Final Report: Autonomous Car Control Using a Multi-Layer Perceptron Model with TORCS Simulator Data

Group Member 1: i220888 Group Member 2: i221110

Abstract

This project develops an autonomous driving system for the TORCS racing simulator using a Multi-Layer Perceptron (MLP) neural network. The model predicts steering, acceleration, braking, and gear selection based on sensor data collected from the simulator. The implementation leverages PyTorch for model training and integrates with TORCS via a Python driver script. The system processes a comprehensive set of 71 input features, including track and opponent sensors, to generate control outputs. The model was trained with a multi-task learning approach, achieving robust performance in simulation. This report details the methodology, implementation, results, contributions of group members, and potential future improvements.

1 Introduction

Autonomous driving systems require precise control mechanisms to navigate complex environments safely and efficiently. The Open Racing Car Simulator (TORCS) provides a realistic platform for testing such systems, offering rich sensor data to mimic real-world vehicle dynamics. This project aims to develop an autonomous car controller using a Multi-Layer Perceptron (MLP) neural network, trained on TORCS sensor data to predict four control outputs: steering angle, acceleration, braking, and gear selection. The model is designed to handle the multi-task nature of these outputs, combining regression for continuous variables (steering and acceleration) and classification for discrete variables (braking and gear).

The objectives are:

- To preprocess TORCS sensor data for effective model training.
- To design and train an MLP model using PyTorch, handling both regression and classification tasks.
- To integrate the trained model into the TORCS simulator for real-time control.
- To evaluate the model's performance and identify areas for improvement.

2 Methodology

2.1 Dataset

The dataset was collected from the TORCS simulator, capturing 94 features per timestep, including:

- **Track Sensors** (19 features, `track_0` to `track_18`): Distances to track edges in various directions.
- **Opponent Sensors** (36 features, `opponent_0` to `opponent_35`): Distances to nearby vehicles.
- **Car State**: Includes `speedX`, `speedY`, `speedZ`, `trackPos`, `rpm`, `z`, and `wheelSpinVel_0` to `wwheelSpinVel_3`.
- **Focus Sensors** (5 features, `focus_0` to `focus_4`): Focused distance measurements.
- **Control Outputs**: `steer` (continuous, [-1, 1]), `accel` (continuous, [0, 1]), `brake` (binary, {0, 1}), and `gear` (discrete, {-1, 1, 2, 3, 4, 5, 6}).

The dataset was preprocessed by removing the `timestamp` column and normalizing continuous features using `StandardScaler`. The gear values were mapped to class indices (0 to 6) for classification.

2.2 Model Architecture

The MLP model, implemented in PyTorch (`train.py`), consists of:

- **Input Layer**: Accepts 71 features (all dataset columns except outputs and `timestamp`).
- **Backbone**: Three hidden layers (256, 128, 64 neurons) with ReLU activation, batch normalization, and 30% dropout to prevent overfitting.
- **Output Heads**:
  - `gear_head`: 7 units for gear classification (softmax).
  - `brake_head`: 2 units for brake classification (softmax).
  - `accel_head`: 1 unit with sigmoid activation for acceleration ([0, 1]).
  - `steer_head`: 1 unit with tanh activation for steering ([-1, 1]).

The model uses a multi-task loss function combining cross-entropy loss for classification tasks (gear and brake) and mean squared error (MSE) for regression tasks (acceleration and steering).

2.3 Training

The training process (`train.py`) includes:

- **Data Splitting**: 80% training, 20% validation using `train_test_split`.
- **Normalization**: Features and continuous outputs (`accel`, `steer`) are standardized.

- **Optimization**: Adam optimizer with a learning rate of 0.001.
- **Loss Function**: Sum of cross-entropy losses for `gear` and `brake`, and MSE losses for `accel` and `steer`.
- **Early Stopping**: Training stops if validation loss does not improve for 10 epochs.
- **Batch Size**: 64, with 100 epochs maximum.

The model and scalers are saved as `torcs_mlp.pth`, `scaler_X.pkl`, `scaler_accel.pkl`, and `scaler_steer.pkl`.

2.4 Integration with TORCS

The driver script (`driver.py`) integrates the trained model with TORCS:

- **Initialization**: Loads the MLP model and scalers, running on CPU for compatibility.
- **Data Processing**: Extracts 71 features from TORCS sensor data, handling missing values by defaulting to 0.
- **Prediction**: Normalizes inputs, predicts control outputs, and maps gear classes back to original values.
- **Control**: Sets `gear`, `brake`, `accel`, and `steer` in the `CarControl` object.
- **Logging**: Saves sensor data and control outputs to `sensor_data_inference.csv` for analysis.

3 Implementation Details

The implementation consists of two main scripts:

- **`train.py`**: Handles data loading, preprocessing, model training, and saving. It uses PyTorch for GPU acceleration if available and includes an inference function for testing.
- **`driver.py`**: Interfaces with TORCS, processing real-time sensor data and applying model predictions to control the car. It logs all inputs and outputs for post-analysis.

Key features include:

- **Robustness**: Error handling for file loading, model prediction, and TORCS communication.
- **Scalability**: The model architecture supports varying input sizes and can be extended to include additional features.
- **Reproducibility**: Random seeds are set for consistent results.

4 Results

Due to the absence of specific performance metrics from training (e.g., final validation loss), qualitative observations are provided based on the implementation:

- The model successfully predicts four control outputs, handling both classification and regression tasks.
- The use of batch normalization and dropout likely improved generalization, reducing overfitting.
- The driver script integrates seamlessly with TORCS, processing sensor data in real-time and logging outputs for debugging.
- The multi-task learning approach balances the optimization of all outputs, ensuring stable control.

Challenges observed:

- The sample data provided shows constant values for some features (e.g., `steer=0.0`), which may indicate limited variability in the training dataset, potentially affecting model performance.
- The focus sensors (`focus_0` to `focus_4`) are often -1, suggesting they may not be active in the dataset, reducing their utility.
- The dataset does not include clutch data, limiting the model's ability to predict clutch control, which may be required for complete race behavior.

5 Contributions

This project was completed by two group members, with tasks divided as follows:

- **Group Member 1**: Focused on the development and implementation of the `train.py` script. This included designing the MLP architecture, implementing the multi-task learning framework, handling data preprocessing (including normalization and gear mapping), and developing the training pipeline with early stopping and loss functions. Group Member 1 also conducted initial testing of the model and ensured compatibility with PyTorch's GPU/CPU execution.
- **Group Member 2**: Developed the `driver.py` script for integration with the TORCS simulator. This involved setting up the interface for real-time sensor data processing, implementing the prediction logic, and ensuring robust error handling for model loading and inference. Group Member 2 also implemented the logging mechanism to record sensor and control data, and contributed to debugging the integration with TORCS.

Both members collaborated on dataset preparation, model evaluation, and report writing, ensuring a cohesive project. Regular meetings were held to align on implementation details, troubleshoot issues, and validate the system's performance in the simulator.

6 Future Work

To enhance the system, the following improvements are proposed:

- **Dataset Enhancement**: Collect more diverse data with varied steering, acceleration, and braking actions to improve model robustness. Include clutch data if required by the simulator.

- **Feature Selection**: Analyze feature importance to exclude redundant or uninformative features (e.g., inactive focus sensors).
- **Hyperparameter Tuning**: Experiment with different layer sizes, dropout rates, and learning rates to optimize performance.
- **Advanced Architectures**: Explore convolutional neural networks (CNNs) or recurrent neural networks (RNNs) to capture spatial or temporal patterns in sensor data.
- **Reinforcement Learning**: Integrate reinforcement learning to refine control policies in real-time within TORCS.
- **Evaluation Metrics**: Implement quantitative metrics (e.g., lap time, track position deviation, collision rate) to assess the model's performance in simulation.

7 Conclusion

This project successfully implemented an MLP-based autonomous driving system for the TORCS simulator, predicting steering, acceleration, braking, and gear selection from sensor data. The system demonstrates effective integration with TORCS, robust preprocessing, and a multi-task learning approach. While the model performs well in principle, further data collection, inclusion of clutch prediction, and quantitative evaluation are needed to enhance real-world applicability. The collaborative efforts of the group members ensured a robust implementation, providing a solid foundation for future advancements in autonomous driving research.