.. _numerical_accuracy: Numerical accuracy ================== In modern computers, floating point numbers are represented using IEEE 754 standard. For more details on floating point arithmetic 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. 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-dimensional 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.: .. code:: python 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 .. _Linear Algebra Stability: Linear algebra (``torch.linalg``) \---------- Non-finite values """""""""""""""""""""""""" The external libraries (backends) that ``torch.linalg`` uses provide no guarantees on their behaviour when the inputs have non-finite values like ``inf`` or ``NaN``. As such, neither does PyTorch. The operations may return a tensor with non-finite values, or raise an exception, or even segfault. Consider using :func:`torch.isfinite` before calling these functions to detect this situation. Extremal values in linalg "" Functions within ``torch.linalg`` have more `Extremal Values`_ than other PyTorch functions. :ref:`linalg solvers` and :ref:`linalg inverses` assume that the input matrix ``A`` is invertible. If it is close to being non-invertible (for example, if it has a very small singular value), then these algorithms may silently return incorrect results. These matrices are said to be `ill-conditioned `_. If provided with ill-conditioned inputs, the result of these functions they may vary when using the same inputs on different devices or when using different backends via the keyword ``driver``. Spectral operations like ``svd``, ``eig``, and ``eigh`` may also return incorrect results (and their gradients may be infinite) when their inputs have singular values that are close to each other. This is because the algorithms used to compute these decompositions struggle to converge for these inputs. Running the computation in ``float64`` (as NumPy does by default) often helps, but it does not solve these issues in all cases. Analyzing the spectrum of the inputs via :func:`torch.linalg.svdvals` or their condition number via :func:`torch.linalg.cond` may help to detect these issues. TensorFloat-32(TF32) on Nvidia Ampere (and later) devices \---------- On Ampere (and later) Nvidia GPUs, PyTorch can use TensorFloat32 (TF32) to speed up mathematically intensive operations, in particular matrix multiplications and convolutions. When an operation is performed using TF32 tensor cores, only the first 10 bits of the input mantissa are read. This may reduce accuracy and produce surprising results (e.g., multiplying a matrix by the identity matrix may produce results that are different from the input). By default, TF32 tensor cores are disabled for matrix multiplications and enabled for convolutions, although most neural network workloads have the same convergence behavior when using TF32 as

they have with fp32. We recommend enabling TF32 tensor cores for matrix multiplications with ``torch.backends.cuda.matmul.allow_tf32 = True`` if your network does not need full float32 precision. If your network needs full float32 precision for both matrix multiplications and convolutions, then TF32 tensor cores can also be disabled for convolutions with ``torch.backends.cudnn.allow_tf32 = False``. For more information see :ref:`TensorFloat32`. Reduced Precision Reduction for FP16 and BF16 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`` A similar flag exists for BF16 GEMM operations and is turned on by default. If BF16 reduced-precision reductions are problematic, they can be turned off with ``torch.backends.cuda.matmul.allow_bf16_reduced_precision_reduction = False`` For more information see :ref:`allow_fp16_reduced_precision_reduction` and :ref:`allow_bf16_reduced_precision_reduction` Reduced Precision Reduction for FP16 and BF16 in Scaled Dot Product Attention (SDPA) \---------- A naive SDPA math backend, when using FP16/BF16 inputs, can accumulate significant numerical errors due to the usage of low-precision intermediate buffers. To mitigate this issue, the default behavior now involves upcasting FP16/BF16 inputs to FP32. Computations are performed in FP32/TF32, and the final FP32 results are then downcasted back to FP16/BF16. This will improve numerical accuracy of the final output for the math backend with FP16/BF16 inputs, but increases memory usages and may cause the performance regressions in the math backend as computations shift from FP16/BF16

BMM to FP32/TF32 BMM/Matmul. For scenarios where reduced-precision reductions are preferred for speed, they can be enabled with the following setting: ``torch.backends.cuda.allow_fp16_bf16_reduction_math_sdp(True)`` .. _fp16_on_mi200: Reduced Precision FP16 and BF16 GEMMs and Convolutions on AMD Instinct MI200 devices \---------- On AMD Instinct MI200 GPUs, the FP16 and BF16 V_DOT2 and MFMA matrix instructions flush input and output denormal values to zero. FP32 and FP64 MFMA matrix instructions do not flush input and output denormal values to zero. The affected instructions are only used by rocBLAS (GEMM) and MIOpen (convolution) kernels; all other PyTorch operations will not encounter this behavior. All other supported AMD GPUs will not encounter this behavior. rocBLAS and MIOpen provide alternate implementations for affected FP16 operations. Alternate implementations for BF16 operations are not provided; BF16 numbers have a larger dynamic range than FP16 numbers and are less likely to encounter denormal values. For the FP16 alternate implementations, FP16 input values are cast to an intermediate BF16 value and then cast back to FP16 output after the accumulate FP32 operations. In this way, the input and output types are unchanged. When training using FP16 precision, some models may fail to converge with FP16 denorms flushed to zero. Denormal values more frequently occur in the backward pass of training during gradient calculation. PyTorch by default will use the rocBLAS and MIOpen alternate implementations during the backward pass. The default behavior can be overridden using environment variables, ROCBLAS_INTERNAL_FP16_ALT_IMPL and MIOPEN_DEBUG_CONVOLUTION_ATTRIB_FP16_ALT_IMPL. The behavior of these environment variables is as follows: +---------------+-----------+-----------+ | | forward | backward | +===============+===========+===========+ | Env unset | original | alternate | +---------------+-----------+-----------+ | Env set to 1 | alternate | alternate | +---------------+-----------+-----------+ | Env set to 0 | original | original | +---------------+-----------+-----------+ The following is the list of operations where rocBLAS may be used: * torch.addbmm * torch.addmm * torch.baddbmm * torch.bmm * torch.mm * torch.nn.GRUCell * torch.nn.LSTMCell * torch.nn.Linear * torch.sparse.addmm * the following torch._C._ConvBackend implementations: * slowNd *

slowNd_transposed * slowNd_dilated * slowNd_dilated_transposed The following is the list of operations where MIOpen may be used: * torch.nn.Conv[Transpose]Nd * the following torch._C._ConvBackend implementations: * ConvBackend::Miopen * ConvBackend::MiopenDepthwise * ConvBackend::MiopenTranspose