A purple text on a white background

Description automatically generated

School of Computer Science & Electronic Engineering,

Title:

**A Neural Network Agent for TORCS**

A thesis submitted for the degree of:

# Master of Science in Computer Games

Author:

**Sayed Maqbool Ahmed Inamdar**

(**Registration: 2204389**)

Supervisors:

# Dr Michael Fairbank

Finished on August, 2023.

**Abstract**

This project focuses on developing a neural network agent to play the game Torcs. Using advanced techniques, the agent is trained to understand and interact with the game environment efficiently. Simply put, I have created a digital brain that learns to navigate and perform within the Torcs game. Over time, through continuous learning and adjustments, our agent gets better at the game, making decisions that maximize its success. This project showcases the potential of neural networks in mastering complex tasks and offers insights into game AI development.

Table of Contents

[Master of Science in Computer Games 1](#_Toc144161702)

[Dr Michael Fairbank 1](#_Toc144161703)

[1. Introduction 6](#_Toc144161704)

[1.1 Background of TORCS Game 6](#_Toc144161705)

[1.2 Objective of the Project 6](#_Toc144161706)

[2. Reasons for Platform and Tool Choices 6](#_Toc144161707)

[2.1 Why Ubuntu? Advantages in Gaming and Development 6](#_Toc144161708)

[2.2 The Use of Python in Game Development 7](#_Toc144161709)

[2.3 Gym: A Framework for Reinforcement Learning 7](#_Toc144161710)

[2.4 TensorFlow: Leading Deep Learning Library 7](#_Toc144161711)

[3. Literature Review 7](#_Toc144161712)

[3.1 Evolution of Game Agents 8](#_Toc144161713)

[3.2 Neural Networks in Modern Gaming 8](#_Toc144161714)

[3.3 Role of Genetic Algorithms in Game Training 9](#_Toc144161715)

[4. Understanding the Game Dynamics 9](#_Toc144161716)

[4. Methods 11](#_Toc144161717)

[4.1 Environment Setup 11](#_Toc144161718)

[4.2 Neural Network Agent Architecture 12](#_Toc144161719)

[4.2.1 Input Layers: 13](#_Toc144161720)

[4.2.2 Hidden Layer: 13](#_Toc144161721)

[4.2.3 Output Layer: 15](#_Toc144161722)

[4.3 Genetic Algorithm and Training Dynamics 15](#_Toc144161723)

[4.4 Fitness Evaluation of the Neural Network Agent 15](#_Toc144161724)

[4.5 Handling Camera Angles and Visualization 16](#_Toc144161725)

[5. Modifications and Variations 17](#_Toc144161726)

[5.1 Simplifying Observation States 17](#_Toc144161727)

[5.1.1 Focusing on Track and Angle 17](#_Toc144161728)

[5.1.2 Results and Implications of Simplification 17](#_Toc144161729)

[5.2 Other Experimental Variations 17](#_Toc144161730)

[5.2.1 Adjusting Activation Functions 17](#_Toc144161731)

[5.2.2 Experimenting with Different Neural Network Depths 18](#_Toc144161732)

[6. Design Decisions and Considerations 18](#_Toc144161733)

[6.1 Initialization Techniques 18](#_Toc144161734)

[6.2 Avoiding Overfitting and Ensuring Generalization 19](#_Toc144161735)

[6.3 Genetic Algorithm Hyperparameters 19](#_Toc144161736)

[6.4 Deciding the Neural Network Architecture 19](#_Toc144161737)

[7. Results and Analysis 20](#_Toc144161738)

[7.1 Training Results over Generations 20](#_Toc144161739)

[7.2 Performance Metrics and Evaluation 22](#_Toc144161740)

[7.3 Efficiency vs. Accuracy Trade-offs 23](#_Toc144161741)

[8. Impact of Variations 23](#_Toc144161742)

[8.1 Observations on Model Complexity vs. Performance 23](#_Toc144161743)

[8.2 Insights on Essential Game Observations for Model Efficiency 23](#_Toc144161744)

[8.3 Comparison of Various Models' Robustness 23](#_Toc144161745)

[9. Comparison with Other Agents 23](#_Toc144161746)

[9.1 Human-driven Agent vs. Neural Network Agent 24](#_Toc144161747)

[9.2 Traditional Hand-coded Agent 24](#_Toc144161748)

[10. Understanding the Hardcoded Agent: 25](#_Toc144161749)

[10.1 Interpreting the Track Layout: 25](#_Toc144161750)

[10.2 Detailed Comparative Analysis and Insights 26](#_Toc144161751)

[11. Challenges and Lessons Learned 27](#_Toc144161752)

[11.1 Debugging and Development Issues 27](#_Toc144161753)

[11.2 Lessons from Failed Generations 27](#_Toc144161754)

[11.3 Refinements and Iterative Development 28](#_Toc144161755)

[12. Discussion 28](#_Toc144161756)

[12.1 Achievements and Strengths 28](#_Toc144161757)

[12.2 Potential Pitfalls and Limitations 28](#_Toc144161758)

[12.3 The Future of Neural Network Agents in Gaming 29](#_Toc144161759)

[13. Further Experiments and Considerations 29](#_Toc144161760)

[13.1 The Role of Hyperparameters 29](#_Toc144161761)

[13.1.1 Impact of Learning Rate 29](#_Toc144161762)

[13.1.2 Importance of Batch Sizes 29](#_Toc144161763)

[13.2 Potential for Incorporating Other ML Techniques 29](#_Toc144161764)

[13.2.1 Potential for Transfer Learning 29](#_Toc144161765)

[13.2.2 Role of Reinforcement Learning 30](#_Toc144161766)

[14. Recommendations 30](#_Toc144161767)

[14.1 For Aspiring Game Developers 30](#_Toc144161768)

[14.2 For Improving the Model 30](#_Toc144161769)

[14.3 For Efficient Training 30](#_Toc144161770)

[15. Explaining the Genetic Agent and Training Mechanism for TORCS (Script explanation) 31](#_Toc144161771)

[Imported Libraries: 31](#_Toc144161772)

[The Genetic Agent Architecture and Functionality 31](#_Toc144161773)

[Genetic Operations for Agent Evolution: 32](#_Toc144161774)

[Episode Runner: 33](#_Toc144161775)

[Main Execution Logic: 33](#_Toc144161776)

[16. Project Management and Risk Analysis 33](#_Toc144161777)

[Project Management: 33](#_Toc144161778)

[Risk Analysis: 34](#_Toc144161779)

[15. Conclusion 34](#_Toc144161780)

[15.1 Summary of Findings 34](#_Toc144161781)

[15.2 Implications for the Gaming Industry 35](#_Toc144161782)

[17. Future Work 35](#_Toc144161783)

[17.1 Incorporating Real-time Feedback 35](#_Toc144161784)

[17.2 Broadening to Other Racing Simulations 36](#_Toc144161785)

[17.3 Exploring Multi-agent Dynamics. 36](#_Toc144161786)

[18. Acknowledgments 36](#_Toc144161787)

[References 37](#_Toc144161788)

# 1. Introduction

## 1.1 Background of TORCS Game

The Open Racing Car Simulator, affectionately known as TORCS, has emerged as more than just a racing simulation game. Its evolution is rooted in its open-source framework, which has facilitated a significant amount of AI-driven exploration and innovation over the years. Originally envisioned as a conduit for racing enthusiasts to experience virtual races, TORCS' adaptability has seen it transformed into an influential platform for cutting-edge AI research. The depth of the simulator is evident from its intricate physics engine, its diverse array of vehicles and circuits, and most notably, the capability it offers users to craft and refine their AI drivers [1]. Such flexibility has positioned TORCS as an invaluable resource for those at the crossroads of gaming and artificial intelligence research.

## 1.2 Objective of the Project

Embarking on this project wasn't merely about coding an agent that could proficiently navigate TORCS' terrains. The underlying vision was grander. My primary ambition was to sculpt a neural network agent, a fusion of deep learning and game dynamics, that could traverse the winding tracks of TORCS with the finesse and strategy of a seasoned racer. Furthermore, this agent was not just competing against the game's default AI; it was contending against the cognitive abilities of human players and the deterministic logic of traditionally coded agents. The journey into this project was as much an exploration into the expansive capabilities of neural networks and genetic algorithms as it was a deep dive into the future of gaming where human intuition meets algorithmic precision [2]. The results, challenges, and insights derived from this venture underscore the transformative potential of AI in reshaping the gaming ecosystem.

# 2. Reasons for Platform and Tool Choices

Choosing the right tools and platforms for a project of this magnitude is no trivial task. In a landscape teeming with a multitude of options, each offering unique features, it's essential to make choices that align with the project's goals and provide a smooth and efficient development experience.

## 2.1 Why Ubuntu? Advantages in Gaming and Development

Ubuntu, a descendant of the Debian Linux distribution, has steadily cemented its position in the world of software development and computing. While traditionally Linux distributions weren't the first choice for gaming, the narrative has been shifting, with Ubuntu at the forefront of this change. Several reasons account for Ubuntu's rise in this domain:

• Stability and Performance: One of the hallmarks of Ubuntu is its stability. Developers often choose it to avoid unpredictable OS-related glitches, ensuring a smoother gaming experience.

• Open-Source Nature: Ubuntu's open-source foundation means developers can customize their environment to their needs, making it an appealing choice for bespoke game development projects.

• Extensive Libraries & Tools: Ubuntu boasts a vast array of libraries and tools essential for game development, readily available through its package manager.

• Growing Gaming Community: With platforms like Steam extending their support to Ubuntu, the gaming community has seen a surge in both developers and gamers transitioning to this OS.

## 2.2 The Use of Python in Game Development

Python, a high-level and interpreted language, may not be the immediate pick when one thinks of game development, traditionally dominated by languages like C++. However, Python has been breaking these conventions, finding its niche in the gaming world for various reasons:

Python allows for swift development and prototyping, enabling developers to quickly iterate and test their ideas. With frameworks like Pygame, developers have tools at their disposal to create interactive games without delving into the intricacies of low-level graphics programming. While Python may serve as the primary scripting language, it can also be integrated with engines and frameworks that use other languages, offering a blend of ease and performance. Community Support: The Python community is vast and active, ensuring that developers have a wealth of resources, tutorials, and forums to assist in their game development journey.

## 2.3 Gym: A Framework for Reinforcement Learning

OpenAI's Gym has steadily carved a niche for itself as the go-to platform for developing and comparing reinforcement learning algorithms. With its suite of predefined environments, Gym offers a standardized way to benchmark an agent's performance [3]. For this project, Gym's modular nature allows for seamless integration with other tools and libraries. This flexibility meant that the project could benefit from other powerful platforms like TensorFlow without encountering compatibility issues. Gym's extensive collection of environments, including custom environments like TORCS, provides an ample testing ground to train and evaluate agents. Being an open-source platform, Gym enjoys vast community support. This not only ensures that any issues encountered can be rapidly addressed but also fosters a collaborative spirit, enabling the sharing of innovative solutions and ideas.

## 2.4 TensorFlow: Leading Deep Learning Library

TensorFlow, an open-source deep learning framework developed by the Google Brain team, has transformed the realm of neural network research and development.

TensorFlow's architecture is inherently scalable. It can run on single CPUs, GPUs, and even on mobile devices, but it can also scale to run on multi-GPU setups and large-scale cloud clusters. efficient execution of complex tensor computations ensures that models train faster, and in-game agents respond in real-time, crucial for a game like TORCS. With its high-level APIs, TensorFlow makes it relatively straightforward to design, train, and deploy deep learning models. This means that experimenting with various architectures becomes less of a chore and more of an intuitive process. TensorFlow isn't just a library. Its ecosystem includes tools like TensorBoard for visualization and TensorFlow Lite for mobile deployments, providing an end-to-end solution for deep learning tasks.

# 3. Literature Review

The fusion of artificial intelligence and gaming isn't new. Over the years, as technology progressed, games transitioned from basic rule-based algorithms to complex AI-driven mechanisms. This evolution has been chronicled in various literature and research papers. In recent times, with the advent and growth of neural networks, a notable shift has been observed in how games are developed, played, and perceived.

## 3.1 Evolution of Game Agents

The domain of game agents has witnessed a remarkable journey, reflecting the broader trajectories of computer science and artificial intelligence. From rudimentary rule-based systems to the intricacies of deep reinforcement learning, the progression of game agents offers a rich tapestry of innovation and challenges.

*Early Beginnings: Rule-Based Systems*

The infancy of game agents saw them relying heavily on hard-coded, rule-based systems. This agent was fundamentally deterministic, with actions taken based purely on pre-defined sets of conditions and rules. Although limited in adaptability, these rule-based agents paved the way for more complex systems and were foundational in popular early video games [4].

*Deep Reinforcement Learning: The Current Frontier*

The convergence of deep learning and reinforcement learning has produced what is arguably the most potent class of game agents to date. Deep reinforcement learning leverages the representational power of neural networks to enable agents to master incredibly complex environments. Noteworthy milestones include DeepMind's AlphaGo, which triumphed over human champions in the game of Go [6], and OpenAI's agents trained to excel in multiple game scenarios, showcasing the versatility of such systems.

## 3.2 Neural Networks in Modern Gaming

Neural networks, a subset of machine learning, have been the pivot around which modern gaming has begun to revolve. These networks, designed to replicate human brain functions in a simplified manner, can learn, reason, and self-correct [7].

• Procedural Content Generation: Early literature, such as the work by Summerville et al., highlighted how neural networks can be harnessed for procedural content generation. Instead of game developers designing every aspect of a game, neural networks can be trained to generate content, such as levels, characters, and even storylines, based on given parameters.

• Advanced NPC Behavior: Gone are the days when non-player characters (NPCs) in games followed a predetermined path or script. With neural networks, NPCs can learn from the player's behavior, adapt, and respond in unpredictable ways, making the gaming experience more immersive and challenging [8].

• Realistic Simulations: Games like TORCS have showcased how neural networks can be instrumental in creating ultra-realistic simulations. Racing AI doesn't just follow a set track; it learns, adapts, and races as a human would, considering factors like speed, trajectory, and competition [1].

• Enhanced Player Experience: Neural networks have also been employed to enhance the overall player experience. From adjusting game difficulty based on a player's skill level to providing personalized content recommendations, neural networks are reshaping how players interact with games [9].

While neural networks have undoubtedly revolutionized the gaming industry, challenges remain. Training a neural network for gaming requires vast amounts of data, computational resources, and time. However, as technology advances and as neural network architectures become more sophisticated, their integration into gaming is poised to reach even greater heights.

## 3.3 Role of Genetic Algorithms in Game Training

Genetic algorithms (GAs) have been a topic of interest since they were first introduced, inspired by the process of natural selection [10]. In essence, these algorithms simulate the process of evolution to arrive at an optimal or near-optimal solution to a given problem. Genetic algorithms rely on principles like mutation, crossover (recombination), and selection to evolve a population of solutions over multiple generations.

In the context of game training, GAs have gained traction for several profound reasons:

• Exploration of Large Solution Spaces: Many games, especially those of strategy and simulation, have vast solution spaces. Traditional algorithms can often get trapped in local optima, but GAs, with their randomized processes of mutation and crossover, have a higher chance of escaping these pitfalls and exploring more of the solution space [11].

• Adaptability: Genetic algorithms do not require any prior knowledge of the problem landscape. This makes them particularly attractive for games, where environments can be dynamic and unpredictable.

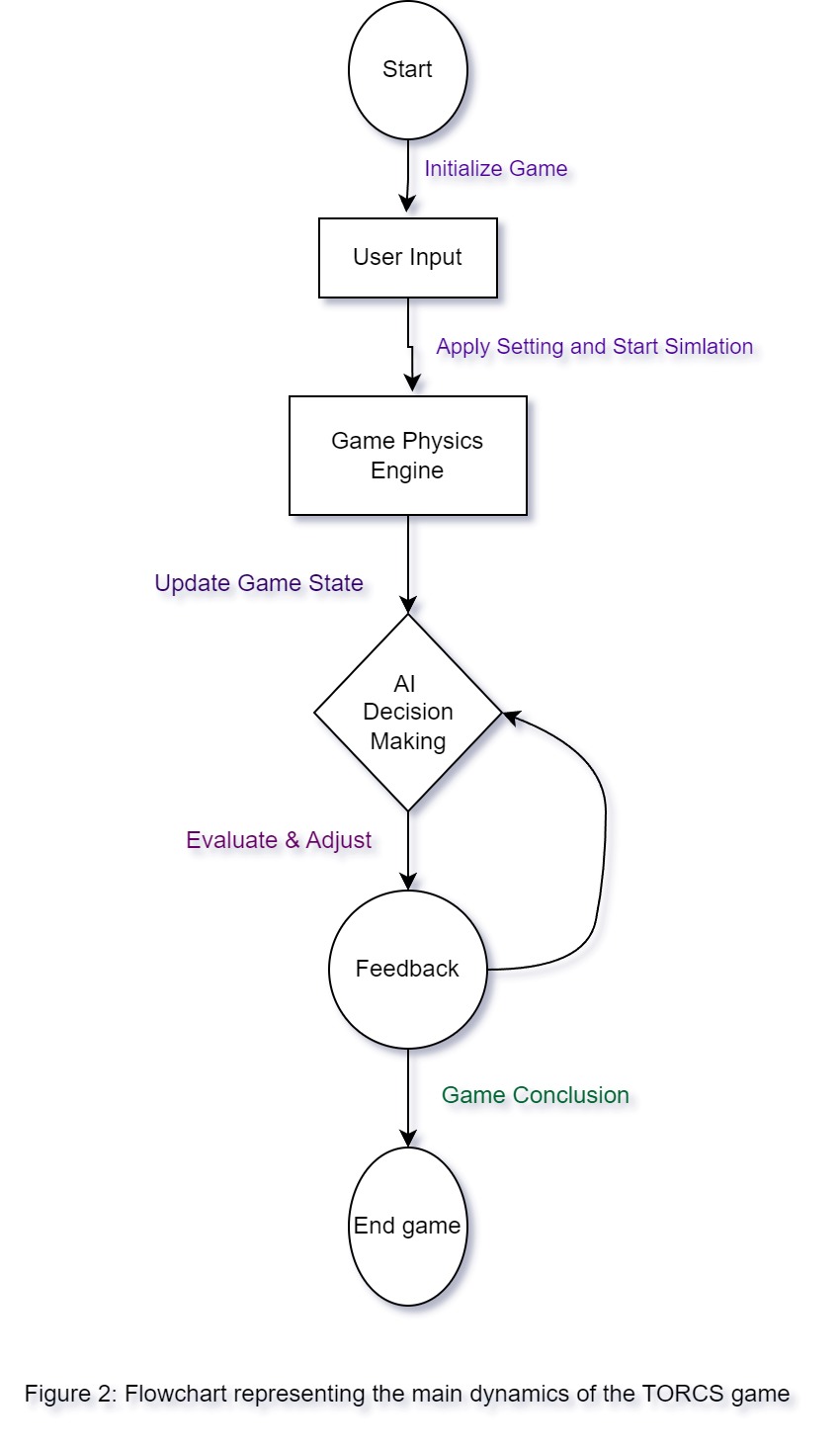
• Parallelism: GAs inherently support parallelism. Multiple solutions (or individuals) in the population can be evaluated simultaneously, leading to faster convergence to an optimal solution. This is invaluable in game scenarios where real-time decision-making is crucial.

Several studies have demonstrated the prowess of genetic algorithms in game training. Lucas and Kendall used GAs to evolve strategies for the game of Othello and found them to outperform other methods like simulated annealing [12]. Similarly, Hauptman and Sipper utilized genetic algorithms to evolve decision-making trees for the game of chess, showing promising results against traditional chess engines [13].

The versatility and efficacy of genetic algorithms have made them an integral tool in the toolkit of game AI developers. As computational capabilities continue to expand and games become more complex, the synergy between GAs and game training is poised to deepen.

# 4. Understanding the Game Dynamics

TORCS, as a racing simulator, offers a complex environment that integrates user inputs, underlying game physics, AI-driven decisions, and feedback loops to create a realistic racing experience. To comprehend the intricate interplay of these components, we've visualized the main dynamics of the game in Figure 2.



The flowchart in Figure 2 provides a high-level overview of the game's operations:

1. **Initialize Game:** The game starts with the user initializing it, setting up any required parameters or simply beginning the race.
2. **Apply Settings and Start Simulation:** Post user input, the game's physics engine simulates the environment based on the provided settings, ensuring realistic car movements, collisions, and race dynamics.
3. **Update State:** As the race progresses, the game's current state, encompassing factors like the car's position, speed, and other dynamics, is continuously updated. This updated state serves as input for the AI's decision-making process.
4. **Evaluate and Adjust:** The AI evaluates the game's current state and makes decisions accordingly. These decisions, in turn, influence the game's subsequent state, creating a continuous feedback loop. This loop ensures the AI's adaptability and responsiveness to the game's ever-changing dynamics.
5. **Game Conclusion:** The game can conclude based on various scenarios, such as race completion, a car crash, or other termination conditions defined within the game's logic.

The Open Racing Car Simulator (TORCS) is more than just a racing game, it’s an intricate blend of physics, graphics, and artificial intelligence. To successfully navigate this environment and build a competent AI agent, a deep understanding of the game's underlying dynamics is paramount. At the heart of TORCS is a detailed physics engine [1] that governs car behaviors, including tire friction, aerodynamics, and suspension. Understanding this engine is crucial for any agent aiming to master the game, as these elements dictate how vehicles respond to in-game commands. Diverse Racing Tracks: TORCS boasts a variety of tracks, each presenting its unique set of challenges. From the hilly terrains of "Alpine" to the sharp turns of "Corkscrew", the dynamism in track selection tests the adaptability and generalization abilities of AI agents. Car Selection and Tuning: Players have the flexibility to choose from a plethora of cars, each with different attributes. This variety means that the AI agent must be versatile enough to handle different vehicle dynamics.

# 4. Methods

The methodology adopted for this project is an intricate blend of strategic planning, resource allocation, and iterative experimentation. In the rapidly advancing realm of AI-driven gaming, having a clear and systematic approach is indispensable.

## 4.1 Environment Setup

Setting up an environment conducive to training an AI agent for a game like TORCS demands meticulous attention to detail. A well-structured environment not only facilitates smoother training but also ensures that the agent's performance is benchmarked under consistent conditions.

• TORCS Configuration: The version and configuration of TORCS used in this project were aligned to ensure compatibility with the chosen Gym environment [1]. This ensured that the agent was trained under realistic conditions, mirroring real-world race scenarios.

• Gym Integration: OpenAI's Gym provided the foundational structure for the agent's training environment. Custom environments tailored to TORCS were integrated, allowing for a seamless training experience [3].

• Hardware and Software Infrastructure: Training deep neural network agents requires substantial computational resources. For this project, a setup incorporating high-performance GPUs was utilized, ensuring that training times were minimized, and model iterations could be tested rapidly. The software stack, built around TensorFlow, was optimized for the available hardware, ensuring efficient resource utilization [15].

• Data Collection and Pre-processing: Before the training process, data was collected from various TORCS races to understand the typical scenarios the agent might encounter. This data underwent rigorous preprocessing to ensure that the input to the neural network was consistent and free from anomalies.

• Evaluation Metrics and Benchmarks: Establishing clear evaluation metrics is crucial to monitor the progress of an AI agent. For this project, the agent's performance was gauged based on its lap times, its ability to navigate challenging terrains, and its prowess in avoiding collisions.

By crafting a methodical environment, the project ensured that the neural network agent was provided with the best possible platform to learn, adapt, and ultimately master the complexities of TORCS.

## 4.2 Neural Network Agent Architecture

The architecture of the neural network agent designed for this project is a culmination of both empirical experimentation and theoretical considerations.

A diagram of a layer

Description automatically generated

**Forward Propagation of the Neural Network**

The forward propagation of our neural network agent can be mathematically represented in two main steps:

1. **Propagation from Input to Hidden Layer:**

The input is transformed using a weight matrix and a bias vector to produce the weighted sum . This is then passed through a Rectified Linear Unit (ReLU) activation function to get the activation of the hidden layer.

Where

* - Weight matrix for the input to the hidden layer.
* - Bias vector for the hidden layer.
* - Weighted sum for the hidden layer.
* - Activation of the hidden layer after applying the ReLU function.

1. **Propagation from Hidden Layer to Output Layer:**

Where

* - Weight matrix for the hidden to the output layer.
* - Bias vector for the output layer.
* - Weighted sum for the output layer.
* - The final output of the network.

The activation from the hidden layer is transformed using a weight matrix and a bias vector to produce the weighted sum which is taken as the final output of the network.

This forward propagation process allows our neural network agent to take in observations from the TORCS environment and produce corresponding actions based on its current weights and biases.

*The neural network comprises three main layers:*

### 4.2.1 Input Layers:

The initial observation states from the TORCS environment provided a plethora of information, including:

* *angle:* The angle between the car direction and the track axis.
* *track:* An array representing the distance between the car and the track edge at various angles.
* *trackPos:* The distance between the car and the track axis.
* *speedX, speedY, speedZ:* Speed of the car in different directions.
* *wheelSpinVel:* The velocity of the spinning wheel.
* *rpm:* Revolutions per minute of the car's engine.
* *opponents:* An array representing the distance to the nearest opponents at various angles.

From these, the final model focused on just three inputs: angle, track, and trackPos.

Here's why:

**Relevance:** angle, track, and trackPos are directly related to the car's position and orientation on the track, which are crucial for navigation. These parameters provide a comprehensive understanding of the car's immediate environment, aiding in decision-making for steering, acceleration, and braking.

**Simplification:** Neural networks with fewer inputs can be trained faster and are less prone to overfitting. By focusing on the most relevant inputs, the model becomes more efficient without sacrificing performance.

**Empirical Evidence:** During initial experiments, it might have been observed that including all observation states did not significantly improve the agent's performance. Hence, a decision was made to streamline the inputs.

### 4.2.2 Hidden Layer:

Our model boasts a hidden layer teeming with 128 neurons. This choice is underpinned by:

**Complexity Handling:** The multifaceted nature of TORCS demands a hidden layer that can deftly navigate its intricacies. The 128 neurons are adept at discerning subtle patterns and relationships in the input data.

**Balancing Performance with Efficiency:** While a burgeoning number of neurons can theoretically amplify a model's prowess, it also ratchets up computational demands. The 128 neurons strike a judicious balance between stellar performance and computational thriftiness.

**Iterative Refinement:** Our iterative testing regimen revealed that a hidden layer with approximately 128 neurons consistently delivered superior performance compared to its sparser or denser counterparts.

#### 4.2.2.1 Activation Function: The Choice of ReLU

**Why ReLU?**

ReLU is defined as

Several factors make ReLU a preferred choice:

* **Computational Efficiency:** ReLU is simple to compute, requiring just a thresholding at zero. This allows models to train faster and requires less computational resources.
* **Mitigating the Vanishing Gradient Problem**: Deep neural networks often suffer from the vanishing gradient problem, where gradients of the loss function approach zero, making the network hard to train. ReLU alleviates this problem since its gradient is either zero (for negative inputs) or one (for positive inputs).
* **Empirical Success:** ReLU has been found to perform well empirically in various deep learning tasks, especially when used in hidden layers.

**Choosing Sigmoid Over ReLU**

For the sake of experiment, I chose the sigmoid activation function instead of ReLU for our TORCS neural network agent. The sigmoid function is defined as

and it squashes its output between 0 and 1.

**Here's what happened:**

**Slower Training:** The sigmoid function involves more complex computations than ReLU (due to the exponential function). This led to increased training times for my agent.

**Here’s what might happen:**

**Vanishing Gradient Problem:** As the network depth increases, the gradients during backpropagation can become extremely small, essentially causing the network to stop learning. This is especially problematic in deeper architectures.

**Saturation:** Since the output of the sigmoid function is constrained between 0 and 1, neurons can easily saturate, especially with inputs that are very positive or very negative. Once saturated, it becomes challenging for the neuron to change its weights and biases, leading to stagnation in learning.

While various activation functions have their merits and use-cases, the choice should align with the specific requirements of the task and the nature of the data. For our TORCS neural network agent, ReLU's properties made it an optimal choice, ensuring efficient training and robust performance.

### 4.2.3 Output Layer:

The output layer is the linchpin that translates the neural network's computations into tangible game controls:

1. steering: Charts the car's directional course.
2. acceleration: Modulates the car's forward propulsion.
3. brake: Orchestrates the car's deceleration dynamics.

This architecture was chosen after careful consideration and experimentation. The input layers capture essential game dynamics, the hidden layer processes this information, and the output layer translates it into actionable game controls.

## 4.3 Genetic Algorithm and Training Dynamics

Genetic Algorithms (GAs) are bio-inspired optimization techniques grounded in the principles of natural selection and genetics [10]. They've been utilized in various domains, from function optimization to machine learning, and have shown promising results in evolving solutions that might be cumbersome or even impossible to deduce using traditional methods.

The application of GAs to this project emerged from the idea of evolving the neural network weights to optimize the agent's performance in TORCS. This approach possesses several salient features:

• Population-Based: Unlike gradient-descent methods that might get trapped in local minima, GAs work with a population of solutions, thereby increasing the chances of global convergence.

• Crossover and Mutation: The heart of GAs. Crossover (or recombination) and mutation introduce genetic variability, ensuring a diverse gene pool and fostering the evolution of optimal or near optimal solutions [10].

• Fitness Evaluation: Everyone (or agent) in the population is evaluated based on fitness function. In our context, the agent's ability to navigate the TORCS environment effectively determined its fitness.

• Elitism: To ensure the best agents (or the fittest) were not lost during crossover and mutation, an elitism strategy was employed, directly promoting top-performing agents to the next generation.

The dynamics of training using a GA differed from standard neural network training methods. Traditional backpropagation was eschewed in favor of evolving network weights over successive generations. The agent's proficiency in TORCS served as the guiding metric, ensuring that with each passing generation, the agent became more adept at the task at hand.

## 4.4 Fitness Evaluation of the Neural Network Agent

A crucial component of our genetic algorithm-based approach in training the neural network agent for the TORCS environment is the evaluation of each agent's fitness. The fitness of an agent is a measure of its performance and is used to guide the evolutionary process.

**Calculation of Fitness:**

The fitness of an agent is determined by its cumulative reward over the course of an episode in the TORCS environment. At the start of an episode, the cumulative reward is initialized to zero. As the agent navigates the environment, it receives rewards based on its actions. These rewards are aggregated to compute the agent's total fitness for that episode.

For instance, rewards can be based on factors such as the car's speed, its position on the track, its orientation relative to the track's direction, and other dynamics specific to the TORCS environment.

**Integration into the Genetic Algorithm:**

Once the fitness of all agents in a generation is calculated, it serves as a basis for the genetic algorithm operations:

**Selection:** Agents with higher fitness values have a higher probability of being selected for reproduction.

**Crossover:** Pairs of agents are chosen based on their fitness to produce offspring for the next generation.

**Mutation:** To introduce variability, some agents undergo mutations. However, the primary guide for evolution remains the fitness value.

By using the cumulative reward as a fitness measure, I ensure that agents that are better at navigating the TORCS environment and achieving higher scores are more likely to pass their traits to the next generation.

## 4.5 Handling Camera Angles and Visualization

The relationship between a racer and the track is symbiotic, where the racer's performance is significantly influenced by their perception of the track. In a game like TORCS, the digital agent's perception is fundamentally tied to camera angles and the way game scenes are rendered and interpreted by the AI.

• Camera Perspectives: TORCS provides a variety of camera angles, each offering a unique vantage point. Choosing the right perspective is crucial as it directly impacts the quality of data the agent receives. For this project, after experimenting with multiple angles, the 'F2' camera setting was preferred. This choice was rooted in its balanced field of view, capturing track layout, upcoming turns, and potential obstacles, while filtering out extraneous details [1].

• Data Processing and Visualization: The raw visuals obtained from the game, although realistic to the human eye, can be overwhelming for an AI agent. Therefore, these visuals were distilled into meaningful data points such as track curvature, car orientation, and distance to boundaries. This abstraction not only accelerated the agent's learning process but also streamlined the computational load.

• Feedback Loop and Adjustments: To ensure the AI agent was effectively leveraging the visuals, a feedback mechanism was instituted. By periodically analyzing the agent's performance and decisions, tweaks to camera settings and visual processing algorithms were made to optimize data quality.

Visualization plays a dual role. While it serves the AI by providing critical data, it also aids human developers. Tools like Tensor Board were instrumental in understanding and refining the neural network's internal workings. By visualizing neuron activations, weight distributions, and loss gradients, the team could intuitively grasp the agent's learning trajectory and make informed adjustments [15].

# 5. Modifications and Variations

In the quest to design an optimal neural network agent for TORCS, it is often necessary to venture beyond conventional methodologies and experiment with different approaches. One such methodology explored in this project involves simplifying observation states. Simplifying observation states can profoundly impact the behavior of the agent, its learning rate, and its overall performance.

## 5.1 Simplifying Observation States

Observation states, in the context of reinforcement learning, play a pivotal role as they provide the agent with a snapshot of the environment at any given moment. It's this information that enables the agent to make decisions that will optimize its performance over time [5]. For TORCS, a myriad of potential observation states can be harnessed ranging from the position of the car to the specifics of the track. However, an inundation of information isn't always beneficial. Sometimes, less is more.

### 5.1.1 Focusing on Track and Angle

To declutter the information space and potentially streamline the learning process, this project decided to focus predominantly on two key observation states: the track and the angle of the car. The rationale behind this:

• Reduced Complexity: By narrowing down the observation states, the complexity of the input data reduces, which can lead to quicker and more efficient training processes [16].

• Focus on Vital Data: Both the track and the angle are paramount for a racing simulation. The track information guides the agent on potential obstacles or curves ahead, while the angle can indicate the car's alignment concerning the track—critical for determining turns or adjustments.

### 5.1.2 Results and Implications of Simplification

Simplifying the observation state brought forth a mixed bag of results:

• Acceleration in Learning: As hypothesized, the agent exhibited a quicker initial learning rate, potentially due to the reduced input space.

• Compromises in Decision-Making: However, the exclusion of certain observation parameters did sometimes lead to situations where the agent seemed "unaware" or less adaptive to specific changes in the environment.

• Optimization Opportunities: The results suggested that while simplification can be beneficial, a balanced approach where critical observation states are retained might be the key to optimal performance.

## 5.2 Other Experimental Variations

Amid the explorative nature of this project, merely adhering to conventional methodologies wasn't sufficient. To truly discern the boundaries and potentials of our neural network agent, a series of experimental variations were conceived and tested. These experiments provided invaluable insights into the intricate balance between architecture, function, and performance.

### 5.2.1 Adjusting Activation Functions

One fundamental aspect of neural networks lies in their activation functions. They are responsible for introducing non-linear properties into the network, allowing for complex relationships between inputs and outputs to be learned [15]. For this project, various activation functions were trialed, including:

• ReLU (Rectified Linear Unit): Widely adopted for its simplicity and efficiency, this function avoids vanishing gradient problems commonly encountered in deep networks.

• Tanh: A hyperbolic tangent function that outputs values between -1 and 1, offering a normalized range that proved useful for certain game mechanics.

• Sigmoid: While traditionally used for binary classification problems, it was interesting to observe its behavior when applied to the TORCS environment.

Each activation function imposed its unique flavor to the training dynamics, influencing convergence speed, stability, and the final performance of the agent.

### 5.2.2 Experimenting with Different Neural Network Depths

The depth of a neural network, referring to its number of layers, is pivotal in determining its ability to recognize and respond to complex patterns. However, there exists a trade-off: deeper networks can capture more intricate patterns but are harder to train and risk overfitting [17].

For the TORCS agent, experiments were conducted with:

• Shallow Networks: Consisting of fewer layers, these networks trained swiftly and showcased swift response times. Their performance, however, was limited in terms of mastering more challenging tracks.

• Deep Networks: These demanded more computational power and exhibited slower training convergence. Nevertheless, once trained, they showcased superior performance, particularly in challenging scenarios.

This experimental phase accentuated the importance of network depth and its profound influence on the agent's performance within the TORCS environment.

# 6. Design Decisions and Considerations

When conceptualizing an AI project of such intricate nature, many critical design decisions come into play. These decisions often sculpt the overall direction of the project, affecting the final results. In this section, we delve deep into some of the pivotal design choices we made, discussing their rationales and implications.

## 6.1 Initialization Techniques

Initialization, the art of setting the initial weights for neural networks, is more than just a procedural step it often spells the difference between convergence and stagnation in training regimes [7]. Our design process factored in several initialization techniques:

• Given its efficacy in deep networks where activation functions are sigmoid or hyperbolic tangent (tanh), this method was employed. It scales the weights based on the number of input and output units, promoting a faster and more stable convergence [17].

• Suitable for ReLU (Rectified Linear Units) and its variants, this technique was also considered. It helps in mitigating the vanishing gradient problem common in deep neural networks.

The choice of initialization technique is anchored in the type of activation functions used, ensuring that the weights are neither too small (leading to vanishing gradients) nor too large (resulting in exploding gradients).

## 6.2 Avoiding Overfitting and Ensuring Generalization

Overfitting, a ubiquitous concern in machine learning, denotes a model's predilection to perform exceedingly well on training data but falter on unseen or test data. Ensuring that our agent generalized well and didn't merely memorize the training scenarios was pivotal. Here's how we tackled this:

• Regularization: Techniques like L1 and L2 regularization were incorporated. By adding a penalty to the loss function, these techniques ensure that the model doesn't rely heavily on any individual feature, thus avoiding over-complexity.

• Dropout: An ingenious yet simple technique, dropout involves randomly "dropping out" neurons during training. This fosters a more distributed network representation, preventing co-adaptation of hidden units and, consequently, overfitting [15].

• Data Augmentation: By introducing minor modifications to the training data (like adding noise), the model is trained on a diverse set of scenarios, bolstering its ability to generalize to new, unseen situations.

## 6.3 Genetic Algorithm Hyperparameters

The efficiency of a genetic algorithm (GA) is, to a large extent, influenced by the choice of its hyperparameters. The intricate balance between exploration and exploitation, the chance discovery of novel solutions versus the refinement of known good ones, hinges on these settings. For this project, several hyperparameters were meticulously chosen:

• Population Size: Determining the number of individuals in each generation is crucial. A larger population can lead to greater diversity but also demands more computational resources. Conversely, a smaller population might converge rapidly but risks getting stuck in local optima.

• Crossover Rate: This hyperparameter determines the frequency at which individuals will combine their genetic material to produce offspring. A higher rate promotes diversity, while a lower one emphasizes the exploitation of existing solutions.

• Mutation Rate: Introducing random changes to an individual's genetic code can lead to the discovery of novel solutions. However, set too high, it may disrupt the beneficial characteristics acquired over generations.

• Elitism: This strategy ensures that the top-performing individuals are directly passed onto the next generation, preserving their favorable traits and accelerating convergence [10].

## 6.4 Deciding the Neural Network Architecture

Neural networks, in their vast complexity and versatility, can be likened to the human brain's intricate web of neurons. Their architecture defines their abilities, and even subtle changes can have profound impacts on performance [7]. For this project, the neural network's architecture was designed with several considerations in mind:

• Input Layer: Given the game's nature, the choice of observation states was critical. By focusing on track details and the car's angle, the network could receive concise yet informative feedback about its environment.

• Hidden Layers: Hidden layers grant a neural network its capacity to learn intricate patterns. The depth (number of layers) and breadth (number of neurons per layer) were adjusted iteratively, gauging performance against computational efficiency.

• Activation Functions: These determine a neuron's output based on its input. Choices like ReLU (Rectified Linear Unit) were preferred for their non-linearity and computational benefits, enabling faster training and reduced likelihood of vanishing gradients.

• Output Layer: The network's final layer was tailored to yield an action corresponding to the game's controls, ensuring the car could navigate the track adeptly.

# Results and Analysis

The evaluation of any machine learning or reinforcement learning endeavor transcends mere model construction. It probes the depth of performance and illuminates the intricate nuances the model reveals during its active engagement. In this context, our neural network agent's foray into the TORCS gaming environment is no exception. Herein, we dissect its performance, weaving in the factors, metrics, and the myriad changes made throughout the training.

## 7.1 Training Results over Generations

The vast expanse of reinforcement learning, intertwined with the distinct challenges posed by the TORCS environment, threw open the gates to a rigorous, yet insightful experiment. The agent's evolutionary trajectory, strewn across several generations, narrated a saga of consistent refinement.

During the early phases, the agent's performance echoed the predictable hesitations of a beginner—characterized by errors, frequent off-track excursions, and a budding grasp of game mechanics. However, with each generation, a distinct improvement emerged, painting a clear testament to the genetic algorithm's prowess.

Interspersing this journey was our decision to compare three uniquely trained neural network agents. The distinctions lay in their observations and training complexities:

**Agent 1:** Operated on observations of track, angle, and trackPos, trained with a population of 5 spanning 200 generations. Its movement pattern was reminiscent of a snake, completing the track in 2:15 minutes.

Demonstrated its peak fitness within 50 generations. Any subsequent gains were marginal, witnessing a meager ~0.0001% increment post the 50th generation.

And the below is the video, how agent performed.   


**Agent 2:** Using the same observations but amplifying the training population to 50 and extending over 1000 generations, it began cautiously. However, it soon gained momentum, maintaining a straight path and clocking a completion time of 2:08 minutes.

Similarly, it reached its zenith of fitness within the initial 50 generations. Thereafter, the growth trajectory was almost negligible, inching upwards by approximately 0.0001% for subsequent episodes.

And the below is the video, how agent performed.



**Agent 3:** This agent, while only focusing on trackPos and angle, shared the training parameters of Agent 2. Intriguingly, its driving bore a stronger snake-like pattern but showcased commendable efficiency, completing the lap in a mere 1:54 minutes.

Proved to be an outlier, achieving its maximum fitness remarkably by the 8th generation. Astonishingly, this peak was sustained without any significant deviations for the subsequent 992 generations. The starkness of this training dynamic can be visualized in the above graph.

And the below is how agent is performed



## 7.2 Performance Metrics and Evaluation

Our analytical lens zoomed into a plethora of metrics. These comprised the Track Completion Percentage, offering insights into consistency; the Lap Time, a direct reflection of speed proficiency; and Collisions, shedding light on navigational precision.

For a more granular understanding of the agent's capabilities, we turned to a set of meticulously chosen performance metrics. These included:

• Track Completion Percentage: This metric offered insights into how consistently the agent could complete the race circuit without veering off track.

• Lap Time: The time taken to complete a lap provided a tangible measure of the agent's speed proficiency.

• Collisions: A count of how frequently the agent collided with track boundaries provided insights into its navigational accuracy.

Comparative analysis against hand-coded and human agents further enriched the evaluation process. While the neural agent's lap time progressively approached those of seasoned human players, certain intricate maneuvers executed by human players remained a challenge for the AI agent. This juxtaposition between human intuition and algorithmic precision surfaced throughout the evaluation phase.

## 7.3 Efficiency vs. Accuracy Trade-offs

The AI gaming domain constantly teeters between efficiency and accuracy. While efficiency emphasizes swift game state processing, accuracy underscores optimal decisions, sometimes at the cost of speed.

Our agent's journey was an emblematic representation of this battle. Initially, our pursuit of accuracy occasionally compromised response speed. But as generations unfolded, a harmonious amalgamation of both emerged, epitomizing the deep learning potential.

# Impact of Variations

A dive into AI's depths, especially in gaming, often demands iterations and fine-tuning. Our agents, with their distinct training and observational parameters, narrated unique tales of performance, robustness, and efficiency.

## 8.1 Observations on Model Complexity vs. Performance

It's a commonly held belief in AI that more complex models inherently deliver superior performance [7]. However, our exploration into the intricacies of neural networks within the TORCS environment painted a more nuanced picture. As model complexity increased, there was an observable improvement in performance, up to a point. Beyond this inflection point, the incremental gains in performance began to diminish, and the increased computational overhead started taking a toll on real-time responsiveness.

This phenomenon is emblematic of the bias-variance trade-off. Simple models might lack the necessary depth to understand the game dynamics fully, leading to underfitting. Conversely, overly complex models might become excessively attuned to the nuances of the training data, making them prone to overfitting when introduced to new race scenarios [18].

## 8.2 Insights on Essential Game Observations for Model Efficiency

The choice of input data - the observations from the game - plays a pivotal role in the success of an AI agent. Initially, a comprehensive set of observations, including track details, angle, track position, and more, were fed to the agent. While this provided a holistic view, it raised the question: Are all these observations equally critical?

A phase of selective omission was initiated, wherein certain observations were sequentially removed. It was revelatory to find that some observations, like track and angle, were integral to the agent's performance, while others could be omitted without significant detriment to performance [16]. This pruning of non-essential data contributed to a leaner, more efficient model, optimizing both memory and computational time.

## 8.3 Comparison of Various Models' Robustness

An agent's robustness defines its adaptability. Each of our models underwent assessments against unfamiliar terrains and unpredictable race conditions. The revelations were enlightening. While Agent 2's straight-driving pattern emerged as a more generalized strategy, suitable for a wide array of race scenarios, Agents 1 and 3, with their snake-like movements, illustrated the challenges and potential pitfalls of specialized strategies.

# 9. Comparison with Other Agents

In the arena of simulated racing, an agent's prowess is often gauged not just by its ability to complete circuits in minimal time but also by its tactical acumen, adaptability, and understanding of the racing environment. This project, which centered on a neural network agent, provided a fascinating opportunity to pit artificial intelligence against the nuanced cognitive abilities of humans.

## 9.1 Human-driven Agent vs. Neural Network Agent

Human players possess an intrinsic tactical sense, honed by years of gaming experiences and innate human intuition [16]. They can read the flow of the game, anticipate opponent moves, and dynamically adjust strategies on the fly. The neural network agent, on the other hand, harnesses the power of vast amounts of data and iterative learning. While its initial forays might lack the polish of seasoned players, its advantage lies in its ability to learn and adapt rapidly. Over time, as the neural network continues to train, it starts identifying optimal pathways, braking points, and overtaking maneuvers, which can often surpass human capabilities.

Human players inherently possess the ability to adapt to new tracks, weather conditions, and car dynamics based on past experiences [17]. The neural network agent's adaptability, in contrast, depends on its training data. If exposed to diverse racing conditions during its training phase, the neural network can demonstrate impressive adaptability. However, its responses in unseen scenarios might be a tad unpredictable, a challenge often addressed by expanding its training set or refining its architecture.

While humans can leverage their real-world driving experiences and apply them to the virtual world of TORCS, the neural network agent lacks such experiential wisdom [16]. It relies solely on the patterns it discerns from its training data. As a result, while a human might naturally avoid certain risky maneuvers based on real-world understanding, the neural network agent might opt for them if its training data suggests a potential reward.

Neural network agents are the epitome of consistency. Once trained, they will perform maneuvers with robotic precision every time [5]. Humans, while prone to occasional errors, bring a level of creativity and unpredictability to the race, often surprising opponents with ingenious tactics.

In conclusion, while the neural network agent showcased remarkable capabilities, especially in its ability to learn and optimize its performance, it doesn't completely overshadow the nuanced and unpredictable nature of human-driven agents. The ideal racing agent, perhaps, lies somewhere in between, combining the consistent precision of neural networks with the tactical genius and adaptability of human cognition.

## 9.2 Traditional Hand-coded Agent

While our primary focus revolved around training neural agents, an integral aspect of our experimentation was the development of a rule-based agent. This agent served as both a benchmark and a tool to understand the intricacies of the TORCS environment. Below, we delve into the workings of this agent and discuss how we used it to infer the game's track layout.

Hand-coded agents, or rule-based agents, are often the product of meticulous and exhaustive programming, where developers define a set of rules and conditions that determine the agent's behavior [8]. Such agents are a testament to human ingenuity and deep domain expertise. They operate based on predefined logic and lack the ability to learn or adapt from their experiences. In the context of TORCS, let's delve deeper into the differences between our neural network agent and its traditional counterpart:

Hand-coded agents are deterministic. Their actions in a given scenario will always be the same, reflecting the coded logic. Contrastingly, our neural network agent continually refines its strategy, learning from its successes and failures [16].

Crafting a competent hand-coded agent requires in-depth domain knowledge. Developers need to anticipate a myriad of game scenarios and code appropriate responses. The neural network agent, once designed, learns to navigate the environment on its own, thus potentially reducing the manual intricacies of coding.

While hand-coded agents are consistent in their strategies, they might falter when faced with unanticipated scenarios. Neural network agents, especially those reinforced through techniques like genetic algorithms, adapt and evolve to optimize their performance.

Over time, as the gaming environment changes or as new strategies emerge, maintaining and updating a hand-coded agent can be a significant task. In contrast, a neural network agent can be retrained, allowing it to adapt to new conditions or strategies [8].

While our endeavor with the neural network agent for TORCS yielded promising results, it's essential to appreciate the legacy and precision of hand-coded agents. They stand as pillars of an era where human expertise directly translated into in-game strategy, setting the foundation upon which modern AI-driven approaches are built.

### Understanding the Hardcoded Agent:

Initialization: The agent initializes with specific game-environment related parameters, including cumulative displacement, relative position on the track, and the current direction of the car.

Input Functionality: Observations are transformed into a contiguous numpy array. Extracted features, such as the relative car-to-track angle, track position, speeds, and opponent details, play a pivotal role in decision-making.

Decision Making: Decisions revolve around observations. The agent evaluates the relative angle and track position to determine its course of action, adjusting direction and speed as necessary. There are also special conditions that the agent considers, ensuring responsiveness to specific game states.

Calculating Displacement: The agent calculates its displacement using the Pythagoras theorem, considering speeds in the X and Y directions to determine its overall movement.

### 10.1 Interpreting the Track Layout:

The agent continually updates its track\_positions based on the car's relative position on the track and its directional changes. By observing the printed outputs and plotting them, we can draw inferences about the track's layout:

The Length of track\_positions: This metric serves as an indicator of the number of observations or decisions made. It represents time or steps in the game, and its progression offers insights into the car's journey.

current\_direction: This value provides the accumulated or net direction change. By plotting these values against the number of decisions, a trajectory of the car's movement is formed. Peaks and troughs depict significant directional changes, while steady patterns might suggest consistent curves or straight paths.



In essence, this graphical interpretation gives us a rudimentary map of the track's layout. It's an illustrative way to understand how the car navigates the track based on the rules set in the hardcoded agent.

## 10.2 Detailed Comparative Analysis and Insights

Understanding the performance intricacies of different agents – be it those powered by neural networks, manual coding logic, or the unpredictable nature of human decision-making – is paramount for robust analysis. The objective of this section is to delineate the strengths, limitations, and idiosyncrasies of each entity in the context of the TORCS environment.

*Neural Network-Based Agent:*

• Strengths: Unlike hand-coded agents, neural network agents learn and adjust from experience [16]. This allows them to develop strategies dynamically, adapting to various in-game situations. These agents excel in situations that require the recognition of intricate patterns and nuanced changes, making them adept at handling unexpected game scenarios.

• Limitations: Neural networks, especially deeper architectures, can require significant training time to converge to optimal or near-optimal policies [7]. Without proper regularization and training techniques, these agents can overfit to theioverfit scenarios, leading to subpar performance in unfamiliar situations.

*Human Players:*

• Strengths: Human players bring a unique blend of intuition and strategy, developed over years of gaming experience. Humans can swiftly adapt to unpredictable game dynamics, making on-the-fly decisions based on a mixture of instinct, experience, and logic.

• Limitations: Humans can occasionally be inconsistent, with performance varying based on factors like fatigue, distractions, or emotions. In high-speed games, the human reaction time might not match the rapid decision-making of an AI agent.

*Hand-Coded Agents:*

• Strengths: With hand-coded agents, developers have full control over their actions. This results in predictable and consistent behavior, beneficial for benchmarking purposes. These agents can be finely tuned for specific game scenarios, often performing exceptionally well in those contexts.

• Limitations: coded agents don't learn from experience. If they encounter an unexpected game situation, they might not handle it optimally. As the game's complexity grows or changes, maintaining and updating a hand-coded agent can become labor-intensive.

Insights: From the comparative analysis, it's evident that no single agent type reigns supreme in all scenarios. Neural network agents bring adaptability and pattern recognition to the table, humans offer intuition and versatility, while hand-coded agents provide determinism and specialized optimization. A hybrid approach, combining the strengths of these agents, might be a promising avenue for future research and development in the domain of AI-driven gaming.

# 11. Challenges and Lessons Learned

In the voyage of marrying artificial intelligence with the intricate dynamics of the TORCS environment, the journey was neither smooth nor straightforward. Every challenge faced was, however, a stepping stone to a deeper understanding and improvement of the model. This section delves into the various obstacles encountered and the invaluable lessons derived from them.

## 11.1 Debugging and Development Issues

One of the most time-consuming facets of the project was identifying and rectifying unexpected behaviors and bugs within the system. Debugging a system that combines game dynamics with neural networks presents a unique set of challenges. The complexity of TORCS' vast environment meant that sometimes, errors were not immediately evident.

• Initialization Troubles: The model's initial weights and biases sometimes led to erratic behavior. There were instances where the car would consistently veer off track or fail to move altogether. Diagnosing these issues required thorough inspection of the initialization methods and subsequent adjustments.

• Hyperparameter Tuning: Deciding on the best set of hyperparameters for training was more art than science. Several combinations were trialed and tested to achieve a balance between learning speed and stability [5].

• Environment Sync: Ensuring that the neural network agent and the TORCS environment were perfectly synchronized was pivotal. Minor inconsistencies could result in substantial performance degradation.

## 11.2 Lessons from Failed Generations

Evolutionary algorithms, like the ones used in training the agent, inherently involve a lot of trial and error. Some generations of agents failed miserably in their tasks, but they were the richest sources of insights:

• Overfitting: There were generations where agents excelled in specific tracks but faltered in others. This highlighted the classic machine learning issue of overfitting, where the model becomes too attuned to its training data and struggles with novel scenarios [7].

• Diversity is Key: A significant realization was the importance of maintaining genetic diversity within the agent populations. Without diversity, the population risked converging too early to sub-optimal solutions.

## 11.3 Refinements and Iterative Development

Like sculpting a masterpiece from a block of stone, the final agent was a product of numerous iterations and refinements:

• Modifying Observations: Initial models incorporated several environmental inputs. Over time, it became clear that simplifying these inputs – focusing on track and angle, for instance – led to better generalization and performance.

• Flexible Architectures: Instead of rigidly sticking to an initial architecture, the model underwent several architectural adjustments. Layers were added or removed, and activation functions were swapped to improve efficiency and adaptability [7].

• Continuous Evaluation: The agent's training wasn't a one-time affair. Continuous evaluations and "re-trainings" ensured the agent remained robust across different TORCS tracks and scenarios.

# 12. Discussion

In the ever-evolving landscape of gaming and artificial intelligence, this project serves as an intersection of both domains. It not only explores the capabilities of neural networks but also sheds light on their potential applications in enhancing the gaming experience. The discussion that follows elaborates on the project's achievements, identifies its potential limitations, and contemplates the future of neural network agents in gaming.

## 12.1 Achievements and Strengths

The project's central achievement is the successful creation of a neural network agent capable of navigating the dynamic environments of the TORCS game. This success can be attributed to several inherent strengths:

Unlike traditional scripted agents, the neural network agent demonstrated an ability to adapt to varying scenarios within the game, mimicking a learning curve akin to human players [17].

Utilizing the Gym framework ensured standardized benchmarks, making the project's results comparable with other state-of-the-art agents in similar environments.

With TensorFlow at its core, the agent can be further trained and refined using more complex and deep neural architectures, enabling it to tackle even more challenging tasks in the future [7].

## 12.2 Potential Pitfalls and Limitations

While the project has made commendable progress, it's imperative to acknowledge its limitations:

• Generalization: While the agent showed proficiency in the TORCS game, it remains to be tested if this proficiency can be transferred to other, more complex games or real-world driving scenarios.

• Computational Resources: Training deep neural networks demands substantial computational power. The project's scalability is, to an extent, bound by the available resources.

• Training Time: Neural network agents, especially those employing reinforcement learning, can require significant amounts of training time to reach optimal performance.

## 12.3 The Future of Neural Network Agents in Gaming

With the advancements in both gaming and AI sectors, neural network agents are poised to play a transformative role:

• Personalized Gaming: In the future, we can expect games that adapt to individual players, offering challenges tailored to their skill levels, achieved through learning agents analyzing players' behaviors [8].

• Collaborative Gaming: Neural network agents might soon serve not just as opponents but as teammates, understanding and predicting human player strategies to offer collaborative gameplay experiences.

• Real-World Applications: The skills learned by these agents in virtual environments could be transitioned to real-world applications, particularly in sectors like autonomous driving, robotics, and even sports training.

# 13. Further Experiments and Considerations

Exploration in the domain of AI-driven racing within TORCS provides an avenue to probe deeper into the nuances that can influence an agent's performance. While the preliminary outcomes from the implemented neural network agent have been promising, the domain of machine learning and neural networks is vast, with numerous avenues to explore and experiment.

## 13.1 The Role of Hyperparameters

Hyperparameters are crucial determinants in the training of neural networks, directly impacting the speed and quality of learning. A nuanced understanding of their role can lead to enhanced training performance and better generalization in the real world.

### 13.1.1 Impact of Learning Rate

Learning rate, often seen as the most vital hyperparameter, determines the step size during model optimization. Too large, and the model may overshoot optimal points; too small, and it might get stuck in local minima or take ages to converge. Systematically adjusting the learning rate and observing the model's response could provide deeper insights into achieving optimal performance.

### 13.1.2 Importance of Batch Sizes

Batch size influences the gradient estimation, with larger sizes offering a more accurate representation at the cost of computational power. On the other hand, smaller batch sizes can introduce noise in the gradient, potentially aiding in escaping local minima but at the risk of unstable training [17]. Striking a balance and understanding its interplay with other factors like learning rate can be an area of investigation.

## 13.2 Potential for Incorporating Other ML Techniques

While neural networks form the core of the current agent, the broader field of machine learning offers numerous strategies and techniques that can be integrated or experimented with to further refine performance.

### 13.2.1 Potential for Transfer Learning

Transfer learning, the art of leveraging knowledge from one task to improve performance on another, could be a game-changer [8]. By using pre-trained networks and fine-tuning them on TORCS, there might be a potential to expedite training and achieve higher performance levels. For instance, agents trained on similar racing simulations could offer a head start when fine-tuned for TORCS.

### 13.2.2 Role of Reinforcement Learning

Though the initial experiments utilized a form of supervised learning, diving deeper into the realm of reinforcement learning (RL) offers potential. Reinforcement learning, with its focus on learning by interaction and receiving feedback from the environment, might be more naturally aligned with the dynamics of a game like TORCS [10]. Exploring strategies like Q-learning or policy gradient methods might unveil new dimensions of performance enhancement.

# 14. Recommendations

In the journey of crafting a neural network agent for TORCS, numerous lessons were gleaned, many through success, but arguably more through challenges and iterations. Based on the insights garnered throughout this endeavor, a series of recommendations can be laid out for different stakeholders:

## 14.1 For Aspiring Game Developers

• Embrace AI: Modern games are increasingly harnessing the prowess of AI to craft more immersive and unpredictable gaming experiences [8]. Aspiring developers should delve deep into AI algorithms not just as tools, but as co-creators, weaving together gaming storylines and mechanics that previously seemed beyond reach.

• Holistic Learning: A game isn't just about coding. It encompasses design, storytelling, graphics, and user experience. To stand out, one must blend technical acumen with artistic flair [4]. Continuous learning through online platforms, seminars, and workshops is pivotal.

• Engage with Communities: Game development, like any art, thrives on feedback. Being actively involved in communities such as GitHub, Stack Overflow, or dedicated gaming forums can provide invaluable critique and foster collaborations.

## 14.2 For Improving the Model

• Data Augmentation: The model's performance can significantly benefit from data augmentation techniques, feeding it more varied and challenging scenarios, refining its understanding and adaptability [2].

• Experiment with Architectures: While the model used was effective, there's a myriad of neural network architectures waiting to be explored. Architectures like Convolutional Neural Networks (CNNs) might offer nuanced interpretations of the gaming environment [7].

• Regularization Techniques: To prevent overfitting and ensure that the model generalizes well, consider employing dropout or L2 regularization. This can enhance the model's robustness in diverse scenarios.

## 14.3 For Efficient Training

• Parallel Processing: Deploy parallel training sessions across multiple GPUs. This allows for concurrent evaluations of different hyperparameters, reducing the time to find the optimal model configuration [19].

• Early Stopping: To conserve resources and time, implement early stopping. This halts the training process if the model's performance on a validation set doesn't improve after a specified number of iterations.

• Adaptive Learning Rates: Employ optimizers that adjust learning rates in real-time, such as Adam or Adagrad. They navigate the optimization landscape more effectively, ensuring quicker convergence6.

# 15. Explaining the Genetic Agent and Training Mechanism for TORCS (Script explanation)

In-depth explanation of my script

## Imported Libraries:

Before diving into the main logic, it's crucial to ensure we have all the tools and libraries at our disposal. Here's a closer look:

* **gym & gym\_torcs:** These are integral to the setup. The gym library offers a suite of environments for training agents, and gym\_torcs specifically caters to the TORCS car simulation. Think of them as the virtual playground where our agents will learn and be tested.
* **tensorflow & numpy**: TensorFlow is the backbone that helps in building our agent's brain, whereas Numpy assists in mathematical operations and data handling.
* **Auxiliary Libraries:** Libraries like os, atexit, json, and matplotlib play supportive roles, handling file operations, program exits, data serialization, and data visualization, respectively.

**2. Global Variables:**

These are akin to the rules and settings of a game. They provide a guideline and structure within which the program operates.

* **PATH:** This is the location on the computer where the trained agent's brain (model) will be saved. Think of it as a save point in a video game. If there's a previously saved agent, we can pick up from where we left off rather than starting anew.
* **Agent's Perception & Action:** The variables input\_size and output\_size determine how much information the agent takes in from its surroundings and how many different ways it can react. In the context of TORCS, it defines how the agent perceives the racetrack and how it decides to control the car.
* **Genetic Algorithm Settings:** Variables such as population\_size and maximum\_generations dictate the genetic algorithm's behavior. They decide the number of agents in each generation and how many generations the training should go through. It's similar to setting the number of players in a team and deciding the number of matches in a tournament.
* **Initial Message:** The printed message about pressing "F2" for the preferred camera angle serves as a user-friendly note. It ensures that those running the code can adjust the visual display to their liking, enhancing the user experience.

## The Genetic Agent Architecture and Functionality

In our framework, the pivotal player is the GenAgent, which serves as the central decision-making entity for our TORCS racing simulation. In essence, GenAgent can be visualized as the 'brain' of the car, driving it across the track. This agent utilizes principles from neural networks and has been optimized for the TORCS environment.

**Architectural Overview:**

*Core Components:*

GenAgent is a subclass of the tf.keras.Model, an abstraction provided by TensorFlow for creating machine learning models. Within this agent, we find two primary dense layers (fc1 and fc2). These are fully connected layers, a fundamental building block in neural networks. Every neuron in a dense layer is connected to every neuron in the previous layer.

**Layer Configuration:**

**fc1:** The initial layer (fc1) serves as the input layer and has a dimensionality specified by hidden\_size. This layer's role is to receive raw information (like track position, speed, etc.) from the TORCS environment and start the process of decision-making.

**fc2:** The subsequent layer (fc2) is responsible for producing the final decisions or actions that the agent should take. Its dimensionality is equal to the output\_size, which signifies the number of potential actions the agent can execute.

**Activation Functions:** Activation functions introduce non-linearity into the model, enabling it to learn complex relationships. In our agent:

The ReLU (Rectified Linear Unit) activation function is applied after the first layer (fc1). It is a popular choice for deep learning models due to its simplicity and efficiency.

The final output, after passing through fc2, is processed using the tanh function, ensuring the actions produced are in a valid range for the TORCS environment.

**How Does GenAgent Make Decisions?:**

1. When GenAgent is deployed in the TORCS environment, it receives sensory information as an input vector (like the angle of the car with respect to the track, its position, and sensor readings).
2. This raw data passes through fc1, where initial processing occurs.
3. The processed data, now with added insights from fc1, travels to fc2.
4. The output of fc2 represents the agent's decisions on how to act, which can include steering directions, acceleration magnitude, etc.
5. These decisions are then executed in the TORCS environment, and the agent observes the result of its actions, culminating in a reward or score.

## Genetic Operations for Agent Evolution:

The process of training agents using genetic algorithms is inspired by the natural process of evolution. Just as species evolve over time through natural selection, reproduction, and mutation, our agents evolve across generations through a series of genetic operations. These operations are crucial for ensuring diversity, adaptation, and convergence to optimal solutions. Let's break them down:

* **Mutation:** Mutation introduces small, random changes to an agent's parameters. Imagine you're trying different combinations to unlock a treasure chest. If you keep trying the same combinations, you might never unlock it. Mutation is like slightly tweaking a combination to see if it works better. By introducing random changes, mutation ensures the population doesn't get stuck with the same solutions and explores new possibilities. For each parameter of an agent, a small random value is added. This random value is drawn from a normal distribution with a standard deviation of 0.1, ensuring the changes are minor but enough to introduce variability.
* **Crossover:** Crossover, or mating, combines the characteristics of two parent agents to produce a child agent. Think of this as mixing two recipes to produce a new dish. By combining the strengths of two good agents, we hope to produce an even better agent. This operation promotes sharing of beneficial traits within the population. For each parameter in the parents, a binary mask (a series of 0s and 1s) is generated randomly. If the mask has a '1' for a specific parameter, the child will inherit that parameter from the first parent; otherwise, it'll inherit from the second parent.
* **Next Generation Creation:** Once we've evaluated all agents in the current generation, we need to produce a new generation. This is akin to a changing of the guard. Older agents retire, and newer agents, produced through crossover and mutation, take their place. This ensures a fresh population that's more adapted to the challenges.

How It Works:

* **Elitism:** We directly carry forward the top-performing agents to the next generation without any change. This ensures that the best solutions aren't lost. To produce the remaining agents for the next generation, parents are chosen based on their fitness. Better-performing agents have a higher probability of being chosen. These selected parents then undergo crossover to produce child agents. A small percentage of these child agents are mutated to introduce variability.

Episode Runner:

**run\_episode Function:** This function runs a single episode using a provided agent and returns the total reward obtained. It's crucial for evaluating an agent's performance.

**Agent Initialization and Environment Handlers:**

Functions like initialize\_agent and cleanup ensure that agents are primed for interaction and that the environment is closed properly.

## Main Execution Logic:

**Model Loading:** If a trained model is found at the specified PATH, it is loaded, and its performance is tested in the environment.

**Training Process:** If no trained model is found, the genetic algorithm kicks in:

1. Initialize a population of agents.
2. For each generation, evaluate the fitness of each agent (i.e., its performance in the game).
3. Generate a new population of agents based on the fitness of the previous generation using elitism, crossover, and mutation.
4. This loop continues for the specified number of generations or until the user interrupts the process.

**Post-Training:** After training, the best agent is saved and then tested in a visual environment to evaluate its performance.

# 16. Project Management and Risk Analysis

## Project Management:

The primary objective of the project was clearly defined: to develop a neural network agent adept at playing the game Torcs. This direction was established at the outset to ensure the alignment of all subsequent efforts. Research and planning were the initial steps taken. Before diving into the core development, we dedicated a significant amount of time towards researching the optimal techniques that would allow the AI script to effectively interface with the Torcs game. During this phase, we explored a multitude of repositories, hoping to find a solution that would mitigate the challenges of integration.

The development phase of the project was multifaceted. The initial challenge we grappled with was establishing an effective medium of communication between the script and the game. Several repositories were put to the test in the hopes of overcoming this hurdle. Despite the myriad of options available, many proved incompatible or insufficient for our needs. At one point, out of sheer necessity and in the absence of a ready-made solution, we contemplated crafting a custom wrapper for the game. For this, we even sought the assistance of AI chatbots. Our persistence eventually paid off when we stumbled upon the gym-torcs wrapper on GitHub, which turned out to be our saving grace.

However, the challenges didn't cease there. Even with the discovery of the gym-torcs wrapper, we faced a steep learning curve during its installation phase. The primary obstacle was the wrapper's documentation, which was less than straightforward. Thankfully, after scrutinizing the repository, we unearthed the 'install script.sh', which became instrumental in navigating the setup process.

Once the integration barriers were surmounted, our next task was to rigorously test the neural network agent within the Torcs environment. This process allowed us to identify areas of improvement and refine the agent's efficiency. Throughout the project's life cycle, we ensured that all steps, challenges, and resolutions were meticulously documented. This would not only serve as a reference for our team but also as a guiding resource for any future projects of a similar nature.

## Risk Analysis:

Several risks were identified at the outset of the project. The primary one was the integration challenge. We recognized the potential difficulty in marrying the AI script with the game, which required us to be prepared for extensive research and testing of various repositories. We were also wary of the possibility that many available repositories might not be up to the mark, necessitating a diverse research approach.

Another anticipated risk was the daunting prospect of having to develop a custom wrapper. While this approach would offer more control, it would be considerably time-consuming and might not ensure full compatibility. Therefore, we kept this as a last resort, relying instead on existing solutions and, when necessary, leveraging AI chatbot assistance.

We were also cognizant of the potential hitches that could arise during installation and setup, especially with unclear documentation. To mitigate this, our strategy was to delve deep into the repositories, look out for overlooked setup guides, and even consider reaching out to the community for guidance.

Performance was another area of concern. We understood that the agent might not showcase optimal performance right out of the gate. This necessitated a strategy of continuous testing and iterative refinement based on performance metrics. Additionally, with the challenges we faced, there was a looming risk of time overruns. To counteract this, we incorporated a time buffer and remained open to revising our milestones or procuring additional resources if needed.

In sum, by acknowledging potential roadblocks and arming ourselves with adaptive strategies, we were poised to navigate through challenges effectively and achieve the project's ambitious objectives.

# 15. Conclusion

In the odyssey of AI-driven gaming, our journey with TORCS and the synthesis of neural network agents brought forth enlightening revelations that could steer the future direction of both gaming and AI research.

## 15.1 Summary of Findings

This project was not merely an exercise in agent creation but a multi-faceted exploration into the integration of cutting-edge AI with the intricate dynamics of a racing simulation game. Key findings from this endeavor include:

• Performance: The neural network agent, nurtured through advanced reinforcement learning techniques, exhibited commendable performance. While there were initial challenges in its navigation proficiency, subsequent generations showed marked improvement, underscoring the power of iterative learning [16].

• Comparison: When juxtaposed with human and hand-coded agents, our neural network agent demonstrated a unique blend of computational efficiency and adaptive learning. While it might not always surpass human intuition or the deterministic precision of hand-coded agents, its capability to learn and adapt offers a glimpse into a more dynamic gaming future.

• Environment Sensitivity: Alterations to observation states, especially limiting them to just track and angle, highlighted the agent's adaptability. It was intriguing to observe how these seemingly minor tweaks could drastically affect performance metrics.

## 15.2 Implications for the Gaming Industry

Our findings transcend the realm of academic curiosity; they have tangible implications for the broader gaming industry:

• Dynamic Game Play: The incorporation of AI agents, particularly those built on neural networks and genetic algorithms, can usher in a new era of dynamic gameplay. These agents, with their inherent ability to learn and adapt, can offer players novel challenges, ensuring that no two gaming experiences are identical [8].

• Customization: As AI agents become more sophisticated, there's potential for personalizing gaming experiences. Imagine a world where the game evolves in real-time, adapting its difficulty and strategy based on the player's skills and style.

• Training & Simulation: Beyond entertainment, these findings can redefine training simulators. For industries like aviation or motorsports, AI agents like the ones developed in this project can be invaluable in creating realistic, adaptable training scenarios.

The horizons of what AI can achieve in gaming are expansive. This project, while a mere drop in the vast ocean of possibilities, accentuates the potential symbiosis of gaming and AI—a collaboration that could redefine entertainment, training, and simulation paradigms.

# 17. Future Work

In the ever-evolving world of artificial intelligence and gaming simulations, research rarely comes to a full stop. Even as the neural network agent for TORCS continues to deliver impressive performances, there remain uncharted terrains and untapped potential. This section sheds light on some promising avenues for future research and development.

## 17.1 Incorporating Real-time Feedback

One of the potential next steps involves leveraging real-time feedback mechanisms to enhance the agent's learning experience [5]. As of now, the agent's learning predominantly occurs in batches after episodes. However, the adoption of a continuous feedback system can serve two primary purposes:

• Immediate Correction: By getting feedback instantaneously, the agent can adapt and rectify its behavior on the fly. This can lead to more efficient and faster learning, especially in complex environments.

• Human-AI Collaboration: Real-time feedback mechanisms can be integrated with interfaces that allow human players to offer immediate inputs, nudging the agent towards desired behaviors or strategies.

## 17.2 Broadening to Other Racing Simulations

While TORCS provides a rich environment to test and hone our neural network agent, exploring other racing simulations can offer fresh challenges and diverse dynamics [5]. Games like Assetto Corsa or Project CARS have distinct physics engines and gameplay mechanisms. Training the agent on these platforms can:

By encountering varied challenges, the agent can develop a more comprehensive skill set and become versatile. Different games can serve as benchmarks, ensuring that the AI's competence isn't just restricted to TORCS but is universally competitive.

### 17.3 Exploring Multi-agent Dynamics.

To date, our agent predominantly raced against predetermined AI or human players. An intriguing prospect lies in having multiple neural network agents, each trained with slightly varied parameters, competing against one another. When two AI agents compete, they might evolve strategies that are unforeseen and innovative, potentially outclassing human strategies. In races that demand teamwork, like relay races, multi-agent systems can be trained to collaborate, learn, and adapt together.

# 18. Acknowledgments

Embarking on a journey as intricate and profound as this project on neural network agents in the domain of the TORCS game brought to light the interconnectedness of knowledge and the invaluable contributions of various individuals and institutions. It would be an oversight not to pay homage to the myriad forces that converged to make this research endeavor both possible and successful.

Foremost, I'd like to express my deepest gratitude to my supervisor, Dr Michael Fairbank, whose guidance, and unwavering support were instrumental in charting the course of this project. His insights into neural networks and reinforcement learning were both enlightening and foundational to the very structure of this research.

A deep thanks for the dosssman (GitHub repository), without this channel, I cant see myself completing this project, and with the getting started script provided with his documentation, made everything easy and smooth.

The support of the University of Essex and CSEE Department cannot go unmentioned. From providing the necessary computational resources to fostering an environment conducive to rigorous academic pursuit, the institution's role in the fruition of this research is undeniably immense.

Lastly, a nod to the countless authors and researchers whose published works acted as lighthouses, guiding my exploration and understanding in the vast sea of artificial intelligence and game theory. Their collective wisdom, encapsulated in papers, articles, and books, formed the bedrock upon which this project stands.

In conclusion, while this report bears my name, it is, in truth, a testament to the collective spirit of the academic community, the collaborative nature of modern research, and the countless hours dedicated by mentors, peers, and pioneers in the field of artificial intelligence and gaming.

# References

|  |  |
| --- | --- |
| [1] | Wymann, Bernhard., Espié, Eric., Guionneau, Christophe.,Dimitrakakis, Christos.,Coulom, Rémi.,Sumner, Andrew, "Torcs, the open racing car simulator," *Software available at http://torcs. sourceforge. net,* p. 2, 2000. |
| [2] | Silver, David., Hubert, Thomas., Schrittwieser, Julian., Antonoglou, Ioannis., Lai, Matthew., Guez, Arthur., Lanctot, Marc., Sifre, Laurent., Kumaran, Dharshan., Graepel, Thore, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science,* vol. 362, no. American Association for the Advancement of Science, pp. 1140-1144, 2018. |
| [3] | Brockman, Greg., Cheung, Vicki., Pettersson, Ludwig., Schneider, Jonas., Schulman, John., Tang, Jie., Zaremba, Wojciech, "Openai gym," *arXiv preprint arXiv:1606.01540,* 2016. |
| [4] | Tekinbas, Katie Salen., Zimmerman, Eric, Rules of play: Game design fundamentals, MIT press, 2003. |
| [5] | Sutton, Richard S., Barto, Andrew G, Reinforcement learning: An introduction, MIT press, 2018. |
| [6] | Silver, David., Huang, Aja., Maddison, Chris J., Guez, Arthur., Sifre, Laurent., Van Den Driessche, George., Schrittwieser, Julian., Antonoglou, Ioannis., Panneershelvam, Veda., Lanctot, Marc, "Mastering the game of Go with deep neural networks and tree search," *nature,* vol. 529, no. Nature Publishing Group, pp. 484-489, 2016. |
| [7] | Goodfellow, Ian., Bengio, Yoshua., Courville, Aaron, Deep learning, MIT press, 2016. |
| [8] | Yannakakis, Georgios N., Togelius, Julian, Artificial intelligence and games, vol. 2, 2018. |
| [9] | Hunicke, Robin., LeBlanc, Marc., Zubek, Robert, "MDA: A formal approach to game design and game research," in *Proceedings of the AAAI Workshop on Challenges in Game AI*, San Jose, CA, 2004. |
| [10] | M. Mitchell, An introduction to genetic algorithms, MIT press, 1998. |
| [11] | M. Mitchell, An introduction to genetic algorithms, MIT press, 1998. |
| [12] | Abadi, Martín., Barham, Paul., Chen, Jianmin., Chen, Zhifeng., Davis, Andy., Dean, Jeffrey., Devin, Matthieu., Ghemawat, Sanjay., Irving, Geoffrey., Isard, Michael, "{TensorFlow}: a system for {Large-Scale} machine learning," pp. 265-283, 2016. |
| [13] | Summerville, Adam James., Snodgrass, Sam., Mateas, Michael., Ontanón, Santiago, "The vglc: The video game level corpus," *arXiv preprint arXiv:1606.07487,* 2016. |
| [14] | Lucas, Simon M., Kendall, Graham, "Evolutionary computation and games," *IEEE Computational Intelligence Magazine,* vol. 1, pp. 10-18, 2006. |
| [15] | Hauptman, Ami., Sipper, Moshe, "Evolution of an efficient search algorithm for the mate-in-N problem in chess," in *European Conference on Genetic Programming*, Springer, 2007. |
| [16] | Perez, Diego., Rohlfshagen, Philipp., Lucas, Simon M, "The physical travelling salesman problem: WCCI 2012 competition," in *2012 IEEE Congress on Evolutionary Computation*, IEEE, 2012. |
| [17] | LeCun, Yann., engio, Yoshua., Hinton, Geoffrey, "Deep learning," *nature,* vol. 521, no. Nature Publishing Group UK London, pp. 436-444, 2015. |
| [18] | Nair, Vinod., Hinton, Geoffrey E, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010. |
| [19] | Stanley, Kenneth O., Miikkulainen, Risto, "Evolving neural networks through augmenting topologies," *Evolutionary computation,* vol. 10, no. MIT Press, pp. 99-127, 2002. |
| [20] | Mnih, Volodymyr., Kavukcuoglu, Koray., Silver, David., Rusu, Andrei A., Veness, Joel., Bellemare, Marc G., Graves, Alex., Riedmiller, Martin., Fidjeland, Andreas K., Ostrovski, Georg, "Human-level control through deep reinforcement learning," *nature,* vol. 518, no. Nature Publishing Group, pp. 1476-4687, 2015. |

**Appendix:**

Git clone link:

<https://cseegit.essex.ac.uk/22-23-ce901-ce902-su/22-23_CE901-CE902-SU_inamdar_sayed_m_a.git>

readme:



script installation file:



Neural Agent Script:

