F1 Intelligent Racing

Ishaan Sathaye Krishnanshu Gupta

June 9, 2024

1 Project Description

This project aims to develop an AI-based system capable of learning and navigating an optimal racing path for a user-created Formula 1 (F1) track, achieving the fastest possible lap times. The primary objective is to utilize Neural Networks, Reinforcement Learning, and Evolutionary Algorithm techniques to simulate and optimize the performance of a simulated F1 car under realistic driving conditions such as braking, speed, acceleration, and handling. The project features a graphical user interface (GUI) that visually displays the user-created tracks and the simulated cars, displaying the training process and visualizing the AI's learning progress in real-time.

The system employs Evolutionary Algorithms inspired by biological evolution to iteratively enhance the racing strategies of the AI-driven F1 cars. These algorithms enable the cars to evolve by selecting the best-performing individuals from each generation and using their characteristics to produce the next generation, thereby gradually improving lap times and overall performance. In combination with this, two separate agents will be developed using Deep Reinforcement Learning and Neural Networks to compare their algorithmic performances.

The goal of this project is to produce an AI that can consistently find and navigate the fastest racing paths on a variety of randomly generated tracks, with complex corners and track shapes. The output will be the optimized paths for the F1 car and the fastest lap times, which will be visualized through the GUI in the later generations, demonstrating the effectiveness of the AI's learning and optimization processes.

2 Project Design

The core simulation environment is implemented in Python by Tomas Brezina, whose work can be found on GitHub [1]. repo provides a track generation as well as a evolutionary algorithm that mutates agents based on a neural network. The `core.py` file provides classes for managing the track, cars, checkpoints, and collision detections. The `Simulation` class orchestrates the behavior of cars, updates their positions, and handles the checkpoint tracking. Components such as these and messages/menus were already implemented in the original code base to enable developers to save models configurations and render the simulation environment.

The reinforcement learning agent is contained in the `rl_agent.py` file and contains the `RLAgent` class, which is used to train the agent and feed the evolutionary algorithm with the best agents. The agent objects are stored and rendered based on the configuration given in the entry point of our project in the `__main__.py` file.

There are several existing components that we have utilized in our project. We used `pyglet, NumPy, PyTorch, and Matplotlib`. Pyglet was used for rendering the simulation environment so the graphics and also handling the user input events. NumPy was used greatly for handling the mathematical operations on the data such as the car's position, velocity, and acceleration as well as the images of the track. PyTorch was newly introduced to the original code base to implement an RL algorithm called DeepQNetwork. Finally, Matplotlib was used to visualize the training process of the agents.

3 Implementation

User interface was implemented using Pyglet, which is a Python library for developing games and multimedia applications. The user interface was designed to show the agents agents racing on the track and mutating after each generation. Menu options with key presses are also included to zoom in and out of the track, follow cars, and save model configurations.

[1] `https://github.com/TomasBrezina/NeuralNetworkRacing/`

3.1 Reinforcement Learning: Deep Q-Learning

The first part of our project was to implement a reinforcement learning agent that works with the existing evolutionary algorithm. We also chose to a single random track to train the agent on rather than have a new track generated each time. This was done to simplify the training process and to allow us to focus on the agent's behavior rather than the track's complexity.

The reinforcement learning algorithm is implemented in the `rl_agent.py` file, where the RLAgent class serves as the primary component for the reinforcement learning agent. In terms the revolutionary learning algorithm, the environment is the racing sim, track and other cars, then the state would be the car's position and surrounding objects, actions would be the steering, and the reward signal would be factors such as avoiding collisions with the boundary and staying on track. The implementation is based on the Deep Q-Network (DQN) algorithm, a variant of the Q-learning algorithm that utilizes a deep neural network to approximate the Q-value function.

3.1.1 Under the Hood: Deep Q-Network

The RLAgent class initializes the Deep Q-Network (DeepQNetwork) as the policy network, and a separate target network is used for stabilizing the training process. The agent interacts with the simulation environment by receiving state observations, which are preprocessed and fed into the policy network. The network then outputs Q-values for each possible action, and the agent selects the action with the highest Q-value using an epsilon-greedy strategy.

After taking an action and observing the next state and reward, the agent stores this experience in a replay memory buffer. During the training process, batches of experiences are sampled from the replay memory, and the agent updates the policy network weights using the temporal difference error between the predicted Q-values and the target Q-values computed from the target network.

3.1.2 Training Process

The update_policy method in the RLAgent class orchestrates this training process. It samples a batch of experiences from the replay memory, computes the target Q-values using the target network and the observed rewards, and

then updates the policy network weights using back-propagation and a loss function (mean squared error loss).

To ensure stable learning, the target network weights are periodically updated to match the policy network weights using the update_target_network method. Additionally, the reproduce method allows for creating new agents by mutating the weights of the policy network, facilitating the evolutionary process.

3.2 Neural Network: Optimal Lap Time

The Neural Network processed input data from the simulated racing environment and output the appropriate actions for the F1 car to navigate the track. The network architecture is a series of layers, each containing a specified number of neurons, and the input layer receives sensory data from the car, such as speed, position, and proximity to track boundaries, which is generated through waypoints in the center of the track. This data is then propagated through hidden layers, where it is transformed by weighted connections and activation functions, ultimately reaching the output layer, which determines the car's actions, such as rotation, speed, and position.

The neural network utilizes various activation functions, including sigmoid, ReLU, tanh, and leaky ReLU, each of which introduces non-linearity into the model, thereby allowing it to capture more complex patterns in the data. This was necessary to deal with especially tricky corners and track designs. These functions are accompanied by their respective derivatives, which are used during backpropagation to compute gradients. We also utilized the Stochastic Gradient Descent (SGD) and Adam optimizers to update the network's weights based on the computed gradients. SGD is simpler and typically faster for small-scale problems, while Adam is more adaptive and can handle more complex scenarios due to its momentum and learning rate adjustment capabilities.

Batch normalization and dropout techniques were also integrated into the network to enhance training efficiency and robustness. Batch normalization usually helps in stabilizing and speeds up the learning process by normalizing the inputs for each layer. Dropout, on the other hand, prevents over-fitting by randomly dropping units during the training process, and making sure that the network also remains generalized. During training, the neural network goes through a forward pass, where the input data is propagated through the layers to generate predictions. Then the algorithm does backpropagation,

which re-computes the gradients of the loss function with respect to the network's weights. These gradients are then used by the optimizer to update the weights, minimizing the loss function over time.

The network's architecture and parameters, such as the number of layers, activation functions, batch normalization, and dropout rates, are configurable, allowing for experimentation and optimization to achieve the best performance. Additionally, the network includes functions for saving and loading trained models, enabling persistence and speeding up the training process by utilizing weights that were previously performing well.

3.3 Evolutionary Algorithm

The Evolutionary Algorithms are inspired by the principles of natural selection, and these algorithms help evolve a population of neural networks, representing different cars, over multiple generations to improve their racing ability and performance.

The core operations of this algorithm include mutation, crossover, and selection. The mutation process introduces random variations to the network's weights and biases, promoting more exploration of the solution space. Without this, the models may never go down more optimal routes once a singular successful path is found. Crossover combines the weights of two successful parent networks to produce a child network. A portion of the best-performing networks (elites) are directly passed onto the next generation, ensuring that some good solutions are still preserved and not drowned out by poor mutations.

To initiate the evolutionary process, an initial population of neural networks is generated, each with randomly assigned weights. These networks undergo training and evaluation to determine their fitness. The best-performing networks are selected as parents for the next generation, and the mutation and crossover operations are applied to produce new offspring. This process is iterated over multiple generations, with each generation exhibiting improved performance.

The integration of neural networks and evolutionary algorithms enables the AI system to autonomously learn and optimize its racing strategy. The networks continuously improve their ability to navigate the track, adapting to the track conditions and shapes, and refining the cars' racing performance.

4 Testing

One of the core functionalities of the project is the simulation environment, which encompasses track generation, car behavior, and collision detection. Testing this component involved generating a diverse set of tracks which was done in the existing code base. Additionally, edge cases were considered, such as tracks with narrow passages or sharp turns, to assess the system's ability to handle challenging scenarios. The collision detection algorithms were rigorously tested by simulating cars approaching track boundaries at different angles and speeds, ensuring accurate detection and handling of collisions. There were times at which if the speeds and acceleration where increased many of the cars would go off the track's boundaries during RL. Another case we had to handle was cars moving backwards on the track, which was not handled in the original code base.

The integration between the simulation environment and the reinforcement learning agent was also thoroughly tested. This involved simulating scenarios where multiple cars were present on the track, each with its own agent, and ensuring that their interactions, such as passing and collision avoidance, were handled correctly. Edge cases in this area included situations where cars were in close proximity or approaching intersections or narrow passages simultaneously.

5 Analysis

5.1 Discussion of Project

The project aimed to develop an AI-based system capable of learning and optimizing the racing strategy for navigating Formula 1 (F1) tracks. It utilized Neural Networks, Reinforcement Learning, and Evolutionary Algorithms to simulate and optimize the performance of a simulated F1 car under realistic driving conditions, and optimize the agents' performance over multiple generations.

5.2 Challenges

Throughout the project, several challenges were encountered, primarily related to research, platforms, and dealing with existing code. We conducted

extensive research and testing to find suitable simulation software and programs to create realistic driving conditions for testing the algorithms. We explored various simulators, including Microsoft AirSim, CARLA, TORCS, Udacity Self-Driving sim, and Carnegie Mellon University's Learn to Race software. However, these options required Linux and expensive Nvidia GPU access, which we did not have access to within our computers. We also tried to utilize Docker and Amazon EC2 instances with GPU enabled, but they proved to be too expensive to run feasibly for the project's scope.

Thus, we pivoted to looking for existing Python simulation code and after testing and trying out hundreds of different repositories related to F1 and racing simulations, we landed on Tomas Brezina's project using Pyglet and Python. We utilized this repository's code for the simulation and track generation aspects as a foundation, and then adapted and developed custom Neural Networks and Evolutionary Algorithms. Additionally, we introduced a Reinforcement Learning agent to assess its performance alongside the neural network.

5.3 Performance

Both algorithms, Neural Network and Reinforcement Learning, eventually learned and optimized navigation for the track; however, the neural network demonstrated significantly faster convergence and made more effective decisions quickly.

Typically, 100-150 generations were sufficient to achieve well-performing lap times using the Neural Network algorithm. However, running the code continuously led to faster results and lap-times, even up to generation 300-400, albeit at a much slower improvement rate. In contrast, the Reinforcement Learning agent required approximately 300-400 generations to achieve well-performing results on complex tracks due to its slower learning pace and differing design.

5.4 Suggestions for Extensions

Our immediate next steps would be to continue fine-tuning the algorithms and their respective hyperparameter, to allow for faster convergence and to improve generalization. Additionally, we would like to incorporate more realistic simulations through more sophisticated environments that simulate real complexities of F1 racing, such as weather and tire degradation. We

would also like to recreate existing F1 Tracks, and test our current algorithms on them to analyze differences in lap times and racing lines.

Longer-scale future plans would include utilizing high-fidelity racing simulators with advanced physics engines and real-world conditions, ideally suited for F1 races. We would also like to design a multi-objective optimization strategy that could help account for and balance objectives like lap times, overtaking, car degradation, and pit stops, including all aspects that real racers and teams have to keep in mind. Furthermore, a side-development could be obtaining real-time race data such as telemetry, lap times, track layouts, and current team performances to create dynamic and accurate simulations for modeling past races and predicting future ones. Lastly, developing a web-based or mobile platform could facilitate collaborative experiences and creating advanced track generation tools allowing custom constraints, obstacles, and environmental conditions could challenge AI in unique ways. By addressing these extensions and future plans, the project can further enhance its capabilities and contribute to the advancement of AI-driven racing strategies in simulated and real-world environments.

6 Github Repository

F1 Racing Optimal Path:

Link: https://github.com/Krishnanshu-Gupta/F1-Racing-Optimal-Path