Technical University of Crete
School of Electrical and Computer Engineering

Autonomous Agents

Mario Kart DS — RL Agent

Autonomous Navigation using DQN and CNN

Project Documentation

Student:
Karalis Asterinos ID: 2020030107

Contents

1 Introduction **2**
1.1 Project Objective . . . 2
1.2 About Mario Kart DS . . . 2
1.2.1 Core Mechanics and Gameplay . . . 2
1.2.2 Project's enviroment . . . 2

2 Environment and Setup **3**
2.1 Environment Configuration . . . 3
2.2 DQN-CNN Architectural Integration . . . 3

3 Code Implementation **4**
3.1 Project Structure . . . 4
3.2 File Roles and Descriptions . . . 4
3.2.1 Core Environment . . . 4
3.2.2 Training and Execution . . . 4
3.2.3 Analysis and Monitoring . . . 5
3.2.4 Utilities and Configuration . . . 5
3.3 Reward Function and Watchdog Logic . . . 5
3.3.1 Standard Reward Calculation . . . 5
3.3.2 Watchdog Termination Systems . . . 6
3.3.3 Completion Reward . . . 6
3.4 Analysis and Results . . . 6
3.4.1 Telemetry and Spatial Visualization . . . 6
3.4.2 Training Metrics and Learning Curves . . . 7
3.4.3 Demonstration and Real-time Evaluation . . . 7

4 Execution Workflow **8**
4.1 Phase 1: Environment Initialization . . . 8
4.2 Phase 2: Training the Agent . . . 8
4.3 Phase 3: Data Analysis and Plot Generation . . . 8
4.4 Phase 4: Agent Evaluation and Demonstration . . . 9

5 Experimental Results **10**
5.1 Experiment Configuration . . . 10
5.2 Visual Performance Analysis . . . 10

References **12**

1 Introduction

1.1 Project Objective

The primary objective of this project is the development of an autonomous Reinforcement Learning (RL) agent capable of navigating the complex, high-dimensional environment of *Mario Kart DS*. The architecture employs a Deep Q-Network (DQN) that maps raw visual inputs to discrete driving actions through a Convolutional Neural Network (CNN) backbone. By integrating visual perception with low-level telemetry data, the agent learns to optimize its racing line through iterative trial and error.

Mario Kart DS presents a significant challenge for autonomous agents due to the necessity of real-time decision-making in a dynamic, stochastic environment while interpreting pixel-based observations. To improve convergence, this project utilizes a dual-input methodology:

- **Visual Processing:** A CNN processes the top-screen display buffer to extract spatial features.
- **Telemetry Integration:** Internal RAM variables including speed, off-road status, and checkpoint progress are extracted to shape a robust and informative reward function.

This hybrid approach ensures the agent remains reactive to visual cues while being strictly grounded in the physical constraints and mechanics of the game engine.

1.2 About Mario Kart DS

Mario Kart DS is a landmark kart-racing title developed by Nintendo for the Nintendo DS, released in late 2005. As the fifth installment in the series, it features iconic characters and tracks based on locations within the Mario universe. The game was a critical and commercial success, selling approximately 23.6 million copies worldwide by 2016.

1.2.1 Core Mechanics and Gameplay

The racing environment involves competing against seven opponents on various circuitous tracks (Grand Prix mode), solo driving on the same maps for the best time (Time Trials mode) and other minigames. Central to the gameplay is the use of items obtained from "Item Boxes" scattered across the course that allow racers to attack and decelerate opponents (e.g., Shells) or provide temporary speed boosts to gain a competitive advantage (e.g., Mushrooms).

1.2.2 Project's enviroment

For the purposes of this project, the agent is evaluated within the **Time Trial** mode on the *Figure-8 Circuit*, utilizing a specific save state to ensure repeatable training conditions. This was chosen in order to simplify the enviroment as much as possible (no opponents, no item boxes, simple racing track). In this mode, the agent's only goal is to finish 3 laps as fast as possible.

2 Environment and Setup

The training environment bridges Nintendo DS hardware abstraction with modern Reinforcement Learning frameworks by wrapping the `DeSmuME` emulator within a custom `Gymnasium` interface (`MKDSEnv`). This architecture enables high-frequency state observation and low-latency action execution.

- **Emulator Interface:** `py-desmume` facilitates direct interaction with DS internals for real-time buffer capture and RAM manipulation.
- **RL Framework:** *Stable-Baselines3 (SB3)* provides the DQN implementation.
- **Image Processing:** `OpenCV` and `Pillow` downsample RGBX buffers into 84×84 grayscale tensors.

2.1 Environment Configuration

The `MKDSEnv` class manages the emulator lifecycle and state transitions.

- **State Representation:** Observations use the top-screen display, cropped and downsampled.
- **Temporal Context:** A `VecFrameStack` of 4 frames allows the agent to perceive velocity and direction.
- **Action Space:** Discretized into three outputs (Straight (Gas), Left (Gas+Left), and Right (Gas+Right))to accelerate initial convergence.
- **Reset Logic:** The environment utilizes `mkds_boot.dst` to ensure every episode begins at the Figure-8 Circuit starting line.

2.2 DQN-CNN Architectural Integration

The agent employs a Deep Q-Network where a Convolutional Neural Network (CNN) acts as the function approximator for state-action values, $Q(s, a)$.

- **Input Tensors:** The state s is a 4-frame grayscale stack of shape $(4, 84, 84)$.
- **Feature Extraction:** The *Nature CNN* uses three convolutional layers (32 filters 8×8, 64 filters 4×4, and 64 filters 3×3) to identify track boundaries and positioning.
- **Policy Head:** A 512-unit fully connected layer outputs 3 neurons predicting future rewards for the discrete action set $\{Straight, Left, Right\}$.

The agent follows an ϵ-greedy policy: $a = \arg\max_a Q(s, a)$. Crucially, while the CNN processes visual pixels, telemetry data (speed, off-road status) is used exclusively for the reward calculation (R_t) to guide the weight updates during backpropagation.

3 Code Implementation

The project consists of the following files. You can find the latest version of this project on my GitHub repository: `https://github.com/Asterinos1/Mario-Kart-DS-RL-Agent`. The current build referenced here is the 'MKDS - Project Submission Version' release (tag v1.0.0)

3.1 Project Structure

The following directory tree outlines the organization of the project's files:

```
Mario-Kart-DS-RL-Agent (root)/
├── analysis/ ..... Data processing and visualization
│   ├── plot_generator.py ..... For processing csv log files
│   └── tf_event_parser.py ..... For processing DQN log files
├── env/ ..... Gymnasium environment wrapper
│   └── mkds_gym_env.py ..... Custom class for setting the environment used in training
├── media/ ..... Contains media files (mp4 and gif)
├── logs/ ..... TensorBoard training logs, (will be generated during training)
├── outputs/ ..... Generated models and telemetry, (will be generated during training)
│   └── {run_id}/ ..... Example of what the insides will look like after training
│       ├── logs/ ..... Contains a csv file with various metrics captured during training
│       ├── models/ ..... Saved models from training
│       └── plots/ ..... Generated plots from the analysis scripts
├── rom/ ..... Game ROM directory
├── src/utils/ ..... Helper scripts and configurations
│   ├── callbacks.py
│   ├── config.py ..... Helper scripts and configurations.
│   └── ram_vars_testing.py ..... Helper scripts and configurations
├── demo.py ..... Model evaluation script
├── mkds_boot.dst ..... Initialization save state
├── requirements.txt ..... Required packages for enviroment setup
└── train_sb3_dqn.py ..... Main training entry point
```

3.2 File Roles and Descriptions

The implementation is categorized into four functional areas: the Environment, Training, Analysis, and Utilities.

3.2.1 Core Environment

- `mkds_gym_env.py`: Defines the `MKDSEnv` class, which inherits from `gymnasium.Env`. It manages the emulator lifecycle, captures the top-screen display for CNN input, and extracts RAM-based telemetry to calculate the reward signal.

3.2.2 Training and Execution

- `train_sb3_dqn.py`: The primary entry point for training. It configures the Stable-Baselines3 DQN agent, sets up the `SubprocVecEnv` for parallel processing, and manages checkpoint logic.

- `demo.py`: A specialized script for evaluating trained agents. It enables emulator visualization and loads a selected model to run autonomously.

3.2.3 Analysis and Monitoring

- `plot_generator.py`: Processes the `telemetry_log.csv` to create heatmaps, action distribution charts, cumulative reward and terminal reason summaries.
- `tf_event_parser.py`: Extracts data from TensorBoard event files to generate various custom plots.

3.2.4 Utilities and Configuration

- `config.py`: Centralizes hyperparameters and memory addresses for the USA version of the ROM.
- `callbacks.py`: Implements `MKDSMetricsCallback` to record telemetry data periodically during training.
- `ram_vars_testing.py`: A diagnostic tool used to verify that pointer-based memory extraction (speed, grip, etc.) is functioning correctly.

3.3 Reward Function and Watchdog Logic

To facilitate effective policy learning, the environment implements a complex reward shaping mechanism. This mechanism transforms raw RAM telemetry into a scalar reinforcement signal, guiding the agent toward optimal racing behavior while penalizing undesirable states.

3.3.1 Standard Reward Calculation

During nominal driving conditions, the reward is primarily driven by the forward velocity and progression through the course's checkpoint system:

- **The Checkpoint System:** Checkpoints are internal, invisible boundaries placed linearly along the track. In *Mario Kart DS*, the game engine uses these to verify that a player is traversing the circuit in the correct order and to calculate lap completion. For the RL agent, these act as "milestones" that break the long-term goal of finishing a lap into smaller, achievable tasks.
- **Progression Bonus:** A significant sparse reward of +15.0 is granted whenever the agent successfully crosses a new checkpoint ($CP_i > CP_{i-1}$). This prevents the agent from simply driving in circles or staying at the start line to accumulate small velocity rewards.
- **Velocity Reward:** The agent receives a continuous reward calculated as $R_v = v \times 2.0$, where v is the current forward speed extracted from the base pointer.
- **Surface Penalty:** To discourage driving on grass or dirt, the total reward is scaled by 0.5 if the `offroad` modifier drops below 0.9, effectively halving the agent's efficiency.

3.3.2 Watchdog Termination Systems

Because DQN can struggle with "sparse" failures where an agent is stuck but still accumulating small time-based rewards, several "Watchdog" systems are implemented to terminate suboptimal episodes early:

1. **Backward Driving Detection:** If the current checkpoint index is lower than the previous index within the same lap, the agent is assumed to be driving in the wrong direction. The episode is terminated with a penalty of -50.0.
2. **Checkpoint Timeout:** The agent must make progress within a specific internal time window. If the race timer exceeds 10 internal ticks without a checkpoint change, the episode is truncated with a -15.0 penalty to prevent idling.
3. **Collision Detection:** A sudden drop in velocity (greater than 50%) while at low speeds is interpreted as a wall collision. This results in immediate termination and a -30.0 penalty.
4. **Stuck Detection:** The environment monitors the Euclidean distance between the agent's current position (x, y, z) and its position from 80 cycles prior. If the distance moved is less than 50 units, a "stuck" state is triggered, resulting in a -20.0 penalty.

3.3.3 Completion Reward

Upon successfully completing three laps ($lap > 3$), the agent receives a final completion bonus of $+100.0$, signaling the successful achievement of the project's objective.

3.4 Analysis and Results

The evaluation phase of this project relies on custom-built analytical tools that process raw training data into interpretable visualizations. These tools allow for the objective assessment of the agent's spatial awareness, action preferences, and the effectiveness of the watchdog systems.

3.4.1 Telemetry and Spatial Visualization

The `plot_generator.py` script serves as the primary tool for post-run analysis. By parsing the `telemetry_log.csv` file, it generates four critical visualizations:

- **Cumulative Reward:** A plot showing the total reward as it accumulates during the training of the agent.
- **Track Position Heatmap:** Utilizing a Gaussian Kernel Density Estimate (KDE) via `seaborn`, this plot visualizes the density of the agent's (x, z) coordinates. It provides immediate feedback on the agent's chosen racing line and identifies areas where the agent frequently goes off-road.
- **Action Distribution:** This bar chart maps numeric actions to their descriptive counterparts (*Gas*, *Gas+Left*, *Gas+Right*). It is used to detect "action collapse," where an agent might over-rely on a single input, such as constant steering.

- **Termination Analysis:** A pie chart categorizing the reasons for episode endings—such as *stuck*, *collision*, *timeout*, or *finished*. This is crucial for tuning the penalty weights in the reward function.

3.4.2 Training Metrics and Learning Curves

Long-term performance trends are monitored using TensorBoard logs and the `tf_event_parser.py` script. Key metrics analyzed include:

- **Exploration Rate :** Tracks the epsilon decay process, visualizing the transition from random exploration to deterministic exploitation of the learned policy.
- **Training Efficiency and Throughput:** Monitored via both `rollout/fps` and `time/fps` to measure the hardware processing speed and the overhead of the parallelized environments.
- **Network Convergence Metrics:** Analyzes `train/loss` and `train/learning_rate` to ensure stable optimization of the Q-network and to verify the behavior of the learning rate scheduler.
- **Episode Duration (`rollout/ep_len_mean`):** Measures the number of steps per episode, providing insight into whether the agent is completing laps more quickly or surviving longer before a watchdog trigger.

3.4.3 Demonstration and Real-time Evaluation

The `demo.py` script provides a qualitative assessment by allowing the user to select a specific model checkpoint (`.zip`) and observe the agent's driving in real-time. Unlike the training phase, the demonstration environment forces the `visualize` parameter to `True`, enabling the SDL window to display the emulator's output directly. This allows for a direct comparison between the analytical plots and the actual physical behavior of the agent on the track.

4 Execution Workflow

The following section outlines the end-to-end procedural pipeline for deploying the Mario Kart DS RL agent. The workflow is designed to be modular, allowing for seamless transitions between environment preparation, model training, and performance analysis.

4.1 Phase 1: Environment Initialization

Before execution, the software dependencies and game assets must be correctly positioned within the project structure:

1. **Dependency Installation:** Install the required Python packages using the provided requirements file: `pip install -r requirements.txt`.
2. **ROM Placement:** Place the Mario Kart DS (USA version) ROM in the `rom/` directory. The `config.py` script is designed to auto-detect any `.nds` file in this location.
3. **Save State Verification:** Ensure the `mkds_boot.dst` file is in the root directory. This state is critical for placing the agent at the Figure-8 Circuit start line during every reset cycle.

4.2 Phase 2: Training the Agent

The training process utilizes the Stable-Baselines3 DQN implementation and supports both fresh starts and resuming from checkpoints:

- **Execution:** Run the main training script: `python train_sb3_dqn.py`.
- **Model Selection:** Upon execution, an interactive menu scans the `outputs/` directory. The user can select an existing `.zip` model to resume training or press Enter to initiate a new session.
- **Safe Interruption:** Users may interrupt training at any time using `Ctrl+C`. The script includes a `finally` block that triggers an "Emergency Save," capturing the current model and replay buffer state before exiting.

4.3 Phase 3: Data Analysis and Plot Generation

Once training data is captured in the `telemetry_log.csv` and TensorBoard files, visualizations are generated to interpret the results:

1. **Telemetry Visualization:** Execute `python plot_generator.py` from the `analysis/` directory. Select the desired `run_id` to generate heatmaps and action distribution charts.
2. **Metric Parsing:** Run `python tf_event_parser.py` to convert raw TensorBoard events into readable performance curves.
3. **Output:** All generated graphs are automatically saved to the `outputs/{run_id}/plots/` folder for inclusion in research reports.

4.4 Phase 4: Agent Evaluation and Demonstration

The final phase involves observing the agent's learned behavior in a live environment:

- **Live Demo:** Run `python demo.py` to launch the emulator with visualization enabled.
- **Deterministic Testing:** The demo script loads the selected model and executes actions in a non-stochastic manner, allowing for a qualitative assessment of the agent's optimized racing line.

5 Experimental Results

The evaluation of the DQN agent was conducted through a series of training episodes on the *Figure-8 Circuit*. This section outlines the quantitative and qualitative performance of the agent, grounded in the data extracted via `plot_generator.py` and `tf_event_parser.py`.

5.1 Experiment Configuration

To ensure reproducibility, the agent was trained using a standardized set of hyperparameters and a parallelized environment setup. The training process leveraged the `SubprocVecEnv` wrapper to run four concurrent emulator instances, maximizing data collection efficiency.

Hyperparameter	**Value**
Total Timesteps	300.000
Learning Rate	0.00025
Batch Size	128
Buffer Size	50,000
Gamma (γ)	0.99
Frame Stacking (n)	4

Table 1: DQN Training Hyperparameters

5.2 Visual Performance Analysis

Below are the plots generated by the analysis tools for the above configuration.

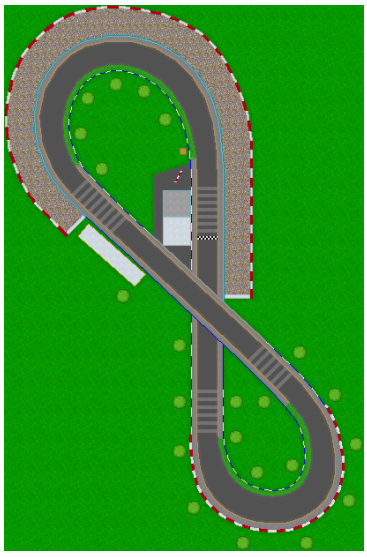

Figure 1: Figure-8 Circuit Reference Map

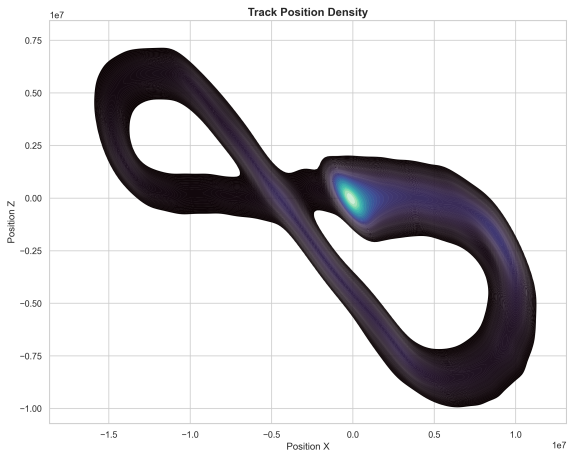

Figure 2: Track Position Density (Heatmap)

From the comparison, we can see how the agent learns the layout of the map and avoids crashing on the walls and driving off-road (grass sections).

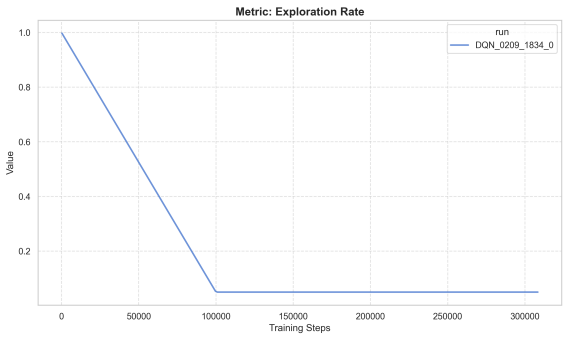

Figure 3: Exploration Rate (Epsilon Decay)

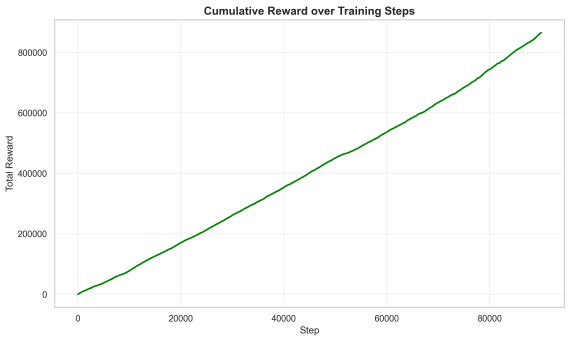

Figure 4: Cumulative Reward Growth

For the given run we can see how the exploration rate decays and how the total reward accumulates. Something to note here is the cumulative reward is plotted up to approx. 90.000. This was taken from a another test (due to time restrictions) but the behavior is the same on every run.

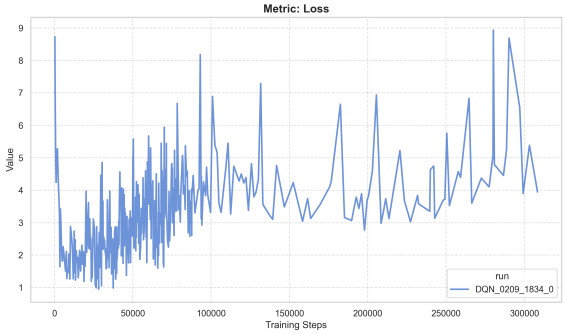

Figure 5: Neural Network Training Loss

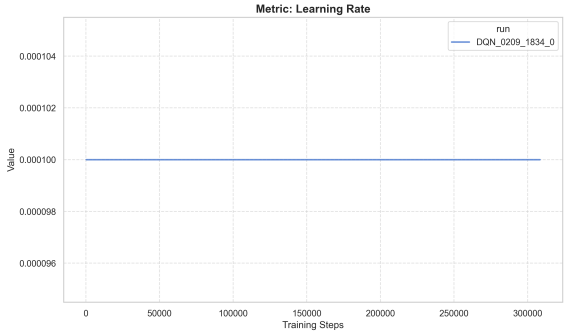

Figure 6: Learning Rate Schedule

Here we have some other metrics. Learning Rate is set to steady for the entire run while loss seem to fluctuate through out the run. This can be attributed to the agent "learning" a portion of the course at a time.

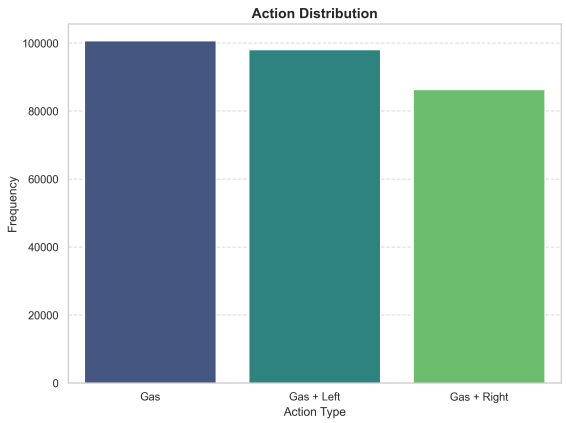

Figure 7: Input Action Distribution

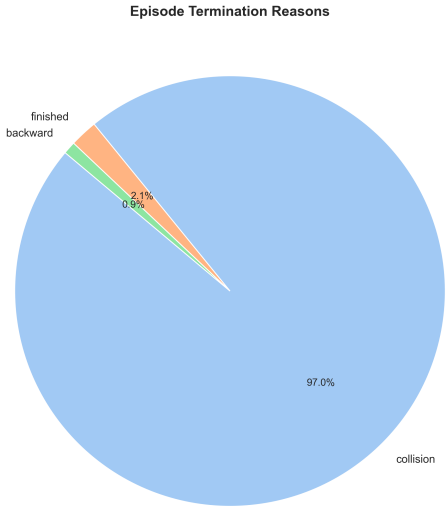

Figure 8: Terminal State Reason Analysis

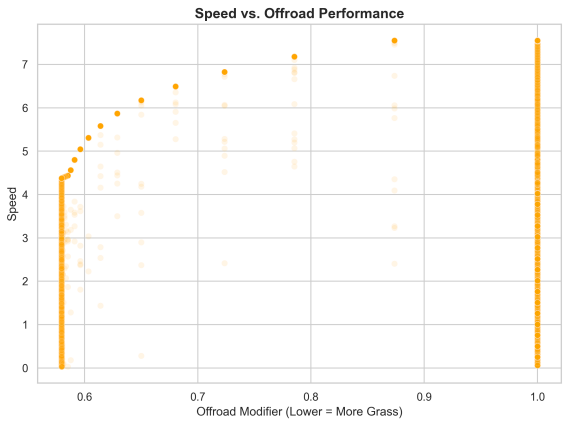

Figure 9: Speed vs. Offroad Density Correlation

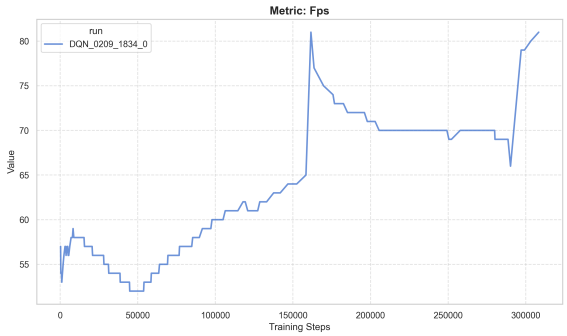

Figure 10: System Throughput (FPS)

In the last four graphs we can see more details of the agent's behavior. How he tends to steer more to the left (attributed to learning the first left corner, if we watch the mp4 we will observe how he tends to stick to the right part of the road constantly), the main reason for resetting a run is mainly crashing on walls, with a small portion of finishing and very rarely driving backwards. Also we can observe that more often (almost double) the agent drives at full speed rather than being slowed on off-road or sliding against walls. Finally the performance of running 4 instances is getting a lot better at around 150.000, this is because the agent is getting better and tends to reset less (resetting causes extra load on the CPU).

References

Stable-Baselines3: `https://github.com/DLR-RM/stable-baselines3`

Gymnasium: `https://gymnasium.farama.org/`

py-desmume: `https://pypi.org/project/py-desmume/`

DeSmuME Emulator: `https://desmume.org/`

Mario Kart DS: `https://en.wikipedia.org/wiki/Mario_Kart_DS`

MKDS TAS Community Works: `https://tasvideos.org/userfiles/game/1652`