Artifact Evaluation for Paper 591

An Inductive Synthesis Framework for Verifiable Machine Learning

1. Structure of the Project

```
VRL/
    ├── benchmarks/
    ├── ddpg_chkp/
    ├── DDPG.py
    ├── pympc/
    ├── vcsos.py
    ├── z3/
    ├── z3verify.py
    ├── Environment.py
    ├── shield.py
    ├── main.py
    ├── metrics.py
    ├── plotting.py
    ├── run_***.sh
```

`benchmarks/` : All the benchmarks we presented in the paper.

`ddpg_chkp/` : Our pre-trained neural networks and shields.

`DDPG.py` : DDPG is a reinforcement learning implementation for training neural network controllers in control systems.

`pympc/` : Toolset for generating inductive invariants for discrete-time systems.

`vcsos.py` : Toolset for generating Julia scripts which can infer inductive invariants for continuous time systems.

`z3/` : z3py is a python interface for Z3 solver.

`z3verify.py` : Toolset for encoding continuous/discrete time systems for SMT based verification.

`Environment.py` : A reinforcement learning environment, encapsulating basic operations such as `step`, `observe`, etc.

`shield.py` : Shield (deterministic policy program) script, containing source code of basic shield operations such as `train_shield`, `call_shield`, `test_shield`, etc.

`main.py` : The main loop of learning, testing, and verification of deterministic policy programs.

`metrics.py` : Functions for measuring properties of neural policies and deterministic policy programs.

`plotting.py` : Functions for plotting.

`run*.sh` : Run benchmarks in batch. See *Run All Benchmarks* for details.

2. Getting Started Guide:

One can follow the steps in this section to get our artifact to run. We provide two ways: (1) downloading the docker image (recommended) and (2) building from the source code (full documentation is provided below).

2.1. Docker Image(Recommanded)

The experiment results collected in the docker file may be different than what we reported in the paper because (1) it is a docker environment with relatively limited computational resources (2) the tool in the docker file may show different behaviors, i.e., the random generator used by the TensorFlow library may be different, which may lead to a different search direction for both synthesized programs and inferred inductive invariants. However, the artifact suffices to prove the reproducibility of our approach.

1. Install Docker:

https://docs.docker.com/get-started/

1. Pull Verifiable Reinforcement Learning Environment docker image:

```
docker pull caffett/vrl_env
```

1. Start the docker environment:

```
docker run -it caffett/vrl_env:v0 /bin/bash
```

1. Clone VRL code:

```
git clone git@github.com:caffett/VRL_CodeReview.git
```

2.2. Building from the Source Code

2.2.1. Install

Require Python2.7 and usual python packages such as numpy, scipy, matplotlib, ...

2.2.1.1. pip install:

1. pip install tensorflow
2. pip install tflearn
3. pip install numba

2.2.1.2. Install Z3:

1. From https://github.com/Z3Prover/z3/releases, download Z3 binary.
2. `export PYTHONPATH=PATH_UNPACK_Z3/z3/bin/:` .

2.2.1.3. Install Julia-0.6

1. Download Julia-0.6 from https://julialang.org/downloads/oldreleases.html.
2. `export PATH="$PATH:PATH_UNPACK_JULIA/julia/bin"`

2.2.1.4. Install SumOfSqure tool:

Download and license the Mosek solver

- Download the MAC OS 64bit x86 MOSEK Optimization Suite distribution from https://mosek.com/downloads/ and unpack it into a chosen directory.
- For MAC, run the command:

  > `python <MSKHOME>/mosek/8/tools/platform/osx64x86/bin/install.py`
  >
  > where `<MSKHOME>` is the directory where MOSEK was installed. This will set up the appropriate shared objects required when using MOSEK.

- Add the path `<MSKHOME>/mosek/8/tools/platform/osx64x86/bin` to the OS variable PATH.
- License the tool.

2.2.1.5. Configure Julia

1. Enter Julia command line.
2. Enter the following commands:

```
julia> Pkg.status()

julia> Pkg.add("SumOfSquares")

julia> Pkg.status("SumOfSquares") # SumOfSquares 0.2.0

julia> Pkg.status("JuMP") # JuMP 0.18.2

julia> Pkg.add("MathOptInterface")

julia> Pkg.add("DynamicPolynomials")

julia> Pkg.add("SCS")

julia> Pkg.add("MathOptInterfaceMosek")

julia> Pkg.add("Clp")
```

3. Step-by-Step Instructions

This section introduces how to reproduce the results in our paper. In Sec 3.1, we show the commands to run our benchmarks. Then, in Sec 3.2, we show how to run the experiments in the tables of our paper and obtain the data in the tables.

3.1. Run Command

Pretrained neural networks are provided in our artifact. We also provide an interface to retrain neural networks. However, note that retraining a neural network typically requires significant manual efforts to adjust a number of training parameters and is often time-consuming. Our neural network training time is estimated by training with 1000 epochs each of which has 100 steps. In practice, it usually takes a longer time than what we reported in the paper. Our pre-trained neural models are stored at `ddpg_chkp/<model_name>/<network_structure>`.

Our tool provides a python script for each of our benchmarks. Given a benchmark, the user just needs to type:

```
python benchmarks/<benchmark_name>

[--nn_test | --shield_test | --retrain_shield | --retrain_nn \

| --test_episodes=TEST_EPISODES]
```

There are 5 flags:

--nn_test: adding this flag runs the pre-trained neural network controller alone without shield protection.

--shield_test: adding this flag runs the pre-trained neural network controller in tandem with a pre-synthesized and verified program distilled from the network to provide shield protection.

--retrain_shield: adding this flag re-synthesizes a deterministic program to be used for shielding.

--retrain_nn: adding this flag retrains a neural network controller.

--test_episodes: This parameter specifies the number of steps used for each simulation run. The default value is 100.

3.2. Getting Results

3.2.1. Run a Single Benchmark

After running a benchmark, our tool reports the total simulation time and the number of times that the system enters an unsafe region.

For example, running `python benchmarks/4-car-platoon.py --nn_test --test_episodes=1000` may produce the following result:

```
Success: 992, Fail: 8
```

```
test run time: 3679.60405302 s
```

The system is indeed unsafe since a number of safety violations were observed.

Running a neural network controller in tandem with a verified program distilled from it can eliminate those unsafe neural actions. Our tool produces in its output the number of interventions from the program on the neural network controller. It also gives the total running time including the additional cost of shielding.

Running `python benchmarks/4-car-platoon.py --shield_test --test_episodes=1000` may produce the following result (using a pre-synthesized and verified program):

```
shield times: 10, shield ratio: 2e-06
test_shield run time: 3924.33369899 s
```

Based on a neural network's simulation time without and with running a shield, we can calculate the overhead of using the shield.

$$Overhead = \frac{(ShieldTestRunningTime - NeuralNetworkTestRunningTime)}{NeuralNetworkTestRunningTime} \times 100\%$$

For each benchmark, with the protection of a shield, our system never enters an unsafe region. We may get the following result for all our benchmarks.

```
Success: 1000, Fail: 0
```

Running with --retrain_shield can re-synthesize a new deterministic program to replace the original one. After re-synthesis, our tool produces total synthesis time. For example, we may get the following result by running `python 4-car-platoon.py --retrain_shield --test_episodes=1000`.

```
train_shield run time: 86.1307971478 s
```

We count how many iterations our verification algorithm needs to synthesize a deterministic program and this result corresponds to the size of a synthesized program (i.e., the number of branches in the synthesized

program).

```
Verification algorithm finds the controller True
```

3.2.2. Run All Benchmarks

We also provide some scripts to run all of our benchmarks in a batch mode:

`./run_test_[100|1000]ep.sh` : Run all the benchmarks with pre-trained neural networks. The number of epochs used in each simulation run is set to 100.

`./run_test_[100|1000]ep_table_[1|2].sh` : Run all the benchmarks in table `[1|2]` with pre-trained neural networks. The number of epochs used in each simulation run is set to `[100|1000]` .

`./run_retrain_shield.sh` : Retrain policy program for all the benchmarks.

`./run_retrain_neural_network.sh` : Retrain neural programs for all the benchmarks in table 1.

4. Build A Benchmark From Scratch

In this section, we show how to build a new benchmark step by step. Section 4.1 gives an introduction to the reinforcement learning environment we used and how to use such an environment to define a new control system. In section 4.2, we explain how to train neural network controllers using DDPG, a state-of-the-art reinforcement learning algorithm. Finally, in section 4.3, we elaborate on how to synthesize and use shields.

The following program is an example to use our system to train and shield a learning-based inverted pendulum control system. The program is divided into three pieces by `# Sec 4.x` where `x=1, 2, and 3` . We give a detailed explanation for each piece below.

```
#Sec 4.1 Reinforcement Learning Environment
  m = 1.
  l = 1.
  g = 10.

  A = np.matrix([
    [ 0., 1.],
    [g/l, 0.]
    ])
  B = np.matrix([
    [          0.],
    [1./(m*l**2.)]
    ])
```

```
  s_min = np.array([[-0.35],[-0.35]])
  s_max = np.array([[ 0.35],[ 0.35]])

  Q = np.matrix([[1., 0.],[0., 1.]])
  R = np.matrix([[.005]])

  #safety constraint
  x_min = np.array([[-0.5],[-0.5]])
  x_max = np.array([[ 0.5],[ 0.5]])
  u_min = np.array([[-15.]])
  u_max = np.array([[ 15.]])

  env = Environment(A, B, u_min, u_max, s_min, s_max, x_min, x_max, Q, R, continuous=
True)

# Sec 4.2 Train and Test Neural Networks
  if retrain_nn:
    args = { 'actor_lr': 0.0001,
             'critic_lr': 0.001,
             'actor_structure': actor_structure,
             'critic_structure': critic_structure,
             'buffer_size': 1000000,
             'gamma': 0.99,
             'max_episode_len': 500,
             'max_episodes': 1000,
             'minibatch_size': 64,
             'random_seed': 6553,
             'tau': 0.005,
             'model_path': train_dir+"retrained_model.chkp",
             'enable_test': nn_test,
             'test_episodes': test_episodes,
             'test_episodes_len': 1000}
  else:
    args = { 'actor_lr': 0.0001,
             'critic_lr': 0.001,
             'actor_structure': actor_structure,
             'critic_structure': critic_structure,
             'buffer_size': 1000000,
             'gamma': 0.99,
             'max_episode_len': 500,
             'max_episodes': learning_eposides,
             'minibatch_size': 64,
             'random_seed': 6553,
             'tau': 0.005,
```

```
              'model_path': train_dir+"model.chkp",
              'enable_test': nn_test,
              'test_episodes': test_episodes,
              'test_episodes_len': 1000}

  actor = DDPG(env, args)

# Sec 4.3 Train and Test Shields

  model_path = os.path.split(args['model_path'])[0]+'/'
  linear_func_model_name = 'K.model'
  model_path = model_path+linear_func_model_name+'.npy'

  shield = Shield(env, actor, model_path, force_learning=retrain_shield, debug=retrai
n_shield)
  shield.train_shield(learning_method, number_of_rollouts, simulation_steps, eq_err=0
, explore_mag = 1.0, step_size = 1.0)
  if shield_test:
    shield.test_shield(test_episodes, 1000, mode="single")

  actor.sess.close()
```

4.1. Reinforcement Learning Environment

We provide two kinds of reinforcement learning environments, one for linear systems and the other for polynomial systems.

4.1.1. Environment for Linear Systems

One can create a reinforcement learning environment for linear systems by calling

```
Environment(self, A, B, u_min, u_max, s_min, s_max, x_min, x_max, Q, R, continuous)
```

A linear system's state transition is based on a function like this:

$$f(x_t, u_t) = A \cdot x_t + B \cdot u_t$$

A, B are matrixes, x_t is the state at step `t`, and u_t is the control action to be taken at step `t`.

When `continuous` is `True`, this environment is continuous and the state at step `t+1` is determined by

$$x_{t+1} = x_t + timestep \cdot f(x_t, u_t)$$

$timestep$ means how much time is taken at each step. Here `f` serves for a **standard** linear first order

differential equation.

Parameters `u_min` and `u_max` specify the lower and upper bounds of a control action respectively. `s_min` and `s_max` specify the range of an initial state. Any initial state must be between `s_min` and `s_max`. The safety range of a system state is defined within `x_min` and `x_max`. States outside the safety range are deemed unsafe. `Q` and `R` specify the coefficients of our reward function, the sum of which over all states in a trajectory we want the learning algorithm to maximize (we prefer small states and actions):

$$reward(x_t, u_t) = -|Q \cdot x_t^2| - |R \cdot u_t^2|$$

As an example, the inverted pendulum example above defines a linear continuous system:

```
#Sec 4.1 Reinforcement Learning Environment
  m = 1.
  l = 1.
  g = 10.

  #Dynamics that are continuous
  A = np.matrix([
    [ 0., 1.],
    [g/l, 0.]
    ])
  B = np.matrix([
    [          0.],
    [1./(m*l**2.)]
    ])

  #intial state space
  s_min = np.array([[-0.35],[-0.35]])
  s_max = np.array([[ 0.35],[ 0.35]])

  #reward function
  Q = np.matrix([[1., 0.],[0., 1.]])
  R = np.matrix([[.005]])

  #safety constraint
  x_min = np.array([[-0.5],[-0.5]])
  x_max = np.array([[ 0.5],[ 0.5]])
  u_min = np.array([[-15.]])
  u_max = np.array([[ 15.]])

  env = Environment(A, B, u_min, u_max, s_min, s_max, x_min, x_max, Q, R, continuous=
True)
```

When `continuous` is `False`, this environment is discrete and the state at step `t+1` is determined by

$$x_{t+1} = f(x_t, u_t)$$

Here `f` serves for a **standard** linear first order *difference* equation.

4.1.2. Environment for Polynomial Systems

One can similarly create a reinforcement learning environment for polynomial systems by calling

```
PolySysEnvironment(f, f_to_str,rewardf, testf, unsafe_string, ds, us, Q, R, s_min, s_
max, u_max, u_min)
```

`f` is similar to the state transition function defined in the linear system environment above. The main difference is that `f` can be defined as a nonlinear polynomial function, e.g.,

$$x_{t+1} = x_t + timestep \cdot f(x_t, u_t)$$

where $f(x_t, u_t) = u_t \cdot x_t^2$. `f_to_str` is a function that translates the state transition function `f` to Julia code. `rewardf` is the reward function similar to the one defined above. `testf` is a function that returns `True` is an input initial state leads to a safe trajectory, and returns `False` if the input initial state leads to an unsafe trajectory. `unsafe_string` is a function that encodes the unsafe properties of the system to Julia code. `ds` and `us` are the dimension size of the state space and the dimension size of the action space of the system respectively. The rest of the parameters are equivalent to that defined in the above linear system environment.

Benchmark **biology** is one such model based on PolySysEnvironment. One can refer to biology.py#L74 as an example of how to create a polynomial system environment.

4.2. Train and Test Neural Networks

One can train neural network controllers for our cyber-physical system benchmarks using Deep Deterministic Policy Gradient (DDPG). One can adjust the hyperparameters of the machine learning algorithm by changing the values defined in the `args` dictionary (see below). For example, if one wants to set the structure of an actor neural network to 240x200, simply sets the `actor_structure` to `[240, 200]`.

`model_path` specifies the path to store a trained neural network.

To test if a neural network controller is well-trained, one can set the `enable_test` parameter in `args` to `True`. After finishing the training, the testing phase is automatically started. We elide the discussion of the other training paramters.

Calling function `DDPG(env, args)` starts the training and testing process for a neural network control policy, and the function returns an actor (a neural network program) as a controller.

For example, the above inverted pendulum example uses the following piece of code to train a neural network policy to control the system. `retrain_nn` specifies if one would like to train a neural network from scratch or reuse an existing one stored on the corresponding `model_path`.

```
# Sec 4.2 Train and Test Neural Networks
  if retrain_nn:
    args = { 'actor_lr': 0.0001,
             'critic_lr': 0.001,
             'actor_structure': actor_structure,
             'critic_structure': critic_structure,
             'buffer_size': 1000000,
             'gamma': 0.99,
             'max_episode_len': 500,
             'max_episodes': 1000,
             'minibatch_size': 64,
             'random_seed': 6553,
             'tau': 0.005,
             'model_path': train_dir+"retrained_model.chkp",
             'enable_test': nn_test,
             'test_episodes': test_episodes,
             'test_episodes_len': 1000}
  else:
    args = { 'actor_lr': 0.0001,
             'critic_lr': 0.001,
             'actor_structure': actor_structure,
             'critic_structure': critic_structure,
             'buffer_size': 1000000,
             'gamma': 0.99,
             'max_episode_len': 500,
             'max_episodes': learning_eposides,
             'minibatch_size': 64,
             'random_seed': 6553,
             'tau': 0.005,
             'model_path': train_dir+"model.chkp",
             'enable_test': nn_test,
             'test_episodes': test_episodes,
             'test_episodes_len': 1000}

  actor = DDPG(env, args)
```

To use a two hidden layer (1200×900) neural network as an executable controller for this system, set the

`actor_structure` to `[1200,900]` and only enable test when **--test_nn** is set. See pendulum.py#L44-L76 as a complete example.

4.3. Train and Test Shields (Deterministic Policy Programs)

By calling the following code, one can obtain a shield distilled from a neural network controller.

```
shield = Shield(env, actor, model_path)

shield.train_shield(learning_method, number_of_rollouts, simulation_steps)
```

`env` is the reinforcement environment we defined above.

`actor` is the neural network controller program learned by `DDPG(env, args)`.

`model_path` specifies the path to store a synthesized shield (deterministic programmatic policy).

Calling `shield.train_shield` synthesizes a deterministic program as a shield using the approach we presented in the paper.

To test a neural policy with a shield, one can call

```
shield.test_shield(test_episodes, test_steps, mode="single")
```

where `test_episodes` specifies the number of simulations and `test_step` specifies the number of steps in each simulation run. Please ignore the parameter `mode` for now.

For example, the above inverted pendulum example uses the following piece of code to synthesize and evaluate a shield distilled from the neural network controller as specified in `actor`.

```
  model_path = os.path.split(args['model_path'])[0]+'/'
  linear_func_model_name = 'K.model'
  model_path = model_path+linear_func_model_name+'.npy'

  shield = Shield(env, actor, model_path, force_learning=retrain_shield, debug=retrai
n_shield)
  shield.train_shield(learning_method, number_of_rollouts, simulation_steps, eq_err=0
, explore_mag = 1.0, step_size = 1.0)
  if shield_test:
    shield.test_shield(test_episodes, 1000, mode="single")

  actor.sess.close()
```

One can refer to pendulum.py#L79-L86 as a complete example.

In conclusion, one can follow the steps in Sec 4.1 to create a new linear or polynomial continuous or discrete time system where the controller is a neural network. Our approach then uses the code in Sec 4.2 to train the neural network and the code in Sec 4.3 to synthesize a shield as a deterministic program distilled from the neural network. The shield can then be used at runtime to retain safety guarantee for the neural network controller.