ECE 449/590, Fall 2022
Project 2: ctypes and Scalar Operations

Due: 10/09 (Sun.), by the end of the day (Chicago time)

1 Summary

In this project, we will implement scalar operations for EasyNN in C++. This is first of the three projects where we will provide full support of EasyNN in C++ incrementally. We will implement matrix and tensor operations in C++ in Proejct 3 and 4 respectively.

The ctypes library is utilized to bridge Python and C++ code. ctypes is a foreign function library for Python that allows a Python program to interact with shared libraries developed in other languages. While there are other mechanisms enabling interoperations between Python and other languages, ctypes is fairly straightforward to work with.

In EasyNN, the python code uses ctypes to call the following functions that you will need to implement in C++. You can find a list of these functions in `src/libeasynn.h`. Note that since these functions serve as the interface between Python and C++, you are not supposed to change their signatures, i.e. the return type, the function name, and the parameter list.

- `program *create_program()`: this is the first function the EasyNN Python code will call. It should create and return a `program` object that holds the DAG in the SSA form. This `program` object will be passed back as the first parameter of other functions (discussed later) so that you can store information like expressions into it.

- `append_expression(prog, expr_id, op_name, op_type, inputs, num_inputs)`: this function is called for each expression in the SSA form, following the topological order that the expressions should be evaluated. The `program` object you created and returned in `create_program()` is passed back to you as the `prop` parameter. The remaining parameters cover various aspects of the expression and please refer to the lecture slides for details.

- `int add_op_param_double(prog, key, value)`: this function is called for each scalar parameter for the operator in the expression that is decribed in the previous `append_expression()`. If an operator does not use parameters, then this function

will not be called. For Project 2, the only operator that will use a parameter is `Const` – the `key` will be `"value"` (a C-style string), and the `value` will be the actual scalar parameter in `double`. This function should return `0` for success.

- `int add_op_param_ndarray(prog, key, dim, shape, data)`: this function is similar to `add_op_param_double()` but is for the parameters that are tensors. For Project 2, this function will not be called and you can simply return `0`.

- `evaluation *build(prog)`: this function is called to initialize an evaluation of the DAG in the SSA form held by `prog`. It should create and return an `evaluation` object that holds necessary information for the evaluation, like the values of the inputs. Keep in mind that you should not store such information in the `program` object since we may evaluate a DAG multiple times, each with a different set of inputs.

- `add_kwargs_double(eval, key, value)`: this function is called for each scalar input. For example, if the input is `a`, then the `key` will be `"a"` (a C-style string), and the `value` will be the actual scalar input in `double`.

- `add_kwargs_ndarray(eval, key, dim, shape, data)`: this function is similar to `add_kwargs_double()` but is for the inputs that are tensors. For Project 2, this function will not be called and you can leave it empty.

- `int execute(eval, p_dim, p_shape, p_data)`: this function should actually perform the evaluation and return the result to Python. Since quite some details of ctypes and numpy are required to convert from the result in C++ back to Python, we provide the code for such conversion. You should not modify the code of `execute()` in `src/libeasynn.cpp` and should implement the steps for evaluation in `evaluation::get_result()`.

For your convenience, we provide initial implementations of the above functions in `src/libeasynn.cpp`, together with a few initial/recommended class type definitions, and a testing program. While we will discuss many of them in the lectures, you are still required to read the code and run the testing program to observe its output in order to understand how the various parts of the C++ implementations work together.

This project should be done individually. Discussions are encouraged. However, all the programs (except those from the lectures) and writings should be by yourself. COPY without proper CITATION will be treated as PLAGIARISM and called for DISCIPLINARY ACTION.

NEVER share your programs/writings with others.

2 Working with Your Projects

For Project 2 and all following projects, you should still work with the Git repository you worked with for Project 1. Here is a brief introduction of the files we will use for Project 2.

- `easynn.py` and `easynn_golden.py`: they are the same as those in Project 1. You should not modify them.
- `src`: this directory contains all your C++ implementations. You'll need to modify the .cpp and .h files inside. You may add/remove .cpp and .h files as needed.
- `easynn_test.cpp`: this is the testing program helping you to debug your C++ implementations. It includes a very simple test case so you should not modify it until your implementation starts to work. Then you may need to update it for more complex test cases.
- `Makefile`: this file defines how to create the shared library from your C++ implementations, as well as the testing program. You should not modify this file.
- `easynn_cpp.py`: this is the Python driver utilizes ctypes to interact with your C++ implementations. It provides the same Builder interface as `easynn_golden.py` so that it can be used the same way as our reference implementation in NumPy. You should not modify this file.
- `grade_p2.py`: this is the grading script to verify whether your C++ implementations in `src` are correct or not. There are 10 questions. You should not modify this file.

After creating the shared library using "make", run the grading script to see if all questions pass.

```
make
python3 grade_p2.py
```

Note that this command only tests your code in your own VM. Any tests you run in your own VM, passed or not, do NOT count towards your project grades. You will need to commit and push your code to the CI system to start the grading process and will need to access the grading report on uranus.ece.iit.edu to correct any issues before the deadline.

3 Deliverables and Grading

We obtain a copy of all your source files in `src` as you push the changes to the central Git repository so there is no need for you to submit them to us using any other mechanisms. Moreover, please be advised that since to learn the use of Git and a CI system is among the objectives of this course, we will NOT accept project submissions outside the central Git repository, e.g. via emails. If you have difficulty accessing the central Git repository, it is your responsibility to act promptly to seek help from us well before the project deadline; otherwise, not able to access the central Git repository is NOT an excuse for late submissions.

Project 2 will have a full grade of 100 points. Each function, if passed, will give you 10 points. Since you are required to use the CI system to troubleshoot any issues with your code before the deadline, a failed function will earn 0 points.

The following submission checklist is provided for your convinience. Detailed instructions are available from Section IV of Guide to System Setup and Work Flow.

- ☐ Run `python3 grade_p2.py` in VM to make sure all 10 tests pass.
- ☐ Commit and push your changes to the central Git repository.
- ☐ Run `/home/ece449/show` on uranus.ece.iit.edu to access your grading report and correct any issues before the deadline.

4 Hints

Since this is probably the first time you are working with existing code instead of writing every line of code by yourself from the beginning, you may feel that it is very difficult to start. Here are a few steps that you may follow.

1. Read `easynn_test.cpp` and try to reason with it: what are the expressions and what is the expected output? You should be able to find the answers in our lectures.
2. Without changing any code, run `make` and then `./easynn_test`. For each line in the output, locate the source file that generates it and explain what that line means.
3. Modify your code, print the information regarding the expressions like its id, operator, and operands in `evaluation::evaluation`. You will need to pass those expressions from `program` to `evalution` int the `build` function.
4. Modify your code, in `evaluation::add_kwargs_double`, print the value of the input `a` and find a place to store it so it can be used later.
5. Modify your code, print the value of the input `a` in `evaluation::execute`. This is the value you have stored somewhere in 4.

6. Modify your code, print the result of the computation in `evaluation::execute`.

7. Modify your code, in `evaluation::get_result`, print the result of the computation. Similar to 4 and 5, you store this value somewhere in 6 to be used here.

8. Modify your code to generate correct results in `./easynn_test`. If you run `python3 grade_p2.py` now, Q1 should pass.

9. To pass Q2, you will need to understand the difference between the two functions `add_kwargs_double` and `add_op_param_double()` since the `Const` operator is handled by the latter. Read Section 1 of this instruction again and observe the output of Q2 from `python3 grade_p2.py`.

10. You will feel more comfortable working on Project2 after you solve Q2. Nevertheless, please practice incremental and interactive developement to solve Q3 to Q10 one at a time, instead of trying to solve all of them at once.