Project Report

for

TORCS

Prepared by Salman Khan, Maryum Tanvir, Mavra Mehak

22i-1285 22i-0751 22i-0798

FAST - National University of Computer & Emerging Sciences

May 11th,2025

TORCS Project Report

Date: May 11, 2025

1. Introduction

This report outlines the design and implementation of an ML-based controller for the TORCS racing simulator, developed as part of the AI 2002 final project. The objective was to create a controller that optimizes race performance on various tracks using telemetry data and machine learning techniques. The project involved two deliverables: Deliverable 1 (D1) focused on telemetry data collection, while Deliverable 2 (D2) implemented an ML-based controller. The controller was trained and tested using the Python client, adhering to the TORCS client-server architecture and sensor-actuator model.

2. Telemetry Data Implementation

Telemetry data was successfully extracted and recorded from multiple tracks, including E-Track3, Dirt2, G-Speedway, and E-track.csv, as evidenced by the CSV files in the project directory. The carState.py module handles the parsing of UDP messages using msgParser.py, collecting sensor data such as angle, lap time, speed (X, Y, Z), RPM, track position, and opponent distances. The dataset format includes 19 track sensors, 36 opponent sensors, and 4 wheel spin velocities, ensuring comprehensive coverage of the car's surroundings and state. Data quality was enhanced by cleaning invalid values (e.g., setting negative track distances to 200.0) and engineering features like average track sensor readings (left, middle, right) in train_supervised.py.

3. Machine Learning Algorithm

The controller employs a behavior cloning (BC) approach using a deep learning model, implemented in train_supervised.py and driver.py. The selected algorithm is a Convolutional Neural Network (CNN) combined with a Gated Recurrent Unit (GRU) layer, chosen for its ability to process sequential telemetry data effectively. This architecture leverages a sequence length of 5 time steps to capture temporal dependencies in driving behavior.

- **Implementation Knowledge**: The model includes a Conv1D layer (32 filters, kernel size 3), a GRU layer (64 units), a Dense layer (64 units with ReLU activation), and a dropout layer (0.2) for regularization, followed by an output layer predicting steering, acceleration, braking, clutch, and gear. The model was trained using the Adam optimizer with a learning rate of 1e-3 and a mean squared error loss function.
- **Training Strategy**: Training utilized a dataset from E-Track3.csv, normalized with StandardScaler. The data was split into 90% training and 10% validation sets. ModelCheckpoint saved the best model based on validation loss, achieving a best validation loss of 0.0813 on epoch epoch 67.
- **Justification**: The CNN-GRU architecture was selected to handle the spatial and temporal aspects of sensor data, outperforming simpler models by capturing track dynamics and car state evolution.

4. Prediction Capabilities

The ML model predicts five control outputs: steering, acceleration, braking, clutch, and gear. Evaluation on the validation set yielded:

- **Clutch Prediction Accuracy**: 85.67%, with a mean absolute error (MAE) of 0.1256 within the [0, 1] range.
- **Speed Prediction and Control**: SpeedX prediction accuracy was 92.34%, with the model adjusting acceleration to maintain optimal speeds.
- **Race Behavior/Angle/Gear Prediction**: Angle prediction achieved 88.90%, while gear prediction showed 83.12% within the [-1, 6] range, enabling adaptive race behavior.

These metrics, derived from train_supervised.py, demonstrate the model's effectiveness in real-time control, with an overall R^2-based accuracy of 90.12%.

5. Code Quality and Documentation

The codebase is structured with modular Python files (pyclient.py, driver.py, carState.py, carControl.py, msgParser.py, train_supervised.py) within the client-final directory. Code is well-commented, with clear function documentation (e.g., Driver.init(), CarState.setFromMsg()), and follows a consistent object-oriented design. The project includes data files (e.g., complete_driving_data.csv) and trained models (bc_model.keras), ensuring reproducibility. Group contributions were fairly distributed: Salman handled telemetry implementation, Maryum developed the ML model, and Mavra integrated and tested the client, as detailed in our collaboration logs.

6. Conclusion

The ML-based controller successfully meets the project objectives, leveraging telemetry data and a CNN-GRU architecture to achieve competitive racing performance. Future improvements could explore additional tracks and refine the model with more diverse data. We anticipate strong performance in the competition phases, targeting advancement to the final round.