PA1

Edit New page

ThanhVu (Vu) Nguyen edited this page on Sep 17 · 16 revisions

CS695/SWE699 (Fall'03)

PA1: Symbolic Execution on Deep Neural Networks

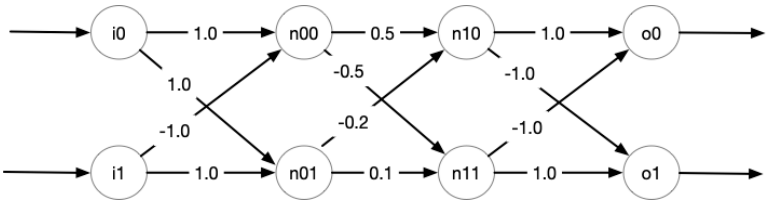
Goal

In this assignment, you will implement symbolic execution (SE) on a given fully-connected (deep) neural network (DNN). SE is a popular and well known software testing approach while DNN is an increasingly important and practical application --- this PA will combine those two. You can consider DNN as a specific type of programs and thus you can "execute" it. Symbolic execution would executes the DNN on symbolic inputs and returns symbolic outputs, i.e., a constraint representation as discussed in class. In addition, we can represent the symbolic outputs as a logical formula and use a constraint solving such as *Z3* to check for "assertions", i.e., properties about the DNN.

You will use Python (3.x is preferred) and the Z3 SMT solver for 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.

Complete Example

Consider the following fully-connected DNN with 2 inputs, 2 hidden layers, and 2 outputs.



1. **DNN Encoding**: We can encode this DNN and additional details using Python:

```
# (weights, bias, use activation fu
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, ou
```

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).

Clone this wiki locally

https://github.com/nguyenthanh

2. **Symbolic States**: After performing symbolic execution on `dnn`, we obtain `symbolic states`, represented by a logical formula relating input and output variables.

```
# my_symbolic_execution is something y
# it returns a single (but large) formula r
symbolic_states = my_symbolic_execution(dnn
...
"done, obtained symbolic states for DNN wit
assert z3.is_expr(symbolic_states)  #symbol

# your symbolic states should look somethin
print(symbolic_states)
# And(n0_0 == If(i0 + -1*i1 <= 0, 0, i0 + -
#     n0_1 == If(i0 + i1 <= 0, 0, i0 + i1),
#     n1_0 ==
#     If(1/2*n0_0 + -1/5*n0_1 <= 0, 0, 1/2*
#     n1_1 ==
#     If(-1/2*n0_0 + 1/10*n0_1 <= 0, 0, -1/
#     o0 == n1_0 + -1*n1_1,
#     o1 == -1*n1_0 + n1_1)
```

1. **Cosntraint Solving**: We can use a constraint solver such as `Z3`` to query various things about this DNN from the obtained symbolic states. Examples include:

    i. Generating random inputs and obtain outputs. Note that these are random values (that satisfy the symbolic states), so your results might look different.

```
z3.solve(symbolic_states)
[n0_1 = 15/2,
    o1 = 1/2,
    o0 = -1/2,
    i1 = 7/2,
    n1_1 = 1/2,
    n1_0 = 0,
    i0 = 4,
    n0_0 = 1/2]
```

ii. Simulating a concrete execution.

```
i0, i1, n0_0, n0_1, o0, o1 3.

# finding outputs when inputs ar
g = z3.And([i0 == 1.0, i1 == -1.
z3.solve(z3.And(symbolic_states,
[n0_1 = 0,
o1 = -1,
o0 = 1,
i1 = -1,
n1_1 = 0,
n1_0 = 1,
i0 = 1,
n0_0 = 2]
```

iii. Checking assertions, i.e., verifying/proving properties about the DNN or disproving/finding counterexamples

```
print("Prove that if (n0_0 (
g = z3.Implies(z3.And([n0_0 > 0.
print(g)  #  Implies (And(n0_0 >
z3.prove(z3.Implies(symbolic_sta

print("Prove that when (i0 - i1
g = z3.Implies(z3.And([i0 - i1 >
print(g)  # Implies(And(i0 - i1
z3.prove(z3.Implies(symbolic_sta
# proved

print("Disprove that when i0 - i
g = z3.Implies(i0 - i1 > 0.0, o(
print(g)  # Implies(And(i0 - i1
z3.prove(z3.Implies(symbolic_sta
# counterexample  (you might get
# [n0_1 = 15/2,
# i1 = 7/2,
# o0 = -1/2,
# o1 = 1/2,
# n1_0 = 0,
# i0 = 4,
# n1_1 = 1/2,
# n0_0 = 1/2]
```

TIPS

1. It is strongly recommend that you do symbolic execution 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 symbolic states 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 Z3 format (e.g., you would declare the inputs as symbolic values `i0 = z3.Real("i0")` , then compute the neurons and outputs, etc)

    iii. Finally, convert what you have into a general program that would work for any DNN inputs.

2. You can use the below method to construct Z3 formula for `relu(v)=max(v,0)`

    ```
    @classmethod
    def relu(cls, v):
       return z3.If(0.0 >= v, 0.0, v)
    ```

3. Two ways of doing symbolic execution to obtain symbolic states

    i. You can follow the traditional SE method which produces the symbolic execution trees in which each condition representing ReLU forks into two paths. You can decide whether to do a depth-first search or breadth-first search here (they will have the same results). At the end, the `symbolic states` are a disjunction of the path conditions (i.e., `z3.And[path_conds]`).

ii. But since we are using Z3, a much much simpler way, is to encode all those forked paths directly as a formula. For example

```
if (x + y > 0):
   r = 3
else:
   r = 4
```

Using traditional SE, you would have 2 paths, e.g., path 1: `x+y > 0 && r = 3` and path 2 : `x+y <= 0 && r ==4` , from which you take the disjunction and get `(x+y > 0 && r == 3) || (x+y <= 0 && r ==4)`). But for this assignment, instead of having to fork into two paths, you can use Z3 to encode both branches using the `If` function, e.g., `If(x+y>0, r==3, r==4)` or `r==If(x+y>0,3,4)` . The results of these two methods are exactly the same at the end, just that the prior, traditional one you do more work while the later you do less. It is up to you.

4. When `bias` is none-zero, then it will simply be added to the neuron computation, i.e., `neuron = relu(sum(value_i * weight_i) + bias)` . For example, if `bias` is `0.123` then for neuron `n0_0` we would obtain `n0_0 == If(i0 + -1*i1 + 0.123 <= 0, 0, i0 + -1*i1 + 0.123)` .

5. To install Z3, you can simply use `pip3`

```
cse fac/tnguyen> pip3 install z3-s
Defaulting to user installation because
Collecting z3-solver
Downloading z3_solver-4.8.9.0-py2.py3-n
   |████████████████████████████████| 3
Installing collected packages: z3-solve
Successfully installed z3-solver-4.8.9.
WARNING: You are using pip version 20.2
```

```
You should consider upgrading via the ',
pyenv: cannot rehash: /usr/local/python,
cse fac/tnguyen> python
Python 3.7.2 (default, Aug 15 2019, 13:4
[GCC 4.8.5] on linux
Type "help", "copyright", "credits" or '
>>> import z3
>>> z3.get_version()
(4, 8, 9, 0)
```

6. Z3 Resources

   Some links for Z3 that you might find useful

   - https://microsoft.github.io/z3guide/programming/Z3%20Python%20-%20Readonly/Introduction : good intro for beginner.
   - https://z3prover.github.io/api/html/namespacez3py.html : Z3 API document

   However, you don't really need to know much about Z3 except for a few methods used in class and given throughout this PA. Here's probably all you need to know

```
import z3

x = z3.Real("x")
y = z3.Real("y")
f0 = x == 3.5  # an equality x == 3.5
f1= x == y + 3.5  # an equality x = y
+ 3.5
f2 = z3.If(y <= 0, 0, y)

f3 = z3.Implies(x <= 3.7, x <= 5)  #x
<= 3.7 => x <= 5
z3.prove(f3)  #can prove

f4 = z3.Implies(x<=5, x<=3.7)
z3.prove(f4)   # cannot prove

mylist = [x <= 3,   y <= 5]
f5 = z3.And(mylist)
z3.solve(f5)  #check if f5 is
```

```
satisfiable,  should be yes
```

7. Using `Tensorflow Keras` (or `Pytorch`) In my example above, I create a DNN using simple Python lists. But you can also use `Tensorflow Keras` (or `Pytorch`) to create a DNN. It might be simpler that way as then you can reuse existing methods from these Python packages. You will still need to implement symbolic execution, just that instead the input DNN being Pythong lists, they are now Tensorflow or Pytorch. Also note that if you use `tensorflow` or `pytorch`, then be sure to indicate what packages are needed to run your program in the **README** file, i.e., `pip install ...`.

   Below is the tensorflow representation of the DNN example above.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Se
from tensorflow.keras.layers import De
from tensorflow.keras import activatio

model = Sequential()

# layer 0: nodes n00, n01
model.add(Dense(units=2,
                input_shape=(2, ),
                activation=activations
                kernel_initializer=tf.
                    [[1.0, 1.0],  # we
                    [-1.0, 1.0]  # wei
                    ]),
                bias_initializer=tf.co
                    [[0.0],  # bias of
                    [0.0]]  # bias of
                ),
                dtype='float64'
                ))
```

```
# layer 1: nodes n10, n11
model.add(Dense(units=2,
                activation=activations
                kernel_initializer=tf.
                    [[0.5, -0.5], [-0.
                bias_initializer=tf.co
                dtype='float64'
                ))

# last layer: nodes represent outputs
model.add(Dense(units=2,
                activation=None,  # no
                kernel_initializer=tf.
                    [[1.0, -1.0], [-1.
                bias_initializer=tf.co
                dtype='float64'
                ))
```

Conventions and Requirements

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

1. a main function `my_symbolic_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 *symbolic states* of the DNN represented as a `Z3 expression` (a formula) as shown above
   - 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`)
    - the `jth` neuron at the `ith` layer as `ni_j` . Note that the first layer is the `0th` layer
3. a testing function `test()` 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 `test` will include
    - the specification of the 2x2x2x2 DNN in the Figure
    - run the symbolic execution on the dnn (as shown above, it should output the dimension of the dnn and runtime)
    - output the symbolic states results
    - apply Z3 to the symbolic states to obtain
        - random inputs and associated outputs
        - simulate concrete execution
        - checking the 3 assertions as shown
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 `test()` (run SE on it, output results, apply Z3 etc). You will need to have 2 assertions that are true and 1 assertion that is false (just like above).

        - Initially you might randomly generate weights and bias values, but it is likley that you will need to manually adjust them so that you can prove some correct assertions (randomly generated values probably will not give you a meaningful DNN with any asserted properties).

You can be as creative as you want, but your SE must not run for too long and must not take up too much space (e.g., do not generate over 50MB of files). Use the `README` to tell me exactly how to compile/run your program, e.g., `python3 se.py` .

What to Turn In

You must submit a **zip file** containing the following files via Blackboard. Use the following names:

1. the zip file must be named `pa1-yourgroupname.zip` (1 submission per group)
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 `my_symbolic_execution` function and two test functions `test and test2()` as indicated above
3. a `readme.txt` file (**IMPORTANT**: see the details below, I've made some additional requirement)
4. 2 screen shots showing how (1) you start your program (e.g., the commands used to compile and run your program) and (2) the end of the run (e.g., of your program on the terminal screen)
5. Nothing else, do not include any binary files or anything else.

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

1. **Comlete 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 your SE algorithm

    ii. What do you think the most difficult part of this assignment? (e.g., programming symbolic execution, using Z3, etc)

    iii. What advice do you give other students (e.g., students in this class next year) about this PA?

Grading Rubric

Grading (out of 20 points):

- 15 points — for a complete SE as described
    - 10 points if your SE works for the given example (i.e., `test()` produces similar results)
    - 3 points if your SE works for the addition example you created
    - 2 points if your SE works on my own example (not provided). This means you should try to generalize your work and test it on various scenarios.
- 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 TXT

+ Add a custom footer