Autograd mechanics

This note will present an overview of how autograd works and records the operations. It's not strictly necessary to understand all this, but we recommend getting familiar with it, as it will help you write more efficient, cleaner programs, and can aid you in debugging.

How autograd encodes the history

Autograd is reverse automatic differentiation system. Conceptually, autograd records a graph recording all of the operations that created the data as you execute operations, giving you a directed acyclic graph whose leaves are the input tensors and roots are the output tensors. By tracing this graph from roots to leaves, you can automatically compute the gradients using the chain rule.

Internally, autograd represents this graph as a graph of :class:`Function` objects (really expressions), which can be :meth:`~torch.autograd.Function.apply` ed to compute the result of evaluating the graph. When computing the forwards pass, autograd simultaneously performs the requested computations and builds up a graph representing the function that computes the gradient (the `.grad_fn` attribute of each :class:`torch.Tensor` is an entry point into this graph). When the forwards pass is completed, we evaluate this graph in the backwards pass to compute the gradients.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 23); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 23); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 23); *backlink*

Unknown interpreted text role "class".

An important thing to note is that the graph is recreated from scratch at every iteration, and this is exactly what allows for using arbitrary Python control flow statements, that can change the overall shape and size of the graph at every iteration. You don't have to encode all possible paths before you launch the training - what you run is what you differentiate.

Saved tensors

Some operations need intermediary results to be saved during the forward pass in order to execute the backward pass. For example, the function $x \mapsto x^2$ saves the input x to compute the gradient.

When defining a custom Python :class:`~torch.autograd.Function`, you can use :func:`~torch.autograd.function._ContextMethodMixin.save_for_backward` to save tensors during the forward pass and :attr:`~torch.autograd.function.Function.saved_tensors` to retrieve them during the backward pass. See :doc:`/notes/extending` for more information.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 48); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 48); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 48); *backlink*

Unknown interpreted text role "attr".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 48); *backlink*

Unknown interpreted text role "doc".

For operations that PyTorch defines (e.g. :func:`torch.pow`), tensors are automatically saved as needed. You can explore (for educational or debugging purposes) which tensors are saved by a certain `grad_fn` by looking for its attributes starting with the prefix `_saved`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 54); *backlink*

Unknown interpreted text role "func".

```
x = torch.randn(5, requires_grad=True)
y = x.pow(2)
print(x.equal(y.grad_fn._saved_self))  # True
print(x is y.grad_fn._saved_self)  # True
```

In the previous code, `y.grad_fn._saved_self` refers to the same Tensor object as x. But that may not always be the case. For instance:

```
x = torch.randn(5, requires_grad=True)
y = x.exp()
print(y.equal(y.grad_fn._saved_result))  # True
print(y is y.grad_fn._saved_result)  # False
```

Under the hood, to prevent reference cycles, PyTorch has *packed* the tensor upon saving and *unpacked* it into a different tensor for reading. Here, the tensor you get from accessing `y.grad_fn._saved_result` is a different tensor object than `y` (but they still share the same storage).

Whether a tensor will be packed into a different tensor object depends on whether it is an output of its own *grad_fn*, which is an implementation detail subject to change and that users should not rely on.

You can control how PyTorch does packing / unpacking with :ref`saved-tensors-hooks-doc`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 87);

backlink

Unknown interpreted text role "ref".

Locally disabling gradient computation

There are several mechanisms available from Python to locally disable gradient computation:

To disable gradients across entire blocks of code, there are context managers like no-grad mode and inference mode. For more fine-grained exclusion of subgraphs from gradient computation, there is setting the `requires_grad` field of a tensor.

Below, in addition to discussing the mechanisms above, we also describe evaluation mode (:meth:`nn.Module.eval()`), a method that is not actually used to disable gradient computation but, because of its name, is often mixed up with the three.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst`, **line 103**); *backlink*

Unknown interpreted text role "meth".

Setting `requires_grad`

:attr:`requires_grad` is a flag, defaulting to false *unless wrapped in a* `nn.Parameter`, that allows for fine-grained exclusion of subgraphs from gradient computation. It takes effect in both the forward and backward passes:

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst`, **line 110**); *backlink*

Unknown interpreted text role "attr".

During the forward pass, an operation is only recorded in the backward graph if at least one of its input tensors require grad. During the backward pass (`.backward()`), only leaf tensors with `requires_grad=True` will have gradients accumulated into their `.grad` fields.

It is important to note that even though every tensor has this flag, *setting* it only makes sense for leaf tensors (tensors that do not have a `grad_fn`, e.g., a `nn.Module`'s parameters). Non-leaf tensors (tensors that do have `grad_fn`) are tensors that have a backward graph associated with them. Thus their gradients will be needed as an intermediary result to compute the gradient for a leaf tensor that requires grad. From this definition, it is clear that all non-leaf tensors will automatically have `require_grad=True`.

Setting `requires_grad` should be the main way you control which parts of the model are part of the gradient computation, for example, if you need to freeze parts of your pretrained model during model fine-tuning.

To freeze parts of your model, simply apply `.requires_grad_(False)` to the parameters that you don't want updated. And as described above, since computations that use these parameters as inputs would not be recorded in the forward pass, they won't have their `.grad` fields updated in the backward pass because they won't be part of the backward graph in the first place, as desired.

Because this is such a common pattern, `requires_grad` can also be set at the module level with :meth:`nn.Module.requires_grad_()`. When applied to a module, `.requires_grad_()` takes effect on all of the module's parameters (which have `requires_grad=True` by default).

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst`, **line 141**); *backlink*

Unknown interpreted text role "meth".

Grad Modes

Apart from setting `requires_grad` there are also three possible modes enableable from Python that can affect how computations in PyTorch are processed by autograd internally: default mode (grad mode), no-grad mode, and inference mode, all of which can be togglable via context managers and decorators.

Default Mode (Grad Mode)

The "default mode" is actually the mode we are implicitly in when no other modes like no-grad and inference mode are enabled. To be contrasted with "no-grad mode" the default mode is also sometimes called "grad mode".

The most important thing to know about the default mode is that it is the only mode in which `requires_grad` takes effect. `requires_grad` is always overridden to be `False` in both the two other modes.

No-grad Mode

Computations in no-grad mode behave as if none of the inputs require grad. In other words, computations in no-grad mode are never recorded in the backward graph even if there are inputs that have `require_grad=True`.

Enable no-grad mode when you need to perform operations that should not be recorded by autograd, but you'd still like to use the outputs of these computations in grad mode later. This context manager makes it convenient to disable gradients for a block of code or function without having to temporarily set tensors to have `requires_grad=False`, and then back to `True`.

For example, no-grad mode might be useful when writing an optimizer: when performing the training update you'd like to update parameters in-place without the update being recorded by autograd. You also intend to use the updated parameters for computations in grad mode in the next forward pass.

The implementations in :ref:`nn-init-doc` also rely on no-grad mode when initializing the parameters as to avoid autograd tracking when updating the intialized parameters in-place.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst`, **line 186**); *backlink*

Unknown interpreted text role "ref".

Inference Mode

Inference mode is the extreme version of no-grad mode. Just like in no-grad mode, computations in inference mode are not recorded in the backward graph, but enabling inference mode will allow PyTorch to speed up your model even more. This better runtime comes with a drawback: tensors created in inference mode will not be able to be used in computations to be recorded by autograd after exiting inference mode.

Enable inference mode when you are performing computations that don't need to be recorded in the backward graph, AND you don't plan on using the tensors created in inference mode in any computation that is to be recorded by autograd later.

It is recommended that you try out inference mode in the parts of your code that do not require autograd tracking (e.g., data processing and model evaluation). If it works out of the box for your use case it's a free performance win. If you run into errors after enabling inference mode, check that you are not using tensors created in inference mode in computations that are recorded by autograd after exiting inference mode. If you cannot avoid such use in your case, you can always switch back to no-grad mode.

For details on inference mode please see Inference Mode.

For implementation details of inference mode see RFC-0011-InferenceMode.

Evaluation Mode (`nn.Module.eval()`)

Evaluation mode is not actually a mechanism to locally disable gradient computation. It is included here anyway because it is

sometimes confused to be such a mechanism.

Functionally, `module.eval()` (or equivalently `module.train(False)`) are completely orthogonal to no-grad mode and inference mode. How `model.eval()` affects your model depends entirely on the specific modules used in your model and whether they define any training-mode specific behavior.

You are responsible for calling `model.eval()` and `model.train()` if your model relies on modules such as :class:`torch.nn.Dropout` and :class:`torch.nn.BatchNorm2d` that may behave differently depending on training mode, for example, to avoid updating your BatchNorm running statistics on validation data.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 230); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 230); *backlink*

Unknown interpreted text role "class".

It is recommended that you always use `model.train()` when training and `model.eval()` when evaluating your model (validation/testing) even if you aren't sure your model has training-mode specific behavior, because a module you are using might be updated to behave differently in training and eval modes.

In-place operations with autograd

Supporting in-place operations in autograd is a hard matter, and we discourage their use in most cases. Autograd's aggressive buffer freeing and reuse makes it very efficient and there are very few occasions when in-place operations actually lower memory usage by any significant amount. Unless you're operating under heavy memory pressure, you might never need to use them.

There are two main reasons that limit the applicability of in-place operations:

1. In-place operations can potentially overwrite values required to compute gradients.
2. Every in-place operation actually requires the implementation to rewrite the computational graph. Out-of-place versions simply allocate new objects and keep references to the old graph, while in-place operations, require changing the creator of all inputs to the :class:`Function` representing this operation. This can be tricky, especially if there are many Tensors that reference the same storage (e.g. created by indexing or transposing), and in-place functions will actually raise an error if the storage of modified inputs is referenced by any other :class:`Tensor`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 256); *backlink*

Unknown interpreted text role "class".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 256); *backlink*

Unknown interpreted text role "class".

In-place correctness checks

Every tensor keeps a version counter, that is incremented every time it is marked dirty in any operation. When a Function saves any tensors for backward, a version counter of their containing Tensor is saved as well. Once you access `self.saved_tensors` it is checked, and if it is greater than the saved value an error is raised. This ensures that if you're using in-place functions and not seeing any errors, you can be sure that the computed gradients are correct.

Multithreaded Autograd

The autograd engine is responsible for running all the backward operations necessary to compute the backward pass. This section will describe all the details that can help you make the best use of it in a multithreaded environment. (This is relevant only for PyTorch 1.6+ as the behavior in previous version was different.)

User could train their model with multithreading code (e.g. Hogwild training), and does not block on the concurrent backward computations, example code could be:

```
# Define a train function to be used in different threads
def train_fn():
    x = torch.ones(5, 5, requires_grad=True)
    # forward
    y = (x + 3) * (x + 4) * 0.5
    # backward
    y.sum().backward()
    # potential optimizer update


# User write their own threading code to drive the train_fn
threads = []
for _ in range(10):
    p = threading.Thread(target=train_fn, args=())
    p.start()
    threads.append(p)

for p in threads:
    p.join()
```

Note that some behaviors that user should be aware of:

Concurrency on CPU

When you run `backward()` or `grad()` via python or C++ API in multiple threads on CPU, you are expecting to see extra concurrency instead of serializing all the backward calls in a specific order during execution (behavior before PyTorch 1.6).

Non-determinism

If you are calling `backward()` on multiple thread concurrently but with shared inputs (i.e. Hogwild CPU training). Since parameters are automatically shared across threads, gradient accumulation might become non-deterministic on backward calls across threads, because two backward calls might access and try to accumulate the same `.grad` attribute. This is technically not safe, and it might result in racing condition and the result might be invalid to use.

But this is expected pattern if you are using the multithreading approach to drive the whole training process but using shared parameters, user who use multithreading should have the threading model in mind and should expect this to happen. User could use the functional API :func:`torch.autograd.grad` to calculate the gradients instead of `backward()` to avoid non-determinism.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 330); *backlink*

Unknown interpreted text role "func".

Graph retaining

If part of the autograd graph is shared between threads, i.e. run first part of forward single thread, then run second part in multiple threads, then the first part of graph is shared. In this case different threads execute `grad()` or `backward()` on the same graph might have issue of destroying the graph on the fly of one thread, and the other thread will crash in this case. Autograd will error out to the user similar to what call `backward()` twice with out `retain_graph=True`, and let the user know they should use `retain_graph=True`.

Thread Safety on Autograd Node

Since Autograd allows the caller thread to drive its backward execution for potential parallelism, it's important that we ensure thread safety on CPU with parallel backwards that share part/whole of the GraphTask.

Custom Python `autograd.Function` is automatically thread safe because of GIL. For built-in C++ Autograd Nodes (e.g. AccumulateGrad, CopySlices) and custom `autograd::Function`s, the Autograd Engine uses thread mutex locking to ensure thread safety on autograd Nodes that might have state write/read.

No thread safety on C++ hooks

Autograd relies on the user to write thread safe C++ hooks. If you want the hook to be correctly applied in multithreading environment, you will need to write proper thread locking code to ensure the hooks are thread safe.

Autograd for Complex Numbers

The short version:

- When you use PyTorch to differentiate any function $f(z)$ with complex domain and/or codomain, the gradients are computed under the assumption that the function is a part of a larger real-valued loss function $g(input) = L$. The gradient computed is $\frac{\partial L}{\partial z^*}$ (note the conjugation of z), the negative of which is precisely the direction of steepest descent used in Gradient Descent algorithm. Thus, all the existing optimizers work out of the box with complex parameters.
- This convention matches TensorFlow's convention for complex differentiation, but is different from JAX (which computes $\frac{\partial L}{\partial z}$).
- If you have a real-to-real function which internally uses complex operations, the convention here doesn't matter: you will always get the same result that you would have gotten if it had been implemented with only real operations.

If you are curious about the mathematical details, or want to know how to define complex derivatives in PyTorch, read on.

What are complex derivatives?

The mathematical definition of complex-differentiability takes the limit definition of a derivative and generalizes it to operate on complex numbers. Consider a function $f : \mathbb{C} \to \mathbb{C}$,

$$`f(z = x + yj) = u(x, y) + v(x, y)j`$$

where u and v are two variable real valued functions.

Using the derivative definition, we can write:

$$f'(z) = \lim_{h \to 0, h \in C} \frac{f(z + h) - f(z)}{h}$$

In order for this limit to exist, not only must u and v must be real differentiable, but f must also satisfy the Cauchy-Riemann equations. In other words: the limit computed for real and imaginary steps (h) must be equal. This is a more restrictive condition.

The complex differentiable functions are commonly known as holomorphic functions. They are well behaved, have all the nice properties that you've seen from real differentiable functions, but are practically of no use in the optimization world. For optimization problems, only real valued objective functions are used in the research community since complex numbers are not part of any ordered field and so having complex valued loss does not make much sense.

It also turns out that no interesting real-valued objective fulfill the Cauchy-Riemann equations. So the theory with homomorphic function cannot be used for optimization and most people therefore use the Wirtinger calculus.

Wirtinger Calculus comes in picture ...

So, we have this great theory of complex differentiability and holomorphic functions, and we can't use any of it at all, because many of the commonly used functions are not holomorphic. What's a poor mathematician to do? Well, Wirtinger observed that even if $f(z)$ isn't holomorphic, one could rewrite it as a two variable function $f(z, z^*)$ which is always holomorphic. This is because real and imaginary of the components of z can be expressed in terms of z and z^* as:

$$Re(z) = \frac{z + z^*}{2}$$

$$Im(z) = \frac{z - z^*}{2j}$$

Wirtinger calculus suggests to study $f(z, z^*)$ instead, which is guaranteed to be holomorphic if f was real differentiable (another way to think of it is as a change of coordinate system, from $f(x, y)$ to $f(z, z^*)$.) This function has partial derivatives $\frac{\partial}{\partial z}$ and $\frac{\partial}{\partial z^*}$. We can use the chain rule to establish a relationship between these partial derivatives and the partial derivatives w.r.t., the real and imaginary components of z.

$$\frac{\partial}{\partial x} = \frac{\partial z}{\partial x} * \frac{\partial}{\partial z} + \frac{\partial z^*}{\partial x} * \frac{\partial}{\partial z^*}$$

$$= \frac{\partial}{\partial z} + \frac{\partial}{\partial z^*}$$

$$\frac{\partial}{\partial y} = \frac{\partial z}{\partial y} * \frac{\partial}{\partial z} + \frac{\partial z^*}{\partial y} * \frac{\partial}{\partial z^*}$$

$$= 1j * (\frac{\partial}{\partial z} - \frac{\partial}{\partial z^*})$$

From the above equations, we get:

$$\frac{\partial}{\partial z} = 1/2 * (\frac{\partial}{\partial x} - 1j * \frac{\partial}{\partial y})$$

$$\frac{\partial}{\partial z^*} = 1/2 * (\frac{\partial}{\partial x} + 1j * \frac{\partial}{\partial y})$$

which is the classic definition of Wirtinger calculus that you would find on Wikipedia.

There are a lot of beautiful consequences of this change.

- For one, the Cauchy-Riemann equations translate into simply saying that $\frac{\partial f}{\partial z^*} = 0$ (that is to say, the function f can be written entirely in terms of z, without making reference to z^*).
- Another important (and somewhat counterintuitive) result, as we'll see later, is that when we do optimization on a real-valued loss, the step we should take while making variable update is given by $\frac{\partial Loss}{\partial z^*}$ (not $\frac{\partial Loss}{\partial z}$).

For more reading, check out: https://arxiv.org/pdf/0906.4835.pdf

How is Wirtinger Calculus useful in optimization?

Researchers in audio and other fields, more commonly, use gradient descent to optimize real valued loss functions with complex variables. Typically, these people treat the real and imaginary values as separate channels that can be updated. For a step size $\alpha/2$ and loss L, we can write the following equations in $\mathbb{R}^2$:

$$x_{n+1} = x_n - (\alpha/2)*\frac{\partial L}{\partial x}$$

$$y_{n+1} = y_n - (\alpha/2)*\frac{\partial L}{\partial y}$$

How do these equations translate into complex space $\mathbb{C}$?

$$\begin{aligned} z_{n+1} &= x_n - (\alpha/2)*\frac{\partial L}{\partial x} + 1j*(y_n - (\alpha/2)*\frac{\partial L}{\partial y}) \\ &= z_n - \alpha*1/2*(\frac{\partial L}{\partial x} + j\frac{\partial L}{\partial y}) \\ &= z_n - \alpha*\frac{\partial L}{\partial z^*} \end{aligned}$$

Something very interesting has happened: Wirtinger calculus tells us that we can simplify the complex variable update formula above to only refer to the conjugate Wirtinger derivative $\frac{\partial L}{\partial z^*}$, giving us exactly the step we take in optimization.

Because the conjugate Wirtinger derivative gives us exactly the correct step for a real valued loss function, PyTorch gives you this derivative when you differentiate a function with a real valued loss.

How does PyTorch compute the conjugate Wirtinger derivative?

Typically, our derivative formulas take in *grad_output* as an input, representing the incoming Vector-Jacobian product that we've already computed, aka, $\frac{\partial L}{\partial s^*}$, where L is the loss of the entire computation (producing a real loss) and s is the output of our function. The goal here is to compute $\frac{\partial L}{\partial z^*}$, where z is the input of the function. It turns out that in the case of real loss, we can get away with *only* calculating $\frac{\partial L}{\partial z^*}$, even though the chain rule implies that we also need to have access to $\frac{\partial L}{\partial z^*}$. If you want to skip this derivation, look at the last equation in this section and then skip to the next section.

Let's continue working with $f : \mathbb{C} \to \mathbb{C}$ defined as $f(z) = f(x + yj) = u(x, y) + v(x, y)j$. As discussed above, autograd's gradient convention is centered around optimization for real valued loss functions, so let's assume f is a part of larger real valued loss function g. Using chain rule, we can write:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 534)

Error in "math" directive: unknown option: "label".

```
.. math::
    \frac{\partial L}{\partial z^*} = \frac{\partial L}{\partial u} * \frac{\partial u}{\partial z^*} + \frac{\partial L}{\part
    :label: [1]
```

Now using Wirtinger derivative definition, we can write:

$$\frac{\partial L}{\partial s} = 1/2*(\frac{\partial L}{\partial u} - \frac{\partial L}{\partial v}j)$$

$$\frac{\partial L}{\partial s^*} = 1/2*(\frac{\partial L}{\partial u} + \frac{\partial L}{\partial v}j)$$

It should be noted here that since u and v are real functions, and L is real by our assumption that f is a part of a real valued function, we have:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 550)

Error in "math" directive: unknown option: "label".

```
.. math::
    (\frac{\partial L}{\partial s})^* = \frac{\partial L}{\partial s^*}
    :label: [2]
```

i.e., $\frac{\partial L}{\partial s}$ equals to $grad\_output^*$.

Solving the above equations for $\frac{\partial L}{\partial u}$ and $\frac{\partial L}{\partial v}$, we get:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 558)

Error in "math" directive: unknown option: "label".

```
.. math::
    \begin{aligned}
        \frac{\partial L}{\partial u} = \frac{\partial L}{\partial s} + \frac{\partial L}{\partial s^*} \\
        \frac{\partial L}{\partial v} = -1j * (\frac{\partial L}{\partial s} - \frac{\partial L}{\partial s^*})
    \end{aligned}
    :label: [3]
```

Substituting :eq:`[3]` in :eq:`[1]`, we get:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 565); *backlink*

Unknown interpreted text role "eq".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 565); *backlink*

Unknown interpreted text role "eq".

$$\frac{\partial L}{\partial z^*} = (\frac{\partial L}{\partial s} + \frac{\partial L}{\partial s^*}) * \frac{\partial u}{\partial z^*} - 1j * (\frac{\partial L}{\partial s} - \frac{\partial L}{\partial s^*}) * \frac{\partial v}{\partial z^*}$$

$$= \frac{\partial L}{\partial s} * (\frac{\partial u}{\partial z^*} + \frac{\partial v}{\partial z^*} j) + \frac{\partial L}{\partial s^*} * (\frac{\partial u}{\partial z^*} - \frac{\partial v}{\partial z^*} j)$$

$$= \frac{\partial L}{\partial s^*} * \frac{\partial (u + vj)}{\partial z^*} + \frac{\partial L}{\partial s} * \frac{\partial (u + vj)^*}{\partial z^*}$$

$$= \frac{\partial L}{\partial s} * \frac{\partial s}{\partial z^*} + \frac{\partial L}{\partial s^*} * \frac{\partial s^*}{\partial z^*}$$

Using :eq:`[2]`, we get:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 575); backlink

Unknown interpreted text role "eq".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 577)

Error in "math" directive: unknown option: "label".

```
.. math::
    \begin{aligned}
        \frac{\partial L}{\partial z^*} &= (\frac{\partial L}{\partial s^*})^* * \frac{\partial s}{\partial z^*} + \frac{\part
                                        &= \boxed{ (grad\_output)^* * \frac{\partial s}{\partial z^*} + grad\_output * {(\frac
    \end{aligned}
    :label: [4]
```

This last equation is the important one for writing your own gradients, as it decomposes our derivative formula into a simpler one that is easy to compute by hand.

How can I write my own derivative formula for a complex function?

The above boxed equation gives us the general formula for all derivatives on complex functions. However, we still need to compute $\frac{\partial s}{\partial z}$ and $\frac{\partial s}{\partial z^*}$. There are two ways you could do this:

- The first way is to just use the definition of Wirtinger derivatives directly and calculate $\frac{\partial s}{\partial z}$ and $\frac{\partial s}{\partial z^*}$ by using $\frac{\partial s}{\partial x}$ and $\frac{\partial s}{\partial y}$ (which you can compute in the normal way).
- The second way is to use the change of variables trick and rewrite $f(z)$ as a two variable function $f(z, z^*)$, and compute the conjugate Wirtinger derivatives by treating z and z^* as independent variables. This is often easier; for example, if the function in question is holomorphic, only z will be used (and $\frac{\partial s}{\partial z^*}$ will be zero).

Let's consider the function $f(z = x + yj) = c*z = c*(x + yj)$ as an example, where $c \in \mathbb{R}$.

Using the first way to compute the Wirtinger derivatives, we have.

$$\frac{\partial s}{\partial z} = 1/2*(\frac{\partial s}{\partial x} - \frac{\partial s}{\partial y} j)$$

$$= 1/2*(c - (c*1j)*1j)$$

$$= c$$

$$\frac{\partial s}{\partial z^*} = 1/2*(\frac{\partial s}{\partial x} + \frac{\partial s}{\partial y} j)$$

$$= 1/2*(c + (c*1j)*1j)$$

$$= 0$$

Using :eq:`[4]`, and *grad_output = 1.0* (which is the default grad output value used when :func:`backward` is called on a scalar output in PyTorch), we get:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 618); backlink

Unknown interpreted text role "eq".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 618); backlink

Unknown interpreted text role "func".

$$\frac{\partial L}{\partial z^*} = 1*0 + 1*c = c$$

Using the second way to compute Wirtinger derivatives, we directly get:

$$\frac{\partial s}{\partial z} = \frac{\partial (c*z)}{\partial z}$$

$$= c$$

$$\frac{\partial s}{\partial z^*} = \frac{\partial (c*z)}{\partial z^*}$$

$$= 0$$

And using :eq:`[4]` again, we get $\frac{\partial L}{\partial z^*} = c$. As you can see, the second way involves lesser calculations, and comes in more handy for faster calculations.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 633); backlink

Unknown interpreted text role "eq".

What about cross-domain functions?

Some functions map from complex inputs to real outputs, or vice versa. These functions form a special case of :eq:`[4]`, which we can derive using the chain rule:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 639); *backlink*
>
> Unknown interpreted text role "eq".

- For $f: \mathbb{C} \to \mathbb{R}$, we get:

$$\frac{\partial L}{\partial z^*} = 2 * grad\_output * \frac{\partial s}{\partial z^*}$$

- For $f: \mathbb{R} \to \mathbb{C}$, we get:

$$\frac{\partial L}{\partial z^*} = 2 * Re(grad\_out^* * \frac{\partial s}{\partial z^*})$$

Hooks for saved tensors

You can control :ref:`how saved tensors are packed / unpacked <saved-tensors-doc>` by defining a pair of `pack_hook` / `unpack_hook` hooks. The `pack_hook` function should take a tensor as its single argument but can return any python object (e.g. another tensor, a tuple, or even a string containing a filename). The `unpack_hook` function takes as its single argument the output of `pack_hook` and should return a tensor to be used in the backward pass. The tensor returned by `unpack_hook` only needs to have the same content as the tensor passed as input to `pack_hook`. In particular, any autograd-related metadata can be ignored as they will be overwritten during unpacking.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 658); *backlink*
>
> Unknown interpreted text role "ref".

An example of such pair is:

```
class SelfDeletingTempFile():
    def __init__(self):
        self.name = os.path.join(tmp_dir, str(uuid.uuid4()))

    def __del__(self):
        os.remove(self.name)

def pack_hook(tensor):
    temp_file = SelfDeletingTempFile()
    torch.save(tensor, temp_file.name)
    return temp_file

def unpack_hook(temp_file):
    return torch.load(temp_file.name)
```

Notice that the `unpack_hook` should not delete the temporary file because it might be called multiple times: the temporary file should be alive for as long as the returned *SelfDeletingTempFile* object is alive. In the above example, we prevent leaking the temporary file by closing it when it is no longer needed (on deletion of the *SelfDeletingTempFile* object).

> **Note**
>
> We guarantee that `pack_hook` will only be called once but `unpack_hook` can be called as many times as the backward pass requires it and we expect it to return the same data each time.

> **Warning**
>
> Performing inplace operations on the input of any of the functions is forbidden as they may lead to unexpected side-effects. PyTorch will throw an error if the input to a pack hook is modified inplace but does not catch the case where the input to an unpack hook is modified inplace.

Registering hooks for a saved tensor

You can register a pair of hooks on a saved tensor by calling the :meth:`~torch.autograd.SavedTensor.register_hooks` method on a :class:`SavedTensor` object. Those objects are exposed as attributes of a `grad_fn` and start with the `_raw_saved_` prefix.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 711); *backlink*
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 711); *backlink*
>
> Unknown interpreted text role "class".

```
x = torch.randn(5, requires_grad=True)
y = x.pow(2)
y.grad_fn._raw_saved_self.register_hooks(pack_hook, unpack_hook)
```

The `pack_hook` method is called as soon as the pair is registered. The `unpack_hook` method is called each time the saved tensor needs to be accessed, either by means of `y.grad_fn._saved_self` or during the backward pass.

> **Warning**
>
> If you maintain a reference to a :class:`SavedTensor` after the saved tensors have been released (i.e. after backward has been called), calling its :meth:`~torch.autograd.SavedTensor.register_hooks` is forbidden. PyTorch will throw an error most of the time but it may fail to do so in some cases and undefined behavior may arise.
>
> > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 729); *backlink*
> >
> > Unknown interpreted text role "class".
>
> > **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 729); *backlink*

Unknown interpreted text role "meth".

Registering default hooks for saved tensors

Alternatively, you can use the context-manager :class:`~torch.autograd.graph.saved_tensors_hooks` to register a pair of hooks which will be applied to *all* saved tensors that are created in that context.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\(pytorch-master)(docs)(source)(notes)autograd.rst, line 738); *backlink*

Unknown interpreted text role "class".

Example:

```
# Only save on disk tensors that have size >= 1000
SAVE_ON_DISK_THRESHOLD = 1000

def pack_hook(x):
    if x.numel() < SAVE_ON_DISK_THRESHOLD:
        return x
    temp_file = SelfDeletingTempFile()
    torch.save(tensor, temp_file.name)
    return temp_file

def unpack_hook(tensor_or_sctf):
    if isinstance(tensor_or_sctf, torch.Tensor):
        return tensor_or_sctf
    return torch.load(tensor_or_sctf.name)

class Model(nn.Module):
    def forward(self, x):
        with torch.autograd.graph.saved_tensors_hooks(pack_hook, unpack_hook):
          # ... compute output
          output = x
        return output

model = Model()
net = nn.DataParallel(model)
```

The hooks defined with this context manager are thread-local. Hence, the following code will not produce the desired effects because the hooks do not go through *DataParallel*.

```
# Example what NOT to do

net = nn.DataParallel(model)
with torch.autograd.graph.saved_tensors_hooks(pack_hook, unpack_hook):
    output = net(input)
```

Note that using those hooks disables all the optimization in place to reduce Tensor object creation. For example:

```
with torch.autograd.graph.saved_tensors_hooks(lambda x: x, lambda x: x):
    x = torch.randn(5, requires_grad=True)
    y = x * x
```

Without the hooks, `x`, `y.grad_fn._saved_self` and `y.grad_fn._saved_other` all refer to the same tensor object. With the hooks, PyTorch will pack and unpack *x* into two new tensor objects that share the same storage with the original *x* (no copy performed).