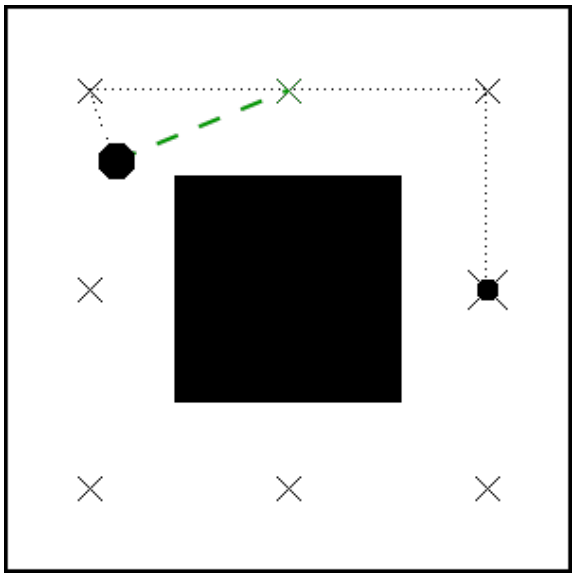


Figure 12: The pickup we are looking for is around a few corners, on the other side of a solid wall. The path to the pickup is marked. We would provide the direction to the green node as the direction signal.

backward, and strafing left and right, rotational control, primary and alternate fire, jumping, and crouching. Additionally, there is a motor for each weapon which is used to activate that weapon. Most of the motors were designed to handle movement through space. They were designed to give the agents direct control over movement, rather than moving to or away from things.

The lateral motion motor controls, like the environmental sensors, work relative to the direction of the bot; for example, the forward motor always moves the bot in the direction the bot is facing. The movement motors are handled collectively, as Pogamut does not handle multiple movement actions well. Instead, each motor sets or alters some information which the bot tracks. After all of the motors have been processed we can calculate the desired movement vector. Once we know where we are moving to a movement action is created to perform the translation.

The rotation controls are also built into the lateral movement system, changing the look vector of the bot based on how the agent desired to turn. The rotation is calculated similarly to the movement, such that the motors inform the agent how they wish to turn. The rotation is then performed relative to the bots current facing direction.

To exemplify this, the motors can be thought to pull the bot in different directions. When they receive a signal, they provide a motivator in their own direction. Fig-

Pickups Tracked
Health Pack
Mini Health Pack
Super Health Pack
Super Shield Pack
Extra Damage Powerup
Assault Rifle Ammo
Assault Rifle Grenade
Bio Rifle Ammo
Flak Cannon Ammo
Lightning Gun Ammo
Link Gun Ammo
Minigun Ammo
Redeemer Ammo
Sniper Ammo

Table 2: The non-weapon pickups which were tracked by the system.

ure 13 arranges the motors so that we can visualize about how they affect the bot. Opposite signals cancel each other out, so an equal signal to both the left and right motors would result in a net-zero change.

The original design of this movement control system was based on the idea of each movement direction being a single motor. However, since the signals are positive only, this was replaced with two motors per axis. This is because the motors were originally designed for the signal range of -1 to 1. It would also have been possible to scale the activation, such that 0.5 is the midpoint, rather than using 0 as the point of inversion.

Primary and alternate fire, jumping, and crouching are all activation motors with a high bias. Once the signal reaches a certain level, the motor is activated. In the case of the jump and crouch motors, this means the motor creates an action for the bot to execute which is then sent to Pogamut. In the case of primary and alternate fire, this means that a flag is set in the bot determining whether it should be firing, and the bot handles any changes in this state similarly to how the lateral motion is handled.

Because of the way crouching and jumping are handled, if both fire one overrides the other, as it is not possible to perform both simultaneously. A controller could have

Weapons Tracked
Translocator
Shield Gun
Assault Rifle
Bio Rifle
Shock Rifle
Link Gun
Minigun
Flak Cannon
Rocket Launcher
Lightning Gun
Sniper
Redeemer

Table 3: The weapons which were tracked by the system. Each weapon also counted as a pickup.

been created for this, but did not seem necessary for just jumping and crouching. While the controller might have allowed for double jumping to be handled, this was a lower priority task and a simple method for handling this was not found.

The weapon activation motors were the only motors which did not involve spatial motion. Weapon activation was handled differently from jumping, as a bias system would allow for many weapons to be activated at once. This produced a scenario where the bot would continuously change weapons, often alternating between the two earliest defined, activated motors. Instead, the motors produce a bid value, using the signal received as the bid, which they register to an arbitration object attached to the bot itself. Once all of the motors are processed, the arbitration system looks at which weapon is most highly desired and activates that weapon. This eliminates the rapid switching and gives the bot a bit more freedom in how it uses these motors.

6.3 Project lifespan

Some elements of the project changed as the project developed. Generally, these were responses to failures in the existing systems or programming errors. These changes impacted how the final testing was carried out.

Type	Number
Movement	4
Rotation	2
Fire	1
Alternate Fire	1
Jump	1
Crouch	1
Activate Weapon	12

Table 4: Numbers of types of motors used in the system.

The original tests were primarily to get the bots functional and to test the basic motors and sensors. After a few runs it was noted that the populations were rapidly converging upon single networks. This turned out to be because NEAT was not configured properly and was not actually performing any crossover or mutation. Additionally, many sensors and motors did not work properly. Eventually, a debug system was added in order to test that the inputs and outputs signals were being produced correctly, which greatly aided the testing.

Some sensors and motors changed over the course of the project, being replaced with better systems; for instance, the enemy detection was originally made to use raycasting like the terrain detection does. The weapon selection originally used activation, like jump and crouch, but this resulted in the rapid switching effect described above and so the motors were changed. The distance to pickups was also a later addition, and the first set only told the direction to move in for each pickup.

Initial testing attempted to train the agents to be competitive by having native bots to play against. However, at later stages, once the system was working properly, this mostly resulted in the agents learning to avoid play, as we shall see in the next chapter. In order to give the bots a chance to learn to perform other behaviors without having to learn to hide or evade first, a system of co-evolution was decided upon. In addition, the population size varied many times over the course of different tests to try and find a good number.

6.4 Experimental procedures

The final testing system used co-evolution on a population of eighteen. These were decided upon based on the results of the system trials. Co-evolution gives the agents

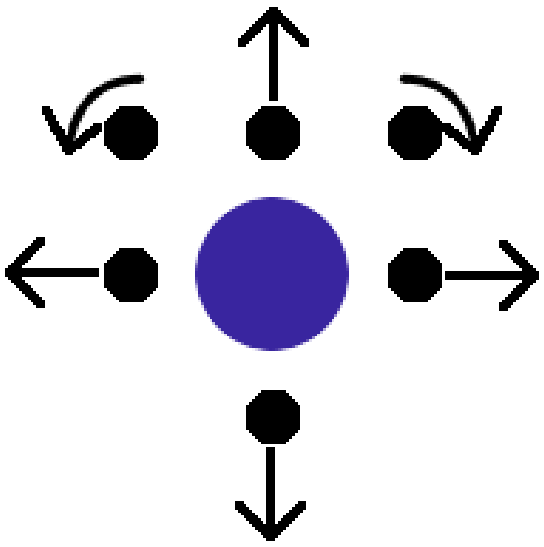


Figure 13: The movement motors induce a change in the desired position or looking direction of the bot. A forward signal motivates the agent forward, while a backward signal does the reverse. If both of these fire at the same time, they have a nullifying effect upon each other.

a chance to experiment and adapt in an environment which they largely dominate. Since they are free to explore and the most deadly or dangerous element in the environment is the agents themselves they are never far outmatched.

The population of 18 was chosen after experimenting with population sizes ranging from 6 to 32. A population of six was felt to be too small. The map is quite large, so six players wandering randomly would not frequently meet. Additionally, a population of only six has limited space for exploration and speciation. Conversely, a population of thirty-two resulted in the map being quite crowded. The population of eighteen resulted in a good consistency across most of the map.

The game mode chosen for testing was FFA. This implies that the agents are not learning cooperative behaviors, but to play individually. This simplifies some of the state, as this is the simplest game mode. The agents did not need to be provided wth state information for allies or or learn to avoid friendly fire.

Since the system uses saving and loading, it could be seamlessly stopped and restarted later, allowing it to be run for long periods without needing to run continuously. The target for the final test was to run for a thousand generations. In practice, this is enough to exhibit basic behaviors in many cases. At a thousand generations we can expect to see at least some of the basic behaviors expect of an

agent in the game to developed.

An additional bonus to this saving and loading system is it provides additional data. For each generation the fitness of every agent for that play session is known. Additionally, the generation can be reloaded to review the agents' behaviors.

For the duration of testing, no human interaction was added; the agents played entirely on their own. At various intervals an observer logged into the game as a spectator to evaluate the behaviors of the agents.

7 Practical testing and assessment

Over the course of the development of the project, several formal and informal trials were conducted. While most of the trials were intended to be to evaluate the function of portions of the implementation, such as the motors, they nevertheless provided interesting perspectives on the application and results of the architecture overall.

We also observed many interesting behaviors, both in the system tests and the long experiment trial. While the results were not as effective as hoped, they were still quite interesting and promising.

Ultimately, the system is found unviable for such a large and complex environment. Improvements might allow it to function, however.

7.1 Initial testing and experimentation practices

Several shorter trials were carried out with changes and improvements between them. Many of these were used to evaluate the status of the system at the time and the improvements were based on the results of the previous trial. These tests were intended to be used to evaluate the state of the system at various points, rather than to evaluate the system overall.

Initial trials did not yet have a server controller, so the map was changed to a randomly selected map when the game timer expired. The controller was added to give control over the map change to the testing environment. This allowed the testing to work on a very fixed timing system and to always use the same map. From this point onward the Asbestos map was used.

The Asbestos map was selected for various reasons. A key point is that it is a deathmatch map, meaning it is not designed for capture the flag or teams, but

instead to give to run around and to have fairly balanced and distributed spawn points. Secondly, it does not have any terrain hazards – things like lava, acid, or pits – which kill anything which falls into them. This means the bots don't have to learn to avoid such hazards. It also has very few lifts, meaning the navigation is simplified to mostly linear terrain. It included many ramps and there were elevations, but with only one meaningful lift which can be safely ignored it was one of the simplest environments to traverse. Additionally, it is a large map, which gave enough space for a larger population.

In order to observe and evaluate the bots, the server was joined as a spectator. This allows the observer to fly around the game environment without interacting with it. The bots were tagged so that they could be found rapidly within the space, and movement vectors were attached to them to allow the observer to see in what direction they were moving. Health bars and other such information were also provided via Pogamut.

In addition to in-game observation, in each game session one of the bots was selected to provide signal information from its sensor and motors. This was provided as a table view in a basic Java GUI window. While difficult to read, this information could be used to produce a general impression of how the agents were functioning and what they were seeing.

The fitness system developed in several ways over the course of the short-term trials. Initially, fitness was based on kills and deaths, just as a temporary placeholder. In the final system, fitness also valued pickups at varying values as well as damaging enemies. Table 5 shows the fitness score values for certain events. For a short time a fitness value was attached to distance moved, but this was eventually eliminated as it valued running in circles more highly than exploring.

Event	Value
Damage Other	1.0
Kill	10.0
Take Damage	-0.5
Death	-4.0
Picked Up Ammo	2.0
Picked Up Health	2.0
Picked Up Armor	2.0
Picked Up Shield	2.0
Picked Up Weapon	2.0

Table 5: Fitness scores of certain actions and events.

7.2 Results from initial testing

For most of the short-term trials, native bots were spawned for the agents to play against. The bots never developed to be at all competitive with these scripted bots, so the native bots were removed. However, the bots did learn to move randomly when damaged, and to jump randomly when moving. Many of them moved constantly, though some occluded themselves in geometry immediately after being spawned, not moving until injured. During a time when the enemy sensors were broken, majority of the bots tended to occlude themselves, rather than evading enemies they could not detect.

For a few of the trials, NEAT was configured to not perform crossover or mutation. In these trials the populations rapidly converged upon the most fit individuals. By the third or fourth game session all of the bots would perform the same behaviors. This resulted in two populations of bots which rotated in place and fired until out of ammo. One of the populations jumped constantly, while the other crouched. The first population was lacking motors to crouch, and neither population had item pickup sensors. This was before the enemy sensor rewrite.

Towards the end of the system development the trials were usually carried out for longer periods to test how the agents were developing. The most commonly observed behavior was for the agents to move randomly when injured and to attempt to occlude themselves behind geometry.

7.3 Results from the long-term trial

The primary, long-term trial was executed on a locally hosted server with the bots and controller attached. No native bots were used, and the game itself did not have a score or time limit. Instead, the server controller from the project managed the restarting of the game and initialization of the bots. The system was left to run for over a thousand generations, which required about 80 hours.

The results of the long-term trial were varied. Mostly, the bots stood in place unless influenced by something in the immediate vicinity, like a very close pickup or enemy. Some of the bots did explore the space at what appeared to be at random until they encountered a pickup or enemy.

Some of the bots which did not explore instead tended to occlude themselves within geometry. Occasionally they would fire from their concealed locations, particularly using the area-of-effect fire. Generally, this fire was focused away from immediate geometry.

Once a nearby pickup was encountered most of the bots would generally move towards it. Upon encountering a nearby enemy the bots would generally attempt to fire upon them. Sometimes the bot waited until the enemy was appropriately in their line-of-fire, though often they would simply fire if an enemy were visible. They frequently made use of the area-of-effect alternate fire in lieu of aiming.

The bots never exhibited any evidence of aiming. Some would hold their fire unless they would likely hit the enemy, but the closest any of the bots got to aiming was to rotate slowly until aligned.

Often, bots would be seen dueling each other in groups. These bots were generally injured, though few kills resulted. Most of the bots dueled by moving in a circle facing outward; they would turn and strafe in order to circle around a point behind them. They did not consistently keep an enemy in front of them this way, and would sometimes produce overlapping circles. If they encountered geometry then they simply collided with it and continued moving in the same pattern.

The bots rarely score a kill even after over a thousand generations, though most game sessions would have two or more. They are still not well-performing and do not present any kind of challenge appropriate to play against a human player. The bots are often out of ammo, meaning they cannot fire upon enemies. They do not seem to have established a connection between these two values, as many would attempt to fire upon enemies even when out of ammo.

The bots never seemed to learn to seek anything they did not have. Generally they would pick things up opportunistically, but they did not navigate or explore well enough for this behavior to benefit them majority of the time. Since all bots started at half of maximum health and with only basic weapons they should have had many things to seek.

Even the bots which randomly explored the space did not do so quickly or well. They had serious issues navigating or maneuvering, often running into the world geometry. When they did exhibit an intent in movement towards something, they would strafe to the item and continuously rotate. At times when strafing from their current orientation would not bring them nearer to the target they would pause and rotate in place. This would produce odd arcs of movement and leave them waiting in place for periods of time.

Most of the signal patterns observed had the agents producing very slight motor signals for the forward and backward motors and very strong signals for the left and right motors and rotational motors. The minor signals for front and back were approximately equal, causing them to effectively nullify the other. The left and right motor and rotational motors produced a similar nullifying effect upon each other. In practice, in order to stop moving the agents would maximize both the left and right motors, rather than dropping them to zero. The rotational motors were utilized similarly, though they rarely stopped rotating.

The weapon selection signals observed indicated that the agents' preferred weapon was often actually the starting weapon, though occasionally they preferred the minigun. The next ranking weapon, after these two weapons, was the flak cannon. They rarely possessed other weapons besides the starting weapons, however, so it is unknown whether they would have learned to use other weapons or not.

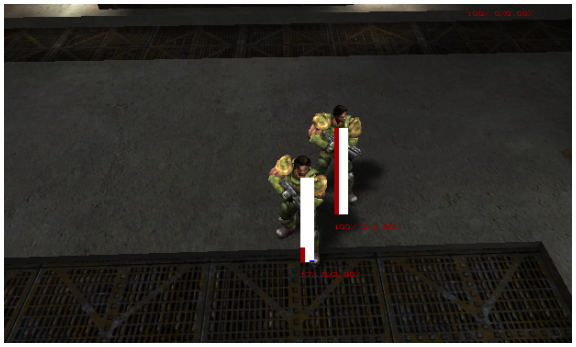


Figure 14: When two or more bots occupied an area, they often would perform what looked like a duel. Neither could aim and often victory went to the one with the most ammo or the one which got off a lucky shot with a grenade. Many duels ended in the bots both running out of ammo.

Figure 15: The agents frequently were observed placing themselves in corners or holes, occluding themselves behind geometry. The one pictured here rotated constantly.

7.4 Analysis

Overall, the agents are neither very human-like nor well-performing. They do have some interesting behaviors that are somewhat human-like. Ultimately, the agents are not functional as adaptive agents, and would not be capable of providing a challenge to a human player; they adapt far too slowly.

In terms of keeping a player in the flowchannel, the resulting bots only succeed for a short period of time. They are interesting to observe, but far too easy to kill and avoid, so they pose no difficulty to a human player. It might be possible to adapt the system to produce agents which hold the player in the flowchannel more consistently, however.

7.4.1 Assessment of behaviors

The bots have been observed to occlude themselves behind geometry in many of the trials. This is likely an attempt to hide for safety. It improves the bot's chances of not dying, at the cost of not finding many pickups or inflicting much damage. Some of the bots which exhibited this behavior would occasionally fire, even if they could not see an enemy. This was probably an attempt to improve their fitness value via inflicting damage to opponents. Often this fire occurred when an enemy was present, though some would launch area-of-effect grenades even when they could not see an enemy. Randomly firing grenades into hallways or down slopes is a relatively effective tactic for dealing damage when you cannot see enemies or have difficulty aiming at them.

Hiding like this is a reasonably valid strategy, as it blocks enemy line of sight and fire (See Figure 16). This decreases the chances of the bot being noticed by enemies. It also means that an enemy has increased difficulty when attempting to hit the bot.

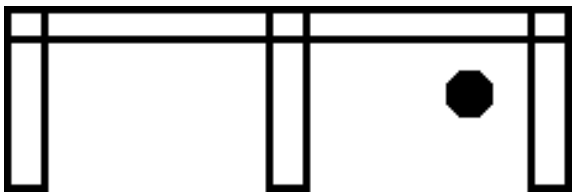


Figure 16: With the geometry of the level partially occluding this bot, it is more difficult to see or fire upon.

The hiding behavior was prevalent in the short trials. This is likely in part due to the fact that they had limited or broken enemy sensors, meaning they could not correctly determine the cause or source of their deaths. They also did not have much chance of fighting back this way.

In the longer trails the hiding behavior was still reasonably common but not as pervasive. Many of the agents would appear to explore a bit more, as well as to opportunistically collect pickups.

This hiding behavior might have been a local optimum. It's difficult to change the strategy without losing the element which makes it a strong tactic: being difficult to shoot at. In order to improve beyond this point, agents must actually leave concealment. This is sometimes difficult for new players, as well.

In the longer trials, the agents have also been observed to collect pickups opportunistically. This is a very common tactic amongst human players; as a player moves about the map, they will frequently collect whatever is nearby. While many human players will move about the map seeking something in particular, it is not uncommon for a human player to simply run about the map at random. The success of the opportunistic pickup behavior is good. This is a commonly exhibited behavior of human players, and if carried out well can make the agent seem very human-like.

In the shorter trials, the agents never exhibited any tendency towards exploration. In the longer trials, however, there were some bots which would. This exploration appeared mostly random, rather than directed towards certain distant items, implying that the agents were not seeking distant weapons. The exploration might have been induced to encourage encountering enemies, which would be productive for the agents. It might also have been to encourage encountering pickups, which could be collected opportunistically.

The exploration occurred mostly as only somewhat guided movement. The agents generally moved into open space, occasionally clinging to walls. It seems that the exploration was essentially a "move anywhere" behavior, rather than a "what's over there?" kind of behavior.

Anytime the agents occurred in groups, they would appear to duel. This consisted largely of random movements and poor attempts to fire upon each other. While the behavior did not occur well, its existence is promising. Killing and damaging enemies is a very fit activity for the agents, so this did maximize their fitness. The methods the agents employed also reflected the intent of dealing damage, while deal-

ing with other skill limitations. Agents would either fire normally when they might hit an enemy, or attempt to fire an area-of-effect grenade. These are both common behaviors in new players, as they maximize possible damage without requiring much skill in aiming. The first approach is known as “spray and pray”, attempting to hit an enemy by filling the space with projectiles, rather than with directed fire. The second simply minimizes the need to aim.

The agents generally lacked a seeking behavior. This is unfortunate, as it is critical to playing well. It is not uncommon for new players to find themselves out of ammo frequently, should they succeed in surviving for long enough periods, but they fairly quickly learn to obtain ammo and keep an eye on the current amount they have. While this means that even human players fail this task, the agent is performing at the very worst level in this regard.

The agents seemed to have selected the primary weapon as a preferred weapon. This may have simply be random chance, as the agents were not seen to collect multiple weapons during play, however it might have been a learned behavior. The agents relied on the methods of fire from the primary weapon, particularly the direct fire and area-of-effect secondary fire, so moving to other weapons could have been detrimental to their overall fitness.

Another perspective on the possibility of the selection of weapons being learned, is that the weapons most preferred were generally some of the best weapons in the game for people with lower skill. The minigun provides a high rate of fire and works well for the spray and pray strategy the agents exhibited. The flak cannon, which also had a high bid value, is very good for filling an area, as it has a larger area of fire. It also possesses an area-of-effect alternate fire. If the agents had been given more experience with weapon selection, they may have preferred this weapon over the others.

While not strictly a behavior, how the agents move contributes to their human-like appearance. Since they do not strictly follow the Floyd-Warshall map, they move more like human players would. This makes them more convincing than the native bots of the game. However, they typically move erratically, rather than with an obvious intent. While observation tends to indicate that they move towards things, they do not move directly towards things.

7.4.2 Implementation assessment

A number of design choices in the implementation of the system may have led to some of the less ideal results in the agents. We should identify these and separate them from the issues in the architecture and method itself. It is possible these would be improvable without altering the architecture; rather, only elements of the implementation need to be changed. We also should consider what improvements can be made to these implementation areas.

Many of the issues the bots had were related to movement and orientation. It is likely the bots would have performed far better if this problem were resolved. It may be possible to produce a better collection of motors for enabling the bots to move about the space.

Since the bots were constantly producing conflicted motion actions, it is likely that providing them with opposing motors was in error. Simplifying this system to a single motor per axis would likely improve the bots' ability to maneuver. The easiest way to do this is to use the middle value as the zero point, treating the motor signals similarly to the inputs provided by the control sticks on a gamepad controller.

An alternative solution for the motor control issue is to utilize some form of higher-level abstraction, such as an arbitrated task-selection method which would execute predefined behaviors such as moving towards the nearest pickup of a type or different movement patterns for evading and attacking enemies. This method would diminish the versatility of the bots, making them play much more like a standard AI. This would improve the relative skill level of the agents compared to this trial. However, it would reduce the possibility of human-like behaviors in the agent.

If the motor controls were altered to handle horizontal rotation as a single motor, it would make sense to alter the input signals from the pathing sensors to match. As such, the signal passed by a direction sensor would be the rotation signal to provide in order to turn to face that direction. This would allow nearly linear pass-through of values to produce logical results.

The bots spent a lot of time out of ammo, often stagnating the game in a state where none of them could even fire upon each other. This is probably a failure of the fitness system, as there was no cost or indicator that attempting to fire while out of ammo was wrong. If a negative fitness value had been applied to either being out of ammo or attempting to fire while out of ammo this might have improved. However, it

may not have produced ammo seeking. If being out of ammo were penalized they may have instead learned to conserve their ammo. If firing while out of ammo were penalized they may have learned to simply not to attempt firing when out of ammo.

Obtaining pickups was already valued as fit, but one possibility would have been to value obtaining needed pickups more highly. Additional fitness could have been attached to obtaining health when running low, or on collecting ammo if low. Alternatively, have larger supplies of things could have been evaluated as fit, giving the agent a higher fitness score for the items in its inventory.

Since the bots never learned to aim, it is likely that a fitness value for being aligned to hit an enemy would have been beneficial to this. However, determining whether the bot's current weapon would have hit an enemy might have been an overly challenging and computationally expensive task simply to provide a fitness bonus. Instead, a negative fitness value for missing would be much simpler to provide and potentially function similarly. This runs the risk of the agents learning not to fire at all, but if the penalty is far outweighed by a positive fitness value from hitting enemies then it might work.

When designing the system, we were presented with the difficulty of determining what information to provide to the agents and how. Given the results of the experiments, it is difficult to evaluate how well we accomplished this. The information provided to the agents is reasonably thorough, though some spatial information is missing in terms of geometry detection. It is possible that the number of sensors contributed to the agents' slowness of learning, but providing less information may have impaired their ability to learn overall.

The terrain sensors likely need to be improved, as they provide fairly sparse data. A wider battery of sensory information could be used, though this may require changes to the architecture. One possible method to improve the geometry detection may be to use areacasting, as in the enemy sensors. This would eliminate the blindspots in the agents' sense of the local terrain. Backward-facing detection may also be a good improvement, to compensate for the lack of memory of local terrain.

A system of saturation could also be applied to simulate memory, such that sensors changed values more smoothly. This could help smooth out reaction times as well as the reactions themselves. This could also help limit the agents to reacting within time frames which seem fairly human.

An alternative version of the enemy sensors could utilize bleeding, in order to sim-

ulate a general sense of an enemy being in a particular direction. This could also be combined with tighter, smaller partitions. If the focus partitions had a higher sensitivity while the outer partitions bleed more this could produce a fairly accurate simulation of human perception. This would be best implemented if combined with signal smoothing.

The generic nature of the ammo and weapon sensors resulted in a larger number of sensors than strictly necessary, as well a set of twelve sensors which are on in a mutually exclusive fashion. Not all weapons even have a secondary ammo type, so we did not need a sensor for each weapon's secondary ammo. For most of these weapons, the weapon's primary ammo was reported again as secondary ammo. However, this was not viewed as a problem in context, as the impact was seen to be fairly low. It is unclear if this redundancy had any affect on the agents' learning.

The weapon activation sensors could have been compressed into a single sensor, rather than using a sensor per weapon, if we wished to reduce the number of sensors. In this case, we preferred to keep the information channels more direct, even though this added a need to learn the associations. It is not clear whether one system would have been easier for the agent to understand over the other. The method chosen was to provide information which would make the most sense to a human user or observer.

It's possible that the agents did not learn to seek anything as the information signals to navigate towards the pickups which were provided did not map to the motor signals required to actually do so. Rotating toward the next point on the path to the pickup would be most easily accomplished by rotating in the direction of the angle until it reduced to zero. The simplest way of doing this would be to rotate in one direction in all cases until correctly oriented to move towards the pickup, which is actually very close to what's observed from the agents.

The jump motor actually limits the agents' ability to move. It does not support a double-jump, which is a mid-air jump which can be performed in-game. This is not a huge issue, but for advanced navigation purposes it would be good to have. The crouch motor and the jump motor also conflict, but this is more a limitation of the game than the implementation here.

In the experiments, we attempted to train the agents on all tasks simultaneously. It might be more productive to attempt to train agents on single tasks before moving on to others, as used in NERO. For example, agents could be trained to move towards enemies through fitness scores, and then once this has been learned have

the same population trained to aim and fire upon the enemy. There was success in this approach in NERO, so this might be a viable method of training the agents in a more complex environment as well. This would also allow for training agents to play in different ways.

After testing was complete, it was noted that the agents did not have sensors for rocket launcher ammo or normal shield packs. For this set of trials, this oversight did not seem to matter, however for future implementations it would make sense to include these pickups in the list of tracked items.

7.4.3 Discussion of the architecture

Some issues in the results may require the architecture itself to change. It is important to consider these and possible extensions to the architecture by which the results may be improved, without breaking the goals of the original architecture.

The fact that hiding may be a locally optimal solutions indicates that the system is still reasonably susceptible to this problem. One method which could be used to encourage moving away from hiding is to attach a cost to standing still. In comparison, humans explore most when bored. If the agents could be made to be 'bored' in a sense, generally to find repetitive actions or standing still to be costly, then they may be motivated towards new behaviors. This could benefit their appearance as human-like agents as well as improve their rate of learning. It could also be used to encourage exploring the search-space, similar to simulated annealing.

One of the major limitations of the architecture is it lacks any properties of persistent state. The network can only respond to its current state, and not what it was looking at or doing previously. This can result in the agents changing direction rapidly in short spans of time, sometimes bouncing between two points. This movement pattern was observed during the initial trials. Additionally, agents forget any enemies they lose sight of.

Adding any form of state tracking could enhance the agents in this regard. One option is to utilize a second set of sensors which decay over time, allowing for previous signals to be retained in some way. If each signal from the primary sensor was factored into such a retaining sensor then a smoother transition would be evident. Alternatively, many of the sensors could function this way, only changing gradually.

Another option for state retention is to have feedback built into the network, where the previous motor signals are provided to the network. This could also utilize the

decay method above to provide the motors signals smoothed out over several frames.

A more advanced extension to the system could be to apply a chemical-based state system, akin to that used in the Creatures series. The agents could contain chemicals, which exist within the agent at concentrations from 0 to 1. We could then have a sensor for each chemical which used the concentration directly.

In order to manipulate these chemical concentrations, an organ-like system could be applied. Organs would be able to transport chemicals from the environment into the agent, or push them from the agent into the environment. Other organ functions could be to consume some chemicals to produce others or to store chemicals for release under certain events. It would be possible to make some organs into motors, allowing the agent to release chemicals itself.

These two systems could give the agent a form of memory and allow for more complex state feedback. With memory and state, the agents might be more consistent. This also introduces a system through which boredom or other emotional states might be implemented. If an agent could learn to associate some chemical states with positive or negative fitness, it might be able to react believably to them during gameplay.

The sensory perception of terrain could have been improved using fields, rather than raytracing. However, any field-based information would still require simplification to neural sensors. Drawing from cognitive research and the human eye, some preprocessed information could be provided for where boundaries and elements exist, such as edge detection or distance approximation sensors. It's possible that just having less focused sensors would improve detection, such as having fuzzy area detection.

Alternatively, the chemical system could be utilized to help the agent understand the map. The agent could leave pheromones on the map, allowing it to have a form of memory related to navigation. This might require the ability to sense nearby chemicals on the map, as well as chemicals in different directions. This information may not be easy to provide to the agent, particularly when trying to detect nearby, non-immediate chemicals. Detection of chemicals at the agent's location could easily be accomplished using a duplication of the standard chemical sensors which looked at the agent's position rather than their internal chemical concentrations. Detection of chemicals in the local area could be implemented as larger scope detection, detection of nearby points, or detection of largest, nearby concentrations. In team-based environments, map-based chemical detection could even be used as a form of indirect communication between the agents.

Despite the shorter trials not being meant to demonstrate longer-term learning or to produce well-performing bots, we still established some properties of the system overall from them.

Even in the shorter trails, it was evident that learning was taking place. The underlying system and architecture works towards producing more fit agents over time. When no mutations were occurring, convergence towards the most fit solution of the initial population was guaranteed. This reflects positively on the system as a whole, but is not new information.

Any work on motors or sensors frequently requires modifying the number of sensors. This causes already trained agents to become worthless frequently, as a population of agents has a fixed number of inputs and outputs. This makes iterative development on the AI infeasible.

The system is highly susceptible to minor errors in motors or sensors, and any changes to them may change how an agent behaves. Additionally, the amount of information required is hard to gage, so it is difficult to estimate what information should be provided to the network and how. In theory, more information is better, as the network can disregard useless information, but this can greatly increase the necessary training time. The more information that is provided, the more time the agent requires to build useful models from it.

Training of the agents required several days of time. If run continuously, the training time used would have covered slightly more than three days. Even at this point the agents were not well-performing, so a much longer training time would be called for to produce well-performing agents. This presents the issue that, if it cannot be accelerated, training takes a very long time.

Combined, these qualities make the system infeasible for use in a more time-limited environment, such as the game industry.

There may be solutions to each of these general problems, such as accelerated training or the ability to evolve new connections to added motors or sensors. If the training can be run is a rapidly simulated environment it can be trained in less time. This would permit far more generations to be achieved in shorter time, as well as reducing the cost of iterative development. If a system for connecting motors or sensors which did not previously exist could be utilized then previously trained networks could be retained. This would nearly eliminate the cost of modifying the sensors and motors, making the system much more feasible.

7.5 General assessment and moving forward

The system did not produce as effective results as the other methods presented in the paper. A good deal of this is the simplified nature of the architecture. Because the goal was to see what results this architecture would produce in this context, we did not wish to attach other architectural elements to it to improve play. Additionally, we did not wish to provide pre-defined behaviors to the agents in order to allow the architecture to produce novel behaviors.

Most of the behaviors observed, both in the long and short trials, were interesting. The bots were often reasonably human-like, if terribly stupid. They developed a number of behaviors typical of beginning players, such as running and hiding, though with lower quality motor control. Ultimately, the system does seem capable of producing agents with reasonably human-like behaviors, but the inherent complexity of the FPS game used and the limitation of time limited their reachable potential.

The agents succeed in a very basic and simple illusion of human-like behaviors. This is created in this scenario partially by the range of motion the bots exhibit. The erratic nature of the agents' behaviors also factors into their human-like appearance. This is not a strictly maintainable part, as improving and learning would reduce how erratically they move, though it is likely that it would transition to an unpredictable nature and only slightly erratic actions.

It is possible that the bots would eventually learn behaviors to help them succeed in the game. They have learned approximations of or parts of generally good behaviors, such as firing on enemies or picking up items, which supports the idea and possibilities of the system, but it is unknown how many generations it would require for the agents to reach a level of performance necessary to engage a human player. It is not even known if any number of generations would produce well-performing agents. The task space is complex enough that even at a thousand generations they are not well-performing.

The system in its current state is not rapid nor reliable enough to be practical for most applications. It takes too long to train a population and the result is not guaranteed to accomplish anything.

The system presents a number of opportunities. It may be possible to alter it to produce agents which are well-playing. With more training time and better systems it may be possible to produce interesting bots, though the work required may be too large for normal game development environments.

From an academic perspective, it seems reasonable that this system could be adapted to produce the desired results. The most important improvement would likely be the ability to retain previously trained populations. With this, the long training time would be better invested and changes could be made more freely.

In future tests of this system, it would be beneficial to have an environment which could be simulated more rapidly, allowing for more time efficient training. If the agents could play a full length game in a fraction of the time, many more generations could be produced.

In this environment, the entire population was replaced each game session. It might be interesting to test an elitist approach to see how it compared. Alternatively, a collection of the best bots could be used to produce each population, rather than the previous population.

8 Conclusions

We have tested using NEAT to create human-like, interesting, and well-playing agents for a FPS game. The resultant agents were interesting and somewhat human-like, but not well-playing. It was generally concluded that the task is too complex for the system to produce good solutions within a thousand generations.

We utilized NEAT to evolve neural agents with over one hundred fifty sensors and twenty two motors. The testing was accomplished using Pogamut communicating with a GameBots system in UT2004. Final testing was performed as in co-evolution environment, without native bots.

Our original question was if this architecture can be applied to FPS style games to produce human-like and well-playing agents. The results suggest that it is not a practical approach, especially in a time limited environment. The training seems to be neither rapid nor reliable. It is possible that later generations would have been better performing or that changes to the system may improve results, however. The results suggested that the architecture worked overall, but not in a reasonable time frame. Additionally, the architecture is very sensitive to minor changes, such as a change in the sensors and motors.

Overall, at slightly over a thousand generations, the agents still could not play at a beginner level. They had difficulty traversing the environment and rarely seemed to be able to accomplish basic tasks. Neither do they have the skills necessary to aim

nor do they have any concept of needing ammo to be able to fire. It is uncertain that they would develop there skills or connections with further training, making the system unreliable.

In order to reach even a thousand generations required days of continuous training. In a realistic environment, training cannot take more than a few hours; at most a day. This disparity indicates that the system is no where near the requirements of a rapid learning system, making it completely impractical for online learning.

The results indicate that agents have learned some basic behaviors appropriate to beginning players. They fire when enemies are present, attempt to evade damage, and sometimes appear to attempt to conceal themselves. This suggests that additional time spent training could produce agents with better skills. The resultant behaviors also lacked the usual robotic appearance of most game AI. A large portion of this was their erratic behavior, but the lack of rails to the behaviors made the agents appear to act more smoothly than a standard bot. This achieves, at least in part, the desire for human-like behavior.

The greatest limitation of the system was the need to discard previously trained agents when the underlying system changed. This limits iterative development and means that correcting errors or adding missing sensors or motors is extremely costly. If this challenge can be solved then populations would only need to be discarded if they cannot be transitioned, which could save a lot of training time.

Several possible additional fitness metrics were proposed in the analysis above. These include penalizing missing enemies or firing when out of ammo or giving positive fitness for looking at an enemy. The exact results of such fitness metrics is difficult to predict, it is possible that penalizing missing may train an agent to not fire for example, but it is reasonable to test them.

Alternatively to standard NEAT, real-time NeuroEvolution of Augmenting Topologies (rtNEAT) could be applied to allow agents to train while the game is running. This would remove the need for discrete game sessions and allow agents to improve over the course of a single game.

It was noted that limitations of the system may be the cause of the agents' inability to play well, such as the locomotion controls being poorly designed. The most obvious next step for this system would be to test with different motors or sensors, to see how that affects the results. Two immediate directions which could be taken to improve this system were presented.

It was suggested that non-direct control over movement would be a viable direction to approach. This loses some of the flexibility and power derived from direct control, but reduces the task complexity significantly. An approach similar to the methods used by Westra [Westra, 2007] may be appropriate here.

One limitation to the system is that the agents do not have any form of memory. Some form of internal state is proposed to solve this issue, such as feedback motors or event sensors with decays.

It may be more appropriate to test this architecture in an environment which is more open, allowing the agents to explore more. In general, simplifying the task environment may help improve the results. One option is to use the methods used in NERO, where the agents are trained on specific tasks at any given time, rather than all tasks at once. Alternatively, a simpler game environment may be used.

There are many possible directions for new research and improvements which could be applied to this architecture and evaluated. While this version does not reliably or rapidly produce human-like or well-playing agents, it seems to show potential.

References

Agogino et al., 1999 Agogino, A., Stanley, K. O., and Miikkulainen, R. (1999). Online interactive neuro-evolution. *Neural Processing Letters*.

Andrade et al., 2005 Andrade, G., Ramalho, G., Santana, H., and Corruble, V. (2005). Challenge-sensitive action selection: an application to game balancing. In *Intelligent Agent Technology, IEEE/WIC/ACM International Conference on*, page 194–200. IEEE.

Arrabales et al., 2009 Arrabales, R., Ledezma, A., and Sanchis, A. (2009). Towards conscious-like behavior in computer game characters. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, page 217–224. IEEE.

Bakkes et al., 2009 Bakkes, S., Spronck, P., and van den Herik, J. (2009). Rapid and reliable adaptation of video game AI. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(2):93–104.

Braun and Weisbrod, 1993 Braun, H. and Weisbrod, J. (1993). Evolving neural

feedforward networks. In *Artificial Neural Nets and Genetic Algorithms*, page 25–32. Springer.

Buckland, 2005 Buckland, M. (2005). *Programming game AI by example*. Jones & Bartlett Learning.

Champandard, 2007 Champandard, A. J. (2007). Teaming up with Halo's AI: 42 tricks to assist your game. [Online; accessed 18-February-2014].

Cole et al., 2004 Cole, N., Louis, S., and Miles, C. (2004). Using a genetic algorithm to tune first-person shooter bots. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 139–145 Vol.1. IEEE.

Eshelman, 1990 Eshelman, L. J. (1990). The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. *Foundation of Genetic Algorithms*, page 265–283.

Fogel et al., 1990 Fogel, D. B., Fogel, L. J., and Porto, V. W. (1990). Evolving neural networks. *Biological Cybernetics*, 63(6):487–493.

Fogel et al., 2004 Fogel, D. B., Hays, T. J., and Johnson, D. R. (2004). A platform for evolving characters in competitive games. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, page 1420–1426. IEEE.

Gamez et al., 2011 Gamez, D., Fountas, Z., and Fidjeland, A. K. (2011). A neurally-controlled computer game avatar with human-like behaviour. In *Computational Intelligence and AI in Games, IEEE Transactions on*, volume 5, page 1–14.

Gemrot et al., 2009 Gemrot, J., Kadlec, R., Bída, M., Burkert, O., Píbil, R., Havlíček, J., Zemčák, L., Šimlovič, J., Vansa, R., Štolba, M., et al. (2009). Pogamut 3 can assist developers in building AI (not only) for their videogame agents. In *Proceedings of the First International Workshop on Agents for Games and Simulations*, pages 1–15. Springer-Verlag, Berlin, Heidelberg.

Glosser, 2013 Glosser (2013). Colored neural network. [Online; accessed 13-February-2014].

Gomez and Miikkulainen, 1999 Gomez, F. J. and Miikkulainen, R. (1999). Solving non-Markovian control tasks with neuroevolution. In *IJCAI*, volume 99, page 1356–1361.

Gruau et al., 1996 Gruau, F., Whitley, D., and Pyeatt, L. (1996). A Comparison between Cellular Encoding and Direct Encoding for Genetic Neural Networks. In *Proceedings of the First Annual Conference on Genetic Programming*, page 81–89. MIT Press.

Harman, 2007 Harman, M. (2007). The current state and future of search based software engineering. In *2007 Future of Software Engineering*, page 342–357. IEEE Computer Society.

Harman and Jones, 2001 Harman, M. and Jones, B. F. (2001). Search-based software engineering. *Information and Software Technology*, 43(14):833–839.

Hingston, 2009 Hingston, P. (2009). A Turing test for computer game bots. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(3):169–186.

Hingston, 2013 Hingston, P. (2013). Botprize website. [Online; accessed 18-October-2013].

Hunicke and Chapman, 2004 Hunicke, R. and Chapman, V. (2004). AI for dynamic difficulty adjustment in games. In *Challenges in Game Artificial Intelligence AAAI Workshop*, pages 91–96.

Johnson and Wiles, 2001 Johnson, D. and Wiles, J. (2001). Computer games with intelligence. In *FUZZ-IEEE*, page 1355–1358.

Karpov et al., 2012 Karpov, I. V., Schrum, J., and Miikkulainen, R. (2012). Believable bot navigation via playback of human traces.

Laird and VanLent, 2001 Laird, J. and VanLent, M. (2001). Human-level AI's killer application: Interactive computer games. *AI magazine*, 22(2):15.

Laursen and Nielsen, 2005 Laursen, R. and Nielsen, D. (2005). Investigating small scale combat situations in real-time-strategy computer games. *Master's thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark*.

Lippmann, 1987 Lippmann, R. P. (1987). An introduction to computing with neural nets. *ASSP Magazine, IEEE*, 4(2):4–22.

Montana and Davis, 1989 Montana, D. J. and Davis, L. (1989). Training Feedforward Neural Networks Using Genetic Algorithms. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'89, page 762–767, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Polceanu, 2013 Polceanu, M. (2013). Mirrorbot: Using human-inspired mirroring behavior to pass a turing test. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, page 1–8. IEEE.

Priesterjahn et al., 2006 Priesterjahn, S., Kramer, O., Weimer, A., and Goebels, A. (2006). Evolution of human-competitive agents in modern computer games. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, page 777–784. IEEE.

Radcliffe, 1993 Radcliffe, N. J. (1993). Genetic set recombination and its application to neural network topology optimisation. *Neural Computing & Applications*, 1(1):67–90.

Samuel, 1959 Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229.

Schaffer et al., 1992 Schaffer, J., Whitley, D., and Eshelman, L. (1992). Combinations of genetic algorithms and neural networks: a survey of the state of the art. In *International Workshop on Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92*, pages 1–37.

Schrum et al., 2011 Schrum, J., Karpov, I. V., and Miikkulainen, R. (2011). Utˆ2: Human-like behavior via neuroevolution of combat behavior and replay of human traces.

Schrum et al., 2012 Schrum, J., Karpov, I. V., and Miikkulainen, R. (2012). Humanlike combat behavior via multiobjective neuroevolution.

Spronck et al., 2004 Spronck, P., Sprinkhuizen-Kuyper, I., and Postma, E. (2004). Difficulty scaling of game AI. In *Proceedings of the 5th International Conference on Intelligent Games and Simulation (GAME-ON 2004)*, pages 33–37.

Stanley, 2007 Stanley, K. O. (2007). Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines*, 8(2):131–162.

Stanley et al., 2005 Stanley, K. O., Bryant, B. D., and Miikkulainen, R. (2005). Real-time neuroevolution in the nero video game. *Evolutionary Computation, IEEE Transactions on*, 9(6):653–668.

Stanley and Miikkulainen, 2002 Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127.

Sutton, 1988 Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44.

Togelius et al., 2011 Togelius, J., Yannakakis, G., Stanley, K., and Browne, C. (2011). Search-Based Procedural Content Generation: A Taxonomy and Survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):172–186.

Tozour, 2002 Tozour, P. (2002). The perils of AI scripting. In *AI Game Programming Wisdom*, volume 1, page 541–547.

Traish and Tulip, Sept Traish, J. and Tulip, J. (Sept.). Towards adaptive online RTS AI with NEAT. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 430–437.

Turing, 1950 Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236):433–460.

van der Werf, 2005 van der Werf, E. C. D. (2005). *AI techniques for the game of Go*. UPM, Universitaire Pers Maastricht.

Westra, 2007 Westra, J. (2007). Evolutionary neural networks applied in first person shooters. *Master's thesis, University Utrecht.*

Woodcock et al., 2000 Woodcock, S., Laird, J. E., and Pottinger, D. (2000). Game AI: The state of the industry. *Game Developer Magazine*, 1.