SafeBench

SafeBench Team

Mar 06, 2023

OVERVIEW

1 Safebench Components **1**

2 Challenge Tracks **3**
- 2.1 Track 1 (Perception Attack) . . . 3
- 2.2 Track 2 (Perception Defense) . . . 4
- 2.3 Track 3 (Planning Attack) . . . 4
- 2.4 Track 4 (Planning Defense) . . . 4

3 Reference **5**

4 Installation **7**
- 4.1 Step 1. Setup Safebench . . . 7
- 4.2 Step 2. Setup Carla . . . 7
- 4.3 Potential Issue . . . 8

5 Run Example **9**
- 5.1 Option 1. Desktop Users . . . 9
- 5.2 Option 2. Remote Server Users . . . 9

6 How to Create Agent **11**
- 6.1 Create Perception Agent . . . 11
- 6.2 Create Planning Agent . . . 11

7 How to Create Scenario **13**
- 7.1 Create Perception Agent . . . 13
- 7.2 Create Planning Agent . . . 13

8 Hosting a Challenge via EvalAI **17**
- 8.1 Step 1: Set Up Main Repo . . . 17
- 8.2 Step 2: Set Up Challenge Configuration . . . 17
- 8.3 Step 3: Define Evaluation Code . . . 18
- 8.4 Step 4: Edit Challenge HTML Templates . . . 21

9 How to Submit Results **23**

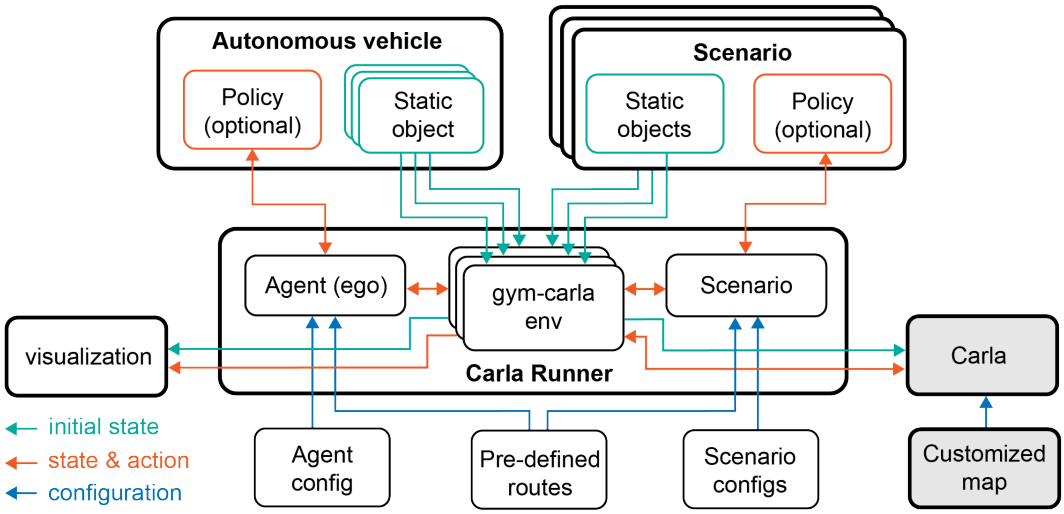
10 Modules and Functions **25**
- 10.1 Carla Runner (safebench/carla_runner.py) . . . 25
- 10.2 Gym module (safebench/gym_carla/) . . . 25
- 10.3 Agent Module (safebench/agent/) . . . 25
- 10.4 Scenario Module (safebench/scenario/) . . . 25

CHAPTER

ONE

SAFEBENCH COMPONENTS

Safebench is a benchmarking platform based on Carla simulator for evaluating safety and security of autonomous vehicles. The info flowchart of different modules is shown below:



The design of Safebench seperates objects and the policies that control objects to achieve flexible training and evaluation. Three importort features make Safebench different from existing autonomous driving benchmarks:

- **Efficient Running**

Safebench supports parallel running of multiple scenarios in one map, which dramatically increases the efficiency of colletcing training data and evaluation.

- **Comprehensive Scenarios**

Safebench intergrates lots of scenario-generation algorithms, including data-driven generation, rule-based design and adversarial example.

- **Usage Coverage**

This benchmark supports three types of running:

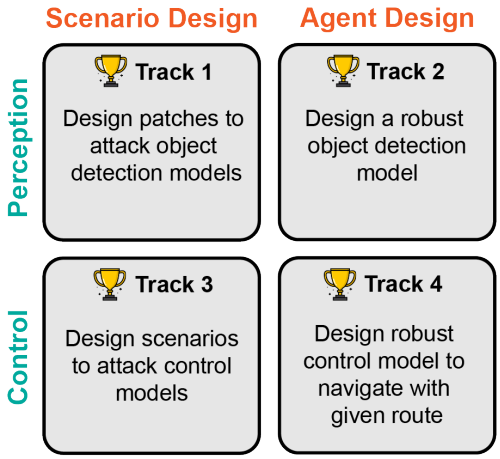
1. Train Agent: train policy model of autonomous vehicle with pre-defined scenarios.
2. Train Scenario: train policy model of scenario participants against pre-trained autonomous vehicles.
3. Evaluation: fix both autonomous vehicle and scenario to do evaluation.

CHAPTER

TWO

CHALLENGE TRACKS

In the challenge, we prepare 4 tracks covering the safety and security of both perception and control modules in autonomous driving. Two of the tracks aims to develop robust agents to deal with potential safety and security problems in scenarios. The other two tracks focus on designing scenarios that can attack agents. The detailed description of these tracks are introduced below.



2.1 Track 1 (Perception Attack)

2.1.1 Mission

This track still focuses on the object detection task but the goal is to design texture of traffic objects to attack object detection models.

2.1.2 Metric

The metrics we evaluate are the IoU and accuracy of the detection results.

2.2 Track 2 (Perception Defense)

2.2.1 Mission

This track focuses on the object detection task for front view images. There are many powerful off-the-shelf models but they may not be strong enough to defense attacks happen traffic scenarios, e.g., a mask on stop sign. In this track, the participants are required to design a robust object detection model. This model will be evaluated under scenarios that contains attacks on textures of stop sign and surrounding vehicle.

TODO: add one screenshot

2.2.2 Metric

The metric we evaluate is the IoU and accuracy of the detection results.

2.3 Track 3 (Planning Attack)

2.3.1 Mission

This track still focuses on control tasks but the goal is to design scenarios that can make agents either fail to finish the route or collide with other objects.

2.3.2 Metric

The metrics we evaluate are collision rate, out of road length, distance to driving route, incomplete driving route, and running time. We will calculate a final score based on the 5 metrics.

2.4 Track 4 (Planning Defense)

2.4.1 Mission

This track focuses on evaluating the safety and security of control models that follow a given route. We assume that the agent can access perfect observation, including the position and surrounding vehicles, traffic light status, and planning route. The goal is to design robust agents that can avoid collision and finish route.

TODO: add one screenshot

2.4.2 Metric

The metrics we evaluate are collision rate, out of road length, distance to driving route, incomplete driving route, and running time. We will calculate a final score based on the 5 metrics.

CHAPTER

THREE

REFERENCE

The source code of Safebench is built on several open-source library:

- carla
- gym-carla
- ScenarioRunner

More details about Safebench can be found in our Neurips 2022 paper. If you find Safebench useful in your research, please consider citing:

```
@article{xu2022safebench,
  title={SafeBench: A Benchmarking Platform for Safety Evaluation of Autonomous Vehicles},
  author={Xu, Chejian and Ding, Wenhao and Lyu, Weijie and Liu, Zuxin and Wang, Shuai and He, Yihan and Hu, Hanjiang and Zhao, Ding and Li, Bo},
  journal={Advances in Neural Information Processing Systems},
  volume={36},
  year={2022}
}
```

The source code of Safebench is contributed by:

- Wenhao Ding, CMU
- Chejian Xu, UIUC
- Haohong Lin, CMU
- Shuai Wang, CMU
- Jiawei Zhang, UIUC
- Weijie Lyu, CMU
- Shiqi Liu, CMU
- Zuxin Liu, CMU

CHAPTER

FOUR

INSTALLATION

We provide a detailed instruction of how to install Safebench. The installation does not require docker or ROS.

4.1 Step 1. Setup Safebench

We recommand using anaconda for creating a clean environment.

```
conda create -n safebench python=3.8
conda activate safebench
```

Then, clone the code from github in an appropriate folder with

```
git clone git@github.com:trust-ai/SafeBench_v2.git
```

Enter the folder of safebench and install some necessary packages

```
cd SafeBench_v2
pip install -r requirements.txt
pip install -e .
```

4.2 Step 2. Setup Carla

Download our built CARLA_0.9.13 and extract it to your folder with

```
mkdir carla && cd carla
tar -zxvf CARLA_0.9.13-2-g0c41f167c-dirty.tar.gz
```

Add the Python API of Carla to the `PYTHONPATH` environment variable. You can add the following commands to your `~/.bashrc`:

```
export CARLA_ROOT={path/to/your/carla}
export PYTHONPATH=$PYTHONPATH:${CARLA_ROOT}/PythonAPI/carla/dist/carla-0.9.13-py3.8-
↪linux-x86_64.egg
export PYTHONPATH=$PYTHONPATH:${CARLA_ROOT}/PythonAPI/carla/agents
export PYTHONPATH=$PYTHONPATH:${CARLA_ROOT}/PythonAPI/carla
export PYTHONPATH=$PYTHONPATH:${CARLA_ROOT}/PythonAPI
```

4.3 Potential Issue

Run `sudo apt install libomp5` as per this git issue.

CHAPTER

FIVE

RUN EXAMPLE

We provide an example with a dummy agent and simple scenarios to test whether the installation is successful.

5.1 Option 1. Desktop Users

Enter the CARLA root folder and launch the CARLA server.

```
# Launch CARLA
./CarlaUE4.sh -prefernvidia -windowed -carla-port=2000

# Launch SafeBench in another terminal
python scripts/run.py --agent_cfg=dummy.yaml --scenario_cfg=example.yaml
```

where `--agent_cfg` is the path to the configuration files of agent and `--scenario_cfg` is the path to the configuration files of scenarios.

5.2 Option 2. Remote Server Users

If you use remote server that does not connect any screen, you can use the following commands without visualization window. Enter the CARLA root folder and launch the CARLA server with headless mode.

```
# Launch CARLA
./CarlaUE4.sh -prefernvidia -RenderOffScreen -carla-port=2000

# Launch SafeBench in another terminal
SDL_VIDEODRIVER="dummy" python scripts/run.py --agent_cfg=dummy.yaml --scenario_
↪cfg=example.yaml
```

5.2.1 Visualization on Remote Server

You can also visualize the pygame window using TurboVNC. First, launch CARLA with headless mode, and run our platform on a virtual display.

```
# Launch CARLA
./CarlaUE4.sh -prefernvidia -RenderOffScreen -carla-port=2000

# Run a remote VNC-Xserver. This will create a virtual display "8".
```

(continues on next page)

(continued from previous page)

```
/opt/TurboVNC/bin/vncserver :8 -noxstartup

# Launch SafeBench on the virtual display
DISPLAY=:8 python scripts/run.py --agent_cfg=dummy.yaml --scenario_cfg=example.yaml
```

You can change display 8 to any number but note that vncserver fails if there is already a virtual screen running on this display. The parameter `-noxstartup` means no virtual desktop. If you want to launch all things in a virtual desktop, please remove this parameter.

Then, you can use the TurboVNC client on your local machine to connect to the virtual display.

```
# Use the built-in SSH client of TurboVNC Viewer
/opt/TurboVNC/bin/vncviewer -via user@host localhost:n

# Or manually forward connections to the remote server
ssh -L fp:localhost:5900+n user@host
/opt/TurboVNC/bin/vncviewer localhost::fp
```

where `user@host` is your remote server, `fp` is a free TCP port on the local machine, and `n` is the display port specified when you started the VNC server on the remote server ("8" in our example).

CHAPTER

SIX

HOW TO CREATE AGENT

We provide step-by-step instructions on how to create your own (perception / planning) agent policy to be trained and evaluated in Safebench.

6.1 Create Perception Agent

TBD

6.2 Create Planning Agent

We provide a `BasePolicy` as a template for implementing policy, which is stored in file `safebench/agent/base_policy.py`. To create a new agent policy, you need to inherit `BasePolicy` and implement the following 6 functions.

1. Init policy with parameters.

```
    def __init__(self, config, logger):
```

This function processes the configuration and initialize models in polciy.

2. Train models in policy.

```
    def train(self, replay_buffer):
```

This function takes the replay_buffer as input and train the models. To get samples from the replay_buffer, you should call replay_buffer.sample() to get the dictionary of all data.

3. Set the ego vehile object and global route.

```
    def set_ego_and_routes(self, ego_vehicles, routes):
```

The object of ego vehicle is created in gym environment. We allow the users access this object in policy. The gloabl routes that the ego vehicle is asked to complete is also provided. Users can properly uses these two things to design the policy.

4. Set train or eval mode.

```
    def set_mode(self, mode):
```

To switch the mode of the models (e.g., neural networks), you should implement this function. You can leave this function blank if your models do not have a mode.

5. Get action from policy.

```
    def get_action(self, state, infos, deterministic):
```

This function takes the state as input and return the action. If the deterministic is True, the action should be deterministic. The information of state can be found in the code environment.

6. Load model from file.

```
    def load_model(self):
```

This function loads the model from file. You can leave this function blank if your models do not need to be loaded from file.

7. Save model to file.

```
    def save_model(self):
```

This function saves the model to file. You can leave this function blank if your models do not need to be saved to file.

CHAPTER

SEVEN

HOW TO CREATE SCENARIO

We provide step-by-step instructions on how to create your own (perception / planning) scenarios to be used in Safebench.

7.1 Create Perception Agent

TBD

7.2 Create Planning Agent

There are three steps to create a new scenario with static objects and dynamic actors. The first step is to define the route file, scenario type file, and scenario config file. The second step is to implement the scenario class, which create actors, define initial behavior of actors, and create static objects. The third step is to implement the scenario policy, which controls the behavior of the actors if necessary.

7.2.1 Step 1: Define Route and Scenario Config

Firt, you should define the routes of the scenario in `safebench/scenario/scenario_data/route`. Then, you should define the combination of scenarios and route by writing a json file in `safebench/scenario/config/scenario_type`. Each item in this file represent a specific scenario. Finally, a configuration file of your scenario should be written in `safebench/scenario/config`. This file contains the configuration of the scenario, e.g., the number of actors, the number of time steps, the number of static objects, etc.

7.2.2 Step 2: Implement Scenario Class

All scenarios need to inherit the `BasicScenario` defined in `safebench/scenario/scenario_definition/base_scenario.py`. One scenario should create actors in the scenario and update the behavior of the actors at each time step. To create a new scenario policy, you need to inherit `BasePolicy` and implement the following 6 functions.

1. Init policy with parameters.

```
    def __init__(self, config, logger):
```

You can use the config to initialize the scenario. The logger is used to print the information of the scenario.

2. Create initial behavior of actors.

```
def create_behavior(self, scenario_init_action):
```

Scenarios usually contains a initial action to create the initial behavior of the actors. You can use this function to create the initial behavior of the actors.

3. Update behavior of actors.

```
def update_behavior(self, scenario_action):
```

This function updates the behavior of the actors. The scenario_action is the action from the scenario policy.

4. Initialize actors in the scenario.

```
def initialize_actors(self):
```

This function creates the actors in the scenario with the behavior (e.g., positions) created in function `create_behavior`.

5. Define conditions to stop scenario.

```
def check_stop_condition(self):
```

You can define your own stop condition to stop the scenario. If this is left blank, the scenario will be stopped after some pre-defined conditions, e.g., the number of time steps and collision dection.

7.2.3 step 3 (Optional): Implement Scenario Policy

Your scenarios may contain a policy to cotnrol the behavior of the actors. If so, you need to implement a scenario policy. If not, you can just use `DummyPolicy`. We provide a `BasePolicy` as a template for implementing policy, which is stored in file `safebench/scenario/scenario_policy/base_policy.py`. To create a new scenario policy, you need to inherit `BasePolicy` and implement the following 6 functions.

1. Init policy with parameters.

```
    def __init__(self, config, logger):
```

You can use the config to initialize the scenario. The logger is used to print the information of the scenario.

2. Train models in policy.

```
    def train(self, replay_buffer):
```

This function takes the replay_buffer as input and train the models. To get samples from the replay_buffer, you should call replay_buffer.sample() to get the dictionary of all data.

3. Set train or eval mode.

```
    def set_mode(self, mode):
```

To switch the mode of the models (e.g., neural networks), you should implement this function. You can leave this function blank if your models do not have a mode.

4. Get action from policy.

```
    def get_action(self, state, infos, deterministic):
```

This function takes the state as input and return the action. If the deterministic is True, the action should be deterministic. The information of state can be found in the code environment.

5. Load model from file.

```
    def load_model(self):
```

This function loads the model from file. You can leave this function blank if your models do not need to be loaded from file.

6. Save model to file.

```
    def save_model(self):
```

This function saves the model to file. You can leave this function blank if your models do not need to be saved to file.

CHAPTER

EIGHT

HOSTING A CHALLENGE VIA EVALAI

This document provides an overview on how to host a code-upload based challenge on EvalAI. A code-upload based challenge is usually a reinforcement learning challenge in which participants upload their trained model in the form of a Docker image. The environment is also a docker image.

Info below extracted from the EvalAI Documentation.

8.1 Step 1: Set Up Main Repo

Clone this EvalAI-Starter Repo as a template. For info on how to use a repo as a template, see this.

8.2 Step 2: Set Up Challenge Configuration

Open the "challenge_config.yml" in the repo. Update the values of the features in the file based on the characteristics of the challenge. More info about the features can be found here.

Note that the following two features have to have the following values:

1) remote_evaluation: True
2) is_docker_based: True

For evaluation to be possible, an AWS Elastic Kubernetes Service (EKS) cluster might need to be created. The following info is needed:

1) aws_account_id
2) aws_access_key_id
3) aws_secret_access_key
4) aws_region

This info needs to emailed to team@cloudcv.org, who will set up the infrastructure in the AWS account.

8.3 Step 3: Define Evaluation Code

A evaluation file needs to be created to determine which metrics will be determined at which phase. This will also evalute the participants' submissions and post a score to the leaderboard. The environment image should be created by the host and the agent image should be pushed by the participants.

The overall structure of the evaluation code is fixed for architectural reasons.

To define the evaluation code:

1) Open the environment.py file located in EvalAI-Starters/code_upload_challenge_evaluation/environment/.

2) Edit the evaluator_environment class.

```
class evaluator_environment:
    def __init__(self, environment="CartPole-v0"):
        self.score = 0
        self.feedback = None
        self.env = gym.make(environment)
        self.env.reset()

    def get_action_space(self):
        return list(range(self.env.action_space.n))

    def next_score(self):
        self.score += 1
```

There are three methods:

a) __init__: initialization method b) get_action_space: returns the action space of the agent in the environment c) next_score: returns or updates the reward achieved

Additional methods can be added as need be.

3) Edit the Environment class in environment.py.

```
class Environment(evaluation_pb2_grpc.EnvironmentServicer):
    def __init__(self, challenge_pk, phase_pk, submission_pk, server):
        self.challenge_pk = challenge_pk
        self.phase_pk = phase_pk
        self.submission_pk = submission_pk
        self.server = server

    def get_action_space(self, request, context):
        message = pack_for_grpc(env.get_action_space())
        return evaluation_pb2.Package(SerializedEntity=message)

    def act_on_environment(self, request, context):
        global EVALUATION_COMPLETED
        if not env.feedback or not env.feedback[2]:
            action = unpack_for_grpc(request.SerializedEntity)
            env.next_score()
            env.feedback = env.env.step(action)
        if env.feedback[2]:
            if not LOCAL_EVALUATION:
                update_submission_result(
```

(continues on next page)

(continued from previous page)

```
                    env, self.challenge_pk, self.phase_pk, self.submission_pk
                )
            else:
                print("Final Score: {0}".format(env.score))
                print("Stopping Evaluation!")
                EVALUATION_COMPLETED = True
        return evaluation_pb2.Package(
            SerializedEntity=pack_for_grpc(
                {"feedback": env.feedback, "current_score": env.score,}
            )
        )
```

gRPC servers are used to get actions in the form of messages from the agent container. This class can be edited to fit the needs of the current challenge. Seriailzation and deserialization of the messages to be sent across gRPC is needed. The following two methods may be helpful for this:

a) unpack_for_gprc: this method deserializes entities from request/response sent over gRPC. This is useful for receiving messages (for example, actions from the agent). b) pack_for_gprc: this method serializes entities to be sent over a request over gRPC. This is useful for sending messages (for example, feedback from the environment).

4) Edit the requirements file based on the required packages for the environment.

5) Edit environment Dockerfile located in EvalAI-Starters/code_upload_challenge_evaluation/docker/environment/ if need be.

6) Fill in the docker enviroment variables in docker.env located in EvalAI-Starters/code_upload_challenge_evaluation/docker/environment/:

```
AUTH_TOKEN=x
EVALAI_API_SERVER=https://eval.ai
LOCAL_EVALUATION = True
QUEUE_NAME=x
```

7) Create a docker image on upload on Amazon Elastic Container Registry (ECR). More info on pushing a docker image to ECR can be found here.

```
docker build -f <file_path_to_Dockerfile>

aws ecr get-login-password --region <region> | docker login --username AWS --
↪password-stdin <aws_account_id>.dkr.ecr.<region>.amazonaws.com
docker tag <image_id> <aws_account_id>.dkr.ecr.<region>.amazonaws.com/<my-
↪repository>:<tag>
docker push <aws_account_id>.dkr.ecr.<region>.amazonaws.com/<my-repository>:<tag>
```

8) Add environment image to challenge configuration for challenge phase. For each challenge phase, add the link to the environment image.

```
...
challenge_phases:
    - id: 1
    ...
    - environment_image: <docker image uri>
...
```

9) Create a starter example for creating the agent: the participants are expected to create a docker image with the

policy and methods to interact with the environment. To create the agent environment:

a) Create the starter script. A template, agent.py, is provided in EvalAI-Starters/code_upload_challenge_evaluation/agent/.

```
import evaluation_pb2
import evaluation_pb2_grpc
import grpc
import os
import pickle
import time

time.sleep(30)

LOCAL_EVALUATION = os.environ.get("LOCAL_EVALUATION")

if LOCAL_EVALUATION:
    channel = grpc.insecure_channel("environment:8085")
else:
    channel = grpc.insecure_channel("localhost:8085")

stub = evaluation_pb2_grpc.EnvironmentStub(channel)

def pack_for_grpc(entity):
    return pickle.dumps(entity)

def unpack_for_grpc(entity):
    return pickle.loads(entity)

flag = None

while not flag:
    base = unpack_for_grpc(
        stub.act_on_environment(
            evaluation_pb2.Package(SerializedEntity=pack_for_grpc(1))
        ).SerializedEntity
    )
    flag = base["feedback"][2]
    print("Agent Feedback", base["feedback"])
    print("*"* 100)
```

b) Edit requirements.txt located in EvalAI-Starters/code_upload_challenge_evaluation/requirements based on package requirements. c) Edit the Dockerfile (if need be) located in EvalAI-Starters/code_upload_challenge_evaluation/docker/agent/ which will interact run agent.py to interact with the environment. d) Edit docker.env located in EvalAI-Starters/code_upload_challenge_evaluation/docker/agent/ to be:

```
LOCAL_EVALUATION = True
```

8.4 Step 4: Edit Challenge HTML Templates

Update the HTML templates in EvalAI-Starters/templates. The submission-guidelines.html should be detailed to ensure participants can upload their submissions. The participants are expected to submit links to their docker images using evalai-cli (more info here). The command is:

```
evalai push <image>:<tag> --phase <phase_name>
```

CHAPTER

NINE

HOW TO SUBMIT RESULTS

CHAPTER

TEN

MODULES AND FUNCTIONS

To help developers quickly understand the structure of the code, we briefly introduce important modules of Safebench.

10.1 Carla Runner (safebench/carla_runner.py)

This is the entry point that manages all modules by hosting a loop to run all scenarios.

10.2 Gym module (safebench/gym_carla/)

This is the gym-style interface for Carla. The implementation of environments is in the `envs` folder. Each env will contains one scenario and a vectorized wrapper is implemented in `env_wrapper.py` to manager all scenarios that simultaneously run on the same map.

10.3 Agent Module (safebench/agent/)

The implementations of autonomous vehicle agents are placed here. The configuration files corresponding to these agents are placed in the `config` folder. The saved model files corresponding to these agents are placed in the `model_ckpt` folder.

10.4 Scenario Module (safebench/scenario/)

The implementations of traffic scenarios are placed here. The folder `scenario_data` stores data and model that scenarios use, including scenario routes, scenario models, and adversarial attack templates. The folder `config` stores configurations of scenarios, where the `.yaml` files will call different types of scenarios in the `scenario_type` folder. File `scenario_data_loader.py` contains a data loader to sample scenario configurations for training and evaluation.

The folder `srunner` contains partial files of Carla Scenario Runner for parsering configuration files (`scenario_configs`), scenario implementation (`scenarios`), and managing scenarios (`scenario_manager`). New scenarios implemented by the users should be placed into folder `scenarios`.