Reinforcement Learning - Assignment 3

Marco Turchetti
Student ID: 1996734

Theory: REINFORCE

1 Problem Definition

We are considering an environment with a 2-dimensional state representation $x(s) \in \mathbb{R}^2$ and two possible actions $a \in \{0, 1\}$. The policy is defined via a Linear Function Approximator (Logistic Regression):

$$\pi(a = 1|s) = \sigma(w^\top x(s)) = \frac{1}{1 + e^{-w^\top x(s)}} \tag{1}$$

Initial parameters and environment constants are:

- Initial weights: $w = [0.8, 1]^\top$
- Learning rate: $\alpha = 0.1$
- Discount factor: $\gamma = 0.9$

2 Policy Rewriting and Gradient Derivation

2.1 Why the policy can be rewritten

In binary action spaces, we can represent the probability of any action $a \in \{0, 1\}$ using a single expression. Let $z = w^\top x(s)$. We define:

$$\pi(a|s) = \sigma(z)^a \cdot (1 - \sigma(z))^{1-a} \tag{2}$$

This formulation is used because:

1. If $a = 1$: $\pi(1|s) = \sigma(z)^1 \cdot (1 - \sigma(z))^0 = \sigma(z)$.
2. If $a = 0$: $\pi(0|s) = \sigma(z)^0 \cdot (1 - \sigma(z))^1 = 1 - \sigma(z)$.

This allows us to handle both cases with a single logarithmic derivative, which is mathematically more efficient for gradient ascent.

2.2 Gradient of the Log-Policy

To apply the REINFORCE update $\Delta w = \alpha \nabla_w \ln \pi(a|s) G_t$, we calculate the gradient:

$$\ln \pi(a|s) = a \ln(\sigma(z)) + (1 - a) \ln(1 - \sigma(z))$$
$$\frac{\partial \ln \pi}{\partial z} = a \frac{\sigma'(z)}{\sigma(z)} + (1 - a) \frac{-\sigma'(z)}{1 - \sigma(z)}$$

Using the identity $\sigma'(z) = \sigma(z)(1 - \sigma(z))$:

$$\begin{aligned}\frac{\partial \ln \pi}{\partial z} &= a(1 - \sigma(z)) - (1 - a)\sigma(z) \\ &= a - a\sigma(z) - \sigma(z) + a\sigma(z) \\ &= a - \sigma(z)\end{aligned}$$

Applying the chain rule ($\nabla_w z = x(s)$):

$$\nabla_w \ln \pi(a|s) = (a - \sigma(w^\top x(s)))x(s) \tag{3}$$

3 Numerical Execution

The trajectory obtained is:

- $t = 0 : x(s_0) = [1, 0]^\top, a_0 = 0, r_1 = 0$
- $t = 1 : x(s_1) = [1, 0]^\top, a_1 = 1, r_2 = 1$
- $t = 2 : x(s_2) = [0, 1]^\top$ (Terminal)

3.1 Iteration $t = 0$

1. Calculate Return (G_0):

$$G_0 = r_1 + \gamma r_2 = 0 + 0.9(1) = 0.9$$

2. Calculate Policy Probability:

$$z_0 = w^\top x(s_0) = [0.8, 1] \begin{bmatrix}1\\0\end{bmatrix} = 0.8$$

$$\sigma(0.8) = \frac{1}{1 + e^{-0.8}} \approx 0.69$$

3. Update Weights:

$$\begin{aligned}w_{new} &= w + \alpha(a_0 - \sigma(z_0))x(s_0)G_0 \\ w_{new} &= \begin{bmatrix}0.8\\1\end{bmatrix} + 0.1(0 - 0.69)\begin{bmatrix}1\\0\end{bmatrix}(0.9) \\ w_{new} &= \begin{bmatrix}0.8\\1\end{bmatrix} + \begin{bmatrix}-0.0621\\0\end{bmatrix} = \begin{bmatrix}0.7379\\1\end{bmatrix}\end{aligned}$$

3.2 Iteration $t = 1$

1. Calculate Return (G_1):

$$G_1 = r_2 = 1$$

2. Calculate Policy Probability:

$$z_1 = w_{new}^\top x(s_1) = [0.7379, 1] \begin{bmatrix}1\\0\end{bmatrix} = 0.7379$$

$$\sigma(0.7379) = \frac{1}{1 + e^{-0.7379}} \approx 0.68$$

3. Update Weights:

$$w_{final} = w_{new} + \alpha(a_1 - \sigma(z_1))x(s_1)G_1$$
$$w_{final} = \begin{bmatrix} 0.7379 \\ 1 \end{bmatrix} + 0.1(1 - 0.68) \begin{bmatrix} 1 \\ 0 \end{bmatrix} (1)$$
$$w_{final} = \begin{bmatrix} 0.7379 \\ 1 \end{bmatrix} + \begin{bmatrix} 0.032 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.7699 \\ 1 \end{bmatrix}$$

4 Summary of Results

The initial weights $w = [0.8, 1]^\top$ were updated over one episode. The first update moved the weights away from action $a = 0$ (since $r = 0$ and the return was weighted by the future reward), and the second update moved the weights toward action $a = 1$ (due to $r = 1$). The final weights are approximately **[0.77, 1.00]**.

Practice: Car-Racing with TRPO

This part of the report details the implementation and comparative analysis of a Trust Region Policy Optimization (TRPO) agent for the `CarRacing-v2` environment. To ensure stable policy updates, a custom TRPO solver is developed using Conjugate Gradient descent to enforce a hard KL-divergence constraint. The study investigates the impact of reward shaping on learning dynamics by evaluating three distinct configurations: a baseline unshaped reward, a stability-focused penalty, and an over-constrained braking incentive. Results demonstrate that the unshaped baseline yields the most robust policy, achieving a peak training reward of ~ 835 and an evaluation score of ~ 650. While the refined shaping mechanism significantly improved sample efficiency—converging in 170 epochs versus the baseline's 188—it failed to outperform the baseline in raw scoring, suggesting that heuristic constraints can inadvertently limit asymptotic performance. The final agent demonstrates competent racing behavior, with the performance gap to the solved threshold (900) primarily attributed to high variance inherent in limited-batch on-policy learning.

Introduction

The `CarRacing-v2` environment presents a challenging continuous control problem ($s \in \mathbb{R}^{96\times96\times3}$, $a \in \mathbb{R}^3$) characterized by "bang-bang" control dynamics and the risk of spin-outs on sharp turns.

Trust Region Policy Optimization (TRPO) is implemented to ensure monotonic improvement. The implementation is strongly inspired by the PyTorch TRPO minimal implementation described in the article by Vladyslav Yazykov [3], adapting the Fisher Vector Product calculation for the specific CNN architecture required by pixel-based inputs.

5 Methodology

5.1 Network Architecture

The policy $\pi_\theta(a|s)$ is parameterized by a Convolutional Neural Network (CNN) with the following structure:

- **Feature Extractor:** Three convolutional layers (kernel sizes 8, 4, 3) with ReLU activations.
- **Actor Head:** A fully connected layer mapping features to the mean μ of a Gaussian distribution.
- **Critic Head:** A separate value network $V_\phi(s)$ used for Generalized Advantage Estimation (GAE).

5.2 Action Distribution Modeling

The choice of a Gaussian distribution is standard for continuous control, as it provides a differentiable probability density over the action space necessary for policy gradient calculations. However, a standard Gaussian has infinite support $(-\infty, +\infty)$, while the environment's actions are physically bounded (e.g., $[-1, 1]$ for steering).

To mitigate "action saturation" where the majority of samples are clamped to extreme values, a hyperbolic tangent (tanh) activation is applied to the predicted mean:

$$\mu_{bounded} = \tanh(\mu_{raw}) \tag{4}$$

This bounds the center of the Gaussian within $(-1, 1)$, ensuring that the distribution remains focused on the valid action space.

5.3 Trust Region Policy Optimization (TRPO)

To prevent catastrophic performance collapses during training, TRPO formulates the update as a constrained optimization problem. Following the lecture slides, the objective and constraint are approximated using Taylor expansions.

5.3.1 Simplified Trust-Region Solution via Whitening

The expected advantage is maximized subject to a trust region constraint on the KL divergence:

$$\max_{\Delta} \nabla^T \Delta \quad \text{s.t.} \quad \Delta^T F \Delta \leq \delta \tag{5}$$

where $\Delta = \theta - \theta_t$ is the update step, F is the Fisher Information Matrix (FIM), and $\nabla = \nabla_\theta J(\pi_{\theta_t})$ represents the gradient of the objective function (expected advantage).

The constraint $\Delta^T F \Delta \leq \delta$ describes an ellipsoid in the parameter space, making it difficult to find the optimal direction directly (the gradient is not necessarily the steepest ascent direction in this curved space). To solve this, the "Whitening" transformation is used to map the ellipsoid to a sphere:

$$\tilde{\Delta} := F^{1/2} \Delta \tag{6}$$

Substituting this into the optimization problem yields:

$$\max_{\tilde{\Delta}} (F^{-1/2} \nabla)^T \tilde{\Delta} \quad \text{s.t.} \quad \tilde{\Delta}^T \tilde{\Delta} \leq \delta \tag{7}$$

In this whitened space, the constraint is a simple ball of radius $\sqrt{\delta}$. The optimal direction $\tilde{\Delta}_{max}$ is now collinear with the transformed gradient vector $F^{-1/2}\nabla$, scaled to hit the boundary:

$$\tilde{\Delta}_{max} = \sqrt{\frac{\delta}{\nabla^T F^{-1} \nabla}} F^{-1/2} \nabla \tag{8}$$

Mapping back to the original parameter space using $\Delta_{max} = F^{-1/2} \tilde{\Delta}_{max}$ gives the final Natural Policy Gradient update rule:

$$\Delta_{max} = \sqrt{\frac{\delta}{\nabla^T F^{-1} \nabla}} F^{-1} \nabla \tag{9}$$

5.3.2 Conjugate Gradient

Directly computing F^{-1} is computationally prohibitive ($N \times N$ matrix). Instead, the system $Fx = g$ is solved using the Conjugate Gradient (CG) algorithm to find the search direction $x = F^{-1}g$. This only requires computing the matrix-vector product Fv, which can be done efficiently using the "double backpropagation" trick without storing F.

6 Implementation Details

This section presents the core code implementation corresponding to the mathematical derivations above, derived directly from the `student.py` source.

6.1 Fisher Vector Product

The core of the "matrix-free" optimization is computing Fv efficiently using the double-backpropagation trick.

```python
def fisher_vector_product(self, states, p, damping=0.01):
    """
    COMPUTES F*p WITHOUT STORING THE MATRIX F.
    Trick: F*p = grad( (grad(KL) * p) )
    """
    p.detach()
    mu, std = self.forward(states)
    dist = Normal(mu, std)

    # We compute KL against a fixed version of itself
    mu_old = mu.detach()
    std_old = std.detach()
    dist_old = Normal(mu_old, std_old)

    kl = torch.distributions.kl_divergence(dist_old, dist).mean()

    # 1st Derivative
    kl_grad = torch.autograd.grad(kl, self.actor_parameters(), create_graph=True)
    kl_grad_flat = torch.cat([grad.reshape(-1) for grad in kl_grad])

    # Dot Product
    kl_grad_p = (kl_grad_flat * p).sum()

    # 2nd Derivative
    kl_hessian_p = torch.autograd.grad(kl_grad_p, self.actor_parameters())
    kl_hessian_p_flat = torch.cat([grad.reshape(-1) for grad in kl_hessian_p])

    return kl_hessian_p_flat + damping * p
```

Listing 1: Computing F*v (Fisher Vector Product)

6.2 Conjugate Gradient

CG is used to solve $Fx = g$ to find the search direction inside the Trust Region.

```python
def conjugate_gradient(self, states, b, n_steps=10, residual_tol=1e-10, damping=0.01)
    :
    """
    Solves Fx = b for x using Conjugate Gradient.
    """
    x = torch.zeros_like(b)
    r = b.clone()
    p = b.clone()
    rdotr = torch.dot(r, r)

    for _ in range(n_steps):
        # The expensive part: calculating F*p
        Ap = self.fisher_vector_product(states, p, damping=damping)
        alpha = rdotr / torch.dot(p, Ap)
        x += alpha * p
        r -= alpha * Ap
        new_rdotr = torch.dot(r, r)
        if new_rdotr < residual_tol:
            break
        beta = new_rdotr / rdotr
        p = r + beta * p
        rdotr = new_rdotr
    return x
```

Listing 2: Conjugate Gradient Solver

6.3 TRPO Update Step

The final update step involves calculating the surrogate loss gradient, solving for the natural gradient, determining the step size β, and performing a line search.

```python
def _trpo_step(self, batch, max_kl=0.01, damping=0.001):
    # ... (Data unpacking omitted) ...

    # 1. COMPUTE SURROGATE LOSS GRADIENT (g)
    # ... (Forward pass & Advantage calculation) ...
    surr_loss = (ratio * advantages).mean()
    grads = torch.autograd.grad(surr_loss, self.actor_parameters())
    g = torch.cat([grad.reshape(-1) for grad in grads]).detach()

    # 2. CONJUGATE GRADIENT: Solve Fx = g for x
    x = self.conjugate_gradient(states, g, damping=damping)

    # 3. COMPUTE STEP SIZE (Beta)
    # We scale x so that the KL divergence is exactly max_kl
    Fx = self.fisher_vector_product(states, x, damping=damping)
    xFx = torch.dot(x, Fx)
    beta = torch.sqrt(2 * max_kl / (xFx + 1e-8))
    full_step = beta * x

    # 4. LINE SEARCH (Backtracking)
    # We ensure the update actually improves the policy and respects the KL
    constraint
    for j in range(10):
        new_step = step_frac * full_step
        new_params = old_params + new_step
        vector_to_parameters(new_params, self.actor_parameters())
        # ... (Check KL <= max_kl * 1.5 and surr_loss improvement) ...
```

Listing 3: TRPO Optimization Step

7 Experimental Setup

The training process was rigorously structured to ensure reproducibility and stability.

7.1 Hyperparameters & Training Configuration

All experiments were conducted with a batch size of 4096 interactions (frames) per update. The maximum training duration was uniformly set to 400 epochs for all three experimental configurations.

To ensure computational efficiency and prevent overfitting, a strict early stopping mechanism was employed. Training was configured to terminate automatically if the true reward failed to improve for 50 consecutive epochs. As a result of this convergence-based criterion, none of the three models reached the maximum 400-epoch limit; all training runs terminated early once the policy performance stabilized or plateaued.

7.2 Exploration Strategy (Standard Deviation)

A critical component of the training was the management of the policy's standard deviation (σ), which governs the exploration-exploitation trade-off.

- **Initialization ($\sigma \approx 1.00$):** At the start, high variance is maintained to encourage broad exploration of the state space.
- **Convergence ($\sigma \approx 0.45$):** As the policy improves, σ naturally decays. A final value of ~ 0.45 was found to be the effective lower bound; a value lower than this typically indicates premature

convergence to a local minimum (stopping exploration too early), while a value significantly higher implies the model failed to confidently select optimal actions.

7.3 Stopping Conditions

To optimize resource usage, two termination criteria were implemented:

1. **Early Stopping (Patience):** Training triggers a stop if the true reward shows no improvement for 50 consecutive epochs.
2. **Solved Threshold:** The training terminates immediately if both the shaped reward and the optimized reward exceed 900 points.

8 Reward Shaping & Velocity Control

A primary issue identified was the agent's tendency to enter curves at maximum velocity. To mitigate this, a custom reward shaping mechanism was implemented directly in the training loop.

8.1 Implementation Logic

The code supports three distinct shaping modes controlled by the `reward_shaping` parameter.

```python
if reward_shaping != 0:
    # --- REWARD SHAPING LOGIC ---
    steer = np.abs(clipped_action[0])
    gas   = clipped_action[1]
    brake = clipped_action[2]

    # 1. STABILITY PENALTY
    # Penalize simultaneous high steering and high gas.
    stability_penalty = 0.05 * steer * gas
    reward -= stability_penalty

    if reward_shaping == 2:
        # 2. BRAKE REWARD
        # Encourages braking during turns
        reward += 0.03 * brake * steer
```

Listing 4: Reward Shaping Logic from Training Loop

8.2 Comparative Analysis of Training Types

Three distinct phases of reward shaping were tested corresponding to the code logic above:

1. **Phase 1 (Baseline - `reward_shaping=0`):** No shaping. The optimization score is equivalent to the true environmental score. The agent relies solely on the sparse environmental feedback.
2. **Phase 2 (Stability Only - `reward_shaping=1`):** This mode enabled only the Stability Penalty ($0.05 \times steer \times gas$) and disabled the Brake Reward. This configuration aimed to teach the agent to "lift off" the throttle during turns without explicitly incentivizing braking.
3. **Phase 3 (Stability + Brake - `reward_shaping=2`):** This mode activated both the Stability Penalty and the Brake Reward ($0.03 \times brake \times steer$). This configuration was designed to encourage active deceleration in addition to throttle reduction during cornering.

9 Results

The following figures illustrate the training progress for each of the three evaluated approaches.

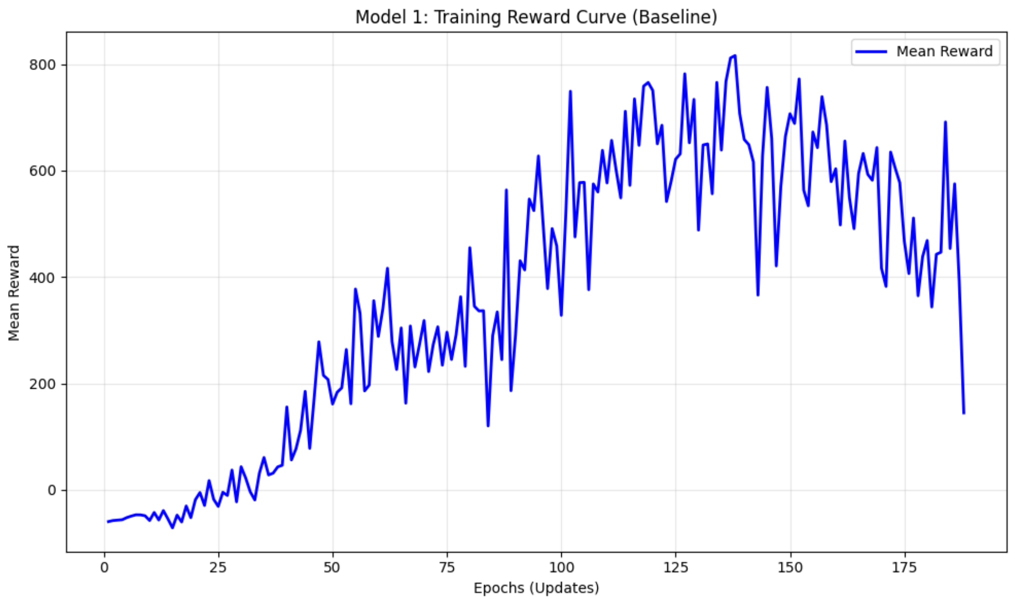

Figure 1: **Model 1 (Baseline):** The training demonstrates a standard learning curve. The agent overcomes initial high-variance exploration to achieve a robust mean reward of ~ 835, though it exhibits typical fluctuations associated with the unshaped sparse reward signal.

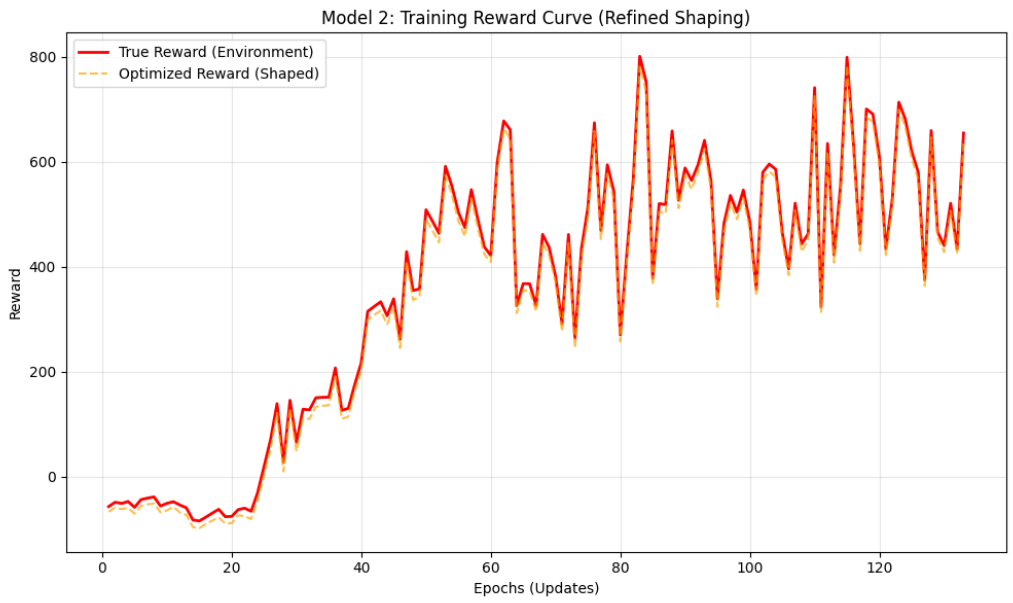

Figure 2: **Model 2 (Over-Constrained):** This model, corresponding to Phase 2, yields the poorest performance (Mean ~ 527). The overlapping curves indicate that the agent successfully minimized the "Optimized" penalty, but it did so by driving extremely conservatively (slowly) to avoid the heavy stability costs. This resulted in a failure to learn aggressive racing lines, leading to a sub-optimal True Reward.

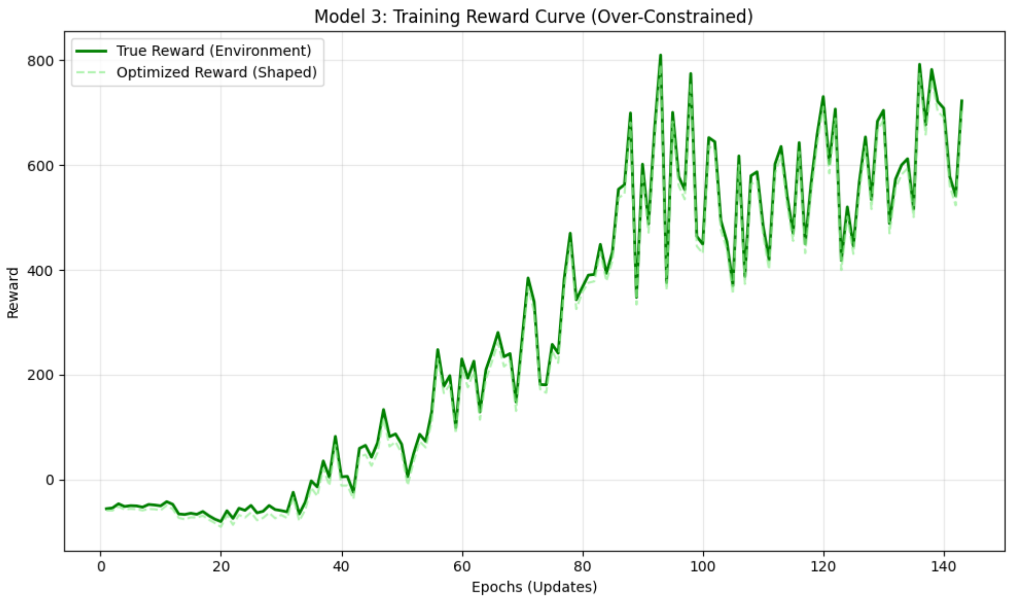

Figure 3: **Model 3 (Refined):** This model, corresponding to Phase 3, demonstrates high sample efficiency, converging and triggering early stopping at epoch 170. The visual gap between the curves is a result of the reward clipping mechanism used to stabilize the high-variance updates. Despite this, the agent successfully maximized the true objective, achieving a competitive performance similar to the baseline but in significantly fewer epochs.

The comparative results can be summarized as follows:

- **Model 1 (Baseline):** Established a strong baseline, stabilizing at a mean reward of ~ 800.
- **Model 2 (Over-Constrained):** Performing significantly worse than the baseline (plateau at ~ 600), hampered by excessive penalties.
- **Model 3 (Refined):** Achieved similar stability to the baseline (~ 800) but demonstrated higher sample efficiency by reaching convergence in just 170 epochs compared to the baseline which required significantly more epochs to stabilize.

9.1 Deviation from Theoretical Monotonicity

Theoretically, Trust Region Policy Optimization (TRPO) guarantees monotonic improvement of the expected return. However, as consistently observed in the training logs across all three evaluated models, the practical reward curves exhibit significant stochastic oscillations rather than a strictly monotonic ascent. This discrepancy arises from specific practical approximations and hardware-imposed constraints.

9.1.1 Stochastic Estimation and Hardware Constraints

The advantage is estimated using a finite batch of trajectories (4096 timesteps). Due to hardware limitations, specifically regarding GPU memory capacity, it was not possible to increase this batch size further. In a high-variance environment like `CarRacing-v2`, this limited sample size captures only a small subset of possible track configurations per update (approximately 4-5 episodes). Consequently, a policy update that maximizes the return for a specific batch may not generalize perfectly to the global distribution, causing occasional dips in the validation score. It is hypothesized that with fewer hardware constraints, increasing the number of samples per batch would significantly reduce this variance by providing a more representative estimate of the true policy gradient.

9.1.2 Approximation Errors and Advantage Accuracy

The inverse of the Fisher Information Matrix is approximated using Conjugate Gradient (10 steps), resulting in a search direction that is close to, but distinct from, the true Natural Gradient. Additionally, the Taylor expansion of the KL constraint is a local approximation that may not hold perfectly in the non-linear parameter space of a CNN. Furthermore, the update quality relies heavily on the Generalized Advantage Estimation (GAE), which depends on a learned Value function $V_\phi(s)$. Since this value function is itself an approximation, any inaccuracies in the Advantage calculation introduce noise into the gradient step. A more accurate Advantage function—achievable through more extensive training or larger effective batch sizes—would likely mitigate these approximations and yield more stable, monotonic results.

9.2 Post-Training Evaluation

Following the training phase, a rigorous evaluation was conducted by running 30 test episodes for each model to determine their generalization capabilities. The results indicate a significant divergence in performance:

- **Model 1 (Baseline):** Achieved the highest mean reward of ~ 650. This suggests that the raw environmental reward, while sparse, allows the agent to find the most optimal racing lines without being constrained by artificial biases.
- **Model 2 (Stability Only):** Performed the poorest, with a mean reward of ~ 354. The low score indicates that the stability penalty was effectively too high for complex tracks relative to the baseline. Instead of refining the trajectory, the penalty likely forced the agent into overly conservative behaviors or local minima to avoid negative shaping, sacrificing race progression.
- **Model 3 (Stability + Brake):** Achieved a mean reward of ~ 634. While this improves significantly upon Model 2, it still falls slightly short of the Baseline. The addition of the positive braking reward likely counteracted some of the conservatism induced by the stability penalty, restoring some performance but not exceeding the original unshaped approach.

These findings suggest that, with the current configuration, the **original reward function is superior**. The imposed shaping constraints, particularly in Model 2, appear to interfere with the optimal policy rather than guiding it, highlighting the difficulty of tuning penalty coefficients in sensitive continuous control tasks.

10 Conclusion

This study successfully implemented a custom Trust Region Policy Optimization (TRPO) agent, leveraging Conjugate Gradient descent to solve the continuous control challenges of `CarRacing-v2`. The experimental analysis yielded critical insights regarding the efficacy of reward shaping versus unconstrained learning.

First, the comparison between the over-constrained model and the baseline highlights the extreme sensitivity of Reinforcement Learning agents to penalty coefficients. As evidenced by Model 2, conflicting incentives (braking bonuses opposing stability penalties) drove the policy into sub-optimal local minima, resulting in the lowest evaluation performance (mean reward ~ 354).

Second, contrary to the initial hypothesis, the **original unshaped reward function (Model 1) proved to be the most effective strategy**. While the refined reward shaping (Model 3) demonstrated superior sample efficiency by converging in just 170 epochs, it achieved a slightly lower evaluation score (~ 634) compared to the baseline (~ 650). This suggests that even carefully tuned heuristic constraints can introduce bias that prevents the agent from discovering the most aggressive and optimal racing lines. The unshaped baseline, while requiring more epochs to stabilize, allowed the agent to maximize the true environmental objective without artificial hindrance.

In conclusion, the final agent demonstrates robust racing proficiency. The remaining gap to the "solved" threshold (900) is attributed to residual instability in high-curvature sections and the high variance inherent in on-policy learning with limited batch sizes ($N = 4096$). Future work addressing these hardware constraints and implementing reward normalization could likely bridge this final gap.

Bibliography

References

[1] N. Becker, *Deriving the Sigmoid Derivative for Neural Networks*, GitHub Pages, 2017. Available at: `https://beckernick.github.io/sigmoid-derivative-neural-network/`

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd Edition, MIT Press, 2018 (Section 13.3: REINFORCE: Monte Carlo Policy Gradient).

[3] Vladyslav Yazykov, "TRPO: Minimal PyTorch Implementation," *Medium*, 2020. [Online]. Available: `https://medium.com/@vladogim97/trpo-minimal-pytorch-implementation-859e46c4232e`.

[4] Mike Woodcock. *Solving CarRacing with PPO*. NotAnyMike GitHub Pages. Available at: `https://notanymike.github.io/Solving-CarRacing/` (Accessed: December 20, 2025).

[5] Felipe Gomes. *AI Car Racing Guide: Teach an AI to Drive with Python & Gym*. Flip Bits, July 2025. Available at: `https://flipbits.com.br/master-car-racing-ai-reinforcement-learning-guide/` (Accessed: December 20, 2025).