[[Page Maintainers:|Where or how should I add documentation?]] @alband, @soulitzer

Scope

- Understand how backpropagation works in theory
- Understand how to derive backward formulas and how to add a backward formula to an operator
- Understand what a composite autograd operators is and when it is useful
- Know when to use gradcheck and custom autograd Functions
- (optional) Understand how the autograd graph gets built and executed

Introduction to backpropagation

Read through link.

What should I do when I implement a new function?

The answer varies a lot depending on the properties of the function you're writing. You can use the table below to see what is the best solution for you. The next sections in this document give more information on each solution. All terms are defined below the table.

Where Level	Tensor	Third party lib	Raw data pointer access
Aten native functions	Composite	derivatives.yaml	derivatives.yaml
In pytorch/pytorch outside of aten	Composite	Custom Function	Custom Function
Outside of pytorch/pytorch	Composite	Custom Function	Custom Function

Definition of the Levels: - "Tensor" means that your function is always working with PyTorch Tensors and using differentiable ops. - "Third party lib" means that your function uses a third party lib that doesn't work with Tensors directly (numpy, CUDA, etc). - "Raw data pointer access" means that your function extracts the data_ptr from the Tensor and work with that directly (for example our c++ implementations of many functions).

Definition of the Wheres: - "Aten native function" means functions that are defined in `native_functions.yaml`. - "In pytorch/pytorch outside of aten" means functions that are in pytorch/pytorch but not in aten (both in python and c++). - "Outside of pytorch/pytorch" means functions implemented outside of pytorch/pytorch.

Defintion of solutions: - “Composite” means that this new function is not an elementary op from the point of view of autograd. And autograd will track all the other functions you’re using inside. - “derivatives.yaml” means that you should implement your derivatives using `tools/autograd/derivatives.yaml`. - “Custom Function” means that you should use custom autograd Function to wrap your function.

What is a Composite function and how to write one?

We use the name composite here is used to refer to functions that are not “elementary” from the point of view of the autograd. This means that the autograd will ignore it and simply look at the functions that are called by this function and track these. A function can only be composite if it is implemented with differentiable functions.

Every function you write using pytorch operators (in python or c++) is composite. So there is nothing special you need to do.

Note that if you are working with native_functions.yaml, you need to use the CompositeImplicit key (which is the default if no dispatch at all is specified).

An example of such operator is the `torch.narrow()` function for example. You can see it defined in native functions here. Other examples include all the backward formulas that live in FunctionsManual.cpp or python functions such as the lp-pooling implementation here.

Given an operator, how do I derive a backward formula for it?

If you cannot use a composite function based on the table above, you will need to write the backward formula for your function by hand either in derivatives.yaml or in a custom Function. So the first step is to write down on paper what this formula is.

- How to derive a simple formula: torch.sin link.
- How to derive a more advanced formula: torch.mm link.

Given a new operator, how do I write a new backward formula? (using derivatives.yaml)

Implementing the backward using derivatives.yaml is the simplest. Add a new entry in `tools/autograd/derivatives.yaml` for your function. The name should match the signature you added to native_functions.yaml. Then you should add one entry for each input for which you implement the backward formula. The codegen will then take care of everything for you!

You can find more details in the documentation at the top of the derivatives.yaml file and browse that file for all the functions that are already implemented with

this method such as acos.

What are custom autograd Functions?

Custom autograd functions are the way to extend autograd outside of core. In particular, you will need to implement both the forward and backward functions that will be used to evaluate and compute the gradient for your function.

See details in the doc for how to implement such a Function link.

We use this feature in core both in python (to implement for example complex functions like lobpcg) and in c++ (to implement some low level interaction between the distributed RPC framework and autograd here). This is also used extensively outside of core to implement new functions that need to interact with autograd, for example in torchvision here.

How do I test an autograd formula?

Now that you have your function implemented and supporting autograd, it is time to check if the computed gradients are correct. We provide a builin tool for that called `autograd.gradcheck`. See here for a quick intro (toy implementation). This can be used to compare the gradient you implemented with a finite difference approximation.

This tool is used extensively in our automated test system (in particular OpInfo-based tests). But is also explicitly called in some cases such as in linalg here. Full details on how gradcheck work can be found in this note.

Try out the Autograd Onboarding Lab

https://github.com/pytorch/pytorch/wiki/Autograd-Onboarding-Lab

Autograd gotchas

There are a lot of small details in the autograd. Here are a few of them that are important for the lab above.

- Tensor full of zeros, `None` in python and undefined Tensors in c++ all mean the same thing for gradients. This means that your backward function need to properly handle potential None/undefined Tensors and behave as-if they were Tensors full of zeros. Similarly, you backward can return None/undefined Tensors instead of a Tensor full of zeros if needed.
- Don't forget to use `ctx.set_materialize_grads()` described in the extending doc on your custom Function to prevent zero Tensors from being materialized.
- The dtype that the backward functions support might be different than the ones that the forward supports for some already defined functions.

OpInfo provides many options to specify dtypes: `dtypes`, `dtypesIfCUDA`, `backward_dtypes` and `backward_dtypesIfCUDA`.