Instruction for Project 2

Reinforcement Learning for RC car racing

1. Introduction

In this assignment, you will develop an agent capable of autonomously driving an RC car using the **Proximal Policy Optimization (PPO)** reinforcement learning algorithm. Unlike Behavior Cloning (Project 1), which cloned expert demonstration data, PPO involves an agent learning a **policy** on its own by directly interacting with the simulation environment.

The agent learns through extensive trial and error, aiming to maximize the **cumulative reward**. Your goal is to implement the PPO algorithm, design an appropriate reward function, and finally tune hyperparameters to effectively train an agent that can complete the track quickly and stably without driving off.

2. Policy-Based RL

Unlike DQN, where its main objective is to learn the Q-values, in policy-based methods, the main objective is to learn the policy π directly. In this case, we use a new neural network called the "policy network", which is also called the **Actor**. This Actor network takes the current state s as input and outputs a probability distribution over possible actions, $\pi(a|s)$, directly deciding what to do. The network is trained using a method called **Policy Gradient**, which intuitively updates the policy based on outcomes: if an action leads to a good result (e.g., a high total reward R), the network is updated to increase that action's probability. This method is highly effective for continuous action spaces (like steering angles) where DQN's Q-value approach is infeasible.

However, relying on the total reward R causes a **high variance** problem, as "good" or "bad" may be relative which results in unstable learning. The solution to this problem is the **Actor-Critic** architecture, which introduces a **Critic** network to learn the state-value function $V(s)$, or the expected average reward from that state. Instead of using R, the Actor is now guided by the **Advantage Function** A, which measures how much *better* or *worse* an action was compared to the average expectation (e.g., $A = r + \gamma V(s') - V(s)$). If $A > 0$, the Actor increases the action's probability and if $A < 0$, the Actor decreases the probability.

3. PPO Algorithm

In the Actor-Critic architecture, there still lies a problem. In cases where the Actor is updated as in a single overly large update, this may potentially destroy the policy. To solve this problem, Proximal Policy Optimization (PPO) implements a simple clipping process where it limits large policy updates.

This process begins by running the current policy $\pi_{\theta_{old}}$ to collect a batch of transitions. For each transition, the algorithm estimates how much better an action was than the expected average, using an **Advantage function, $\widehat{A_t}$.** This is typically calculated using Generalized Advantage Estimation (GAE), which computes a variance-reduced estimate based on the temporal difference (TD) errors: $\delta_t = r_t + \gamma V(s_{t+1})(1 - d_t) - V(s_t)$,

$$\widehat{A_t} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \, \delta_{t+l}$$

where d_t is a done signal, λ is the GAE parameter and again, $V(s_t)$ is the state-value function. The returns, $R_t = \widehat{A_t} + V(s_t)$, are also calculated to serve as the target for the value function update.

Once the data is collected, PPO optimizes its policy and value networks for several epochs. The core of PPO is its "clipped surrogate objective function." This function relies on the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$, which measures how good the new policy π_θ's probability for an action is compared to the old policy's. The policy objective is then defined as:

$$L^{CLIP}(\theta) = \widehat{E_t}\left[\min\left(r_t(\theta)\widehat{A_t},\ clip(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon)\widehat{A_t}\right)\right]$$

Here, ϵ is a small hyperparameter (e.g., 0.2) that defines the "clipping" range. This objective takes the minimum of the standard policy gradient objective ($r_t(\theta)\widehat{A_t}$) and a version where $r_t(\theta)$ is clamped within $[1-\epsilon, 1+\epsilon]$. This clipping ensures that when an action is much better than expected ($\widehat{A_t} > 0$), the policy update is capped, preventing it from becoming too aggressive. Likewise, if an action is much worse ($\widehat{A_t} < 0$), the update is also bounded. The final loss function to be minimized combines this policy loss, a value function loss $L^{VF} = (V_\theta(s_t) - R_t)^2$, and an entropy bonus S to encourage exploration:

$$L(\theta) = \widehat{E_t}[-L^{CLIP}(\theta) + c_1 L^{VF}(\theta) - c_2 S[\pi_\theta](s_t)]$$

By minimizing this combined loss (where c_1 and c_2 are coefficients), PPO achieves stable, reliable policy updates without the complex mathematical overhead of its predecessors.

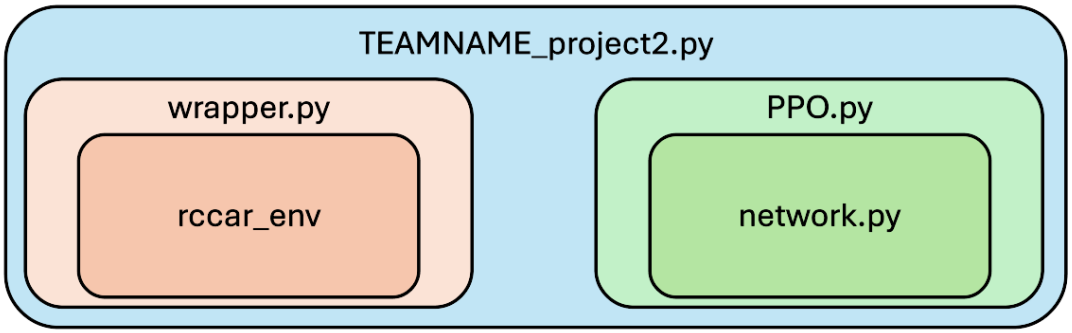
The basic pseudo code is as follows.

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

4. Code Implementation: TODO

1. Basic Code Structure of Project 2



TEAMNAME_project2.py calls wrapper.py to create a custom wrapped setting for the existing rccar_env within the rrcar_gym. Within the wrapper, you will implement reward functions and any additional variables or parameters needed. For both training and evaluation, TEAMNAME_project2.py also calls the **PPO.py** code for the actual PPO algorithm and its RL model. Here, the PPO will again call **network.py** to implement the Actor-Critic network architecture. You will be required to fill in the PPO algorithm and network architecture. Further details are provided regarding each code below.

2. [TEAM_NAME]_project2.py

The main code you execute to train or evaluate the PPO agent policy.

You can define additional methods and classes or modify the existing ones.

a. *get_args* function

You can add or change any arguments as you want.

e.g.: *--model_name* to load or save, *--mode* to train or validate the model, *--checkpoint_freq* to adjust the frequency of saving intermediate models, etc.

(Optionally, you can use *--wandb* or *--tb* options to monitor your training progress.)

b. *RCCarPolicy* class

You can add or change the attributes of *RCCarPolicy*, to support other methods.

Also, you can add or modify the methods (**except *query_callback***) as you want.

i. *train* method

Initialize the vectorized environment of RCCar and the *PPO* agent to train and then start training. You can also wrap your *vec_env* with other wrappers.

ii. *load* method

Loads the trained model specified by --*model_name*, which is used by the *get_action* method.

iii. *get_action* method

Predicts an action from the current lidar observation (*scan_obs*) using the loaded PPO model. It is recommended to pre/post-process observations and actions within the model's methods (from the PPO class at algo.py)

3. wrapper.py

a. *RCCarEnvTrainWrapper* class

This class wraps the RCCar environment during the training process, to customize desired reward function.

i. *__init__* method

Initialize your own attributes to support reward computation.

ii. *reset* method

Reset your own attributes to support reward computation.

iii. *step* method

Design your own **reward** function for this RC Car environment.

4. algo.py

a. *PPO* class

Main implementation of the training and inference of your PPO model. You can use, modify, or ignore any methods here.

i. *__init__* method

Initializes the PPO agent by storing hyperparameters like *learning_rate*, *gamma*, *n_steps*, and *ent_coef* as class attributes. Refer the original paper and these two posts to see the functionality of each hyperparameter.

Also constructs the *ActorCritic* network (*self.policy*), sets up the Adam optimizer to train its parameters, and pre-allocates the fixed size numpy buffers (e.g., obs_buffer, rewards_buffer) required for efficient data collection during the rollout phase.

And the key hyperparameters are backed up in *self.kwargs* to ensure the model can be correctly saved and reloaded later by the *save* and *load* method. (If you add some attributes, then it must be in *self.kwargs* to be loaded for evaluation)

ii. *learn* method

Implements the entire training process

Main loop: continues until the number of total environment interactions reaches *total_timesteps*.

The first phase: data collection, or "rollout"

A nested loop runs a fixed number of steps, allowing the agent's current policy to interact with vectorized environments. At each step, it collects observations, actions, rewards, done signals, and the log probabilities of the policy and value estimates – into large, pre-allocated buffers. (HINT: *_select_action* method and *self.env.step* function would be helpful)

The second phase: calculating the advantages and returns.

This process works backward in time, starting from the last step of the rollout. First, it estimates the value of the final observation to bootstrap the return calculation. Then, it reversely iterates through the collected data, applying the GAE formula to compute the advantage value for every time step. These advantages are stored in a new buffer, along with the calculated returns.

The final phase: updating the model for *n_epochs* over the collected dataset.

In each epoch, the data is first randomly shuffled, then divided into smaller mini-batches (size: *batch_size*). For every mini-batch, the agent re-evaluates the old observations and actions using its current policy. This

provides the new log probabilities, new value estimates, and the current policy entropy, which are necessary to compute the PPO loss function. (HINT: *_evaluate_actions* method would be helpful)

For each mini-batch, the policy (actor) loss is calculated using the "clipped surrogate objective." Simultaneously, the value (critic) loss and the entropy loss are calculated. These losses are combined into a single loss function. Then the optimizer uses this final loss to update the weights of the actor and critic networks, completing one update step. (HINT: zero_grad(), step() of the self.optimizer, backward() of torch tensor of loss, and *nn.utils.clip_grad_norm_* function (with *max_grad_norm*) would be helpful)

iii. *predict* method

Samples actions from your own network, *self.policy*, which is defined in the *network.py*

iv. *_process_obs* method

Preprocesses your observations for the input of the network.

v. *_process_act* method

Postprocesses your actions for the output of the model.

vi. *_log* method

Helps you log metrics to monitor your training progress.

5. network.py

a. *ActorCritic* class

Design your own networks for the PPO model.

i. *__init__* method

Design your own Actor network (for policy) and Critic network (for value). For the Actor network, you need to declare not only the network that outputs the mean of the action, but also the parameter for the standard deviation of the action.

(HINT: use *nn.Sequential* for the value and the (mean of) action network, and *nn.Parameter* for the std of action)

ii. *forward* method

Performs the forward pass, feeding observations through your networks to get the policy outputs and the value estimate.

5. Running Codes

Project Repository

"git pull" in the ~/**Intelligent-Systems-RLLAB/** folder for updated project files

[TEAM_NAME]_project2.py

When you want to train a new model or evaluate an existing model, you can run project2 code with "--mode" argument.

- Training

```
ros2 run rccar_bringup RLLAB_project2 --mode train
# Replace RLLAB with your team name
```

- Evaluation

```
ros2 run rccar_bringup RLLAB_project2
# Replace RLLAB with your team name
```

if you want to load your model with a different name for evaluation

```
ros2 run rccar_bringup RLLAB_project2 --model_name <YOUR_MODEL_NAME.pt>
# Replace RLLAB with your team name and YOUR_MODEL_NAME with the corresponding model name
```

6. Grading Criteria for Project2

Grading for Project 2 will be based on how many maps a single query can finish completely. For example, if your team has submitted two queries where the first query managed to finish maps 1~4 and the second query finished maps 5~10, your grade will be calculated upon the second query where it finished 6 maps successfully.

However, in the next final project, where you will be competing against other teams, ranking will be one of the main criteria for scoring so try your best to speed up your model.

7. Cautions

Before you run the codes, you must add a new entry point for RLLAB_project2 in your setup.py file as follows.

```
'RLLAB_project2 = rccar_bringup.project.IS_RLLAB.project.RLLAB_project2:main'
# Replace RLLAB with your team name
```

Manually publishing map topic

For each project code, you can publish "/query" topic manually using following command in another terminal.

```
ros2 topic pub --once /query message/msg/Query "{id: '0', team: 'RLLAB', map: 'map1', trial: 0, exit: false}"
# Replace RLLAB with your team name and you can use other maps we provide in 'maps' directory
```

Note that you can publish the topic once with "--once" argument.

8. Submission format

To evaluate your code in a local PC, follow the instructions described in the class repository(https://github.com/rllab-snu/Intelligent-Systems-RLLAB.git).

Commit and push your codes on github and make a query on our project web server.

Project 2 is due on **11/24(Mon) at 23:59 KST**, and late queries will be given penalties as informed.