F1 Intelligent Racing

Ishaan Sathaye Krishnanshu Gupta

June 9, 2024

1 Project Description

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

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

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.

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

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

6 Github Repository

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