Lab 7: Completing the from-scratch library

5/2/2023

100 Possible Points

Attempt 1

4/27/2023
Next Up: Review Feedback

Attempt 1 Score:
N/A

Add Comment

Unlimited Attempts Allowed

Details

In this week's lab you will submit your completed library. You will not yet use your library to train a neural network.

Incorporate Your Instructor's Feedback

You must resubmit your graded backpropagation derivations from Lab 6 along with your Lab 7 submission that reworks these derivations. Be sure to address any concerns provided in the feedback to your Lab 6. In many cases, this will mean reworking the derivations on a new sheet of paper included alongside the original derivations. In some cases, you can make minor edits suggested by the instructor.

You must also submit your feedforward equations from Lab 3 to which you compare the output of your test client. In these notes, please clearly label and mark the result you expect your simple client to produce. Resubmit your test client as in Lab 6, including the output that you predict your client should produce.

Your paper derivations for backpropagation should include:

- A single example of testing the `step` function
- A single example of testing the `accumulate_grad` function
- All the numeric inputs and outputs needed to test each type of layer through backpropagation
  - There should be separate, independently-runnable tests for each layer. Don't just compute backprop on the whole network, compute backprop for each layer

independently so that the tests confirm the input gradients for that layer are correct.
  - Each network layer should accept arbitrary output gradients, even if we know the gradient is always 1 for the use-cases in which we will use it.
  - You don't need clearly label which numbers are for which as long as your derivations show enough detail to give the idea.
  - But the numbers and calculations need to be there.
  - **I strongly recommend working these examples by hand** rather than using a calculator or parallel torch autograd implementation.
  - **Show your work.** If you worked out numbers on scratch paper, write some notes about what those scratches mean. Explain how you came to your numeric answers either through showing your work (preferred) or through a written explanation of how you reached the result
- **Tests should be useful.** For example, don't simply use backpropagation tests where the numeric results collapse to a scalar 0 gradient. A matrix with the correct shape and somewhat-interesting numbers is required.

If you have doubts about what a layer is supposed to do, please consult with your instructor as early in the week as possible. This is the week in which you master your own library before training it on real data next week.

Your library does NOT need to include a stochastic gradient descent loop. You will write this loop next week with your client.

The `backward`, `accumulate_grad`, and `step` methods

You will implement the the `backward`, `accumulate_grad`, and `step` methods and any other methods you see that are needed to complete your library classes.

The multi-variate chain rule states that if $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})$ and $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are all functions of $\boldsymbol{y}$, then $\frac{\partial x}{\partial y} = \frac{\partial x}{\partial a}\frac{\partial a}{\partial y} + \frac{\partial x}{\partial b}\frac{\partial b}{\partial y} + \frac{\partial x}{\partial c}\frac{\partial c}{\partial y}$. Because of the multi-variate chain rule, we can compute the gradient of a layer as the sum of the gradients from all layers that receive that layer's output directly. This motivates our implementation of the `backward` and `accumulate_grad` methods:

The `backward` method should perform back-propagation through the single layer. At the time backward is called, you may assume that `self.grad` already contains $\frac{\partial J}{\partial \text{output}}$, that is, the derivative of the objective function with respect to this layer's output. It then computes $\frac{\partial J}{\partial \text{input}}$ for each input to this layer. For example, the Layer Linear class would receive $\frac{\partial J}{\partial \mathbf{z}}$ and compute $\frac{\partial J}{\partial \mathbf{W}}$, $\frac{\partial J}{\partial \mathbf{b}}$, and $\frac{\partial J}{\partial \mathbf{x}}$. These gradients are for an arbitrary Linear layer, not just the first layer of the example network used for the backpropagation derivations. Although $\frac{\partial J}{\partial \mathbf{x}}$ is not needed in our entire network as $\mathbf{x}$ is not a learnable parameter, it would be needed, for example, when doing style transfer to update the input image or when computing the gradients for the second linear layer.

The backward method then accumulates the input gradients into the $\frac{\partial J}{\partial \text{output}}$ weights gradients of the previous layers using the `accumulate_grad` method of each layer, to prepare for backpropagation to continue into the earlier layers.

The `accumulate_grad` method simply adds the gradient it receives to this layer's `self.grad`. The `clear_grad` simple sets self.grad to zero. Both `accumulate_grad` and `clear_grad` should ensure that the grad does not change size.

The `step` method performs a single step of stochastic gradient descent, updating the weights of the current layer based on a `step_size` parameter it takes and the current gradients. (This method does not need to support more advanced learning algorithms like Adam or Adagrad.)

You may even wish to write tests to confirm backpropagation is computed successfully by your network. If you do this, ensure your examples also include enough examples of step and `accumulate_grad` that you are comfortable with your coverage of those as well. Each test should test only a single layer, with input layers feeding that layer and receiving its gradients. The method `np.testing.assert_allclose` may again be useful for testing that a pytorch tensor has the values you expect. Again see the examples in `test_sum.py`.

Backpropgation Unit Tests

The new methods should be unit tested with the numeric examples you developed last week.

Submission Requirements

Your submission will consist of:

- You must submit your derivations on paper:
  - Your complete derivations supporing your unit tests:

        - The equations and numeric examples used for your full-network forward test(s).
        - The forward numeric examples for your forward unit tests for each layer
        - Your backpropagation derivation equations, incorporating any instructor feedback.
        - The backward numeric examples supporing your backward unit tests for each layer.
    - You may resubmit the same marked-up pages you submitted earlier where your revisions are minor.
- Your entire library source-code as .py Python files with unit tests, submitted to
    - The provided network.py and layers.py should have the same names with which they were provided.
    - All test files should be named test_....py.
    - These should be submitted to Canvas. Submit individual files. Do **NOT** zip the files.

	File Name	Size	
	test_relu-1.py	1.01 KB	✓
	test_network-1.py	2.1 KB	✓
	test_layer-2-2.py	1.43 KB	✓
	test_layer-2.py	1.43 KB	✓
	test_linear-1.py	1.37 KB	✓
	test_sum-1.py	1.43 KB	✓
	test_regu...tion-1.py	1.18 KB	✓
	test_mseloss-1.py	1.17 KB	✓

	File Name	Size	
	network-1.py	3.23 KB	✓
	test_softmax-1.py	1.46 KB	✓
	layers-1.py	14.1 KB	✓
	test_input-1.py	1.07 KB	✓

ZOOM

```
from unittest import TestCase
import layers
import numpy as np
import torch
import unittest


class TestReLU(TestCase):
    """
    Please note: I (Dr. Yoder) may have assumed different p
than you use.
    TODO: Update these tests to work with YOUR definitions
variables.
    """
    def setUp(self):
        self.v = layers.Input((2,2))

        self.v.set(torch.tensor([[10, 6],[-10, 0]]).float()

        self.ReLU = layers.ReLU(self.v)

    def test_forward(self):
        self.ReLU.forward()
```

```
            np.testing.assert_allclose(self.ReLU.output.numpy()
                                       np.array([[10,6],[0,0]])

        def test_backward(self):
            self.ReLU.forward()
            self.ReLU.accumulate_grad(np.array([[-1,10],[100,-1
            self.ReLU.backward()

            np.testing.assert_allclose(self.v.grad.numpy(),
```

Previous (https://msoe.instructure.com/courses/13814/modules/items/564452)

Try Again

Next (https://msoe.instructure.com/courses/13814/modules/items/563780)