University of Minho
School of Engineering

Luís Paulo Ferreira Gomes Neto

Development of a bot like entity to emulate an user in a tridimensional virtual environment

october 2023

University of Minho
School of Engineering

Luís Paulo Ferreira Gomes Neto

Development of a bot like entity to emulate an user in a tridimensional virtual environment

Masters Dissertation
Master's in Informatics Engineering

Dissertation supervised by
Paulo Jorge Sousa Azevedo
Rui Manuel Ribeiro de Castro Mendes

october 2023

Acknowledgements

This undertaking would not have been feasible without the invaluable support and opportunities extended to me by Paulo Jorge Sousa Azevedo and Rui Manuel Ribeiro de Castro Mendes. I am sincerely grateful for their unwavering guidance and insightful contributions throughout this endeavor. Their belief in both myself and my vision, even during the initial stages when clarity was lacking, has been instrumental to my progress. I extend my heartfelt appreciation to my colleagues for their continuous support throughout this arduous journey. The shared moments of academic challenges and triumphs shall forever be cherished, filling my heart with profound gratitude. Furthermore, I would like to express my deep gratitude to the Department of Informatics at the University of Minho for shaping me and equipping me with the necessary tools to navigate the realm of software development with confidence and proficiency.

Statement of Integrity

University of Minho, Braga, october 2023

Luís Paulo Ferreira Gomes Neto

Abstract

This dissertation explores a novel approach for the development of a virtual entity or artificial intelligence capable of simulating user behavior within the immersive and expansive virtual realm of the World of Warcraft video game. Classified as a three-dimensional massively multiplayer online role-playing game, World of Warcraft serves as an exemplary context for studying and refining techniques that can be readily adapted to other applications. The research methodology employed in this study involves a systematic analysis of the application's process memory space, with a focus on identifying crucial memory data locations. Furthermore, the investigation entails the identification and preservation of pathways leading to the aforementioned memory data points, ensuring their efficient and viable accessibility. To enable the creation of the virtual entity, the Neuroevolution of Augmenting Topologies technique is employed, which facilitates the generation and intricate development of an artificial neural network—the entity's simulated brain. By utilizing the previously acquired memory data points as sensory inputs, and emulating the entity's responses as inputs within the running process, a comprehensive framework for emulating user behavior is established. The findings presented in this dissertation contribute to the advancement of knowledge in the field of virtual entity creation and artificial intelligence, offering practical implications for a range of applications beyond World of Warcraft.

Keywords Machine Learning, Artificial Intelligence, Neural Network, Neuroevolution, Neuroevolution of Augmenting Topologies, Genetic Algorithm, Reverse Engineering, Input Emulation, Process Memory Scanning, Process Memory Reading, Bot, 3D, MMORPG, PVE, Video Game, Player Emulation, Exploit, Scripting, Automation, Hooking, DLL Injection

Resumo

Esta dissertação explora uma abordagem inovadora para o desenvolvimento de uma entidade virtual ou inteligência artificial capaz de simular o comportamento do usuário no ambiente imersivo e expansivo do jogo de vídeo World of Warcraft. Classificado como MMORPG, este serve como um contexto exemplar para estudar e aprimorar técnicas que podem ser facilmente adaptadas a outras aplicações. A metodologia de pesquisa empregada neste estudo envolve uma análise sistemática do espaço de memória do processo do aplicativo, com foco na identificação de locais cruciais de dados na memória. Além disso, a investigação envolve a identificação e preservação de caminhos que levam aos pontos de dados de memória mencionados anteriormente, garantindo uma acessibilidade eficiente e viável. Para permitir a criação da entidade virtual, é empregada a técnica de Neuroevolução de Topologias Ampliáveis, que facilita a geração e o desenvolvimento intrincado de uma rede neural artificial - o cérebro simulado da entidade. Ao utilizar os pontos de dados de memória adquiridos anteriormente como entradas sensoriais e emular as respostas da entidade como entradas no processo em execução, é estabelecido um framework abrangente para simular o comportamento do usuário. As descobertas apresentadas nesta dissertação contribuem para o avanço do conhecimento no campo da criação de entidades virtuais e inteligência artificial, oferecendo implicações práticas para uma variedade de aplicações além do World of Warcraft.

Palavras-chave Aprendizagem de Máquina, Inteligência Artificial, Rede Neuronal, Neuroevolução, NEAT, Algoritmo Genético, Engenharia Reversa, Emulação de Input, Pesquisa de Memoria, Leitura de Memória, Bot, 3D, MMORPG, PVE, Video Jogo, Emulação de Jogador, Exploit, Scripting, Automação, Hooking, Injecção DLL

Contents

List of Figures

List of Tables

Acronyms

2D Two-dimensional. x, 32

3D Three-dimensional. 2, 7, 8

AI Artificial Intelligence. 3, 4, 6, 8, 17, 27, 32, 36, 37, 38, 39, 40, 41, 43, 55, 68, 70, 72, 77, 85, 86, 87

ANN Artificial Neural Network. 9, 34, 35, 55

API Application Programming Interface. 53, 75, 87

ASLR Address Space Layout Randomization. 41, 52

CPPNs Compositional Pattern Producing Networks. 34

DLL Dynamic-link library. 42, 52, 53, 69, 74

Double-QL Double Q-Learning. 39

DQN Deep Q-Network. x, 12, 38, 39

EA Evolutionary Algorithm. 9, 13, 16, 27, 28, 34, 36, 55, 102, 104

GA Genetic Algorithm. ix, 2, 3, 9, 10, 13, 14, 17, 21, 28, 29, 38, 55, 73, 82, 83

GE Grammatical Evolution. 29

GUI Graphical User Interface. x, xi, 69, 72, 73, 74, 75, 76, 77, 79, 80, 81, 85, 102

HTTP Hypertext Transfer Protocol. 87

HyperNEAT Hypercube-based Neuroevolution of Augmenting Topologies. 34, 35

ID Identification. ix, 21, 26, 33

JSON JavaScript Object Notation. 105

LLM Large Language Model. xi, 88

MAP-Elites Multi-dimensional Archive of Phenotypic Elites. 28, 29

MDP Markov Decision Process. 39

ML Machine Learning. 4, 6, 9, 27, 35, 37, 38, 55

MMORPG Massively Multiplayer Online Role-playing Game. 2, 5, 7, 8, 46

NE NeuroEvolution. x, 15, 16, 17, 30, 31, 34, 35, 36, 38, 55

NEAT Neuroevolution of Augmented Topologies. ix, x, xii, 2, 3, 4, 9, 10, 11, 12, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 28, 31, 32, 33, 34, 35, 36, 37, 40, 43, 46, 55, 60, 65, 70, 71, 73, 74, 75, 76, 77, 78, 80, 81, 83, 85, 86, 87, 99, 100, 101, 105

NERO NeuroEvolving Robotic Operatives. 37

NLP Natural Language Processing. 87

NN Neural Network. ix, x, 2, 3, 4, 9, 10, 11, 12, 15, 16, 17, 18, 19, 20, 21, 25, 26, 27, 30, 31, 32, 33, 34, 35, 36, 37, 40, 43, 46, 47, 55, 57, 62, 63, 64, 69, 70, 71, 80, 87, 100, 102, 103, 104

NPC Non-player Character. 3, 7, 96

NS Novelty Search. 29

PCG Procedural Content Generation. 31

PVE Player versus Environment. ix, 5, 7

PVP Player versus Player. 5, 7

QD Quality Diversity. 28, 29

QL Q-Learning. 38, 39

RAM Random-Access Memory. 47

ReLU	Rectified Linear Unit. 11
RL	Reinforcement Learning. 4, 27, 33, 38, 39
rtNEAT	Real-time Neuroevolution of Augmenting Topologies. x, 36, 37
SA	Simulated Annealing. 29
SS	Surprise Search. 29
tanh	Hyperbolic Tangent. 11
TWEANN	Topology and Weight Evolving Artificial Neural Network. 17, 26
WoW	World of Warcraft. ix, x, xi, 2, 7, 8, 9, 41, 42, 44, 52, 53, 69, 70, 78, 86, 87, 94, 96, 97

Part I

Introductory material

Chapter 1

Introduction

1.1 Thesis Topic and Approach

This thesis delves into a comprehensive study, analysis, and reverse engineering of an existing application, leading to the development and provision of a tool capable of emulating human-like behavior within a specific environment.

The focal point of this research is the immersive video game **World of Warcraft** (**WoW**), renowned as a **Three-dimensional** (**3D**) **Massively Multiplayer Online Role-playing Game** (**MMORPG**).

The initial phase of this investigation involves a meticulous reverse engineering process to extract pertinent information from the application's memory space. This critical endeavor enables the acquisition of real-time and precise data on the game's dynamic state, which serves as the fundamental input for the **Neural Network** (**NN**).

To engender the intricate logic required to emulate human-like behavior, the author adopts the pioneering technique known as **Neuroevolution of Augmented Topologies** (**NEAT**) Stanley and Miikkulainen [2002]. **NEAT** represents a specialized form of **Genetic Algorithm** (**GA**) that demonstrates proficiency in evolving **Neural Network**s, thereby facilitating the progressive development of the virtual entity's cognitive capabilities.

The resultant output generated by the **NN** is seamlessly integrated back into the game environment through the utilization of AutoIt, an embeddable freeware scripting language that bears resemblance to the BASIC programming language. AutoIt assumes the role of a powerful automation tool, effectively enabling the provision of a lifelike experience to unsuspecting observers.

1.2 Artificial Intelligence and Genetic Algorithms

Artificial Intelligence (**AI**) encompasses a wide range of techniques and programming methodologies employed in various applications. In the realm of computer games, **AI** plays a crucial role in enhancing player experiences by introducing **Non-player Character**s (**NPC**s) with individual goals and predefined actions. **NPC**s such as bankers, merchants, and guards exhibit behaviors that can evolve and become increasingly intricate. For instance, a guard may deviate from its assigned role and transform into a thief, while a thief might demonstrate altruistic behavior by aiding others. Implementing natural and believable behaviors within the game environment remains a challenge, as human players exhibit a multitude of diverse behavioral patterns, making it difficult to ascertain a definitive correct approach.

Genetic Algorithms (**GA**s) serve as effective tools for optimizing **AI** behavior Koza [1994]. **GA**s involve evaluating and comparing similar solutions to refine them using successful solutions as templates. Refinement entails introducing small randomized changes to the **AI**s. **GA**s progress through generations in a systematic exploration of the problem space, with each new generation further refining the solutions. The refinement process requires a reasonable likelihood of generating improved solutions. However, **GA**s can encounter stagnation in their evolutionary process, leading to a plateau in improvement. To address this, the definition of well-performing solutions may be redefined, with innovation being one notable performance indicator. The emphasis placed on legible performance versus other criteria may vary depending on the specific use case.

Neuroevolution of Augmented Topologies (**NEAT**) stands as a noteworthy type of **GA** that specializes in the evolution of **Neural Networks** (**NN**s) to address complex **AI** challenges. **NEAT** introduces a unique approach to evolving neural architectures by allowing the networks to grow in complexity over successive generations. This ability to augment the network's topology enables **NEAT** to discover more intricate and efficient solutions.

In **NEAT**, each **NN** represents an individual solution within a given generation. Initially, these networks start with minimal structure, typically consisting of a few input and output nodes. As the evolution progresses, **NEAT** employs various genetic operators, such as mutation and crossover, to introduce new nodes and connections into the networks. This incremental growth of neural complexity enables **NEAT** to navigate and adapt to increasingly intricate problem spaces.

By combining both historical markings and explicit compatibility measures, **NEAT** ensures the preservation of innovation throughout the evolutionary process. Historical markings allow **NEAT** to track the origin of genes within a population, facilitating crossover operations that maintain innovation. Explicit compatibility measures help assess the similarity between different **NN**s, enabling **NEAT** to determine if a particular network structure can be considered a modification of a previously existing one.

Furthermore, **NEAT** employs a speciation mechanism, wherein similar **NN**s are grouped into species. This mechanism ensures diversity within the population and encourages the exploration of various network architectures. By promoting both exploration and exploitation, **NEAT** strikes a balance between maintaining diversity and converging towards optimal solutions.

NEAT's flexibility in evolving complex **NN** architectures makes it well-suited for addressing heterogeneous objectives. Networks generated by **NEAT** can exhibit diverse structures, enabling them to handle a wide range of **AI** tasks and adapt to varying environments. The ability to dynamically grow and complexify neural topologies grants **NEAT** the potential to discover novel and effective solutions to complex **AI** challenges.

Overall, **NEAT** represents a powerful approach within the realm of genetic algorithms, harnessing the potential of evolving neural networks with augmented topologies. Its unique characteristics make **NEAT** a valuable tool for tackling complex **AI** problems and advancing the field of artificial intelligence.

Additionally, there exist alternative methods that directly compete with the aforementioned approaches. **Reinforcement Learning** (**RL**) serves as an umbrella term for various algorithms within the **Machine Learning** (**ML**) domain. **RL** algorithms learn from rewards and punishments to make optimal sequences of decisions, maximizing future rewards. **RL** finds particular relevance in video games, where well-defined rules and explicit goals are prevalent. Notably, one of the earliest successful **RL** programs was developed for playing checkers Samuel [1959]. While simple tabular algorithms suffice for small problem spaces, more complex scenarios necessitate approximators like **NN**s. **RL** demonstrates efficiency in generating approximate or even optimal solutions by mapping states to decisions, also known as actions Russell and Norvig [2010].

1.3 Video Games as Industry and Software

The video game industry has emerged as a highly profitable and continuously expanding market, with global sales surpassing US$192.7 billion annually Richter [2022]. For reference, during the same fiscal period, the global music industry yielded an approximate revenue of US$25.9 billion, as documented by the International Federation of the Phonographic Industry (IFPI) Richter [2022].

The video game industry's significant economic impact underscores its prominence and renders it a domain that cannot be overlooked. The genesis of this expansive industry can be traced back to 1952 when A.S. Douglas, a professor at Cambridge University, created the world's first video game titled "OXO". This pioneering endeavor introduced **Player versus Environment** (**PVE**) gameplay, as players engaged in a tic-tac-toe simulation against a computer opponent. Subsequently, in 1958, William Higinbotham created the second video game, "Tennis for Two," marking the advent of diverse game genres and the proliferation of both standalone offline and multiplayer online games.

Among the vast array of online multiplayer games, **MMORPG**s occupy a prominent position. These games allow players[1] to develop their own characters within expansive virtual worlds. In such games, players typically undertake quests and engage in repetitive tasks, known as grinding[2] to acquire loot[3] or experience points, thereby strengthening their characters and enabling them to overcome progressively challenging opponents in both **Player versus Environment** (**PVE**) and **Player versus Player** (**PVP**) contexts.

As players accumulate valuable loot and experience, virtual currency gains significant worth. In fact, there was a time when an unofficial exchange rate existed, with numerous listings on platforms like eBay facilitating the trade of online currency for real money and vice versa. This virtual currency sometimes exceeded the value of certain real-life currencies, particularly in economies affected by hyperinflation. Consequently, some individuals resorted to farming[4] virtual currency as their primary source of income Economist [2019] 365 [2021].

However, an inherent vulnerability in video games lies within their software foundation. By manipulating, tweaking, or exploiting the underlying software, players can gain an unfair advantage and engage in cheating practices. Exploitation of online games is commonplace, driven by motivations ranging from financial gain to the desire for a perceived superiority over opponents or the pursuit of alternative means

[1] Often referred to as gamers.

[2] Grinding is the act of collecting something, either an item or coin, continuously or repeatedly.

[3] Virtual items.

[4] Usually referred as grinding.

to level the playing field due to time constraints.

This thesis focuses on the exploitation of online games, viewing them as intriguing testbeds for the application and experimentation of ever-evolving concepts in **Machine Learning**. As **Artificial Intelligence** permeates various aspects of modern life, this research offers a glimpse into potential future scenarios and may aid the industry in enhancing its defenses against emerging threats posed by a new breed of bots. Moreover, this approach may provide the industry with a valuable tool or perspective during player testing phases, wherein techniques such as those explored in this thesis could be employed to uncover bugs or other nuances that may arise during gameplay mechanics testing or unguided playtesting sessions.

By delving into the realm of video games, this thesis seeks to contribute to the academic discourse surrounding the intersection of gaming, **AI**, and software exploitation, offering insights into the potential applications, challenges, and implications within this dynamic and rapidly evolving industry.

1.4 Target Application - World of Warcraft

World of Warcraft (**WoW**) is a prominent and widely acclaimed **Massively Multiplayer Online Role-playing Game** (**MMORPG**) developed and released by Blizzard Entertainment in 2004 Entertainment [2004]. As with other **MMORPG**s, **WoW** allows players to create personalized character avatars and immerse themselves in an expansive virtual world, presented from a third or first-person perspective. Within this virtual realm, players embark on adventures, combat various monsters, undertake quests, and engage in interactions with **Non-player Characters** (**NPC**s) as well as other players. While **WoW** promotes cooperative gameplay, enabling players to form alliances and undertake shared quests, it also offers a solo play experience for those who prefer independent exploration and progression. The game features a diverse range of activities, including dungeon exploration and both **Player versus Player** (**PVP**) and **Player versus Environment** (**PVE**) combat encounters. However, the core focus of **WoW** revolves around character advancement, whereby players accumulate experience points to enhance their character's abilities, acquire in-game currency to obtain superior equipment, and master various gameplay systems.

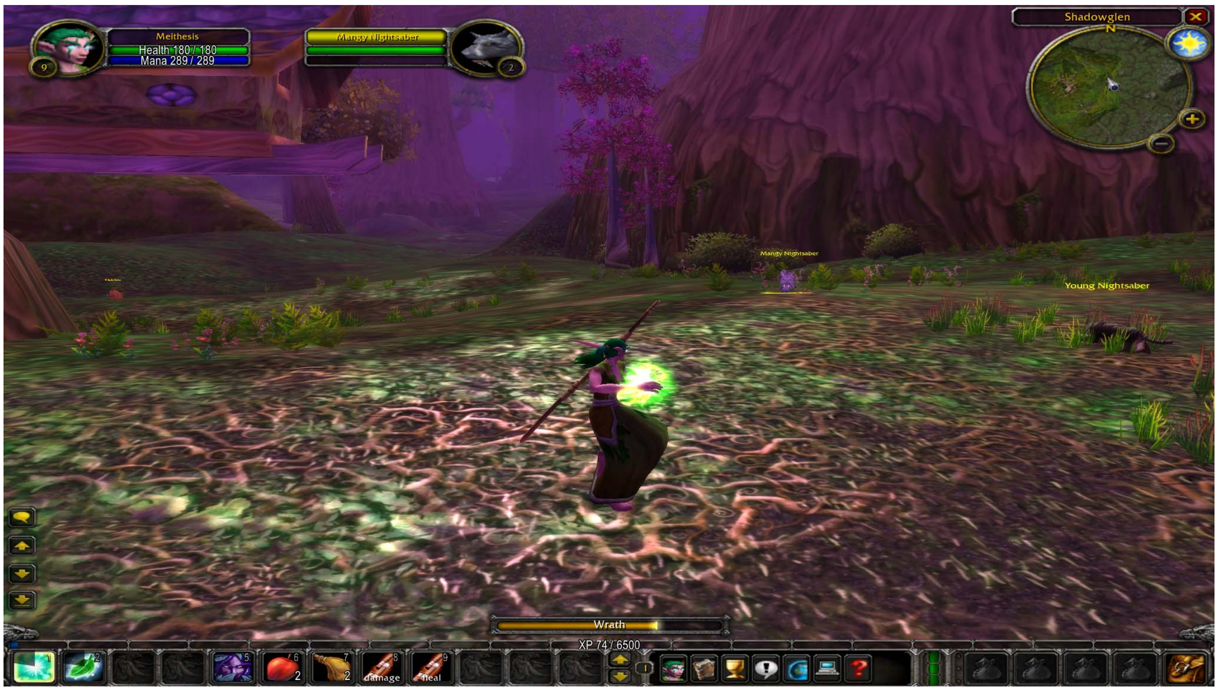


Figure 1: Example of a player engaging in **PVE** combat on **WoW**.

The underlying motivation of this thesis is to devise an artificial entity that emulates the behavior of a player, enabling it to systematically engage in combat with in-game creatures within the **Three-**

dimensional (**3D**) environment of **WoW**. By leveraging this external software, the aim is to achieve a continuous accumulation of in-game experience points, subsequently benefiting the controlled character. This endeavor entails exploring the mechanics and intricacies of **WoW**, identifying strategic patterns, and formulating an algorithmic approach to optimize the efficiency of monster slaying, ultimately augmenting the character's progress within the game world.

Through the academic exploration of **WoW** as the target application, this thesis endeavors to contribute to the scholarly discourse surrounding **AI**-driven automation in the context of **MMORPG**s. By addressing the technical challenges and ethical considerations associated with the deployment of bot-like entities within virtual environments, this work aims to shed light on the implications, opportunities, and potential ramifications of such automated systems within the gaming landscape.

Chapter 2

State of the Art

The thesis leverages the scope of **Machine Learning** and **Evolutionary Algorithms** (**EAs**) with the aim to develop and evolve strategies for achieving automated gameplay in **World of Warcraft**. This section provides a comprehensive background on the fundamental concepts and theories that form the basis of these algorithms, thereby facilitating a better understanding of the subsequent sections.

2.1 Artificial Neural Networks Concepts

An **Artificial Neural Network** (**ANN** or **NN**) is designed to mimic the behavior of the human brain by utilizing interconnected nodes, also known as neurons. These neurons process and transmit information through a network of connections. The configuration of these connections, known as the network's topology, determines the **NN**'s ability to analyze input data and make predictions. During training, the weights and biases of these connections are adjusted to optimize the performance of the **NN** Dreyfus [2005] Kacprzyk [2016].

In addition to training **NN** using traditional methods, another approach to optimizing their performance is through the use of **Evolutionary Algorithms** (**EAs**) or **Genetic Algorithms** (**GAs**), such as **Neuroevolution of Augmented Topologies** (**NEAT**). **EAs** and **GAs** are inspired by the process of natural evolution and mimic the principles of genetic variation and selection.

In the context of this thesis, **NEAT** can be utilized to evolve **NNs** that demonstrate intelligent gameplay strategies in **World of Warcraft**, enhancing the performance of the bot-like entities in continuously slaying monsters and accumulating in-game experience points.

In the context of **NEAT**, the **GA** operates by maintaining a population of **NNs**, each representing a solution to a given task. These **NNs** have their own unique structure, referred to as their topology, which includes the arrangement of nodes and connections. Initially, the population consists of randomly generated **NNs** with simple structures.

During the evolution process, the **NN**s are subjected to mutation and potential crossover. Mutation involves making small random changes to the structure and weights of individual **NN**s, introducing variation into the population. This variation allows for exploration of different configurations and strategies. Crossover, on the other hand, involves combining the characteristics of two or more parent **NN**s to create offspring with a new set of traits.

The performance of each **NN** in the population is evaluated based on its ability to accomplish the desired task, such as successfully playing the game or solving a specific problem. The fittest individuals, those with the best performance, are selected to reproduce and pass their genetic information to the next generation. This process is known as "survival of the fittest" Darwin [1859], as the better-performing individuals have a higher chance of contributing their genetic material to the next generation.

Over successive generations, the population evolves through the iterative application of mutation, crossover, and selection. Through this process, **NEAT** seeks to discover and refine effective **NN** topologies for the given task, gradually improving their performance.

By combining the principles of **GA**s with **NN**s, **NEAT** provides an alternative approach to training **NN**s, allowing for the evolution of their structures to better suit the task at hand.

In the simplest form of a **NN**, input data is provided as decimal numbers arranged in an array-like structure. The number of input values corresponds to the number of input nodes in the **NN**. After performing computations, the **NN** generates an array of decimal values as output, typically with the same size as the number of output nodes. To facilitate computation, the input and output values are often clamped or normalized within a specific range, such as [-1, 1] Shao et al. [2020].

The connections between nodes in a **NN** consist of directed weighted edges. These edges propagate values through the network, allowing information to flow from input nodes to output nodes. In a basic **NN**, cycles are not allowed, meaning there are no feedback connections. The three main types of nodes in a **NN** are input nodes, hidden nodes, and output nodes.

Input nodes store the initial input values, while the output nodes receive the final output values after the computations. The hidden nodes, as the name suggests, are not directly connected to the input or output but are located between them. Hidden nodes play a critical role in capturing and processing intermediate information, enabling the **NN** to learn complex patterns and make accurate predictions.

Figure 2 illustrates a generalized structure of a **NN**, showcasing the arrangement of input, hidden, and output nodes, as well as the connections between them.

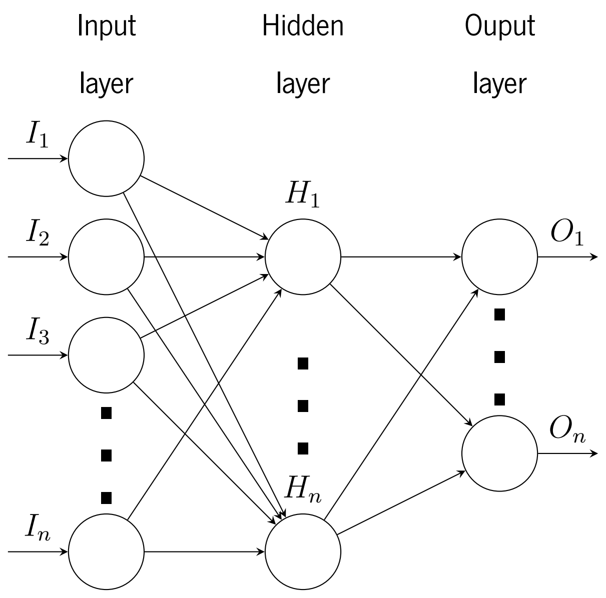


Figure 2: Example of a **Neural Network** Structure or Topology.

Activation functions play a crucial role in **NN**s as they determine how the nodes process and produce output signals based on the input data P Sibi and Siddarth [2013]. When values from multiple edges are combined into a node, the resulting sum can sometimes become large, which may not be desirable. Activation functions help mitigate this issue by clamping the output of the nodes within specific bounds, ensuring the stability and effectiveness of the **NN**'s computations.

There are several commonly used activation functions, including the sigmoid, the **Hyperbolic Tangent** (**tanh**), and the **Rectified Linear Unit** (**ReLU**). The sigmoid function maps the input values to a sigmoid-shaped curve between 0 and 1, while the **tanh** function maps the values to a curve between -1 and 1. The **ReLU** function, on the other hand, sets the output to zero for negative inputs and keeps the positive inputs unchanged. These activation functions offer different properties and can be selected based on the specific requirements of the problem at hand.

In the context of the **NEAT** algorithm, activation functions play a vital role in shaping the behavior and performance of the evolved **NN**s. The **NEAT** algorithm aims to optimize the structure and weights of the **NN**s by genetic evolution, and activation functions are an integral part of these **NN**s. They influence how the nodes within the **NN**s process information and make predictions.

The integration of activation functions in **NEAT** occurs during the evaluation and propagation of the **NN**s. During evaluation, input values are passed through the **NN**, and the activation functions determine the transformed output values at each node. These transformed values are then propagated through the network to generate the final output.

The choice of activation functions depends on the nature of the problem being addressed. For classification tasks, where discrete labels are assigned to the inputs, the activation functions help in selecting the output node with the highest value, representing the most appropriate label for the given inputs. In regression tasks, where continuous values are sought, the activation functions allow mapping the outputs to larger numbers, potentially by scaling the output values to the desired range.

The use of activation functions and floating-point arithmetic in general is costly. In the **NEAT** algorithm **NN**s are computed with a time complexity of $O(|V|+|E|)$. There are however better ways to process **NN**s. **NN**s can be represented as matrices, which in turn can be multiplied efficiently using open-source math libraries Gaël Guennebaud et al. [2010]. The code for **DQN** is one such example that makes use of matrix multiplications internally.

2.2 Evolutionary Algorithms

2.2.1 Genetic Algorithms

In the field of computer science and operations research, **Genetic Algorithms** (**GA**s) are metaheuristic algorithms inspired by the principles of natural selection. **GA**s belong to the broader class of **Evolutionary Algorithms** (**EA**s) and are widely utilized for solving optimization and search problems. These algorithms employ biologically inspired operators such as mutation, crossover, and selection to generate high-quality solutions.

GAs are particularly suitable for tackling optimization problems where finding the optimal solution would be computationally expensive or time-consuming. They are capable of providing estimations of solutions that are sufficiently good to handle such problems efficiently.

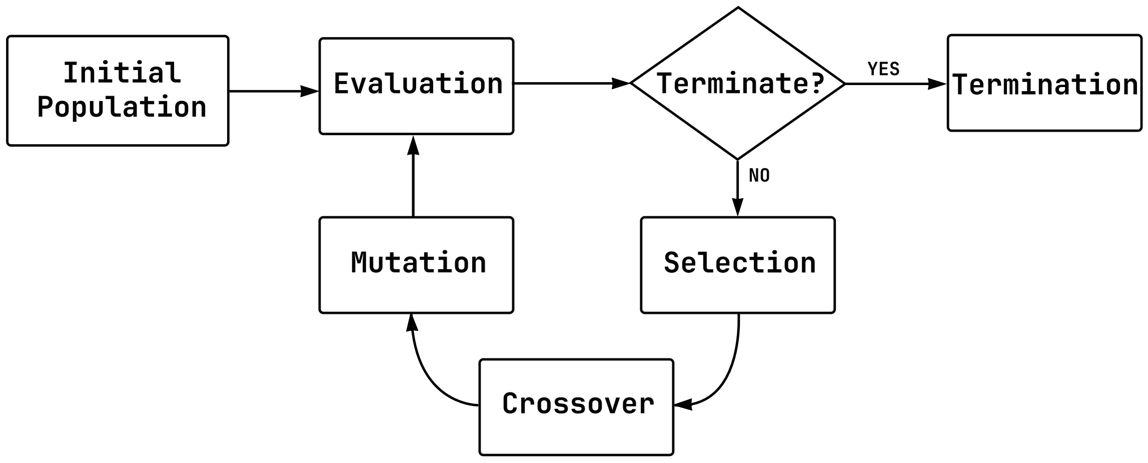


Figure 3: Schematics of a standard **Genetic Algorithm**.

GAs can be applied to various types of optimization problems, including both single-objective and multi-objective scenarios Mohammadi et al. [2017]. A general pattern can be observed in the execution of **GA**s, as depicted in Figure 3. The optimization process revolves around the iterative execution of three fundamental steps: evaluation, selection, and mutation.

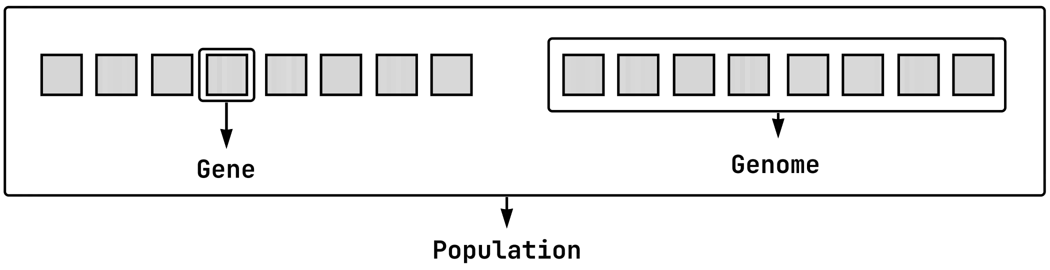


Figure 4: **Genetic Algorithm** genetic encoding.

During evaluation, all candidate solutions[1] in the population are tested, often in a simulated environment, and assigned fitness values based on their performance. The fitness values serve as a measure of the quality or suitability of each solution.

Selection involves choosing a subset of solutions from the population to form the next generation. Solutions with higher fitness values are more likely to be selected, as they exhibit superior performance. In some variations of **GA**s, uniqueness or distinctiveness is considered during selection. Solutions are categorized into species based on their distinctive qualities, enabling the promotion of innovation and diversity within the population. This approach helps avoid getting trapped in local optima.

Mutation is a crucial operator in **GA**s, which involves making small random modifications to selected solutions. It introduces exploratory changes that can potentially lead to improved solutions. In some cases, crossover is also performed, where characteristics from two parent solutions are combined to create a new solution.

Implementation details of **GA**s can vary depending on the specific problem and requirements. Parameters such as mutation rates, divergence search, and complexification strategies may be fine-tuned to enhance the performance and convergence of the algorithm.

[1] i.e. Each individual.

2.2.2 NeuroEvolution Challenges Addressed by NEAT

Competing Conventions Problem

The Competing Conventions Problem, also referred to as the Permutations Problem Radcliffe [1993], poses a significant challenge in **NeuroEvolution** Montana and Davis [1989], particularly in the context of weight optimization problems with **NN**s encoded in different ways. This problem arises when different genomes encoding **NN**s produce the same solution but with different permutations or orders of certain elements.

To illustrate this problem, let's consider two **NN**s with their respective genome encodings: [A, B, C] and [C, B, A]. These genomes represent the order of hidden nodes in the **NN**s. When these networks undergo crossover or mating, the resulting offspring may have mixed or incorrect orderings, such as [C, B, C] or [A, B, A]. These offspring representations have lost the correct ordering of nodes present in both parents, resulting in suboptimal or non-functional solutions.

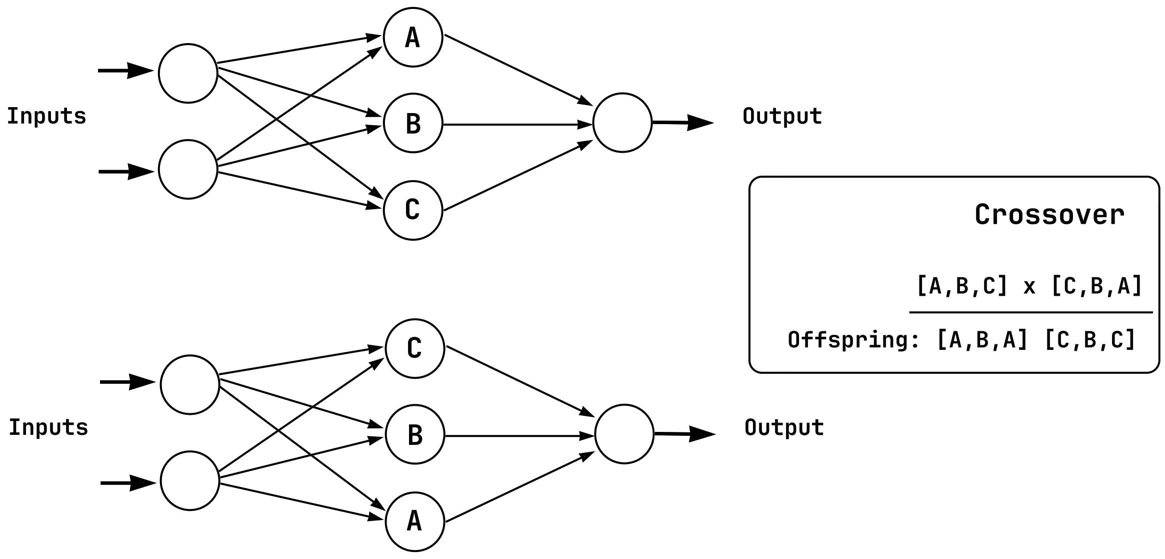


Figure 5: Illustration of a competing conventions problem. In this scenario, two neural networks perform identical computations, despite their hidden units being arranged differently and represented by distinct genetic information, rendering them unsuitable for standard crossover operations. The figure illustrates that when applying single-point recombinations, both methods fail to include one of the three essential elements from each solution. Adapted from Stanley and Miikkulainen [2002].

The consequence of this issue is that these bad solutions, which lack the correct ordering of nodes, increase the computational time needed for the evolution process unnecessarily. It becomes crucial to address the competing conventions problem in **NeuroEvolution** (**NE**) to ensure that the evolution process can effectively converge towards optimal solutions.

By finding a solution to the competing conventions problem, **NE** algorithms can prevent the loss of vital information during crossover operations and avoid generating offspring with incorrect or mixed permutations. This enhances the efficiency of the evolutionary process by reducing the search space and promoting the preservation of meaningful and functional genetic information within the population.

Topological Innovation Problem

The Topological Innovation Problem is a significant challenge faced by neuroevolutionary algorithms when introducing new structures to **NN**s Stanley and Miikkulainen [2002]. When a new connection or node is added to a network, it can impact the optimization process in two main ways.

Firstly, larger **NN**s with increased structures tend to optimize at a slower rate compared to smaller networks. The complexity of the network increases with the addition of new connections or nodes, leading to a more extensive search space. This increased complexity often requires more time and resources for the **EA** to converge towards optimal solutions.

Secondly, when a new connection is added to a network, the fitness value of the network initially decreases before the connection weights are properly optimized. This decrease in fitness is because the newly augmented structure may not contribute positively to the overall network performance initially. The weights associated with the new connection need to be optimized through further iterations of the evolutionary process to improve the network's fitness.

Due to the initial decrease in fitness caused by the addition of new structures, the newly augmented structure is less likely to survive beyond a single generation. The lower fitness value associated with the newly introduced structure makes it more susceptible to being removed or overwritten during the selection and reproduction process of the evolutionary algorithm.

Addressing the topological innovation problem is crucial for neuroevolutionary algorithms to effectively incorporate new structures into **NN**s. By employing mechanisms such as speciation, which protects and allows for optimization within individual species, the algorithm can provide the necessary time and opportunities for the newly added structures to evolve and demonstrate their usefulness. This helps overcome the initial fitness decrease and allows the augmented structures to potentially contribute to the network's performance in subsequent generations.

2.3 NEAT - Neuroevolution of Augmented Topologies

Neuroevolution of Augmented Topologies (**NEAT**) aims to achieve a delicate balance between the fitness of evolved solutions and the diversity among them by dynamically adjusting both the weighting parameters and structures of **NN**s. This algorithm is built upon three key techniques: tracking genes with historical markers, speciation, and complexification, complemented by the fundamental genetic operators of mutation and crossover Stanley and Miikkulainen [2002].

NeuroEvolution (**NE**) represents a type of **Genetic Algorithm** (**GA**) that leverages **Genetic Algorithms** (**GAs**) as the foundation for its solution policies. It finds applications in diverse fields such as general game playing, evolutionary robotics, and artificial life EvoStar [2019]. Particularly effective in simulated environments, **NEAT** operates within the framework of **GA**s, displaying robust performance. This section provides an in-depth exploration of **NEAT**'s operational principles, accompanied by practical examples.

NEAT possesses the versatility to be applied to various problem domains Stanley and Miikkulainen [2002]. It is well-suited for both discrete and continuous environments. Despite its general applicability, **NEAT** operates on a fundamentally straightforward basis. Often, a simple reward function suffices as the algorithm incrementally learns and adapts during its execution. For instance, a **NEAT** reward function may focus on monitoring a single parameter's behavior in a simulation, rather than attempting to capture complex combinations of parameters over time.

NEAT generates random **NN**s that serve as policies for **AI** agents. The primary objective is to enhance these **NN**s by iteratively modifying their topologies and adjusting the connection weights. Similar to many **GA**s, **NEAT** excels at rapidly approximating solutions, making it particularly valuable when confronted with expansive state-spaces where finding an optimal solution within a reasonable timeframe is challenging. **NEAT** places significant emphasis on safeguarding and promoting innovation. This entails preserving seemingly suboptimal **AI**s with low fitness scores if they possess sufficiently unique characteristics. The determination of what qualifies as uniqueness remains an active research question. At its core, innovation in **NEAT** signifies that an **AI**'s **NN** exhibits a distinctive structure compared to other concurrently evolving **AI**s. Innovation serves as a guiding principle within the evolutionary process, propelling it towards viable solutions in complex problem spaces by helping overcome local optima.

In **NEAT**, both the network topology and connection weights evolve concurrently[2], enabling the algorithm to explore and discover optimal network architectures. However, as the complexity of the problem

[2] Essentially classifying it as a **TWEANN** algorithm.

increases, the likelihood of finding an optimal network diminishes. Notably, **NEAT** operates without any prior information about rewards; instead, it is the experimenter's responsibility to make informed judgments regarding score assignment following tests. Rewards are assigned only once to the entire solution, rather than individual components.

By integrating historical markers, speciation, and complexification mechanisms, **NEAT** possesses the ability to dynamically adapt **NN** structures and weights. This fosters innovation and facilitates navigation through intricate problem spaces. A comprehensive understanding of **NEAT**'s principles and characteristics contributes to a deeper appreciation of its efficacy and potential for addressing a wide array of computational challenges.

Historical Markings

Historical markings play a crucial role in neuroevolutionary algorithms such as **NEAT** by providing essential ancestral information about genes. These markings serve to establish the structural relationships between genes, indicating whether they represent the same underlying **NN** structure, regardless of variations in connection weights. In **NEAT**, a global innovation number is assigned to each newly created gene resulting from structural mutation. These innovation numbers are permanent and immutable, serving as historical markers throughout the evolutionary process Stanley and Miikkulainen [2002].

By utilizing historical markings, **NEAT** ensures the proper identification and preservation of structural information within the population of neural networks. This mechanism allows for accurate tracking of the evolutionary history of genes and enables the successful implementation of crossover operations. Matching genes, characterized by having the same innovation number in both parents, can be selected for crossover, promoting the exchange of genetic material and the potential emergence of novel combinations.

The utilization of historical markings within **NEAT** effectively addresses the challenge of maintaining structural integrity during the evolutionary process, which essentially addresses the competing conventions problem. By assigning unique innovation numbers to new genes and tracking their historical origins, **NEAT** ensures the preservation of ancestral information and enables the exploration of diverse network topologies. This approach promotes the generation of undamaged offspring through crossover and contributes to the algorithm's ability to efficiently search and adapt to complex problem domains.

Genetic Encoding

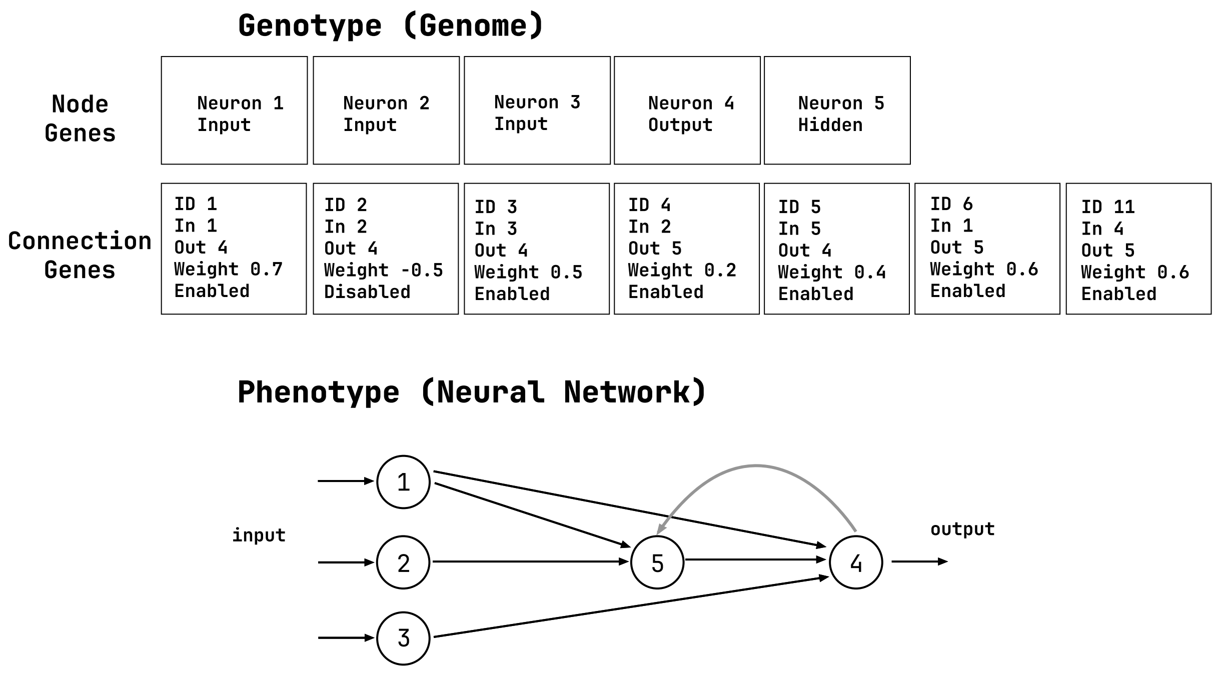

Genotype (Genome)
Node Genes
Neuron 1 Input
Neuron 2 Input
Neuron 3 Input
Neuron 4 Output
Neuron 5 Hidden
Connection Genes
ID 1 In 1 Out 4 Weight 0.7 Enabled
ID 2 In 2 Out 4 Weight -0.5 Disabled
ID 3 In 3 Out 4 Weight 0.5 Enabled
ID 4 In 2 Out 5 Weight 0.2 Enabled
ID 5 In 5 Out 4 Weight 0.4 Enabled
ID 6 In 1 Out 5 Weight 0.6 Enabled
ID 11 In 4 Out 5 Weight 0.6 Enabled
Phenotype (Neural Network)
input
output
1
2
3
5
4

Figure 6: A genotype (genome) to phenotype (**Neural Network**) mapping example. A genotype is depicted that produces the shown phenotype. Note that the second gene is disabled, so the connection that it specifies (between nodes 2 and 4) is not expressed in the phenotype. ID is the global innovation number. Adapted from Stanley and Miikkulainen [2002].

NEAT employs a genetic encoding scheme to represent the **NN** architecture such as depicted in figure 6 Stanley and Miikkulainen [2002]. This encoding scheme consists of two main categories: node genes and connection genes. The node genes represent all the nodes within the network and are categorized based on their type, which can be input nodes, hidden nodes, or output nodes. These node genes provide a comprehensive description of the network's structure.

The connection genes capture the connections present in the network. Each connection gene contains information about the input node, the output node, the weight associated with the connection, an enable/disable bit to indicate the status of the connection, and an innovation number that serves as a unique identifier to track the origin of the gene. The innovation number is particularly important for tracking the historical relationships between genes and allows **NEAT** to identify matching genes during the crossover process.

By using this genetic encoding scheme, **NEAT** effectively represents both the structure and the weights of the **NN** in a compact and easily manipulable form. The combination of node genes and connection genes enables **NEAT** to evolve the network's architecture and weight values simultaneously, allowing for the exploration of a diverse range of network topologies and facilitating the discovery of effective solutions to the given problem.

The genetic encoding scheme of **NEAT** plays a crucial role in the algorithm's ability to dynamically adapt the network's structure through mutations and crossovers, promoting innovation and exploring the solution space effectively. It provides a means to maintain the historical information of genes, which is essential for speciation and preserving diversity during the evolutionary process.

Speciation

Speciation is a key technique employed in **NEAT**, which involves dividing the population of **NN**s into distinct species based on their similarities in network topologies. This approach addresses the challenge of topological innovation and promotes the exploration of diverse solutions within the evolving population. By creating species, **NEAT** provides a niche where **NN** with unique topologies can optimize their structures without being overshadowed by networks with different architectures.

The process of speciation in **NEAT** ensures that individual fitness is compared only among solutions belonging to the same species when selecting individuals for the next generation. This way, **NN** with similar topologies compete with each other, allowing them to evolve their structures effectively. However, species with low average fitness may eventually become extinct as they fail to adapt and improve.

$$\delta = \frac{c1E}{N} + \frac{c2D}{N} + c3 \cdot \overline{W} \tag{2.1}$$

To determine whether two **NN** belong to the same species, a comparison is performed based on the differences in connection weights and the number of dissimilar genes. The difference between two networks is calculated using Equation 2.1 Szudzik [2006], where δ represents the total difference, $c1$, $c2$, and $c3$ are coefficients adjusting the importance of each factor, N is the number of genes in the larger genome, E represents the number of disjoint genes within the same numbered range, D represents the number of excess genes outside that range, and $\overline{W}$ denotes the weighted differences between common gene pairs.

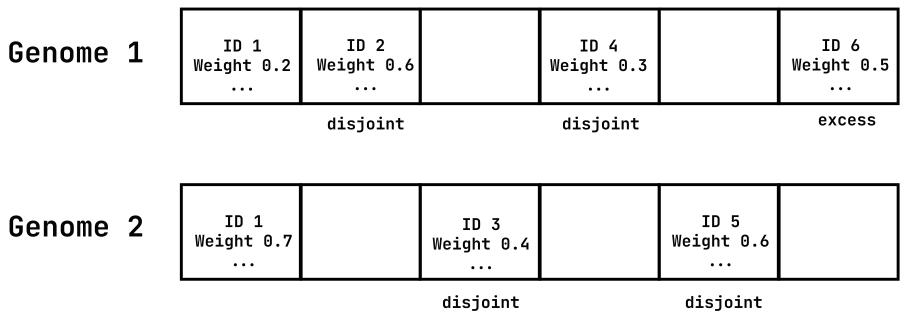


Figure 7: **ID** represents innovation number and the *Weight* represents the connection weight value. In this example, E would be 1, since there is 1 excess gene, gene **ID** 6. D would be 4, since genes **ID** 2 and 4 are not in genome 2, and genes **ID** 3 and 5 are not in genome 1. W would be (|0.7 - 0.2| / 1) = 0.5, since only gene **ID** 1 is shared between the two genomes.

If the calculated difference δ exceeds a fixed threshold, the two compared solutions are classified into separate species. To expedite this process, comparisons are performed once for every possible species, rather than comparing all solutions within the same species against each other. The threshold can be dynamically adjusted to maintain a stable number of species, even as the population of **NN**s scales. If there are too many species, the threshold is increased, and if there are too few species, the threshold is decreased.

Speciation in **NEAT** serves as a protective mechanism for promoting and preserving topological innovation within the evolving population Stanley and Miikkulainen [2002]. One of the challenges in evolving **NN**s is that new changes introduced through mutations may not immediately prove useful or beneficial. These changes need time to undergo further optimization and refinement before their potential benefits can be fully realized.

By organizing the population into species based on similarity in network topologies, **NEAT** ensures that **NN**s with unique structures are not immediately eliminated or overshadowed by networks with more established architectures. This allows innovative solutions, even those initially exhibiting lower fitness scores, to persist and have the opportunity to further optimize their structures over successive generations.

In traditional **GA**s, where all individuals compete directly with each other, new mutations that deviate significantly from the existing solutions may struggle to survive and propagate. However, in **NEAT**'s speciation approach, individual fitness is primarily compared within species rather than across the entire population. This means that **NN**s with novel topologies have the chance to compete and improve against other networks with similar architectures, rather than being directly pitted against well-established solutions.

By providing a separate niche for species with unique topologies, **NEAT** allows for a more thorough exploration of the solution space. It recognizes that innovative changes may not yield immediate improvements in fitness, but they hold the potential to overcome local optima and eventually converge on superior solutions which results in an effective solution when tackling the topological innovation problem. As a result, **NEAT** strikes a balance between preserving diversity and promoting competition, enabling the population to discover and exploit effective structures over time.

Crossover

The crossover process in **NEAT** is possible by the use of historical markings, which enable the identification of genes and their corresponding matches between parents Stanley and Miikkulainen [2002]. Each gene in a genome is assigned an innovation number, which serves as a unique identifier. Genes with the same innovation number in both parents are considered matching genes. However, there are cases where a gene in one parent does not have a corresponding match in the other parent. In such instances, these genes are categorized as either disjoint or excess genes.

Disjoint genes are those with innovation numbers that fall within the range of innovation numbers of the other parent, while excess genes have innovation numbers outside that range. During crossover, matching genes are randomly selected from both parents and included in the offspring. All disjoint and excess genes are always included in the offspring as well.

When it comes to selecting non-matching genes, the more fit parent contributes all of its non-matching genes to the offspring. In cases where both parents have the same fitness value, these non-matching genes are chosen randomly. This approach ensures that both parent genomes have an opportunity to contribute their genetic material to the offspring, promoting diversity and preserving potentially beneficial genetic information.

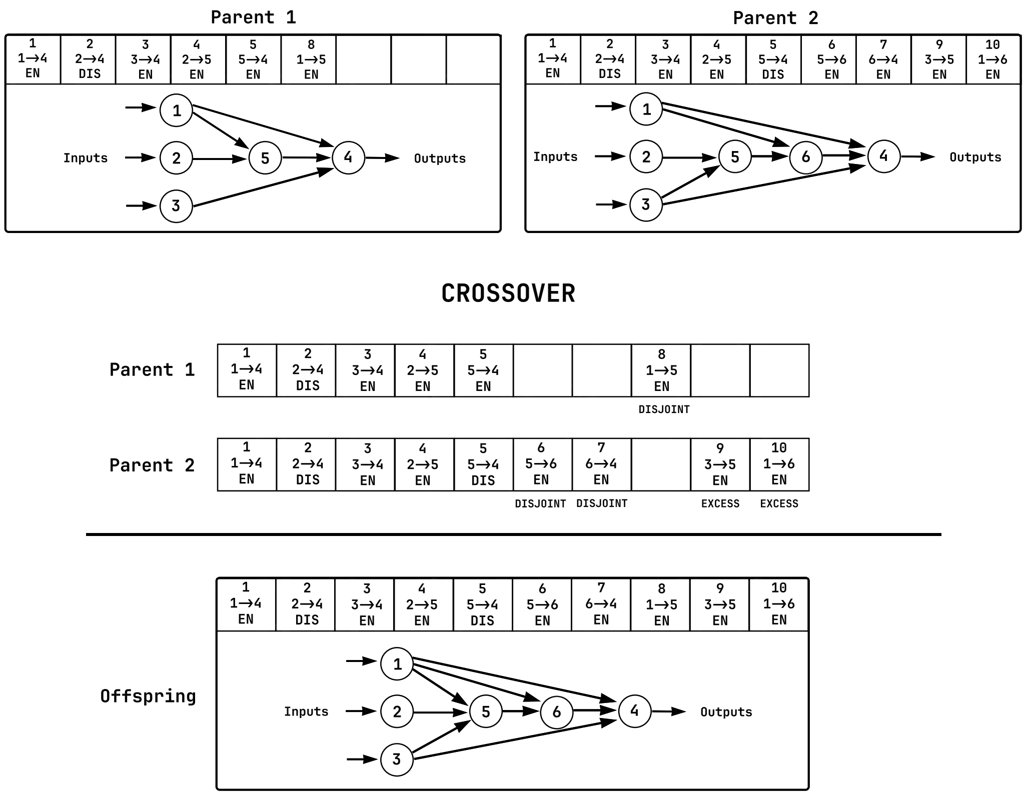

Parent 1
1 1→4 EN | 2 2→4 DIS | 3 3→4 EN | 4 2→5 EN | 5 5→4 EN | 8 1→5 EN
Inputs
Outputs
Parent 2
1 1→4 EN | 2 2→4 DIS | 3 3→4 EN | 4 2→5 EN | 5 5→4 DIS | 6 5→6 EN | 7 6→4 EN | 9 3→5 EN | 10 1→6 EN
CROSSOVER
DISJOINT
DISJOINT
DISJOINT
EXCESS
EXCESS
Offspring
1 1→4 EN | 2 2→4 DIS | 3 3→4 EN | 4 2→5 EN | 5 5→4 DIS | 6 5→6 EN | 7 6→4 EN | 8 1→5 EN | 9 3→5 EN | 10 1→6 EN

Figure 8: Crossover operation example in **NEAT** aligning genetic information across various network topologies through the use of innovation numbers. Adapted from Stanley and Miikkulainen [2002]

Mutation

Mutation plays a vital role in **NEAT** by introducing variation and allowing for exploration of the search space. In **NEAT**, mutation can modify both the connection weights and the network structures of individuals in the population Stanley and Miikkulainen [2002]. The mutation rate, which determines the probability of each connection being perturbed, influences the extent of mutation in each generation.

The choice of mutation type depends on the specific scenario and the desired outcomes. The fundamental mutation is to alter the weight of a gene, but if this modification alone is insufficient, a more significant change in the network topology becomes necessary. When mutation occurs, the decision of what to mutate is typically made randomly to ensure diversity. To achieve a uniform distribution of mutation choices, a weighted random selection can be employed, where all possible choices are assigned the same weight.

It is important to note that an excessive number of mutations within a generation can potentially lead to a significant decrease in performance. If there is only one optimal solution, any mutation is more likely to be detrimental rather than beneficial. Hence, it is advisable to limit the number of mutations per generation to mitigate the risk of performance degradation. **NEAT** encompasses four main types of mutations, namely:

- Creating a new gene between existing nodes. (With the constraint of avoiding network cycles unless it is a recurrent gene);
- Creating a new node. (By replacing a gene with two new ones and placing the node in between);
- Enabling or disabling a gene;
- Adjusting the weight of a gene.

By understanding the various mutation types and their implications, **NEAT** can strike a balance between exploration and exploitation, facilitating the discovery of novel and effective solutions. The careful management of mutation rates and types ensures that **NEAT** can adapt and evolve over generations while avoiding excessive disruption that could hinder performance improvement.

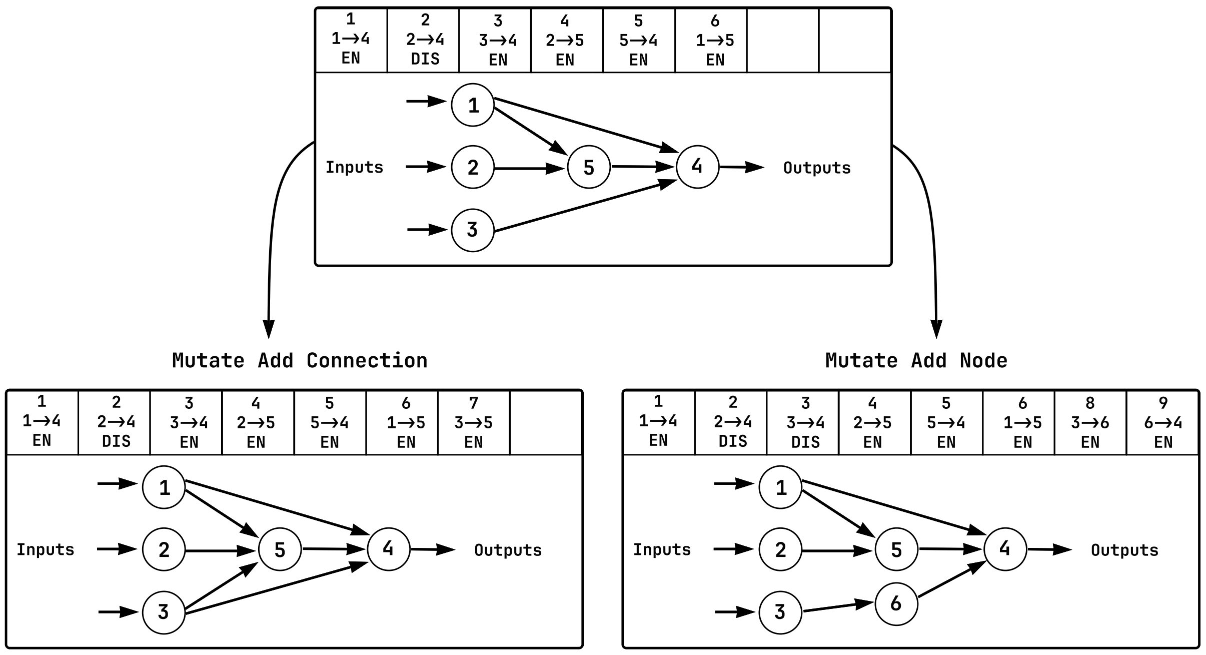


Figure 9: The two types of structural mutation in **NEAT**. The top number in each genome represents the innovation number assigned to that particular gene. "**EN**" indicates the gene expression is enabled. "**DIS**" indicates the gene expression is disabled. Adapted from Stanley and Miikkulainen [2002].

In **NEAT** as shown in the figure 9 the process of adding a new connection and adding a new node involves different genetic operations and strategies to ensure the preservation of genetic information and facilitate effective crossover between genomes.

When adding a new connection through the add connection mutation, a new gene is created. This new gene represents the newly established connection between two existing nodes in the **NN**. It is assigned a unique global innovation number, which helps track the origin of the gene. Additionally, the gene is initially enabled, meaning its expression is active in the network. (Notice on figure 9 how it simply adds a new gene indicating the new connection (3 to 5) and that the gene expression is enabled.)

On the other hand, when adding a new node through the add node mutation, two new genes are created. The first gene represents the connection from the previous node to the newly added intermediate node, while the second gene represents the connection from the intermediate node to the previous destination node. These genes also receive unique global innovation numbers to track their origins. Importantly, the previous gene that directly connected the original nodes is not removed but rather its expression is disabled. By disabling the previous gene (setting its expression to disabled), the information regarding the direct connection between the original nodes is preserved in the genome. (Notice on figure 9 how it needs to create two new genes, one indicating a new connection between the previous origin and the new intermediate node (3 to 6) and the other indicating a new connection between the intermediate node to the previous destiny (6 to 4), also verify how the previous gene isn't removed but instead its expression is set to disabled (gene **ID** 3)).

This approach of adding new genes and disabling previous ones guarantees that no genetic information is lost during the mutation process. It allows for seamless crossover to occur between genomes during the reproductive phase of the algorithm. Crossover can recombine genes from different parents, including genes representing connections and nodes, while maintaining the structural integrity of the neural networks encoded by the genomes. This preservation of genetic information and seamless crossover mechanism enable the exploration and propagation of beneficial structural modifications in the population over generations.

Complexification

In the case of **NEAT**, the algorithm starts with a minimal initial topology, typically consisting of input and output nodes directly connected without any hidden nodes. Through a combination of mutation and crossover operations, **NEAT** introduces structural changes to the network over successive generations, allowing for the evolution of more complex topologies Stanley and Miikkulainen [2002]. This process involves the addition and removal of nodes and connections, enabling the network to adapt and develop more sophisticated architectures that better capture the problem at hand.

By evolving both the weights and the topology of the **NN**, **NEAT** explores a larger search space, seeking to find optimal solutions to a given problem. The algorithm leverages the principles of natural evolution, such as fitness evaluation and selection, to guide the evolutionary process towards solutions that exhibit improved performance and adaptability.

NEAT's **Topology and Weight Evolving Artificial Neural Network** (**TWEANN**) approach offers the advantage of dynamically adjusting the neural network's structure to match the problem complexity. This capability allows **NEAT** to address a wide range of problem domains, from simple to highly complex, and discover effective network architectures that optimize performance.

2.4 Related Work

In the realm of **Artificial Intelligence** (**AI**), extensive research has been conducted, exploring various aspects related to machine learning and video games. This section aims to provide an overview and summary of relevant work that intersects with these domains. The related work discussed herein sheds light on crucial considerations and special circumstances within simulations, which can significantly impact the performance of **AI** systems. By leveraging the insights gained from these contributions, further advancements can be made to enhance the work presented in this thesis.

Numerous research studies have investigated the application of **ML** techniques in video games, with a particular focus on training intelligent agents to play and excel in game environments. These studies explore different aspects of **AI**, such as **Reinforcement Learning**, **Evolutionary Algorithm**s, and **Neural Network**s. By utilizing these methodologies, researchers have achieved remarkable results in developing **AI** agents capable of competing against human players and achieving high levels of performance.

Furthermore, specific research contributions have emphasized the importance of addressing unique challenges presented by video game simulations. These challenges include complex decision-making, real-time constraints, and the need for adaptive behavior. By considering these factors, researchers have devised novel approaches and algorithms to overcome these challenges and improve the effectiveness of **AI** in gaming scenarios.

The insights gained from the related work in **ML** and video games provide valuable contributions to the development of **AI** systems. They offer valuable guidance for designing algorithms and methodologies that are tailored to the unique demands of gaming environments. By incorporating these findings, this thesis aims to build upon the existing knowledge and push the boundaries of **AI** capabilities in the context of video game exploitation.

2.4.1 Genetic Algorithms

In the early stages of **Genetic Algorithm** (**GA**) implementations, fitness was primarily based on performance alone. However, more recent approaches have introduced alternative ways to define fitness, with an emphasis on rewarding diverse behavior. This section highlights research that demonstrates how diversity can enhance the performance of **GA**s, along with techniques that can improve the overall structure of evolved networks.

While it has been observed that excluding diversity can accelerate training for simple problems, such an approach, akin to greedy algorithms, may fail to find the optimal solution due to the lack of randomness. For many problems, solely rewarding novel solutions and disregarding fitness altogether is inefficient, as it takes a considerable amount of time to generate meaningful results.

Surprisingly, ignoring fitness has been successfully employed as an evolution strategy in **Neuroevolution of Augmented Topologies** (**NEAT**). In a study conducted by Joel Lehman and Kenneth O. Stanley, comparing the performance of using fitness versus ignoring fitness when training a biped-walker, it was found that the biped walked farther when fitness was ignored Lehman and Stanley [2011].

Algorithms that promote both diversity and efficiency are referred to as **Quality Diversity** (**QD**) algorithms. Within this domain, diversity is considered a prerequisite for using **GA**s Shimodaira [1997] Ursem [2002]. **QD GA**s that reward diversity demonstrate improved resistance to deception Stanley and Lehman [2015] Adam Gaier and Mouret [2019] Cazenille [2019]. Following paths that solely lead to a specific goal can be deceptive, as they may prove ineffective. Effective **QD** algorithms come in various forms, some of which involve reducing the reliance on fitness values and focusing instead on traits such as behavior or solution divergence from historical attempts Mouret and Clune [2015] Brant and Stanley [2017] Stephane Doncieux and Coninx [2019]. By reducing the need for fitness comparisons, it has been observed that if the training environment co-evolves with the solutions, good solutions can still be generated. Gradually adapting the environment's difficulty based on the existing set of solutions helps mitigate local optima Wang et al. [2019].

An effective technique **Multi-dimensional Archive of Phenotypic Elites** (**MAP-Elites**) can be utilized to explore the space of possible solutions in a structured and efficient manner. This technique is often employed in **Evolutionary Algorithm** (**EA**) to discover a diverse set of high-performing solutions.

The core concept of **MAP-Elites** involves dividing the solution space into a grid of cells, where each cell represents a specific combination of characteristics or dimensions. The best-performing solution (elite) is stored for each cell in the grid. As the algorithm progresses, the grid becomes populated with elites from different cells, enabling a systematic exploration of the solution space.

Compared to other **QD** algorithms, **MAP-Elites** offers advantages in terms of simplicity and reliability Mouret and Clune [2015]. It maps traits to a discrete space, allowing for effective representation and exploration of diverse solutions. For example, in the vehicle routing problem, dimensions such as vehicle type, delivery count, and emissions can be used Neil Urquhart and Hart [2019].

Another approach, known as **Novelty Search** (**NS**), incorporates historical information to direct evolution and prevent redundancy. While it favors divergence, determining which type of divergence should be prioritized remains a challenge Stephane Doncieux and Coninx [2019]. Subsequent algorithms, such as **Surprise Search** (**SS**), build upon the objective of **NS** and demonstrate increased efficiency Yannakakis and Liapis [2016] Daniele Gravina and Yannakakis [2018]. Combining **SS** and **MAP-Elites** in an experiment has resulted in improved performance Daniele Gravina and Yannakakis [2019].

GAs have been employed in evolving behavior trees for the game Super Mario Michele Colledanchise and Ögren [2018]. Behavior trees consist of modules arranged in a specific order, where each module represents an action or specifies how other modules should be executed. The **GA**s used to evolve behavior trees employed a strategy of initially introducing a high mutation rate and gradually decreasing it over time. This concept of adjusting the intensity of mutations is similar to the process of **Simulated Annealing** (**SA**) Scott Kirkpatrick and Vecch [1983]. Notably, the introduction of de-evolution was introduced as a means to remove redundant modules. Experimental results demonstrated that complex trees could be simplified without sacrificing performance.

Additionally, **Grammatical Evolution** (**GE**) approach was developed to map bit-strings to behavior trees for Super Mario Perez et al. [2011]. **GE** exhibited fast mutation rates, as only bits needed to be flipped. However, crossover operations posed a challenge, as an additional module type was required to indicate crossover points. **GE** generated reactive solutions but was not well-suited for path planning.

2.4.2 NeuroEvolution

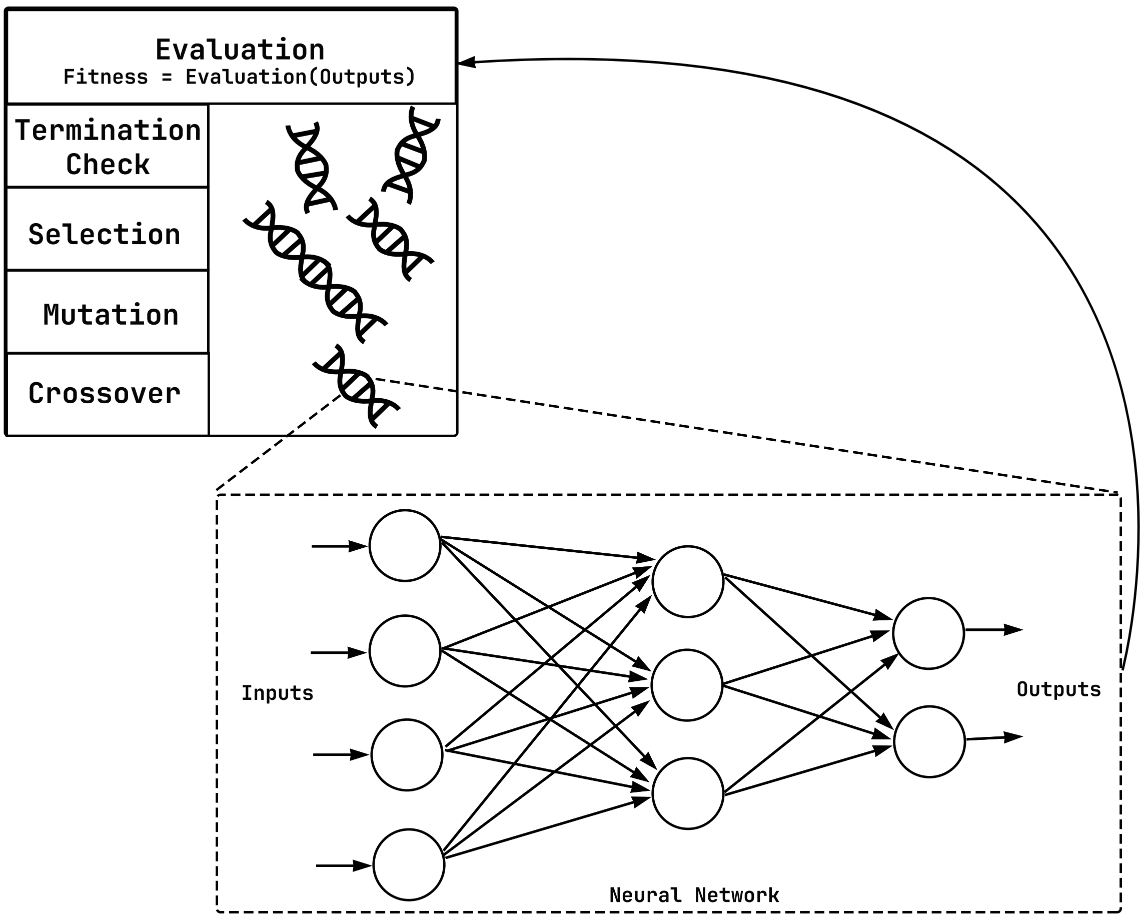


Figure 10: **NeuroEvolution** Loop Schematics.

Neuroevolutionary algorithms typically follow a set of steps to evolve and converge towards improved solutions. These steps are as follows:

1. Initialization: The algorithm begins by generating a random population of individuals. Each individual is encoded in a specific representation, such as a genotype, which is then decoded into a corresponding **NN** structure or phenotype.

2. Evaluation: The next step involves evaluating the performance of each **NN** in the population. The **NN**s are subjected to the environment or task they are designed to solve, and their behavior or output is assessed. Based on the performance, a fitness value is assigned to each genotype in

the population, indicating its quality or success in the given task. The algorithm may terminate if a termination criterion, such as reaching a desired fitness threshold, is satisfied.

3. Reproduction: After evaluation, the algorithm creates a new generation of individuals through the application of selection, mutation, and crossover operators to the population. Selection involves choosing individuals from the current population based on their fitness values, giving preference to those with higher fitness. Mutation introduces random changes to the genetic material of selected individuals, exploring new regions of the search space. Crossover combines genetic material from two or more selected individuals to create offspring with a combination of their characteristics.

These steps of evaluation and reproduction are typically iterated for multiple generations, gradually improving the population's overall fitness. Over time, the algorithm converges towards individuals that exhibit better performance or solutions to the given problem.

By repeating the process of evaluation, selection, mutation, and crossover, neuroevolutionary algorithms explore the search space of possible solutions, adapt the population to the environment, and promote the emergence of more fit individuals. Through this iterative process, the algorithm aims to converge towards optimal or near-optimal solutions for the given task or problem.

In the section 2.3 it was explained what **Neuroevolution of Augmented Topologies** (**NEAT**) was and how it functions. It was made clear that network topologies can both be optimized and made more complex at the same time. It involves training **NN**s by selecting those that perform well in a given environment, rather than updating their weights based on a specific learning rule. A lot of research has been made for algorithms that use similar strategies as those applied in **NEAT**. Some of those research topics are mentioned here.

Different mutation strategies for **NeuroEvolution** (**NE**) have been suggested. As **NN**s grow more complex, changes to certain connections will have a vast impact on the output. One successful mutation strategy handles this problem by scaling edge mutations with respect to the network as a whole. Experiments showed that implementing scaled mutations resulted in a more stable and better-performing training process that enabled simple networks to solve problems in high-dimensional domains that otherwise would require deep or recurrent networks Lehman et al. [2018]. Another interesting notion is Rechenberg's mutation rate control Kramer [2017]. Its mechanisms are simple, increase the mutation rate when some number of consecutive generations have had increased overall fitness or decrease it otherwise. Experiments indicate that this increases performance.

Among other things, **NEAT** has been used in projects by artists for **Procedural Content Generation** (**PCG**). What makes it interesting for **PCG** is that the algorithm at heart doesn't require a goal. It is driven by rewards, and there are no rules for how rewards should be assigned. It keeps mutating innovative solutions if there is no hard-set target.

One example of where **NEAT** can and was implemented is to build an **AI** for the **2D** game Flappy Bird Cordeiro et al. [2019]. In Flappy Bird the goal is to prevent a bird from colliding with obstacles for as long as possible. The bird may only move vertically by allowing it to fall due to gravity or fly upwards by jumping. Obstacles move from right to left in the form of pillars with a hole that enables the bird to potentially fly through. The holes are located at random positions to make navigation harder. If the bird is controlled by a **NN** one may assume that relevant input nodes would contain information about its vertical position, the vertical position of the center of the upcoming hole and the horizontal distance as well as the current vertical velocity of the bird. These numbers would have to be compressed or normalized to scalar values between -1 to 1 before being passed on to the **NN**. In total this translates to 4 input nodes. Lastly, one single output node would be needed, telling the bird to jump or not to jump.

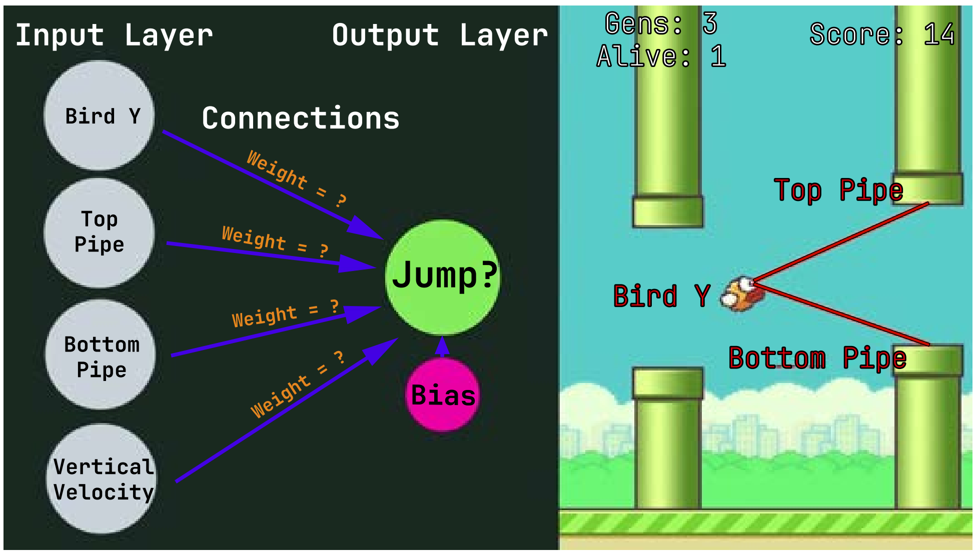


Figure 11: **Neural Network** schema for playing **2D** game Flappy Bird. In **NEAT**, a bias is an input node that is always set to 1.0 and that can connect to any node other than inputs; The bias connections are not always needed depending on the solution. Stanley and Miikkulainen [2002]

To attempt to solve this Flappy Bird paradigm, **NEAT** starts out with a generation of randomly generated **NN**s. All solutions are then tested in simulations and assigned a fitness value based on how far they made the bird travel without dying and, of course, a time limit is also necessary to prevent it from potentially going on indefinitely.

Selected solutions change through mutations and crossover operations. A mutation makes changes to the **NN** itself randomly. An example of performing a mutation is by randomly picking an element from the earlier described array and changing its weight. Crossover combines innovations of two solutions into one child. This is achieved by sorting two arrays from two **NN** representations such that they are ordered by increasing **ID** numbers. If two **ID**s are the same then both **NN**s share a common edge. The child gets a direct copy of the array from the parent with higher fitness. To make it a crossover, the child has a chance to retrieve the weights in the parent with a lesser fitness. Any element with an **ID** that exists in both arrays has a fifty-fifty chance of being from either parent. By repeating this process of selecting good solutions and adding mutations, one will eventually emerge that always makes the bird jump at the right moment.

Starting with a simple topology without hidden layers causes solutions to be less complex than necessary. Furthermore, it is easier to analyze small networks. If one manages to generate a complex **NN** through **RL** and back-propagation having 100 or more nodes that can play Flappy Bird, it is not easy to see which edges make a difference. If on the other hand there is a shorter variant generated with **NEAT** that only has 5 nodes it is easier to see directly from the edges how the bird will behave for specific states without having to test it.

Using **NEAT** for Flappy Bird and similar games offers several advantages. Starting with a simple initial topology allows for easy analysis of the network's behavior. It also prevents unnecessary complexity and facilitates understanding of the important connections between nodes. **NEAT**'s ability to evolve and mutate the **NN**s allows for the discovery of innovative strategies and behaviors that may not have been initially anticipated.

NEAT's capacity to evolve and mutate **NN**s fosters the exploration of novel and unforeseen strategies, unlocking innovative behaviors that may lead to superior game-playing performance. These advantages make **NEAT** the ideal algorithm for the processing layer in this thesis, as it facilitates the development of sophisticated game-playing agents capable of achieving high levels of performance and adaptability in dynamic gaming environments. By starting with a simple initial topology, the behavior of the network becomes more transparent and easier to analyze, promoting a deeper understanding of the critical connections between nodes.

2.4.3 HyperNEAT - Hybercube-based Neuroevolution of Augmenting Topologies

HyperNEAT introduces a novel approach to evolve large-scale **NN** by exploiting geometric regularities in the problem domain Kenneth O. Stanley and Gauci [2009]. By representing the **NN** in a high-dimensional space, **HyperNEAT** leverages the advantages of indirect encoding, allowing for the emergence of more complex and structured **NN**.

Research in **NeuroEvolution** (**NE**), i.e. evolving **ANN**s through **Evolutionary Algorithms**, is inspired by the evolution of biological brains. Because natural evolution discovered intelligent brains with billions of neurons and trillions of connections, perhaps **NE** can do the same. Yet while **NE** has produced successful results in a variety of domains, the scale of natural brains remains far beyond reach. **Hypercube-based Neuroevolution of Augmenting Topologies** (**HyperNEAT**) aims to narrow that gap. **HyperNEAT** employs an indirect encoding called connective **Compositional Pattern Producing Networks** (**CPPNs**) that can produce connectivity patterns with symmetries and repeating motifs by interpreting spatial patterns generated within a hypercube as connectivity patterns in a lower-dimensional space Kenneth O. Stanley and Gauci [2009]. The advantage of this approach is that it can exploit the geometry of the task by mapping its regularities onto the topology of the network, thereby shifting problem difficulty away from dimensionality to underlying problem structure. Furthermore, connective **CPPNs** can represent the same connectivity pattern at any resolution, allowing **ANN**s to scale to new numbers of inputs and outputs without further evolution. The main conclusion is that the ability to explore the space of regular connectivity patterns opens up a new class of complex high-dimensional tasks to **NE**. Kenneth O. Stanley and Gauci [2009]

In short, **HyperNEAT** is an extension of **NEAT** that is based on a theory of representation that hypothesizes that a good representation for an **Artificial Neural Network** should be able to describe its pattern of connectivity compactly. This kind of description is called an encoding. The encoding in **HyperNEAT** is called **Compositional Pattern Producing Networks** and is designed to represent patterns with regularities such as symmetry, repetition, and repetition with variation. Thus **HyperNEAT** is able to evolve **ANN**s with these properties. The main implication of this capability is that **HyperNEAT** can efficiently evolve very large **ANN**s that look more like neural connectivity patterns in the brain (which are repetitious with many regularities, in addition to some irregularities) and that are generally much larger than what prior approaches to neural learning could produce.

The other unique and important facet of **HyperNEAT** is that it actually sees the geometry of the problem domain. It is strange to consider, but most **NE** algorithms (and most neural learning algorithms in general) are completely blind to domain geometry. For example, when a checkers board position is input into an **ANN**, it has no idea which piece is next to which piece. If it ever comes to understand the board geometry, it must figure it out for itself. In contrast, when humans play checkers, we know right away the geometry of the board, we do not have to infer it from hundreds of examples of gameplay. **HyperNEAT** has the same capability. It actually sees the geometry of its inputs (and outputs) and can exploit that geometry to significantly enhance learning. To put it more technically, it computes the connectivity of its **NN**s as a function of their geometry.

One implication of **HyperNEAT**'s ability to exploit geometry is that it gives the user a completely new kind of influence over **ANN** learning. The user can now describe the geometry of the domain to **HyperNEAT**, which means there is room to be creative. If someone believes that a domain can be described best in a different geometry, it can be tested using this technique. Thus it opens up a new kind of research direction for **ANN**s. Stanley et al. [2015]

Experiments have been made using both **NEAT** and **Hypercube-based Neuroevolution of Augmenting Topologies** (**HyperNEAT**) on the keepaway soccer game which is a **ML** benchmark problem that requires high-level strategic decision-making and has a fractured decision space where the keepers are charged with preventing the takers from taking the ball for as long as possible where results have shown that although **HyperNEAT** improves upon **NEAT** for a relatively simple fractured problem that benefit is simply not enough to compensate for that fact that **HyperNEAT** is very slow, even on a multi-core processor Jessica Lowell and Grabkovsky [2011].

2.4.4 rtNEAT - Real-time Neuroevolution of Augmenting Topologies

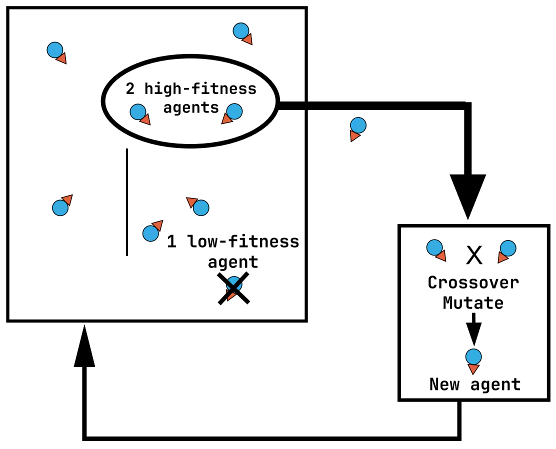


Figure 12: The main replacement cycle in **rtNEAT**.

rtNEAT introduces the concept of real-time **NeuroEvolution**, allowing the evolution of **Neural Network** to occur dynamically during the execution of the **AI** system. This real-time adaptation enables the **AI** to continually adapt and respond to changing environments, making it well-suited for applications that require real-time decision-making Kenneth O. Stanley and Hoang [2008].

Usually in **EA**s, the entire population is replaced at each generation in **Neuroevolution of Augmented Topologies** (**NEAT**). However, in a real time game or a simulation, such a step would look incongruous since every agent's behavior would change at once. In addition, behaviors would remain static during the large gaps between generations. Instead, in **Real-time Neuroevolution of Augmenting Topologies** (**rtNEAT**), a single individual is replaced every few game ticks. One of the worst individuals is removed and replaced with a child of parents chosen from among the best. This cycle of removal and replacement happens continually throughout the game as seen on figure 12 and is largely invisible to the player. This replacement cycle presented a challenge to **NEAT** because its usual dynamics, i.e. protection of innovation through speciation and complexification, are based on equations that assume generational replacement. In **rtNEAT**, these equations are changed into probabilistic expressions that apply to a single reproduction event Kenneth O. Stanley and Bryant [2005a]Kenneth O. Stanley and Bryant [2005b]. The result is an algorithm that can evolve increasingly complex **NN**s fast enough for a user to interact with evolution as it happens in real time. A new genre of games was made possible by **rtNEAT**, where the player trains agents in real time[3]. Kenneth O. Stanley et al. [2003]

[3] One of which is called NERO - Neuro Evolving Robotic Operatives.

One project worth mentioning is called **NeuroEvolving Robotic Operatives** (**NERO**). **NERO** is a result of an academic research project in **Artificial Intelligence**, based on the **Real-time Neuroevolution of Augmenting Topologies** (**rtNEAT**) algorithm. It is also a platform for future research on intelligent agent technology. It is a new kind of **Machine Learning** game being developed at the Neural Networks Research Group, Department of Computer Sciences, University of Texas at Austin. The goals of the project are to demonstrate the power of state-of-the-art **ML** technology, to create an engaging game based on it, and to provide a robust and challenging development and benchmarking domain for **AI** researchers. Kenneth O. Stanley et al. [2003]

In **NERO** the player needs to train robots to battle other robots, in order to colonize a defended but uninhabited planet. The behavior of each robot is controlled by an artificial **NN**, i.e. a "brain" and this brain is scored on how well the robot performs within a given amount of time. Based on these scores (or rewards), the **Neuroevolution of Augmented Topologies** (**NEAT**) then modifies the brains so that they perform better in the future.

The observations acquired from **rtNEAT**, particularly **NERO**, significantly enhance and enrich the main thesis by providing valuable insights into the dynamic adaptability and evolving behaviors of intelligent agents. Through real-time learning and decision-making capabilities, **NERO**'s implementation offers a deeper understanding of the agent's responses to changing conditions, enabling the identification of innovative strategies that optimize performance in challenging environments. These findings bolster the main thesis, showcasing the effectiveness of **rtNEAT** in achieving sophisticated and optimized behaviors in the context of the study's domain.

The **NERO** video game demonstrates that evolutionary computation techniques such as **rtNEAT** are now sufficiently flexible and robust to support real-time interactive learning in challenging sequential decision tasks. While game playing is a significant application on its own, these techniques can also be seen as a significant step towards building intelligent adaptive agents for human environments, such as training environments, robot controllers, and intelligent assistants.

2.4.5 Reinforcement Learning

Reinforcement Learning (**RL**) has risen to prominence as a powerful and widely applied technique in the development of **AI** systems Sutton and Barto [2018]. With its ability to learn from interactions with complex environments and make informed decisions based on rewards, **RL** has become a pivotal area in **ML**, and signaled as a popular alternative to **Genetic Algorithm**s (**GA**s) and **NeuroEvolution**.

RL has been applied to various video games, including the widely known Super Mario Bros, to train **AI** agents and achieve successful performance. In one Super Mario **AI** competition, a plain **Q-Learning** (**QL**) algorithm was evaluated in a simplified environment, demonstrating competitive results compared to other **AI** solutions Jyh-Jong Tsay and Hsu [2011]. Interestingly, the top-performing solutions in this competition did not utilize **RL** but instead employed greedy path-finding approaches, sometimes combined with A* algorithms Julian Togelius and Baumgarten [2010].

At Stanford University, a group of students implemented a **Q-Learning** algorithm for the Super Mario Framework as part of an assignment Yizheng Liao and Yang [2012]. By reducing the problem space to a smaller number of possible states, they were able to create **AI** agents that achieved high success rates in completing single levels. Moreover, the same **AI** agents could generalize their learned behaviors to overcome randomly generated obstacles, showcasing the versatility of the approach.

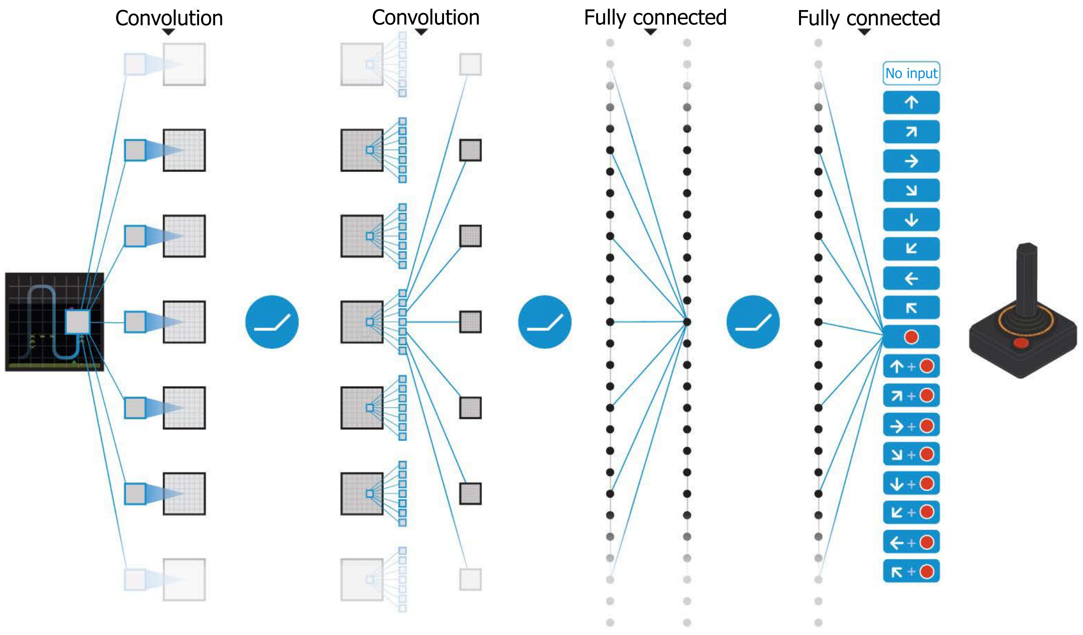


Figure 13: Diagram of **DQN** architecture for playing Atari games. Mnih et al. [2015]

Deep Q-Network (**DQN**) have been successfully employed in completing relatively easy Atari games Kaiser et al. [2019]. This was accomplished by processing a large number of game screen images, enabling the **AI** agent to make accurate predictions for different game states. However, training **DQN**s to achieve optimal performance requires extensive fine-tuning, such as adjusting learning rates, batch sizes, and exploration strategies. Parameter selection can be challenging, and a degree of trial and error is often necessary. The notable work by Mnih popularized **DQN**s by demonstrating their ability to learn Atari games directly from screen pixels to joystick actions (see Figure 13) Mnih et al. [2015].

Another approach explored in this domain is imitation learning, where **AI** agents imitate human behavior by recording actions performed by expert players in the game. This method relies on expert human players and requires them to play the game multiple times to generate training data. Imitation learning involves creating a reward function based on limited observations from the expert player in a **Markov Decision Process** (**MDP**) environment Battle [2018]. Rewards are mapped to features in states based on the player's actions, and **RL** can then be applied using the derived reward function Lee et al. [2014]. In games with large state spaces, **Q-Learning** has been compared to **Double Q-Learning** (**Double-QL**), showing that while training time did not significantly improve with **Double-QL**, the resulting **AI** agent outperformed the solution generated by standard **Q-Learning** Schilperoort et al. [2018] Somasundaram et al. [2018].

Chapter 3

The problem and its challenges

Platformer games present a significant challenge for computers to consistently find optimal solutions in real-time due to their inherent complexity Aloupis et al. [2015]. However, **AI** systems have demonstrated relatively strong performance in certain games, such as AlphaZero's superior gameplay in chess compared to human players Silver et al. [2018].

The difficulty of platformer games stems from the ever-changing environment and the presence of random elements that introduce surprises. Limited information about the environment poses risks for decision-making. In such cases, the circumstances should guide the **AI** based system to choose actions with the highest probability of success. Similar to **AI** surpassing human performance in chess, prior research has shown that **AI** can outperform humans in specific problems like path-finding. An effective and adaptable **AI** should be able to handle changing environments, such as when new types of monsters are introduced.

However, an essential requirement for real-time applications is the ability of **AI** to make prompt decisions. The inference time of the **NN** plays a crucial role in achieving a viable product.

This chapter is divided into three distinct sections, as depicted in Figure 14 [1]. Each section explores specific challenges and concerns raised by the thesis.

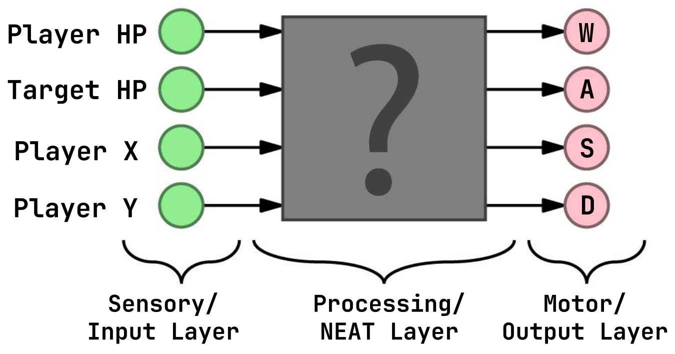


Figure 14: The three layers that compose a **NEAT** bot architecture.

[1] Do note that this is merely a figurative example and not by any means the final architecture.

3.1 The Sensory/Input Layer

The Sensory/Input Layer of the system plays a pivotal role in facilitating the interaction between the game world and the **AI**'s decision-making process. However, reverse engineering the memory address space of **World of Warcraft** (**WoW**) presents significant challenges and requires specialized techniques, such as the utilization of tools like Cheat Engine, to extract the necessary memory data.

Reverse engineering **WoW**'s memory address space involves comprehending the intricate structure and organization of the game's memory in order to identify and extract specific data points. This undertaking necessitates an extensive understanding of the game's internal workings and memory management mechanisms. This thesis will explore the intricacies and complexities associated with this task and propose potential strategies to surmount them.

The retrieval of relevant memory data is of utmost importance for the successful implementation of the system's input layer. This layer acts as the sensory interface through which the **AI** perceives and interacts with the game environment. By accessing and analyzing the game's memory, the **AI** can obtain real-time information regarding various game elements, such as player and enemy positions, health and mana levels, and other pertinent attributes.

The identification of essential data from **WoW**'s memory poses a significant challenge. **WoW**'s memory address space is expansive and continually changing as the game progresses. This thesis will discuss methodologies and techniques employed to locate and extract the pertinent memory data, including pattern recognition, dynamic memory analysis, and debugging tools.

Moreover it will be shown the significance of selecting an appropriate set of memory data to provide a comprehensive sensory input to the system. By carefully choosing the relevant data points, the system can access the necessary information to make informed decisions and effectively respond to the dynamic game environment.

Undoubtedly, reverse engineering is a formidable undertaking, particularly when software developers employ countermeasures like **Address Space Layout Randomization** (**ASLR**) to fortify against memory exploitation vulnerabilities. **ASLR** introduces randomness to the positions of critical data areas in the process's address space Marco-Gisbert and Ripoll [2019], impeding an attacker's ability to reliably locate specific functions or data.

The reverse engineering process is arduous and time-consuming, requiring tenacity and expertise. While no definitive guidebook for reverse engineering exists, the rewards can be substantial once the essential information is acquired.

Once the relevant values from **WoW**'s memory layout are obtained, the subsequent challenge lies in programmatically accessing those values. **DLL** injection proves to be a valuable technique in this regard. **DLL** injection involves loading a **Dynamic-link library** (**DLL**) into another process's address space, enabling external programs to influence the behavior of the target program Iczelion [2002].

By employing **DLL** injection, the injected code can intercept and modify system function calls Shewmaker [2006], facilitating the reading of the process's memory. This technique, often utilized by reverse engineering tools like Cheat Engine, offers opportunities to access the necessary data for the system's input layer.

However, it is important to note that **DLL** injection and code hooking can raise security concerns and potentially violate the game's terms of service or anti-cheat measures. Consequently, it must be exercised caution and insurance to the adherence of legal and ethical standards.

By leveraging these techniques, the author can overcome the challenges associated with accessing **WoW**'s memory and retrieve the pertinent data required for the system's input layer. This approach establishes a solid foundation for the system's perception and comprehension of the game environment.

By delving into the intricacies of reverse engineering **WoW**'s memory address space and addressing the complexities of retrieving relevant memory data, this thesis strives to establish a robust framework for the system's sensory/input layer, fostering a comprehensive understanding of the game world and enabling informed decision-making processes.

3.2 The Processing/NEAT Layer

The Processing/**NEAT** Layer is a critical component of the system. The primary objective is to explore how the selection of appropriate input values can enhance **AI** performance, enabling it to exhibit more natural behavior. Additionally, the author will explore the necessity of adjusting the fitness function to provide better suited individuals and discuss the idea of subdividing the problem into several domains.

The selection of suitable input values plays a vital role in shaping the behavior of the system. By carefully considering which aspects of the game environment should be fed as input to the **NN**, the **AI** can better understand and respond to the game dynamics. For example, variables such as player position, enemy positions, power-up locations, and terrain features can significantly impact the **AI**'s decision-making process. By incorporating relevant input values, the **AI** can exhibit more intelligent and natural behavior in navigating the game world.

Moreover, it will be discussed the need to adjust the fitness function to guide the evolution of the **AI** individuals. The fitness function serves as a measure of performance and determines which individuals are selected for reproduction and further evolution. By fine-tuning the fitness function, the system can prioritize desired behaviors and encourage the emergence of more effective strategies.

To tackle the complexity of the platformer game, this thesis proposes subdividing the problem into several domains. Each domain represents a specific subset of the game, focusing on distinct challenges or objectives. By breaking down the problem, the system can adapt and specialize in different aspects of the game, leading to more efficient and robust performance.

This will imply the necessity to address the issue of interchangeably switching between domains. This flexibility allows the system to dynamically adapt its focus based on the game's current context. By switching between different domains, the system can effectively handle changing environments, varying levels of difficulty, and the introduction of new elements or challenges.

Feedback loops enable the system to receive information about the consequences of its actions, facilitating adaptation and refinement of the decision-making process by altering the fitness accordingly. This feedback mechanism allow for the algorithm to evaluate the state of fitness of each individual by monitoring game events and analyzing changes in game state which will then lead to the refinement of the system's behavior based on observed outcomes.

The **NEAT** or Processing Layer thus encompasses various considerations and techniques related to selecting appropriate input values, adjusting the fitness function, subdividing the problem into domains, and enabling seamless interchangeability between these domains. Through a comprehensive exploration of these aspects, the author aims to enhance the system's overall performance and adaptability in the platformer game environment.

3.3 The Motor/Output Layer

The Motor/Output Layer is a pivotal component in the system, focusing on emulating the processed input back into the running **World of Warcraft** (**WoW**) process. The objective is to establish seamless interaction between the system and the game allowing it to influence the game world based on its decision-making process.

Emulating the output back into the **WoW** process necessitates careful consideration of the game's mechanics and the potential impact of the system's actions. This thesis will explore various strategies for translating the system's decisions into meaningful actions within the game, such as controlling the player character's movements, performing combat actions, and interacting with the environment.

One technique for emulating the output involves direct injection of commands or inputs into the game process. This approach entails sending appropriate signals or instructions to the game, simulating user input as if it were generated by a human player. The author will delve into the associated challenges and considerations, including the need for synchronization, timing, and adherence to the game's rules and limitations.

The Output Layer assumes a critical role in bridging the gap between the system and the game world. By discussing techniques for emulating the system's output back into the running **WoW** process, this thesis aims to empower the system to actively participate and exert influence over the game environment based on its acquired behaviors and decision-making capabilities.

Part II

Core of the Dissertation

Chapter 4

Contribution

4.1 Introduction

The target application of this thesis is a **Massively Multiplayer Online Role-playing Game** (**MMORPG**), which is implemented as a software program. Like most software applications, this **MMORPG** is susceptible to implementation bugs and design flaws that can be exploited by hackers or experienced software developers.

The architecture of this **MMORPG** follows a common pattern used by many similar games. A central bank of servers facilitates real-time communication with individual players over the Internet. Players use client programs on their own Internet-connected devices to interact with the game world.

The game client is a software program that runs on the player's device. It receives input from the user and communicates with the central servers over the Internet. The client software provides a graphical view of the virtual world, displaying the player's location, other players, and ongoing actions.

In order to handle the real-time interactions and actions of thousands of players, the game client maintains a client-side state. The game engine, situated within the client software, processes user input and updates the game state as time progresses. Similar to any computer program, the state of the game is defined as the current values of memory locations, secondary storage, registers, and other components of the system.

The challenge arises from the significant bandwidth required to transmit the entire game state from the server to all clients in real-time. This limitation led **MMORPG** designers to allow the client software to preserve and manage some portions of the game state. This approach ensures that the game actions appear seamless to the players.

This technical detail regarding the client-side state preservation and management is crucial to the feasibility of this project. By accessing this preserved state, it becomes possible to extract the specific data required as input to the **Neural Network** (**NN**). The project involves constructing a sensory/input layer, a processing/**NEAT** layer, and combining them with the motor/output layer to produce the desired result.

4.2 Building the Sensory/Input Layer

The sensory layer, also known as the input layer, defines the structure and manner in which data is input into **NN**s. In the context of analyzing game states, the input data may be obtained by utilizing reverse engineering techniques to probe the memory of a running process, specifically that of the game's client application. The sensory layer is constructed by utilizing a variety of software tools that have been developed over time, such as debuggers, decompilers, and disassemblers. These tools facilitate the exploration and comprehension of software, allowing for the extraction of relevant information that can be retained in order to build the sensory layer.

4.2.1 Scanning Memory using Cheat Engine

Cheat Engine is a powerful memory scanner that searches a game's operating memory, which resides in **Random-Access Memory** (**RAM**). Understanding a game's state is a fundamental requirement for effectively engaging with it, but unlike human cognition, it is complicated for software to discern a game's state by visual observation alone. However, the underlying memory space of a game program houses a numerical representation of the game state, which software can comprehend effortlessly. As such, numerical analysis of game state memory structures serves as a critical foundation for developing the intended software. Memory scanners such as this one are utilized to locate the relevant values in a game's memory. Subsequently it is possible to access the memory in those specific locations to discern the game's state. This technique allows to gather information about a game's internal state.

All data in the memory of a game is stored at a specific location, known as a memory address. In essence, the memory can be thought of as a large array of bytes, and a memory address is an index that points to a particular value in that array. When a memory scanner is instructed to search for a particular value x in the memory of a game, it iterates through the byte array, searching for any values that are equal to x. Whenever it finds a match, it adds the index of the match to a result list.

However, given the vast size of a game's memory, x can potentially be present in hundreds of locations. For instance, if x is the player's current health, which is 500, it is highly likely that the value of 500 is present in many other locations besides the one that represents the player's health. To eliminate these extraneous values, the memory scanner provides an option to rescan the result list, removing any addresses that no longer hold the same value as x, whether x is still 500 or has changed.

To conduct effective rescans, the game's overall state must have significant entropy or disorder. The entropy can be increased by modifying the in-game environment, such as by moving around, killing creatures, or changing characters. As the entropy increases, unrelated addresses are less likely to hold the same value as x, and with enough entropy, a few rescans should filter out all false positives and leave the true address of x. Additional details of this process are provided in the appendix A.1.

4.2.2 Pointer Scanning using Cheat Engine

Pointer scanning is a technique used to locate dynamic memory addresses by tracing chains of pointers that lead to them. The process involves first identifying a static memory address that points to the dynamic memory location we're interested in. We then use Cheat Engine to scan the game's memory to find all addresses that contain that static memory address.

Once we have a list of addresses that reference the static address, we can begin the pointer scanning process. The goal is to identify other static memory addresses that point to the first static address, forming a chain of pointers. To do this, we use Cheat Engine's pointer scanner to follow the pointer chain and identify additional addresses that reference the dynamic memory location.

Pointer scanning can be a more complicated process than simple memory scanning, as it often requires a greater understanding of the game's memory structure and the relationships between different memory locations.

The memory pointer is the starting address of the chain and the offsets make up the path to the desired value. Each offset is essentially an instruction that tells the program to move a certain number of bytes from the current address to reach the next address in the chain. By following the pointer chain, Cheat Engine can ultimately locate the address in dynamically allocated memory that holds the value of interest.

```
list[int] chain = [start, offset1, offset2, (...) ]
```

The first value in this pointer chain (*start*) is called a memory pointer. It's an address that starts the chain. The remaining values (*offset1, offset2*, and so on) make up the route to the desired value, called a pointer path.

This *pseudocode* show how a pointer chain might be read:

The function **read_pointer_chain(chain)** takes a pointer chain called chain as input, which is interpreted as a list of memory offsets from the memory pointer located at **chain[0]** ❶ .

Next, the function iterates over each offset in the pointer chain, adding each offset to the current value of **mem_value**, and reads the value at the resulting memory address using the **read()** function. The

Algorithm 1 Pseudocode to read a pointer chain

```
procedure read_pointer_chain(chain)
    ❶ $mem\_value = read(chain[0]);$
    for ($i = 1; i \neq chain.len - 1; i++$) do
        $offset = chain[i];$
        $mem\_value = read(mem\_value + offset)$
    return $mem\_value$
```

value read is then assigned to **mem_value**. This process is repeated for each offset in the chain until the end of the chain is reached.

Once the end of the pointer chain has been reached, the final value of **mem_value** is returned as the result of the function. In other words, the function reads a series of memory addresses located using a pointer chain and returns the value at the last address in the chain.

Which with the loop unrolled and with a concrete example would look something like this:

Algorithm 2 Pseudocode to read a pointer chain with loop unrolled

```
procedure read_pointer_chain()
    $list < int > chain = \{0xDEADCAFE, 0xBC, 0x11, 0x05\}$
    $mem\_value = read(0xDEADCAFE)$
    $mem\_value = read(mem\_value + 0xBC)$
    $mem\_value = read(mem\_value + 0x11)$
    $mem\_value = read(mem\_value + 0x05)$
    return $mem\_value$
```

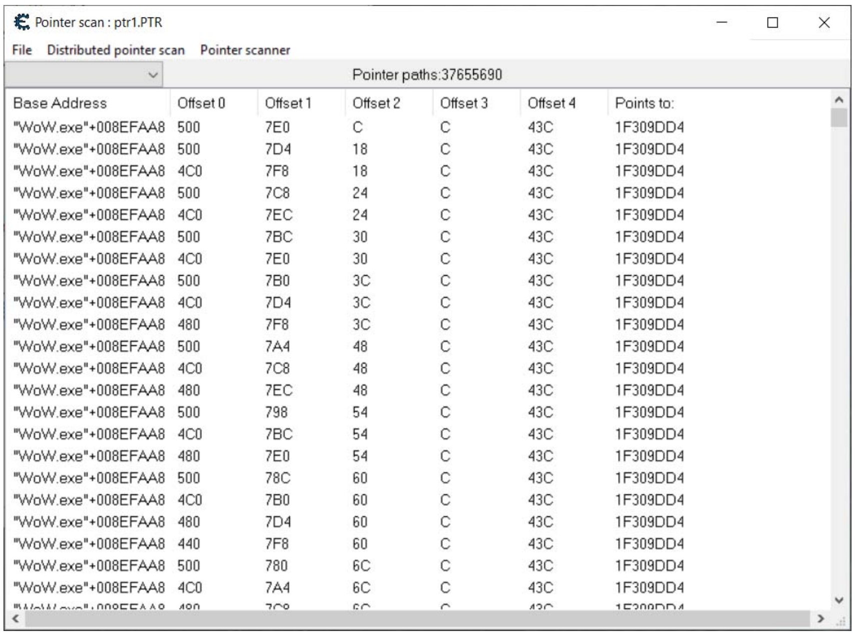
Pointer chains are essentially lists of memory offsets that lead to a dynamically allocated memory chunk. These chains are useful for those who want to modify values stored in dynamic memory. However, because of their complex structure, pointer chains cannot be easily located using traditional memory scanning methods.

To overcome this challenge, pointer scanning techniques were developed, which involve brute-forcing every possible pointer chain until the desired memory address is located. The pointer scanning process involves recursively iterating over the possible pointer paths and updating the value of a memory pointer at each step until the target memory address is reached.

While it is possible to locate and analyze the assembly code to deduce the pointer path used to access the desired value, this process is time-consuming and requires advanced tools. Pointer scanners are a faster and more efficient method for identifying pointer chains and accessing dynamic memory.

To initiate a pointer scan in Cheat Engine, you first need to locate a dynamic memory address in your cheat table. Once you've found the address you're interested in, right-click on it and select "Pointer scan for this address" from the context menu. This will open a dialog box asking you where to save the scan results as a .ptr file.

When you've selected a location to save the scan results, click "Ok" and wait for the scan to complete. This process can take a while, depending on the complexity of the game and the number of pointer chains that need to be searched. Once the scan is finished, you'll be presented with a list of possible pointer chains that lead to the target memory address as shown on figure 15.



Pointer paths:37655690

Base Address	Offset 0	Offset 1	Offset 2	Offset 3	Offset 4	Points to:
"WoW.exe"+008EFAA8	500	7E0	C	C	43C	1F309DD4
"WoW.exe"+008EFAA8	500	7D4	18	C	43C	1F309DD4
"WoW.exe"+008EFAA8	4C0	7F8	18	C	43C	1F309DD4
"WoW.exe"+008EFAA8	500	7C8	24	C	43C	1F309DD4
"WoW.exe"+008EFAA8	4C0	7EC	24	C	43C	1F309DD4
"WoW.exe"+008EFAA8	500	7BC	30	C	43C	1F309DD4
"WoW.exe"+008EFAA8	4C0	7E0	30	C	43C	1F309DD4
"WoW.exe"+008EFAA8	500	7B0	3C	C	43C	1F309DD4
"WoW.exe"+008EFAA8	4C0	7D4	3C	C	43C	1F309DD4
"WoW.exe"+008EFAA8	480	7F8	3C	C	43C	1F309DD4
"WoW.exe"+008EFAA8	500	7A4	48	C	43C	1F309DD4
"WoW.exe"+008EFAA8	4C0	7C8	48	C	43C	1F309DD4
"WoW.exe"+008EFAA8	480	7EC	48	C	43C	1F309DD4
"WoW.exe"+008EFAA8	500	798	54	C	43C	1F309DD4
"WoW.exe"+008EFAA8	4C0	7BC	54	C	43C	1F309DD4
"WoW.exe"+008EFAA8	480	7E0	54	C	43C	1F309DD4
"WoW.exe"+008EFAA8	500	78C	60	C	43C	1F309DD4
"WoW.exe"+008EFAA8	4C0	7B0	60	C	43C	1F309DD4
"WoW.exe"+008EFAA8	480	7D4	60	C	43C	1F309DD4
"WoW.exe"+008EFAA8	440	7F8	60	C	43C	1F309DD4
"WoW.exe"+008EFAA8	500	780	6C	C	43C	1F309DD4
"WoW.exe"+008EFAA8	4C0	7A4	6C	C	43C	1F309DD4

Figure 15: Cheat Engine's pointer scan result to a dynamic memory address.

To identify the correct pointer chain, you'll need to analyze the results and look for a chain of offsets that leads from a static memory address to the dynamic address you're interested in. This can be a time-consuming process, but once you've identified the correct pointer chain, you'll be able to use it to access the desired value through it.

The pointer scanner in Cheat Engine offers a rescan feature that can help to reduce false positives. However, in rare cases, rescanning using a rescan loop may still leave a large list of possible paths. In such cases, it may be necessary to restart the game, locate the address that holds the value, and use the rescan feature on this address to further narrow the results. It is important to leave the "Only filter out invalid pointers" option unchecked and enter the new address in the "Address to find" field.

If the results are still not narrow enough, running the same scan across system restarts or on different systems may help. If this still yields a large result set, each result can be safely considered static because more than one pointer chain can resolve to the same address.

Once the result set is narrowed down, usable pointer chains can be added to the cheat table by double-clicking on them. If multiple chains with identical offsets that start with the same pointer but diverge after a certain point are found, it is possible that the data is stored in a dynamic data structure. In such cases, it is recommended to select the chain with the fewest offsets. In Appendix A.2 there is additional information about this method with a concrete example in order to facilitate comprehension.

Understanding the Representation of Variables and Data in Computer Memory

Manipulating a game's state can indeed be a challenging task, as it involves understanding the structure and layout of the game's memory. Simply using tools like Cheat Engine to scan for values may not always yield the desired results, especially when trying to manipulate multiple related values simultaneously.

To successfully manipulate game state, it is often necessary to identify patterns and structures within the game's memory. This may involve analyzing the game's code and memory dumps, as well as reverse engineering the game's data structures.

In the development of memory reading tools, it may be necessary to reconstruct the original structures within the code. This requires a deep understanding of how variables and data are laid out in the game's memory.

To gain this understanding, it may be necessary to use example code, OllyDbg memory dumps and Hex-ray's IDA Pro dissembled binaries, and tables to tie everything together. These resources are necessary to identify the game's memory structures and create effective tools that can manipulate game state.

4.2.3 Countermeasures Against Reverse Engineering

Reverse engineering is a challenging and complex process that involves analyzing software programs and their code to extract information or data from them. One of the most significant obstacles in reverse engineering is **Address Space Layout Randomization** (**ASLR**), a security technique employed by modern operating systems to randomize the memory layout of a program during runtime, thereby making it more difficult for attackers to exploit vulnerabilities in the software.

It is certainly within one's capabilities to search for and modify data within a game program. However, it is important to keep in mind that the game program itself is also capable of searching memory for any alterations made to its own code or data. This is often achieved through the implementation of integrity checking measures, which may scan for any injected code or data that has been placed into memory via active malware scanning techniques.

Many of the techniques utilized in this process involve modifying code, adjusting data bits, and injecting threads or **DLL**s into the game process. However, such activities are not without their drawbacks, as they can be detected by certain game software. One example of this is Blizzard's Warden, which serves as a protective measure for **WoW**. As such, it is possible to hide from and defeat many forms of scanning, but the process can often be complex and may require advanced measures.

In order to accelerate development an external memory reading library named Blackmagic was used for this purpose.

While these countermeasures can make it more difficult to reverse engineer games, they are not foolproof. There can often be found ways around them, and this creates an ongoing arms race between game developers and reverse engineers. Despite this, using countermeasures to protect against reverse engineering is an important step in ensuring the integrity and security of games and other software.

4.2.4 DLL Injection, Hooking and Process Memory Reading

Dynamic-link library (**DLL**) injection is used to inject a custom **DLL** into the memory space of the running **World of Warcraft** process. The injected **DLL** contains code that hooks into the game's rendering function, **Endscene**, and modifies its behavior to allow the reading of process memory.

The **DLL** injection process is carried out by the **Inject.InjectDLL** function, which is called in the **BmWrapper.Start** function. This function loads the custom **DLL** into memory and then uses the Windows **API** function **CreateRemoteThread** to execute the **DLL** 's **DllMain** function in the context of the **World of Warcraft** process.

Once the custom **DLL** is successfully injected into the game's memory space, it sets up hooks to modify the behavior of the **Endscene** function. This is done through the **Endscene.Init** function, which allocates memory in the game's process space using the **AllocateCaves** function. The allocated memory is used to store the detour and code cave functions that will be executed by the hooked **Endscene** function.

The **Endscene.Init** function then sets up the detour function using the **inject** function to write assembly code into the allocated memory. The detour function pushes the registers and then modifies the **ecx** register to access the **IDirect3DDevice9** interface, which contains the rendering data for the game. The **mov** instruction then stores the **Endscene** function address in the allocated memory space, and the detour function then jumps to the custom code cave function. The **IsSceneEnd2** address is then used to jump back to the original **Endscene** function once the custom code cave function has executed.

The custom code cave function is created using the **CreateCodeCave** function, which again uses the **inject** function to write assembly code into the allocated memory space. This function is executed within the detour function and is used to execute the custom code.

The **CreateDetour** function is used to create the second detour that is called only when the player is in-game. This is done by checking the value of *0xB4B424*, which is a pointer to a memory address that holds a value indicating whether the player is in-game or not. If the player is in-game, the **CreateDetour** function uses a pointer to the player object and calls the custom code cave function. The result of the custom code execution is then returned to the hooked **Endscene** function, which can then modify the game's rendering output as needed.

Overall, the **DLL** injection process is critical to modifying the behavior of the game's rendering function to allow for the reading of process memory. The custom code executed within the code cave function can then be used to gather information about the game's state, which can be used to build automated tools such as the one presented by this thesis.

Code Caves

A code cave refers to a block of unused memory space that is created within the target process to allow the injection of custom code. Code caves can be used to modify the behavior of an application by hijacking the execution flow of a particular function.

The code cave in the provided functions is created using the **AllocateCaves** function. The Inject class allocates a block of memory within the process using the **VirtualAllocEx** function, which returns a pointer to the allocated memory. The size of the memory block is specified in bytes as a parameter to the function.

Once the memory is allocated, the code for the code cave is assembled into an array of strings that represent the assembly instructions. These instructions are then injected into the allocated memory block using the inject function. The injected code will overwrite the existing instructions at the memory address specified by the code cave pointer.

In the provided functions, a code cave is used to execute custom code injected by the user. The **CreateCodeCave** function creates a code cave where the custom code is injected. When the **IsSceneEnd** function is called, it will jump to the code cave instead of returning to the calling function. The custom code will then execute and return a value that will be used by the calling function.

Overall, code caves provide a powerful technique for modifying the behavior of a process by injecting custom code. They can be used for a wide range of purposes, including bypassing security measures, debugging, and reverse engineering.

4.3 Building the Processing/NEAT Layer

For this layer an external framework was used called Sharp**NEAT**.

Sharp**NEAT** is an open-source software library that implements the **Neuroevolution of Augmented Topologies** (**NEAT**) algorithm, which is a type of **NeuroEvolution** (**NE**). **NE** is a subfield of **Artificial Intelligence** (**AI**) that combines **Evolutionary Algorithms** (**EA**s) with **Artificial Neural Network**s (**ANN**s or **NN**s) as previously discussed. The **NEAT** algorithm is particularly useful for evolving **NN**s, as it gradually increases their complexity over time, allowing them to perform better on a given task.

The **NEAT** algorithm is a form of **Genetic Algorithm** (**GA**) that evolves **NN**s by starting with a minimal network architecture and then adding or modifying nodes and connections over time. This allows the network to become increasingly complex and specialized for the task at hand.

Sharp**NEAT** is written in C# and provides a range of tools and features for evolving **NN**s using the **NEAT** algorithm. One of the key features of Sharp**NEAT** is its user-friendly interface, which allows researchers and developers to create and evolve **NN**s with ease. Sharp**NEAT** also provides a range of visualization tools, which can be used to explore and analyze the behavior of the evolved **NN**s.

Sharp**NEAT** is capable of evolving **NN**s for a wide range of tasks, including classification, prediction, and control. For example, it can be used to develop **NN**s that can recognize handwritten digits, predict stock prices, or control a robot's movements. It does this by providing a range of fitness functions, mutation and crossover operators, and other parameters that can be tweaked to optimize the performance of the evolved **NN**s.

The primary goal of Sharp**NEAT** is to provide a flexible and efficient tool for researchers and developers who are interested in using **GA**s to evolve **NN**s. By making it easier to create and evolve **NN**s, Sharp**NEAT** can help to accelerate progress in the fields of **AI**, **Machine Learning** (**ML**), and computational neuroscience and in this particular case was crucial in obtaining a viable and polished product.

The objective at hand was divided into two domains: navigation and combat. Both domains were implemented entirely within the Sharp**NEAT** framework and then integrated by creating an interface that enabled them to operate concurrently.

Additional details on how to define a domain in Sharp**NEAT** and the process of the setting up the default configuration files are provided in the appendices A.3 and A.4 respectively.

4.3.1 Navigation Domain and Strategy

In the navigation domain, the central objective is for an individual to proficiently navigate towards a pre-determined target with the intention of initiating an attack. The individual continually evaluates its own position in relation to that of a nearby enemy, and its performance is assessed based on the reduction of distance between them. A shorter distance signifies a more effective navigation strategy, leading to higher fitness scores and indicating superior performance.

Once the individual successfully reaches the designated target, a strategic transition occurs. This transition involves the individual orienting itself towards the target and seamlessly transitioning to the combat domain. This shift in focus represents the individual's preparedness to engage in combat and effectively confront the target.

The navigation domain plays a critical role within the broader framework, emphasizing the individual's ability to navigate tactically towards the target while maintaining an optimal position relative to the enemy. By continually striving to minimize the distance between itself and the enemy, the individual demonstrates its capability to execute a successful navigation strategy.

Domain Schematics

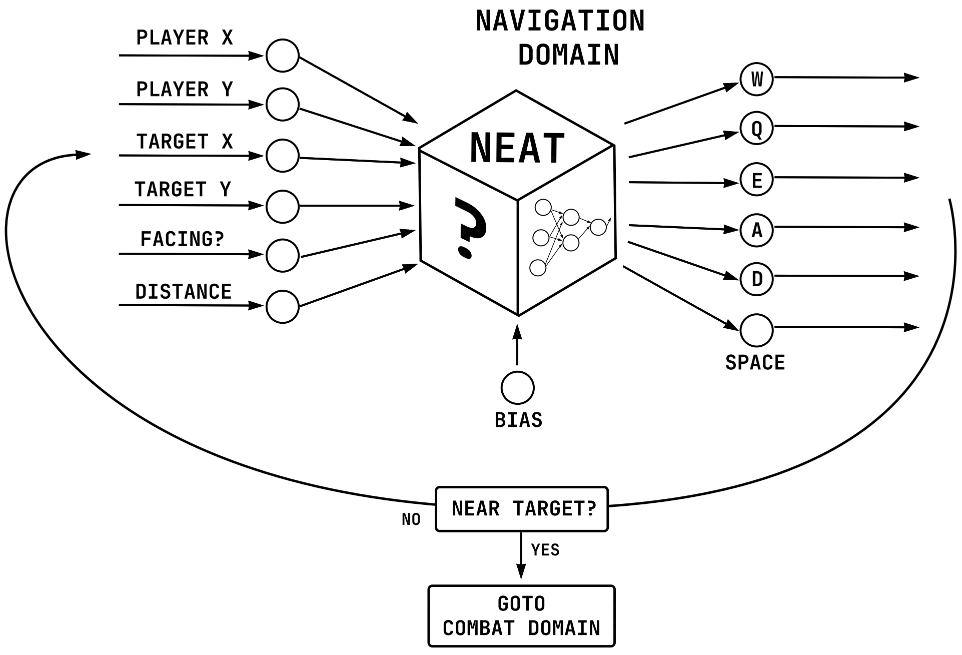


Figure 16: Schematics of the Navigation Domain.

Inputs

The sensory/input layer of the domain utilizes the following inputs to provide information to the **NN**:

1. Bias: A constant input used to provide a bias or offset to the network's computations.
2. Player X position: The X-coordinate of the player's position in the virtual world.
3. Player Y position: The Y-coordinate of the player's position in the virtual world.
4. Target X position: The X-coordinate of the target's position in the virtual world.
5. Target Y position: The Y-coordinate of the target's position in the virtual world.
6. If Player is Facing Target: This input represents whether the player is facing towards the target. It is calculated using the atan2 function, which computes the angle between the player's position and the target's position.
7. Distance from Player to Target: The distance between the player and the target in the virtual world.

By providing these inputs to the **NN**, the sensory/input layer conveys essential information about the player's position, the target's position, and their spatial relationship. These inputs serve as the foundation for the network to make decisions and generate appropriate outputs in response to the given task or objective.

Outputs

The motor/output layer of the domain utilizes the LeakyReLU activation function and provides the following possible outputs:

1. w: This output corresponds to the action of moving forward.
2. q: This output corresponds to the action of moving with a strafe to the left.
3. e: This output corresponds to the action of moving with a strafe to the right.
4. a: This output corresponds to the action of turning to the left.
5. d: This output corresponds to the action of turning to the right.
6. SPACE: This output corresponds to the action of jumping.

These outputs represent the possible actions that the entity or player can take in the given domain or virtual environment. By selecting one or a combination of these outputs, the entity's behavior can be controlled and manipulated within the domain. The LeakyReLU activation function is applied to the outputs to introduce non-linearity and capture complex relationships between the inputs and outputs, allowing for more flexible and adaptive decision-making.

Fitness

In the given domain, the fitness of the entity or player is determined based on certain conditions after testing the output. The fitness is increased under the following conditions:

1. Distance to target decreased: When the distance between the entity and the target decreases, the fitness is substantially increased. This condition incentivizes the entity to approach the target and rewards progress towards reaching it.

2. Player is currently facing the target: If the entity is facing the target, the fitness is slightly increased. This condition encourages the entity to align its orientation with the target, potentially indicating a more accurate or advantageous position.

By rewarding the entity for reducing the distance to the target and aligning its orientation, the fitness function promotes behaviors that are conducive to successfully completing the task or objective in the given domain. The fitness value serves as a measure of how well the entity is performing based on these criteria, guiding the evolutionary process to improve the entity's decision-making and overall performance.

Stop Condition

In the navigation task of the domain, the stop condition is defined as follows:

- The distance between the Player and the Target is less than X: The navigation task is considered fulfilled when the distance between the Player and the Target becomes less than a specified threshold value X, which in this case is set to 25. Once this condition is met, it indicates that the Player has successfully reached the target location or is in close proximity to it.

Upon fulfilling the navigation task, a "flag" is raised to initiate the combat domain. This flag serves as a signal or marker to indicate the transition from the navigation domain the combat domain. It triggers the system or algorithm to switch its focus or behavior from navigation-related actions to combat-related actions.

By establishing a clear stop condition based on the distance between the Player and the Target, the system can determine when the navigation objective has been achieved and proceed to the next phase of the domain, which in this case is the combat domain.

NEAT's Parameters - Navigation Domain

Table 1: NEAT's Parameters for the Navigation Domain.

Parameter	**Value**
Name	WoW Cheese
Is Acyclic	False
Activation Function Name	LeakyReLU
Evolution Algorithm:	
Species Count	5
Elitism Proportion	0.2
Selection Proportion	0.2
Offspring Asexual Proportion	0.5
Offspring Sexual Proportion	0.5
Interspecies Mating Proportion	0.01
Reproduction (Asexual):	
Connection Weight Mutation Probability	0.4
Add Node Mutation Probability	0.1
Add Connection Mutation Probability	0.3
Delete Connection Mutation Probability	0.025
Reproduction (Sexual):	
Secondary Parent Gene Probability	0.1
Population Size	10
Initial Interconnections Proportion	0.1
Connection Weight Scale	5.0
Complexity Regulation Strategy:	
Strategy Name	Relative
Relative Complexity Ceiling	30
Min Simplification Generations	10
Degree of Parallelism	1
Enable Hardware Accelerated Neural Nets	True
Enable Hardware Accelerated Activation Functions	True

4.3.2 Combat Domain and Strategy

Within the combat domain, the primary objective is for an individual to successfully eliminate the designated target enemy. This is achieved through constant monitoring and comparison of the target's health status. The individual's performance is evaluated based on its ability to diminish the target's health, resulting in higher fitness scores.

Throughout the combat engagement, the individual employs a strategic approach tailored to effectively deplete the target's health. It utilizes a range of offensive and defensive abilities and evolves and adapts its combat strategy based on the target's behavior, optimizing its chances of successfully slaying the enemy.

Upon achieving its objective, which is the elimination of the target enemy, the individual transitions back to the navigation domain. This resumption of the navigation process marks the completion of the combat phase and signifies the individual's readiness to pursue subsequent objectives.

The combat domain represents a critical phase within the overall framework, where the individual's combat prowess and strategic decision-making abilities are put to the test. The efficiency with which the individual reduces the target's health directly influences its overall performance and fitness. By effectively executing combat strategies and adapting to changing circumstances, the individual increases its chances of successfully eliminating the target enemy.

Domain Schematics

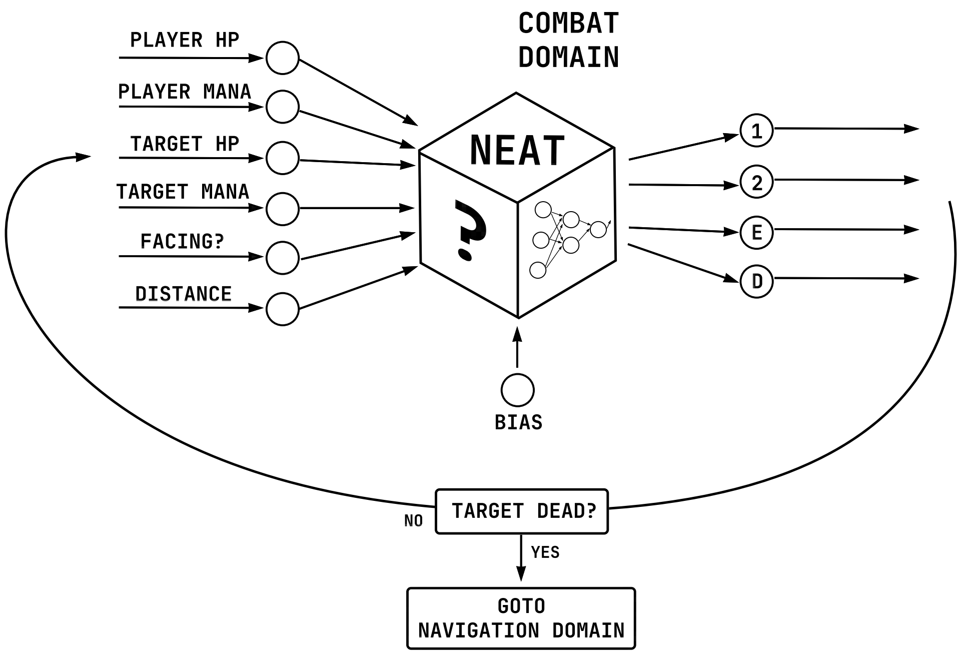


Figure 17: Schematics of the Combat Domain.

Inputs

The chosen inputs for this domain are as follows:

1. Bias: A constant input used to adjust the activation of neurons in the **NN**.
2. Player Health: The current health level of the player character. This input provides information about the player's remaining health, which can be used by the **NN** to make decisions based on the player's vulnerability or need for self-preservation.
3. Player Mana: The current mana level of the player character. This input represents the player's available magical energy or resource, which can be used for casting spells or performing special abilities.
4. Target Health: The current health level of the target enemy or opponent. This input allows the **NN** to assess the target's vulnerability and make decisions based on the target's remaining health.
5. Target Mana: The current mana level of the target enemy or opponent. Similar to the player's mana input, this provides information about the target's available magical energy or resource.
6. If Player is Facing Target: This input is determined using the atan2 function, which calculates the angle between the player and the target. It indicates whether the player is facing towards the target or not, which can be crucial in determining the effectiveness of certain actions or attacks.
7. Distance from Player to Target: This input represents the Euclidean distance between the player and the target. It provides information about the proximity of the target to the player and can be used to guide decision-making related to engagement or pursuit.

By incorporating these inputs, the system can take into account various factors such as the player's and target's health, mana levels, relative orientation, and distance to make informed decisions during combat scenarios.

Outputs

For this domain, the chosen outputs for the **NN** are as follows:

1. 1: This output represents an action to inflict damage on the target enemy or opponent. The neural network can activate this output to initiate an attack or offensive maneuver against the target.

2. 2: This output indicates an action to restore the player character's health. Activating this output can trigger healing abilities or actions to replenish the player's health.

3. a: This output instructs the player character to turn or rotate in the left direction. It allows the neural network to control the player's movement and orientation during combat scenarios.

4. d: This output instructs the player character to turn or rotate in the right direction. Similar to the "Turn Left" output, it enables the neural network to control the player's movement and orientation.

The chosen activation function, LeakyReLU, is applied to the outputs to introduce non-linearity and flexibility in the neural network's decision-making process. By combining these outputs with the chosen inputs, the **NN** can learn and adapt its behavior to optimize combat strategies, including attacking, healing, and maneuvering based on the given inputs and desired outcomes.

Fitness

The fitness function for this combat domain is based on evaluating the performance of the **NN** in achieving specific objectives. The fitness is increased under the following conditions:

1. Target Health Decreased: When the target's health decreases, it indicates that the **NN** is effectively inflicting damage and progressing towards defeating the opponent. This condition significantly increases the fitness score, rewarding successful combat actions.

2. Player is Currently Facing the Target: If the player character is facing the target enemy, it demonstrates good positioning and awareness in combat. This condition slightly increases the fitness score, encouraging the **NN** to maintain an advantageous position relative to the target.

By rewarding the **NN** for reducing the target's health and exhibiting appropriate orientation towards the target, the fitness function incentivizes the development of combat strategies that prioritize damaging the opponent and maintaining a favorable position. This encourages the **NN** to learn effective combat techniques and improve its decision-making abilities during battles.

Stop Condition

In the combat task of the domain, the stop condition is based on the status of the target. The combat task will be considered fulfilled when the target is defeated, indicated by the target's health reaching zero. Once the target's health becomes zero, it signifies that the combat objective has been accomplished.

Upon fulfilling the combat task, a "flag" is raised to indicate the completion of combat, and the system will proceed to restart the navigation task. This flag serves as a trigger for transitioning between tasks, allowing the system to switch from combat mode back to navigation mode, where the entity will resume its navigation behavior.

By setting the stop condition based on the target's health reaching zero, the system ensures that the combat task continues until the target is defeated. This enables the **NN** to learn effective combat strategies and adapt its behavior to overcome challenges posed by different opponents.

NEAT's Parameters - Combat Domain

Table 2: NEAT's Parameters for the Combat Domain.

Parameter	**Value**
Name	WoW Combat Cheese
Is Acyclic	False
Activation Function Name	LeakyReLU
Evolution Algorithm:	
Species Count	5
Elitism Proportion	0.2
Selection Proportion	0.2
Offspring Asexual Proportion	0.5
Offspring Sexual Proportion	0.5
Interspecies Mating Proportion	0.01
Reproduction (Asexual):	
Connection Weight Mutation Probability	0.4
Add Node Mutation Probability	0.1
Add Connection Mutation Probability	0.3
Delete Connection Mutation Probability	0.025
Reproduction (Sexual):	
Secondary Parent Gene Probability	0.1
Population Size	10
Initial Interconnections Proportion	0.1
Connection Weight Scale	5.0
Complexity Regulation Strategy:	
Strategy Name	Relative
Relative Complexity Ceiling	30
Min Simplification Generations	10
Degree of Parallelism	1
Enable Hardware Accelerated Neural Nets	True
Enable Hardware Accelerated Activation Functions	True

4.3.3 Atan2 - Determining if a player is facing a target

The atan2 function is a mathematical tool that plays a crucial role in many areas of computer science and engineering. One such area is the field of video game development, where it is used to determine the position of enemies on the screen relative to the player.

The atan2 function is a mathematical function that takes in two arguments, usually represented as x and y, and returns the angle between the positive x-axis and the line connecting the origin to the point (x, y). This angle is measured in radians and can range from -π to π.

The atan2 function is defined as follows:

$$atan2(y, x) = arctan(y/x)$$

However, unlike the arctan function, which has a limited range of $-\pi/2$ to $\pi/2$, the atan2 function takes into account the signs of both x and y to determine the correct quadrant in which the angle lies. This means that the atan2 function can return angles in all four quadrants of the coordinate plane.

In practical terms, this means that if we have a point (x, y) in a 2D coordinate system, we can use the atan2 function to find the angle between the positive x-axis and the line connecting the origin to that point. This angle can then be used to determine the position of the point relative to the player or another point of interest.

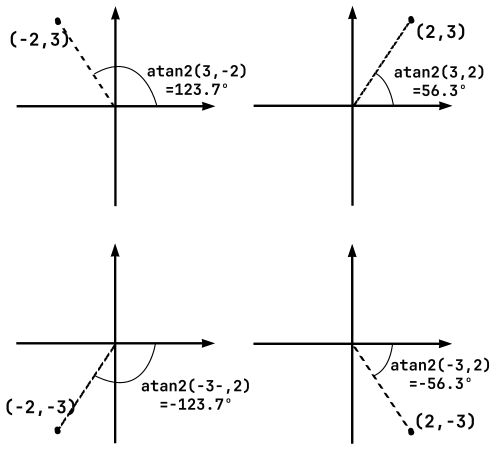


Figure 18: Pratical examples of **atan2**.

In video games, the atan2 function is often used to check the position of enemies relative to the player. By calculating the angle between the player's position and the enemy's position, it is possible to determine the direction in which the enemy or the player is facing Marschner and Shirley [2009].

4.3.4 Saving and Loading State

In order to facilitate the preservation and seamless transition of the state between the navigation and combat domains, the system incorporates mechanisms for saving and loading the current state of each domain. This is achieved through the utilization of static variables within the navigation and combat *EvaluationScheme* classes.

Specifically, the static variables *_static_ea_runner*, *_static_neatPop*, *_static_ea_runner_combat*, and *_static_neatPop_combat* serve as repositories for storing the state of the navigation and combat domains, including vital information such as the best genome and population.

Prior to switching from one domain to another, the system ensures that the current state of the active domain is saved to the corresponding static variables. This preservation of state guarantees that essential information is retained and can be accessed when needed during subsequent domain switches. By employing this approach, the system effectively enables a seamless context switching experience while maintaining the integrity of the state between domains.

Saving To and Loading From File

To provide users with the capability to save and reload the state of each domain, the system incorporates functionality for saving to and loading from file. Specifically, the best genome and population of both the navigation and combat domains can be saved to separate files, allowing for independent storage and retrieval. These operations are facilitated through the utilization of the NeatGenomeSaver and NeatPopulationSaver components.

When saving the state, the system employs the appropriate saver components to store the genomes and populations of the navigation and combat domains separately. Each domain's data is saved to its respective file, ensuring that the information remains distinct and easily accessible. Subsequently, when the need arises to reload a specific domain, the system leverages the saved files to retrieve the corresponding state. By doing so, the system can seamlessly restore the previous state of the domain, including the best genome and population.

The ability to save and load the state to and from file provides users with a valuable means of preserving and sharing their work. It allows for the persistence of crucial data between different sessions and facilitates the exchange of results or experiments among different instances of the system. Through the implementation of these file-based operations, the system enhances usability and empowers users with the flexibility to manage and manipulate their data effectively.

4.4 Building the Motor/Output Layer

Building the Motor/Output Layer involves establishing a mechanism for emulating the system's output and translating it into meaningful actions within the game. Initially, one approach considered was direct injection of commands or inputs into the game process. However, this approach was ultimately discarded due to several challenges and limitations.

Direct injection of commands or inputs involves sending signals or instructions directly to the game. While this approach can provide real-time interaction with the game, it poses challenges in terms of synchronization, timing, and adherence to the game's rules and limitations. Ensuring that the system's actions are within the game's constraints and avoiding actions that may be considered cheating or violating the game's terms of service is crucial[1]. Additionally, synchronizing the system's actions with the game's state and events can be complex and prone to errors.

Instead of direct injection, a more discrete technique was adopted by embedding the AutoIt scripting language into the C#[2] environment. AutoIt provides a scripting interface that allows automation of user actions and interactions with Windows-based applications. By utilizing AutoIt, the system can execute the desired actions within the game, simulating user input as if it were generated by a human player. It can control the player character's movements, combat actions, and interactions with the environment, among other things. This approach provides more control and flexibility in designing **AI** behaviors while ensuring compliance with the game's mechanics and rules.

[1] It has to pass as unsuspicious to the game's anti-cheating system.

[2] Which was used as the main programming language.

4.4.1 Autoit - Scripting Language

Figure 19: **Autoit** scripting language. Ltd [2023]

AutoIt is a scripting language used for automating the Windows **Graphical User Interface** (**GUI**). It provides various functions for automating tasks such as keystrokes, mouse clicks, and window management. One of the most useful functions that AutoIt provides is the ControlSend function.

The motor layer was contructed using the AutoIt's ControlSend function to send input to the **World of Warcraft** (**WoW**) game window. The ControlSend function in AutoIt is particularly useful when it comes to automating user input for applications. With ControlSend, we can send input to a specific window or control within a window, making it an ideal tool for these type of automation.

To use ControlSend in a C# project, first we need to download the AutoItX **DLL** from the autoit website then we need to import it. We can do this by selecting the **DLL** as a dependency to the project and copying it to the source folder. Once we have imported the **DLL** , we can use the ControlSend function to send any input to any running process with a **GUI**. Such input can come from a **NN**'s output, essentially accomplishing what's required to this stage.

To use the ControlSend function in our C# code, we first need to initialize the AutoIt environment importing it: *using AutoIt;*. We can then call the AutoItX.ControlSend function, passing in the necessary parameters.

The code snippet below shows an example of how we can use the AutoItX.ControlSend function to send input to the **WoW** game window:

```
AutoItX.ControlSend("World of Warcraft", "", "", "key");
```

In this code, "**World of Warcraft**" is the title of the game window, "key" is the text or input to be sent, and it's also possible to change the mode in which to send the input, to allow for raw input, although for this use case, the default parameters works just fine.

By having access to this tools and functions, it is not only possible, but rather trivial to send the **NN** output as input to the running process.

However, when using ControlSend with the Sharp**NEAT** framework or any other multithreaded environment, it is essential to wrap the calls using a mutex to prevent race conditions that could cause errors or crashes. In the code snippet below, it is demonstrated how to use a mutex to safely send input to the **World of Warcraft** game window using ControlSend:

```
mutex.WaitOne();
AutoItX.ControlSend("World of Warcraft", "", "", "key");
mutex.ReleaseMutex();
```

Using AutoIt's ControlSend function in conjunction with the output from a **NN** allows us to automate complex user input tasks in video games, making it possible to create bots or other automated tools that can play the game more efficiently. By combining different technologies, we can achieve impressive results and explore the possibilities of automation and **AI** in video games and other applications.

4.5 Summary

This chapter focused on the development of a bot for game automation using the Sharp**NEAT** framework. The chapter explored various components involved in building the bot's Sensory/Input layer, Processing/**NEAT** layer, and Motor/Output layer. Additionally, it discussed the utilization of other tools and technologies such as Cheat Engine and AutoIt to explore the application's memory space.

The chapter began with an introduction to the topics covered, providing an overview of the subsequent sections. It emphasized the importance of constructing an effective bot for game automation, highlighting the significance of each layer in the overall architecture.

The building of the Sensory/Input layer was addressed, highlighting the utilization of tools like Cheat Engine for memory scanning and pointer scanning. These techniques allowed for the retrieval of game-related information from the memory space. Furthermore, countermeasures against reverse engineering were discussed to safeguard the integrity of the game.

Next, the chapter delved into the processing/**NEAT** layer, which involved defining the domain in the Sharp**NEAT** framework. The process of configuring and setting up **Neural Network** evolution was explained, along with the utilization of default configuration files as a starting point for parameter adjustment.

The navigation domain and strategy were then explored, focusing on the objective of getting the bot close enough to the target for initiating an attack. Inputs such as player and target positions, distance, and facing direction were considered in the design. The fitness function aimed to minimize the distance to the target, reflecting improved performance as the bot approached the objective.

Subsequently, the combat domain and strategy were discussed, wherein the bot's goal was to defeat the target enemy once engaged. Inputs such as player and target health, along with facing direction, were incorporated into the combat domain. The fitness function was designed to encourage the reduction of the target's health, signifying successful combat performance.

Users can seamlessly switch between these domains while preserving the state and information specific to each. This is facilitated through the utilization of static variables that store the current state of the active domain. Additionally, the system provides functionalities for saving and loading the state of each domain, allowing users to save their progress and reload it at a later time. The saved state, including the best genome and population, can be stored in separate files for independent retrieval. These file-based operations enhance usability, enabling users to persist their work, share results, and conduct experiments efficiently.

The atan2 function was used as a tool for determining if the player was facing the target. By calculating the angle between the player's position and the target's position, the atan2 function provided valuable information on the direction of the player's orientation relative to the target. This feature was crucial for effective decision-making in combat scenarios.

The chapter further explored the Motor/Output layer, highlighting the utilization of the AutoIt scripting language and its ControlSend function for automating user input in the Windows **GUI**. By leveraging AutoIt, the bot could send input commands to specific windows or controls, such as game windows. This functionality enabled the bot to perform actions within the game environment based on its decision-making processes.

In essence this chapter provided a comprehensive overview of the bot development process, emphasizing the integration of different layers and tools to create an efficient game automation system. The combination of several technologies facilitated the exploration of automation and **AI** in gaming applications. It showcased the potential for creating sophisticated bots capable of automating complex tasks in video games, opening up new avenues for research and innovation in the field.

Chapter 5

Application

5.1 Introduction

This chapter provides an overview of the architecture used in the project's application and explains how the different layers of the system interact. It describes the interaction between the **Graphical User Interface** (**GUI**), Sensory/Input layer, Processing/**NEAT** layer, and Motor/Output layer. The chapter also explores the management of context switching between the navigation and combat domains, discussing how the system handles the transition between these domains and manages the entity's behavior accordingly.

In addition, the chapter explains the setup process required for the application to execute effectively. It covers the necessary configurations and preparations needed to ensure a smooth execution of the system, as well as the steps involved in preparing the entity for its evolutionary process.

The chapter further discusses the data visualizations provided by the **GUI**, which enable users to monitor the evolving complexity of the Processing/**NEAT** layer. It explores the various visualizations such as graphs, charts, histograms, and rank plots, which help users gain insights into the behavior and performance of the **NEAT** algorithm.

Moreover, the chapter presents the end result of the project and highlights relevant observations made during its development. These observations provide valuable insights into the entity's learning and decision-making processes, showcasing its cognitive capabilities and the efficacy of the applied **GA**s.

Finally, the chapter concludes by summarizing the key points covered throughout, including the architecture, context switching, setup process, data visualizations, end result, and observations. It provides a comprehensive overview of the applications chapter and its contributions to the project.

5.2 Architecture

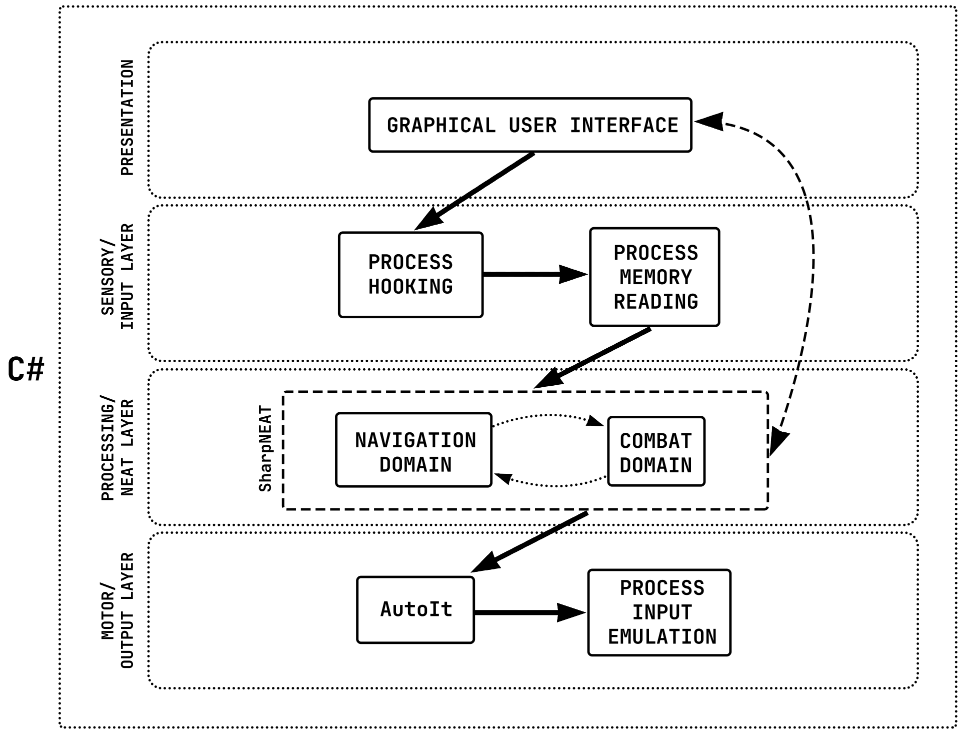


Figure 20: System's Architecture. C# as the main language for the project.

The figure 20 illustrates the comprehensive interaction between the various layers comprising the system architecture. This visualization aims to enhance the reader's understanding of the system's internal workings.

The application incorporates an intuitive and user-friendly **Graphical User Interface** (**GUI**) to facilitate user interaction. One of the initial points of engagement involves hooking onto the target process, a straightforward process accomplished seamlessly through the **GUI**. By establishing this connection, the application gains the capability to directly access and read data from the target process memory. This crucial functionality enables the domains developed using the Sharp**NEAT** framework to acquire the necessary information to train, evaluate, and evolve individuals within the system. Subsequently, the preferred actions or outputs of these individuals are emulated back into the running process. This emulation is made possible by leveraging AutoIt's **DLL**, which grants the application the ability to execute input within the target process space.

There exists a bidirectional arrow linking the Sharp**NEAT** domains to the **GUI**. This linkage underscores the fact that the **GUI** primarily comprises resources provided by the Sharp**NEAT** framework, supplemented by user-specific customizations. Furthermore, it signifies the architectural decision to indirectly manipulate the **GUI**, thereby enabling efficient and intuitive context switching between domains. This design choice arises from the interweaving of logic within the framework, as it directly retrieves information from the textual data presently displayed in the **GUI**.

The system architecture is meticulously crafted to ensure seamless integration and communication between the **GUI**, Sharp**NEAT** domains, and the target process. The user-friendly **GUI** empowers users to effortlessly hook onto the target process, extract pertinent data, train and evaluate individuals using Sharp**NEAT**, and effectively emulate preferred actions within the target process. By adopting this architecture, the application embodies a sophisticated framework that facilitates complex interactions while maintaining user-friendly functionality.

Process Hooking

Processing hooking is a technique used to intercept and modify the behavior of a target process. It involves injecting custom code into the process's execution flow to gain control and access its memory. This technique is commonly employed to perform various tasks, such as debugging, reverse engineering, and system monitoring.

To implement processing hooking, several steps need to be followed, as explained on section 4.2.4. Firstly, a hooking mechanism is employed to intercept function calls or system events within the target process. This can be achieved through various means, including code injection, dynamic linking, or **API** hooking Sikorski and Honig [2012].

Once the hook is established, the custom code is executed in response to the intercepted events. This code can perform a range of actions, such as modifying function parameters, redirecting execution flow, or reading memory from the target process.

In this case code injection was used to hook and read memory from the hooked process. By gaining control over the target process's execution, the hooking code can access and extract specific data stored in its memory.

5.3 Domains and Automatic Context Switching

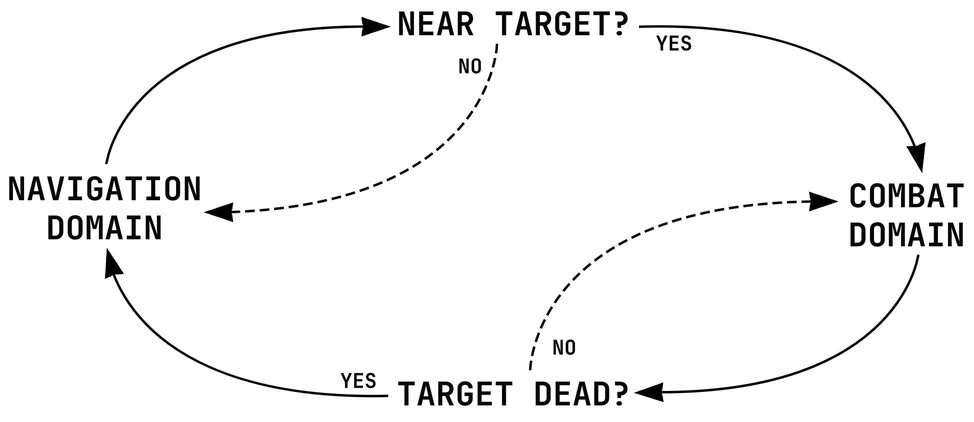


Figure 21: Automatic Context Switching Between Domains.

To enable seamless transition between domains, a sophisticated system was implemented to preserve the state of each domain, allowing for later resumption of progress. Additionally, continuous monitoring in the background ensures accurate determination of the active domain.

A dedicated background thread operates in parallel, consistently assessing whether the specified objective of the current domain has been achieved. Based on this evaluation, the system dynamically switches between domains or continues within the same domain for subsequent generations.

Each domain incorporates a static variable functioning as a flag, signifying the fulfillment or pending status of the domain's primary objective. When this flag is modified, the background thread detects the change and triggers the **GUI** to transition to the corresponding domain in the Sharp**NEAT** experiment. Consequently, the framework is primed to recognize and accommodate the desired configuration of the experiment. The state of the domain is loaded from static variables that retain essential information concerning the evolution algorithm and the entire population of individuals specific to the domain.

By adopting this approach, the system facilitates efficient domain switching while preserving the integrity of each domain's progress. The background thread diligently monitors the status of the current domain, ensuring smooth transitions and uninterrupted execution of the evolving process. This design not only enhances the system's flexibility but also enables effective coordination between the **GUI** and the Sharp**NEAT** framework.

The implementation of automatic context switching between domains demonstrates a sophisticated mechanism that enables the system to seamlessly transition between navigation and combat domains. This dynamic functionality, facilitated by the background thread and the state-preservation capabilities, significantly enhances the overall performance and adaptability of the system.

5.4 Post Launch Application Priming

In this section, the necessary steps for preparing the application and witnessing the emergence of the **AI** entity will be elucidated.

Upon starting the application, there are a few essential steps that need to be completed before the entity can commence its actions. These preliminary procedures ensure the proper alignment of all components involved. The following steps outline the initial setup process:

1. **Attach to Process**: The user begins by clicking the "Attach" button, which opens a pop-up window displaying the available instances that can be hooked into. From this list, the user selects a suitable instance to establish the necessary connection as depicted in figure 23.

2. **Select the Domain**: Within the application's **Graphical User Interface** (**GUI**), the user navigates to the "Experiment/Task" drop-down menu and chooses the appropriate domain. For example, in the case of the navigation domain, the user selects "WoW **NEAT** Cheese[1]" from the options as depicted in figure 24.

3. **Load Default Parameters**: To ensure consistent configuration, the user selects the "Load Experiment Default Parameters" button within the **GUI** as depicted in figure 25. This action populates the **GUI** fields with the default parameters defined in the configuration file.

4. **Create Initial Population**: The user has the option to generate an initial population based on the configuration settings. By selecting the "Create Random Population" button, the application generates a population that adheres to the specified parameters[2] as depicted in figure 26. Alternatively the user can load a previously saved population. Additionally, the user can modify these settings directly within the **GUI** if so desired as depicted in figure 28.

With these preparations complete, the application is primed for execution. By selecting the "Start/-Continue" button, the **AI** entity springs to life as depicted in figure 27, and its behavior within the target application becomes observable.

By following these systematic steps, users can effectively initialize the application and observe the **AI** entity's dynamic behavior as it evolves and interacts with the designated environment.

[1] In video games, the term "cheese" is usually reserved to refer to unfair, unorthodox strategies.

[2] On the experiments *.config.json file.

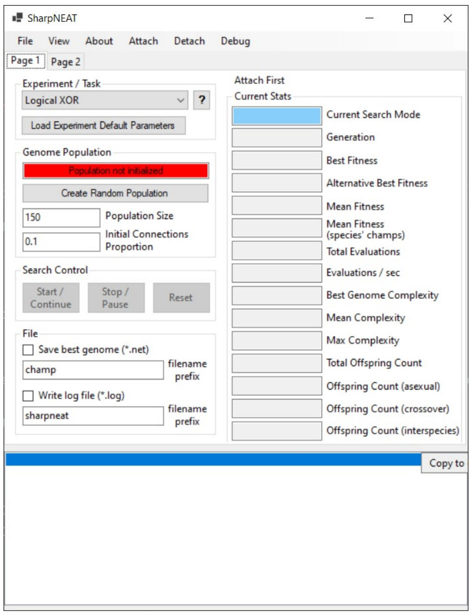


Figure 22: Default state of the application at startup.

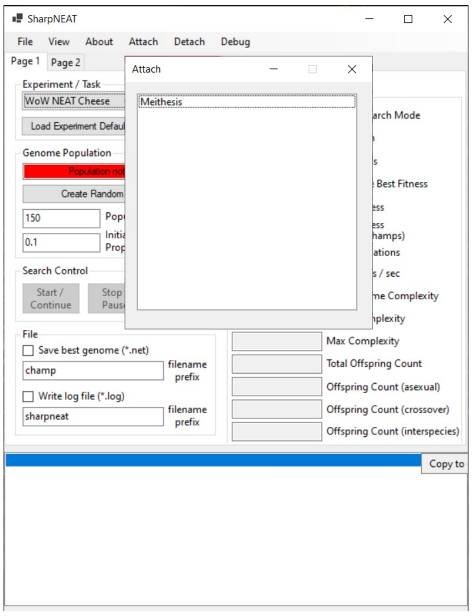


Figure 23: Hooking the application the relevant **WoW** instance. We use the character's in-game name to better discern viable instances.

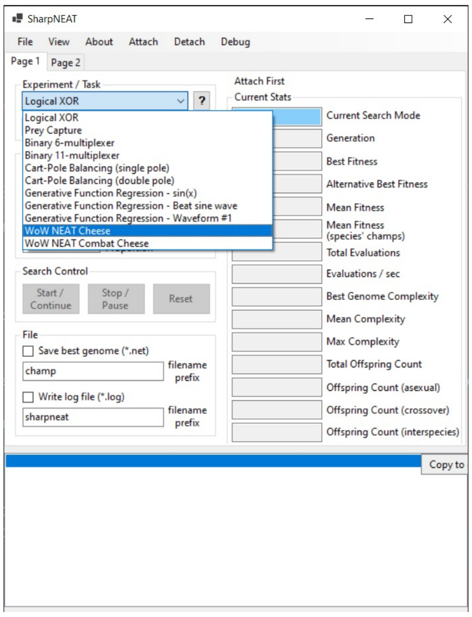


Figure 24: Changing the Experiment/Task to **WoW NEAT** Cheese. - The Navigation Domain

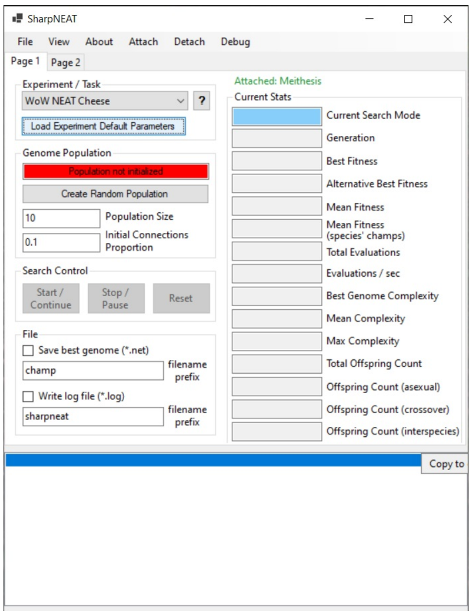
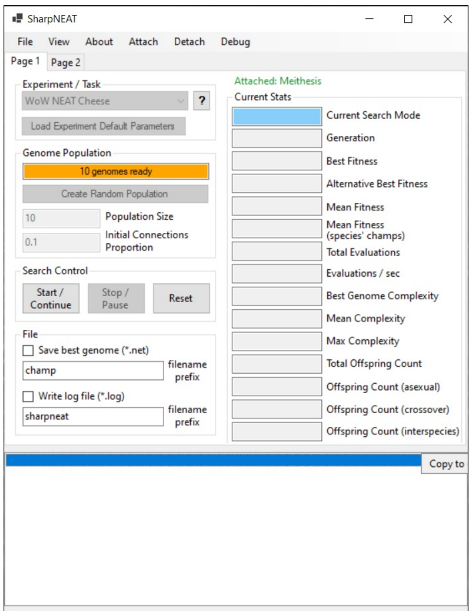


Figure 25: Loading the experiment default parameters that we set in the configuration file for the selected experiment.



Attached: Meithesis

Experiment / Task: WoW NEAT Cheese

Genome Population: 10 genomes ready

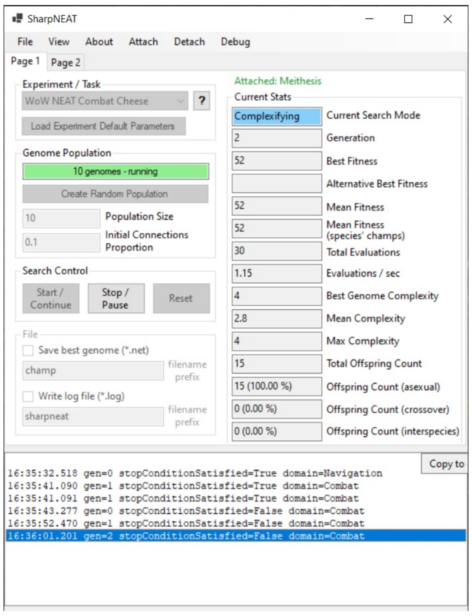
10	Population Size
0.1	Initial Connections Proportion

File

- [ ] Save best genome (*.net) — champ (filename prefix)
- [ ] Write log file (*.log) — sharpneat (filename prefix)

Current Stats: Current Search Mode, Generation, Best Fitness, Alternative Best Fitness, Mean Fitness, Mean Fitness (species' champs), Total Evaluations, Evaluations / sec, Best Genome Complexity, Mean Complexity, Max Complexity, Total Offspring Count, Offspring Count (asexual), Offspring Count (crossover), Offspring Count (interspecies)

Figure 26: Creating a random population of genomes following the guidelines set in place.



Attached: Meithesis

Experiment / Task: WoW NEAT Combat Cheese

Genome Population: 10 genomes - running

10	Population Size
0.1	Initial Connections Proportion

File

- [ ] Save best genome (*.net) — champ (filename prefix)
- [ ] Write log file (*.log) — sharpneat (filename prefix)

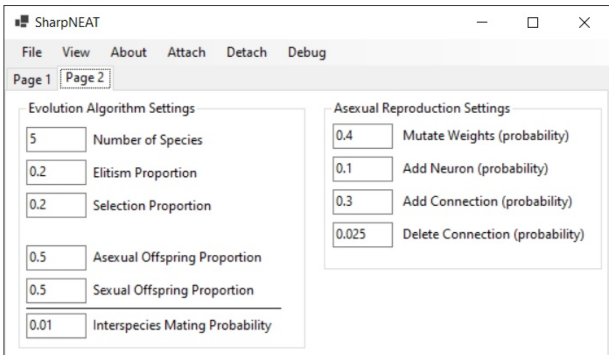
Current Stats

Complexifying	Current Search Mode
2	Generation
52	Best Fitness
	Alternative Best Fitness
52	Mean Fitness
52	Mean Fitness (species' champs)
30	Total Evaluations
1.15	Evaluations / sec
4	Best Genome Complexity
2.8	Mean Complexity
4	Max Complexity
15	Total Offspring Count
15 (100.00 %)	Offspring Count (asexual)
0 (0.00 %)	Offspring Count (crossover)
0 (0.00 %)	Offspring Count (interspecies)

```
16:35:32.518 gen=0 stopConditionSatisfied=True domain=Navigation
16:35:41.090 gen=1 stopConditionSatisfied=True domain=Combat
16:35:41.091 gen=1 stopConditionSatisfied=True domain=Combat
16:35:43.277 gen=0 stopConditionSatisfied=False domain=Combat
16:35:52.470 gen=1 stopConditionSatisfied=False domain=Combat
16:36:01.201 gen=2 stopConditionSatisfied=False domain=Combat
```

Figure 27: Starting the Experiment/Task on the Navigation Domain, eventually switching to the Combat Domain.

There is also the possibility to change the following configurations directly in the **GUI** before starting[3]:



Evolution Algorithm Settings

5	Number of Species
0.2	Elitism Proportion
0.2	Selection Proportion
0.5	Asexual Offspring Proportion
0.5	Sexual Offspring Proportion
0.01	Interspecies Mating Probability

Asexual Reproduction Settings

0.4	Mutate Weights (probability)
0.1	Add Neuron (probability)
0.3	Add Connection (probability)
0.025	Delete Connection (probability)

Figure 28: The possibility of changing Evolution and Reproduction settings directly through the **GUI**.

[3] By selecting “Page 2” and then altering the intended parameters.

5.5 GUI Live Data Visualization

The SharpNEAT framework offers a range of powerful tools for live data visualization, providing users with valuable insights into the ongoing NEAT algorithm.

One of these tools is the "Best Genome" visualization, which presents a graph representation of the NN (i.e. the phenotype) of the current best individual in the domain. This visual depiction showcases the input neurons positioned at the top, the output neurons at the bottom, and the hidden neurons and their respective connections in between. By observing this graph, users can gain a deeper understanding of the structure and connectivity of the NN.

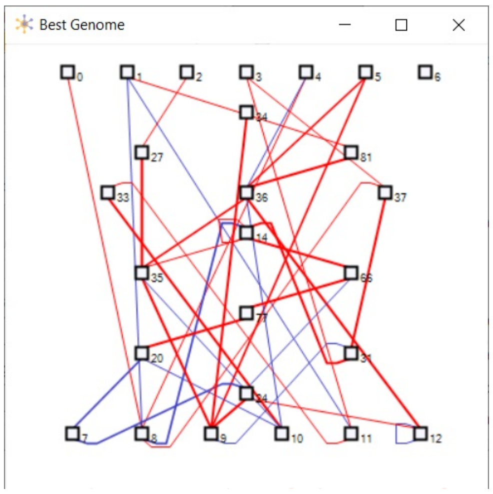


Figure 29: Seeing the current best genome phenotype (NN) of the Navigation Domain through the GUI.

The "Charts" view in the GUI provides three major types of charts for live data visualization. The "Time Series" charts offer three options: fitness (both best and mean), complexity (both best and mean), and evaluations per second. These charts allow users to track the progress of fitness and complexity measures over time, providing insights into the performance and evolution of the NEAT algorithm throughout the execution.

The "Histograms" section within the Charts view offers a range of histograms for visual analysis. Users can explore the distribution of species sizes, mean fitness, mean complexity, and genome fitness and complexity. These histograms enable users to examine the population characteristics and understand the variation and distribution of fitness and complexity values within the evolving population.

Lastly, the "Rank Plots" in the GUI provide further visualization options. Users can observe species size by rank, species fitness by rank (both best and mean), species complexity by rank (both best and mean), and genome fitness and complexity by rank. These rank plots offer a comprehensive view of how species and genomes are ranked and provide insights into the relative performance and complexity of different individuals.

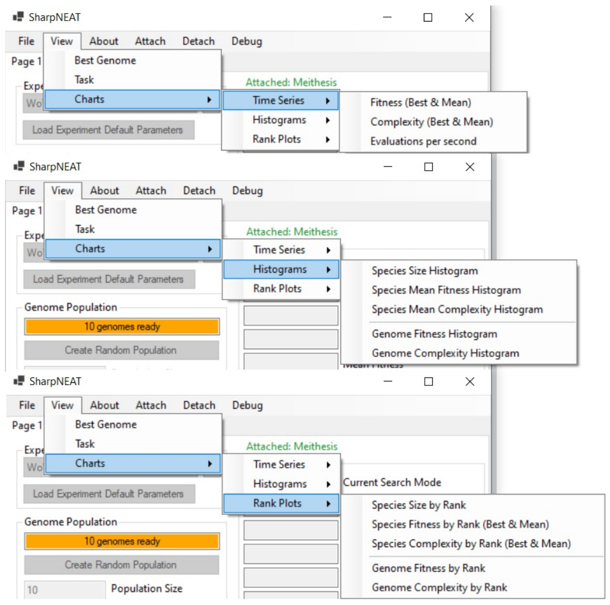


Figure 30: Time Series, Histograms and Rank Plots that can be monitored through the **GUI**.

By leveraging these visualizations, users can gain valuable insights into the ongoing **NEAT** algorithm, monitor its progress, and make informed decisions based on the observed trends and patterns. The combination of graph representations, time series charts, histograms, and rank plots empowers users to analyze and understand the dynamic behavior of the **NEAT** algorithm in a more professional and academic manner.

5.6 End Result and Observations

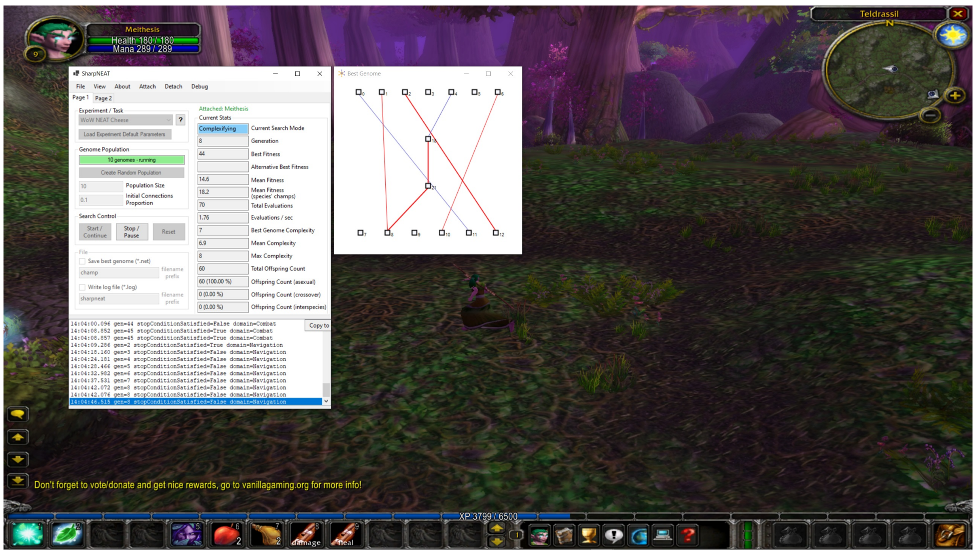
During the initial stages of development, the entity experienced subpar performance due to a lack of clear separation between domains and specific objectives. Without a well-defined sense of direction, the entity struggled to process the input and select appropriate outputs, resulting in confusion and inefficiency. Despite continuous modifications to the fitness function and feedback mechanisms, the entity's behavior remained far from the intended objectives, exhibiting erratic and unpredictable patterns.

To overcome this challenge, a significant breakthrough was achieved by adopting a divide-and-conquer approach. The original objective was divided into smaller, more manageable sub-objectives, leading to improved performance. Two distinct domains, namely navigation and combat, emerged as separate cognitive zones within the entity's framework. This strategic partitioning allowed for the development of desired behaviors and demonstrated noticeable generational improvements over time.

The subsequent observations and details gathered during the development process provided valuable insights into the entity's learning and decision-making processes. These observations shed light on the intricacies of its cognitive capabilities and the effectiveness of the applied **GA**. The observations deepened the understanding of how the entity adapts, learns, and makes decisions, providing valuable information for further refinement and optimization.

Overall, the adoption of a divide-and-conquer approach and the emergence of distinct navigation and combat domains proved to be pivotal in improving the entity's performance. It will now be discussed some observations that were made throughout the development process that led to the enhanced understanding of the entity's cognitive abilities and the efficacy of the **GA** employed.

5.6.1 Navigation Domain



Experiment / Task: WoW NEAT Cheese
Load Experiment Default Parameters
Genome Population: 10 genomes - running
Create Random Population
Population Size: 10
Initial Connections Proportion: 0.1
Search Control: Start / Continue, Stop / Pause, Reset
File: Save best genome (*.net), champ filename prefix; Write log file (*.log), sharpneat filename prefix

Attached: Meithesis

Current Stats

Complexifying	Current Search Mode
8	Generation
44	Best Fitness
	Alternative Best Fitness
14.6	Mean Fitness
18.2	Mean Fitness (species' champs)
70	Total Evaluations
1.76	Evaluations / sec
7	Best Genome Complexity
6.9	Mean Complexity
8	Max Complexity
60	Total Offspring Count
60 (100.00 %)	Offspring Count (asexual)
0 (0.00 %)	Offspring Count (crossover)
0 (0.00 %)	Offspring Count (interspecies)

```
14:04:00.096 gen=44 stopConditionSatisfied=False domain=Combat
14:04:08.852 gen=45 stopConditionSatisfied=True domain=Combat
14:04:08.857 gen=45 stopConditionSatisfied=True domain=Combat
14:04:09.286 gen=2 stopConditionSatisfied=True domain=Navigation
14:04:18.160 gen=3 stopConditionSatisfied=False domain=Navigation
14:04:24.181 gen=4 stopConditionSatisfied=False domain=Navigation
14:04:28.466 gen=5 stopConditionSatisfied=False domain=Navigation
14:04:32.982 gen=6 stopConditionSatisfied=False domain=Navigation
14:04:37.531 gen=7 stopConditionSatisfied=False domain=Navigation
14:04:42.072 gen=8 stopConditionSatisfied=False domain=Navigation
14:04:42.076 gen=8 stopConditionSatisfied=False domain=Navigation
14:04:46.515 gen=8 stopConditionSatisfied=False domain=Navigation
```

Figure 31: Showing the application in action currently on the Navigation Domain.

The development of the navigation domain yielded few unexpected challenges, as the cognitive region(i.e. **NEAT** and the custom interface) responsible for navigation demonstrated effective strategies in navigating the virtual environment. Notably, the emergence of successful navigation strategies showcased the organism's ability to adapt and overcome obstacles through genetic mutations.

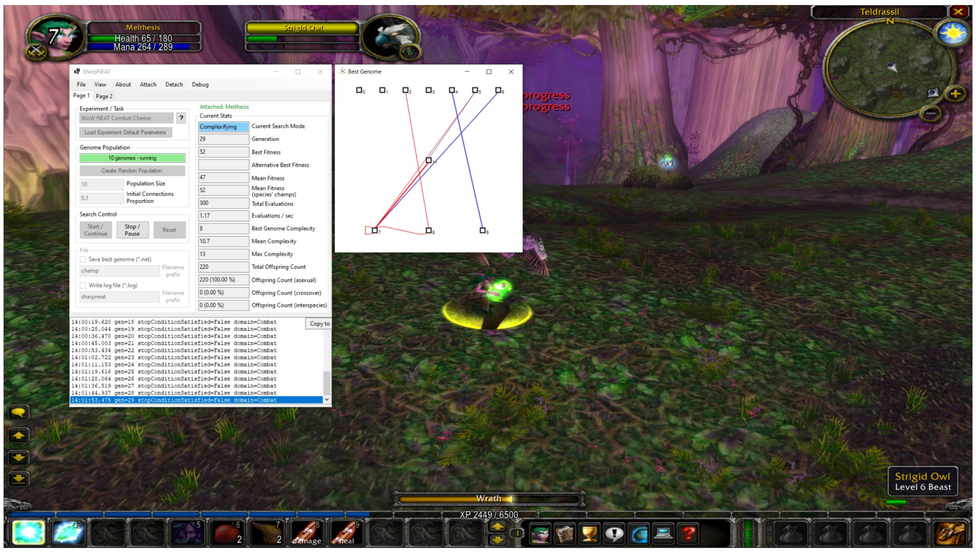
To ensure continuous progress, the system incorporates a periodic change in the target object, preventing the entity from becoming fixated on an unreachable objective. Although the initial movements may appear random, over time, the entity refines its decision-making process, exhibiting increasingly precise and natural movement patterns.

In certain instances, it was observed that the entity appeared to be trapped in a repetitive rotation loop. However, with successive generations, the entity successfully transitioned to more effective strategies, enabling it to break free from the loop and resume traversing a more conventional path.

Moreover, it is worth noting that the navigation domain exhibited a trend of diminishing returns, whereby noticeable improvements in performance became less discernible after approximately the 100th generation. While initial iterations showed favorable progress and visible advancements, subsequent generations demonstrated relatively smaller incremental enhancements that were not easily perceivable.

This observation highlights the potential limitations and constraints of the applied **GA** within the navigation domain. After a certain point, the entity's navigation capabilities may have reached a plateau, suggesting that further optimizations or alternative approaches may be necessary to achieve significant breakthroughs beyond a certain generational threshold.

5.6.2 Combat Domain



Experiment / Task: WoW NEAT Combat Cheese
Load Experiment Default Parameters
Genome Population: 10 genomes - running
Create Random Population
10 Population Size
0.1 Initial Connections Proportion
Search Control: Start / Continue | Stop / Pause | Reset
File
Save best genome (*.net) — champ (filename prefix)
Write log file (*.log) — sharpneat (filename prefix)

Attached: Meithesis
Current Stats

Complexifying	Current Search Mode
29	Generation
52	Best Fitness
	Alternative Best Fitness
47	Mean Fitness
52	Mean Fitness (species' champs)
300	Total Evaluations
1.17	Evaluations / sec
8	Best Genome Complexity
10.7	Mean Complexity
13	Max Complexity
220	Total Offspring Count
220 (100.00 %)	Offspring Count (asexual)
0 (0.00 %)	Offspring Count (crossover)
0 (0.00 %)	Offspring Count (interspecies)

```
14:00:19.620 gen=18 stopConditionSatisfied=False domain=Combat
14:00:28.044 gen=19 stopConditionSatisfied=False domain=Combat
14:00:36.470 gen=20 stopConditionSatisfied=False domain=Combat
14:00:45.003 gen=21 stopConditionSatisfied=False domain=Combat
14:00:53.434 gen=22 stopConditionSatisfied=False domain=Combat
14:01:02.722 gen=23 stopConditionSatisfied=False domain=Combat
14:01:11.153 gen=24 stopConditionSatisfied=False domain=Combat
14:01:19.616 gen=25 stopConditionSatisfied=False domain=Combat
14:01:28.064 gen=26 stopConditionSatisfied=False domain=Combat
14:01:36.519 gen=27 stopConditionSatisfied=False domain=Combat
14:01:44.937 gen=28 stopConditionSatisfied=False domain=Combat
14:01:53.475 gen=29 stopConditionSatisfied=False domain=Combat
```
Copy to

Figure 32: Showing the application in action currently on the Combat Domain.

The combat domain, while conceptually simpler in terms of cognitive capacity compared to the navigation domain, unveiled intricate and nuanced behaviors that were not initially anticipated during the preliminary trial runs of the system.

A notable observation in the combat domain was the emergence of unintended behavior in certain generations. It was observed that some individuals exhibited a tendency to lock their direction to the target and primarily focused on healing themselves rather than prioritizing damage to the target, which is the primary objective and the behavior that yields greater rewards. This suboptimal strategy appeared to arise due to lower mutation rates in the early generations, constraining the exploration of alternative strategies. However, through increased mutation rates and greater room for exploration, the organisms adapted and successfully adopted a more efficient strategy that emphasized damaging the target while occasionally healing themselves to mitigate the risk of failure or death.

Similar to the navigation domain, the combat domain also demonstrated diminishing returns over time. Beyond a certain point, generational improvements became significantly limited or barely noticeable, and in some cases, less effective strategies emerged. This suggests the existence of an optimal performance peak, beyond which further enhancements become challenging to attain. While one possible approach to counteract this effect would be to disable mutation and crossover, maintaining the organism at its peak performance, identifying and capturing this peak can be challenging in practice. Moreover, fixing the strategy to the current best one could introduce rigidity to the phenotypes, potentially restricting adaptability and responsiveness to changing conditions.

5.7 Summary

Chapter 5 provides a comprehensive overview of the implementation and results of the **AI** entity, focusing on the navigation and combat domains. The chapter begins by describing the system architecture, highlighting the comprehensive interaction between different layers and the user-friendly **GUI** that facilitates user interaction.

The chapter then delves into the concept of automatic context switching between domains, which allows for efficient and seamless transitions between navigation and combat. It explains the meticulous design of the system to preserve the state of each domain and ensure uninterrupted execution of the evolving process. This sophisticated mechanism, facilitated by a background thread, enables the system to dynamically switch between domains based on the fulfillment of objectives.

Post launch application priming is another important aspect covered in this chapter. It outlines the necessary steps to prepare the application for the emergence of the **AI** entity. These steps include attaching to the target process, selecting the domain, loading default parameters, and creating an initial population. By following these steps, users can effectively initialize the application and observe the **AI** entity's dynamic behavior.

The chapter also discusses **GUI** live data visualization, which plays a crucial role in providing users with valuable insights into the ongoing **NEAT** algorithm. It explains various visualization tools offered by the Sharp**NEAT** framework, including the "Best Genome" visualization, time series charts, histograms, and rank plots. These visualizations enable users to monitor the progress, performance, and distribution of fitness and complexity measures, gaining a deeper understanding of the **NEAT** algorithm's behavior.

Finally, the chapter concludes with an analysis of the end results and observations. It highlights the challenges faced during the initial stages of development, the breakthrough achieved through the divide-and-conquer approach, and the entity's adaptive capabilities in both navigation and combat domains. The observations reveal the entity's learning and decision-making processes, as well as the limitations and diminishing returns observed over successive generations.

Chapter 6

Conclusions and future work

6.1 Conclusions

The development and exploration of the **AI**-based entity within the virtual realm of the **WoW** video game have provided valuable insights into the challenges and potential solutions involved in creating intelligent agents capable of navigating and combating within complex virtual environments.

One of the critical factors contributing to the success of the project was the subdivision of the problem into two distinct domains: navigation and combat. By breaking down the monolithic objective into smaller, more concrete objectives, the entity was able to focus on specific tasks and develop effective strategies within each domain. This approach allowed for a more targeted and smooth exploration of the problem space, leading to improved performance and generational advancements.

The utilization of the Sharp**NEAT** framework played a vital role in enabling the development of the **AI**-based entity. The framework's live data visualization tools provided real-time monitoring and analysis of the entity's behavior, facilitating the identification of patterns, trends, and areas for improvement. Additionally, the flexible architecture of the framework provided a solid foundation for the development process allowing for personal user modification which led to the possibility of concurrent exploration of different domains and objectives. Notably, this capability was not initially included or anticipated in the framework's original design.

The analysis of the process memory space and the preservation of crucial memory data locations have enabled the creation of a virtual entity capable of emulating user behavior by utilizing acquired memory data as sensory inputs, having those inputs processed by **NEAT** and by integrating AutoIt it was made possible the ability to emulate the response, i.e. the outputs, back into the running process.

In conclusion, this dissertation has presented an unusual approach for the development of an **AI**-based entity capable of simulating user behavior within the **WoW** video game. As technology continues to advance, further research and experimentation in this area have the potential to lead for the development of more sophisticated and intelligent virtual entities in the future.

6.2 Prospect for future work

The field of **AI** is rapidly evolving, and recent advancements in **Natural Language Processing** (**NLP**) have opened up new possibilities for enhancing **AI** systems. Building upon the **AI** framework[1] developed for the **WoW** environment, there is a potential avenue for future work that involves integrating the application with the ChatGPT **API** and leveraging its language processing capabilities[2].

Instead obtaining possible solutions by evolving **NN**s using the **NEAT** technique, a novel approach could involve retaining the sensory and motor layers of the existing framework while replacing the processing layer with an integration of the ChatGPT **API**. This would entail sending **HTTP** POST requests with the data being read from the **WoW** process, specifically instructing the ChatGPT model on the desired actions based on the provided inputs. The returned response from the language model would provide the **AI** agent with the best possible actions to take within the **WoW** environment.

The integration with the ChatGPT **API** would enable the **AI** agent to obtain context-aware and language-based instructions for decision-making and action execution. Every second, the **WoW** process data would be sent as input to the ChatGPT model, which would generate a response containing the recommended actions to be taken. These actions, represented as key presses or commands, would then be parsed and injected or emulated back into the **WoW** process, creating a continuous and operative entity.

This future work prospect leverages the recent advancements in **NLP**, particularly the capabilities of OpenAI's ChatGPT model 3.5, which has demonstrated impressive language understanding and generation capabilities. The low inference time and complexity of the ChatGPT model 3.5 make it suitable for real-time decision-making and action generation in dynamic environments like **WoW**.

By integrating the **AI** framework with the ChatGPT **API**, the **AI** agent would benefit from more sophisticated and contextually aware decision-making, potentially enhancing its performance and adaptability within the **WoW** environment. Furthermore, this approach opens up opportunities for exploring the combination of traditional sensory data with natural language understanding, allowing for a more comprehensive and versatile **AI** system.

In conclusion, future work involving the integration of the **WoW** **AI** framework with the ChatGPT **API** holds great potential for enhancing the decision-making and action execution capabilities of the **AI** agent. The integration of **NLP** advancements in **AI** systems enables more natural and context-aware interactions within virtual worlds like **WoW**, paving the way for exciting research and advancements in the field of **AI** for complex environments.

[1] Consider the AI framework, in this case, as the sum of all the tools developed for the sensory, processing and motor layers.

[2] One of the current downsides involves having to pay OpenAI for a subscription to have access to their **API**.

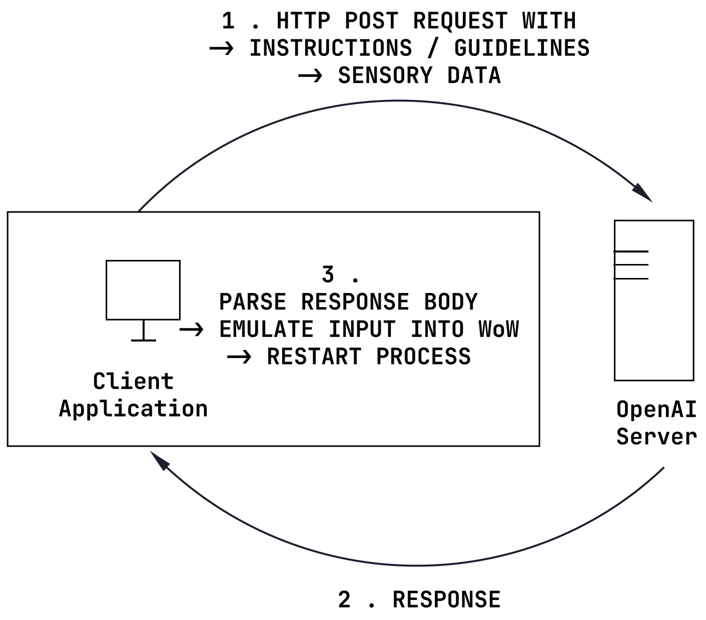


Figure 33: Diagram demonstrating how a **LLM** can be used as the processing layer.

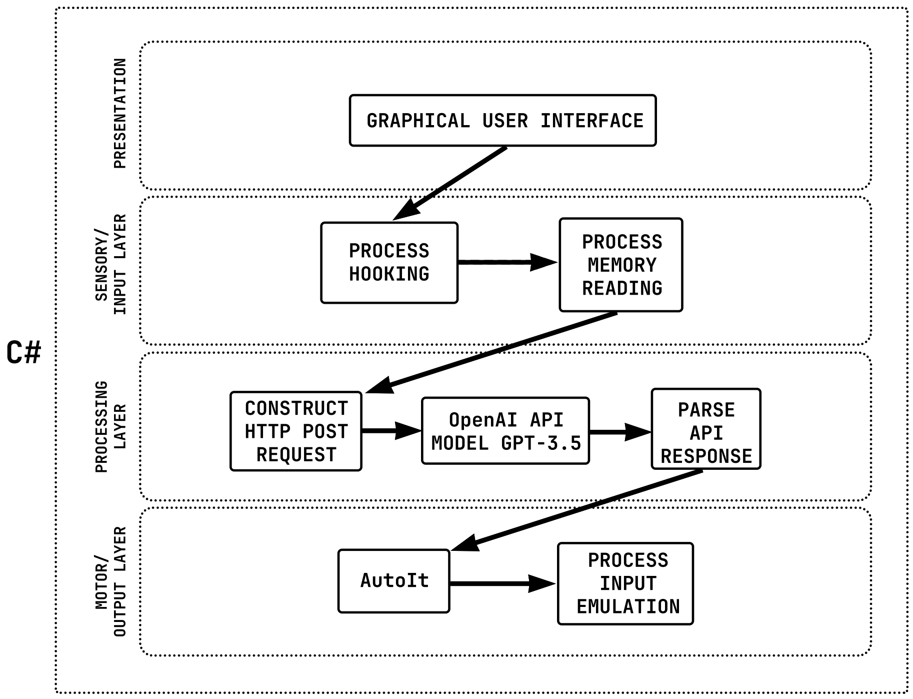


Figure 34: Updated architecture now using a **LLM** as the core component of the processing layer.

Bibliography

Game Freaks 365. How venezuelans took over video game gold farming, 2021. URL `https://gamefreaks365.com/how-venezuelans-took-over-video-game-gold-farming/`.

Alexander Asteroth Adam Gaier and Jean-Baptiste Mouret. Are quality diversity algorithms better at generating stepping stones than objective-based search? In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 115–116, 2019.

Greg Aloupis et al. Classic nintendo games are (computationally) hard. In *Theoretical Computer Science 586*, pages 135–160, 2015.

Steve Battle. Imitation learning. In *Principles of Robot Autonomy*. University of the West of England, Bristol, 2018.

Jonathan C Brant and Kenneth O Stanley. Minimal criterion coevolution: a new approach to open-ended search. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 67–74, 2017.

Leo Cazenille. Comparing reliability of grid-based quality-diversity algorithms using artificial landscapes. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 249–250, 2019.

Matheus G Cordeiro et al. A minimal training strategy to play flappy bird indefinitely with neat. In *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pages 21–28. IEEE, 2019.

Antonios Liapis Daniele Gravina and Georgios N Yannakakis. Quality diversity through surprise. In *IEEE Transactions on Evolutionary Computation 23.4*, pages 603–616, 2018.

Antonios Liapis Daniele Gravina and Georgios N Yannakakis. Blending notions of diversity for map-elites. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 117–118, 2019.

Charles Darwin. On the origin of species. 1859.

Gérard Dreyfus. *Neural Networks: Methodology and Applications*. Springer-Verlag, 2005.

The Economist. Venezuela's paper currency is worthless, so its people seek virtual gold, 2019. URL https://www.economist.com/the-americas/2019/11/21/venezuelas-paper-currency-is-worthless-so-its-people-seek-virtual-gold.

Blizzard Entertainment. Wow, 2004. URL https://worldofwarcraft.com/en-gb/story/timeline/chapter-6.

EvoStar. Applications of evolutionary computation 22nd international conference. In *Theoretical Computer Science and General Issues*, 2019.

Benoît Jacob Gaël Guennebaud et al. *Eigen v3*. 2010.

Iczelion. Tutorial 24: Windows hooks, 2002. URL https://web.archive.org/web/20080801155929/http://win32assembly.online.fr/tut24.html.

Kir Birger Jessica Lowell and Sergey Grabkovsky. Comparison of neat and hyperneat on a strategic decision-making problem. In *CS 6140 Final Project*, 2011.

Sergey Karakovskiy Julian Togelius and Robin Baumgarten. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

Chao-Cheng Chen Jyh-Jong Tsay and Jyh-Jung Hsu. Evolving intelligent mario controller by reinforcement learning. In *2011 International Conference on Technologies and Applications of Artificial Intelligence*, pages 266–272. IEEE, 2011.

Janusz Kacprzyk. Artificial neural network modelling. In *Studies in Computational Intelligence*, page 628, 2016.

Lukasz Kaiser et al. Model-based reinforcement learning for atari. 2019.

David D' Ambrosio Kenneth O. Stanley and Jason Gauci. A hypercube-based indirect encoding for evolving large-scale neural networks. In *Artificial Life journal*. MIT Press, 2009.

Risto Miikkulainen Kenneth O. Stanley and Bobby D. Bryant. Real-time neuroevolution in the nero video game. In *IEEE Transactions on Evolutionary Computation*, pages 653–668, 2005a.

Risto Miikkulainen Kenneth O. Stanley and Bobby D. Bryant. Evolving neural network agents in the nero video game. In *IEEE Symposium on Computational Intelligence and Games*, 2005b.

Risto Miikkulainen Kenneth O. Stanley and Xuan Huy Hoang. Real-time challenge balance in an rts game using rtneat. In *Proceedings of the 2008 Conference on Computer and Games*, 2008.

Risto Miikkulainen Kenneth O. Stanley et al. Nero 2.0, 2003. URL https://nn.cs.utexas.edu/nero/.

John R Koza. Genetic programming as a means for programming computers by natural selection. In *Statistics and computing*, pages 87–112, 1994.

Oliver Kramer. *Genetic algorithm essentials*. Springer, vol. 679 edition, 2017.

Geoffrey Lee et al. Learning a super mario controller from examples of human play. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2014.

Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. In *Evolutionary computation 19.2*, pages 189–223. IEEE, 2011.

Joel Lehman et al. Safe mutations for deep and recurrent neural networks through output gradients. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 117–124, 2018.

AutoIt Consulting Ltd. Autoit, 2023. URL https://www.autoitscript.com/site/.

Hector Marco-Gisbert and Ismael Ripoll Ripoll. Address space layout randomization next generation. In *Applied Sciences*, 2019.

Steve Marschner and Peter Shirley. In *Fundamentals of Computer Graphics*. A. K. Peters, Ltd.63 South Avenue Natick, MA, United States, 2009.

Ramviyas Parasuraman Michele Colledanchise and Petter Ögren. Learning of behavior trees for autonomous agents. In *IEEE Transactions on Games 11.2*, pages 183–189, 2018.

Mnih et al. Human-level control through deep reinforcement learning, 2015. URL https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf.

Arash Mohammadi et al. Openga, a c++ genetic algorithm library. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2051–2056. IEEE, 2017.

D. J. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 762–767. Morgan Kaufmann, 1989.

Jean-Baptiste Mouret and Jeff Clune. *Illuminating search spaces by mapping elites*. 2015.

Silke Höhl Neil Urquhart and Emma Hart. An illumination algorithm approach to solving the micro-depot routing problem. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1347–1355, 2019.

S Allwyn Jones P Sibi and P Siddarth. Analysis of different activation functions using back propagation neural networks. In *Journal of theoretical and applied information technology*, pages 1264–1268, 2013.

Diego Perez et al. Evolving behaviour trees for the mario ai competition using grammatical evolution. In *European Conference on the Applications of Evolutionary Computation*, pages 123–132. Springer, 2011.

N. J. Radcliffe. Neural computing and applications. In *Genetic set recombination and its application to neural network topology optimisation*, pages 67–90, 1993.

Felix Richter. Are you not entertained?, 2022. URL `https://www.statista.com/chart/22392/global-revenue-of-selected-entertainment-industry-sectors/`.

Stuart J Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Third international edition edition, 2010.

Arthur L Samuel. Some studies in machine learning using the game of checkers. In *IBM Journal of research and development*, pages 210–229, 1959.

Jits Schilperoort et al. Learning to play pac-xon with q-learning and two double q-learning variants. In *2018 IEEE Symposium Series on Computational Intelligence*, pages 1151–1158. IEEE, 2018.

C Daniel Gelatt Scott Kirkpatrick and Mario P Vecch. Optimization by simulated annealing. In *science*, pages 671–680, 1983.

Jie Shao et al. Is normalization indispensable for training deep neural network? In *Advances in Neural Information Processing Systems*, 2020.

James Shewmaker. Analyzing dll injection, 2006.

Hisashi Shimodaira. *DCGA: A diversity control oriented genetic algorithm*. 1997.

Michael Sikorski and Andrew Honig. In *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

David Silver et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. In *Science 362.6419*, pages 1140–1144, 2018.

Thamarai Selvi Somasundaram et al. Double q–learning agent for othello board game. In *2018 Tenth International Conference on Advanced Computing (ICoAC)*, pages 216–223. IEEE, 2018.

Kenneth O Stanley and Joel Lehman. *Why greatness cannot be planned: The myth of the objective*. Springer, 2015.

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. 2002.

Kenneth O. Stanley et al. The hybercube-based neuroevolution of augmenting topologies (hyperneat) users page, 2015. URL `http://eplex.cs.ucf.edu/hyperNEATpage/`.

Alban Laflaquière Stephane Doncieux and Alexandre Coninx. Novelty search: a theoretical perspective. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 99–106, 2019.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Matthew Szudzik. “an elegant pairing function. In *Wolfram Research (ed.) Special NKS 2006 Wolfram Science Conference*, pages 1–12, 2006.

Rasmus K Ursem. Diversity-guided evolutionary algorithms. In *International Conference on Parallel Problem Solving from Nature*, pages 462–471. Springer, 2002.

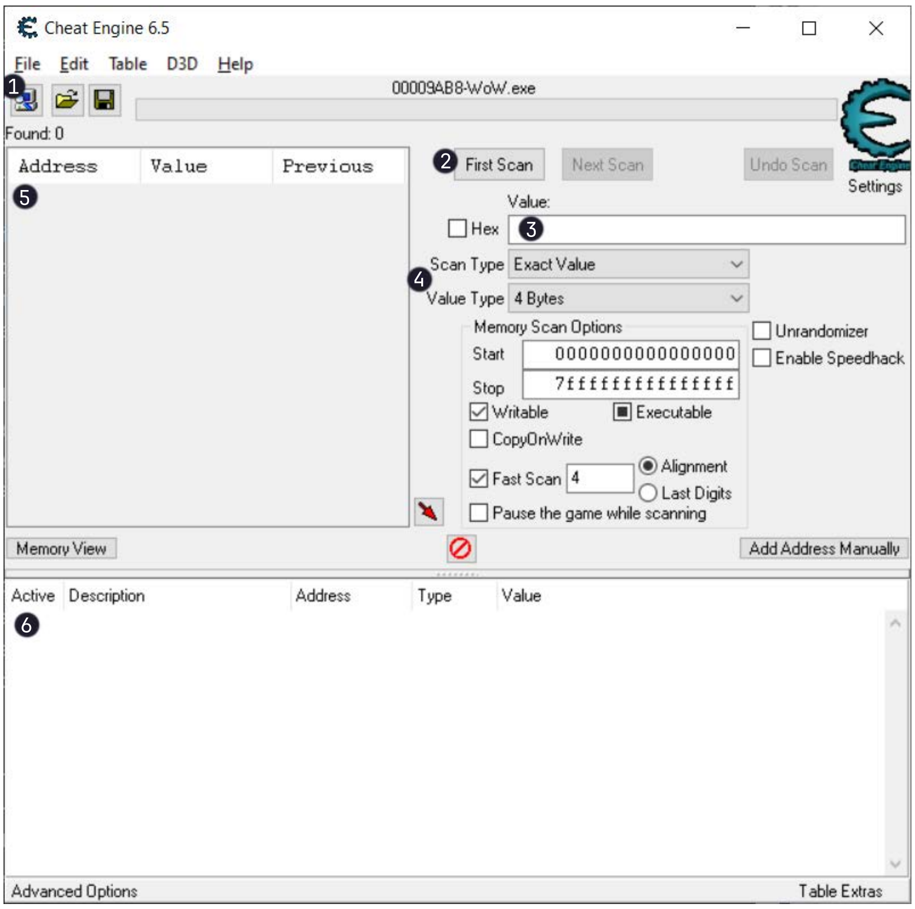
Rui Wang et al. Paired open-ended trailblazer. In *Endlessly generating increasingly complex and diverse learning environments and their solutions*, 2019.

Georgios N Yannakakis and Antonios Liapis. Searching for surprise. In *ICCC*, 2016.

Kun Yi Yizheng Liao and Zhe Yang. Technical report. In *Cs229 final report reinforcement learning to play mario*. Stanford University, 2012.

Appendix A
Details of results

A.1 Additional Details of Memory Scanning Using Cheat Engine



00009AB8-WoW.exe

Found: 0

Value:

Hex ☐

Scan Type: Exact Value

Value Type: 4 Bytes

Start: 0000000000000000

Stop: 7fffffffffffffff

☑ Writable ▣ Executable

☐ CopyOnWrite

☑ Fast Scan 4 ● Alignment ○ Last Digits

☐ Pause the game while scanning

☐ Unrandomizer

☐ Enable Speedhack

Figure 35: Cheat Engine main screen with "**WoW**.exe" attached.

To begin scanning a game's memory, one must select the *"Attach"* icon ❶ to attach to a process and then enter the scan value we wish to locate ❸.

Cheat engine allows for the selection of two different scan directives called *"Scan Type"* and *"Value Type"* ❹ but for this particular use case, it can be left as is.

To initiate the initial scan for values, one must click on the *“First Scan”* ❷ button. The scanner then proceeds to conduct the scan and populates the results list ❺. The results list contains addresses, indicated in either green or black font, which respectively correspond to static or dynamically allocated memory. While static addresses remain consistent across program restarts, dynamically allocated memory is allocated at runtime and is thus variable. Upon initial population of the results list, the real-time value of each address is displayed. During subsequent rescans, the value of each address during the previous scan is also displayed, with real-time values updated based on a predetermined interval set.

After the initial scan, the *“Next Scan”* ❷ button becomes available in the scanner interface, offering six additional scan types for more targeted searches. These scan types allow the user to compare the addresses in the results list to their values in the previous scan, providing a more refined search for the specific game state value being sought. This process of iterative scanning and comparison helps to narrow down the possible addresses that hold the desired value, reducing the number of false positives and increasing the accuracy of the search. Ultimately, this approach allows for the location of the exact memory address that holds the value of interest.

In order to effectively narrow down a large result list and locate the correct address, it is typically necessary to employ multiple scan types. The process of eliminating false positives often involves creating entropy within the game, adjusting scan directives strategically, and courageously utilizing the *“Next Scan”* feature. This process may need to be repeated multiple times until only a single remaining address is identified.

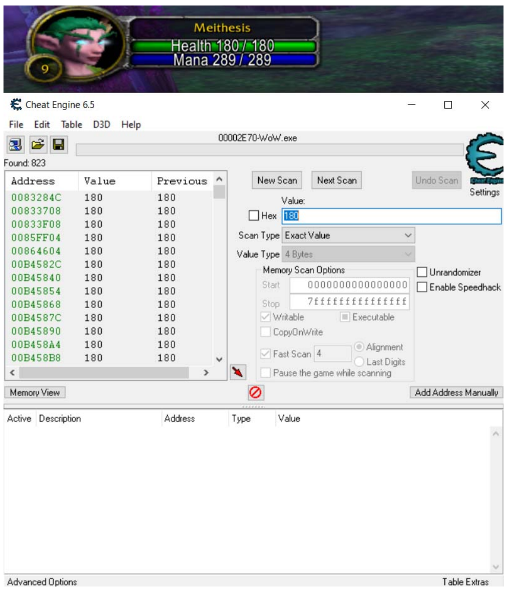
Once the correct memory address is found, it is possible to double-click it to add it to the *cheat table pane* ❻. Addresses in the *cheat table pane* can be modified, watched and saved to the cheat table files for future use.

A.2 Obtaining Player Health Offsets Using Cheat Engine

This section provides a detailed demonstration of how to obtain a specific memory offset from **WoW**. The target value in this case is the player's health.

To begin, the first step is to identify the player's current health points, which in this case is currently 180. Enter this value in the scanner's "*Value:*" field and initiate the scanning process to search for addresses that hold this value.

After the initial scan, there will likely be multiple addresses that contain the desired value. To narrow down the results, further filtering is necessary. One way to achieve this is by modifying the player's health points, for example by engaging in combat with a nearby **NPC**. Take note of the new health value and enter it in the "*Value:*" field for a subsequent scan. By repeating this process, it is possible to identify the address that corresponds to the desired health value. The number of iterations required may vary depending on the specific target value.

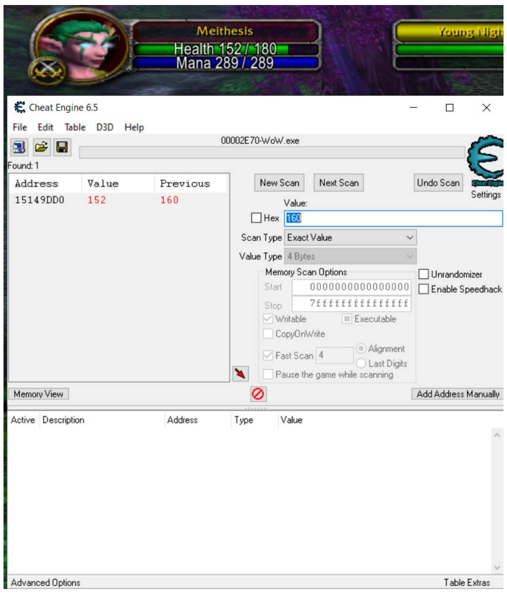


Found: 823

Address	Value	Previous
0083284C	180	180
00833708	180	180
00833F08	180	180
0085FF04	180	180
00864604	180	180
00B4582C	180	180
00B45840	180	180
00B45854	180	180
00B45868	180	180
00B4587C	180	180
00B45890	180	180
00B458A4	180	180
00B458B8	180	180

Value: 180
Hex: ☐
Scan Type: Exact Value
Value Type: 4 Bytes
Memory Scan Options
Start: 0000000000000000
Stop: 7fffffffffffffff
☑ Writable ▨ Executable ☐ CopyOnWrite
☑ Fast Scan 4 ● Alignment ○ Last Digits
☐ Pause the game while scanning
☐ Unrandomizer
☐ Enable Speedhack

Figure 36: Scanning the process memory for the value "180".



Found: 1

Address	Value	Previous
15149DD0	152	160

Value: 160
Hex: ☐
Scan Type: Exact Value
Value Type: 4 Bytes
Memory Scan Options
Start: 0000000000000000
Stop: 7fffffffffffffff
☑ Writable ▨ Executable ☐ CopyOnWrite
☑ Fast Scan 4 ● Alignment ○ Last Digits
☐ Pause the game while scanning
☐ Unrandomizer
☐ Enable Speedhack

Figure 37: Scanning the process memory for the value "160", now "152".

While the obtained address is correct, it is important to note that it is a dynamic address that will become invalid once the game is restarted. To obtain a static pointer chain of offsets, follow the subsequent steps. Start by double-clicking the address to select it, then right-click on the address and choose the option "*Pointer scan for this address*".

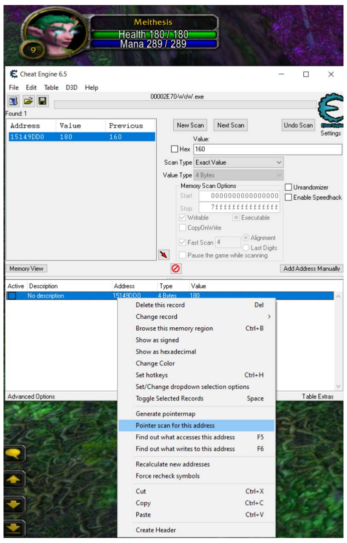
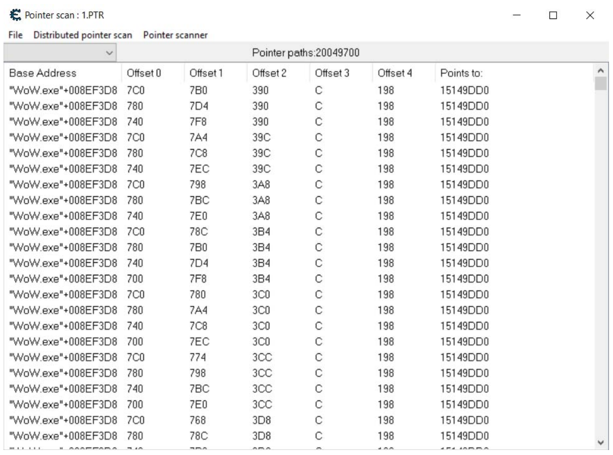


Figure 38: Starting a pointer scan for the selected address.

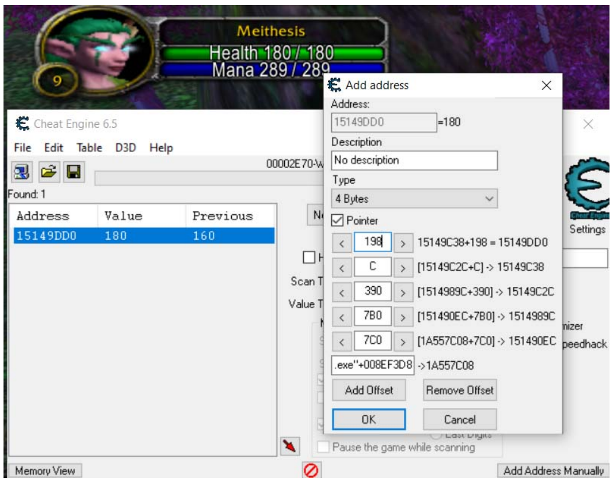
After initiating the pointer scan, a window will display numerous results comprising possible pointer chains that point to the desired final address. However, these results need to be further filtered. To accomplish this, leave the window open and restart the game (**WoW**). Repeat the entire process, but this time use the option to match against the previously obtained and saved pointer chain values. By repeating this process several times, one can be confident in obtaining a static pointer chain that can be utilized in any instance of the game to access the desired value.



Pointer paths:20049700

Base Address	Offset 0	Offset 1	Offset 2	Offset 3	Offset 4	Points to:
"WoW.exe"+008EF3D8	7C0	7B0	390	C	198	15149DD0
"WoW.exe"+008EF3D8	780	7D4	390	C	198	15149DD0
"WoW.exe"+008EF3D8	740	7F8	390	C	198	15149DD0
"WoW.exe"+008EF3D8	7C0	7A4	39C	C	198	15149DD0
"WoW.exe"+008EF3D8	780	7C8	39C	C	198	15149DD0
"WoW.exe"+008EF3D8	740	7EC	39C	C	198	15149DD0
"WoW.exe"+008EF3D8	7C0	798	3A8	C	198	15149DD0
"WoW.exe"+008EF3D8	780	7BC	3A8	C	198	15149DD0
"WoW.exe"+008EF3D8	740	7E0	3A8	C	198	15149DD0
"WoW.exe"+008EF3D8	7C0	78C	3B4	C	198	15149DD0
"WoW.exe"+008EF3D8	780	7B0	3B4	C	198	15149DD0
"WoW.exe"+008EF3D8	740	7D4	3B4	C	198	15149DD0
"WoW.exe"+008EF3D8	700	7F8	3B4	C	198	15149DD0
"WoW.exe"+008EF3D8	7C0	780	3C0	C	198	15149DD0
"WoW.exe"+008EF3D8	780	7A4	3C0	C	198	15149DD0
"WoW.exe"+008EF3D8	740	7C8	3C0	C	198	15149DD0
"WoW.exe"+008EF3D8	700	7EC	3C0	C	198	15149DD0
"WoW.exe"+008EF3D8	7C0	774	3CC	C	198	15149DD0
"WoW.exe"+008EF3D8	780	798	3CC	C	198	15149DD0
"WoW.exe"+008EF3D8	740	7BC	3CC	C	198	15149DD0
"WoW.exe"+008EF3D8	700	7E0	3CC	C	198	15149DD0
"WoW.exe"+008EF3D8	7C0	768	3D8	C	198	15149DD0
"WoW.exe"+008EF3D8	780	78C	3D8	C	198	15149DD0

Figure 39: Results of the first pointer scan for the selected address.



Found: 1

Address	Value	Previous
15149DD0	180	160

Add address

Address: 15149DD0 =180

Description: No description

Type: 4 Bytes

☑ Pointer

```
198   15149C38+198 = 15149DD0
C     [15149C2C+C] -> 15149C38
390   [1514989C+390] -> 15149C2C
7B0   [151490EC+7B0] -> 1514989C
7C0   [1A557C08+7C0] -> 151490EC
.exe"+008EF3D8 ->1A557C08
```

Pause the game while scanning

Figure 40: Manually checking a pointer chain for confirmation.

By following these steps, it is possible to obtain a static pointer chain that allows for consistent access to the desired value in any instance of the game. This information can be invaluable for further analysis, experimentation, or modifications within the game environment.

A.3 How to Define a Domain in SharpNEAT

This section explains the process of defining a new domain within the SharpNEAT framework.

To begin, navigate to the SharpNeat.Tasks project within the framework. Create a new folder with a name that corresponds to the desired domain. For the purpose of this example, we will use [domain_name] as a placeholder.

Once the folder is created, populate it with three essential files: [domain_name]EvaluationScheme.cs, [domain_name]Evaluator.cs, and [domain_name]ExperimentFactory.cs. These files are necessary components for defining and implementing the new domain within the framework.

```
src
└─SharpNeat.Tasks
    └─[domain_name]
        ├─[domain_name]EvaluationScheme.cs
        ├─[domain_name]Evaluator.cs
        └─[domain_name]ExperimentFactory.cs
```

File [domain_name]EvaluationScheme.cs

In the [domain_name]EvaluationScheme.cs file, you will define the evaluation scheme for the new domain. The following steps outline the process:

1. Declare that the file extends the *IBlackBoxEvaluationScheme<double>* interface.

2. Define the number of input nodes using the *InputCount* property, and specify the number of output nodes using the *OutputCount* variable.

3. Specify whether the problem is deterministic or not by setting the value of the *IsDeterministic* boolean variable.

4. Use the *FitnessComparer* variable to track fitness values and determine their comparison order. The *NullFitness* variable can be used to represent a null fitness value if needed.

5. Decide whether evaluators should be instantiated every time an evaluation is performed or if a single static instance should be used by setting the *EvaluatorsHaveState* variable accordingly.

6. Implement the *CreateEvaluator()* function, which should return a new instance of the class defined in [domain_name]Evaluator.cs. This evaluator will be responsible for evaluating the individuals in the domain.

7. Define the *TestForStopCondition(FitnessInfo fitnessInfo)* function to check if the target objective proposed by the algorithm has been achieved. This function can determine when to stop the evolutionary process based on the provided fitness information.

By completing these steps and implementing the necessary functions, you can define the evaluation scheme for the new domain within the Sharp**NEAT** framework.

File [domain_name]Evaluator.cs

In the [domain_name]Evaluator.cs file, you will implement the evaluator for the new domain. Follow these steps to define the evaluator:

1. Declare that the file extends the *IPhenomeEvaluator<IBlackBox<double»* interface.

2. Define the number of trials per evaluation using the *_trialsPerEvaluation* variable. This determines how many times each individual will be evaluated.

3. Implement the *Evaluate(IBlackBox<double> box)* function. This function is crucial as it defines the inputs to be fed into the neural networks (phenotypes) and the expected outputs. It is responsible for activating the box (calling *box.Activate()*) and observing the outputs produced. Within this function, you should also define the fitness function(s) that evaluate how well the **Neural Network** (**NN**) performed on the given task after each trial. You can assess the outputs and calculate fitness based on the desired behavior or objective.

By completing these steps and implementing the necessary functions, you can define the evaluator for the new domain in Sharp**NEAT**. The evaluator will determine how the **NN**s (phenotypes) perform on the given task and assign fitness values accordingly.

File [domain_name]ExperimentFactory.cs

In the [domain_name]ExperimentFactory.cs file, you will define the experiment factory for the new domain. Follow these steps to set up the experiment factory:

1. Declare that the file extends the *INeatExperimentFactory* interface.

2. Define a string variable *Id* which should match the name of your domain. This serves as a unique identifier for the experiment.

3. Implement the function *CreateExperiment(Stream jsonConfigStream)*. This function is responsible for creating the experiment with the desired configuration parameters. You can retrieve the default configuration parameters from the provided *jsonConfigStream* stream.

4. Instantiate a new instance of the evaluation scheme object specific to your domain.

5. Define a *NeatExperiment* object, passing in the evaluation scheme and any additional default settings you want to configure. For example, you can specify whether the experiment is acyclic or cyclic and set the desired activation function.

By completing these steps, you can create the experiment factory for your domain in Sharp**NEAT**. The experiment factory will handle the setup and configuration of the experiment, including the evaluation scheme and other experiment-specific settings.

A.4 Default Configuration Files in SharpNEAT

Once you have defined the necessary files for your domain, the next step is to create the domain's default configuration files for the experiment. These configuration files will be loaded when selecting the desired experiment through the user-friendly **GUI**.

To create the default configuration files, navigate to the SharpNeat.Windows.App project and locate the config directory. In this directory, you can define the configuration files specific to your domain. These files contain the default settings and parameters for the experiment, such as population size, mutation rates, and other relevant parameters.

Having default configuration files allows users to quickly load and run experiments without manually specifying all the settings each time. The **GUI** interface will provide an option to select the desired experiment, and the corresponding default configuration file will be loaded automatically.

By organizing the default configuration files within the config directory of the SharpNeat.Windows.App project, you ensure that they are easily accessible and can be seamlessly integrated into the application's workflow.

```
src
└─SharpNeat.Windows.App
  └─config
    ├─experiments-config
    │ └─ [domain_name].config.json
    ├─experiments-descriptions
    │ └─ [domain_name].txt
    └─experiments.json
```

[domain_name].config.json

The [domain_name].config.json file is a configuration file that contains important parameters and settings for the algorithm used in the domain. Here are the key elements of this configuration file:

- name: Specifies the name of the model or experiment.
- isAcyclic: Indicates whether the model is acyclic or not.
- activationFnName: Defines the activation function used in the **NN**.

The configuration of the **EA** is specified under the "evolutionAlgorithm" section. It includes the following parameters:

- speciesCount: Specifies the number of species in the population.
- elitismProportion: Defines the proportion of top-performing individuals that are preserved in each generation.
- selectionProportion: Specifies the proportion of individuals selected for reproduction.
- offspringAsexualProportion: Defines the proportion of offspring generated through asexual reproduction.
- offspringSexualProportion: Specifies the proportion of offspring generated through sexual reproduction.
- interspeciesMatingProportion: Defines the proportion of offspring generated through mating between different species.

The "reproductionAsexual" section contains parameters related to asexual reproduction:

- connectionWeightMutationProbability: Defines the probability of mutating a connection weight.
- addNodeMutationProbability: Specifies the probability of adding a new node to the **NN**.
- addConnectionMutationProbability: Defines the probability of adding a new connection between neurons.
- deleteConnectionMutationProbability: Specifies the probability of deleting a connection between neurons.

The "reproductionSexual" section defines parameters related to sexual reproduction:

- secondaryParentGeneProbability: Specifies the probability of inheriting genes from a secondary parent during sexual reproduction.
- populationSize: Specifies the size of the model's population.
- initialInterconnectionsProportion: Defines the initial proportion of interconnections between neurons in the **NN**.
- connectionWeightScale: Specifies the scale of the connection weights in the **NN**.

- complexityRegulationStrategy: Specifies the strategy used to regulate the complexity of the model, including the strategy name, relative complexity ceiling, and minimum simplification generations.
- degreeOfParallelism: Defines the degree of parallelism used in the model.
- enableHardwareAcceleratedNeuralNets: Indicates whether hardware acceleration is enabled for the **NN**s.
- enableHardwareAcceleratedActivationFunctions: Indicates whether hardware acceleration is enabled for the activation functions.

By modifying the values in the [domain_name].config.json file, you can customize the behavior and performance of the **EA** according to the specific requirements of your domain.

[domain_name].txt

The [domain_name].txt file is a text file that provides a detailed description of the domain. It serves as a contextual guide for individuals who wish to try the experiment and gain a better understanding of the task being performed by the algorithm. Typically, the contents of the [domain_name].txt file include:

- Domain Name: The file begins by stating the name of the domain.
- Inputs: It describes the number and nature of inputs required by the domain. This helps users understand the type of information or data the algorithm needs as input.
- Outputs: It specifies the possible outputs that can be generated by the domain. This information gives users an idea of the expected results or actions produced by the algorithm.
- Task Description: A brief but informative description of the task being trialed by the algorithm is provided. This description outlines the objective or problem that the algorithm aims to solve within the specific domain. By providing this textual information, the [domain_name].txt file helps users familiarize themselves with the domain and gain insights into the purpose and goals of the experiment.

experiments.json

The experiments.json file plays a crucial role in integrating and configuring the domain within the Sharp**NEAT** framework. This file follows a standard **JSON** structure and is responsible for specifying the necessary files related to the domain.

The contents of the experiments.json file include:

- Domain Name: The file begins by defining the name of the domain.
- Experiment Factory: It specifies the experiment factory file ([domain_name]ExperimentFactory.cs) that was previously created. This file is responsible for creating the experiment instance.
- Configuration File: It identifies the configuration file ([domain_name].config.json) associated with the domain. This file contains the parameters and settings for the algorithm.
- Description File: It denotes the description file ([domain_name].txt) that provides a detailed explanation of the domain and the task being performed by the algorithm.
- Experiment UI Factory: This section defines the experiment UI factory. The framework provides default values, but customization is possible if required for the specific domain.

Once all these files are created and populated, selecting SharpNeat.Windows.App as the main startup project and launching the application will allow you to see your defined domain as one of the available options. This integration ensures that your domain is recognized and accessible within the Sharp**NEAT** application, providing a seamless user experience for running experiments and analyzing results.