Autonomous Racing Controller for TORCS Using Machine Learning

Group Members:
Mahad Rehman 22i-0792
Muhammad Hashim 22i-0976
Muhammad Masab 22i-1004

Abstract

In this project, we train a feedforward neural network using telemetry data from a racing simulator (TORCS) to predict driving actions — namely throttle, brake, steering, and gear commands. The model takes raw telemetry features such as speed, angle, and track position as input, and learns to predict optimal control outputs using supervised learning.

0 Contributions:

Mahad Rehman Durrani (22i-0792)

- Implemented Car Driving and Model Car Driving Logics
- Collectively worked on the model
- Data Collection

Muhammad Hashim (22i-0976)

- Designed the initial plan of Model
- Worked on Model Code
- Collectively worked on the model

Muhammad Masab (22i-1004)

- Primary Data Collection
- Collectively worked on the model
- Data Preprocessing

1 Introduction

Our goal was to simulate intelligent driving behavior by predicting driver actions based on sensory input from the game. The dataset was collected from various racing tracks and cars, capturing player behavior. A regression-based neural network is trained on these logs to imitate human-like driving.

2 Data Preprocessing

We use multiple CSV files collected from game telemetry. These logs contain real-time values of several parameters like vehicle speed, damage, fuel, track position, and more.

```
CSV_FILES = [
        'telemetry_log_unknown.csv',
        'telemetry_etrack-corolla.csv', ... ] df_list =
[pd.read_csv(f) for f in CSV_FILES] df =
pd.concat(df_list, ignore_index=True)
```

Listing 1: Combining CSV Files

We separate input features and output labels:

- **Input Features:** 75 columns including speed, angle, damage, etc.
- **Labels:** 4 output commands — acceleration, braking, steering, gear.

3 Data Scaling

Before training, we standardize our input features using Z-score normalization:

$$X_{\text{scaled}} = \frac{x - \mu}{\sigma}$$

```
scaler = StandardScaler().fit(X_train) X_train =
scaler.transform(X_train)
X_val = scaler.transform(X_val)
```

Listing 2: Standardizing Features

This step ensures all features contribute equally to model learning.

4 PyTorch Dataset and Dataloader

We convert NumPy arrays into PyTorch Datasets to efficiently load batches.

```
class TelemetryDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.from_numpy(X) self.y =
        torch.from_numpy(y)
    def __len__(self):
        return len(self.X)
    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

Listing 3: Custom Dataset Class

5 Model Architecture

We use a fully-connected feedforward neural network:

- Input Layer: 75 features
- Hidden Layer 1: 256 neurons (ReLU)
- Hidden Layer 2: 128 neurons (ReLU)
- Output Layer: 4 units (accel, brake, steer, gear)

```
class Net(nn.Module):
    def __init__(self, in_dim, out_dim):
        super().__init__() self.net = nn.Sequential(
        nn.Linear(in_dim, 256), nn.ReLU(),
        nn.Linear(256, 128), nn.ReLU(), nn.Linear(128,
        out_dim)
        )

def forward(self, x):
    return self.net(x)
```

Listing 4: Neural Network Architecture

6 Loss Function and Optimizer

We use Mean Squared Error (MSE) for regression and the Adam optimizer for weight updates.

```
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

7 Training Loop

We train the model for 20 epochs, measuring training and validation loss in each round.

```
for epoch in range(1, EPOCHS+1): model.train() for xb,
    yb in train_loader: xb, yb = xb.to(DEVICE),
    yb.to(DEVICE) optimizer.zero_grad() preds =
    model(xb) loss = criterion(preds, yb)
    loss.backward() optimizer.step()
```

Listing 5: Training Loop

We evaluate the model on validation data and save the best model (lowest val loss):

```
if val_loss < best_val_loss:
    torch.save(model.state_dict(), MODEL_OUT)
    best_val_loss = val_loss
```

8 Why 20 Epochs?

An epoch represents one full pass over the training data. Training for 20 epochs was empirically chosen to balance learning without overfitting. More epochs might reduce training loss but can harm generalization.

9 Learning Rate (LR)

We use a learning rate of 1×10^{-3} (or 0.001). This controls the size of the step we take while updating weights. Too large may diverge, too small may be slow.

10 Train/Test Split Justification

We use 90% of the data for training and 10% for validation. This allows sufficient data for learning while retaining some for unbiased model evaluation. Using too much test data would reduce training efficiency.

11 Conclusion

We successfully trained a deep regression model to predict driving commands using PyTorch. This project helped us understand the full pipeline of real-world machine learning — from data preprocessing and scaling to model building, training, and evaluation.