VerifAI

Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward

Apr 17, 2024

CONTENTS

VerifAI is a software toolkit for the formal design and analysis of systems that include artificial intelligence (AI) and machine learning (ML) components. VerifAI particularly seeks to address challenges with applying formal methods to perception and ML components, including those based on neural networks, and to model and analyze system behavior in the presence of environment uncertainty. The current version of the toolkit performs intelligent simulation guided by formal models and specifications, enabling a variety of use cases including temporal-logic falsification (bug-finding), model-based systematic fuzz testing, parameter synthesis, counterexample analysis, and data set augmentation. Our CAV 2019 paper, which is the basis of the tutorial below, illustrates all of these use cases: see our *publications* page for further applications.

VerifAI was designed and implemented by Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia.

If you use VerifAI in your work, please cite our CAV 2019 paper and link to our GitHub repository.

If you have any problems using VerifAI, please submit an issue to the repository or contact Daniel Fremont at dfremont@ucsc.edu or Edward Kim at ek65@berkeley.edu.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Getting Started with VerifAI

VerifAI requires **Python 3.8** or newer. Run `python --version` to make sure you have a new enough version; if not, you can install one from the Python website or using pyenv (e.g. running `pyenv install 3.11`). If the version of Python you want to use is called something different than just `python` on your system, e.g. `python3.11`, use that name in place of `python` throughout the following instructions.

There are two ways to install VerifAI:

- from our repository, which has the very latest features but may not be stable. The repository also contains example scripts such as those used in the tutorial.
- from the Python Package Index (PyPI), which will get you the latest official release of VerifAI but will not include example code, etc.

If this is your first time using VerifAI, we suggest installing from the repository so that you can try out the examples.

Once you've decided which method you want to use, follow the instructions below, which should work on macOS and Linux (on Windows, we recommend using the Windows Subsystem for Linux).

First, activate the virtual environment in which you want to install VerifAI. To create and activate a new virtual environment called `venv`, you can run the following commands:

```
python -m venv venv
source venv/bin/activate
```

Once your virtual environment is activated, make sure your **`pip`** tool is up-to-date:

```
python -m pip install --upgrade pip
```

Now you can install VerifAI either from the repository or from PyPI:

Repository

PyPI

The following commands will clone the VerifAI repository into a folder called `VerifAI` and install VerifAI from there. It is an "editable install", so if you later update the repository with **`git pull`** or make changes to the code yourself, you won't need to reinstall VerifAI.

```
git clone https://github.com/BerkeleyLearnVerify/VerifAI
cd VerifAI
python -m pip install -e .
```

The following command will install the latest full release of Scenic from PyPI:

```
python -m pip install verifai
```

Note that this command skips experimental alpha and beta releases, preferring stable versions. If you want to get the very latest version available on PyPI (which may still be behind the repository), run:

```
python -m pip install --pre verifai
```

You can also install specific versions with a command like:

```
python -m pip install verifai==2.1.0b1
```

Some features of VerifAI require additional packages: the tool will prompt you if they are needed but not installed.

Note: In the past, the GPy package did not always install correctly through the automated process. If necessary, you can build it from source as follows:

```
git clone https://github.com/SheffieldML/GPy
find GPy -name '*.pyx' -exec cython {} \
pip install GPy/
```

1.2 Basic Usage

1.2.1 Setting up the Falsifier

Defining a Sample Space and Choosing a Sampler

There are two ways of defining a feature space.

Method 1: Using Feature APIs in VerifAI

```
from verifai.features import *
from verifai.samplers import *

control_params = Struct({
'x_init': Box([-0.05, 0.05]),
'cruising_speed': Box([10.0, 20.0]),
'reaction_time': Box([0.7, 1.00])
})

env_params = Struct({
        'broken_car_color': Box([0.5, 1], [0.25, 0.75], [0, 0.5]),
        'broken_car_rotation': Box([5.70, 6.28])
    })

cones_config = Struct({
        'traffic_cones_pos_noise': Box([-0.25, 0.25], [-0.25, 0.25], [-0.25, 0.25]),
        'traffic_cones_down_0': Categorical(*np.arange(5)),
        'traffic_cones_down_1': Categorical(*np.arange(5)),
```

(continues on next page)

(continued from previous page)

```
        'traffic_cones_down_2': Categorical(*np.arange(5))
    })

sample_space = FeatureSpace({
        'control_params': Feature(control_params),
        'env_params': Feature(env_params),
        'cones_config': Feature(cones_config)
})

# Examples of Instantiating Some of VerifAI's Supported Samplers
random_sampler                = FeatureSampler.randomSamplerFor(sample_space)
halton_sampler                = FeatureSampler.haltonSamplerFor(sample_space)
cross_entropy_sampler         = FeatureSampler.crossEntropySamplerFor(sample_space)
simulated_annealing_sampler   = FeatureSampler.simulatedAnnealingSamplerFor(sample_space)
```

Method 2: Using Scenic

```
from verifai.samplers import ScenicSampler

path = 'examples/webots/controllers/scenic_cones_supervisor/lane_cones.scenic'
scenic_sampler = ScenicSampler.fromScenario(path)
```

Scenic's sampler, by default, does random sampling (see *ScenicSampler* for the available configuration options). However, it is possible to invoke VerifAI's other samplers from within the scenario using Scenic's external parameters.

Constructing a Monitor

Active samplers sample for the next point in the feature space by accounting for the history of the performance of a system being tested in previous simulations. To use active samplers, a user need to provide a monitor (i.e. objective function). For passive samplers, monitor is optional, but it can be used to populate the error table with the how a system of interest performed in each simulation.

```
from verifai.monitor import specification_monitor

# The specification must assume specification_monitor class
class confidence_spec(specification_monitor):
    def __init__(self):
        def specification(traj):
            return traj['yTrue'] == traj['yPred']
        super().__init__(specification)
```

Writing a Formal Specification with Metric Temporal Logic

Instead of a customized monitor, users can provide a specification using metric temporal logic. In such case, users need to use mtl_falsifier instead of generic_falsifier.

```
from verifai.falsifier import mtl_falsifier

specification = ["G(collisioncone0 & collisioncone1 & collisioncone2)"]
```

Defining Falsifier Parameters

```
from dotmap import DotMap
falsifier_params = DotMap(
        n_iters=1000,   # Number of simulations to run (or None for no limit)
        max_time=None,  # Time limit in seconds, if any
        save_error_table=True,   # Record samples that violated the monitor/specification
        save_good_samples=False,  # Don't record samples that satisfied the monitor/
↪specification
        fal_thres=0.5,    # Monitor return value below which a sample is considered a␣
↪violation
        sampler_params=None   # optional DotMap of sampler-specific parameters
)
```

Setting up Client/Server Communication

VerifAI uses a client/server model to communicate with an external simulator for running tests. The default *Server* (suitable for use with user-provided clients for new simulators) uses network sockets and can be customized as follows:

```
PORT = 8888
BUFSIZE = 4096
MAXREQS = 5

server_options = DotMap(port=PORT, bufsize=BUFSIZE, maxreqs=MAXREQS)
```

When performing falsification with dynamic Scenic scenarios, VerifAI communicates with the simulator through Scenic, and a special *ScenicServer* is required: see below for an example.

Instantiating a Falsifier

Setting up a falsifier is a simple matter of combining the pieces above. For a custom monitor, we can use `generic_falsifier`:

```
from verifai.falsifier import generic_falsifier
falsifier = generic_falsifier(
        sampler=random_sampler,         # or scenic_sampler, etc. as above
        monitor=confidence_spec(),
        falsifier_params=falsifier_params,
        server_options=server_options
)
```

For a specification in Metric Temporal Logic, we can use `mtl_falsifier`:

```
from verifai.falsifier import mtl_falsifier
falsifier = mtl_falsifier(
        sampler=random_sampler,
        specification=specification,
        falsifier_params=falsifier_params,
        server_options=server_options
)
```

After instantiating either kind of falsifier, it can be run as follows:

```
# Wait for a client to connect, run the simulations, then clean up
falsifier.run_falsifier()
```

Dynamic Scenic scenarios can be used with any type of falsifier, but you must specify the *ScenicServer* class (see its documentation for available options). Monitors will be passed the Scenic `Simulation` object resulting from each simulation:

```
from verifai.scenic_server import ScenicServer

scenic_sampler = ScenicSampler.fromScenicCode("""\
model scenic.simulators.newtonian.model
ego = Object with velocity (0, Range(5, 15))
other = Object at (5, 0), with velocity (-10, 10)
terminate after 2 seconds
record final (distance to other) as dist
""")

class scenic_spec(specification_monitor):
        def __init__(self):
                def specification(simulation):
                        return simulation.result.records['dist'] > 1
                super().__init__(specification)

falsifier = generic_falsifier(
        sampler=scenic_sampler,
        monitor=scenic_spec(),
        falsifier_params=DotMap(n_iters=2),
        server_class=ScenicServer
)
falsifier.run_falsifier()
```

1.3 Tutorial / Case Studies

This page describes how to run many of the examples included in the VerifAI repository, which illustrate the main use cases of VerifAI. After cloning the repository, you can install the extra dependencies needed for the examples by running:

```
python -m pip install -e ".[examples]"
```

1.3.1 Emergency Braking with a simple Newtonian simulator

Scenic comes with a simple built-in Newtonian physics simulator, which supports running traffic scenarios. In this example scenario we have a car (in red) whose task is to stay within its lane using a PID controller, while maintaining a safe distance of 5 meters to objects in front.

Task: Falsify the PID lane keeping controller

Sample space: distance from ego to obstacle

Relevant files:

1. `examples/scenic/falsify_distance.py` : Defines type of falsifier (sampler and number of iterations)
2. `examples/scenic/newtonian/carlaChallenge2.scenic` : Scenic program defining the ego vehicle's behavior (i.e. policy) and its environment

Running the falsifier: To run this example go to `examples/scenic`. Then run `python falsify_distance.py`.

The falsifier runs for 5 iterations by default; you can change this by modifying `n_iters` in `falsify_distance.py`.

Expected Output: During the running of the falsifier you should see a top-down view of the simulations taking place. When falsification has completed, the script will print out tables listing all of the samples that were generated and the associated satisfaction value of the specification, rho. Rho represents the quantitative satisfaction of the specification such that the sample satisfies the specification if rho is positive and violates the specification if rho is negative. The samples are separated into two tables, the **error_table** for counterexamples to the specification and the **safe_table** for all other samples.

Learning More: See the README in the `examples/scenic` folder for more options the `falsify_distance.py` script supports, including running similar experiments in the CARLA driving simulator.

1.3.2 Lane keeping with inbuilt simulator

VerifAI comes with an inbuilt simulator developed from this car simulator. In this example we have a car (in red) whose task is to stay within its lane using an LQR controller.

Task: Falsify the LQR lane keeping controller

Sample space: Initial x-position, angle of rotation, and cruising speed.

Relevant files:

1. `src/verifai/simulators/car_simulator/examples/lanekeeping_LQR/lanekeeping_falsifier.py` : Defines the sample space and type of falsifier (sampler and number of iterations)
2. `src/verifai/simulators/car_simulator/examples/lanekeeping_LQR/lanekeeping_simulation.py` : Defines the controller and the simulation environment

Running the falsifier: To run this example open two terminal shells and go to `src/verifai/simulators/car_simulator` in each of them. Then in first one run `python examples/lanekeeping_LQR/lanekeeping_falsifier.py` and wait till you see "Server ready" in the terminal; then run `python examples/lanekeeping_LQR/lanekeeping_simulation.py` in the other one.

The falsifier runs for 20 iterations, you can change this by modifying `MAX_ITERS` in `examples/lanekeeping_LQR/lanekeeping_falsifier.py`. At the end of the runs, you should see "End of all simulations" in the terminal where you ran `python examples/lanekeeping_LQR/lanekeeping_simulation.py`.

Expected Output: During the running of the falsifier you should the samples and the associated value of the specification satisfaction (rho). Rho represents the quantitative satisfaction of the specification such that the sample satisfies the specification if the rho is positive or falsifies the specification if the rho is negative.

You should see two tables in the first terminal where you ran `python examples/lanekeeping_LQR/lanekeeping_falsifier.py`, labeled **Falsified Samples** a collection of all falsified samples and **Safe Samples** a collection of all the samples which were safe.

1.3.3 Data augmentation

In this example we try to falsify a Neural Network (NN) trained to detect images of cars on roads. We re-create the data augmentation example from this paper. We implemented our own picture renderer which generates images by sampling from a low dimensional modification (sample) space.

Task: Falsify the NN trained on the synthetic images generated by the picture rendered

Sample space: Image background (37 backgrounds), number of cars- (x, y) position and type of car, overall image brightness, color, contrast, and sharpness.

Relevant files:

1. `examples/data_augmentation/falsifier.py` : Defines the sample space and type of falsifier (sampler and number of iterations)
2. `examples/data_augmentation/classifier.py` : Interface to the picture renderer and instantiate the NN

Running the falsifier: Open two terminal shells and go to `examples/data_augmentation` in each of them. Then in first one run `python falsifier.py` and wait till you see "Server ready" in the terminal; then run `python classifier.py` in other one.

The falsifier runs for 20 iterations, you can change this by modifying `MAX_ITERS` in `examples/data_augmenatation/falsifier.py`. At the end of the runs, you should see "End of all classifier calls" in the terminal where you ran `python classifier.py`.

The falsifying samples are stored in the data structure **error_table**. We can further analyse the error_table to generate images for retraining the NN. We have introduced three techniques to generate new images for the NN re-training:

1. Randomly sample samples from the error_table
2. Top k closest (similar) sampler from the error_table
3. Use the PCA analysis on the samples to generate new samples

Expected Output: During the running of the falsifier you should the samples and the associated value of the specification satisfaction (rho). Here rho represents the qualitative (boolean) satisfaction. Its **True** if the NN correctly classifies the image.

You should see a table labeled **Error Table** a collection of all falsified samples in the first terminal where you ran `python falsifier.py`. This follows two counterexample sets, one **randomly** generated from the error table and the other with **k closest samples**. We set k = 4 in this example. This is followed by the PCA analysis results, we report the pivot and the 2 principle components.

The images in the two counterexample sets will pop up at the end of the run. The images are saved in the `counterexample_images` folder. The images with the prefix "random_" are from the random samples counterexample set and those with the prefix "kclosest_" are from the k closest counterexample set.

1.3.4 Webots examples

To run these examples you need to download and install Webots 2018 from here. Webots 2018 is not free software, however you can get a 30-day free trial. While choosing license please select PRO license. When you open Webots for the first time go to `Webots->Preferences` and change `Startup Mode` to Pause.

(It should be possible to adapt these examples to work under the newer open-source versions of Webots as well; mainly, the `.wbt` files need to be regenerated using the Webots OSM Importer.)

We do not currently support using Webots 2019 (even though it is free software), since we found its performance to be very poor on machines without GPUs (e.g. personal laptops).

Scene Generation using Scenic

In this example we use the probabilistic programming language Scenic to generate scenes where a road is obstructed by a broken car behind traffic cones. The scene we would like to generate is made up of an ego car (in red) and a broken car (in silver) parked behind three traffic cones.

Task: Generate scenes using Scenic

Sample space: Position of the ego car in the city, position and orientation of the cones, position and orientation of the broken car

Relevant files:

1. `examples/webots/controllers/scenic_cones_supervisor/lane_cones.scenic` : Scenic code to generate scenes
2. `examples/webots/controllers/scenic_cones_supervisor/scenic_cones_sampler.py` : Interface to scenic
3. `examples/webots/controllers/scenic_cones_supervisor/scenic_cones_supervisor.py` : Interface to webots
4. `examples/webots/worlds/shattuck_build.wbt` : Webots world of downtown Berkeley

Running the sampler: Launch Webots and load the world `examples/webots/worlds/shattuck_build.wbt` (`File->Open World`). Open a terminal and go to `examples/webots/controllers/scenic_cones_supervisor` and run `python scenic_cones_sampler.py`. Once that starts running (you will notice a message "Initialized Sampler" in the terminal), you can start the simulation.

The sampler runs for 20 iterations, you can change this by modifying `MAX_ITERS` in `examples/webots/controllers/scenic_cones_supervisor/scenic_cones_sampler.py`. At the end of the runs, you should see "End of scene generation" in the Webots console and the Webots window closes.

Expected Output: During the running of the sampler you should the samples and the associated value of the specification satisfaction (rho). Since in this application we focus only on scene generation, rho does not have any practical relevance (and has been set to infinity).

You should see the collection of all samples generated by the scenic script in the terminal where you ran `python scenic_cones_sampler.py`.

Falsification of accident scene with cones

In this example we choose a scene generated from the above case study and run our falsifier to find small variations in the initial scene variation which lead to the ego car (in red) to crash into the traffic cones. For this purpose, we fix the location of the broken car, cones, and the ego_car but introduce variations in the color of the broken, and noise in the position and orientation of the ego car, cones and the broken car.

The ego car' controller is responsible for safely maneuvering around the cones. To achieve this, we implemented a hybrid controller. Initially, the car tries to remain in its lane. The controller in the ego car relies on a NN we trained to detect the cones and estimate the distance to the cones. When the distance to the cone is less than 15 meters the car starts a lane changing maneuver. This can be seen as the NN warning a human in the car to change lanes. To capture the human behavior, we introduce a reaction time (which is also a parameter) of the human.

Task: Falsify the NN detecting cones and the hybrid controller

Sample space: Initial position and orientation noise of the ego car, cruising speed of the ego car, position and orientation noise in the broken car, position error and orientation of the cones, color of the broken car, reaction time of the human

Relevant files:

1. `examples/webots/controllers/cones_lanechange_supervisor/cones_lanechange_falsifier.py` : Defines the sample space and type of falsifier (sampler and number of iterations)
2. `examples/webots/controllers/cones_lanechange_supervisor/cones_lanechange_supervisor.py` : Interface to webots
3. `examples/webots/worlds/shattuck_buildings_falsif.wbt` : Webots world of downtown Berkeley

Running the falsifier: During the running of the falsifier you should the samples and the associated value of the specification satisfaction (rho). Rho is the quantitative satisfaction.

Launch Webots and load the world `examples/webots/worlds/shattuck_buildings_falsif.wbt` (`File->Open World`). Open a terminal and go to `examples/webots/controllers/cones_lanechange_supervisor` and run `python cones_lanechange_falsifier.py`. Once that starts running (you will notice a message "Initialized Sampler" in the terminal), you can start the simulation.

The falsifier runs for 20 iterations, you can change this by modifying `MAX_ITERS` in `examples/webots/controllers/cones_lanechange_supervisor/cones_lanechange_falsifier.py`. At the end of the runs, you should see "End of simulations" in the Webots console and the Webots window closes.

Each simulation run takes about 16 seconds in the simulation time; but may take more time in real time because of overheads like the neural networks calls and rendering.

Expected Output: You should see a table **Unsafe Samples** the collection of all samples generated that caused the ego car to crash into the cones in the terminal where you ran `python cones_lanechange_falsifier.py`.

Fuzz testing using Scenic

In this example we use Scenic to initial conditions to recreate collision scenarios at an intersection. In this example, the ego car (in green) if going straight through an intersection. The front view of the ego car is obstructed by a set of cars (in silver) which are stopped at the intersection. A human car (in red) attempts to the take a left turn at the intersection. In this scenario, the camera of both the ego car and the human car are obstructed by the silver cars standing at the intersection. We use scenic to sample the initial positions and orientations of the ego car, human car and couple of the standing cars at the intersection.

To reduce the chances of collision, we designed a controller for the ego car which utilizes the information coming from a "smart intersection". The smart intersection sends a warning to the ego car, when the human car approaches the intersection. Based on how early the warning comes in, we reduce the number of collision scenarios.

Task: Generate accident scenarios and test controllers with "smart intersection"

Sample space: Position and orientation of the ego car, human car and standing cars

Relevant files:

1. `examples/webots/controllers/scenic_intersection_supervisor/intersection_crash.scenic` : Scenic code to generate scenes
2. `examples/webots/controllers/scenic_intersection_supervisor/scenic_intersection_sampler.py` : Interface to scenic
3. `examples/webots/controllers/scenic_intersection_supervisor/scenic_intersection_supervisor.py` : Interface to webots
4. `examples/webots/worlds/scenic_intersection.wbt` : Webots world of the intersection

To run this example, you need to install pyproject, `$pip install pyproj`

Running the sampler: Launch Webots and load the world `examples/webots/worlds/scenic_intersection.wbt` (`File->Open World`) . Open a terminal and go to `examples/webots/controllers/scenic_intersection_supervisor` and run `python scenic_intersection_sampler.py`. Once that starts running (you will notice a message "Initialized Sampler" in the terminal), you can start the simulation.

To test the original scenario without the smart intersection, ensure that `ignore_smart_intersection = True` on line 24 in the file `examples/webots/controllers/smart_brake/smart_brake.py`. To test with smart intersection set `ignore_smart_intersection = False`. You can update how early the warning is sent by updating the value of `intersection_buffer` in line 111 in `scenic_intersection_supervisor.py`. If set to 0, then the warning is given only when the human car enters the intersection. If > 0, the warning is sent when the car is some distance away from the intersection. If < 0, the warning is sent when the car is some distance inside the intersection.

The sampler runs for 20 iterations, you can change this by modifying `MAX_ITERS` in `examples/webots/controllers/scenic_intersection_supervisor/scenic_intersection_sampler.py`. At the end of the runs, you should see "End of accident scenario generation" in the Webots console and the Webots window closes.

Each simulation run takes about 16 seconds in the simulation time; but may take more time in real time because of overheads like rendering and communication between controllers.

Expected Output: During the running of the sampler you should the samples and the associated value of the specification satisfaction (rho). Since in this application we focus only on scenario generation, rho does not have any practical relevance (and has been set to infinity).

You should see the collection of all samples generated by the scenic script in the terminal where you ran `python scenic_intersection_sampler.py`.

1.3.5 OpenAI Gym examples

The following two examples require the `baselines` package, which does not support TensorFlow 2 (needed for other examples above), so they cannot be easily run at the moment. We leave them here since they were discussed in the VerifAI paper and provide examples of how to use VerifAI with reinforcement learning algorithms. We would welcome a pull request updating them to use an RL library which is actively maintained!

These examples require at least version 0.1.6 of the `baselines` package. As of May 2020, the version on PyPI is too old and will not work, so you need to install `baselines` from its repository. Follow the installation instructions given there, remembering to first activate your virtual environment if necessary.

Cartpole

In this example we want to test the robustness of a controllers to changes in model parameters and initial states of the cartpole from openAI gym.

We use OpenAI baselines to train a NN to control the cartpole . We train a NN using Proximal Policy Optimization algorithms (PPO) with 100000 training episodes.

We use the reward function as the specification for testing; i.e., if the reward is positive for all the environments the controller is safe. For the cartpole, the specification is: the maximum variation of the pole from the center should be less than 12 degree and maximum variation of the initial position should be less than 2.4m. To test the controller we loosen the training thresholds for angle, by 0.01 radians, and the x variation, by 0.1m, to get the testing thresholds. To capture this we define a specification using metric temporal logic python library which **VerifAI** can convert into a monitor internally.

Task: Test the robustness of the NN controller trained using PPO

Sample space: Initial state x position and rotation of the cartpole, model parameters - mass and length of pole and mass of the cart

Relevant files:

1. `examples/openai_gym/cartpole/cartpole_falsifier.py` : Defines the sample space and type of falsifier (sampler and number of iterations)
2. `examples/openai_gym/cartpole/cartpole_simulation.py` : Interface to OpenAI gym and baselines

Running the falsifier: During the running of the falsifier you should the samples and the associated value of the specification satisfaction (rho). Rho is the quantitative satisfaction.

Open two terminal shells and go to `cd examples/openai_gym/cartpole` in each of them. Then in first one run `python cartpole_falsifier.py` and wait till you see "Initialized sampler" in the terminal; then run `python cartpole_simulation.py` in other one.

The falsifier runs for 20 iterations, you can change this by modifying `MAX_ITERS` in `examples/openai_gym/cartpole/cartpole_falsifier.py`. At the end of the runs, you should see "End of all cartpole simulations" in the terminal where you ran `python cartpole_simulation.py`.

Expected Output: During the running of the falsifier you should the samples and the associated value of the specification satisfaction (rho). Rho is the quantitative satisfaction.

You should see the error table **Counter-example samples** containing all falsified samples in the terminal where you ran `python cartpole_falsifier.py`

Mountaincar

In this example we show the effect of hyper-parameters to train NN to control the mountaincar environment from openAI gym.

Specifically, we see the effects of different training algorithms and size of the training set on synthesizing a NN controller.

We treat the reward function provided by the environment as our specification. For the mountaincar, this is characterized by the distance to the flag. Here, unlike the previous examples, we would like to find the set of parameters which which ensures the NN is able to correctly control the mountaincar. To do this, we negate the specification so that falsifying the specification implies finding a safe controller.

Task: Study the effect of hyperparameters in training NN controllers for mountaincar

Sample space: Training algorithms - Deep RL algorithms, number of training episodes

Relevant files:

1. `examples/openai_gym/mountaincar/mountaincar_falsifier.py` : Defines the sample space and type of falsifier (sampler and number of iterations)
2. `examples/openai_gym/mountaincar/mountaincar_simulation.py` : Interface to OpenAI gym and baselines

Running the falsifier: During the running of the falsifier you should the samples and the associated value of the specification satisfaction (rho). Rho is the quantitative satisfaction.

Open two terminal shells and go to `cd examples/openai_gym/mountaincar` in each of them. Then in the first one run `python mountaincar_falsifier.py` and wait till you see "Initialized sampler" in the terminal; then run `python mountaincar_simulation.py` in the other one.

The falsifier runs for 20 iterations, you can change this by modifying `MAX_ITERS` in `examples/openai_gym/mountaincar/mountaincar_falsifier.py`. At the end of the runs, you should see "End of all mountaincar simulations" in the terminal where you ran `python mountaincar_simulation.py`.

Expected Output: You should see a table **Hyper-parameters leading to good controllers** containing all sampled hyperparamaters which build NN which can safely control the mountaincar in the terminal where you ran `python mountaincar_falsifier.py`.

1.4 Feature APIs in VerifAI

class Feature(*domain*, *distribution=None*, *lengthDomain=None*, *lengthDistribution=None*, *distanceMetric=None*)

A feature or list of features with unknown length.

Parameters

- **domain** – a *Domain* object specifying the Feature's possible values;
- **distribution** (*optional*) – object specifying the distribution of values;
- **lengthDomain** (*Domain*) – if not None, this *Feature* is actually a list of features, with possible lengths given by this *Domain*;
- **lengthDistribution** (*optional*) – distribution over lengths;
- **distanceMetric** (*optional*) – if not None, custom distance metric.

```
# Feature consisting of list of 10 cars
carDomain = Struct({
    'position': Array(Real(), [3]),
    'heading': Box((0, math.pi))
})
Feature(Array(carDomain, [10]))

# Feature consisting of list of 1-10 cars
Feature(carDomain, lengthDomain=DiscreteBox((1, 10)))
```

class FeatureSpace(*features*, *distanceMetric=None*)

A space consisting of named features.

```
FeatureSpace({
    'weather': Feature(DiscreteBox([0, 12])),
    'egoCar': Feature(carDomain),
```

(continues on next page)

(continued from previous page)

```
    'traffic': Feature(Array(carDomain, [4]))
})
```

flatten(*point*, *fixedDimension=False*)

Flatten a point in this space. See Domain.flatten.

If fixedDimension is True, the point is flattened out as if all feature lists had their maximum lengths, with None as a placeholder. This means that all points in the space will flatten to the same length.

meaningOfFlatCoordinate(*index*, *pointName='point'*)

Meaning of a coordinate of a flattened point in this space.

See the corresponding function of Domain. Works only for points flattened with fixedDimension=True, since otherwise a given index can have different meaning depending on the lengths of feature lists.

pandasIndexForFlatCoordinate(*index*)

Pandas index of a coordinate of a flattened point in this space.

See meaningOfFlatCoordinate, and Domain.pandasIndexForFlatCoordinate.

coordinateIsNumerical(*index*)

Whether the value of a coordinate is intrinsically numerical.

See meaningOfFlatCoordinate, and Domain.coordinateIsNumerical.

unflatten(*coords*, *fixedDimension=False*)

Unflatten a tuple of coordinates to a point in this space.

class Domain

Abstract class of domains

uniformPoint()

Sample a uniformly random point in this Domain

flatten(*point*)

Flatten a point in this Domain to a tuple of coordinates.

Useful for analyses that do not understand the internal structure of Domains. This representation of a point is also hashable, and so can be put into sets and dicts.

flattenOnto(*point*, *targetList*)

Flatten a point onto the end of the given list.

meaningOfFlatCoordinate(*index*, *pointName='point'*)

Meaning of a coordinate of a flattened point in this Domain.

If pointName is the name of a variable storing a point in the Domain, then this function returns an expression extracting from that variable the value which would be stored in the given coordinate index when the point is flattened. For example:

```
>>> struct = Struct({'a': Real(), 'b': Real()})
>>> point = struct.makePoint(a=4, b=3)
>>> struct.flatten(point)
(4.0, 3.0)
>>> struct.meaningOfFlatCoordinate(1)
'point.b'
>>> eval(struct.meaningOfFlatCoordinate(1))
3
```

pandasIndexForFlatCoordinate(*index*)

Like meaningOfFlatCoordinate, but giving a MultiIndex for pandas.

coordinateIsNumerical(*index*)

Whether the value of a coordinate is intrinsically numerical.

In particular, whether it makes sense to use the Euclidean distance between different values of the coordinate. This would not be the case for Domains whose points are strings, for example, even if those are converted to numbers for the purpose of flattening.

numericizeCoordinate(*coord*)

Make a coordinate numeric. For internal use.

denumericizeCoordinate(*coord*)

Reconstitute a coordinate's original value. For internal use.

unflatten(*coords*)

Unflatten a tuple of coordinates to a point in this Domain.

unflattenIterator(*coords*)

Unflatten an iterator of coordinates to a point in this Domain.

standardize(*point*)

Map the point into a hyperbox, preserving measure.

If the Domain is continuous, this should map into a unit hyperbox. If it is discrete, it should map into a discrete hyperbox. Which (if either) of these is the case can be determined by calling standardizedDimension and standardizedIntervals: for primitive Domains, at least one will return the 'not supported' value.

standardizeOnto(*point*, *targetList*)

Standardize a point onto the end of the given list.

unstandardize(*coords*)

Unstandardize a tuple of coordinates to a point in this Domain.

unstandardizeIterator(*coords*)

Unstandardize an iterator of coords to a point in this Domain.

partition(*predicate*)

Split this Domain into parts satisfying/falsifying the predicate.

rejoinPoints(**components*)

Join points from the partitioned components of a Domain.

class Constant(*value*)

Domain consisting of a single value

class Categorical(**values*)

Domain consisting of a finite set of values

class Real

Domain of real numbers

class Integer

Domain of integers

class Box(**intervals*)

A hyper-box over the reals.

Points in a Box are tuples of floats.

class DiscreteBox(**intervals*)

A hyper-box over the integers.

Points in a DiscreteBox are tuples of ints.

class Array(*domain*, *shape*)

A multidimensional array of elements in a common domain.

For example, Array(Box((-1, 1)), (10, 5)) represents a 10x5 grid of real numbers, each in the interval [-1, 1].

Points in an Array are nested tuples of elements. For example, a point in the Array above would be a tuple of 10 elements, each of which is a tuple of 5 elements, each of which is a point in the underlying Box domain.

pointWithElements(*it*)

Build a point in this domain from an iterable of elements.

This is similar to numpy.reshape, building a multidimensional array from a flat list of elements. For example:

```
>>> elts = [1, 2, 3, 4, 5, 6]
>>> array = Array(Real(), (2, 3))
>>> array.pointWithElements(elts)
((1, 2, 3), (4, 5, 6))
>>> array = Array(Real(), (3, 2))
>>> array.pointWithElements(elts)
((1, 2), (3, 4), (5, 6))
```

elementsOfPoint(*point*)

Return an iterator over the elements of a point in this domain.

This is similar to numpy.flatten, turning a multidimensional array into a flat list of elements. For example:

```
>>> array = Array(Real(), (3, 2))
>>> list(array.elementsOfPoint(((1, 2), (3, 4), (5, 6))))
[1, 2, 3, 4, 5, 6]
```

class ScalarArray(*domain*, *shape*)

An array whose elements are integers or reals.

This is a specialized implementation of Array optimized for large arrays of scalars like images.

class Struct(*domains*)

A domain consisting of named sub-domains.

The order of the sub-domains is arbitrary: two Structs are considered equal if they have the same named sub-domains, regardless of order. As the order is an implementation detail, accessing the values of sub-domains in points sampled from a Struct should be done by name:

```
>>> struct = Struct({'a': Box((0, 1)), 'b': Box((2, 3))})
>>> point = struct.uniformPoint()
>>> point.b
(2.20215292046797,)
```

Within a given version of VerifAI, the sub-domain order is consistent, so that the order of columns in error tables is also consistent.

1.5 Search Techniques

VerifAI provides several techniques for exploring the semantic search space for verification, testing, and synthesis. These are largely based on sampling and optimization methods. In the tool, we refer to all of these as "samplers".

There are three active samplers (i.e. cross entropy, simulated annealing, and bayesian optimization samplers) and two passive samplers (i.e. random and halton samplers) supported. The details of their implementation can be found in verifai/samplers directory.

1.5.1 How to add a new sampler?

First, add your python script of your sampler in verifai/samplers directory along with other sampler scripts. Second, add an API to call your sampler in verifai/samplers/feature_sampler.py

1.5.2 Sampling from a Scenic program

Defining the semantic feature space using a Scenic program (instead of the *Feature APIs in VerifAI*) requires the use of a special sampler, `ScenicSampler`.

class ScenicSampler(*scenario*, *maxIterations=None*, *ignoredProperties=None*)

Samples from the induced distribution of a Scenic scenario.

Created using the *fromScenario* and *fromScenicCode* class methods.

See Scene Generation in the Scenic documentation for details of how Scenic's sampler works. Note that VerifAI's other samplers can be used from within a Scenic scenario by defining external parameters.

classmethod fromScenario(*path*, *maxIterations=None*, *ignoredProperties=None*, ***kwargs*)

Create a sampler corresponding to a Scenic program.

The only required argument is `path`, and `maxIterations` may be useful if your scenario requires a very large number of rejection sampling iterations. See `scenic.scenarioFromFile` for details on optional keyword arguments used to customize compilation of the Scenic file.

Parameters

- **path** (*str*) – path to a Scenic file.
- **maxIterations** (*int*) – maximum number of rejection sampling iterations (default 2000).
- **ignoredProperties** – properties of Scenic objects to not include in generated samples (see `defaultIgnoredProperties` for the default).
- **kwargs** – additional keyword arguments passed to `scenic.scenarioFromFile`; e.g. `params` to override global parameters or `model` to set the world model.

classmethod fromScenicCode(*code*, *maxIterations=None*, *ignoredProperties=None*, ***kwargs*)

As above, but given a Scenic program as a string.

pointForScene(*scene*)

Convert a sampled Scenic `Scene` to a point in our feature space.

The *FeatureSpace* used by this sampler consists of 2 features:

- `objects`, which is a *Struct* consisting of attributes `object0`, `object1`, etc. with the properties of the corresponding objects in the Scenic program. The names of these attributes may change in a future version of VerifAI: use the *nameForObject* function to generate them.

- `params`, which is a *`Struct`* storing the values of the global parameters of the Scenic program (use *`paramDictForSample`* to extract them).

static nameForObject(*i*)

Name used in the *`FeatureSpace`* for the Scenic object with index i.

That is, if `scene` is a `Scene`, the object `scene.objects[i]`.

paramDictForSample(*sample*)

Recover the dict of global parameters from a *`ScenicSampler`* sample.

1.6 Servers and Clients

1.6.1 Generic

class Server(*sampling_data*, *monitor*, *options={}*)

Generic server for communicating with an external simulator.

class Client(*port*, *bufsize*)

Generic client for running simulations based on samples from the server.

Users must implement the abstract method *`simulate`* to run a simulation.

abstract simulate(*sample*)

Run a simulation from the given sample.

Returns

The outcome of the simulation (e.g. trajectories of objects), to be passed to the monitor.

1.6.2 Dynamic Scenic

class ScenicServer(*sampling_data*, *monitor*, *options={}*)

`Server` for use with dynamic Scenic scenarios.

Supported server options:

- `maxSteps`: maximum number of time steps to run a simulation;
- `verbosity`: verbosity level (as in the Scenic `--verbosity` option);
- `maxIterations`: maximum number of iterations for rejection sampling;
- `simulator`: Scenic `Simulator` to use, or `None` (the default) to use one specified in the scenario.

1.7 Interfacing VerifAI with Dynamic Scenic

1.7.1 Setting up Client/Server Communication

The syntax for setting up a VerifAI falsifier with a dynamic Scenic script is very similar to the setup outlined in *Basic Usage*. The major difference is that the maximum number of timesteps to run each simulation must be provided as an argument to the falsifier:

```
falsifier_params = DotMap(
    n_iters=None,
    save_error_table=True,
    save_safe_table=True,
    max_time=60,
)
server_options = DotMap(maxSteps=300, verbosity=0) # maximum number of timesteps to run␣
↪each simulation.
```

For an example of using dynamic Scenic with VerifAI, see the `examples/multi_objective` folder.

1.8 Running Falsification in Parallel

VerifAI now supports running falsification in parallel, with worker processes simultaneously running dynamic simulations of samples. This API uses the package RAY from UC Berkeley's RiSE lab, which provides encapsulation for process-level parallelism in Python.

To enable parallel falsification, run `pip install ray` or use the `parallel` extra when installing VerifAI (i.e. `pip install ".[parallel]"` from the repository, or `pip install "verifai[parallel]"` from PyPI).

1.8.1 Setting up the Falsifier

This is as simple as changing any line instantiating a `generic_falsifier` to `generic_parallel_falsifier`. An additional parameter accepted by the `generic_parallel_falsifier` class is `num_workers` which determines the number of parallel worker processes that run simulations. By default there are 5 parallel workers.

For an example of using parallelized falsification, see the `examples/multi_objective` folder.

1.9 Multi-Objective Falsification

1.9.1 Specification of Objectives

VerifAI now provides the ability to run falsification on multiple metrics at the same time. For example, consider the following VerifAI monitor:

```
from verifai.monitor import multi_objective_monitor

"""
Example of multi-objective specification. This monitor specifies that the ego vehicle
must stay at least 5 meters away from each other vehicle in the scenario.
"""
class distance_multi(multi_objective_monitor):
    def __init__(self, num_objectives=1):
        priority_graph = nx.DiGraph()
        self.num_objectives = num_objectives
        priority_graph.add_edge(0, 2)
        priority_graph.add_edge(1, 3)
        priority_graph.add_edge(2, 4)
        priority_graph.add_edge(3, 4)
```

(continues on next page)

(continued from previous page)

```
        print(f'Initialized priority graph with {self.num_objectives} objectives')
        def specification(simulation):
            positions = np.array(simulation.result.trajectory)
            distances = positions[:, [0], :] - positions[:, 1:, :]
            distances = np.linalg.norm(distances, axis=2)
            rho = np.min(distances, axis=0) - 5
            return rho

        super().__init__(specification, priority_graph)
```

The monitor computes the distance between the ego vehicle and every other vehicle in the scenario and returns all of these distances. Note that the monitor class extends the `multi_objective_monitor` class, written specifically for vector-valued objectives. Additionally, a *rulebook* is defined in the `priority_graph` variable, which is a partial ordering over the metrics providing some pairwise information about which metrics are considered most important. This rulebook is encoded as a directed acyclic graph (DAG) using the NetworkX library.

1.9.2 Samplers Supporting Multiple Objectives

To mitigate issues with sensitivity to results of initial samples, VerifAI implements the *multi-armed bandit sampler*, an active sampler which uses the Upper Confidence Bound (UCB) algorithm to tradeoff exploration of new regions of the feature space as well as exploitation of previously found counterexamples. To use the multi-armed bandit sampler, either use the `MultiArmedBanditSampler` class or, if using Scenic, add the line

```
param verifaiSamplerType = 'mab'
```

For an example of using multi-objective sampling, see the `examples/multi_objective` folder.

1.10 Publications Using VerifAI

1.10.1 Main Papers

The main paper on VerifAI is:

> Dreossi*, Fremont*, Ghosh*, Kim, Ravanbakhsh, Vazquez-Chanlatte, and Seshia, *VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems*, CAV 2019.

The Scenic environment modeling language is described in another paper (see the Scenic documentation for the most recent bibliography):

> Fremont, Dreossi, Ghosh, Yue, Sangiovanni-Vincentelli, and Seshia, *Scenic: A Language for Scenario Specification and Scene Generation*, PLDI 2019. [full version]

* Equal contribution.

1.10.2 Case Studies

We have also used VerifAI in several industrial case studies:

> Fremont, Chiu, Margineantu, Osipychev, and Seshia, *Formal Analysis and Redesign of a Neural Network-Based Aircraft Taxiing System with VerifAI*, CAV 2020 (to appear). [arXiv preprint]
>
> Fremont, Kim, Pant, Seshia, Acharya, Bruso, Wells, Lemke, Lu, and Mehta, *Formal Scenario-Based Testing of Autonomous Vehicles: From Simulation to the Real World*, ITSC 2020 (to appear). [arXiv preprint]

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

THREE

LICENSE

VerifAI is distributed under the 3-Clause BSD License.

PYTHON MODULE INDEX

INDEX