Instruction for Project 1

Behavior Cloning for RC car racing

1. Introduction

In this assignment, you will develop proficiency in using a behavior cloning (BC) algorithm known as Gaussian Process Regression (GPR) to control an RC car. GPR requires gathering expert demonstrations for training purposes. You should collect these demonstrations using the controller developed in the previous pre project (pre project3). Afterward, you need to train the GPR model you've defined with these expert demonstrations.

2. Algorithm

Figure 1 provides a brief overview of BC for a general configuration space, assuming the environment as a Markov Decision Process (MDP). This includes a set of states S, a set of actions A, and a transition model $P(s, a)$ that describes the environment. The notation τ represents the agent's policy, while $\tau^* = (s_0^*, a_0^*, s_1^*, a_1^*, \dots)$ depicts the trajectories generated through expert demonstrations. With τ^* provided, we treat the state-action pairs as independent and identically distributed (iid) and apply GPR to model these relationships.

1. Collect demonstrations (τ^* trajectories) from expert
2. Treat the demonstrations as i.i.d. state-action pairs: $(s_0^*, a_0^*), (s_1^*, a_1^*), \dots$
3. Learn π_θ policy using supervised learning by minimizing the loss function $L\big(a^*, \pi_\theta(s)\big)$

Figure 1. Pseudo code of BC

3. Basics for Imitation learning

Imitation learning, also known as behavior cloning, is a widely used approach in both machine learning and deep learning. It is essentially identical to supervised learning in that each input is paired with a correct label, which in this case comes from expert demonstrations.

The goal is to learn a policy from the given data that performs well not only on the training set but also on unseen test data.

However, achieving high performance on both is generally difficult due to an inherent trade-off between fitting the training data and generalizing to new situations.

Bias-Variance Trade-off

The bias–variance trade-off explains how the complexity of a model affects its ability to learn and generalize. A simple model (with low capacity) usually cannot capture all the important patterns in the data. This leads to high bias and low variance, meaning the model makes systematic errors but behaves consistently. On the other hand, a complex model (with high capacity) can represent a wide range of functions and may fit the training data almost perfectly. However, it can also start fitting the noise within the data, which causes high variance in its predictions. This situation is called overfitting — the model performs well on the training data but fails to generalize to unseen data because it has essentially learned the noise rather than the true underlying relationship. In practice, good generalization requires finding a balance between bias and variance, so that the model is neither too simple nor too complex.

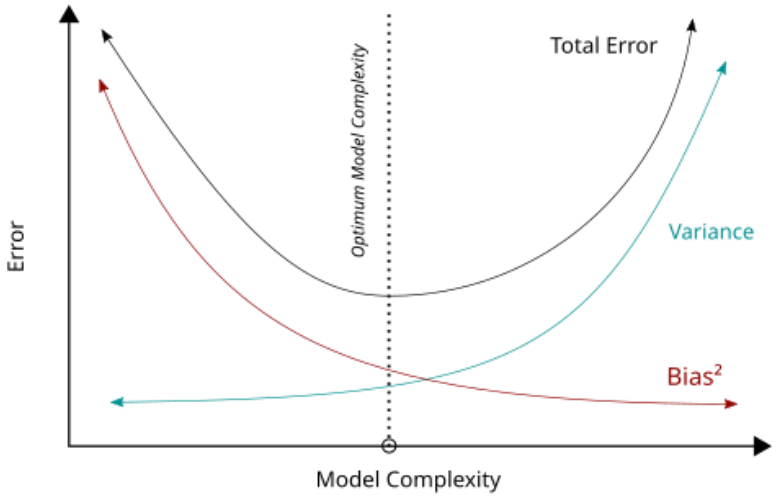

Figure 2. Trade off relation of bias and variance with respect to the model complexity[1]

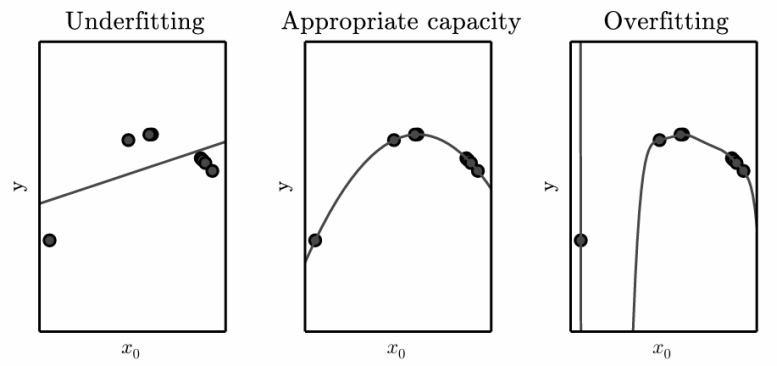

Figure 3. cases of overfitting to underfitting

[1] (Wikipedia, 2025)

Followings are the steps of the BC algorithm that you need to implement in Project 1.

Step 1. Collect demonstration observations and actions using the pure pursuit algorithm from pre project 3.

Step 2. Load expert's demonstration observations and actions. Normalize this data by calculating their mean and standard deviation, and shuffle the data to minimize correlation. We recommend using the scikit-learn library for coding these processes. **Don't forget to pre-process the data for better performance!**

Step 3. Fit the GPR model using the normalized and shuffled demonstration data as targets.

Step 4. Start racing! Continue predicting actions from the fitted GPR model until the environment terminates.

3. Explanation of Code

[TEAM_NAME]_project1.py

We have specified the directory (IS_[TEAM_NAME]/project/trajectory) where your demonstration observations and actions should be stored. However you can modify this part if you want. We assume that you have saved the scan values as observations and the steering and speed values as actions in pre-project 3. The scan values consist of 720-dimensional lidar sensor readings that correspond to angles from -90° to 90°.

You can select the model by using the '--model_name' argument and decide whether to train a new model or load a previously trained one using the '--mode' argument. The paths for the demonstration data and the trained model are specified as 'IS_[TEAM_NAME]/project/trajectory' and 'IS_[TEAM_NAME]/project/model,' respectively. The 'GaussianProcess' class encapsulates the behavior cloning model along with functions to either train or load it.

GaussianProcess.__init__()

Initialize the GPR model. You have the option to customize it by modifying the RBF kernel or adjusting other parameters. You can use other kernels if you want.

GaussianProcess.train()

Load the expert demonstration data and fit into the defined GPR model. Calculate the mean and standard deviation of both demo observations and actions to normalize those values. To break the correlation among data sequences, **shuffle the orders**. Save the trained model and configuration

for pre/post-processing.

GaussianProcess.load()

Load trained model and configurations of pre/post-process to be used for get_action().

GaussianProcess.get_action()

Predict the action from current observation using the GPR model specified by "model_name". **Don't forget to pre-process the current observation and post-process the action output.**

4. Running codes

[TEAM_NAME]_project1.py

When you want to train new model and evaluate, you can run project1 code with "--mode train" argument.

```
ros2 run rccar_bringup RLLAB_project1 --mode train
# Replace RLLAB with your team name
```

When you want to load trained model without training, you can just run without "--mode" argument since its default value is "val".

```
ros2 run rccar_bringup RLLAB_project1
# Replace RLLAB with your team name
```

[map_generation] random_trackgen.py

You can change the parameters defined in random_trackgen.py and randomly generate your own map with --seed argument. You can run random_trackgen.py using following command.

```
cd Intelligent-Systems-RLLAB/Intelligent-Systems-2025-Project/maps
python random_trackgen.py --seed your_seed --name your_map_name
```

Caution

Before you run the codes, you must change the entry point for RLLAB_project1 in your setup.py file as follows.

```
'RLLAB_project1 = rccar_bringup.project.IS_RLLAB.project.RLLAB_project1: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.

5. 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).

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

Project 1 is due on **11/7(Mon) at 23:59 KST**, and the late query will not be submitted.