Draft Report: Autonomous Driving via Proximal Policy Optimization (PPO)

Date: December 16, 2025
Subject: Demonstrator of Self-Driving Car using PPO Method
Environment: Gymnasium CarRacing-v3

1. Introduction

The objective of this project is to develop an autonomous agent capable of navigating a race track effectively. Using **Reinforcement Learning (RL)**, specifically the **Proximal Policy Optimization (PPO)** algorithm, the agent learns to control a vehicle in the `CarRacing-v2` environment.

Unlike rule-based systems, the agent learns purely from visual inputs (pixels) and trial-and-error interactions, aiming to maximize a reward signal based on speed and track completion. This implementation is built from scratch using **PyTorch** and **NumPy**, avoiding high-level abstractions like Stable Baselines3 to demonstrate a deep understanding of the underlying algorithms.

2. Short Theory Explanation

2.1 The Reinforcement Learning Problem

At its core, Reinforcement Learning (RL) involves an agent learning to make decisions by interacting with an environment to achieve a goal. We model this as a Markov Decision Process (MDP):

- **State (s_t):** The agent's current observation of the world (in our case, the stacked grayscale images of the track).
- **Action (a_t):** The decision made by the agent (steering, gas, brake).
- **Reward (r_t):** A scalar feedback signal indicating the immediate success of the action.

The agent's objective is not just to maximize the immediate reward, but to maximize the **cumulative reward** (return) over time.

2.2 The Essence of Policy Gradient and Its Challenges

To solve this MDP, we use **Policy Gradient methods**. These methods directly optimize the policy π_θ (the neural network) to maximize the expected return. They work by adjusting the network parameters θ in the direction that increases the probability of taking actions that lead to high rewards.

The Challenge: Traditional policy gradient methods (like REINFORCE) can be highly unstable.

- If the learning step size is too small, training is agonizingly slow.
- If the step size is too large, a single "bad" update can ruin the policy, causing performance to collapse. Once the policy degrades, the agent collects poor data, and it may never recover.

2.3 Proximal Policy Optimization (PPO) Explained

We utilize **PPO (Proximal Policy Optimization)**, an algorithm designed to strike a balance between sample efficiency and stability. As described in the literature, PPO acts as a successor to earlier methods by putting "guardrails" on the learning process.

2.3.1 The Actor-Critic Architecture

Our implementation uses two distinct networks (or heads):

1. **The Actor (π_θ):** Decides which action to take. It outputs the distribution parameters (Mean and Standard Deviation) for steering, gas, and brake.
2. **The Critic (V_ϕ):** Estimates the value of the current state. This is crucial for computing the **Advantage**. This value acts as a **baseline** that is subtracted from the empirical return (R_t) to calculate the **Advantage** ($A_t = R_t - V(s)$). Theoretically, using this baseline reduces the **variance** of the policy gradient estimates without introducing bias. This relates to the definition of variance, $\mathrm{Var}(X) = \mathbb{E}[(X - \mathbb{E}[X])^2]$; by minimizing the Mean Squared Error (MSE) between the predicted value and the actual return, the Critic effectively "centers" the learning signal, ensuring the training process remains stable.

2.3.2 The Advantage Function

PPO relies on the Advantage Function, $\hat{A}_t$, to guide updates. Instead of looking at raw rewards, the Advantage asks: > "*How much better was this specific action compared to the average action the agent usually takes in this state?*"

$$\hat{A}_t = Q(s,a) - V(s)$$

This helps the algorithm focus specifically on actions that yield *unexpectedly* good results, reducing variance in training.

2.3.3 The Clipped Surrogate Objective (The "Guardrails")

The key innovation of PPO is how it updates the policy. It limits how much the policy can change in a single update using a **clipping mechanism**.

Let $r_t(\theta)$ be the probability ratio between the new policy and the old policy: $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$.

The PPO objective function is:

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

Intuition:

- If the advantage $\hat{A}_t$ is positive, we want to increase the probability of the action ($r_t > 1$).
- However, if r_t grows beyond $1+\epsilon$ (usually 1.2), the clipping term kicks in.
- This effectively stops the update from being too drastic. It ensures the new policy stays **proximal** (close) to the old one, preventing the catastrophic forgetting common in standard RL.

---

3. Specification of Demonstrator Setting

This section defines the “Rules of the Game” and how the agent perceives and interacts with the world.

3.1 Environment Definition, Dynamics and Physics Modeling

It is crucial to understand that `CarRacing-v2` is not merely a video playback, but a rigid-body physics simulation powered by the **Box2D** engine. This engine handles the integration of forces, momentum, friction, and collisions, which directly dictates the complexity of the control task.

- **Environment ID:** `CarRacing-v2` (Gymnasium).
- **Type:** Continuous Control from visual inputs.
- **Rendering:** The environment renders a top-down view of the car and track.

3.1.1 The Vehicle Model

The car is modeled as a compound object governed by Newtonian mechanics. Specifically:

- **Rigid Body Dynamics:** The car consists of a chassis (a rectangular polygon with mass and inertia) connected to four wheels via joints.
- **Rear-Wheel Drive:** The "Gas" action applies torque specifically to the rear wheels, generating propulsion.
- **Inertia and Momentum:** Unlike simple grid-world games, the agent does not control the vehicle's position or velocity directly. Instead, it applies **forces**. This means the car has momentum; it cannot stop instantly and will skid if the steering angle is too sharp at high speeds. The agent must learn to account for braking distances and drift dynamics.

3.1.2 The Track and Surface Properties

To prevent the agent from simply memorizing a sequence of turns (overfitting), the track is **procedurally generated** at the start of each episode using randomized control points connected by Splines (Bézier curves).

Crucially, the physics engine simulates different surface properties:

- **Road (Poly1):** modelel with a high friction coefficient. This allows the tires to grip, enabling acceleration and turning.
- **Grass (Background):** Modeled with very low friction.
- **The "Tiles":** Although the rendering looks continuous, logically the track is divided into discrete "tiles" or checkpoints. These are invisible to the camera but are used by the environment to calculate the progress reward.

3.1.3 Interaction and "Off-Track" Detection Logic

One of the most critical aspects of the environment is how it decides when the car has failed. This is handled via **Box2D Contact Listeners** (sensors):

1. **Wheel Sensors:** The simulation continuously checks what material is essentially "under" each of the four wheels.
2. **Friction Dynamics:** If a wheel touches the road, it has grip. If it touches the grass, it loses traction, making the car hard to control.
3. **The Failure Condition:** The environment flags the car as "outside the track" when **all four wheels** lose contact with the road tiles simultaneously. In our implementation, we use this signal to trigger a harsh

penalty and terminate the episode, teaching the agent that staying on the asphalt is the primary constraint.

3.2 Observation Space (State Representation)

The raw environment provides a $96 \times 96 \times 3$ RGB image. To enable the agent to perceive motion and reduce computational complexity, we implemented a custom wrapper pipeline:

1. **Grayscale Conversion:** The 3-channel RGB image is converted to a single-channel grayscale image (96×96). This focuses learning on shapes and road boundaries rather than colors, reducing the input dimensionality by a factor of 3.
2. **Frame Stacking (Temporal Context):** We stack the **4 most recent frames** into a single observation. This transforms the input shape to $(4, 96, 96)$.
   - *Rationale:* A single static image contains no information about velocity or acceleration. By stacking frames, the Convolutional Neural Network (CNN) can infer the car's speed and turning direction from the differences between channels.
3. **Normalization:** Input pixel values (integers $0 - 255$) are scaled to the floating-point range $[0.0, 1.0]$ immediately before being processed by the neural network. This stabilizes training gradients.

Final State Shape: Tensor of shape `(Batch_Size, 4, 96, 96)`.

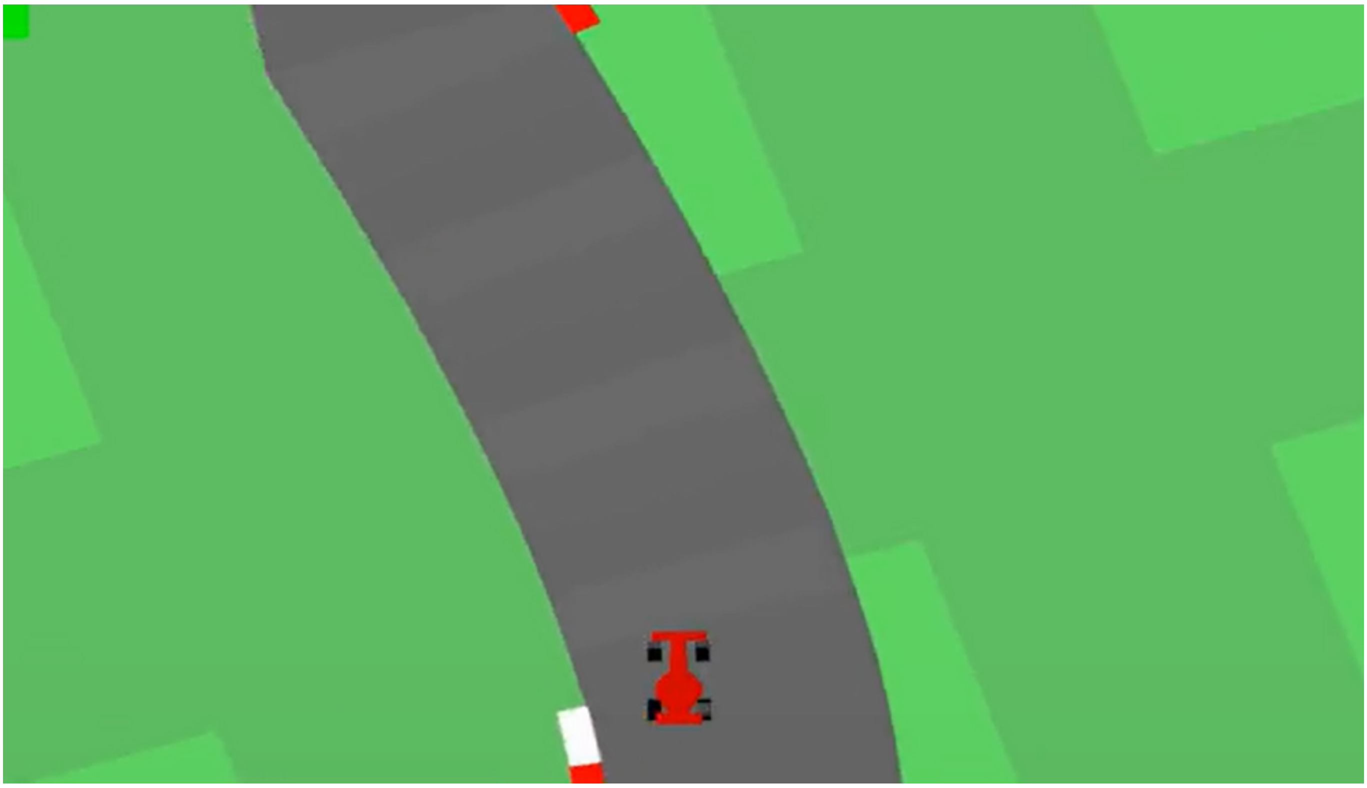

Figure 1: Visual example of the `CarRacing-v2` (Gymnasium) environment.

Final state shape: tensor of shape ((4, 96, 96)) (Channels, Height, Width).

3.3 Action Space (Action Representation)

Unlike discrete environments (e.g., "Press Left" or "Press Right"), driving requires smooth, continuous adjustments. The agent operates in a continuous action space controlling three actuators. The Policy Network outputs a **Gaussian distribution** (mean and learnable standard deviation) for each action, from which values are sampled.

To enable exploration, the Actor network does not output a fixed action. Instead, for each action component, it outputs the parameters of a **Gaussian Distribution**: a **Mean (μ)** and a learnable **Log-Standard Deviation ($\log\sigma$).**

The action a_t sent to the environment is sampled as:

$$a_t \sim \mathcal{N}(\mu(s_t), \sigma(s_t))$$

During evaluation, we bypass sampling and use the mean μ directly for deterministic behavior.

Component	Range	Description
Steering	$[-1.0, 1.0]$	Controls the wheel angle. -1.0 is full Left, 1.0 is full Right.
Gas	$[0.0, 1.0]$	Throttle/Acceleration intensity.
Brake	$[0.0, 1.0]$	Braking intensity.

3.4 Reward Function

The reward function $R(s, a)$ is the core signal that guides learning. We use the standard dense reward structure of `CarRacing-v2`, which implicitly balances speed and safety.

The total reward for an episode is calculated as:

$$ R_{episode} = \sum_{t=0}^{T} \left(\underbrace{r_{tile}}*{\text{Progress}} + \underbrace{r*{time}}*{\text{Speed Incentive}} \right) + \underbrace{r*{penalty}}_{\text{Safety}} $$

Where:

- **Progress Reward (r_{tile}):** The track is composed of N tiles. Visiting a new tile yields $+1000/N$ points. Visiting all tiles results in $+1000$ points.
- **Time Penalty (r_{time}):** A constant penalty of -0.1 is applied at every time step. This forces the agent to drive fast; if it drives too slowly, the accumulated negative time penalty will outweigh the progress reward.
- **Safety Penalty ($r_{penalty}$):** If the car moves completely off the field of play (all wheels on grass), the episode terminates immediately with a penalty of -100.
- **Success Criterion:** The agent is considered to have "solved" the environment if it consistently achieves a score **> 900 points**, implying it completed the track reasonably fast with minimal errors.

---

4. Methods and Implementation Strategy

4.1 Neural Network Architecture

The architecture follows a standard **Actor-Critic** design sharing a common convolutional backbone, implemented in PyTorch.

1. **Visual Encoder (Backbone):**

   - **Input:** $(4, 96, 96)$ (4 stacked grayscale frames).
   - **Conv1:** 32 filters, 8×8 kernel, stride 4. (Reduces spatial dimension efficiently).
   - **Conv2:** 64 filters, 4×4 kernel, stride 2.
   - **Conv3:** 64 filters, 3×3 kernel, stride 1.
   - **Flatten:** Converts the 3D feature maps into a 1D vector of 1024 features.
   - *Activation:* ReLU is used after each convolutional layer to introduce non-linearity.

2. **Heads:**

   - **Actor (Policy):** Two fully connected layers (512 units $\rightarrow$ Action Dim). It outputs the **Mean (μ)** of the action distribution. The **Standard Deviation (σ)** is learned as a separate independent parameter (`actor_logstd`), allowing the agent to adapt its exploration noise during training.
   - **Critic (Value):** Two fully connected layers (512 units $\rightarrow$ 1 unit) estimating the state value $V(s)$.

4.3 Technologies Used

- **Python 3.10+**
- **PyTorch:** neural network definition and gradient-based optimization.
- **NumPy:** numerical utilities and buffer manipulation.
- **Gymnasium:** environment interface (`CarRacing-v3`).

4.4 Training Loop Implementation Details

The training process is orchestrated in `train.py` using a vectorized architecture to maximize GPU efficiency. The algorithm follows a strict cycle of **Collection** $\rightarrow$ **Estimation** $\rightarrow$ **Optimization**, repeated until the total step budget (3M steps) is reached.

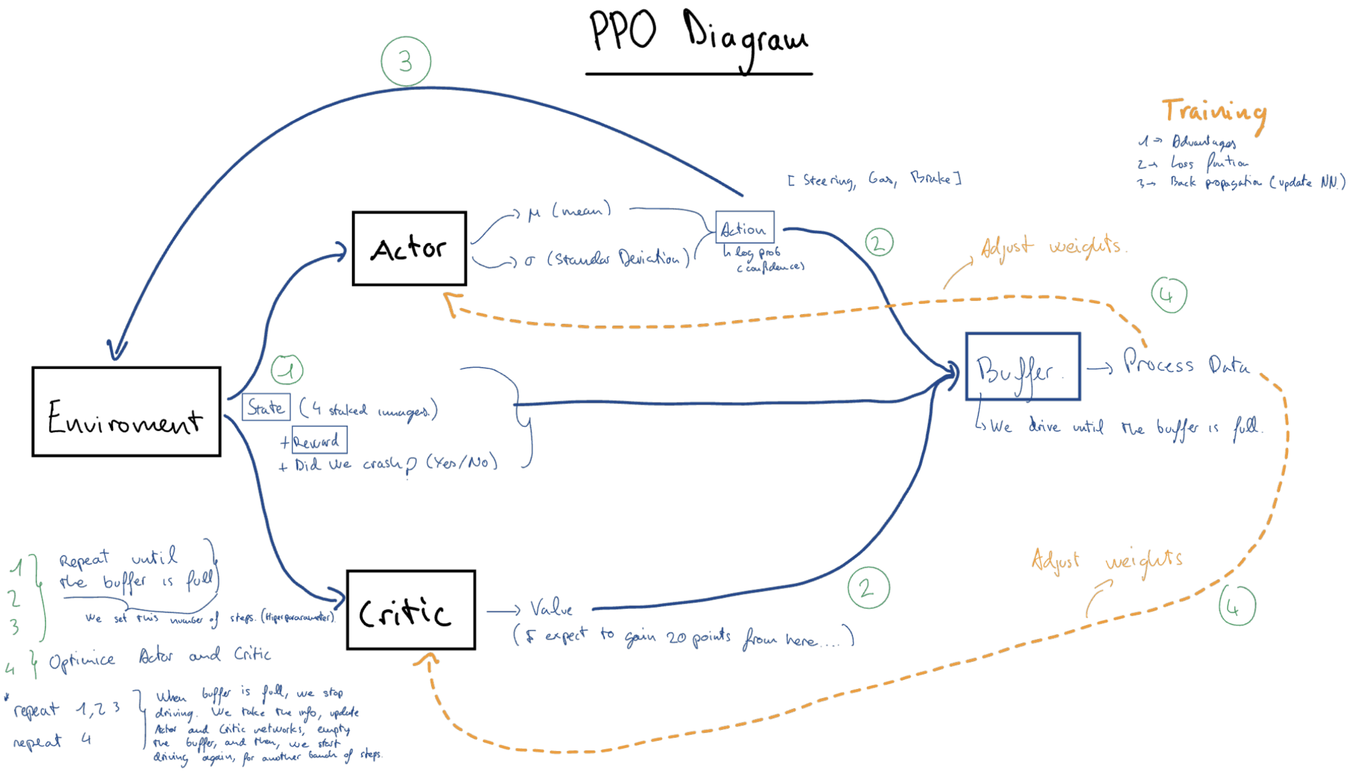
PPO Diagram
3
Actor
→ μ (mean)
→ σ (Standar Deviation)
Action
↳ log prob
(confidence)
[Steering, Gas, Brake]
2
Training
1 → Advantages
2 → Loss function
3 → Back propagation (update NN)
Adjust weights.
4
Buffer.
→ Process Data
↳ We drive until the buffer is full.
Enviroment
1
State
(4 stacked images)
+ Reward
+ Did we crash? (Yes/No)
Critic
→ Value
(I expect to gain 20 points from here...)
2
Adjust weights
4
1
2
3
Repeat until the buffer is full
We set this number of steps. (Hiperparameter).
4
Optimice Actor and Critic
* repeat 1,2 3
repeat 4
When buffer is full, we stop driving. We take the info, update Actor and Critic networks, empty the buffer, and then, we start driving again, for another bunch of steps.

Figure 2: Visual representation of the PPO training cycle, showing the data collection phase (filling the buffer)

and the backpropagation phase (updating Actor and Critic).

Phase 1: Vectorized Environment Setup

Instead of training on a single track, we initialize **8 parallel environments** (`num_envs=8`) using `gym.vector.AsyncVectorEnv`. This allows the agent to collect diverse experiences simultaneously, breaking the correlation between consecutive samples and stabilizing training.

- **Batch Size:** Each environment runs for 1,024 steps per iteration, resulting in a total batch size of **8,192 transitions** per update.
- **Annealing:** The learning rate linearly decays from 3×10^{-4} to 0 over the course of training.

Phase 2: Data Collection (Rollout)

In this phase, the agent interacts with the environment without updating its weights (`torch.no_grad()`). For each step t:

1. **Action Selection:** The Actor network outputs a distribution $\mathcal{N}(\mu, \sigma)$. We sample an action a_t, its log-probability $\log \pi(a_t|s_t)$, and the estimated value $V(s_t)$.
2. **Storage:** The transition $(s_t, a_t, r_t, d_t, \log \pi_t, V_t)$ is stored in a buffer on the GPU.

Phase 3: Generalized Advantage Estimation (GAE)

Once the buffer is full, we calculate the **Advantage** ($\hat{A}_t$), which measures how much better an action was compared to the Critic's expectation. We use GAE to balance bias and variance. The calculation is performed backwards:

$$\delta_t = r_t + \gamma V(s_{t+1})(1 - d_{t+1}) - V(s_t)$$

$$ \hat{A}*t = \delta_t + (\gamma \lambda) \hat{A}*{t+1} $$

Where:

- δ_t: Temporal Difference (TD) error at step t.
- r_t: Immediate reward received from the environment.
- $V(s)$: Value estimated by the Critic network.
- d_{t+1}: "Done" flag (1 if the episode ended, 0 otherwise).
- γ (**Gamma**): Discount factor (set to `0.99`), determining the weight of future rewards.
- λ (**Lambda**): GAE smoothing parameter (set to `0.95`), balancing bias vs. variance.

Phase 4: Optimization (Backpropagation)

The collected batch (8,192 samples) is flattened and shuffled. The optimization runs for **10 epochs** (`update_epochs`) with minibatches of size 256. The weights are updated by minimizing the following **Total Loss function**:

$$ L_{total} = \underbrace{L^{CLIP}(\theta)}*{\text{Policy Loss}} + c*{vf} \underbrace{L^{VF}(\theta)}*{\text{Value Loss}} - c*{ent} \underbrace{S\pi}_{\text{Entropy Bonus}} $$

Where:

- c_{vf}: Value coefficient (set to `0.5`).
- c_{ent}: Entropy coefficient (set to `0.01`).

Implementation details from `train.py`:

1. **Policy Loss (L^{CLIP}):** We calculate the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. To prevent destructive updates, this ratio is clipped:

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

   - ϵ (**Epsilon**): Clipping coefficient (set to `0.2`), limiting the policy change to ± 20.
   - $\hat{A}_t$: Normalized Advantage estimate.

2. **Value Loss (L^{VF}):** The Critic is trained to minimize the Mean Squared Error (MSE) between its prediction and the computed returns:

$$L^{VF} = 0.5 \cdot (V_\theta(s_t) - R_t)^2$$

   - R_t: The calculated return ($A_t + V_t$).

3. **Entropy Bonus ($S[\pi]$):** The entropy of the distribution is subtracted from the loss to encourage exploration, preventing the agent from becoming deterministic too early.

4. **Gradient Clipping:** Finally, the global norm of the gradients is clipped to `0.5` before the optimizer step to ensure stability.

---

5. Evaluation of the Training Process and Results

To validate the effectiveness of the custom PPO implementation, a structured evaluation protocol is applied. The goal is to quantify how performance improves with more experience, how stable the learned policy is, and whether the behavior is suitable for autonomous driving.

5.1 Experimental Setup

5.1.1 Trained Checkpoints

Three PPO agents are trained with increasing interaction budgets:

- **Model A – 500K steps**
- **Model B – 1.0M steps**
- **Model C – 2.0M steps**

Each “step” corresponds to one environment transition in the vectorized setting. All other hyperparameters (learning rate, batch size, discount factor, clipping coefficient, etc.) are kept fixed across runs to isolate the effect of training duration.

5.1.2 Evaluation Protocol

For each checkpoint, the following procedure is used:

- Run **(N = 30)** evaluation episodes with **no exploration noise** (actions given by the mean of the policy).
- Use **fixed random seeds** for reproducibility while still sampling different tracks.
- Record per-episode statistics:
  - Total reward
  - Episode length (number of steps)
  - Per-step rewards
  - Average steering, throttle and brake over the episode

From these, aggregate metrics are computed:

- Mean, median, minimum and maximum reward
- Standard deviation of the reward (stability)
- Mean episode length
- **Win rate:** percentage of episodes with reward (> 900)
- **Success rate:** percentage of episodes with reward (> 0)

(In the final version, this subsection will also reference the JSON logs and plots generated by the evaluation scripts.)

5.2 Impact of Training Duration

This subsection analyzes how increasing the number of training steps affects driving performance.

5.2.1 Learning Curves

For each model, a **Reward vs. Training Step** curve is logged during training. These curves highlight:

- The **rapid improvement** phase at the beginning of training.
- The **“knee” point** where additional training yields diminishing returns.
- The **plateau region** where performance stabilizes around a maximum value.

In practice, Model A (500K) shows only partial mastery of the task, Model B (1M) exhibits significantly smoother control and higher average scores, while Model C (2M) converges towards consistently high rewards.

5.2.2 Quantitative Comparison

The table below summarizes the main evaluation metrics for the three checkpoints (values to be filled in from experiments):

Metric	500K steps	1M steps	2M steps
Mean reward	xxx.x	xxx.x	xxx.x
Std. dev. reward	xxx.x	xxx.x	xxx.x
Min / Max reward	min–max	min–max	min–max
Median reward	xxx.x	xxx.x	xxx.x
Win rate (> 900)	xx.x %	xx.x %	xx.x %

Metric	500K steps	1M steps	2M steps
Success rate (> 0)	xx.x %	xx.x %	xx.x %
Mean episode length	xxx.x	xxx.x	xxx.x

The final report will discuss:

- How much average reward increases from 500K to 2M steps.
- How the standard deviation decreases, indicating more stable behavior.
- At which point the curve effectively saturates (practical training budget).

5.3 Qualitative Behaviour Analysis

Numbers alone do not capture the quality of driving. Therefore, evaluation videos are recorded for each checkpoint.

- **Model A (500K):**
  - Tends to over-steer and oscillate around the track center.
  - Frequently leaves the track, leading to early termination and low scores.
- **Model B (1M):**
  - Handles most corners reasonably well.
  - Still exhibits occasional over-corrections and slower recovery after mistakes.
- **Model C (2M):**
  - Follows the track smoothly with anticipatory steering.
  - Maintains higher speed on straights while braking appropriately before sharp turns.
  - Visually resembles human-like driving and completes most laps without leaving the track.

Selected frames and short clips can be embedded in the presentation to illustrate the progression from unstable to reliable driving.

5.4 Stability and Control Signals

The control signals are also examined to better understand the learned policy:

- **Steering:**
  - High variance at 500K (nervous zig-zagging).
  - Reduced variance at 2M, indicating smoother corrections.
- **Throttle:**
  - Low average throttle at 500K (overly cautious behavior).
  - Higher and more consistent throttle at 2M, reflecting confidence in keeping the car on track.
- **Brake:**
  - Frequent braking at 500K, often unnecessary.
  - More targeted braking at 2M, mostly before tight corners.

These observations align with the improvement in reward and win rate and support the claim that the agent has learned a meaningful driving strategy rather than exploiting spurious artifacts.

5.5 Limitations

Despite the positive results, the evaluation reveals some limitations:

- Performance still degrades on extremely sharp or unusual track configurations.
- The agent is trained and tested in a single simulated environment; transfer to real-world conditions is not addressed.
- All evaluations are done under similar observation and reward settings; robustness to sensor noise or modified rewards is not yet studied.

These points motivate future work, such as domain randomization, curriculum learning, or alternative architectures.

5.6 Generalization vs. Memorization: Random Seeds vs Fixed Track

To study whether the agent truly *learns to drive* or merely *memorizes a specific track*, two PPO agents with identical architectures and hyperparameters are trained:

- **Generalist Agent (Random Seeds):**
    - Trained on procedurally generated tracks with a different random seed at every episode.
    - Objective: learn generic driving skills (cornering, speed control, recovery) that transfer across layouts.
- **Specialist Agent (Fixed Seed):**
    - Trained on a single, fixed track layout using a constant environment seed.
    - Objective: intentionally encourage memorization of that specific circuit.

Both agents are trained for the same number of environment steps (e.g. **1.0M** or **2.0M** steps), so that differences in performance can be attributed to the training distribution rather than to training budget.

5.6.1 Evaluation Protocol

To compare *generalization* and *memorization*, each agent is evaluated under two conditions:

1. **Random-track evaluation (unseen layouts):**
    - Environment seeds are sampled randomly at every episode.
    - Measures how well the policy adapts to new tracks it has never seen before.
2. **Fixed-track evaluation (memorized layout):**
    - The environment is reset with the **same seed used to train the Specialist Agent**.
    - Measures how strongly each policy overfits to that single circuit.

For each condition and each agent we run **N evaluation episodes** (e.g. (N = 20)) with deterministic actions (policy mean, no exploration noise) and record:

- Episode total reward
- Episode length
- Win rate (percentage of episodes with reward (> 900))

5.6.2 Quantitative Results

The following table summarizes the expected behaviour (values to be filled in with experimental results):

Agent / Evaluation seed	Mean reward	Std. dev.	Win rate (> 900)
Generalist – Random tracks	xxx.x	xxx.x	xx.x %
Generalist – Fixed track	xxx.x	xxx.x	xx.x %
Specialist – Random tracks	xxx.x	xxx.x	xx.x %
Specialist – Fixed track	xxx.x	xxx.x	xx.x %

5.6.3 Discussion

The expected pattern is:

- On the **fixed track**, the Specialist Agent should achieve very high scores and win rates, sometimes outperforming the Generalist Agent, as it has effectively memorized that layout.
- On **random tracks**, the Specialist Agent is likely to degrade significantly, with:
  - Lower mean reward,
  - Higher variance,
  - Lower win rate, indicating poor robustness to unseen circuits.
- The **Generalist Agent** should maintain more stable performance across both evaluation conditions, with smaller gaps between fixed and random tracks, which is a strong indication of *true generalization* rather than memorization.

This experiment illustrates a classical trade-off:

- Training on **diverse seeds** leads to more robust but slightly slower-learning agents.
- Training on a **fixed seed** encourages rapid improvement on that specific scenario at the cost of poor out-of-distribution performance.

In the context of autonomous driving, the Generalist Agent is clearly preferable, since real roads are never identical to the training track.

---

6. Delivery and Usage

The final project deliverable is structured as an archive containing:

1. **Source Code**
   - Training script (`train.py`)
   - Evaluation script (`evaluate.py` / extended evaluation tools)
   - Agent and utility modules (`agent.py`, `utils.py`)
2. **Trained Models**
   - Checkpoints for 500K, 1M and 2M training steps.
   - Final best-performing model.
3. **Requirements**
   - `requirements.txt` specifying Python dependencies (Gymnasium, PyTorch, NumPy, etc.).

6.1 How to Run

1. **Install dependencies:**

   ```
   pip install -r requirements.txt
   ```