Final Project Report: PPO for CarRacing

Mahathi Nakka, 23b0965

August 2025

1 Introduction

In this project, we trained a computer agent to play a car racing game called CarRacing-v3. We used a special type of machine learning called Reinforcement Learning (RL). In RL, an agent learns how to behave in an environment by trying different actions. When the agent takes a good action, it gets a reward. If the action is bad, it may get a penalty or a smaller reward. Over time, the agent learns which actions are best to get higher rewards.

We used a technique called Proximal Policy Optimization (PPO). PPO is a modern and reliable RL method that helps the agent learn efficiently without making the training unstable. PPO works by updating the agent's brain (the policy) in small, careful steps. We used PyTorch to build our PPO model, and OpenAI Gym to simulate the racing environment.

2 Environment Setup and Preprocessing

2.1 CarRacing-v3 Environment

CarRacing-v3 is a simulation where an agent drives a car around a randomly generated track. The agent sees the world as images — each image is 96x96 pixels in size and has 3 color channels (red, green, and blue). Every frame is like a small photo showing what the car sees from above.

The agent can control the car by choosing values for three actions:

- **Steering:** This controls the direction of the car (left or right). The value can range from -1 (hard left) to 1 (hard right).
- **Gas:** This controls how much the car accelerates. Values go from 0 (no gas) to 1 (full gas).
- **Brake:** This controls braking force. Values go from 0 (no brake) to 1 (full brake).

The game gives rewards based on how well the car stays on the track and moves forward. If the car crashes or goes off the road too much, the episode ends early.

2.2 CarRacingWrapper

To make learning easier, we created a custom environment wrapper. This wrapper does several useful things:

- **Grayscale Conversion:** We turned colored images into black and white. This reduces complexity and speeds up training.
- **Image Resizing:** We reduced the image size from 96x96 to 84x84 pixels. Smaller images use less memory and are faster to process.
- **Frame Stacking:** We combined the last 4 images into one input. This gives the agent a sense of motion, helping it understand speed and direction.
- **Frame Skipping:** We repeated each action for a few frames to make learning faster.
- **Reward Shaping:** We modified the reward so that the agent gets more points for staying on the road and driving smoothly.
- **Early Stopping:** If the agent is performing badly (e.g., going off track too much), we end the game early to save time.

3 PPO Agent and Neural Network Architecture

3.1 ActorCritic Network

The brain of our agent is a neural network with two parts:

- **Actor:** This part decides what action to take based on the current state.
- **Critic:** This part estimates how good the current situation is (the expected reward).

The network first uses Convolutional Neural Network (CNN) layers to look at the stacked images and find patterns like track edges. After that, the output is sent to two separate parts:

- One part (the actor) outputs a probability distribution for the actions.
- The other part (the critic) gives a single number, which is the value of the current state.

3.2 PPOAgent Class

This is the main class that trains the agent. It uses an algorithm called Generalized Advantage Estimation (GAE) to understand which actions were better than expected. The PPOAgent makes sure that updates to the neural network are not too large, which keeps learning stable.

The PPO algorithm adjusts how the agent behaves by comparing how it performed before and after the update. If the new behavior is much worse than the old one, PPO cancels the update. This is done using something called a "clip ratio."

3.3 PPOBuffer Class

This class stores the agent's experiences during each episode. For each step in the environment, it keeps track of:

- The state (input image)
- The action taken
- The reward received
- The value estimate from the critic
- The log probability of the action (used for training)

After an episode is done or after enough steps, the buffer calculates two important things:

- **Advantage:** How much better the action was compared to what was expected.
- **Return:** The total reward expected in the future starting from that step.

4 PPO Algorithm Details

4.1 Loss Functions

We use three kinds of loss functions to train the network:

- **Policy Loss:** Helps the agent choose better actions.
- **Value Loss:** Helps the critic make better reward predictions.
- **Entropy Loss:** Encourages the agent to try different actions and not always do the same thing.

4.2 Training Process

The PPO algorithm works by collecting experiences (states, actions, rewards) and then updating the policy based on that data. The update happens in small batches to make it more stable. For each batch, the network is trained several times (called epochs). This helps the network learn better from the same data.

Each update step checks whether the policy change is too large. If it is, the update is limited to avoid harming the agent's performance. This is the main idea behind PPO: improving the policy, but not too much at once.

5 Training Setup and Hyperparameters

5.1 Training Loop

Here is how we trained the PPO agent:

1. The environment is reset and the first image is observed.
2. The agent plays the game for 2048 steps, saving its actions and rewards.
3. After collecting the data, the buffer calculates advantages and returns.
4. The PPOAgent updates the neural network using this data.
5. If the agent is performing better than before, we save the model.

This process is repeated for many episodes (training cycles) until the agent learns to drive well.

5.2 Hyperparameters Used

These are the values we chose for different settings in our model:

- **Learning rate:** 2.5×10^{-4} — how fast the network learns
- **Discount factor (γ):** 0.99 — how much future rewards are considered
- **GAE lambda:** 0.95 — smooths advantage estimates
- **Clip ratio:** 0.2 — limits big updates
- **Entropy coefficient:** 0.01 — encourages exploration
- **Steps per update:** 2048
- **Epochs per update:** 10
- **Mini-batch size:** 64

6 Evaluation and Visualization

After training, we tested the agent to see how well it performs. During testing, we removed randomness to make the agent perform its best. We ran the agent for several episodes and recorded its scores.

We also saved videos to visually check how well the car drives. A good model often scores over 900. We also plotted graphs of the total reward per episode to see the improvement over time. These graphs showed that the agent was learning consistently.

7 Results and Observations

- In the beginning, the agent drove off the track often.
- Over time, the agent learned to stay on the road and drive smoothly.
- The gas and brake usage became more balanced.
- The model reached average scores above 900 after several thousand steps.
- We saw steady improvement in the reward plot over episodes.

8 Conclusion

This project showed how PPO can be used to train an agent to drive a car in a simulated environment. We combined several smart techniques:

- A well-designed neural network
- A helpful environment wrapper
- Carefully chosen hyperparameters
- Stable and reliable PPO algorithm

Our PPO agent was able to learn driving skills without any human help. This shows the power of deep reinforcement learning. The project also helped us understand how PPO works and how to implement it from scratch.