PA2

Edit New page

ThanhVu (Vu) Nguyen edited this page 3 weeks ago · 14 revisions

CS695/SWE699 (Fall'03)

PA2: Scalability and Abstraction

Goal

In this assignment, you have 2 parts. Part 2.1 you will show that symbolic execution approach you did in PA1 *does not* scale, i.e., it does not work when you have a large DNNs. Part 2.2 you will implement and show that thhe **interval** abstraction approach you studied in class scale much better.

Just like with PA1, you will use Python (3.x is preferred) and the Z3 SMT solver for part 2.1 of this assignment. **Do not** use external libraries or any extra tools (other than `z3-solver`). If you are stuck, you can post your question as a discussion on Piazza.

Part 1: Scalability Issue of Symbolic Execution

Here you will show that symbolic execution does not scale on large neurons.

1. You will create a function `create_random` that takes as input a list of positive integers representing the sizes of the neurons `[#inputs, #ofnumbers, #ofneurons, ..., #outputs]`, and create a neural network of that size with *random* weights and bias (e.g., between say -5 and 5).
2. You will run symbolic execution on this network as you did in PA1, get its results, and time the solving process.
3. You will do this for various sizes of neurons until you feel that your computer cannot handle it anymore (e.g., until it takes about a minute or two).
4. You will **save these networks** because you will reuse them to measure and compare execution time using abstraction in Part 2.
5. Finally, you will plot the results: the `x-axis` will show the size of the neurons (for simplicity, the product of all numbers in the input list) and the `y-axis` shows the time.

Feel free to be creative in this process (e.g., play around with different sizes/layers), and discuss your findings in the `README` file. Below gives some code example

```
# Create a random network from a given b
l = [2,3,4,3,3]    # 2 inputs, 3 hidden laye
dnn = create_random_nn(l)

symbolic_state = my_symbolic_execution(dnn)

# time the execution
import time
st = time.time()
_ = z3.solve(symbolic_state)
print('time to solve: ', time.time() - st)
```

Part 2: Abstract Interval Domain

Here you will implement the interval abstraction for DNN. The technique is based on Chapter 8 of the Intro to DNNV book (https://arxiv.org/pdf/2109.10317.pdf).

You can reuse the following DNN with 2 inputs, 2 hidden layers, and 2 outputs.

1. **DNN Encoding**: We can encode this DNN and additional details using Python. Note this is exactly as PA1.

   ```
   # (weights, bias, use activation fu   ic
   n00 = ([1.0, -1.0], 0.0, True)
   n01 = ([1.0, 1.0], 0.0, True)
   hidden_layer0 = [n00,n01]

   n10 = ([0.5, -0.2], 0.0, True)
   n11 = ([-0.5, 0.1], 0.0, True)
   hidden_layer1 = [n10, n11]

   # don't use relu for outputs
   o0 = ([1.0, -1.0], 0.0, False)
   o1 = ([-1.0, 1.0], 0.0, False)
   output_layer = [o0, o1]

   dnn = [hidden_layer0, hidden_layer1, out
   ```

   In this DNN, the outputs of the neurons in the hidden layers (prefixed with `n`) are applied with the `relu` activation function, but the outputs of the DNN (prefixed with `o`) are not. These settings are controlled by the `True`, `False` parameters as shown above. Also, this example does not use `bias`, i.e., bias values are all 0.0's as shown. Note that all of these settings are parameterized and I deliberately use this example to show these how these parameters are used (e.g., `relu` only applies to hidden neurons, but not outputs).

2. **Interval (Abstract) Value**: Now you will implement a function called `my_interval_execution` that takes as inputs the dnn and a precondition over inputs, represented by the list of intervals over inputs. The function then evaluate our `dnn` over these input intervals, and obtain a list of output intervals, representing values of the outputs. More specifically, `my_interval_execution` returns a dictionary that contains input, output, and hidden nodes as keys and intervals as values (e.g., `'i0': [-2,1], 'o0':[0, 7], 'n0_0':[-3,4]` , etc).

   i. Unlike PA1, you can assume that the preconditions will be have **input values bounded**, e.g., `0.1 <= x` is not allowed because it has no upperbound but `0.1 <= x <= 1.2` is OK. This also mean that every ReLU and outputs will also be bounded throughout the computation. Note that the post condition still have no restrictions, e.g., you still can have 0.1 <= o1 for post condition , and so if your interval gives you o1 = [0.2, 1.2], then that proves 0.1 <= o1.

   ii. With `my_interval_execution` implemented, you can do various tasks, including:

      a. Simulating a concrete execution

      ```
      # finding outputs when inpu     ar
      states = my_interval_execution(c
      print(states)
      # {... 'o0': [1.0, 1.0], 'o1': [
      ```

      a. Execute using precondition

      ```
      # executing using precondit     e
      states = my_interval_execution(c
      print(states)
      ```

```
# {..., 'o0': [0.0, 0.5], 'o1':
```

a. Checking assertions, i.e., verifying/proving properties about the DNN

```
# MANUAL checking if 0.1 <=     <
# check that the output interval
# If the output is {..., 'o0': [
# If the output is {..., 'o0': [
```

3. **Scalability of Interval**: now reuse the DNNS that you have created in Part1 and run `my_interval_execution` on them. Plot the results and compare with the results from part 1. You should see that interval abstraction scales much better than symbolic execution.

TIPS

1. It is strongly recommend that you do interval abstraction on this DNN example **by hand** first before attempt any coding. This example is small enough that you can work out step by step. For example, you can do these two steps
   i. First, obtain the intervals of each neurons and outputs by hand (e.g., on a paper) for the given DNN
   ii. Then, put what you have in code but also for the given DNN. Use simple list or tuple for intervals if you'd like (e.g., `i0 = [0, 3], i1 = [None, -1]` represent `0 <= i0 <= 3` and `i1 < -1`), then compute intervals for hidden neurons and outputs, etc)
   iii. Finally, convert what you have into a general program that would work for any DNN inputs.

Conventions and Requirements

Your program must have the following (all of these are given in the concrete example above)

Part 1

1. a main function `create_random(sizes)` that
   - takes as input the list `sizes` and returns a DNN of that size as descibed above.
2. a plot (in pdf, png, or jpeg) showing the scalability of `my_symbolic_execution` on various DNNs
3. In the README, write your observation and conclusion about the scalability of symbolic execution. You must refer to the plot and describe it to support your conclusion.

Part 2

1. a main function `my_interval_execution(dnn)` that
   - takes as input the DNN, whose specifications are given as example above (or using `tensorflow` or `pytorch`)
   - This goes without saying: do not hard-code the DNN in your program (e.g., do not hardcode the example given above in your code). Your program should work with any DNN input.
   - returns *states* of the DNN represented as a dictionary mapping neuron name as string to intervals as lists (or tuples) of size 2
   - this goes without saying but do NOT write this function only to work with the given example (i.e., do not hard-code the weight values etc in your program). This function should work with any DNN input (though it could be slow for big DNN's).
2. in your resulting formula, you must name
   - the `nth` input as `in` (e.g., the first input

    is `i0`)
  - the `nth` output as `on` (e.g., the second output is `o1`)
3. a testing function `test1()` where you copy and paste pretty much the complete examples given above to demonstrate that your SE works for the given example. In summary, your `test1` will include
   - the specification of the 2x2x2x2 DNN in the Figure
   - run the interval abstraction on the dnn
   - output the state results as dictionary
   - simulate concrete execution
   - simulate execution with precondition
   - MANUAL CHECKING OF ASSERTIONS AS COMMENTS LIKE ABOVE
4. another testing function `test2()` where you create another DNN:
   - The DNN will have
     - the same number of 2 inputs and 2 outputs, but 3 hidden layers where each layer has 4 neurons, i.e., a 2x4x4x4x2 DNN.
     - non-0.0 bias values.
   - then on this DNN, do every single tasks you did in `test1()` .
5. a plot (in pdf, png, or jpg) showing the scalability of `my_int_execution` on the various DNNs generated in Part 1.
6. In the README, write your observation and conclusion about the scalability of symbolic execution. You must refer to the plot and describe it to support your conclusion.

What to Turn In

You must turn in a **zip file** containing the following files. Use the following names:

1. the zip file must be named `pa2-yourgroupname.zip`
2. One single main source file `se.py` . Indicate in the README file on how I can run it.
   i. your file should include at least a `create_random` function and a `my_interval_execution` function and two test functions `test1() and test2()` as indicated above
   ii. you **do not** need to include code to generate plots. You can use any plotting tool you like (e.g., Excel, Google Sheets, etc) to generate the plots.
3. **Include the plots (e.g., as images) in your zip file and in your README file**.
4. a `readme.md` file (**IMPORTANT**: note that this is a Markdown file, so that you can include graphics. See the details below, I've made some additional requirement)
5. For each part:
   - 2 screen shots showing how (1) you start your program (e.g., the commands used to and run your program) and (2) the end of the run
6. Nothing else, do not include any binary files or anything else.

The `readme.md` file should be a *plain MARKDOWN text file* (not a Word file, not an RTF file, not an HTML file) consisting of the following:

1. **Complete run** show how you run your program **AND** the its complete outputs, e.g.,

```
$ python3 se.py   #this should exec
....              #and show me all outpu
```

2. **Answer the following questions about your SE**

   i. Briefly describe what you did for part 1 and part 2.

ii. Describe the plots and conclusions you have for scalability of both parts 1 and 2 (see above).
iii. What do you think the most difficult part of this assignment? (e.g., programming, plotting, understanding etc)
iv. What advice do you give other students (e.g., students in this class next year) about this PA?

Grading Rubric

Grading (out of 30 points):

- 15 points — for a complete interval abstraction (part 2) as described
  - 10 points if your code works for the given example (i.e., `test()` produces similar results)
  - 3 points if your code works for the addition example you created
  - 2 points if your code works on my own example (not provided). This means you should try to generalize your work and test it on various scenarios.
- 10 points - for complete scalability analysis and plots (part 1 and part 2) as described
  - 5 points for `create_random` code
  - 5 points for scalability plots and explanations
- 5 points — for the answers in the README
  - 5 — clear explanations on running the program and thorough answers for the given questions
  - 2 — vague or hard to understand; omits important details
  - 0 — little to no effort, or submitted an

RTF/DOC/PDF file instead of plain MD