Numerical accuracy

In modern computers, floating point numbers are represented using IEEE 754 standard. For more details on floating point arithmetics and IEEE 754 standard, please see Floating point arithmetic In particular, note that floating point provides limited accuracy (about 7 decimal digits for single precision floating point numbers, about 16 decimal digits for double precision floating point numbers) and that floating point addition and multiplication are not associative, so the order of the operations affects the results. Because of this, pytorch is not guaranteed to produce bitwise identical results for floating point computations that are mathematically identical. Similarly, bitwise identical results are not guaranteed across PyTorch releases, individual commits, or different platforms. In particular, CPU and GPU results can be different even for bitwise-identical inputs and even after controlling for the sources of randomness.

Batched computations or slice computations

Many operations in pytorch support batched computation, where the same operation is performed for the elements of the batches of inputs. An example of this is :meth:`torch.mm` and :meth:`torch.bmm`. It is possible to implement batched computation as a loop over batch elements, and apply the necessary math operations to the individual batch elements, for efficiency reasons we are not doing that, and typically perform computation for the whole batch. The mathematical libraries that we are calling, and pytorch internal implementations of operations can produces slightly different results in this case, compared to non-batched computations. In particular, let `A` and `B` be 3D tensors with the dimensions suitable for batched matrix multiplication. Then `(A@B)[0]` (the first element of the batched result) is not guaranteed to be bitwise identical to `A[0]@B[0]` (the matrix product of the first elements of the input batches) even though mathematically it's an identical computation.

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

Unknown interpreted text role "meth".

Similarly, an operation applied to a tensor slice is not guaranteed to produce results that are identical to the slice of the result of the same operation applied to the full tensor. E.g. let `A` be a 2-dimentional tensor. `A.sum(-1)[0]` is not guaranteed to be bitwise equal to `A[:,0].sum()`.

Extremal values

When inputs contain large values such that intermediate results may overflow the range of the used datatype, the end result may overflow too, even though it is representable in the original datatype. E.g.:

```
import torch
a=torch.tensor([1e20, 1e20]) # fp32 type by default
a.norm() # produces tensor(inf)
a.double().norm() # produces tensor(1.4142e+20, dtype=torch.float64), representable in fp32
```

TensorFloat-32(TF32) on Nvidia Ampere devices

On Ampere Nvidia GPUs, pytorch by default uses TensorFloat32 (TF32) to speed up mathematically intensive operations, in particular matrix multiplications and convolutions. When operation is performed using TF32 tensor cores, only the first 10 bits of the input mantissa are read. This leads to less accurate results, and surprising results such as multiplying a matrix by identity matrix produces results that are different from the input. Most neural network workloads have the same convergence behavior when using tf32 as they have with fp32, however, if better accuracy is desired, TF32 can be turned off with `torch.backends.cuda.matmul.allow_tf32 = False`

For more information see :ref:`TensorFloat32<tf32_on_ampere>`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]numerical_accuracy.rst`, line 66); ***backlink***

Unknown interpreted text role "ref".

Reduced Precision Reduction for FP16 GEMMs

Half-precision GEMM operations are typically done with intermediate accumulations (reduction) in single-precision for numerical accuracy and improved resilience to overflow. For performance, certain GPU architectures, especially more recent ones, allow a few truncations of the intermediate accumulation results to the reduced precision (e.g., half-precision). This change is often benign from the perspective of model convergence, though it may lead to unexpected results (e.g., `inf` values when the final result should be be representable in half-precision). If reduced-precision reductions are problematic, they can be turned off with `torch.backends.cuda.matmul.allow_fp16_reduced_precision_reduction = False`

For more information see :ref:`allow_fp16_reduced_precision_reduction<fp16reducedprecision>`

System Message: ERROR/3 (**`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\notes\[pytorch-master][docs][source][notes]numerical_accuracy.rst`**, **line 74**); ***backlink***

Unknown interpreted text role "ref".