Assignment 3. Reverse-Mode Automatic Differentiation

Due Feb 14 by 11:59pm **Points** 20
Submitting an external tool
Available Jan 31 at 12am - Feb 14 at 11:59pm 15 days

This assignment was locked Feb 14 at 11:59pm.

Due 11:59pm, Feb 14, 2020

Files to submit:

autodiff.py (You only need to finish these "TODO" parts, Do Not change the file name)

Please submit to GradeScope Assignment 3. This assignment will be automatically graded. If all your code passes **autodiff_test.py** by run

- # sudo pip install nose
- # nosetests -v autodiff_test.py

If all your test cases pass, you should see something like this after you submit your code to GradeScope:

Autograder Results

Results Code

test_add_by_const (test_autodiff.TestAutoDiff) (2.0/2.0)

test_add_mul_mix_1 (test_autodiff.TestAutoDiff) (2.0/2.0)

test_add_mul_mix_2 (test_autodiff.TestAutoDiff) (2.0/2.0)

test_add_mul_mix_3 (test_autodiff.TestAutoDiff) (2.0/2.0)

test_add_two_vars (test_autodiff.TestAutoDiff) (2.0/2.0)

test_grad_of_grad (test_autodiff.TestAutoDiff) (2.0/2.0)

test_identity (test_autodiff.TestAutoDiff) (2.0/2.0)

test_matmul_two_vars (test_autodiff.TestAutoDiff) (2.0/2.0)

test_mul_by_const (test_autodiff.TestAutoDiff) (2.0/2.0)

test_mul_two_vars (test_autodiff.TestAutoDiff) (2.0/2.0)

STUDENT
Jia Zou

AUTOGRADER SCORE
20.0 / 20.0

PASSED TESTS
test_add_by_const (test_autodiff.TestAutoDiff) (2.0/2.0)
test_add_mul_mix_1 (test_autodiff.TestAutoDiff) (2.0/2.0)
test_add_mul_mix_2 (test_autodiff.TestAutoDiff) (2.0/2.0)
test_add_mul_mix_3 (test_autodiff.TestAutoDiff) (2.0/2.0)
test_add_two_vars (test_autodiff.TestAutoDiff) (2.0/2.0)
test_grad_of_grad (test_autodiff.TestAutoDiff) (2.0/2.0)
test_identity (test_autodiff.TestAutoDiff) (2.0/2.0)
test_matmul_two_vars (test_autodiff.TestAutoDiff) (2.0/2.0)
test_mul_by_const (test_autodiff.TestAutoDiff) (2.0/2.0)
test_mul_two_vars (test_autodiff.TestAutoDiff) (2.0/2.0)

Files for testing (This test file include and only include the ten gradescope test cases)

autodiff_test.py (You do not need to change this file and you do not need to submit this file)

In this assignment, we would implement reverse-mode auto-diff.

Our code should be able to construct simple expressions, e.g. y=x1*x2+x1, and evaluate their outputs as well as their gradients (or adjoints: gradients from two output edges), e.g. y, dy/dx1 and dy/dx2.

There are many ways to implement auto-diff, as explained in the slides for Lecture 6. For this assignment, we use the approach of a computation graph and an explicit construction of gradient (adjoint) nodes, similar to what MXNet and Tensorflow do.

Key concepts and data structures that we would need to implement are

- Computation graph and Node
- Operator, e.g. Add, MatMul, Placeholder, Oneslike
- Construction of gradient nodes given forward graph
- Executor

Overview

Here we use a simple example to illustrate the API and data structures of the autodiff module.

Suppose our expression is y=x1*x2+x1, we first define our variables x1 and x2 symbolically,

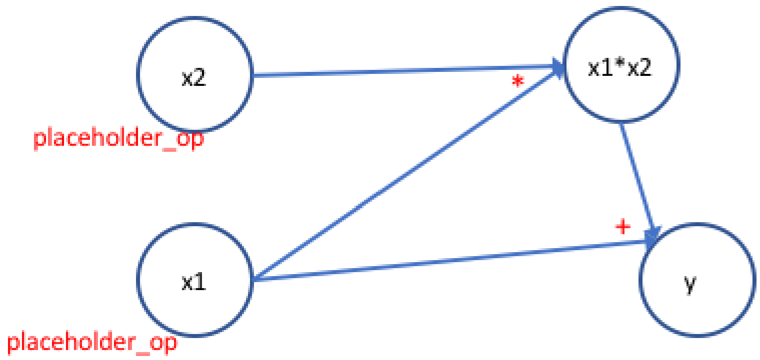
```
import autodiff as ad

x1 = ad.Variable(name = "x1")
```

```
x2 = ad.Variable(name = "x2")
```

Then, you can define the symoblic expression for y,

```
y = x1 * x2 + x1
```

Now, the computation graph looks like this,



[(https://github.com/dlsys-course/assignment1/blob/master/img/hwk1_graph1.png)](https://github.com/dlsys-course/assignment1/blob/master/img/hwk1_graph1.png)

Here, each node is associated with an operator object (we only need a singleton instance for each operator since it is used in an immutable way).

- Node x1 and x2 are associated with Placeholder Op.
- Node (x1*x2) is associated with MulOp, and y with AddOp.

With this computation graph, we can evaluate the value of y given any values of x1 and x2: simply walk the graph in a topological order, and for each node, use its associated operator to compute an output value given input values. The evaluation is done in Executor.run method.

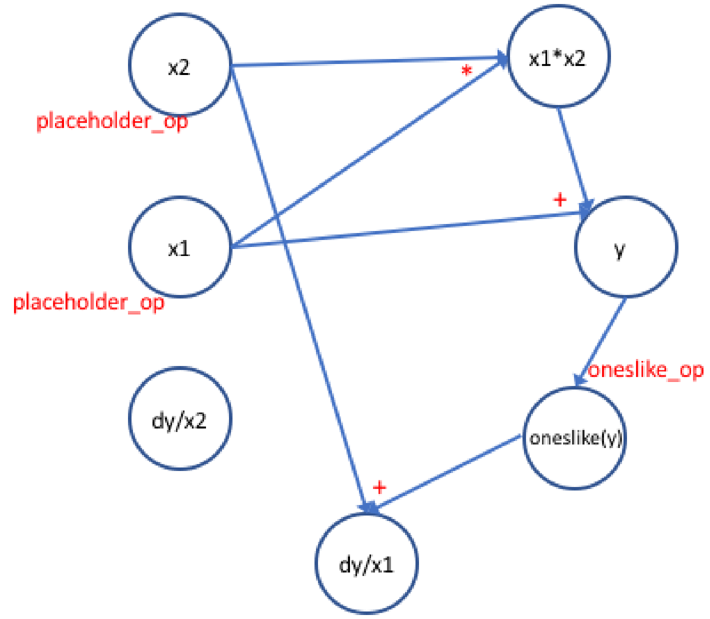
```
executor = ad.Executor([y])
y_val = executor.run(feed_dict = {x1 : x1_val, x2 : x2_val})
```

If we want to evaluate the gradients of y with respect to x1 and x2, as we would often do for loss function wrt parameters in usual machine learning training steps, we need to construct the gradient nodes, grad_x1 and grad_x2.

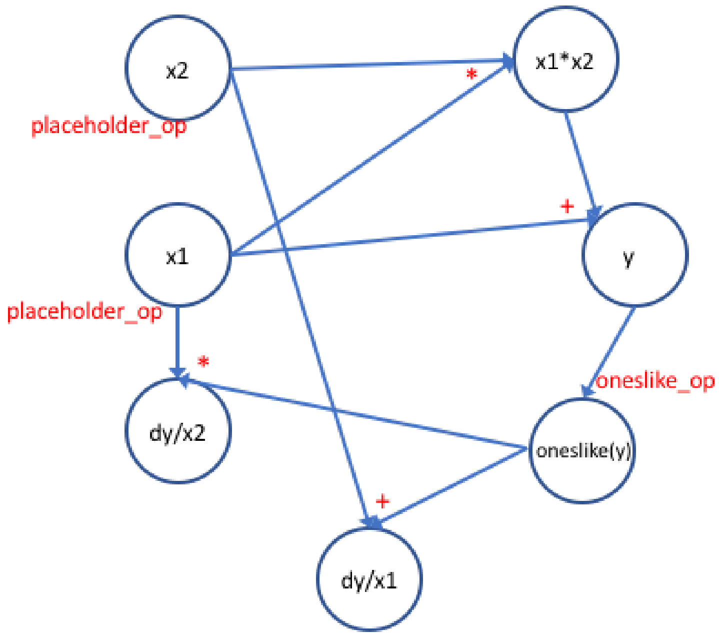
```
grad_x1, grad_x2 = ad.gradients(y, [x1, x2])
```

According to the reverse-mode autodiff algorithm described in the lecture, we create a gradient node for each node in the existing graph and return those that user are interested in evaluating.

We do this in a reverse topological order, e.g., y, (x1+x2), x1, x2, as shown in the figures below



[(https://github.com/dlsys-course/assignment1/blob/master/img/hwk1_graph2.png)](https://github.com/dlsys-course/assignment1/blob/master/img/hwk1_graph2.png)



[(https://github.com/dlsys-course/assignment1/blob/master/img/hwk1_graph3.png)](https://github.com/dlsys-course/assignment1/blob/master/img/hwk1_graph3.png)

Once we construct the gradients node, and have references to them, we can evaluate the gradients using Executor as before,

```
executor = ad.Executor([y, grad_x1, grad_x2])
y_val, grad_x1_val, grad_x2_val = executor.run(feed_dict = {x1 : x1_val, x2 : x2_val})
```

grad_x1_val, grad_x2_val now contain the values of dy/dx1 and dy/dx2.

Special Notes

- For simplicity, our implementation expect all variables to have numpy.ndarray data type. See tests feed_dict usage;

- taking derivative of dy/dx, even though y can be a vector, we are implicitly assuming that we are taking derivative of the reduce_sum(y) wrt x. This is the common case for machine learning applications as loss function is scalar or the reduce sum of vectors. Our code skeletion takes care of this by initializing the dy as ones_like(y) in Executor.gradients method.

What you need to do?

- Understand the code skeleton and tests. Fill in implementation wherever marked """TODO: Your code here""".
- We have 10 tests in autodiff_test.py. We would grade you based on those tests.
- Run all tests with # sudo pip install nose #nosetests -v autodiff_test.py

Bonus points

Once your code can clear all tests, your autodiff module is almost ready to train a logistic regression model. If you are up for a challenge, try

- Implement all missing operators necessary for a logistic regression, e.g. log, reduce_sum.
- Write a simple training loop that updates parameters using gradients computed from autodiff module.

Grading Rubrics

- autodiff_test.test_identity ... 2 pt
- autodiff_test.test_add_by_const ... 2 pt
- autodiff_test.test_mul_by_const ... 2 pt
- autodiff_test.test_add_two_vars ... 2 pt
- autodiff_test.test_mul_two_vars ... 2 pt
- autodiff_test.test_add_mul_mix_1 ... 2 pt
- autodiff_test.test_add_mul_mix_2 ... 2 pt
- autodiff_test.test_add_mul_mix_3 ... 2 pt
- autodiff_test.test_grad_of_grad ... 2 pt
- autodiff_test.test_matmul_two_vars ... 2 pt
- bonus (training logistic regession) ... 1 bonus pt added to your final grade