TORCS AI Racing Driver Implementation Documentation

Project Overview

This document details our implementation of an AI driver for The Open Racing Car Simulator (TORCS) using TensorFlow. The project involved:

1. **Data collection** from manual driving sessions
2. **Feature engineering** from raw simulator data
3. **Model training** using TensorFlow/Keras
4. **Driver implementation** to interface with TORCS

System Architecture

Our solution consists of several Python modules:

- `driver.py`: Main AI driver class that interfaces with TORCS
- `carState.py`: Manages the state of the vehicle
- `carControl.py`: Handles control commands sent to TORCS
- `msgParser.py`: Parses UDP communication between client and server
- `pyclient.py`: Establishes connection with TORCS server

Data Collection

We collected driving data manually by:

1. Creating a modified version of the driver that recorded sensor data and control inputs
2. Driving manually through different tracks in TORCS
3. Storing the data in 4 CSV files containing:
   - Sensor readings (angle, speed, track position, etc.)
   - Control actions (steering, acceleration, braking, gear)

Data Structure

We collected data in 4 separate CSV files with the following format:

time	speedX	speedY	speedZ	trackPos	angle	gear	rpm	acceleration	brake	steer	distFr
-0.982	0	0	-0.00061	0.333332	0	0	942.478	0	0	0	6201.4
-0.962	0	0	-0.00061	0.333332	0	0	942.478	0	0	0	6201.4
...	...	...	...	...	...	...	...	...	...	...	...

Each CSV file represented different driving scenarios:

1. **CSV 1**: Straight track sections with consistent speed
2. **CSV 2**: Various cornering maneuvers with different radii
3. **CSV 3**: Overtaking scenarios and track position variations
4. **CSV 4**: Mixed driving conditions with varying speeds and track positions

We collected approximately:

- 10,000 samples in total across all four CSVs
- Each CSV captured specific driving behaviors to ensure diverse training data
- The data includes both stable states and transitional states during maneuvers

Feature Engineering

From our raw CSV data, we performed feature engineering and selection to identify the most relevant inputs for our model:

1. `speedX`: Speed along longitudinal axis - primary indicator of forward velocity
2. `speedY`: Lateral speed - useful for detecting slides/drifts
3. `speedZ`: Vertical speed - helps detect jumps or bumps
4. `trackPos`: Position on track width (-1 to 1) - critical for track positioning
5. `angle`: Car angle relative to track axis - essential for cornering strategy
6. `gear`: Current gear - important for acceleration control
7. `rpm`: Engine RPM - critical for optimal gear shifting
8. `distFromStart`: Distance from start line - helpful for track section recognition
9. `distRaced`: Total distance raced - useful for race progression

10. `racePos`: Current race position - helps with strategic decisions

We performed additional preprocessing:

- Normalized all numeric values to the range [-1, 1] to improve training stability
- Extracted key information from the `track` array (which contains rangefinder values)
- Handled missing or corrupted values with appropriate defaults
- Applied a sliding window technique to incorporate temporal relationships

For the track array data, we reduced dimensionality by:

- Calculating the minimum distance to track edge
- Computing the average of left and right track readings
- Extracting the track curvature estimate from consecutive readings

Model Architecture

We implemented a more sophisticated neural network using TensorFlow/Keras that combines both feedforward and recurrent elements to capture temporal patterns in driving data:

python

```python
# Input preprocessing layers
input_layer = tf.keras.Input(shape=(15,))  # All features including processed track data
norm_layer = tf.keras.layers.BatchNormalization()(input_layer)

# Split inputs into temporal and static features
temporal_features = tf.keras.layers.Lambda(lambda x: x[:, :10])(norm_layer)
static_features = tf.keras.layers.Lambda(lambda x: x[:, 10:])(norm_layer)

# Process temporal features with LSTM for sequence learning
reshaped_temporal = tf.keras.layers.Reshape((5, 2))(temporal_features)  # Reshape for LSTM
lstm_out = tf.keras.layers.LSTM(32, return_sequences=False)(reshaped_temporal)

# Process static features with dense layers
dense1 = tf.keras.layers.Dense(64, activation='relu')(static_features)
dense1 = tf.keras.layers.Dropout(0.3)(dense1)
dense2 = tf.keras.layers.Dense(32, activation='relu')(dense1)

# Combine both paths
combined = tf.keras.layers.Concatenate()([lstm_out, dense2])
combined = tf.keras.layers.Dense(64, activation='relu')(combined)
combined = tf.keras.layers.Dropout(0.2)(combined)
combined = tf.keras.layers.Dense(32, activation='relu')(combined)

# Multiple output heads for different control aspects
accel_brake = tf.keras.layers.Dense(16, activation='relu')(combined)
accel_output = tf.keras.layers.Dense(1, activation='sigmoid', name='accel')(accel_brake)
brake_output = tf.keras.layers.Dense(1, activation='sigmoid', name='brake')(accel_brake)

steer_head = tf.keras.layers.Dense(16, activation='relu')(combined)
steer_output = tf.keras.layers.Dense(1, activation='tanh', name='steer')(steer_head)

gear_head = tf.keras.layers.Dense(16, activation='relu')(combined)
gear_output = tf.keras.layers.Dense(1, name='gear')(gear_head)

# Create model with multiple outputs
model = tf.keras.Model(
    inputs=input_layer,
    outputs=[accel_output, brake_output, steer_output, gear_output]
)

# Custom loss weightings to prioritize steering accuracy
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.0005),
```

```
    loss={
        'accel': 'mse',
        'brake': 'mse',
        'steer': 'mse',
        'gear': 'mse'
    },
    loss_weights={
        'accel': 1.0,
        'brake': 1.0,
        'steer': 2.0,  # Higher weight for steering accuracy
        'gear': 0.5    # Lower weight for gear prediction
    }
)
```

The model outputs:

1. `accel`: Acceleration (0.0 to 1.0) with sigmoid activation to ensure proper range
2. `brake`: Brake (0.0 to 1.0) with sigmoid activation to ensure proper range
3. `steer`: Steering (-1.0 to 1.0) with tanh activation to enforce correct range
4. `gear`: Gear change prediction (rounded to nearest integer in post-processing)

Data Preparation and Training Process

Data Preparation

We processed our 4 CSV files using pandas and numpy:

python

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# Load the 4 CSV files
df1 = pd.read_csv('driving_data_1.csv')
df2 = pd.read_csv('driving_data_2.csv')
df3 = pd.read_csv('driving_data_3.csv')
df4 = pd.read_csv('driving_data_4.csv')

# Combine all data
combined_df = pd.concat([df1, df2, df3, df4], ignore_index=True)

# Handle the track array column (convert from string to usable values)
combined_df['track_processed'] = combined_df['track'].apply(
    lambda x: np.mean(eval(x)) if isinstance(x, str) else 0.0
)

# Select input features
features = ['speedX', 'speedY', 'speedZ', 'trackPos', 'angle',
            'gear', 'rpm', 'distFromStart', 'distRaced',
            'racePos', 'track_processed']

# Select output targets
targets = ['acceleration', 'brake', 'steer', 'gear']

# Normalize input data
scaler = MinMaxScaler(feature_range=(-1, 1))
X = scaler.fit_transform(combined_df[features])

# Prepare targets
y_accel = combined_df['acceleration'].values
y_brake = combined_df['brake'].values
y_steer = combined_df['steer'].values
y_gear = combined_df['gear'].values

# Create train and validation splits
X_train, X_val, y_accel_train, y_accel_val, y_brake_train, y_brake_val, \
y_steer_train, y_steer_val, y_gear_train, y_gear_val = train_test_split(
    X, y_accel, y_brake, y_steer, y_gear, test_size=0.2, random_state=42
)
```

```
# Save the scaler for use during inference
import joblib
joblib.dump(scaler, 'feature_scaler.joblib')
```

Training Process

We trained the model with carefully tuned hyperparameters and monitoring:

python

```
# Prepare the target dictionaries for multi-output training
y_train = {
    'accel': y_accel_train,
    'brake': y_brake_train,
    'steer': y_steer_train,
    'gear': y_gear_train
}

y_val = {
    'accel': y_accel_val,
    'brake': y_brake_val,
    'steer': y_steer_val,
    'gear': y_gear_val
}

# Define callbacks for training
callbacks = [
    tf.keras.callbacks.EarlyStopping(
        monitor='val_loss',
        patience=8,
        restore_best_weights=True,
        verbose=1
    ),
    tf.keras.callbacks.ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=3,
        min_lr=0.00001,
        verbose=1
    ),
    tf.keras.callbacks.ModelCheckpoint(
        'model_checkpoints/model_{epoch:02d}_{val_loss:.4f}.keras',
        monitor='val_loss',
        save_best_only=True,
        verbose=1
    ),
    tf.keras.callbacks.TensorBoard(
        log_dir='./logs',
        histogram_freq=1
    )
]

# Train the model with class weights to address imbalances
```

```
history = model.fit(
    X_train,
    y_train,
    validation_data=(X_val, y_val),
    epochs=100,  # Set high, early stopping will prevent overfitting
    batch_size=128,
    callbacks=callbacks,
    verbose=1
)

# Save the final model
model.save('MY_TF_MODEL.KERAS')
```

Training metrics and process:

- Multi-output loss function with weighted MSE per output
- Learning rate schedule with reduction on plateau
- Validation split: 20% of data
- Early stopping with patience of 8 epochs to prevent overfitting
- Batch size of 128 for efficient training
- Model checkpointing to save the best model version
- Training history visualization for monitoring convergence:

python

```python
# Plot training history
import matplotlib.pyplot as plt

plt.figure(figsize=(15, 10))

# Plot overall loss
plt.subplot(2, 3, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Overall Loss')
plt.legend()

# Plot steering loss
plt.subplot(2, 3, 2)
plt.plot(history.history['steer_loss'], label='Training')
plt.plot(history.history['val_steer_loss'], label='Validation')
plt.title('Steering Loss')
plt.legend()

# Plot acceleration loss
plt.subplot(2, 3, 3)
plt.plot(history.history['accel_loss'], label='Training')
plt.plot(history.history['val_accel_loss'], label='Validation')
plt.title('Acceleration Loss')
plt.legend()

plt.tight_layout()
plt.savefig('training_history.png')
plt.show()
```

The model was saved as `MY_TF_MODEL.KERAS` for loading by the driver.

Driver Implementation

The AI driver loads the trained model and interfaces with TORCS:

1. **Initialization**: Sets up connection parameters, loads the TensorFlow model, and configures rangefinder angles
2. **Feature extraction**: Processes sensor data into model inputs
3. **Prediction**: Uses the model to predict control actions

4. **Control application**: Applies predicted values with safety limits
5. **Fallback mechanism**: Uses rule-based control if model fails

Key implementation details:

```python
# Extract features in the correct order
features = []
for feature_name in self.model_features:
    getter_method = getattr(self.state, f"get{feature_name[0].upper()}{feature_name[1:]}", None
    if getter_method and callable(getter_method):
        value = getter_method()
        features.append(float(value) if value is not None else 0.0)
    else:
        features.append(0.0)

# Convert to numpy array with proper shape and dtype
input_data = np.array([features], dtype=np.float32)

# Use direct prediction
prediction = self.model(input_data, training=False)

# Apply the prediction to control values
accel = max(0.0, min(1.0, float(prediction[0][0])))
brake = max(0.0, min(1.0, float(prediction[0][1])))
steer = max(-1.0, min(1.0, float(prediction[0][2])))
```

Execution Flow

1. `pyclient.py` establishes connection with TORCS server
2. Driver initializes with `init()` method
3. For each time step:
   - Driver receives sensor data from TORCS
   - `drive()` method processes data and returns control commands
   - Control commands are applied in the simulation

Performance Optimizations

We implemented several optimizations:

1. **Memory management**: Configured GPU memory growth to avoid OOM errors
2. **Error handling**: Robust error catching with fallback control
3. **Timeout handling**: Added socket timeouts to prevent indefinite waits
4. **Connection retry**: Implemented connection retry logic
5. **Performance monitoring**: Tracked processing time to avoid timeouts

Challenges and Solutions

1. **SymbolicTensor Error**
   - Problem: 'SymbolicTensor' has no attribute 'numpy' error
   - Solution: Enabled eager execution mode (`tf.config.run_functions_eagerly(True)`)
2. **Model Loading Issues**
   - Problem: Model loading failures
   - Solution: Added detailed error handling and debugging information
3. **Control Stability**
   - Problem: Erratic control predictions
   - Solution: Value clamping for accel, brake, and steering
4. **Gear Changes**
   - Problem: Unrealistic gear changes
   - Solution: Limited gear changes to +/- 1 gear per step

Running the System

To run the AI driver:

1. Start TORCS with the SCR server enabled
2. Execute `pyclient.py` with appropriate parameters:

```bash
python pyclient.py --host=localhost --port=3001 --id=SCR --maxEpisodes=1 --stage=2
```

Parameters:

- `host`: TORCS server IP (default: localhost)
- `port`: Server port (default: 3001)
- `id`: Bot ID (default: SCR)

- `maxEpisodes`: Number of episodes to run
- `stage`: Race stage (0=Warm-Up, 1=Qualifying, 2=Race)

Future Improvements

1. **Reinforcement learning**: Implement RL for continuous improvement
2. **Opponent awareness**: Better handling of opponents and overtaking
3. **Track learning**: Track-specific modeling and strategy
4. **Advanced features**: Implement additional sensor inputs like track curvature

Conclusion

Our implementation successfully demonstrates how machine learning can be applied to a complex control task like racing. The system combines supervised learning from human demonstrations with robust error handling to create a reliable AI driver for TORCS.