Measuring execution time is a difficult but critical aspect of the PyTorch development workflow. Good measurements allow us to effectively tune the framework and evaluate external contributions , which in turn translates to a good user experience with minimal performance cliffs and gotchas. This page will discuss some of the challenges of taking these measurements, and several experimental new components which have been introduced to make performance measurements easier and more robust. (#38338, README)

`Timer`: Benchmark code snippets

When benchmarking an expression, a number of factors can influence the result: * Number of threads * Environmental fluctuations * Initialization overheads * Device specific considerations (Cuda sync, TurboBoost, etc.) It's easy to overlook one aspect and inadvertently measure something very different from what was intended. For instance, a common stumbling block is to compare a single threaded TH implementation to a multithreaded ATen implementation (e.g. #39967) and inadvertently regress single threaded performance. Even once a benchmark script has resolved all systematic sources of bias, run-to-run variation can still significantly obscure measurements.

To address these challenges, a Timer class is introduced which closely mirrors the API of the builtin timeit module. It is, for the most part, a drop in replacement for the timeit.Timer class, with the following differences: * It is torch-aware, and will handle details like setting the number of threads (a constructor argument which defaults to one and sets num_threads to that value for the duration of the measurement) and CUDA synchronization. * In lieu of the `.repeat` and `.autorange` methods on `timeit.Timer`, the experimental PyTorch Timer exposes a `.blocked_autorange` function. Like autorange, blocked_autorange chooses an appropriate number of steps so that a measurement runs for at least a given period of time. (autorange hard codes 0.2 sec, while blocked_autorange allows the user to specify.) However, while autorange tries to take one measurement which is at least 0.2 seconds by tuning the number of times a statement is run, blocked_autorange finds the minimum number of times that it needs to run a statement while still keeping measurement overhead to a small fraction of the overall run time, and then runs as many replicates as it needs to fill the time budget. This has two benefits: first, it discards much less data (autorange discards 50-75% of its measurements as it tunes the number of runs), and second and more importantly it allows variation analysis to be performed on the measurement.

Rather than returning a float, the PyTorch Timer returns an instance of a `Measurement` class which contains all replicates performed. These Measurements are serializable, and provide some convenience facilities for interpreting results: the string representation will warn if a measurement has an undue amount of variation (as well as a `.has_warnings` property on Measurement to check programatically), and `.significant_figures` will offer a rudimentary estimate for the number of significant figures in a measurement which can be used when

interpreting results.

`Fuzzer`: Highly varied inputs

One challenge of writing performant kernels is that the optimal implementation can be input dependent. This is further complicated by the fact that PyTorch deliberately separates the logical and physical representations of a Tensor. Kernels which are only tuned on contiguous Tensors, power of two sizes and strides, or other such subsets of legal Tensors will tend to suffer from performance cliffs (or indeed, correctness issues) when inputs differ from those implicit assumptions.

The Fuzzer class addresses this challenge by taking a specification for random parameters and Tensors and a seed, and then producing Tensors which conform to the provided spec. The fuzzer handles random state so that measurements are repeatable (e.g. replicates or A/B testing in different processes), and automates key aspects such creating varying dimensional and/or non-contiguous Tensors which would otherwise be tedious to define. The fuzzing example, for instance, gives an example of a simple fuzzer which produces a Tensor with a random shape and rank. This fuzzing methodology has already identified several PRs where implementations overspecialized on a subset of potential inputs; for instance #39850 which would have regressed torch.topk on GPU for large values of k, and packaged fuzzers are planned for all major classes of ops to make it easy for developers to quickly and easily sanity check changes. (Unary and binary op fuzzers are currently provided.)

`Compare`: Result visualization

Compare addresses the converse challenge of Fuzzer: how to interpret measurements taken over a variety of contexts. The Compare class takes a list of Measurements, and converts it into a table summary. In addition to handling the mundane string formatting and alignment work, it will merge replicate runs and can be instructed to trim significant figures and color code the table to highlight trends. Results are grouped using the "label", "sub_label", "description", and "env" optional parameters in Timer (effectively tagging Measurements), as well as the number of threads used when running a measurement. This lets it put together a sensible table representation out of the box while still allowing the user to choose how data is organized.

In summary, if you are interested in measuring performance I would encourage you to try out these components, and if you have thoughts on how they could be improved feel free to open an issue and tag @robieta.