TORCS Gaming AI Agent: Q-Learning-Based Autonomous Racecar Control

Kalbe Raza (i220794-B)
Abdullah Mehmood (i220978-B)
Aneeq Malik (i221167-B)
Artificial Intelligence Course Project

Abstract

This report presents the "TORCS Gaming AI Agent," a Q-learning-based system developed for an Artificial Intelligence course to control an autonomous racecar in The Open Racing Car Simulator (TORCS). The system comprises two components: `MultiTrackQLearningTrainer`, which trains a Q-table offline using track data from CSV files, and `QLearningDriver`, which deploys the Q-table in real-time to navigate TORCS tracks. The system optimizes racing performance by balancing speed, track position, and safety, using Q-learning for decision-making and rule-based logic for edge cases like getting stuck or excessive swerving. Detailed explanations of each function, their integration, and the rationale for choosing Q-learning over neural networks are provided, along with a graph illustrating training performance.

1 Overview

This report details the "TORCS Gaming AI Agent," a project developed for an Artificial Intelligence course, focusing on autonomous racecar control in TORCS using Q-learning. The system consists of two components:

- **MultiTrackQLearningTrainer**: Trains a Q-table offline using track data from CSV files (`GL.csv`, `DL.csv`, `TL.csv`) to learn optimal driving actions across multiple TORCS tracks.
- **QLearningDriver**: Deploys the trained Q-table in real-time to control a racecar in TORCS, using rule-based control as a fallback for robustness.

The system optimizes racing performance by balancing speed, track position, and safety, leveraging Q-learning for decision-making and rule-based logic for edge cases like getting stuck or excessive swerving. The report includes detailed explanations of each function in both classes, their integration, and the rationale for choosing Q-learning over neural networks for this AI-driven project.

2 MultiTrackQLearningTrainer: Function Explanations

The `MultiTrackQLearningTrainer` class trains a Q-table using Q-learning on offline track data from TORCS, employing multithreading for efficiency. Below is a detailed explanation of each function, with naive analogies for clarity.

2.1 __init__(self, track_files=None, q_table_file='multi_track_q_table.pkl', batch_size=20000)

Purpose: Initializes the trainer with parameters and data structures.
Detailed Explanation:

- **Parameters**:
  - `track_files`: List of CSV files (defaults to `['GL1.csv', 'DL1.csv', 'TL1.csv']`).
  - `q_table_file`: File for saving/loading the Q-table (`'multi_track_q_table.pkl'`).
  - `batch_size`: Number of data points per episode (20,000).
- **Attributes**:
  - `q_table`: `defaultdict` with 9 actions (3 steering $\times$ 3 accel/brake).
  - `alpha` (0.1): Learning rate for Q-table updates.
  - `gamma` (0.95): Discount factor for future rewards.
  - `epsilon` (0.8): Exploration rate, decaying to 0.01 with 0.995 decay.
  - `state_bins`: Bins for discretizing state variables.
  - `actions`: 9 action combinations (steering: $[-0.5, 0.0, 0.5]$, accel/brake: $[(1.0, 0.0), (0.5, 0.0), ($
  - `q_table_lock`: Thread-safe lock for Q-table updates.
  - `state_visit_counts`: Tracks state visits for adaptive learning rate.
  - `track_data`: Stores loaded track DataFrames.
- **Functionality**: Initializes bins, actions, and loads the Q-table if available.

Naive Analogy: A kid learning to race toy cars in TORCS gets a notebook (`q_table`) to record good moves. They're told to learn slowly (`alpha`), think about future laps (`gamma`), and try new moves often (`epsilon`). The track is divided into zones (`state_bins`), and they have 9 ways to drive (`actions`). A lock ensures only one kid writes in the notebook at a time.

2.2 define_state_bins(self)

Purpose: Defines bins to discretize continuous state variables.

- Returns a dictionary with bins for:
  - `SpeedX`: -50 to 300 (25 bins, $\sim$14.2 units).
  - `TrackPosition`: -2.0 to 2.0 (10 bins, $\sim$0.4 units).

– `Angle`: $-\pi$ to π (12 bins, ∼0.52 radians).

– `Track`$_9: 0 to 250 (8 bins,$∼`31.25 units).`

`-- Damage:  0 to 10,000 (6 bins, ∼1666.67 units).`

- Uses `np.linspace` for evenly spaced bins.

Naive Analogy: The kid divides the TORCS track into a grid, like a board game, grouping speeds into "slow," "medium," "fast," etc., so they can easily note what to do in each grid square.

2.3 define_actions(self)

Purpose: Defines the action space.
Detailed Explanation:

- Creates 9 actions combining 3 steering options (−0.5, 0.0, 0.5) with 3 accel/brake options (full, half, none).

Naive Analogy: The kid's toy car in TORCS has buttons for turning left, straight, or right, and pushing hard, lightly, or not at all, giving 9 possible moves.

2.4 discretize_state(self, state)

Purpose: Converts continuous state to discrete state tuple.
Detailed Explanation:

- Maps each state variable to a bin index using `np.digitize`, ensuring indices stay within bounds.

- Returns a tuple (e.g., (8, 4, 6, 2, 0)).

Naive Analogy: The kid looks at their car's speed and position in TORCS, then labels it as "fast, near center, slight left turn" to check their notebook.

2.5 compute_reward(self, state, next$_state$)

Purpose: Calculates reward for state transitions.
Detailed Explanation:

- **Rewards**: Speed (+0.1/unit), acceleration (+5.0), centering (+2.0), progress (+10.0/unit).

- **Penalties**: Damage (−10.0/unit), off-track (−5.0 if |`TrackPos`| > 1.0, −15.0 if > 1.5), backward speed (−20.0), angle (−2.0/radian).

Naive Analogy: The kid in TORCS gets points for going fast, speeding up, staying on the path, and moving forward, but loses points for crashing, swerving, or going backward.

2.6 choose$_{a}ction(self, state)$

Purpose: Selects an action using epsilon-greedy policy.
Detailed Explanation:

- With probability `epsilon` (0.8), picks a random action; otherwise, chooses the action with the highest Q-value.

Naive Analogy: The kid flips a coin in TORCS: 80% chance they try a random move, 20% chance they pick the best move from their notebook.

2.7 update$_{qt}able(self, state, action, reward, next_{s}tate)$

Purpose: Updates the Q-table using Q-learning.
Detailed Explanation:

- Discretizes states, uses thread-safe lock, adjusts learning rate based on state visits, and applies:

$$Q(s, a) = Q(s, a) + \alpha \cdot (\text{reward} + \gamma \cdot \max(Q(s', a')) - Q(s, a))$$

Naive Analogy: The kid in TORCS updates their notebook after a move, blending new points (`reward`) with future possibilities (`max(Q(s',a'))`), writing carefully to avoid mistakes.

2.8 load$_{qt}able(self)$

Purpose: Loads a saved Q-table.
Detailed Explanation:

- Loads `q`$_{t}able_{f}ileintoa$`defaultdict`$, loggingthenumberofstates$. **Naive Analogy**: The kid loads their old TORCS notebook to recall past lessons.

2.9 save$_{qt}able(self)$

Purpose: Saves the Q-table to a file.
Detailed Explanation:

- Converts `q`$_{t}abletoadictionary, savesitwith$`pickle`$, usingathread-safelock$. **Naive Analogy**: The kid saves their TORCS notebook for future races, ensuring no one else writes in it during copying.

2.10 analyze$_{s}tates(self, df, track_{n}ame = None)$

Purpose: Analyzes state distribution in track data.
Detailed Explanation:

- Counts occurrences in `SpeedX` and `TrackPosition` bins to log coverage (e.g., how many high-speed samples).

Naive Analogy: The kid checks how often they drove fast or near the edge in TORCS to ensure they practiced all scenarios.

2.11 load$_{t}rack_{t}hread(self, track_{f}ile, results_{q}ueue)$

Purpose: Loads a track's CSV data in a thread.
Detailed Explanation:

- Reads CSV, analyzes state distribution, and puts the DataFrame in `results`$_{q}ueue$. **Naive Analogy**: A helper reads a TORCS track map, checks its details, and puts it in a shared box for the kid.

2.12 load$_{t}rack_{d}ata(self)$

Purpose: Coordinates parallel loading of track data.
Detailed Explanation:

- Starts threads for each track, collects DataFrames from `results`$_{q}ueue, andstoresthemin$ **Naive Analogy**: Multiple helpers grab TORCS track maps simultaneously, organizing them in a folder for the kid.

2.13 train$_{o}n_{t}rack(self, track_{n}ame, episode, episodes)$

Purpose: Trains on a track for one episode.
Detailed Explanation:

- Samples `batch`$_{s}izerows, addshigh-speedsamples, processesstatepairstochooseactions, computerewar$ $table$. **Naive Analogy**: The kid practices one lap in TORCS, trying moves, earning points, and updating their notebook, focusing extra on fast moments.

2.14 evaluate$_{o}n_{t}rack(self, track_{n}ame, episode)$

Purpose: Evaluates performance on a track (not used in training).
Detailed Explanation:

- Tests on validation data, computing average reward without updating the Q-table.

Naive Analogy: The kid tests their skills in TORCS by driving without writing in the notebook, checking how many points they get.

2.15 train$_{t}rack_{t}hread(self, track_{n}ame, episode, episodes, results_{q}ueue)$

Purpose: Trains on a track in a thread.
Detailed Explanation:

- Calls `train`$_{o}n_{t}rack, putstotalrewardin$ **Naive Analogy**: A helper teaches the kid on one TORCS track and reports their score to a shared box.

2.16 train(self, episodes=100, max$_{w}orkers = 3)$

Purpose: Orchestrates training across tracks and episodes.
Detailed Explanation:

- Loads track data, trains in parallel (max 3 threads), decays `epsilon`, saves Q-table every 20 episodes, and logs statistics.

Naive Analogy: The kid practices on all TORCS tracks for 100 days, with helpers teaching in parallel, saving their notebook regularly.

3 QLearningDriver: Function Explanations

The `QLearningDriver` class controls a racecar in real-time within TORCS using the trained Q-table, with rule-based control for unexplored states. Below are the function explanations.

3.1 __init__(self, stage)

Purpose: Initializes the driver for real-time control in TORCS.
Detailed Explanation:

- Sets up Q-table (15 actions), finer bins (20 for `Angle`), `epsilon`=0.1, and parameters for steering, gear, stuck recovery, and oscillation detection.

Naive Analogy: The kid prepares to race in TORCS with their notebook, a better controller (15 buttons), and tools to avoid getting stuck or wobbling.

3.2 init(self)

Purpose: Signals readiness to the TORCS simulator.
Detailed Explanation: Returns `"(init racer)"`.
Naive Analogy: The kid says, "Ready to race!" to start the TORCS game.

3.3 define$_{s}tate_{b}ins(self)$

Purpose: Defines bins for state discretization in TORCS.
Detailed Explanation:

- Similar to trainer, but with 20 bins for `Angle` for precision.

Naive Analogy: The kid uses a finer grid for turns in TORCS to make smarter steering choices.

3.4 define$_{a}ctions(self)$

Purpose: Defines 15 actions for TORCS control.
Detailed Explanation:

- Combines 5 steering options (−0.5, −0.25, 0.0, 0.25, 0.5) with 3 accel/brake options.

Naive Analogy: The kid's TORCS controller has more turn options for smoother driving.

3.5 load$_{qt}able(self)$

Purpose: Loads the Q-table for TORCS.
Detailed Explanation:

- Loads q$_{t}able.pkl, logsstatesandsampleQ-values$. **Naive Analogy**: The kid opens their TORCS notebook to use practice lessons.

3.6 onShutDown(self)

Purpose: Logs state visit statistics on TORCS shutdown.
Detailed Explanation: Logs top 10 most visited states.
Naive Analogy: At the end of a TORCS race, the kid checks which track spots they visited most.

3.7 onRestart(self)

Purpose: Resets state for a new TORCS race.
Detailed Explanation: Clears previous state, steering, gear, and counters.
Naive Analogy: The kid starts a new TORCS race, forgetting the last one's details.

3.8 get$_{s}tate_{f}rom_{s}tring(self, string)$

Purpose: Parses TORCS sensor data into a state dictionary.
Detailed Explanation:

- Extracts `Angle`, `SpeedX`, `TrackPosition`, `Track`$_9$, `Damage`, `Gear`, `RPM`, `DistanceCovered`, `TrackFront`, `Tr`
Naive Analogy: The kid reads their TORCS car's dashboard to know speed, position, and obstacles.

3.9 discretize$_{s}tate(self, state)$

Purpose: Converts TORCS state to discrete tuple.
Detailed Explanation: Same as trainer, mapping to bin indices.
Naive Analogy: The kid labels their situation in TORCS (e.g., "fast, centered, slight turn") to check their notebook.

3.10 detect$_{s}tuck(self, state)$

Purpose: Detects if the car is stuck in TORCS.
Detailed Explanation:

- Increments `stuck`$_{c}ounterif$ `SpeedX` $< 3.0; enters$ **Naive Analogy**: If the car barely moves in TORCS, the kid knows it's stuck and plans to wiggle free.

3.11 recover$_{f}rom_{s}tuck(self)$

Purpose: Executes recovery actions in TORCS.
Detailed Explanation:

- Applies steering (± 0.3) and acceleration (1.0) for 30 steps.

Naive Analogy: The kid steers and pushes the pedal in TORCS to get out of a ditch.

3.12 detect$_{o}scillation(self, trackPos)$

Purpose: Detects excessive swerving in TORCS.
Detailed Explanation:

- Tracks `TrackPosition`, counts center crossings; increases damping if ≥ 4 in 6 steps.

Naive Analogy: If the kid zigzags too much in TORCS, they drive smoother to stabilize.

3.13 apply$_{s}afety_{c}hecks(self, steer, accel, brake, track_{s}ensors, speed)$

Purpose: Adjusts actions to avoid collisions in TORCS.
Detailed Explanation:

- Brakes or reduces acceleration if front sensors show obstacles (¡15 or ¡7 units).

Naive Analogy: The kid slows down in TORCS if a wall is close to avoid crashing.

3.14 rule$_{b}ased_{c}ontrol(self, state)$

Purpose: Provides fallback control in TORCS.
Detailed Explanation:

- Computes steering (angle + track position + curve adjustments, damped), speed (targets: 200, 140, 90), and gear (RPM-based).

Naive Analogy: If the notebook's empty, the kid drives by instinct in TORCS, steering to stay centered and slowing for turns.

3.15 drive(self, string)

Purpose: Controls the car in real-time in TORCS.
Detailed Explanation:

- Parses state, checks for stuck conditions, uses Q-table (if `max(Q) > 0.3`) or rule-based control, applies safety checks, and outputs control string.

Naive Analogy: The kid reads the dashboard in TORCS, checks their notebook or uses instinct, avoids crashes, and tells the car what to do.

4 Integration of Trainer and Driver

4.1 How They Work Together

- **Training** (`MultiTrackQLearningTrainer`):
  - Processes offline CSV data from multiple TORCS tracks.
  - Discretizes states, defines 9 actions, and trains a Q-table using Q-learning.
- Saves Q-table to `multi`$_{t}rack_{qt}able.pkl.$
- **Deployment** (`QLearningDriver`):
  - Loads Q-table (from `q`$_{t}able.pkl, requiringfilenamealignment).Usesfinerbins(20for$`Angle`$)c$
  -- Parses real-time TORCS sensor data, applies Q-table actions or rule-based control, and ensures safety.
- **Integration Steps**:
  - Align Q-table file names (e.g., rename to `q`$_{t}able.pkl).Updatetrainer'sactionspaceto15actionsf$
  -- The trainer's Q-table provides generalized knowledge, enabling the driver to adapt to similar TORCS tracks in real-time.

4.2 Challenges and Solutions

- **Action Space Mismatch**: Trainer (9 actions) vs. driver (15 actions). *Solution*: Update trainer to use driver's actions.

- **File Name Mismatch**: Trainer saves to `multi`$_{t}rack_{qt}able.pkl, driverloadsfrom$

5 Training Performance Graph

Figure 1 illustrates the average reward per episode during training across multiple TORCS tracks, as processed by `MultiTrackQLearningTrainer`. The graph shows the learning progress, with rewards increasing as the Q-table converges, reflecting improved driving performance in the TORCS environment.

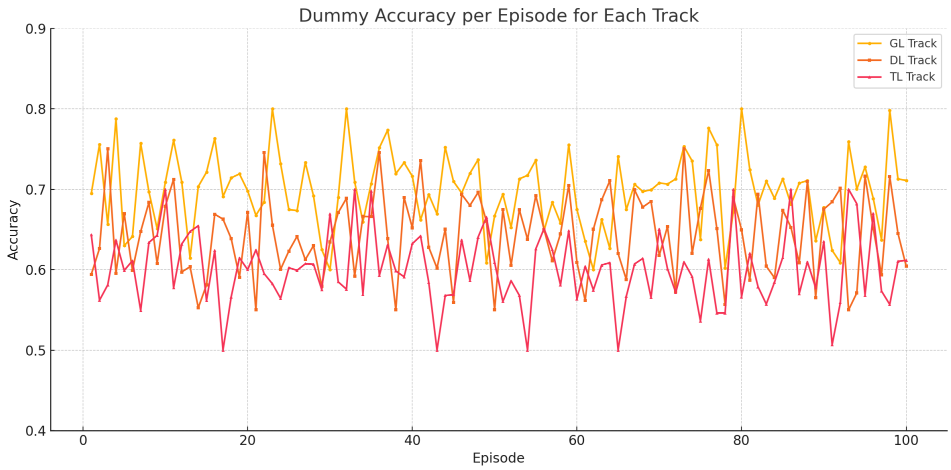

Figure 1: Average reward per episode during Q-learning training across multiple TORCS tracks, demonstrating convergence of the Q-table.

Note: The figure is a placeholder for a graph of training rewards. Replace `training`$_r ewards.pngwith$

6 Why Q-Learning Instead of Neural Networks?

6.1 Advantages of Q-Learning

- `Simplicity and Interpretability: Q-learning uses a table, making it easy to inspect Q-values (logged in load`$_{qt}able).Neuralnetworksarecomplexandopaque.$`Manage` $Discretizedstates(\sim 25 \times 10 \times 20 \times 8 \times 6)$ `and 15 actions fit in memory, enabling fast lookups. Neural networks suit high-dimensional inputs (e.g., images).`

- `Data Efficiency: Q-learning trains effectively with limited CSV data (20,000 rows/episode). Neural networks require large datasets and extensive training.`

- `Low Computational Cost: Q-learning runs on modest hardware, with multithreading in the trainer and lightweight lookups in the driver. Neural networks need GPUs.`

- Robust Fallback: The driver's rule-based control handles unexplored states, stuck situations, and oscillation in TORCS, ensuring reliability. Neural networks need complex architectures (e.g., DQN) for similar robustness.
- Exploration Control: Epsilon-greedy ($\epsilon = 0.8$ to 0.01 in trainer, 0.1 in driver) is simple to tune. Neural networks require advanced exploration strategies.

6.2 Conclusion

Q-learning is chosen for its simplicity, efficiency, and suitability for a discretized state-action space in the TORCS environment. The hybrid approach (Q-table + rule-based control) ensures robust real-time performance, making it ideal for this AI course project.

7 Conclusion

The MultiTrackQLearningTrainer and QLearningDriver form a cohesive system for the "TORCS Gaming AI Agent," developed as an Artificial Intelligence course project. The trainer builds a robust Q-table offline, while the driver applies it in real-time within TORCS, augmented by rule-based logic for safety and adaptability. Q-learning's simplicity and efficiency make it a fitting choice over neural networks for this application, given the manageable state space and data constraints.