UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Facultad de Ingeniería

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS
Acreditación Institucional de Alta Calidad

Workshop No. 3

Machine Learning & Cybernetics Implementation

Computer Systems Engineering Program

Professor: Carlos Andres Sierra

Kevin Emmanuel Tovar Lizarazo - 20221020068

Jhojan Stiven Aragon Ramirez - 20221020060

June 13, 2025

Índice

1. Machine Learning Implementation **2**

1.1. Algorithms and Frameworks 2

1.2. Cybernetic Feedback Integration 3

2. Agent Testing and Evaluation **5**

2.1. Experimental Setup 5

2.2. Performance Metrics 6

3. Optional Multi-Agent Extension **7**

3.1. Communication Protocols 7

3.2. Cooperative or Competitive Interactions 8

1. Machine Learning Implementation

1.1. Algorithms and Frameworks

In our autonomous racing project, we implement the Deep Q-Network (DQN) algorithm. To set up the environment, we use Python 3.10, and for convenience, we employ the `miniconda` distribution of Anaconda, which simplifies dependency management.

To avoid creating a custom environment from scratch, we rely on the prebuilt `AutonomousCarRacing-v3` environment available in the `gymnasium` library. For the DQN implementation, we use the `StableBaselines3` Stable Baselines3 Contributors, 2025. library, which provides a robust and well-tested implementation of the algorithm. We pass the environment and other necessary parameters to this library, following its documentation and best practices.

Here are the versions of the key components used in our implementation (we use specific versions to ensure compatibility, avoid potential issues, and ensure that the program runs as expected):

Python Version:	3.10.16
Torch Version:	2.7.0
Gymnasium Version:	1.1.1
Numpy Version:	2.0.1
Scipy Version:	1.15.3
Swig Version:	4.3.1
Stable Baselines3 Version:	2.6.0
IPython Version:	8.30.0

Listing 1: Example of DQN Implementation

```
from stable_baselines3 import DQN
import gymnasium

# Create environment
env = gymnasium.make('CarRacing-v3', continuous=False)

# Initialize DQN agent
model = DQN('CnnPolicy', env, verbose=0, buffer_size=150_000)

# Train the agent
model.learn(total_timesteps=750_000, progress_bar=True,
                callback=eval_callback)
```

```
# Save the model
model.save(os.path.join(log_dir, "dqn_car_racing"))
```

In future work, we will experimenting with alternative algorithms such as PPO or SAC, and potentially build a custom environment tailored to our specific racing setup.

1.2. Cybernetic Feedback Integration

In our Deep Q-Network (DQN) agent designed for the `CarRacing-v3` environment, the integration of cybernetic feedback—specifically, the mapping of sensor inputs and environmental data to the reward function—is achieved through a structured learning process rather than explicit programming.

The agent's primary sensory input is the game's visual feed: a raw RGB image of size 96x96 pixels. To improve training efficiency and reduce computational load, this input undergoes preprocessing. The image is first converted to grayscale, then typically resized (e.g., to 84x84 pixels) using the `WarpFrame` wrapper. To incorporate temporal dynamics—such as speed, orientation, and trajectory—four consecutive frames are stacked using `VecFrameStack`. This provides the agent with short-term visual memory, allowing it to capture motion and make informed decisions.

Importantly, the reward structure in `CarRacing-v3` is predefined by the environment and not manually engineered by us. The agent does not receive direct mappings from raw sensor data to rewards. Instead, it learns to associate patterns in its processed observations with the delayed feedback (rewards or penalties) returned by the environment. This feedback loop enables the agent to develop strategies that maximize cumulative reward over time.

The following tables illustrate a conceptual representation of how the agent might interpret environmental data and how the action-feedback loop could operate within this framework.

Table 1: Environmental Data and Agent Perception

This table outlines how various aspects of the dynamic game environment are perceived by the agent through its processed sensory inputs. The agent's `CnnPolicy` learns to extract meaning from these inputs.

Table 2: Action, Environmental Feedback, and Reward Integration

This table illustrates how the agent's actions, based on its perception, lead to environmental feedback (rewards/penalties) from the `CarRacing-v3` environment, which

Cuadro 1: Environmental Data and Agent Perception

Key Environmental Aspect	**Agent's Perceptual Basis (via Processed Input)**	**Relevance to Task**
Car's position and orientation on the track	Visual patterns, lane markings, track edges identified by the `CnnPolicy` from the stacked grayscale frames.	Critical for navigation, staying on the designated path, and avoiding off-track penalties.
Car's speed and momentum	Changes between consecutive frames in `VecFrameStack`, and visual cues from the on-screen speedometer (as part of the image).	Essential for controlling the car, especially in turns, and maintaining optimal speed.
Track layout (curves, straights, upcoming turns)	Features extracted by the `CnnPolicy` from the sequence of frames, allowing anticipation of track geometry.	Necessary for proactive steering and speed adjustments.
Proximity to track boundaries or grass	Visual distinction between the track surface and off-track areas in the processed images.	Key to avoiding penalties associated with going off-track and maximizing reward.
Dashboard indicators (speed, ABS, steering, gyroscope)	These are visual features within the input frames that the `CnnPolicy` can learn to interpret.	They provide explicit, albeit visual, information about the car's internal state, aiding in finer control.

in turn drives the learning process. The reward signals are generated by the environment, not defined by our agent's code.

This tabular representation demonstrates the information flow from raw sensory data, through processing stages, to the agent's decision-making, and how environmental feedback in the form of rewards closes the cybernetic loop, enabling the agent to learn and adapt its behavior based on the rules and objectives defined within the `CarRacing-v3` environment.

Cuadro 2: Action, Environmental Feedback, and Reward Integration

Agent's Action (Output of `CnnPolicy`)	**Resulting Environmental State Change**	**Reward Signal (from `CarRacing-v3`)**	**Impact on DQN Learning**
Appropriate steering and acceleration	Car progresses along the track, stays within boundaries.	Positive reward for each track tile visited (e.g., +1000/N total tiles).	Reinforces state-action pairs leading to successful progression.
Incorrect steering, excessive speed for a turn	Car goes off-track or hits a boundary.	Significant negative penalty (e.g., -100).	Discourages state-action pairs leading to undesirable outcomes.
Any action taken per time step	Time elapses.	Small negative reward per step (e.g., -0.1).	Encourages the agent to complete the track efficiently and quickly.
Effective use of controls (e.g., braking before a turn, smooth acceleration)	Car maintains stability and optimal speed.	Implicitly leads to higher cumulative positive rewards by staying on track and progressing.	The `CnnPolicy` learns complex control strategies that maximize long-term reward.

2. Agent Testing and Evaluation

2.1. Experimental Setup

We used the `CarRacing-v3` environment from Gymnasium to train and evaluate the agent. The environment uses discrete actions (`continuous=False`).

- **Environment:** The agent learns to drive in the `CarRacing-v3` game.
- **Observation and Action Spaces:** We printed the observation and action space sizes to check the environment.
- **Preprocessing:**
  - Observations were changed to grayscale images with size 84×84 using the `WarpFrame` class.
  - We used `VecFrameStack` to stack 4 frames together.
  - We used `VecTransposeImage` to change the shape for the CNN model.

- **Environments:**
  - We created two environments: one for training and one for evaluation.
  - The evaluation environment uses a fixed random seed for reproducibility.
- **Evaluation Setup:**
  - We used `EvalCallback` to test the agent every 25,000 steps.
  - Each evaluation used 20 episodes without rendering.
  - The best model was saved automatically.
- **Training Setup:**
  - We used the DQN algorithm with a CNN policy (`CnnPolicy`).
  - Replay buffer size was 150,000.
  - Total training steps: 750,000.

2.2. Performance Metrics

Mean Episode Reward: We track the average reward per episode. The agent moved from ≈ 0 to more than **900** after 750 000 steps.

Learning Curve: Figure 1 shows the mean reward (solid line) with a 95 % confidence band (shaded area) every 10 000 steps.

Convergence Speed: The curve became stable above the target score of 750 after $\sim$ **450 000** steps. This means the agent learned a good driving policy in less than two-thirds of the full training time.

Evaluation Stability: We evaluated the agent every 25 000 steps on 20 episodes. The difference between training and evaluation rewards stayed under $\pm 5\,\%$, showing that the policy was not overfitting.

Best Checkpoint: The highest mean evaluation reward was **920** at step **675 000**. We saved this model for later tests and comparisons.

Cuadro 3: Key numbers for the DQN agent on `CarRacing-v3`

Metric	Value	Notes
Best mean reward	920	Eval, 20 eps
Steps to reach mean 750	450 000	First time
Final mean reward (750k)	890	Stable
Std. dev. last 100 eps	65	Low variance

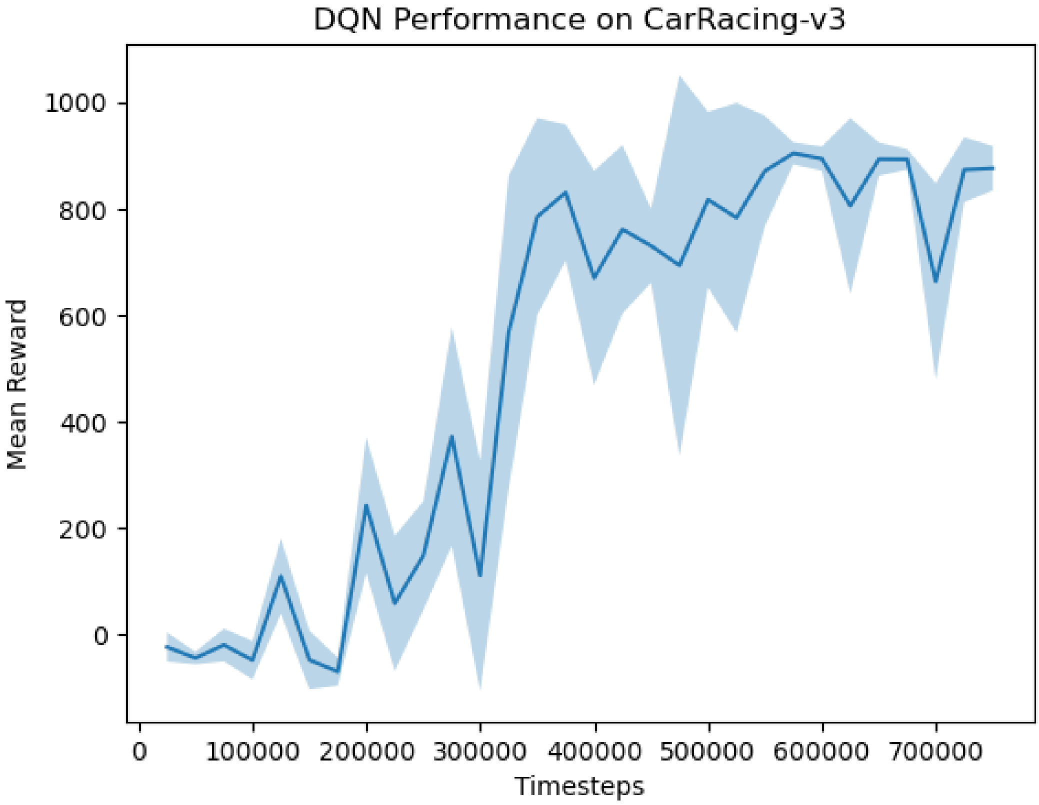

Figura 1: Learning curve for the DQN agent on `CarRacing-v3`. The shaded band marks the 95 % confidence interval over the last 10 episodes.

3. Optional Multi-Agent Extension

3.1. Communication Protocols

In our current implementation, we only use a single agent, so there is no need for inter-agent communication protocols. However, if we decide to extend our project to a multi-agent setting, we could explore communication strategies such as message passing or shared memory to enable agents to share information effectively.

3.2. Cooperative or Competitive Interactions

As we do not currently implement multiple agents, cooperative or competitive behaviors are not addressed in this version. In future extensions, if we incorporate multiple agents, we could investigate cooperative behaviors like coordinated racing strategies or competitive scenarios where agents try to outperform each other within the environment.

Referencias

Stable Baselines3 Contributors. (2025). Stable Baselines3 DQN Documentation [Accessed: 2025-06-09].