Reinforcement Learning Project: Deep Q-Network for CarRacing-v3

By : Corentin CHARPENTIER, Mattéo NOBILE

Introduction and Problem Overview

Reinforcement Learning (RL) is a paradigm focused on training an agent to make optimal decisions within an environment by learning from trial and error. The challenge of autonomous navigation represents a complex, high-dimensional RL problem, as the agent must process a continuous stream of visual data and make precise, sequential decisions.

This project tackles this challenge using the CarRacing-v3 environment. Our objective is to design, implement from scratch, and train an agent to drive a car around a track using only raw pixel data as input.

To solve this, we implemented a **Deep Q-Network (DQN)**. This algorithm is specifically designed to handle high-dimensional state spaces (like images) by combining Q-Learning with a deep convolutional neural network (CNN).

Environment and Preprocessing

1. Environment: CarRacing-v3

We selected the CarRacing-v3 environment, which provides a top-down view of a car on a randomly generated track.

- **State Space (Observation):** The default state is a 96x96x3 RGB image. This raw input is not ideal for learning. Our first key decision was to implement a PreprocessWrapper. This wrapper:
    1. Converts the 96x96 RGB image to grayscale (96x96x1).
    2. **Stacks 4 consecutive frames** together.
- **Justification:** Grayscale conversion simplifies the input, reducing computational load. Frame-stacking is crucial, as a single static image provides no sense of motion. By stacking 4 frames, the agent can infer velocity and angular direction, which is essential for making driving decisions. The final state provided to our agent is a (4, 96, 96) tensor.
- **Action Space:** Our second key decision was to use the discrete version of the environment (continuous=False). This provides 5 distinct actions:
    1. 0: Do nothing
    2. 1: Steer Left

3. 2: Steer Right
4. 3: Gas
5. 4: Brake

- **Justification:** While the environment supports continuous control, a discrete action space vastly simplifies the problem. It allows us to use a value-based method like DQN, which learns to output a single Q-value for each of the 5 possible actions, rather than a more complex policy-gradient method.
- **Reward Function:** The agent is incentivized to make progress efficiently. It receives a **-0.1** penalty for every frame (to encourage speed) and a **+1000 / N** reward for each new track tile it visits (where N is the total number of tiles). If the agent goes beyond the boundary, it will end the run and apply a **-100** penalty.

Algorithm and Training Rationale

1. Algorithm Choice: Deep Q-Network (DQN)

- **Why not basic Q-Learning?** A tabular Q-Learning approach is impossible. The number of possible states from a (4, 96, 96) image input is really large, making it impossible to store in a table.
- **Why DQN?** DQN is the foundational algorithm for solving this exact type of problem: pixel-based inputs with a discrete action space. It uses a **Convolutional Neural Network (CNN)** to handle the visual data and introduces two key mechanisms for stability: **Experience Replay** and a **Target Network**.

2. Core DQN Components

Our implementation in the DQNAgent class is built on these two stability-enhancing components:

1. **Experience Replay (ReplayBuffer):** We use a buffer that stores the agent's experiences (s, a, r, s', done).
    - **Justification:** This breaks the strong temporal correlation between consecutive samples and allows the agent to reuse experiences multiple times, dramatically increasing sample efficiency, by taking random batches.
2. **Target Network (q_network_target):** We use two networks. The *Online Network* is trained at every step, while the *Target Network* is a "frozen" copy that is updated only periodically.
    - **Justification:** This provides a stable target for the Bellman loss calculation. If we used only one network, the target Q-value would change at every single step, creating a "moving target" problem and destabilizing the training.

3. Network Architecture (DQNetwork_CNN)

We implemented a standard CNN architecture, designed to extract features from the stacked frames.

Layer	Type	In Channels	Out Channels	Kernel	Stride	Activation	Output Shape
Input	-	4	-	-	-	-	(N, 4, 96, 96)
Conv 1	Conv2d	4	32	8x8	4	ReLU	(N, 32, 23, 23)
Conv 2	Conv2d	32	64	4x4	2	ReLU	(N, 64, 10, 10)
Conv 3	Conv2d	64	64	3x3	1	ReLU	(N, 64, 8, 8)
Flatten	Flatten	-	-	-	-	-	(N, 4096)
Linear 1	Linear	4096	512	-	-	ReLU	(N, 512)
Output	Linear	512	5	-	-	None	(N, 5)

*(Note: Flattened size is 64 * 8 * 8 = 4096)*

4. Training Setup (train_dqn)

Our training loop is designed for robust, long-running experiments:

1. **Warmup:** Pre-fills the replay buffer with **1,000** steps of random actions.
2. **Exploration:** Uses an epsilon-greedy strategy, decaying epsilon from 1.0 to 0.01.
3. **Checkpointing:** Saves the full agent state (model, optimizer, epsilon) and metrics every **50 episodes**. This allows us to resume training (RESUME_TRAINING = True) without losing progress.
4. **Monitoring:** Saves training plots and records test videos (record_test_video) at each checkpoint to visually inspect the agent's policy evolution, as well as an updated version of the metrics.

Fine Tuning

Fine tuning is a major step to optimize the performance of our DQN agent, we performed a grid search on two critical hyperparameters: the **learning rate (LR)** and the **discount factor (Gamma)**.

We tested the following values:

- **Learning Rate:** [0.01, 0.001, 0.0001]
- **Gamma:** [0.99, 0.95, 0.9]

Each combination was trained for 10 episodes. As it was too long to get results (it took us 2 days to train 1250 episodes so doing a good amount of episodes for each combination is too long). Here are the results :

```
Learning Rate,Gamma,Avg Reward (Last 100),Max Reward,Min Reward
0.01,0.99,-57.04617147117811,-51.29870129870207,-64.51612903225892
0.01,0.95,-57.41611619362125,-42.857142857143536,-66.38655462184931
0.01,0.9,-54.39509466247692,-46.043165467626594,-63.45514950166203
0.001,0.99,-55.5675416072365,-46.04316546762659,-62.61682242990744
0.001,0.95,-56.99219616115488,-50.819672131148295,-61.78343949044676
0.001,0.9,-56.429247724484696,-46.93877551020476,-66.04938271605016
0.0001,0.99,-57.271985703973726,-46.99646643109612,-64.05228758170023
0.0001,0.95,-56.84285255494723,-48.616600790514525,-66.25766871165719
0.0001,0.9,-55.52826148314172,-46.42857142857213,-63.21070234113802
```

We could say that we take the best combination for example lr = 0.01 and gamma = 0.9 as the average reward is the highest, but we don't have enough data to be sure it is the best combination.

Hyperparameters and Rationale

1. Final Hyperparameters

The following parameters were used for our final training run, based on our updated car_racing_proj.py script:

Hyperparameter	Value	Description
Learning Rate (LR)	0.001	Adam optimizer learning rate.
Gamma	0.99	Discount factor for future rewards.

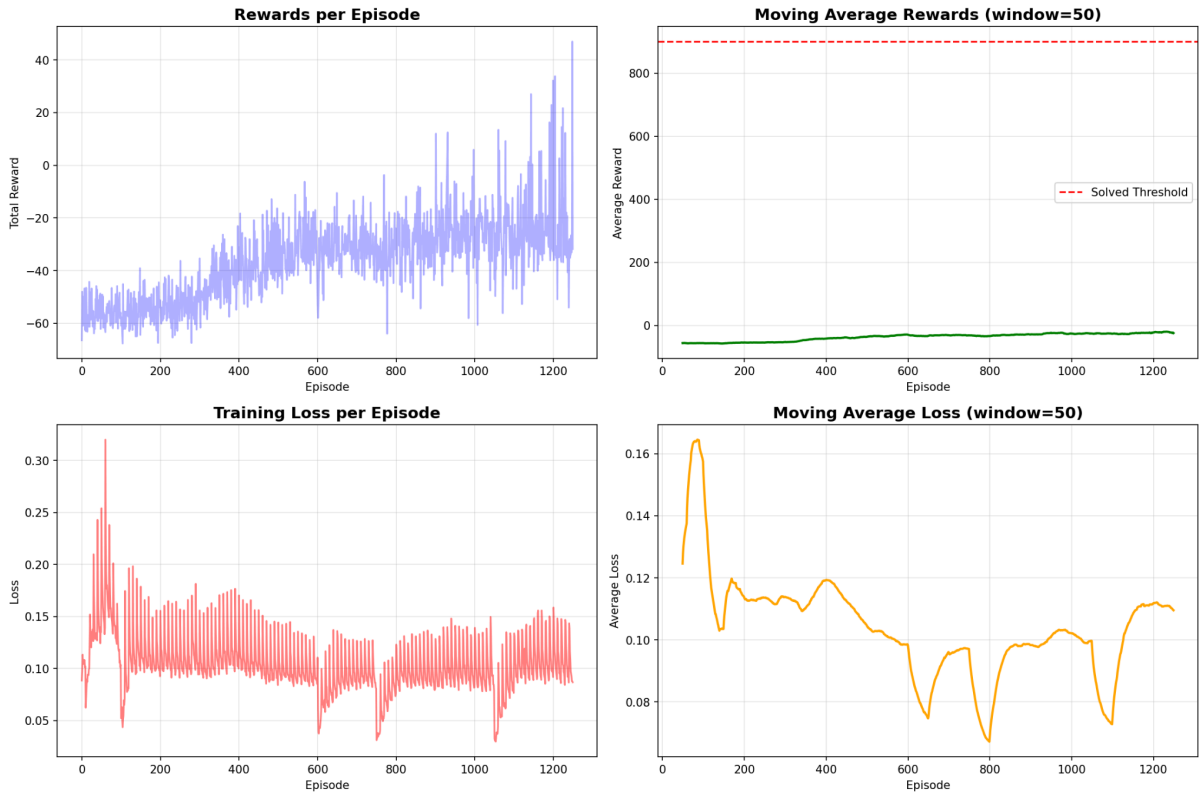
Buffer Capacity	50000	Total experiences stored in the Replay Buffer.
Batch Size	64	Experiences sampled per training step.
Target Update Freq.	10,000	Steps between updating the target network.
Epsilon Start	1.0	Initial exploration rate.
Epsilon End	0.01	Minimum exploration rate.
Epsilon Decay	0.9995	Multiplicative decay factor per episode.
Warmup Steps	1000	Steps to pre-fill the buffer.
Total Episodes	1500	Total episodes for the final training run.
Hidden Layer Size	512	Size of the fully-connected hidden layer.

2. Key Parameter Rationale

- **Learning Rate (0.001):** We chose 0.001 as a starting point. While 0.0001 is also common, our initial tests showed learning was slow. This higher rate promotes faster initial learning, which is then stabilized by the target network.
- **Buffer Capacity (50,000):** This is a trade-off. A larger buffer uses more memory and can contain very old policies. 50k is memory-efficient and ensures experiences are relatively recent.
- **Target Update Freq. (10,000):** This is a critical stability parameter. Updating too fast can destabilize learning. 10,000 steps provides a very stable target for many training batches.

3. Experiments and Results

We tracked the agent's performance by logging the total reward for each episode and the average training loss.



Rewards per Episode

- The raw reward curve (top-left) shows a slightly increasing trend over roughly 1200 episodes, from around -60 to -20 on average.
- However, the curve remains highly noisy, with large fluctuations between consecutive episodes.
- This suggests the agent occasionally finds partially correct trajectories but fails to stabilize consistent driving behavior.

Moving Average of Rewards

- The moving average of total rewards (top-right) remains low and nearly flat between 0 and 20, far below the environment's solving threshold of 900.
- This indicates the agent did not learn a stable or generalized driving policy.
- Despite minor improvements, the lack of any sustained upward spike shows the DQN failed to capture a coherent sequence of successful actions across tiles.

Training Loss

- The loss curve (bottom-left) exhibits high volatility during the first 200 episodes, followed by partial stabilization near 0.1.
- The periodic oscillations correspond to Target Network updates, which cause abrupt shifts in Q-value estimations.
- These oscillations imply the Q-value function remained unstable, meaning the agent struggled to form consistent value estimates.

Moving Average of Loss

- The smoothed loss (bottom-right) confirms partial stabilization around 0.10 ± 0.02, with no significant downward trend after episode 400.
- This plateau suggests the model reached a premature local minimum—it learned to minimize short-term prediction errors but didn't explore sufficiently to discover a high-reward long-term strategy.

Why It Didn't Work

Reward Sparsity

- In CarRacing-v3, the agent receives sparse positive rewards (+1000/N per visited tile) compared to constant penalties (-0.1 per frame).
- The learning signal is thus dominated by negative rewards, making it hard for the network to propagate meaningful gradients through early layers.
- This leads to a credit assignment problem, where the agent can't connect good actions to delayed rewards.

Discrete Action Space

- Using a discrete action space simplifies control but prevents fine-tuned steering and throttle adjustments.
- The agent's behavior becomes jerky ("left–gas–right–brake"), leading to unstable trajectories even when the logic of actions is sound.

DQN Instability in Continuous Environments

- DQN was originally designed for discrete, low-dimensional tasks (like Atari games).

- In continuous environments like CarRacing, Q-value overestimation becomes a serious issue, producing inconsistent action choices and poor policy convergence.

Exploration Decay Too Fast

- Even with an ε-decay of 0.9995, reducing ε from 1.0 to 0.01 across ~1500 episodes is still too fast for a complex environment like this.

- The agent likely stopped exploring too early, getting stuck in a suboptimal driving pattern.

Insufficient State Abstraction

- Grayscale inputs and 4-frame stacking capture local motion, but without mechanisms such as temporal encoding or feature attention, the model fails to infer long-term context (momentum, trajectory, cornering).

- The CNN thus detects local textures (road edges, grass) but doesn't fully understand global motion dynamics.

How to Improve It

Algorithmic Improvements

1. Double DQN (DDQN)
   → Reduces overestimation bias by decoupling action selection from action evaluation.
   → Improves convergence stability and more accurate Q-value learning.

2. Dueling DQN
   → Separates the *state value function* (V(s)) from the *advantage function* (A(s, a)).
   → Allows the network to assess the quality of a state independently of the action taken, which improves training efficiency in visual environments.

3. Prioritized Experience Replay (PER)
   → Samples transitions with high Temporal Difference (TD) error more often, focusing updates where learning is most needed (e.g., cornering events).

4. Reward Shaping
   → Introduce intermediate rewards, e.g.:

- +1 for staying on track for 10 consecutive frames

- +5 for maintaining correct orientation or low skid angle
  This densifies the reward signal and accelerates gradient propagation.

5. Switch to Continuous Control Algorithms (DDPG / TD3 / SAC)
   → These methods are designed for continuous action spaces.
   → Algorithms like TD3 or SAC would allow smoother throttle/steering control and better stability in driving behavior.

Training & Environment Adjustments

- Longer Training (3000+ episodes): CarRacing is notoriously hard to solve; DQN usually needs several thousand episodes for consistent driving.

- Slower ε-decay (0.9997–0.9999): Maintains exploration for a longer portion of training.

- Frame Skip Optimization: Applying the same action for 2–3 frames reduces variance and smooths control.

- Reward Normalization / Clipping: Scaling rewards between -1 and 1 helps stabilize gradients and prevents exploding updates.

Conclusion

Our Deep Q-Network successfully learned basic low-level motion patterns (accelerating, turning), but failed to generalize into a stable driving policy due to sparse rewards, limited action granularity, and Q-learning instability in continuous environments.

Future experiments using Double/Dueling DQN or continuous-action methods (e.g., TD3, SAC), combined with reward shaping and extended training, would likely produce a much more stable and competent driving agent.

6. Team Roles

Corentin : code writing
Matteo : fine tuning and graph analystics