HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

CAPSTONE PROJECT REPORT

CarRacing-v3 Algorithm Analysis and Optimization

Group 15

NGUYEN VIET ANH - anh.nv235894@sis.hust.edu.vn
NGUYEN TUAN DUC - duc.nt235916@sis.hust.edu.vn
PHAM DUC CUONG - cuong.pd235904@sis.hust.edu.vn
CAO CHI CUONG - cuong.cc235902@sis.hust.edu.vn
TRINH HONG ANH - anh.ht235896@sis.hust.edu.vn

Supervisor: Associate Professor Thân Quang Khoát

School: School of Information and Communications Technology

Contents

1 Introduction

1.1 Real-world Problem Motivation

The CarRacing-v3 environment [1] simulates the challenges encountered in real-world autonomous driving, particularly in dynamic and uncertain road conditions. In the era of self-driving technologies, developing intelligent agents capable of perceiving the environment, making real-time decisions, and executing smooth control actions is essential. Modern autonomous vehicles must be able to interpret high-dimensional sensory inputs, such as camera images, and map them effectively to low-level control commands like steering, acceleration, and braking. This closely resembles the setup in CarRacing-v3, where an agent receives raw pixel observations and must navigate a procedurally generated racetrack without prior knowledge of the track layout.

Additionally, safe and efficient control strategies are critical for applications such as autonomous delivery, robotics in agriculture, and mobile robot navigation in urban environments. Testing reinforcement learning algorithms in virtual environments like CarRacing-v3 allows researchers to experiment with various learning strategies without the risks and costs associated with real-world testing. This environment also provides a simplified yet powerful testbed for evaluating the generalization and robustness of reinforcement learning agents under variable terrain, visual occlusion, and delayed feedback all of which are fundamental challenges in autonomous systems.

1.2 Problem Description

Environment: CarRacing-v3 is a continuous control environment provided by Gymnasium, designed to test an agent's ability to navigate a procedurally generated 2D racetrack. Each episode features a new track layout, requiring the agent to generalize its driving policy beyond fixed scenarios. The primary objective is to complete the track by covering as many tiles as possible while minimizing off-track behavior.

Objective: Solve the problem by developing an agent that navigates the car around the track as efficiently as possible, **minimizing time off-track** and **maximizing the number of tiles covered**.

Observation space: the observation space is an image of the race track represented as an RGB pixel matrix: `Box(0, 255, (96, 96, 3), uint8)`

- `Box(0, 255)`: Pixel values range from 0 to 255 (standard 8-bit for color images).
- `(96, 96, 3)`: The input image has a resolution of **96x96 pixels** with **3 color channels (RGB)**.
- `dtype=uint8`: Data type is 8-bit unsigned integers.

Action Space:

- If continuous there are 3 actions
  + 0: steering, -1 is full left, +1 is full right
  + 1: gas
  + 2: braking

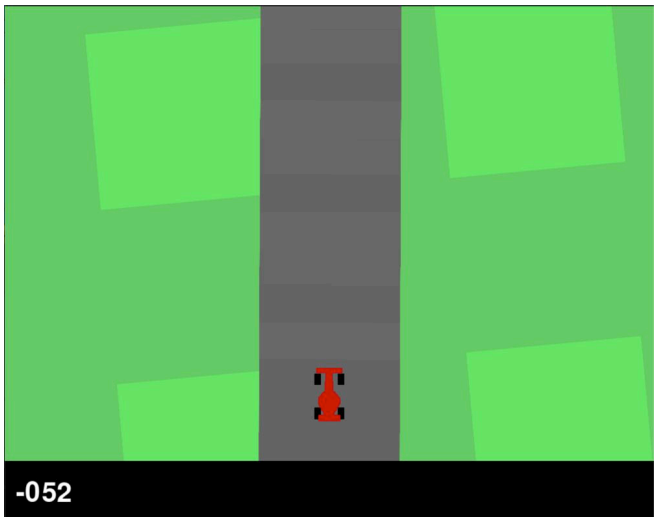


Figure 1: CarRacing game environment

- If discrete there are 5 actions:
  - + 0: do nothing
  - + 1: steer left
  - + 2: steer right
  - + 3: gas
  - + 4: brake

Reward: The reward system in CarRacing primarily incentivizes the agent to stay on the track by awarding positive rewards proportional to the number of track tiles it covers. The reward is -0.1 every frame and $+1000/N$ for every track tile visited, where N is the total number of tiles visited in the track. For example, if you have finished in 732 frames, your reward is $1000 - 0.1 * 732 = 926.8$ points.

Starting state: The car start at rest in the center of the road.

Episode termination: The episode finishes when all the tiles are visited. The car can also go outside the playfield - that is, far off the track, in which case it will receive -100 reward and die.

1.3 Work Accomplished in the Project

Addressing the challenge of controlling a vehicle in the CarRacing-v3 environment, which requires processing image inputs and handling both discrete and continuous actions, this report focuses on the application, evaluation, and analysis of a modern deep reinforcement learning method. Our key works are as follows:

1. Implementing and successfully training a control agent using the Deep Q-Learning (DQN), Advantage Actor-Critic (A2C) and Proximal Policy Optimization (PPO) algorithms directly from raw pixel inputs in the CarRacing-v3 environment.

2. Evaluating the performance of the trained agent using quantitative metrics such as the average score achieved over multiple runs, score stability.

3. Analyzing the impact of key hyperparameters (e.g., learning rate, batch size, entropy coefficient) on the convergence process and final performance of the proposed algorithms within the CarRacing-v3 environment.

The contribution of this work, which we hope will renew interest in this class of methods and lead to new useful applications in solving autonomous driving tasks in simulated environments.

2 Preliminaries

2.1 Reinforcement Learning

Reinforcement Learning (RL) [2] is a subfield of machine learning concerned with how agents should take actions in an environment in order to maximize cumulative reward. It models the learning process of an intelligent agent that interacts with its environment by observing states, performing actions, and receiving scalar feedback in the form of rewards. Unlike supervised learning, RL does not rely on labeled input-output pairs; instead, it learns from trial and error. The ultimate goal of RL is to learn a reward-maximizing policy within a **Markov Decision Process** by mapping states to actions based on long-term outcomes.

A **Markov Decision Process (MDP)** is defined as the 5-tuple (S, A, P, R, γ):

$$M = (S, A, P, R, \gamma)$$

- S is a finite (or countably infinite) set of all **possible states**.
- A is a finite (or countably infinite) set of all **possible actions**.
- $P : S \times A \to P(S)$ is the **state transition probability function**. In detail, upon taking any action $a \in A$ in any state $s \in S$, $P(\cdot \mid s, a)$ defines the probability distribution of the next state.
- $R : S \times A \to \mathbb{R}$ is the **reward function**, which assigns a numerical reward for taking action a in state s.
- $\gamma \in [0, 1]$ is the **discount factor**, which controls how much future rewards are valued relative to immediate rewards.

2.2 Policy

$\pi : S \to \mathcal{P}(A)$ map each state $s \in S$ to a probability distribution over the set of actions A. In short, the **policy** $\pi(s)$ answers the question:

"Given the current state s, what action should the agent take?"

Starting from an initial state $S_0 = s$, the interaction between the agent and the environment evolves over time as a sequence of triplets:

$$\{(A_t, R_t, S_{t+1})\}_{t=0}^{\infty}$$

This sequence follows these rules:

$$A_t \sim \pi(\cdot|S_t), \quad R_t \sim R(S_t, A_t), \quad S_{t+1} \sim P(\cdot|S_t, A_t)$$

- A_t (action at time t) is sampled according to the policy π, given the current state S_t.

- R_t (reward) is sampled based on the environments **reward function**, depending on S_t and A_t.
- S_{t+1} (next state) is sampled based on the environments **transition probability** P, given the current state and action.

The **total return** (also called the cumulative reward) from a trajectory τ is defined as:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

- r_t is the reward at time t.
- $\gamma \in [0, 1]$ is the **discount factor** γ.

In this case, the **expected return** (for whichever measure), denoted by $J(\pi)$, is then:

$$J(\pi) = \int P(\tau|\pi)R(\tau) = \mathbb{E}_{\tau\sim\pi}[R(\tau)].$$

The primary objective in most RL problems is to find an **optimal policy**, denoted as π^*. Thus, The central optimization problem in RL can then be expressed by

$$\pi^* = \arg\max_{\pi} J(\pi),$$

with π^* being the optimal policy.

2.3 Value Functions [3]

2.3.1 The On-Policy Value Funtion

Measures how good it is to be in state s, assuming the agent follows policy π from that state onwards. It's the expected total discounted reward starting from state s.

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R(\tau)|S_0 = s]$$

2.3.2 The On-Policy Action-Value Funtion

Measures how good it is to take a specific action a when in state s, and then follow policy π thereafter. It's the expected total discounted reward after taking action a in state s and subsequently following π.

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[R(\tau)|S_0 = s, A_0 = a]$$

2.3.3 The Optimal Value Function

Measures the maximum possible expected total discounted reward achievable starting from state s, assuming the agent acts optimally (i.e., follows the best possible policy) from that state onwards. This represents the true, best possible value of being in state s.

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

2.3.4 The Optimal Action-Value Function

Measures the maximum possible expected total discounted reward achievable starting from state s, taking action a first, and then acting optimally thereafter. This represents the true, best possible value of taking action a in state s.

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$$

2.4 Bellman Equations

These are fundamental equations in RL that express a recursive relationship between the value of a state (or state-action pair) and the values of its successor states (or state-action pairs). They provide a way to compute or learn the value functions based on self-consistency with a core idea:

"The value of your current situation equals the immediate reward you get plus the discounted value of the situation you expect to land in next."

Bellman Equation for the On-Policy Value Function:

$$V^{\pi}(s) = \mathbb{E}_{A\sim\pi(\cdot|s),S'\sim P(\cdot|s,A)}\left[R(s,A) + \gamma V^{\pi}(S')\right]$$

Relates the value of state s under policy π to the expected value of the next state S', considering the actions A chosen by π and the environment's transitions P.

The Bellman equation for the on-policy action-value function:

$$Q^{\pi}(s,a) = \mathbb{E}_{S'\sim P(\cdot|s,a),A'\sim\pi(\cdot|S')}\left[R(s,a,S') + \gamma Q^{\pi}(S',A')\right]$$

The Bellman equations for the optimal value functions are:

$$V^{*}(s) = \max_{a} \mathbb{E}\left[R(s,a) + \gamma V^{*}(S')\right]$$

$$Q^{*}(s,a) = R(s,a) + \gamma \mathbb{E}_{S'}[V^{*}(S')]$$

3 Approaches

3.1 Deep Q-Network (DQN) [4]

DQN approximates the Q-function with a deep neural network, i.e. $Q(s,a;\theta) \approx Q^{*}(s,a)$. Similar to the functionality of Q-table in Q-learning, Q-network takes in a state as input and outputs the predicted Q-values for each action, given a state. Then, it will store the agents experience at each time-step in replay buffer and randomly samples a subset of the experience for training.

Full algorithm

The core of the DQN algorithm 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 mean squared error:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right],$$

with target Q-value:

$$y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,a\right]$$

To optimize the Q-function, stochastic gradient descent, namely RMSprop, is applied to the loss function (equation 3 in algorithm below):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

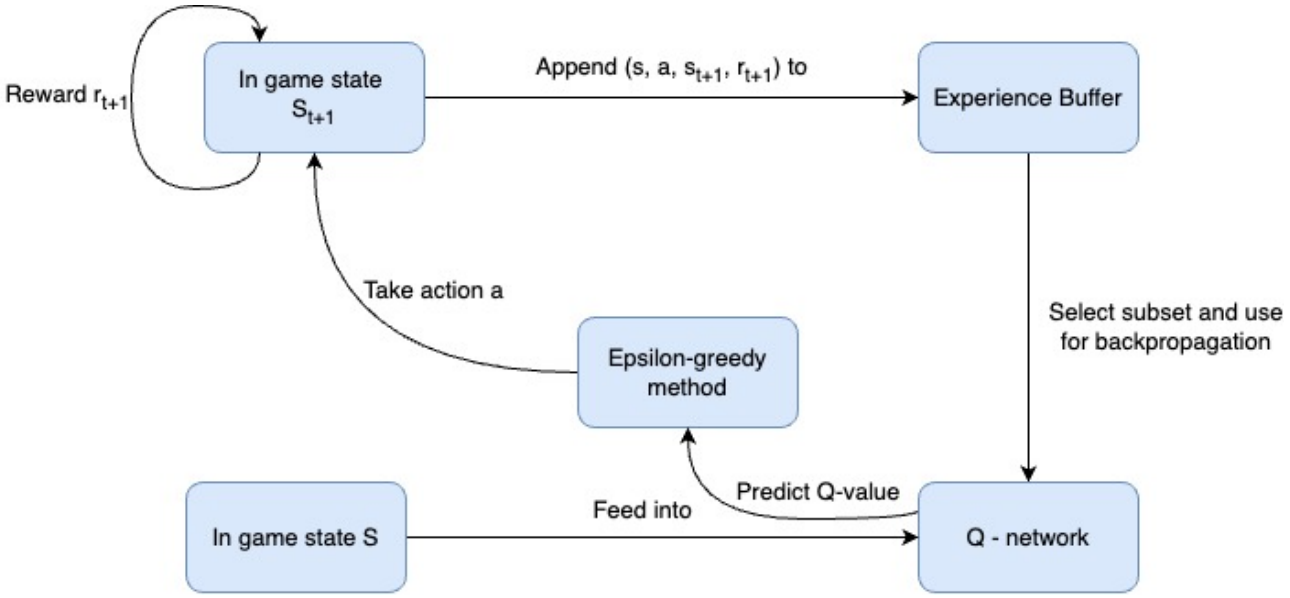


Figure 2: Experience Replay Mechanism in Deep Q-Learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights θ
for episode = 1, M **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \max_a Q(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to Equation 3
    end for
end for

Advantages of DQN

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.

3.2 Advantage Actor-Critic [5]

The Advantage Actor-Critic (A2C) algorithm enhances the basic Actor-Critic (AC) framework within reinforcement learning by combining policy-based action selection with value-based evaluation. In this improved method, an actor component learns and executes the policy, while a critic component learns a value function to provide evaluative feedback on the actor's performance.

Theoretical basis

1. **Synchronous Execution:** A key characteristic of A2C is its synchronous execution model, often involving multiple agents collecting data concurrently in parallel environments, followed by coordinated, simultaneous updates to the shared network parameters.

2. **Advantage Function:** A2C leverages the advantage function primarily to enhance training stability. This is achieved because the advantage measure helps mitigate the high variance commonly associated with policy gradient estimates.

3. **Policy and Value Updates:** A2C optimizes two objectives simultaneously:

   - **The policy objective (actor):** Improves the action policy.
   - **The value objective (critic):** Improves the value estimation of states.

Methodology

1. **Policy Update (Actor)**
   The actor is updated to maximize the expected return $J(\theta)$ represented as:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[\log \pi_\theta(a|s) A^\pi(s,a)]$$

   Advantage function:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

2. **Value Update (Critic)**
   The critic is trained to minimize the **mean squared error (MSE)** of the value function:

$$L(\phi) = \frac{1}{2}\mathbb{E}[(R_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t))^2]$$

   Where:

   - $V_\phi(s_t)$: Estimated value of the state s_t using parameters ϕ.

3. **Entropy Regularization**
   To encourage exploration, A2C adds an entropy term to the loss function, defined as:

$$H(\pi) = -\sum_a \pi(a|s) \log \pi(a|s)$$

   This term ensures the policy does not become too deterministic early in training.

4. **Final Loss Function**
   The total loss is represented as:

$$L_{\text{total}} = L_{\text{actor}} + c_1 L_{\text{critic}} - c_2 H(\pi)$$

   Where:

   Actor loss: Maximize the expected return by adjusting the policy parameters θ. It utilizes the advantage function $A(s,a)$, which measures how much better or worse an action is compared to the average action in a given state:

$$L_{\text{actor}} = -\mathbb{E}[\log \pi_\theta(a|s) \cdot A(s,a)]$$

   Critic loss: Minimize the Mean Squared Error (MSE) between the predicted value $V(s)$ and the TD (temporal difference) target $R_t + \gamma V(s_{t+1})$:

$$L_{\text{critic}} = \frac{1}{2}\mathbb{E}[(R_t + \gamma V(s_{t+1}) - V(s_t))^2]$$

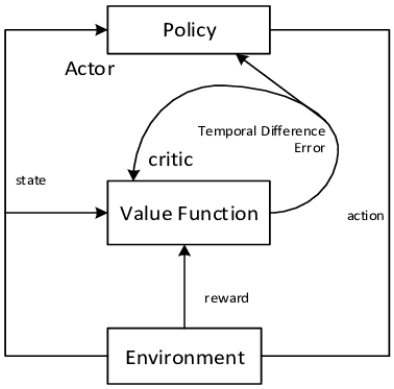


Figure 3: Fundamental Actor-Critic Architecture (Basis for A2C)

Algorithm 2 TD Advantage Actor-Critic

Randomly initialize critic network $V_\pi^U(s)$ and actor network $\pi^\theta(s)$ with weights U and θ
Initialize environment E
for episode = 1 to M **do**
    Receive initial observation s_0 from E
    for $t = 0$ to T **do**
        Sample action $a_t \sim \pi(a|\mu, \sigma) = \mathcal{N}(a|\mu, \sigma)$ according to current policy $\pi^\theta(s_t)$
        Execute action a_t and observe reward r and next state s_{t+1} from E
        Set TD target $y_t = r + \gamma \cdot V_\pi^U(s_{t+1})$
        Update critic by minimizing loss: $\delta_t = (y_t - V_\pi^U(s_t))^2$
        Update actor policy by minimizing loss: $Loss = -\log(\mathcal{N}(a|\mu, \sigma)) \cdot \delta_t$
        Update $s_t \leftarrow s_{t+1}$
    end for
end for

Advantage of A2C

A2C improves training stability by combining policy and value learning, reducing variance with advantage estimation. It's simple to implement, works in both discrete and continuous action spaces, and forms the basis for more advanced methods like PPO.

3.3 Proximal Policy Optimization [6]

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm designed to balance between efficient learning and stability. It is built on policy gradient methods and it uses a clipped surrogate objective to prevent bad updates to the policy, ensuring steady improvement. PPO is simpler to implement than earlier methods like Trust Region Policy Optimization (TRPO) and achieves comparable or better performance in various environments. It is widely used for its robustness and scalability in solving complex decision-making problems.

3.3.1 Introduction to Policy Optimization [7]

Methods in this family represent a policy explicitly as $\pi(a|s)$. They optimize the parameters θ either directly by gradient ascent on the performance objective $J(\theta)$, or indirectly, by maximizing local approximations of $J(\theta)$. This optimization is almost always performed on-policy, which means that each update only uses data collected while acting according to the most recent version of the policy. Policy optimization also usually involves learning an approximator $V(s)$ for the on-policy value function $V(s)$, which gets used in figuring out how to update the policy.

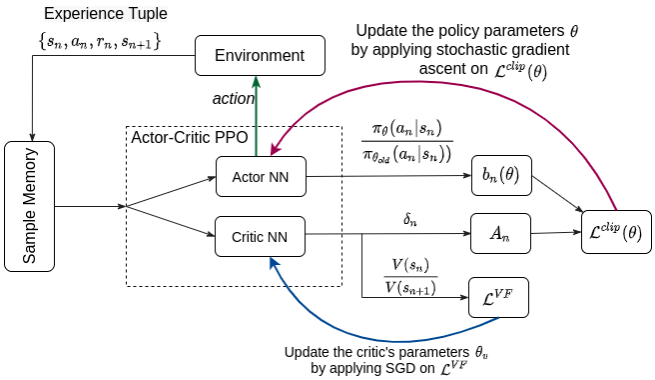


Figure 4: Detailed Diagram of the Proximal Policy Optimization (PPO) Algorithm

3.3.2 Proximal Policy Optimization

In some cases in earlier policy optimization algorithms, when taking the biggest possible improvement step on a policy using the data we currently have, we can step so far and accidentally cause performance collapse. There are several earlier algorithms that try to solve this problem but it is complex to implement these algorithms. PPO methods are simpler to implement and perform well in continuous action space. There are two primary variants of PPO: PPO-Penalty and PPO-Clip. In our project we will use PPO-Clip.

Actor-Critic model's structure: PPO uses the Actor-Critic approach for the agent. This means that it uses two models, one called the Actor and the other called Critic. The Actor model performs the task of learning what action to take under a particular observed state of the environment. In the CarRacing-v3 case, it takes an RGB pixel matrix of the game as input, which represents the current state of our car and gives a particular action to drive the car as output. In our algorithm, we are using layers of Convolution Neural Network. The last layer is the probabilistic distribution layer, which return a Normal distribution of the actions. The Critic model has the same architecture as the Actor model but it has the role of evaluating if the action taken by the Actor led our environment to be in a better state or not and give its feedback to the Actor.

Implementation details: In PPO, the advantage function plays a critical role in estimating how much better an action is compared to the average performance of the policy in a given state, The advantage function $\hat{A}_t$ at time step t is defined as:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \dots$$

Where:

- $\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$ is the **Temporal Difference (TD) error**.
- γ: Discount factor for future rewards (e.g., 0.99).
- λ: GAE parameter for balancing bias-variance tradeoff (e.g., 0.95).

This approach is called **Generalized Advantage Estimation (GAE)**.
Because of using a neural network architecture that shares parameters between the policy and value function, we must use a loss function that combines the policy surrogate and a value function error term. This objective can further be augmented by adding an entropy bonus to ensure sufficient exploration. Combining these terms, we obtain the following objective, which is (approximately) maximized each iteration:

$$L^{\text{CLIP+VF+S}}(\theta) = \mathbb{E}_t\left[L^{\text{CLIP}}(\theta) - c_1 L^{\text{VF}}(\phi) + c_2 S[\pi_\theta](s_t)\right]$$

where c_1, c_2 are coefficients, S denotes an entropy bonus, and L^{VF} is a squared-error loss $(V_\phi(s_t) - V_t^{\text{targ}})^2$.

Clipped Surrogate Loss:

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

Where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$: the probability ratio between the new policy and the old policy.
- $\hat{A}_t$: the advantage function at time step t, which estimates how much better an action a_t is compared to the baseline (value function).
- ϵ: a small hyperparameter (e.g., 0.1 or 0.2) that controls the clipping range.
- $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$: ensures that the probability ratio $r_t(\theta)$ does not move too far from 1.

Value Function Loss: The value function loss minimizes the error between the predicted value $V_\phi(s_t)$ and the actual return R_t:

$$L^{\text{VF}}(\phi) = \mathbb{E}_t \left[(V_\phi(s_t) - R_t)^2 \right]$$

where:

- $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$: Discounted cumulative return.

Entropy Regularization: The entropy bonus encourages exploration by preventing the policy from becoming too deterministic:

$$S[\pi_\theta](s_t) = -\sum_a \pi_\theta(a|s_t) \log \pi_\theta(a|s_t)$$

Algorithm 3 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ do **do**
    Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute rewards-to-go $\hat{R}_t$.
    Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on
    the current value function V_{ϕ_k}.
    Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right)$$

    typically via stochastic gradient ascent with Adam.
    Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} (V_\phi(s_t) - \hat{R}_t)^2$$

    typically via some gradient descent algorithm.
end for

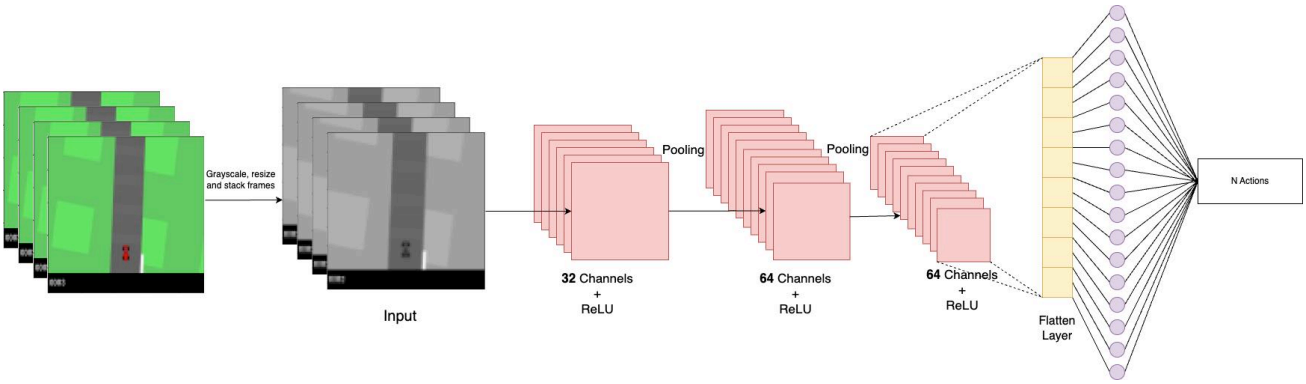


Figure 5: Feature extractor pipeline

Advantage of PPO

PPO offers simplicity in implementation, greater generality, and improved empirical stability and data efficiency. Notably, it performs well on continuous action spaces (where DQN struggles) and doesnt require extensive hyperparameter tuning.

4 Experiments setup and results

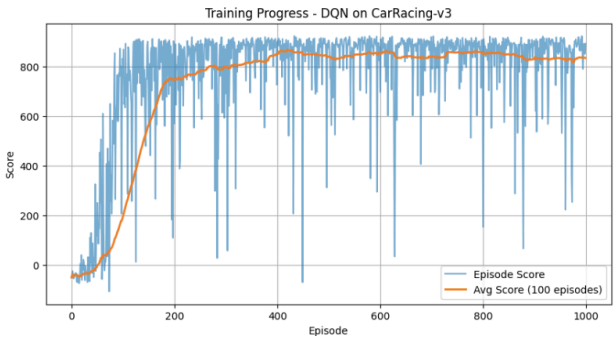
Evaluation Metrics: For evaluation metrics we use the average rewards obtained after the latest 100 episodes.

Preprocessing: During the training process on the `CarRacing-v3` environment, the input data needs to be preprocessed [8] to reduce noise, decrease data complexity, and enhance the model's learning efficiency. First, the original RGB frames are converted to grayscale to eliminate unnecessary color information, which reduces the dimensionality of the input while retaining essential geometric features such as the track layout and objects. Then, the images are resized to a standardized dimension of 84×84 to ensure consistency in input size and reduce computational costs during training. Next, pixel values are normalized to the range $[0, 1]$ by dividing them by 255, which helps stabilize the learning process of the neural network. Finally, to enable the model to capture motion and temporal context, four consecutive frames are stacked together to form a single input state. This allows the model to infer speed and direction of movement information that is not apparent in a single frame. These preprocessing steps are integrated into the `stack_frames()` function, which prepares the input in a format suitable for the network, typically resulting in a tensor of shape $(4, 84, 84)$. The preprocessing pipeline show in Figure 5 contributes to faster convergence, more stable training, and improved performance.

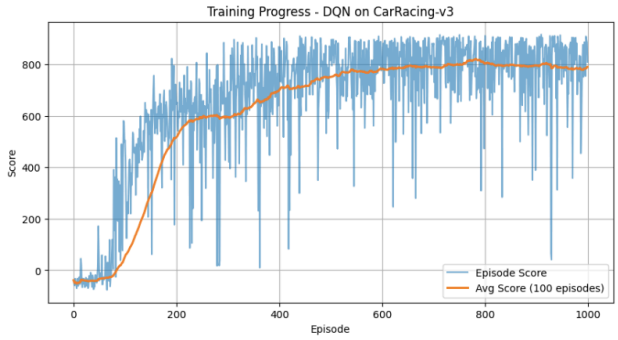
4.1 Deep Q-Learning Network

First, We solve the discrete action space environment by trying to tune the hyperparameters and find that the model works pretty well when we set:

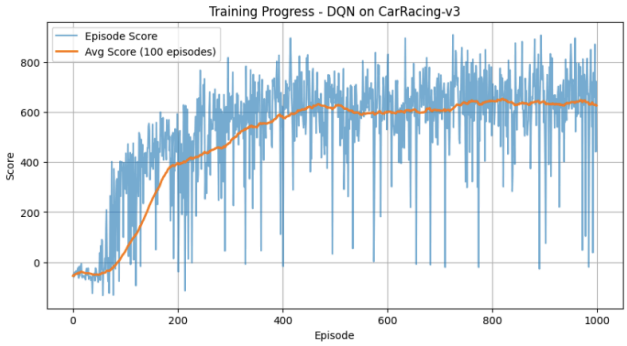
```
BATCH_SIZE = 32           # Number of transitions sampled from replay buffer
GAMMA = 0.99              # Discount factor for future rewards
EPS_START = 0.9           # Starting value of epsilon (exploration rate)
EPS_END = 0.05            # Minimum value of epsilon
EPS_DECAY = 10000         # Controls the rate of epsilon decay
TARGET_UPDATE_FREQ = 1000 # How often to update the target network (in steps)
LEARNING_RATE = 1e-4      # Learning rate for the optimizer
```



(a) Batch 32



(b) Batch 64



(c) Batch 128

Figure 6: DQN agent performance with different batch sizes.

We try to figure out how the batch size affects the result:

Batch Size	Mean Rewards	Mean episodes to reach over 800 score
32	862.02	268
64	806.23	728
128	651.36	1000+

Table 1: Effect of batch size on the DQN model performance.

Based on the experiments with DQN for this task, a batch size of 32 achieved the most effective results. With this configuration, the agent demonstrated rapid learning, achieving stable mean rewards consistently above 800 (considered successful for this task) relatively early, around the 300-320 episode mark. This performance remained stable thereafter, reaching a peak average reward of approximately 862, with the highest individual episode scores reaching up to 930.

In contrast, increasing the batch size to 64 led to degraded performance. The agent required significantly more training time, only reaching the 800 mean reward level around episodes 700-750. Furthermore, the performance at this level was less stable, fluctuating around the 800 mark (sometimes dipping to 780-790) rather than maintaining it consistently. The overall learning trajectory was noticeably slower compared to using a batch size of 32.

Using a batch size of 128 resulted in even poorer outcomes. The agent failed to achieve a mean reward of 800 even after 1000 episodes, reaching a maximum average reward of only around 650 and hovering near that level. The learning progress with this batch size was also considerably slower than with a batch size of 32.

The graphs are shown in Figure 6. It seems that for this DQN setup and task, making the

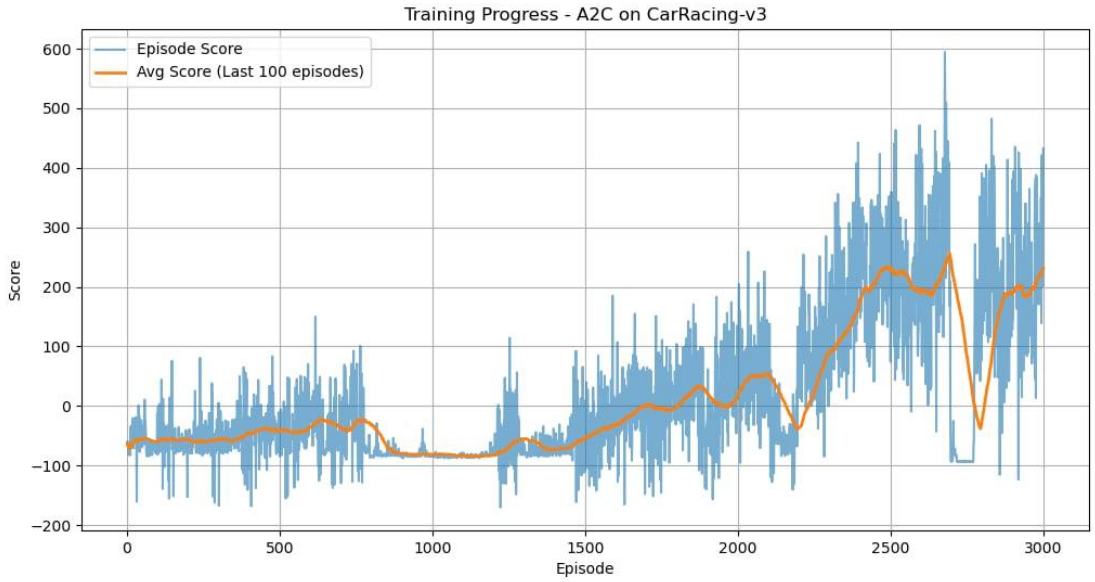


Figure 7: A2C agent performance

batch size bigger actually makes the agent learn less effectively and slower. This might be surprising, but it's likely due to a few reasons: When the batch size is large, the agent's 'brain' (the network) gets updated less often for the same amount of gameplay, which can slow down how quickly it learns and adjusts. Also, the 'shakiness' that comes from smaller batches (like size 32) can actually help the agent avoid getting stuck in 'okay' but not 'great' solutions. Finally, the learning speed setting might be just right for the small batch size, but not work as well (perhaps being too slow) when the batch size gets bigger, making it harder to reach the best scores.

4.2 Advantage Actor-Critic

In order to solve the continuous action space, we also try to implement Advantage Actor-Critic on this problem with the hyperparameters below:

```
ROLLOUT_LENGTH = 1000 # Size of buffer for collected data phase
ALPHA = 0.0001        # Learning rate
GAMMA = 0.99          # Discount factor
VF_COEF = 0.5         # Value function loss coefficient
ENT_COEF = 0.01       # Entropy bonus coefficient (reduced)
```

One of the key challenges we have faced when applying the A2C algorithm to the CarRacing environment lies in the complexity of continuous control. The action space in CarRacing consists of three continuous parameterssteering, gas, and brakewhich significantly increases the difficulty of learning an optimal policy. Moreover, A2C suffers from high variance and lacks mechanisms to regulate policy updates, which can lead to unstable learning, especially in complex or continuous control tasks. These factors combined make A2C sensitive to hyperparameter choices in this environment. Figure 7 describes the reward trajectory. We conclude that this approach is inappropriate for this environment.

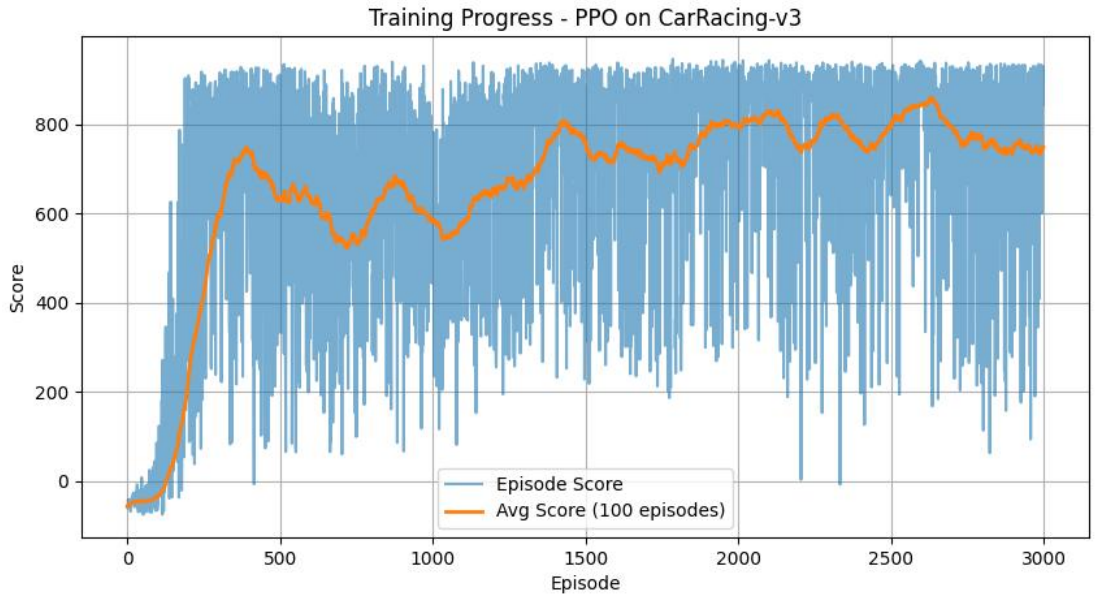


Figure 8: PPO agent performance when turning on LR decay

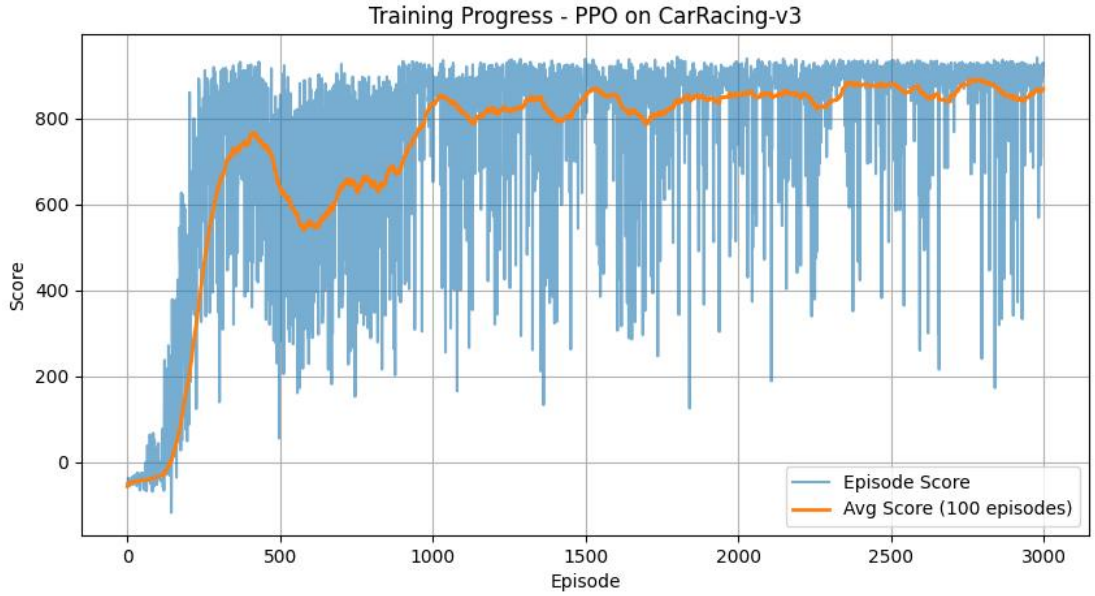


Figure 9: PPO agent performance when not turning on LR decay

4.3 Proximal Policy Optimization

Since Advantage Actor-Critic is not feasible for this problem because of its unstable update and sensitivity to hyperparamters, we come up with a more appropriate and advanced approach by implementing Proximal Policy Optimization. This alternative not only aims to provide a better solution for the task but also serves as a basis for performance comparison.

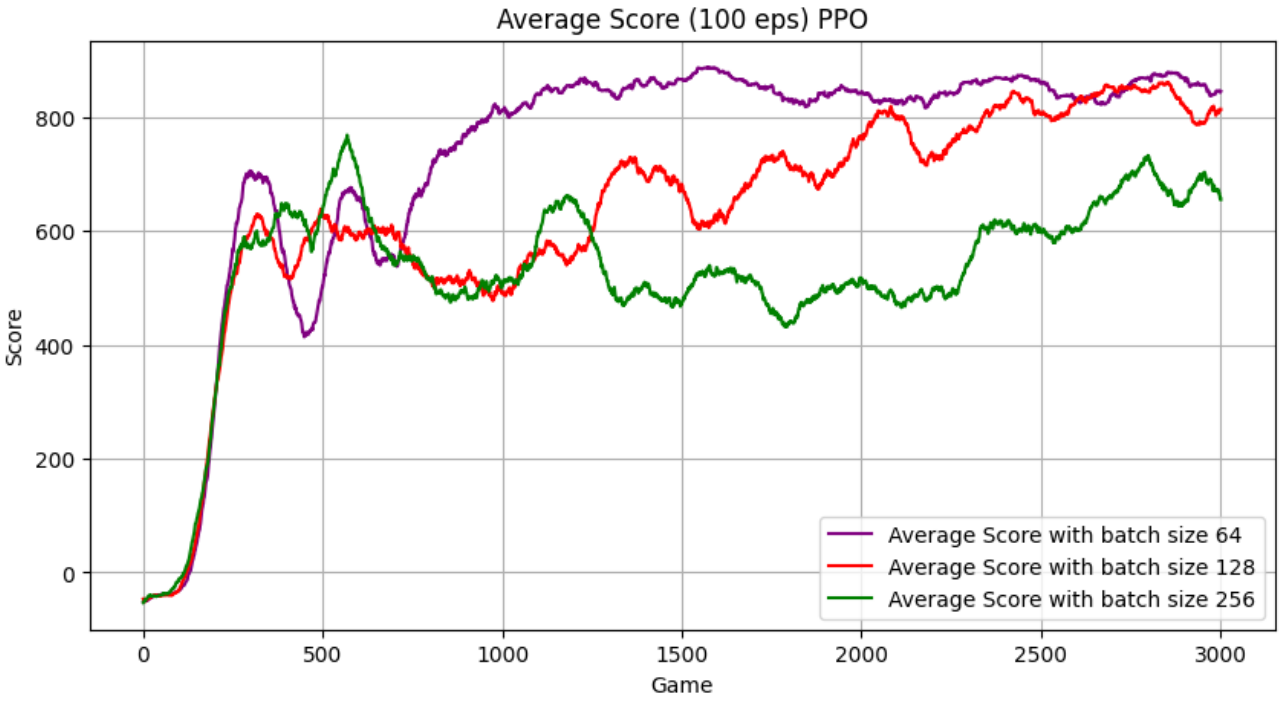


Figure 10: PPO agent performance with different batch sizes

```
N = 8192                          # Steps collected per policy update
BATCH_SIZE = 128                  # Mini-batch size for SGD updates
N_EPOCHS = 4                      # Optimization epochs per data rollout
INITIAL_ALPHA = 1e-4              # Initial learning rate
GAMMA = 0.99                      # Discount factor
GAE_LAMBDA = 0.95                 # Lambda for GAE.
POLICY_CLIP = 0.2                 # PPO clipping parameter.
VF_COEF = 0.5                     # Value Function Coefficient
ENT_COEF = 0.001                  # Entropy Coefficient
USE_LR_DECAY = True               # Enable linear learning rate decay?
FINAL_LR_FACTOR = 0.05            # Final learning rate if decaying.
DECAY_TOTAL_LEARN_ITERS = 1000 # Total iterations for LR decay.
NUM_EVAL_EPISODES = 5             # Episodes run during evaluation.
```

We try to figure out how the result is affected when applying the learning rate decay technique. Based on the experiments with the PPO agent for this task, the use or non-use of a learning rate decay (LR decay) mechanism seemingly did not produce a significant difference in overall performance, as illustrated in Figure 8 and Figure 9.

With the configuration enabling LR decay (Figure 8), the agent demonstrated strong learning capabilities. It rapidly improved its score, achieving a stable average score (Avg Score) above the 750-800 range (which can be considered successful for this task) relatively early, around the 300-500 episode mark. This high level of performance was subsequently maintained stably, with the average score primarily fluctuating between 750 and 850, occasionally peaking near the 850 mark. Also, when turning on LR decay, the model can overcome the local optimal around the first 500 episodes - which we were struggling with at first. Similarly, when not using LR decay (Figure 9), the PPO agent also exhibited an almost identical learning trajectory. The agent also reached the 750-800 average score threshold within the first 500 episodes. Following this initial convergence phase, performance remained stable within the same average score range of 750 to

850, mirroring the behavior observed in the agent using LR decay. The overall learning curve and achieved scores did not show a noticeable difference compared to the case with LR decay.

Batch Size	Mean Rewards
64	887.23
128	861.14
256	660.44

Table 2: Effect of batch size on the PPO model performance.

After having done the experiments with PPO learning rate decay, we come to the conclusion that the initial learning rate has already been sufficiently small and well-optimized for the problem. Moreover, CarRacing-v3 task itself are inherently insensitive to the presence or absence of LR decay within the tested parameter range, or perhaps a longer training duration would be required to reveal more subtle differences.

We further investigate the effect of varying batch sizes (64, 128, 256) on the performance of the PPO agent in the CarRacing-v3 environment. As shown in Figure 10, all configurations eventually reached satisfactory performance levels; however, notable differences were observed in convergence speed and stability.

With batch size 64, the agent achieved rapid performance gains, surpassing the 750 average score threshold within the first 300 episodes. It maintained a consistently high score (800-850) with minimal fluctuation, suggesting fast and stable learning.

Using batch size 128, the agent demonstrated slightly slower convergence (400 episodes) but still reached comparable final performance. The learning curve was more oscillatory, reflecting a balance between stability and responsiveness.

For batch size 256, learning was slower and less stable. The agent struggled to consistently reach the 750 mark, and score improvements plateaued more frequently. Despite smoother gradients, the larger batch size limited update frequency and responsiveness.

In summary, as Table 2, batch size 64 provided the fastest and most stable learning, while batch size 128 offered a balanced alternative. Batch size 256, although more stable in gradient estimation, hindered learning efficiency in this setting.

5 Discussion

Our DQN implementation, adapted for discretized actions, yielded promising results (average reward 862.02 with batch size 32), confirming its pixel-based learning capability when actions are simplified. Smaller batches (32) outperformed larger ones (64, 128) in convergence and peak performance, likely due to frequent updates and beneficial stochasticity. However, DQN's discrete nature inherently limited its capacity for nuanced, smooth continuous control in CarRacing-v3.

Conversely, Advantage Actor-Critic (A2C) significantly struggled, exhibiting high variance and unstable learning despite tuning efforts. This failure is attributed to the inherent instability of vanilla policy gradients in complex, high-dimensional continuous control tasks and A2C's less sophisticated stabilization compared to PPO, leading to hyperparameter sensitivity and poor exploration.

Proximal Policy Optimization (PPO) proved the most successful and robust. It consistently delivered high rewards and stable learning, primarily due to its clipped surrogate objective (constraining policy updates) and Generalized Advantage Estimation (GAE, reducing variance). This combination effectively balanced sample efficiency, stability, and performance, making PPO well-suited for CarRacing-v3's continuous control demands from pixels.

The comparative analysis clearly favors PPO for CarRacing-v3, excelling with continuous actions, pixel inputs, and generalization to procedurally generated tracks due to its stable learning. While DQN was a reasonable compromise with discrete adaptations, its potential was capped. A2C lacked the requisite stability for this demanding environment.

A universal challenge was the computational cost of processing high-dimensional pixels and the extensive, sensitive hyperparameter tuning, highlighting the empirical nature of deep RL research.

6 Conclusion and Future Work

This mini-project successfully implemented and evaluated three widely used deep reinforcement learning algorithms, DQN, A2C, and PPO, in CarRacing-v3. Our objective was to develop an agent capable of navigating procedurally generated racetracks from raw pixel inputs, maximizing rewards while maintaining smooth control.

The study demonstrated that PPO provides the most robust and high-performing solution for this continuous control task, achieving stable learning and effectively generalizing to unseen tracks. Its architecture, particularly the clipped surrogate objective and use of GAE, proved crucial for handling the complexities of the environment. DQN, while requiring a discretized action space, achieved reasonable performance, showcasing its effectiveness for vision-based tasks when controls are simplified. However, A2C struggled with instability and high variance, rendering it less suitable for this specific problem in its standard configuration.

Our analysis of hyperparameter impacts, such as batch size in DQN, provided insights into the practical considerations of training these models. The project ultimately highlights the trade-offs between different RL algorithms in terms of performance, stability, sample efficiency, and suitability for continuous versus discrete control in visually rich, dynamic environments. PPO, in particular, stands out as a strong candidate for autonomous driving tasks in simulated settings due to its balance of these factors.

Beside the proposed solutions, the problem can also be solved through several key enhancements:

Advanced CNN Architectures: We will integrate sophisticated CNNs (e.g., ResNet, Vision Transformers, attention mechanisms) to achieve superior visual feature extraction from pixel inputs, aiming for more nuanced vehicle control and improved generalization across varied track conditions.

Neuroevolution for Policy Discovery: Neuroevolutionary algorithms (such as NEAT or evolutionary strategies) will be explored to directly evolve and optimize the policy network. This approach may uncover novel network topologies and parameter settings, leading to unique and highly effective driving strategies not easily found by gradient-based methods.

LLMs as Teaching Agents: We plan to investigate the novel application of Large Language Models (LLMs) as instructional agents. This includes leveraging LLMs for automated curriculum generation, dynamic reward shaping based on interpretable high-level driving objectives, or providing actionable feedback to accelerate the agent's learning and adaptation.

7 Acknowledgement

This project was carried out as part of the Introduction to Artificial Intelligence course, with the guidance and supervision of Professor Than Quang Khoat. We also gratefully acknowledge the partial support provided by fellow teammates. Pham Duc Cuong, Nguyen Tuan Duc for implementation of Deep Q-learning, Trinh Hong Anh, Cao Chi Cuong with Advantage Actor Critic, and Nguyen Viet Anh with Proximal Policy Optimization. We also would like to thank Nguyen Tuan Duc, Trinh Hong Anh for the written report and slide presentation.

References

[1] Farama Foundation. *Gymnasium*. https://gymnasium.farama.org/index.html

[2] Morales Miguel. (2020). *Grokking Deep Reinforcement Learning*. Manning Publications.

[3] OpenAI. *Spinning Up in Deep Reinforcement Learning*. https://github.com/openai/spinningup

[4] Hugging Face. *Deep RL Course: Unit 2, Introduction to Q-Learning*. https://huggingface.co/learn/deep-rl-course/unit2/introduction

[5] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu. (2016). Asynchronous methods for deep reinforcement learning.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov. (2017). *Proximal Policy Optimization algorithms.*

[7] Intro to Policy Optimization https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html

[8] Florentiana Yuwono, Gan Pang Yen and Jason Christopher. Self-Driving Car Racing: Application of Deep Reinforcement Learning. https://arxiv.org/html/2410.22766v1

[9] Wen-Chung (Andy) Cheng, Zhen Ni, and Xiangnan Zhong. Experimental evaluation of Proximal Policy Optimization and Advantage Actor-Critic RL algorithms using MiniGrid environment.

[10] Hugging Face. *Deep RL course*. https://huggingface.co/learn/deep-rl

[11] jerrykal. *Box2D CarRacing PPO*. https://github.com/jerrykal/CarRacing_PPO

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. (2015). Playing Atari with deep reinforcement learning. *Nature*, *518*(7540), 529–533.

[13] Christopher J. C. H. Watkins & Peter Dayan. *Q-learning*, *Machine learning.*

A. Appendix

This section include some of the remarkable code components that helps readers to generalize the idea and have information about how we implemented specifically for this problem.

A.1. Preprocessing Technique

We implemented the same preprocessing idea to all the algorithms tested to reduce noise, complexity and enhance learning efficiency of the model as shown in Listing 1.

```python
from collections import deque, namedtuple
import cv2
import numpy as np

def preprocess_frame(frame):
    """Converts frame to grayscale and resizes it."""
    gray = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
    resized = cv2.resize(gray, (84, 84), interpolation=cv2.INTER_AREA)
    return np.array(resized, dtype=np.float32) / 255.0

frame_stack_size = 4
stacked_frames = deque(maxlen=frame_stack_size)

def stack_frames(state, is_new_episode):
    """Stacks frames. Handles initial frames."""
    frame = preprocess_frame(state)

    if is_new_episode:
        stacked_frames.clear()
        for _ in range(frame_stack_size):
            stacked_frames.append(frame)
    else:
        stacked_frames.append(frame)

    stacked_state = np.stack(stacked_frames, axis=0)
    return stacked_state
```

Listing 1: Frame preprocessing and stacking.

A.2. Model Architectures

As having researched in some repositories for Atari games with the same problem, also implemented and tested with our own, we come to the conclusion that the architecture we used in Listing 2 and Listing 3 is good enough to tackle our environment.

```python
import torch.nn as nn
import torch.nn.functional as F

class DQN(nn.Module):
    def __init__(self, input_shape, n_actions):
        super(DQN, self).__init__()
        channels, height, width = input_shape
```

```
8
9           self.conv1 = nn.Conv2d(channels, 32, kernel_size=8, stride=4)
10          self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
11          self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
12
13          def conv_output_size(size, kernel_size, stride):
14              return (size - (kernel_size - 1) - 1) // stride + 1
15
16          conv_h = conv_output_size(conv_output_size(conv_output_size(height, 8, 4), 4,
            ↪  2), 3, 1)
17          conv_w = conv_output_size(conv_output_size(conv_output_size(width, 8, 4), 4,
            ↪  2), 3, 1)
18          linear_input_size = conv_h * conv_w * 64
19          self.fc1 = nn.Linear(linear_input_size, 512)
20          self.fc_out = nn.Linear(512, n_actions)
21
22      def forward(self, x):
23          x = F.relu(self.conv1(x))
24          x = F.relu(self.conv2(x))
25          x = F.relu(self.conv3(x))
26
27          x = x.view(x.size(0), -1)
28
29          x = F.relu(self.fc1(x))
30          q_values = self.fc_out(x)
31          return q_values
```

Listing 2: DQN model architecture.

```
1   class DiagonalGaussian(nn.Module):
2       def __init__(self, in_dim, out_dim):
3           super().__init__()
4           self.fc_mean = nn.Linear(in_dim, out_dim)
5           self.log_std = nn.Parameter(torch.full((1, out_dim), -0.5))
6
7       def forward(self, x):
8           mean = torch.tanh(self.fc_mean(x))
9           log_std = self.log_std.expand_as(mean)
10          std = torch.exp(log_std)
11          std = torch.clamp(std, min=1e-6)
12          return torch.distributions.Normal(mean, std)
13
14  class ActorNetwork(nn.Module):
15      def __init__(self, input_shape, n_actions, alpha):
16          super(ActorNetwork, self).__init__()
17          channels, height, width = input_shape
18          self.conv = nn.Sequential(
19              nn.Conv2d(channels, 32, kernel_size=8, stride=4),
20              nn.ReLU(),
21              nn.Conv2d(32, 64, kernel_size=4, stride=2),
22              nn.ReLU(),
23              nn.Conv2d(64, 64, kernel_size=3, stride=1),
24              nn.ReLU()
```

```
        )

        with torch.no_grad():
            conv_out = np.prod(self.conv(torch.zeros(1, *input_shape)).size()).item()

        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(conv_out, 512),
            nn.ReLU(),
            DiagonalGaussian(512, n_actions)
        )

        self.optimizer = torch.optim.Adam(self.parameters(), lr = alpha)
        self.to(device)

    def forward(self, x):
        x = self.conv(x)
        x = self.fc(x)
        return x
```

Listing 3: Example of Actor network in Actor-Critic model architecture.

A.3. Training Loop Components

Instead of presenting entire blocks of code, we dive into each remarkable function. This not only brings clarity to abstract lines of code but also builds a deeper system-level understanding, enabling readers to reuse and adapt code in their own projects more confidently.

A.3.1. DQN Epsilon-Greedy Strategy

In order to have a good learning trajectory, the DQN agent needs to have a method for choosing the action, implemented as Listing 4.

```
    def select_action(self, state):
        """Selects an action using Epsilon-Greedy strategy."""
        sample = random.random()
        eps_threshold = self.eps_end + (self.eps_start - self.eps_end) * \
            math.exp(-1. * self.steps_done / self.eps_decay)
        self.steps_done += 1
        if sample > eps_threshold:
            with torch.no_grad():
                state_tensor = torch.tensor(state, dtype=torch.float32,
                ↪  device=self.device).unsqueeze(0)
                q_values = self.policy_net(state_tensor)
                action_index = q_values.max(1)[1].item()
                return action_index
        else:
            return random.randrange(self.n_actions)
```

Listing 4: Epsilon-Greedy Strategy for choosing action.

A.3.2. A2C Advantage Function

A2C is not just a standard Actor-Critic method, it focuses on reducing the variance of the policy gradient by using the Advantage to detech whether the action is better or worse than expecting value, which is known as Action-Value function.

```
def calculate_discounted_returns(b_rewards, discount_factor):
    discounted_returns = np.zeros_like(b_rewards, dtype=np.float32)
    running_add = 0
    for i in reversed(range(b_rewards.shape[0])):
        running_add = b_rewards[i] + discount_factor * running_add
        discounted_returns[i] = running_add
    return discounted_returns

b_returns_np = calculate_discounted_returns(b_rewards_np, discount_factor)
b_returns = torch.from_numpy(b_returns_np.copy()).to(device, dtype=torch.float32)

b_advantages = b_returns - b_state_values
```

Listing 5: Advantage function computation.

A.3.3. PPO Learning Mechanism

To balance between efficient learning and stability. It is built on policy gradient methods and it uses a clipped surrogate objective function. Also, to reduce variance, Generalized Advantage Estimation (GAE) is applied as Listing 6.

```
def learn(self):
        self.actor.train()
        self.critic.train()

        states_np, actions_np, old_probs_np, vals_np, rewards_np, dones_np, batches
        = \ self.memory.generate_batches()

        advantages = np.zeros(len(rewards_np), dtype=np.float32)
        last_gae_lam = 0
        for t in reversed(range(len(rewards_np))):
            if t == len(rewards_np) - 1:
                next_non_terminal = 1.0 - dones_np[t]
                next_value = vals_np[t]
            else:
                next_non_terminal = 1.0 - dones_np[t]
                next_value = vals_np[t+1]

            delta = rewards_np[t] + self.gamma * next_value * next_non_terminal -
             ↪  vals_np[t]
            advantages[t] = last_gae_lam = delta + self.gamma * self.gae_lambda *
             ↪  next_non_terminal * last_gae_lam
        returns = advantages + vals_np

        states_tensor = torch.tensor(states_np, dtype=torch.float32).to(self.device)
        actions_tensor = torch.tensor(actions_np,
           dtype=torch.float32).to(self.device)
```

```
        old_log_probs_tensor = torch.tensor(old_probs_np,
           dtype=torch.float32).to(self.device)
        advantages_tensor = torch.tensor(advantages,
           dtype=torch.float32).to(self.device)
        returns_tensor = torch.tensor(returns, dtype=torch.float32).to(self.device)

        for epoch in range(self.n_epochs):
            for batch_indices in batches:
                batch_states = states_tensor[batch_indices]
                batch_actions = actions_tensor[batch_indices]
                batch_old_log_probs = old_log_probs_tensor[batch_indices]
                batch_advantages = advantages_tensor[batch_indices]

                batch_adv_mean = batch_advantages.mean()
                batch_adv_std = batch_advantages.std() + 1e-8
                batch_advantages_norm = (batch_advantages - batch_adv_mean) /
                ↪  batch_adv_std

                batch_returns = returns_tensor[batch_indices]

                dist = self.actor(batch_states)
                new_log_probs = dist.log_prob(batch_actions).sum(axis=-1)
                entropy = dist.entropy().mean()

                prob_ratio = torch.exp(new_log_probs - batch_old_log_probs)

                surr1 = prob_ratio * batch_advantages_norm
                surr2 = torch.clamp(prob_ratio, 1 - self.policy_clip, 1 +
                ↪  self.policy_clip) * batch_advantages_norm
                actor_loss = -torch.min(surr1, surr2).mean()

                new_values = self.critic(batch_states).squeeze(-1)
                critic_loss = F.mse_loss(new_values, batch_returns)

                total_loss = actor_loss + self.vf_coef * critic_loss - self.ent_coef
                ↪  * entropy

                self.actor.optimizer.zero_grad()
                self.critic.optimizer.zero_grad()
                total_loss.backward()
                torch.nn.utils.clip_grad_norm_(self.actor.parameters(), max_norm=0.5)
                torch.nn.utils.clip_grad_norm_(self.critic.parameters(),
                ↪  max_norm=0.5)
                self.actor.optimizer.step()
                self.critic.optimizer.step()

        self.memory.clear_memory()
```

Listing 6: Proximal Policy Optimization learning function.

B. Source Code for the Capstone Project

This section presents the complete source code developed as part of our capstone project. The codebase includes all major components implemented throughout the development cycle, including environment setup, model architecture, training scripts, and evaluation tools. We have documented the project thoroughly to ensure that others can easily understand and replicate our results. Additionally, we followed best practices in coding, version control, and modular design to maintain clarity and scalability.

You can access the full implementation and related resources on our official GitHub repository at the following link: https://github.com/Vanh41/CarRacing.