Repeatabliity Package for "Case Study: Verifying the Safety of an Autonomous Racing Car with a Neural Network Controller"

Submitted to HSCC 2020

Radoslav Ivanov, Taylor J. Carpenter, James Weimer,
Rajeev Alur, George J. Pappas, Insup Lee
{rivanov, carptj, weimerj, alur, pappasg, lee}@seas.upenn.edu
University of Pennsylvania

1 Introduction

This document describes the repeatability package for the HSCC 2020 submission titled "Case Study: Verifying the Safety of an Autonomous Racing Car with a Neural Network Controller". All of the code used in the paper can be found at https://github.com/rivapp/autonomous_car_verification. All the data collected during experiments can be found at https://github.com/rivapp/hscc20_data_traces.

The repeatability package consists of seven parts:

1. a simulator for the F1/10 autonomous racing car [1],
2. a modified version of the Verisig tool [4], together with all plant models used in the evaluation (encoded in a format accepted by Verisig),
3. a modified version of the Flow* tool [2],
4. training code implementing the reinforcement learning algorithm deep deterministic policy gradient (DDPG) [5],
5. training code implementing the reinforcement learning algorithm twin delayed deep deterministic policy gradient (TD3) [3],
6. all trained neural network (NN) controllers used in the evaluation,
7. data traces collected during experiments.

The following sections provide a brief description of each item, followed by the installation instructions.

2 Simulator for the F1/10 autonomous racing car

The F1/10 car simulator is implemented in the *Car.py* file in the *simulator* folder. It implements the bicycle model described in the paper (along with an option to include the slip angle model, which was not included due to issues with existing verification tools). Note that there are two sets of plant states: 1) global states and 2) local states with respect to each hallway. The local states are included as they make it easier to compute the LiDAR rays. Thus, when the car enters a new hallway, the local states are reset, effectively introducing a hybrid switch in the dynamics model as well. The LiDAR model implements the three-region hybrid approach described in the paper.

The simulator is consistent with the standard OpenAI gym training environment [6] so that it can be easily used in training scripts that work on OpenAI Gym. In particular, there is a `reset` function as well as a number of instance variables exposed by the OpenAI Gym environments. The simulator allows the creation of arbitrary hallway environments with 90-degree turns. We assume that each hallway is longer than 5m in order to simplify the LiDAR modeling. Users can also vary the LiDAR field of view, as well as the number of rays.

The script *sim_model.py* illustrates how to simulate the F1/10 car with a given neural network controller. We expose the functions `plot_trajectory` and `plot_lidar` in order to plot the car's entire trajectory or plot the current LiDAR scan. Finally, the script *plot_trajectories.py* is used to generate the multiple trajectories shown in Figures 3 and 5 in the paper.

3 Modified Verisig

As noted in the paper, we used Verisig for the verification evaluation. However, we needed to modify the tool since the released version of Verisig accepts plant models in the SpaceEx format, whereas the LiDAR model used in the verification was too complex to be encoded in SpaceEx. In particular, we developed a prototype Python version of Verisig that takes three inputs: 1) a hybrid system description of the plant model stored in a Python dictionary (in a pickle file); 2) a description of the (glue) transitions between the plant and the neural network controller (in a pickle file); 3) a description of the neural network stored in a Python dictionary and exported in a YAML format. Note that both the plant model and the transitions between the plant and the NN have to be given as strings accepted by the Flow* parser. Verisig does not check whether these strings are well-formatted so if there are errors, they will be reported by the Flow* parser.

The initial condition verified in the paper (i.e., the car starts in a 20cm range in the middle of the hallway) is split into multiple subsets in order to keep the approximation error small. That is why, the Verisig script released in this repository is effectively a multi-runner, i.e., it will split up the initial condition into subsets and spawn multiple processes to verify each one. The command to run Verisig has the following form:

```
$ python2 verisig_multi_runner.py <NN.yml> <plant_model.pickle> <glue.pickle>
```

To reproduce the verification results shown in the paper, one needs to choose the plant model (i.e., how many LiDAR rays are used by the controller) and the corresponding NN controller. For example, to verify the 21-ray setup with TD3 64×64, controller 1, one needs to run (from the *verisig* folder):

```
$ python2 verisig_multi_runner.py ../dnns/TD3_L21_64x64_C1.yml ../plant_models/
   ↪ dynamics_21.pickle ../plant_models/glue_21.pickle
```

Note that subset size is currently hardcoded to be 0.5cm. However, some setups require reducing the subset size to be verified. For those setups, please change the hardcoded variable *posOffset*.

3.1 Plant models

As mentioned above, the plant models are given as Python dictionaries exported in the pickle files. The reason we cannot encode these models in SpaceEx is that the hybrid observation model contains tan^{-1} that does not work in Flow* out of the box. To get around this issue, we modified Flow* to add functionality to handle discrete tan^{-1} resets. This functionality is triggered through mode names in the plant model. Thus, the plant models cannot be currently encoded in the standard SpaceEx format. We will improve this functionality in a future release. Yet, we do provide the Python scripts that generate the pickle files (*writeDynamics.py* and *writeCompTransitions.py* in the *plant_models* folder). Users can change the parameters in these scripts if they would like to export and verify a different model.

4 Modified Flow*

Note that we have added some modifications to Flow* in order to add the tan^{-1} functionality and to optimize the neural network verification part. Specifically, we avoid the sigmoid integration described in the original paper [4] but rather obtain Taylor Models analytically.

5 Training using DDPG

The code to train a controller using the DDPG algorithm is provided in the *train_ddpg* folder [5]. Note that we have added certain modifications to the vanilla algorithm, such as a convolutional NN for the critic and cosine annealing for the learning rate. To train the controller, one needs to run the *racing_ddpg.py* script in the *train_ddpg* folder. The script periodically stores a NN controller in a .h5 (i.e., Keras) file in the same folder. To convert the Keras file to a .yml file that can be used by Verisig, please use the *h5_to_yaml.py* script provided in that folder.

6 Training using TD3

The code to train a controller using the TD3 algorithm is provided in the *train_td3* folder [3]. Note that we use the authors' implementation without any modifications, except to adapt it to work with the F1/10 simulator. To train the controller, one needs to run the *racing.py* script in the *train_td3* folder. The script periodically stores a NN controller in a .h5 (i.e., Keras) file in the *models* subfolder. To convert the Keras file to a .yml file that can be used by Verisig, please use the *h5_to_yaml.py* script provided in that folder.

7 Trained NN controllers

All trained NN controllers used in the paper are stored in the *dnns* folder, both in .h5 and .yml format. The .h5 versions can be used to simulate the system, while the .yml versions are used in the verification.

8 Data traces from experiments

We collected all LiDAR traces from our experiments. Due to their size, the traces are stored in a separate git repository. The traces are grouped according to whether the experiments were run in the modified (covered) or unmodified (uncovered) environment. Each subfolder is named after the setup it represents, namely the name contains the training algorithm, the number of LiDAR rays used, the neural network architecture and the index of the controller trained for that setup. Each line in a file corresponds to one LiDAR scan; each line is comma-separated in the following format: $tv\_sec, tv\_usec, y_1, \ldots, y_{1081}, crash$, where tv_sec and tv_usec are seconds and microseconds, respectively, since Jan. 1, 1970 (note that the board time is not updated, so relative times should be considered only), followed by the 1081 LiDAR rays for this scan. Note that LiDAR measurements are sampled at roughly 40Hz, but the controller is executed at roughly 10Hz (although both of these vary slightly from run to run). Finally, the boolean variable *crash* indicates whether a crash has already occurred in the current run.

9 Installation

This section describes the steps required to install all the tools used in this package. The installation procedure primarily consists of installing required libraries and compiling software packages.

9.1 Flow*

The modified version of Flow* is contained in the *flowstar* folder. Installing Flow* consists of installing the required libraries and compiling Flow*.

Flow* requires a MAKE environment and libraries that are only readily available on Linux. As such, Linux is currently the only supported operating system for Flow*. We have verified compatibility with Ubuntu 16.04, but other distributions should work as well. Flow* requires the following packages: **gmplib**, **mpfr**, **gnu gsl**, **bison**, **flex**, **gnuplot**, **glpk**, and **yaml-cpp**. The following procedure can be used to install the program:

1. Install required libraries :
   - `$ apt install libgmp3-dev`
   - `$ apt install libmpfr-dev libmpfr-doc libmpfr4 libmpfr4-dbg`
   - `$ apt install gsl-bin libgsl0-dev`
   - `$ apt install bison`
   - `$ apt install flex`
   - `$ apt install gnuplot-x11`
   - `$ apt install libglpk-dev`
   - `$ apt install libyaml-cpp-dev`
2. From the *flowstar* folder, run the make command: `$ cd flowstar; make`

9.2 Verisig

The main Verisig executable is a Python application, so it can be run on any computer that supports Python. The source code is included in the *verisig* folder. Verisig has been tested on a Ubuntu 16.04 machine running Python 2.7.

9.3 Tensorflow and Keras

The DDPG algorithm currently uses the Tensorflow and Keras libraries. Thus, if one wishes to run the training script, these libraries would need to be installed. There are a number of way to install these libraries, depending on the user's needs, as detailed here: https://www.tensorflow.org/install/pip. We found the easiest way to get started is to use a virtual environment (so as to not affect one's OS Python build). To install and activate a virtual environment (in the home directory), one needs to run:

```
  $ sudo pip2 install -U virtualenv # system-wide install
  $ virtualenv --system-site-packages -p python2 ./venv
  $ source ./venv/bin/activate # sh, bash, ksh, or zsh
```

Once the virtual environment is activated, install Tensorflow and Keras using pip. Our implementation has been tested with Python 2.7 and with versions 1.15.0-rc2 and 2.3.1 for Tensorflow and Keras, respectively (note that Tensorflow 2 does not work with our implementation):

```
  $ pip2 install tensorflow==1.15rc2
  $ pip2 install keras==2.3.1
```

9.4 Pytorch

The TD3 implementation provided by the authors uses the Pytorch library. Thus, if one wishes to run the training script, this library would need to be installed. Our implementation has been tested with Python 3.5 and Pytorch version 1.3.0. To install Pytorch using pip, one needs to run (in a virtual environment):

```
$ pip3 install https://download.pytorch.org/whl/cpu/torch-1.0.1.post2-cp35-cp35m-
    ↪ linux_x86_64.whl
$ pip3 install torchvision
```

9.5 Other Python Libraries

The scripts *plot_trajectories.py*, *sim_model.py* as well as the training scripts require installing a the *gym* and *matplotlib* libraries in Python:

```
$ pip2 install gym
$ pip2 install matplotlib
```

10 Instructions for Reproducing all Figures and Tables in the Paper

For clarity, this section lists the specific scripts that need to be executed to reproduce each computational item in the submission.

10.1 Table 1

To reproduce Table 1, one needs to first run the *verisig_multi_runner.py* script. Since reproducing Table 1 requires a lot of computation and memory, we reproduce it row by row. To reproduce a given row, run the *verisig_multi_runner.py* script with the corresponding controller as input. The output of the script is printed on the console, so we suggest storing it in a text file. For example, to reproduce row 7 (which requires the least amount of computation), one would need to run (from the *verisig* folder):

```
$ python2 verisig_multi_runner.py ../dnns/TD3_L21_64x64_C1.yml ../plant_models/
    ↪ dynamics_21.pickle ../plant_models/glue_21.pickle > results.txt
```

Warning: the above command will spawn multiple processes (either 40 or 100, depending on the initial interval size). Once all instances have terminated (which might take multiple days on a single-core machine), one can parse the output using the *parser.py* script (from the *verisig* folder):

```
$ python2 parser.py results.txt
```

This command will print the descriptive statistics (last 3 columns of Table I) to the console.

To reproduce all other rows, one might need to change the hardcoded *posOffset* variable in *verisig_multi_runner.py* to 0.002 (0.2cm) – the row labeled "Initial interval size" in Table 1 indicates which controllers require that change. Finally, the NN that takes 41 rays as input also works with different plant models, so the command to run that would be:

```
$ python2 verisig_multi_runner.py ../dnns/TD3_L41_64x64_C1.yml ../plant_models/
   ↪ dynamics_41.pickle ../plant_models/glue_41.pickle > results.txt
```

10.2 Table 2

Table 2 can be reproduced from the data traces repository. Calling the *get_safety_outcomes.py* script will process all data traces and will print to the console the safety outcomes for each controller in each environment. Note that five data traces were corrupted, so the output of this script is slightly different from Table 2 for the following controllers: (DDPG, 128x128, Controller2, Env_M), (DDPG, 128x128, Controller2, Env_U), (DDPG, 128x128, Controller3, Env_U), (DDPG, 64x64, Controller1, Env_U), (DDPG, 64x64, Controller3, Env_U). There is also a typo in Table 2 in the paper: Column 6 in the second to last row should read "5/10".

10.3 Figure 3

Figure 3 can be reproduced with the *plot_trajectories.py* script from the *simulator* folder. There are two hardcoded variables that need to be set: *lidar_noise* needs to be set to 0.1 and *missing_lidar_rays* needs to be set to 0. After that, one needs to run *plot_trajectories.py* four times, once per figure:

```
$ python2 plot_trajectories.py ../dnns/DDPG_L21_64x64_C1.h5
$ python2 plot_trajectories.py ../dnns/DDPG_L21_128x128_C2.h5
$ python2 plot_trajectories.py ../dnns/TD3_L21_64x64_C1.h5
$ python2 plot_trajectories.py ../dnns/TD3_L21_128x128_C1.h5
```

Each command will plot a Python figure that can be manually saved as a .png file. Note that these commands require Tensorflow, Keras, matplotlib and gym (the .h5 is a Keras file extension).

10.4 Figure 4

Figure 4 can be reproduced from the data traces repository. Note that the figures are produced with Matlab scripts – we used Matlab R2018a, though other versions should work as well. To reproduce Figures 4a and 4b, execute the *plot_car_lidar_hscc20_mod.m* and *plot_car_lidar_hscc20_unmod.m* scripts, respectively. Note that the styling will be slightly different.

10.5 Figure 5

Figure 5 can be reproduced with the *plot_trajectories.py* script from the *simulator* folder. There are two hardcoded variables that need to be set: *lidar_noise* needs to be set to 0.2 and *missing_lidar_rays* needs to be set to 5. After that, one needs to run *plot_trajectories.py* four times, once per figure:

```
$ python2 plot_trajectories.py ../dnns/DDPG_L21_64x64_C1.h5
$ python2 plot_trajectories.py ../dnns/DDPG_L21_128x128_C2.h5
$ python2 plot_trajectories.py ../dnns/TD3_L21_64x64_C1.h5
$ python2 plot_trajectories.py ../dnns/TD3_L21_128x128_C1.h5
```

Each command will plot a Python figure that can be manually saved as a .png file. Note that these commands require Tensorflow, Keras, matplotlib and gym (the .h5 is a Keras file extension).

References

[1] F1/10 Autonomous Racing Competition. http://f1tenth.org.

[2] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *International Conference on Computer Aided Verification*, pages 258–263. Springer, 2013.

[3] S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[4] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee. Verisig: verifying safety properties of hybrid systems with neural network controllers. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 169–178. ACM, 2019.

[5] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[6] OpenAI. Openai gym. https://gym.openai.com.