Traffic Simulation RL Training Strategy: PPO

AI Assistant

April 9, 2025

1 Business Understanding

- **Objective:** Optimize traffic light control in a simulated intersection to reduce congestion (waiting time), prevent crashes, minimize emissions, and maximize vehicle throughput (average speed). This objective remains the same as with previous approaches.
- **Problem:** While REINFORCE offered improvements over basic reward aggregation, Proximal Policy Optimization (PPO) is often more stable and sample-efficient, potentially leading to faster and more robust learning by constraining policy updates. Initial observations suggest PPO is yielding better results than the REINFORCE implementation in this specific environment.
- **Proposed Solution:** Utilize the Proximal Policy Optimization (PPO) algorithm with an Actor-Critic architecture. The Actor network learns the policy (traffic light control), and the Critic network learns a value function to estimate the expected return from a given state, aiding in advantage calculation.

2 Data Understanding

- **State (S_t):** A `(4, 20, 5)` NumPy array derived from sensor data (vehicle distance, speed, type, acceleration, waiting time) in 4 zones. Processed by the `get_state` method in `neural_model_01.py`. Captures the situation *before* an action is taken.
- **Action (A_t):** A multi-label binary vector (length 4) indicating active traffic lights (e.g., `[1, 0, 1, 0]` means lights 0 and 2 are active). This is sampled based on the Actor network's output probabilities.
- **Actor Output (Policy $\pi_\theta(A|S)$):** The Actor network outputs *logits* for each of the 4 actions. A sigmoid function is applied to these logits to obtain probabilities $p = [p_0, p_1, p_2, p_3]$ for each light being active. The action A_t is then typically sampled stochastically based on these probabilities (e.g., comparing p_i to a random uniform number).
- **Critic Output (Value Function $V_\phi(S)$):** The Critic network outputs a single scalar value representing the estimated expected discounted return from state S, parameterized by weights ϕ.
- **Reward (R_t+1):** Calculated *after* action A_t is taken and the simulation advances one step. It reflects the immediate outcome of the action and the resulting state S_{t+1}. The **per-step reward function** (`calculate_step_reward`) computes this scalar value based on metrics *from the completed step*: Average vehicle speed, waiting vehicles, crashes, and emissions.

3 Data Preparation

- The existing `get_state` function handles the conversion of raw simulation observations into the normalized, fixed-size state representation required by both Actor and Critic models. This remains suitable for the PPO implementation.

4 Modeling (PPO Algorithm)

- **Algorithm Choice:** Proximal Policy Optimization (PPO) with an Actor-Critic framework. PPO is a policy gradient method that improves stability by using a clipped objective function.
- **Architecture:** Two separate neural networks:
  - **Actor (Policy Network π_θ):** Takes state S_t as input, outputs action logits. Trained to maximize the PPO objective.
  - **Critic (Value Network V_ϕ):** Takes state S_t as input, outputs an estimate of the state value $V(S_t)$. Trained to minimize the Mean Squared Error (MSE) between its prediction and the calculated target returns.
- **Key Concepts:**
  - **Advantage Estimation (GAE):** Instead of using just the return G_t, PPO typically uses an Advantage function A_t which estimates how much better an action A_t is compared to the average action in state S_t. Generalized Advantage Estimation (GAE) is commonly used:

    $$\hat{A}_t^{GAE} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

    where $\delta_t = R_{t+1} + \gamma V_\phi(S_{t+1}) - V_\phi(S_t)$ is the TD error, γ is the discount factor (e.g., 0.99), and λ is the GAE parameter (e.g., 0.95). In practice, this sum is calculated over the finite batch trajectory.
  - **Probability Ratio:** $r_t(\theta) = \frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{old}}(A_t|S_t)}$, where θ_{old} are the policy parameters before the update.
  - **Clipped Surrogate Objective (Actor Loss):** PPO maximizes a clipped objective to prevent excessively large policy updates:

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

    where ϵ is the clipping hyperparameter (e.g., 0.2). The loss function used in gradient descent is the negative of this objective: $L_{actor} = -L^{CLIP}(\theta)$. The implementation calculates $\log \pi_\theta(A_t|S_t)$ using the sum of binary cross-entropies for the multi-label action, similar to REINFORCE, and computes the ratio $r_t(\theta)$ using $\exp(\log \pi_\theta(A_t|S_t) - \log \pi_{\theta_{old}}(A_t|S_t))$.
  - **Value Function Loss (Critic Loss):** The Critic is trained by minimizing the mean squared error between its value estimates $V_\phi(S_t)$ and the calculated target returns (often the GAE returns $G_t^{GAE} = \hat{A}_t^{GAE} + V_\phi(S_t)$, though sometimes empirical Monte Carlo returns are used):

    $$L_{critic}(\phi) = \mathbb{E}_t[(V_\phi(S_t) - G_t^{target})^2]$$

    The implementation uses $G_t^{target} = \hat{A}_t^{GAE} + V_{\phi_{old}}(S_t)$ where $V_{\phi_{old}}$ are the values predicted by the critic before the update batch.
  - **Entropy Bonus (Optional):** Sometimes an entropy term is added to the actor loss to encourage exploration, though not explicitly shown in the provided `_compute_ppo_losses` function.
- **Training Update:**
  1. Collect a batch of experience tuples $(S_t, A_t, R_{t+1}, S_{t+1}, \log \pi_{\theta_{old}}(A_t|S_t), V_{\phi_{old}}(S_t))$ using the current policy $\pi_{\theta_{old}}$ and value function $V_{\phi_{old}}$.
  2. Calculate advantage estimates $\hat{A}_t$ (using GAE) and returns G_t^{target} for the batch.
  3. Repeat for `TRAIN_EPOCHS_PER_UPDATE`:
     - Sample mini-batches from the collected batch (or use the whole batch).
     - Compute the Actor loss L_{actor} and Critic loss L_{critic}.
     - Update Actor parameters θ using gradient descent on L_{actor}.
     - Update Critic parameters ϕ using gradient descent on L_{critic}.
     - Optionally, perform early stopping based on KL divergence between π_θ and $\pi_{\theta_{old}}$.

5 Evaluation

- **Metrics:** Continue tracking epoch-level performance using aggregated metrics (sum of step rewards, average speed, total crashes, total waiting steps, total emissions). Monitor the trend of these metrics across epochs, comparing PPO's learning curve to previous methods. Also monitor internal PPO metrics like Actor/Critic loss and approximate KL divergence during training updates.
- **Qualitative Assessment:** Utilize the simulation's rendering capability (controlled by `render_interval`) to visually inspect the traffic light control policy learned by the agent. Observe if traffic flow appears smoother and if the agent adapts reasonably to different traffic densities compared to the REINFORCE agent.

6 Implementation Details (`neural_model_01.py` - PPO)

- `__init` $.Initializeshyperparameters($`GAMMA`, `GAE_LAMBDA`, `CLIP_RATIO`, $learningrates, etc.), replaybuffer($`deque`$), Actor/C$
- `get_action_and_value`: **Takes a state, gets action logits from Actor and value from Critic. Calculates action probabilities (sigmoid), samples an action, computes the log probability of the sampled action, and returns the action, log probability, and state value.**
- `calculate_step_reward`: **Computes the immediate reward based on simulation metrics from the completed step.**
- `store_transition`: **Appends the** $(S_t, A_t, R_{t+1}, S_{t+1}, \textbf{done}, \log \pi_{\theta_{old}}(A_t|S_t), V_{\phi_{old}}(S_t))$ **tuple to the replay buffer.**
- `_compute_ppo_losses`: **Central PPO loss calculation logic within a** `tf.GradientTape` **context. Computes probability ratios, clipped surrogate objective (Actor loss), and value function MSE (Critic loss). Returns losses and gradients.**
- `learn_ppo`: **Orchestrates the PPO learning update. Triggered periodically (**`UPDATE_EVERY_N_STEPS`**). Samples data from the buffer, calculates GAE and returns, normalizes advantages, and runs the optimization loop (calling** `_compute_ppo_losses` **and applying gradients for** `TRAIN_EPOCHS_PER_UPDATE`**). Includes KL-based early stopping.**
- `update` **(Step Function): This method is called each simulation step. It gets the current state, calls** `get_action_and_value` **to determine the next action, calculates the reward for the *previous* transition, stores the completed transition (using** `store_transition`**), updates internal step counters, and triggers** `learn_ppo` **if the step threshold is met. Returns the action for the simulation to execute.**
- `end_epoch`: **Primarily handles epoch-level reporting (printing summary statistics) and manages the rendering logic (pausing with spacebar). Training logic is decoupled into** `learn_ppo`**.**