Gradcheck mechanics

This note presents an overview of how the :meth:`~torch.autograd.gradcheck` and :meth:`~torch.autograd.gradgradcheck` functions work.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]gradcheck.rst`, line 6); ***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]gradcheck.rst`, line 6); ***backlink***

Unknown interpreted text role "meth".

It will cover both forward and backward mode AD for both real and complex-valued functions as well as higher-order derivatives. This note also covers both the default behavior of gradcheck as well as the case where `fast_mode=True` argument is passed (referred to as fast gradcheck below).

Notations and background information

Throughout this note, we will use the following convention:

1. x, y, a, b, v, u, ur and ui are real-valued vectors and z is a complex-valued vector that can be rewritten in terms of two real-valued vectors as $z = a + ib$.
2. N and M are two integers that we will use for the dimension of the input and output space respectively.
3. $f : \mathcal{R}^N \to \mathcal{R}^M$ is our basic real-to-real function such that $y = f(x)$.
4. $g : \mathcal{C}^N \to \mathcal{R}^M$ is our basic complex-to-real function such that $y = g(z)$.

For the simple real-to-real case, we write as J_f the Jacobian matrix associated with f of size $M \times N$. This matrix contains all the partial derivatives such that the entry at position (i, j) contains $\frac{\partial y_i}{\partial x_j}$. Backward mode AD is then computing, for a given vector v of size M, the quantity $v^T J_f$. Forward mode AD on the other hand is computing, for a given vector u of size N, the quantity $J_f u$.

For functions that contain complex values, the story is a lot more complex. We only provide the gist here and the full description can be found at :ref:`complex_autograd-doc`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]gradcheck.rst`, line 33); ***backlink***

Unknown interpreted text role "ref".

The constraints to satisfy complex differentiability (Cauchy-Riemann equations) are too restrictive for all real-valued loss functions, so we instead opted to use Wirtinger calculus. In a basic setting of Wirtinger calculus, the chain rule requires access to both the Wirtinger derivative (called W below) and the Conjugate Wirtinger derivative (called CW below). Both W and CW need to be propagated because in general, despite their name, one is not the complex conjugate of the other.

To avoid having to propagate both values, for backward mode AD, we always work under the assumption that the function whose derivative is being calculated is either a real-valued function or is part of a bigger real-valued function. This assumption means that all the intermediary gradients we compute during the backward pass are also associated with real-valued functions. In practice, this assumption is not restrictive when doing optimization as such problem require real-valued objectives (as there is no natural ordering of the complex numbers).

Under this assumption, using W and CW definitions, we can show that $W = CW^*$ (we use $*$ to denote complex conjugation here) and so only one of the two values actually need to be "backwarded through the graph" as the other one can easily be recovered. To simplify internal computations, PyTorch uses $2 * CW$ as the value it backwards and returns when the user asks for gradients. Similarly to the real case, when the output is actually in $\mathcal{R}^M$, backward mode AD does not compute $2 * CW$ but only $v^T (2 * CW)$ for a given vector $v \in \mathcal{R}^M$.

For forward mode AD, we use a similar logic, in this case, assuming that the function is part of a larger function whose input is in $\mathcal{R}$. Under this assumption, we can make a similar claim that every intermediary result corresponds to a function whose input is in $\mathcal{R}$ and in this case, using W and CW definitions, we can show that $W = CW$ for the intermediary functions. To make sure the forward and backward mode compute the same quantities in the elementary case of a one dimensional function, the forward mode also computes $2 * CW$. Similarly to the real case, when the input is actually in $\mathcal{R}^N$, forward mode AD does not compute $2 * CW$ but only $(2 * CW) u$ for a given vector $u \in \mathcal{R}^N$.

Default backward mode gradcheck behavior

Real-to-real functions

To test a function $f : \mathcal{R}^N \to \mathcal{R}^M, x \to y$, we reconstruct the full Jacobian matrix J_f of size $M \times N$ in two ways: analytically and numerically. The analytical version uses our backward mode AD while the numerical version uses finite difference. The two reconstructed Jacobian matrices are then compared elementwise for equality.

Default real input numerical evaluation

If we consider the elementary case of a one-dimensional function ($N = M = 1$), then we can use the basic finite difference formula from the wikipedia article. We use the "central difference" for better numerical properties:

$$\frac{\partial y}{\partial x} \approx \frac{f(x + eps) - f(x - eps)}{2 * eps}$$

This formula easily generalizes for multiple outputs (Mgt1) by having $\frac{\partial y}{\partial x}$ be a column vector of size $M \times 1$ like $f(x + eps)$. In that case, the above formula can be re-used as-is and approximates the full Jacobian matrix with only two evaluations of the user function (namely $f(x + eps)$ and $f(x - eps)$).

It is more computationally expensive to handle the case with multiple inputs (Ngt1). In this scenario, we loop over all the inputs one after the other and apply the eps perturbation for each element of x one after the other. This allows us to reconstruct the J_f matrix column by column.

Default real input analytical evaluation

For the analytical evaluation, we use the fact, as described above, that backward mode AD computes $v^T J_f$. For functions with a single output, we simply use $v = 1$ to recover the full Jacobian matrix with a single backward pass.

For functions with more than one output, we resort to a for-loop which iterates over the outputs where each v is a one-hot vector corresponding to each output one after the other. This allows to reconstruct the J_f matrix row by row.

Complex-to-real functions

To test a function $g : \mathcal{C}^N \to \mathcal{R}^M, z \to y$ with $z = a + ib$, we reconstruct the (complex-valued) matrix that contains $2*CW$.

Default complex input numerical evaluation

Consider the elementary case where $N = M = 1$ first. We know from (chapter 3 of) this research paper that:

$$CW := \frac{\partial y}{\partial z^*} = \frac{1}{2} * (\frac{\partial y}{\partial a} + i \frac{\partial y}{\partial b})$$

Note that $\frac{\partial y}{\partial a}$ and $\frac{\partial y}{\partial b}$, in the above equation, are $\mathcal{R} \to \mathcal{R}$ derivatives. To evaluate these numerically, we use the method described above for the real-to-real case. This allows us to compute the CW matrix and then multiply it by 2.

Note that the code, as of time of writing, computes this value in a slightly convoluted way:

```
# Code from https://github.com/pytorch/pytorch/blob/58eb23378f2a376565a66ac32c93a316c45b6131/torch/autograd/gradcheck.py#L99-L105
# Notation changes in this code block:
# s here is y above
# x, y here are a, b above

ds_dx = compute_gradient(eps)
ds_dy = compute_gradient(eps * 1j)
# conjugate wirtinger derivative
conj_w_d = 0.5 * (ds_dx + ds_dy * 1j)
# wirtinger derivative
w_d = 0.5 * (ds_dx - ds_dy * 1j)
d[d_idx] = grad_out.conjugate() * conj_w_d + grad_out * w_d.conj()

# Since grad_out is always 1, and W and CW are complex conjugate of each other, the last line ends up computing exactly `conj_w_d + w_d.
```

Default complex input analytical evaluation

Since backward mode AD computes exactly twice the CW derivative already, we simply use the same trick as for the real-to-real case here and reconstruct the matrix row by row when there are multiple real outputs.

Functions with complex outputs

In this case, the user-provided function does not follow the assumption from the autograd that the function we compute backward AD for is real-valued. This means that using autograd directly on this function is not well defined. To solve this, we will replace the test of the function $h : \mathcal{P}^N \to \mathcal{C}^M$ (where $\mathcal{P}$ can be either $\mathcal{R}$ or $\mathcal{C}$), with two functions: hr and hi such that:

$$hr(q) := real(f(q))$$
$$hi(q) := imag(f(q))$$

where $q \in \mathcal{P}$. We then do a basic gradcheck for both hr and hi using either the real-to-real or complex-to-real case described above, depending on $\mathcal{P}$.

Note that, the code, as of time of writing, does not create these functions explicitly but perform the chain rule with the *real* or *imag* functions manually by passing the grad_out arguments to the different functions. When grad_out = 1, then we are considering hr. When grad_out = $1j$, then we are considering hi.

Fast backward mode gradcheck

While the above formulation of gradcheck is great, both, to ensure correctness and debuggability, it is very slow because it reconstructs the full Jacobian matrices. This section presents a way to perform gradcheck in a faster way without affecting its correctness. The debuggability can be recovered by adding special logic when we detect an error. In that case, we can run the default version that reconstructs the full matrix to give full details to the user.

The high level strategy here is to find a scalar quantity that can be computed efficiently by both the numerical and analytical methods and that represents the full matrix computed by the slow gradcheck well enough to ensure that it will catch any discrepancy in the Jacobians.

Fast gradcheck for real-to-real functions

The scalar quantity that we want to compute here is $v^T J_f u$ for a given random vector $v \in \mathcal{R}^M$ and a random unit norm vector $u \in \mathcal{R}^N$.

For the numerical evaluation, we can efficiently compute

$$J_f u \approx \frac{f(x + u * eps) - f(x - u * eps)}{2 * eps}.$$

We then perform the dot product between this vector and v to get the scalar value of interest.

For the analytical version, we can use backward mode AD to compute $v^T J_f$ directly. We then perform the dot product with u to get the expected value.

Fast gradcheck for complex-to-real functions

Similar to the real-to-real case, we want to perform a reduction of the full matrix. But the $2*CW$ matrix is complex-valued and so in this case, we will compare to complex scalars.

Due to some constraints on what we can compute efficiently in the numerical case and to keep the number of numerical evaluations to a minimum, we compute the following (albeit surprising) scalar value:

$$s := 2 * v^T (real(CW) ur + i * imag(CW) ui)$$

where $v \in \mathcal{R}^M$, $ur \in \mathcal{R}^N$ and $ui \in \mathcal{R}^N$.

Fast complex input numerical evaluation

We first consider how to compute s with a numerical method. To do so, keeping in mind that we're considering $g : \mathcal{C}^N \to \mathcal{R}^M, z \to y$ with $z = a + ib$, and that $CW = \frac{1}{2} * (\frac{\partial y}{\partial a} + i \frac{\partial y}{\partial b})$, we rewrite it as follows:

$$
\begin{aligned}
s &= 2*v^T(real(CW)ur + i*imag(CW)ui) \\
&= 2*v^T(\frac{1}{2}*\frac{\partial y}{\partial a}ur + i*\frac{1}{2}*\frac{\partial y}{\partial b}ui) \\
&= v^T(\frac{\partial y}{\partial a}ur + i*\frac{\partial y}{\partial b}ui) \\
&= v^T((\frac{\partial y}{\partial a}ur) + i*(\frac{\partial y}{\partial b}ui))
\end{aligned}
$$

In this formula, we can see that $\frac{\partial y}{\partial a}ur$ and $\frac{\partial y}{\partial b}ui$ can be evaluated the same way as the fast version for the real-to-real case. Once these real-valued quantities have been computed, we can reconstruct the complex vector on the right side and do a dot product with the real-valued v vector.

Fast complex input analytical evaluation

For the analytical case, things are simpler and we rewrite the formula as:

$$
\begin{aligned}
s &= 2*v^T(real(CW)ur + i*imag(CW)ui) \\
&= v^T real(2*CW)ur + i*v^T imag(2*CW)ui) \\
&= real(v^T(2*CW))ur + i*imag(v^T(2*CW))ui
\end{aligned}
$$

We can thus use the fact that the backward mode AD provides us with an efficient way to compute $v^T(2*CW)$ and then perform a dot product of the real part with ur and the imaginary part with ui before reconstructing the final complex scalar s.

Why not use a complex *u*

At this point, you might be wondering why we did not select a complex u and just performed the reduction $2*v^TCWu'$. To dive into this, in this paragraph, we will use the complex version of u noted $u' = ur' + iui'$. Using such complex u', the problem is that when doing the numerical evaluation, we would need to compute:

$$
\begin{aligned}
2*CWu' &= (\frac{\partial y}{\partial a} + i\frac{\partial y}{\partial b})(ur' + iui') \\
&= \frac{\partial y}{\partial a}ur' + i\frac{\partial y}{\partial a}ui' + i\frac{\partial y}{\partial b}ur' - \frac{\partial y}{\partial b}ui'
\end{aligned}
$$

Which would require four evaluations of real-to-real finite difference (twice as much compared to the approached proposed above). Since this approach does not have more degrees of freedom (same number of real valued variables) and we try to get the fastest possible evaluation here, we use the other formulation above.

Fast gradcheck for functions with complex outputs

Just like in the slow case, we consider two real-valued functions and use the appropriate rule from above for each function.

Gradgradcheck implementation

PyTorch also provide a utility to verify second order gradients. The goal here is to make sure that the backward implementation is also properly differentiable and computes the right thing.

This feature is implemented by considering the function $F : x, v \to v^T J_f$ and use the gradcheck defined above on this function. Note that v in this case is just a random vector with the same type as $f(x)$.

The fast version of gradgradcheck is implemented by using the fast version of gradcheck on that same function F.