Solving Self-Driving Car Racing with DQN and PPO
RL Project Report

Adrien Fu, Raphael Faure

March 28, 2025

1 Introduction

1.1 Motivation and Goal

The motivation for this project arises from the increasing interest in self-driving cars within the industry. One of the primary benefits is enhanced road safety; by eliminating human errors such as distractions and impaired driving, autonomous vehicles (AVs) have the potential to significantly reduce traffic accidents and fatalities. They can also help ease traffic congestion by maintaining optimal speeds and distances while minimizing stop-and-go traffic. Additionally, AVs offer environmental advantages, as they are often powered by advanced electric engines. When paired with clean energy sources, these engines can substantially lower greenhouse gas emissions.
The goal of this project is to showcase the effectiveness of Reinforcement Learning (RL) in developing autonomous vehicles. Specifically, we aim to create two AI agents using Deep Q-Learning (DQN) [1] and Proximal Policy Optimization (PPO) [2], each trained to drive a car efficiently in a simulated environment with partial knowledge of its surroundings. The full code and results are available at the following GitHub link: [3].

1.2 Description of the environment

We place ourselves in the Gymnasium CarRacing-v3 discrete environment [4]. This involves learning a policy that maps observations of the environment (images representing the car's view) to actions (steering, acceleration, and braking) in order to navigate the track at high speed.

Observation space

The game environment consists of game frames, where each frame is a 96 x 96 x 3 array representing the RGB image from the car's perspective. Additionally, a black bar at the bottom of the frames displays the accumulated reward and the current actions of the car.

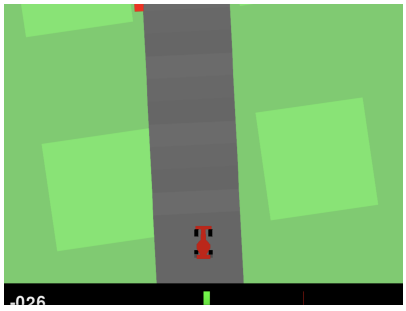

Figure 1: CarRacing-v3 environment

Action space

At each step, there are 5 possible actions: 0 = do nothing, 1 = full steer right, 2 = full steer left, 3 = full gas, and 4 = full brake.

Rewards and implied goal

The agent will receive reward -0.1 every frame and $+1000/N$ for every track tile visited, where N is the total number of tiles visited in the track. The goal is to get the highest accumulated reward (visit as many track tiles as possible) within 1000 frames.

Starting state and episode termination

The car starts at rest in the center of the road. Episode finishes when we reach 1000 frames, or when the car goes off track and dies with -100 reward.

2 DQN Agent

To solve the CarRacing-v3 environment with RL, the problem is first formulated as a Markov Decision Process (MDP), where outcomes are partly random and partly under the control of the agent. The goal is to discover an optimal policy π_*, which is a strategy for the agent that maximizes the expected cumulative reward R_t over time.

The goal of the DQN agent is to get the optimal action-value function (Q-function) given by:

$$Q_*(s,a) = \max_{\pi} \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$$

This function represents the expected return of taking a particular action a in a given state s. At each step, the agent selects the action that maximizes the expected return, according to the Q-function.

The DQN agent uses a deep neural network called Q-network, denoted $Q(s,a;\theta)$ where θ represents the parameters to be optimized, to approximate $Q_*(s,a)$. Moreover, Q-network will store the agent's experience at each time-step in replay buffer and randomly samples a subset of the experience for training.

DQN can handle large state space with raw sensory inputs, such as images or complex state representations. The target network provides a stable target for the online network to learn from, while experience replay reduces the correlation between consecutive samples and helps to break the temporal dependencies and stabilize the learning.

2.1 Preprocessing

We need to first formulated the problem as a Markov decision process (MDP) problem. However, we are given only one current game frame for each step, so this initial observation setting cannot satisfy the Markov property. We cannot guess if the car is moving forward or backward from only one frame, which means we cannot predict the next frame for given the current frame. Thus, we need to modify the environment to be able to satisfy this property.

1. **Modify env.reset() to always skip the first 50 frames**

   The game configuration always just gradually zooms in for the first 50 frames. Since this zoom-in phase is a very small part of the overall game, keeping this during training might hinder our agent from learning to control the car in the main frames after zoom-in. So we decide to skip the first 50 frames of the game, with the action 0 = do nothing.

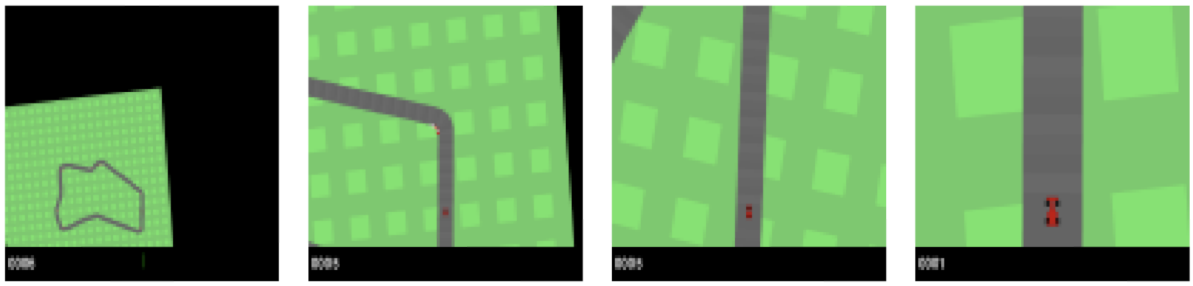

Figure 2: Beginning of the game, the frames go from the left to the right

2. **Convert to grayscale and crop a 60 × 60 image**

   We reduce the number of dimensions, from 96 x 96 x 3 to 60 x 60, to allow for more compact states and to make the training faster. Indeed, there is a black bar at the bottom of the frame that is unnecessary for describing the environment, and the close information of the car is the most important. Note that the initial observation is very pixelated compared to the game environment.

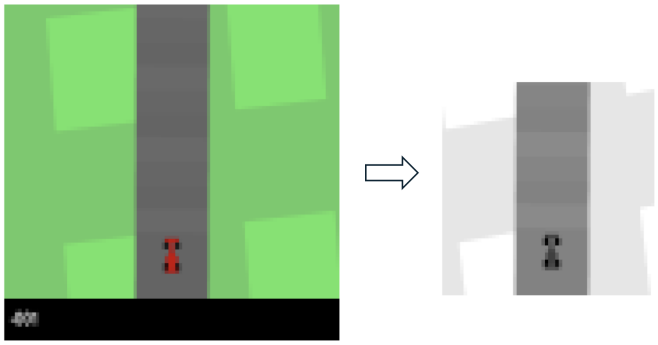

Figure 3: Example of grayscale preprocessing of CarRacing-v3 observation

3. **Modify env.step() to use frame skipping technique**

   Our agent will see and select actions every 4 frames instead of every frame, and its last action is repeated on skipped frames. Hence, we can transform each observation to contain 4 frames simultaneously so that the agent can know whether it is moving forward or backward. This will also help to decrease the training time since we only need 1 action per 4 frames.

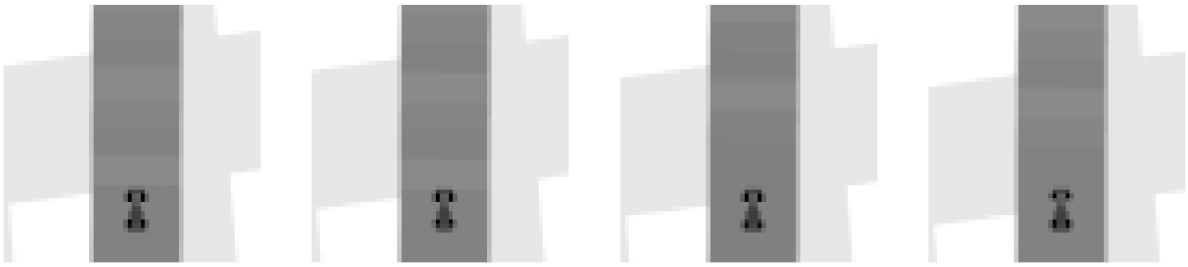

Figure 4: Example of one state of the new environment: we can see the car going forward

2.2 DQN Algorithm

Now that we have a MDP, we can apply DQN algorithm [1]. The core of the DQN method is encapsulated in its loss function for the training of the Q-network. The loss function $L_i(\theta_i)$ quantifies the difference between the predicted Q-value and the target Q-value. It's given by the following mean squared error:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(.)}[(y_i - Q(s,a;\theta_i))^2], \text{with target Q-value } y_i = \mathbb{E}_{s'\sim\epsilon}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a]$$

We then apply stochastic gradient descent to this loss function.

Algorithm 1 Deep Q-learning with Experience Replay [1]

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights θ
for episode $= 1$ to M **do**
  Initialize s_1
  for $t = 1$ to T **do**
    $a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \max_a Q^*(s_t, a; \theta) & \text{with probability } 1 - \epsilon \end{cases}$
    Execute action a_t, get state s_{t+1} and reward r_t
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta) & \text{for non-terminal } s_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ over θ
  end for
end for

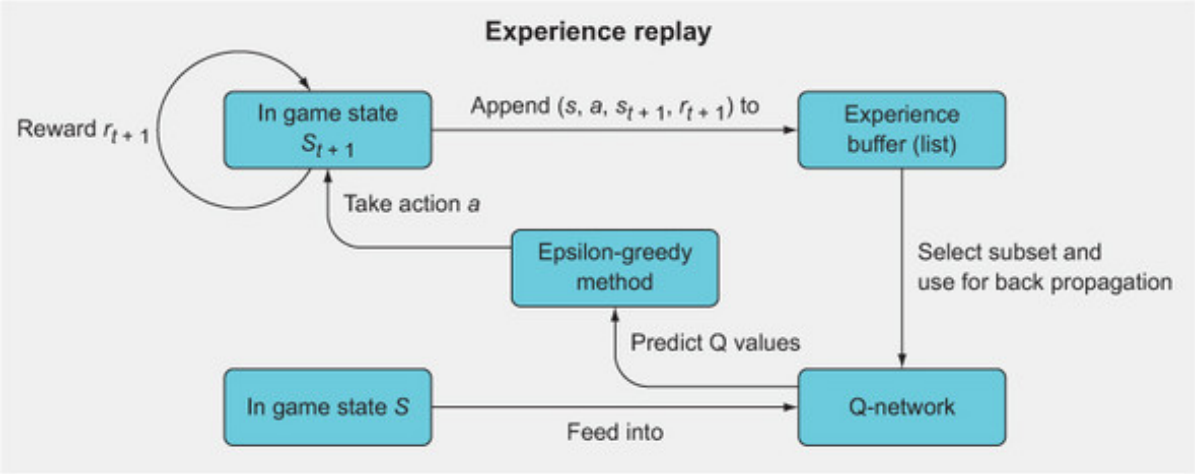

Figure 5: DQN Schema [5]

2.3 Architecture of the Q-network

Our Q-network takes as input arrays of size $60 \times 60 \times 4$ (state of 4 grayscaled frames). It has the following convolutional neural network (CNN) architecture:

– Conv Layer 1: 16 filters of size 8 x 8, stride 4, followed by a ReLU.

- Conv Layer 2: 32 filters of size 4 x 4, stride 2, followed by a ReLU.

- Flatten Layer: The output of the Conv layers is flattened to a 1D vector.

- FC Layer 1: A fully connected layer that produces a vector of length 256.

- FC Layer 2: A fully connected layer that produces a feature vector to be passed to the policy and value networks.

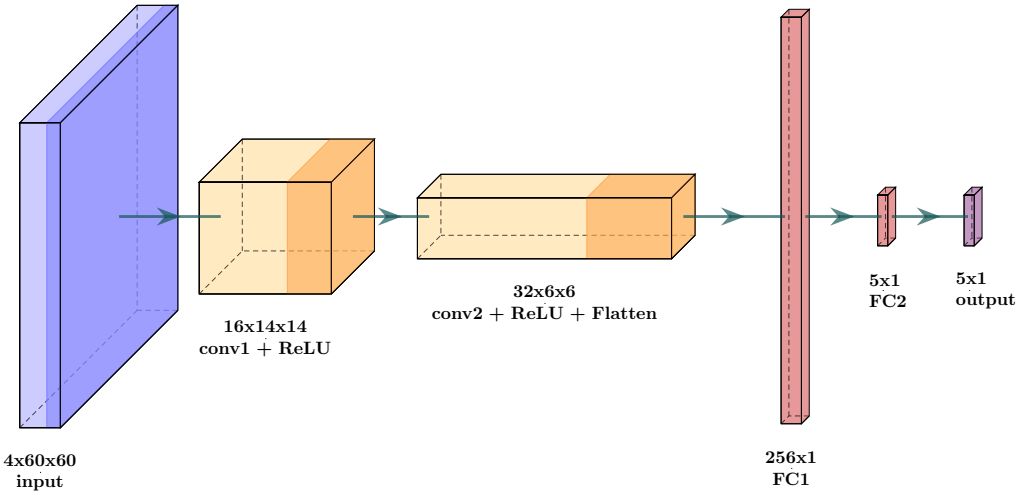

Figure 6: Schema of the Q-network architecture

2.4 DQN Result

We implement our DQN agent from scratch with the Q-network above, an experience replay buffer, and the training algorithm which saves the model and evaluates it every 1000 steps. We use an ϵ-greedy strategy with a decay rate to maximize exploration at the beginning of the training and gradually shift towards exploitation over the course of training.

The algorithm took 12 hours to perform 600,000 steps, with a final average return of 605. We used the RMSprop optimizer with a learning rate of 4×10^{-4}. The full training process is shown below in Figure 7. We can see that the average return value oscillates a lot during training, but has a global increasing trend.

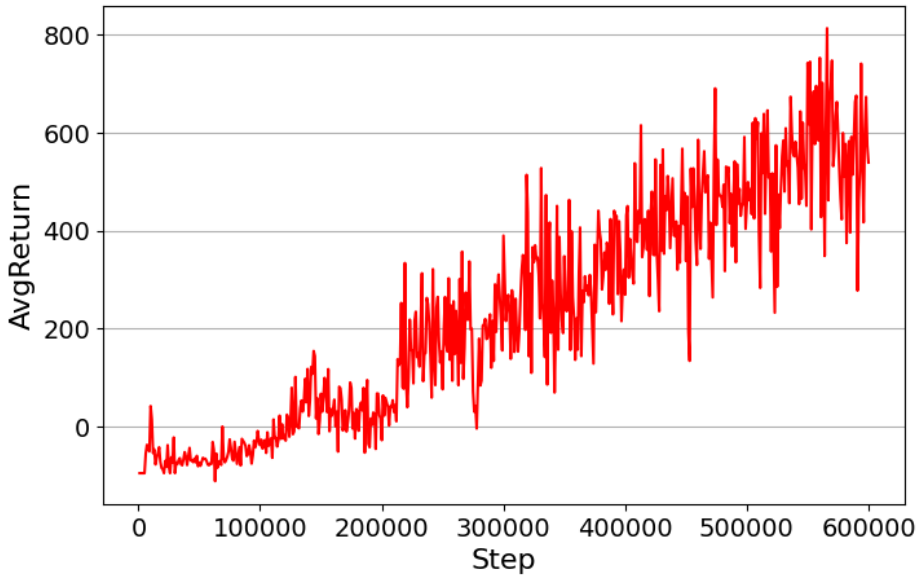

Figure 7: DQN training process

3 PPO Agent

Policy gradient methods compute an approximation of the policy gradient and integrate it into a stochastic gradient ascent approach [2]. The prevalent gradient estimator is typically formulated as:

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t)\, \hat{A}_t \right]$$

Here, π_θ represents a stochastic policy, and $\hat{A}_t$ is an estimate at time step t of the advantage function $A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$. The expectation $\hat{\mathbb{E}}_t[.]$ denotes the empirical average over a finite batch of samples. The estimator $\hat{g}$ is derived by differentiating the objective:

$$\mathcal{L}^{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t)\, \hat{A}_t \right]$$

While it may seem enticing to perform multiple optimization steps on this loss $\mathcal{L}^{PG}$ using the same trajectory, it often results empirically in excessively large policy updates.

3.1 TRPO and PPO

Trust Region Policy Optimization (TRPO) [2] maximizes the following objective while adhering to a constraint on the magnitude of the policy update:

$$\max_\theta \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\, \hat{A}_t \right], \text{ subject to } \hat{\mathbb{E}}_t \left[\mathrm{KL} \left[\pi_{\theta_{old}}(.|s_t), \pi_\theta(.|s_t) \right] \right] \leq \delta$$

Here, θ_{old} represents the vector of policy parameters before the update. This problem can be solved using the conjugate gradient algorithm, but we need for that to make a linear approximation of the objective function and a quadratic approximation to the constraint.

PPO, on the other hand, is a first-order algorithm that emulates the monotonic improvement of TRPO, by introducing a hyperparameter ϵ and optimizing the following clipped surrogate objective:

$$\mathcal{L}^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]$$

where $r_t(\theta) = \dfrac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio, and the clip function restricts $r_t(\theta)$ to the range $[1-\epsilon, 1+\epsilon]$. PPO maintains a lower bound on the unclipped objective, thus penalizing excessively large updates (see Figure 8). It offers simplicity in implementation, greater generality, better computational complexity and stability over TRPO.

3.2 PPO Algorithm

Algorithm 2 PPO, Actor-Critic Style [2]

for iteration $= 1, 2, \ldots$ **do**
  for actor $= 1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L with respect to θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

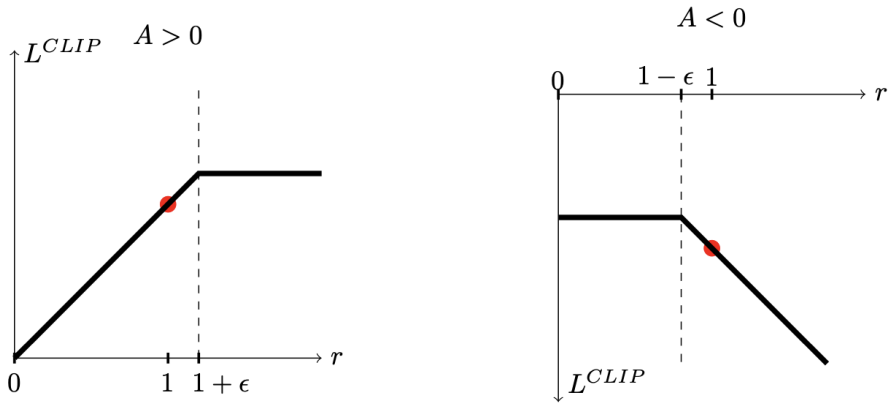

Figure 8: Plots showing one term (i.e. a single timestep) of $\mathcal{L}^{CLIP}$ as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., $r = 1$. Note that $\mathcal{L}^{CLIP}$ sums many of these terms.

3.3 PPO Result

We implement our PPO agent using the "stable_baselines3" library [6]. We train the agent with a CNN as its policy network, a learning rate of 10^{-4} (by default, the PPO uses the Adam optimizer) and a batch size of 32. The batch size determines how many experiences are sampled at each training step. The default CNN takes directly as input the 96 x 96 x 3 array observation, has the following architecture:

- Conv Layer 1: 32 filters of size 8x8, stride 4, followed by a ReLU.
- Conv Layer 2: 64 filters of size 4x4, stride 2, followed by a ReLU.
- Conv Layer 3: 64 filters of size 3x3, stride 1, followed by a ReLU.
- Flatten Layer: The output of the Conv layers is flattened to a 1D vector.
- Fully Connected Layer: A fully connected layer that produces a feature vector to be passed to the policy and value networks.

We took 3h to train the PPO agent in 600,000 steps, and evaluate it every 2000 steps. The final average return is 733. Thus we have reached a better performance than the DQN agent, in 4 times less time. We can also see in Figure 9 that the average return oscillates far less than DQN.

However, after 300,000 steps, the performance started suddenly to decrease, before increasing again after 400,000 steps. We suspected that this is due to a phenomenon called policy collapse, where as agent continues to interact with the environment, their performance degrades [7].

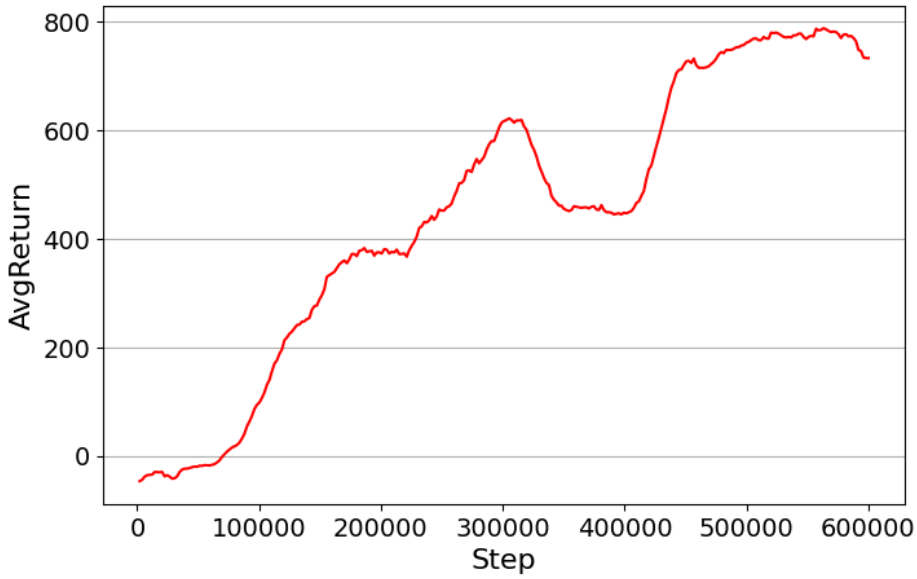

Figure 9: PPO training process

4 Conclusion

This project has highlighted the potential and effectiveness of the DQN and PPO algorithms for autonomous car navigation in a simulated environment. In our case, PPO appears to be the more promising algorithm, achieving better performance in a significantly shorter time. However, while the DQN's average return fluctuates considerably during training, it shows a general upward trend and offers a solid foundation. On the other hand, the PPO agent is more prone to policy collapse. It would be valuable to explore why this happens and whether there are opportunities to enhance the robustness of the PPO algorithm while maintaining its performance.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller (19 Dec 2013). *Playing Atari with Deep Reinforcement Learning*. DeepMind Technologies. Available at: https://arxiv.org/abs/1312.5602

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov (28 Aug 2017). *Proximal Policy Optimization Algorithms*. OpenAI. Available at: https://arxiv.org/abs/1707.06347

[3] A. Fu, R. Faure (28 Mar 2025). *Solving Car Racing v3*. GitHub. Available at: https://github.com/Adri4000/Solving_Car_Racing_v3

[4] Farama-Foundation (2023). *Gymnasium: A toolkit for developing and comparing reinforcement learning algorithms*. Available at: https://github.com/Farama-Foundation/Gymnasium

[5] A. Zai, B. Brown (2020). *Deep Reinforcement Learning in Action*. Chapter 3. Manning. Available at: https://livebook.manning.com/concept/deep-learning/q-network

[6] Raffin, A., et al. (2021). *Stable-Baselines3: A set of reliable implementations of reinforcement learning algorithms*. GitHub. Available at: https://github.com/DLR-RM/stable-baselines3

[7] S. Moalla, A. Miele, D. Pyatko, R. Pascanu, C. Gulcehre (20 Nov 2024). *No Representation, No Trust: Connecting Representation, Collapse, and Trust Issues in PPO*. CLAIRE EPFL, Google DeepMind. Available at: https://arxiv.org/pdf/2405.00662