

Lab 6: Implementing forward, deriving all unit tests

Lab 6: Implementing forward, deriving all unit tests Ungraded, 50 Possible Points

Due: Tue Apr 25, 2023 8:30am4/25/2023

50 Possible Points

Attempt

Attempt 1

SUBMITTED on Apr 24,
2023 3:37pm4/24/2023

Attempt N/
1 Score: <>

Next Up: Review Feedback



Add
Comment

Unlimited Attempts Allowed

▼Details

In this lab, you will start to implement the code for your from-scratch deep neural network library. You will implement and write unit tests for forward propagation and develop numeric test cases on paper for the backpropagation unit tests.

In particular, you will implement:

- the `__init__` methods of all classes
- `forward()` for all concrete Layer classes that will need it.
- `forward()` for the network class.
- Unit tests of the methods above.

During this week's lab, you are implementing the forward pass. You should implement the `__init__` methods of all the classes, the Network's `set_input` and `forward` methods, and the layer class's `clear_grad`, `set`, `randomize`, and `forward` methods. Each layer should maintain the instance variables `output`, which represents the output of that layer.

The `accumulate_grad` and `backward` methods will be implemented in next week's lab.

You will write unit tests for your forward direction, with paper derivations to support the values expected in these layers.

You will also write a unit test for an entire network, using one of the numeric examples you developed in lab 3.

Finally, you will develop numeric backpropagation tests for the backward direction, which will support the backpropagation work you will do in next week's lab. You do not need to implement backpropagation or unit tests in Python this week.

Overview of the Gradient Tape Backpropagation Library

In this lab, you will develop the forward pass of the from-scratch gradient tape backpropagation library. This library will be capable of representing and training arbitrary networks, including the best networks available today such as Inception-v3 and transformer architectures.

Because this library is built on Pytorch, it will be optimized for running on the GPU. But it may not be optimized to reuse memory or communicate between multiple GPUs effectively.

The main thing you will learn from implementing this lab is how backpropagation works which is a central topic in training the world's best neural networks today.

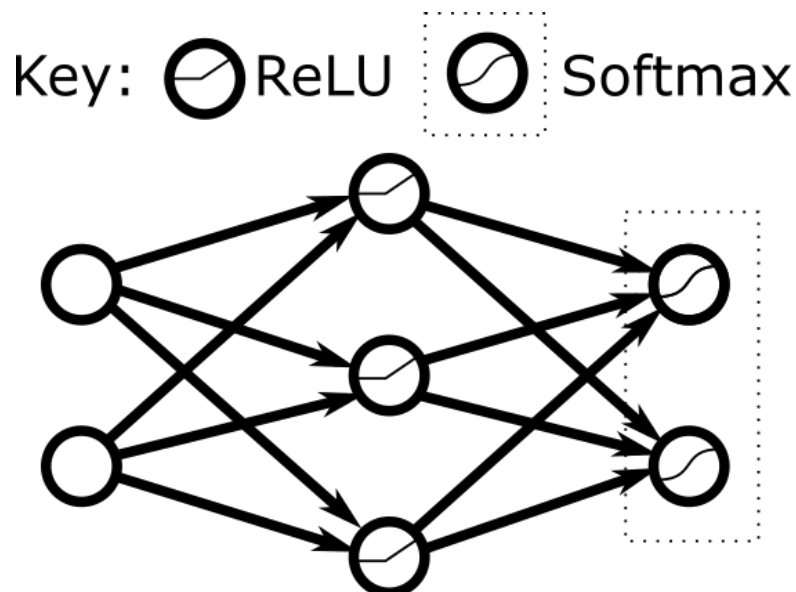
To use your library, the client will be responsible for adding each Layer object to the Network object in the order they will be executed. The Network will then be responsible for maintaining these layers in its gradient tape and evaluating the network in the forward and backward directions.

The client will also be responsible for tying each layer to the layers from which it draws its inputs. During forward propagation, each layer will derive its inputs from the layers that precede it. During backward propagation, each layer will send its input gradients for accumulation in the output gradients of the layers that precede it. So the client needs to give each layer the references it needs to communicate with the previous layers.

In Spring 2023, this was also discussed in Lab 2, and hopefully the motivation for writing Lab 2 in the way we suggested you do it is clearer now. Consider looking at your Lab2 code to see if it meets part of the requirements for this week's lab.

We will treat both inputs and learnable parameters as a single `Input` type. Doing this allows us to use the same code to update learnable parameters as we use to take backpropagation to a previous layer. All layers will collect gradients, so there is no need to distinguish between learnable parameters and inputs in this sense. But you will need some sort of `train=True` parameter to mark those inputs whose output values should train during backpropagation.

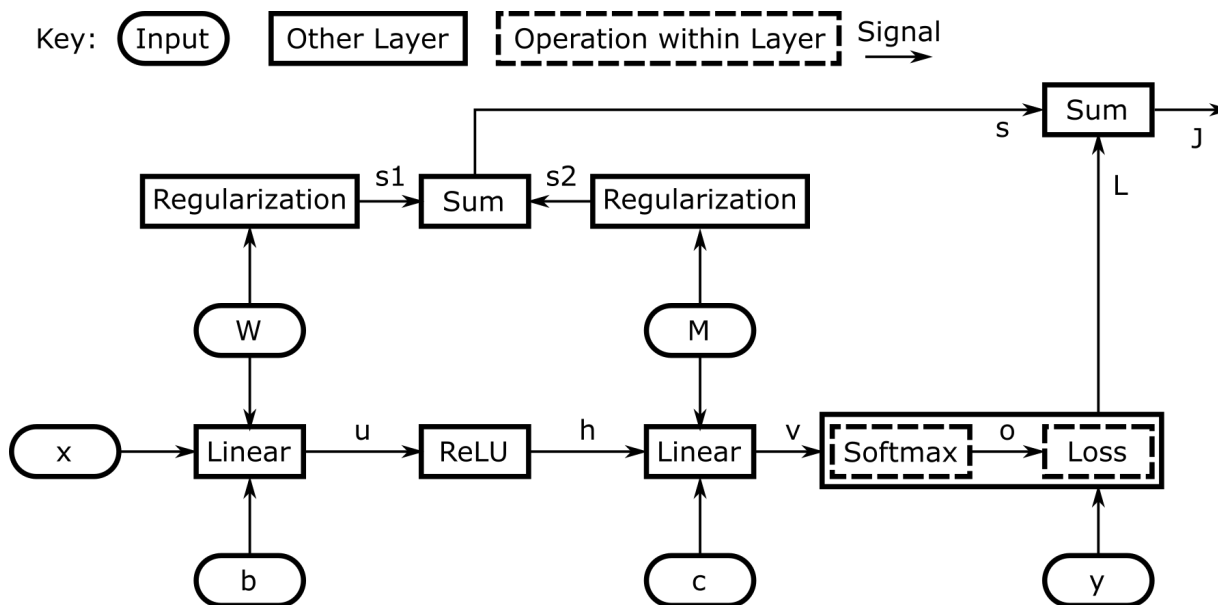
Your library should support a simple network with a single hidden layer:



Node diagram of a simple network

It will also support adding arbitrary layers, as the setup of the layers is left to the client.

Your library should allow the client to build this computation graph:



Computation graph

You are encouraged to adapt the design to meet your own needs. For example, you can split the Softmax and Loss into two layers as long as you hack the computation in the backward direction to use the simple gradients described in last week's lab.

But please do check with your instructor if you are consider major changes to the design. It is important that your design allow for parallel computations and accumuations from multiple sources, so, for example, embedding regularazation into the linear layers would not be approved.

Provided code

To get started, copy the provided code to your working directory:

```
cp -r /data/cs3450/mlp_from_scratch/week4/ .
```

This will create a week3 folder (week 4 of the from-scratch sequence, starting with week 1 being the autograd on linear week) containing:

- `client.py` – You will NOT use this file this week. You will use this file in future weeks to load the training data.
- `network.py` – This holds the Network class, which, as mentioned previously, is responsible for maintaining the gradient tape and organizing the forward and backward passes.
- `layer.py` – This holds the Layer class and all of its subclasses. As mentioned previously, each layer class should maintain references to the layers that come before it. During forward propagation, each layer should derive its own output based on the output of its input layers and during backpropagation, each layer will (in future weeks) determine its input gradients from its output gradients and current inputs, and send these backward for the previous layer to accumulate into its output.
- `test_sum.py` – This is a complete example of a unit test. You should make separate files for each of the layers you are testing with the unit tests for that layer. Having separate test files for each class makes isolating the problems for each class easier.

You can edit these python files through jupyter notebook. You are welcome to convert `client.py` to `client.ipynb` if you wish, which may make saving your output easier.

You only need to implement the methods introduced at the start of the lab.

Use good object-oriented design practices in your implementation. In Python, it is good practice to read instance variables directly (e.g. as in `print(my_layer.output)` or `self.output = -self.input.output`) but it is poor practice to change another classes's instance variables (e.g., as in `self.input.output = -self.output`). Reduce duplicate code by putting code in a place within the hierarchy where it can be written once and called from several places.

You are encouraged to exercise some design thinking while completing the code-base. Although method stubs are provided, the parameters for the methods are not. This is so that you have some flexibility in how you define the inputs and the outputs to your layers. You are even welcome to discuss with your instructor changes to where method stubs are located if you think this may improve your code base. A fair bit of structure is required to give you a sense for how the gradient tape library will function. But our goal is for you to be able to make the library your own.

Forward-pass unit tests

Your tests should cover the forward method for each layer.

Derive at least one numeric example for each layer for the forward direction. Write your derivations on paper.

See `test_sum.py` for an example.

To execute the unit test, include this code at the bottom of the unit-test file:

```
if __name__ == '__main__':
    unittest.main(argv=[''], exit=False)
```

Then execute the code with `python test_sum.py` or run the code as a Jupyter notebook.

For the softmax and cross-entropy layers, it can be helpful to use faked values such as e (the base of the natural logarithm) or $\log(2)$ so that the numbers are simpler to work on paper. For some of the matrix examples, working something out on paper can be necessary to provide a truly parallel test that doesn't depend on your PyTorch implementation.

For other layers, you can write the tests without writing anything on paper, and there is no need to include paper derivations for these layers.

These derivations must be submitted on paper – see more details below.

In addition to your unit tests, we strongly encourage you to write assertions for the shapes and perhaps types of the inputs to a layer with the layer methods as early in the process as these assertions can be tested. For example, the linear layer benefits greatly from confirming that its three inputs have the correct shapes. But other layers benefit as well. These assertions will help you to catch errors both in your client code and in your library code.

Full-network unit test

Write a file `test_network.py` which tests a complete network using the values you derived in Lab 2.

This test should include at least one hidden layer and at least two nodes in each layer, including the output. Your client should include a ReLU function as the activation in the hidden layer but does not need to include any activation function or loss on the output layer and does not need to include regularization. If your Lab 2 examples do not meet these requirements, please make a new example that does and include the derivations in your paper submissions for this lab.

Set the learnable parameters and inputs according to your example. Confirm that when your code runs, it produces the correct outputs for your numeric example.

Numeric examples for backpropagation unit tests

Prepare numeric examples to test the step method and the backward method for each layer. You should write at least one numeric example for each method implementation.

Since you will be writing these numeric examples, you need some idea of how the `backward`, `accumulate_grad`, and `step` will be implemented in next week's lab.

The multi-variate chain rule states that if $\mathbf{x} = f(\mathbf{a}, \mathbf{b}, \mathbf{c})$ and \mathbf{a} , \mathbf{b} , and \mathbf{c} are all functions of \mathbf{y} , then $\frac{\partial \mathbf{x}}{\partial \mathbf{y}} = \frac{\partial \mathbf{x}}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{y}} + \frac{\partial \mathbf{x}}{\partial \mathbf{b}} \frac{\partial \mathbf{b}}{\partial \mathbf{y}} + \frac{\partial \mathbf{x}}{\partial \mathbf{c}} \frac{\partial \mathbf{c}}{\partial \mathbf{y}}$.

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 J w.r.t. the input to this layer. For example, the Layer Linear class would receive $\frac{\partial J}{\partial \mathbf{x}}$ 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` simply 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`.

These derivations must be submitted on paper – see more details below.

In-lab milestone

Consider working on the backpropagation equations first and demoing these during lab.

You are encouraged to try to complete this lab during the lab period if possible.

Submission Requirements

For this lab, you must submit:

- On paper:
 - Forward pass: Numeric examples supporting those layers that need them
 - Backward pass: Numeric examples for testing the `step()` method and the `backward()` method for each layer
- To Canvas: (Submit each file individually)
 - All of your `.py` and `.ipynb` files:
 - `network.py`
 - `layers.py`












- test_network.py
- test_sum.py
- test_....py (any other test files you write)
- And any other python or .ipynb files you use that are needed by the files above.

The paper portion of the lab MUST be turned in on paper. You are encouraged to submit this during lab or at the start of class on Monday. You may also slip it under your instructor's office door before the deadline.

Thanks to Grant Fass for including the code for running unit tests within a Jupyter notebook.

Uploaded files

test_mseloss.py

File Name	Size
 test_mseloss.py	1.17 KB ✓
 test_input.py	1.07 KB ✓
 network.py	3.23 KB ✓
 test_sum.py	1.43 KB ✓
 test_network.py	2.1 KB ✓
 test_softmax.py	1.46 KB ✓
 test_relu.py	1.01 KB ✓
 test_linear.py	1.37 KB ✓
 test_layer.py	1.43 KB ✓
 test_regularization.py	1.18 KB ✓
 layers.py	14.1 KB ✓

Previous Module<Previous

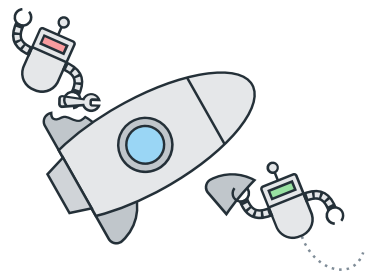
Try Again

Next ModuleNext>

(https://msoe.instructure.com/courses/13814/modules/items/551946)

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

Sorry, Something Broke



Help us improve by telling us what happened

Report Issue