# PT2 Manifesto

PT2 is the beginning of PyTorch eager mode on compilers.

## Background

PyTorch grew up as an eager mode framework, touting the smooth user experience and debuggability of eager mode while still achieving comparable performance to graph mode frameworks on a variety of models. Early in PyTorch's lifetime, the framework overhead incurred by running in eager mode was dwarfed by the cost of matrix multiplies and convolutions that dominated the running time of networks.

In recent times, this position has become increasingly untenable:

- Hardware, including both NVIDIA GPUs as well as the cottage industry of ML hardware startups, is only getting faster. As compute gets faster, (1) framework overhead becomes increasingly visible and (2) it becomes harder to saturate compute, as kernel-by-kernel eager execution becomes increasingly bandwidth bound.
- Compilers for deep learning can already deliver asymptotic speed ups and memory usage improvements, and they will only get better over time. In some cases, e.g., distributed computation, we cannot achieve baseline functionality without having a compiler. Without a strategy for targeting compilers from the normal PyTorch user experience, we will lose out on all these improvements and benefits.
- To reduce framework overhead, more and more of PyTorch has been moved into C++, which has impaired the experience of both users (who can no longer easily dive into the PyTorch library and understand how operators are implemented / copy paste and tweak them for their own needs) and developers (who have to suffer through C++ edit-recompile cycle and all of the general complexity of developing in a low level language.)

PT2 is the beginning of PyTorch eager on compilers; the first release in a series of 2.x releases that will continue to improve PyTorch's eager mode performance with the help of compilers.

## P0 Goals

These goals are must have for the PT2 public release.

- PT2 will **increase the performance** over *training* on a selected set of benchmark models. See PyTorch 2.0 Performance Bets

    - Within Content Understanding and ML, we seek a 30% geomean performance gain across TorchBench + TIMM + HF-Transformers, each individually. This is the “top-line” performance number users will hear about in the PT2 release.
        - These numbers are based off of experiments showing that this performance gain is achievable with only pointwise fusion and matrix multiply epilogue fusion (in fact, at PT2 announce time we did not even have matrix multiply epilogue fusion).
    - These performance improvements are important because they deliver incremental impact and validate our strategy for compiler integration; however, it is less important than providing a solid base for compilers. PT2.0 is not the end of our compiler journey; *we are in this for the long haul.*
    - We expect inference performance will be improved by our efforts, but this is an explicit non-goal for PT2: inference is a crowded space targeted by many existing FX based tools.
    - We expect the majority of compute in our benchmarks to be FP16, as this regime is precisely the area where we will get the most benefit from compilers.
- PT2 will **be correct**: when users switch to using compilers in eager mode, their models will not start suffering from silent correctness issues. It will work, or there will be an error message explaining why it doesn’t work (but ideally, dynamo will have already detected this situation and fallen back to eager mode). That PT2 needs to be correct seems hardly worth stating, but actually achieving this goal will be nontrivial considering the breadth of how many components we are replacing. Here are some of the components that have nontrivial work to be correct:
    - We will evaluate correctness in the following ways:
        - Single step gradient comparison for benchmark models (assuming there is no randomness; see https://github.com/pytorch/pytorch/issues/77659 about ensuring that randomness is the same between eager and graph)
        - Training to convergence on a subset of benchmark models
        - Evaluation on our test suite using cross-ref testing (running both the old and new codepaths on our test suite and comparing for equality)
        - OpInfo testing
    - torchdynamo: the Python bytecode symbolic evaluator reimplements the semantics of all C API calls that it knows how to symbolically interpret against. Dynamo needs to know to bail out when there are things it doesn’t support.
    - Meta/fake tensors: we reimplement in Python shape and other metadata propagation functions; not only do these functions have to accurately produce shape information, but they also have to accurately propagate other metadata (e.g. strides).
    - AOTAutograd: this component is quite subtle as it is responsible for “erasing” all of PyTorch’s transforms, so it needs to be done correctly for every transform (autograd, batching, etc) that PyTorch supports. Some like autograd are quite subtle to do because of higher order derivatives and delayed execution. Even just first-order autograd can be quite subtle in the presence of input mutation, c.f. AOTAutograd 2.0 AOTAutograd and internal grad_fn structure

    - Decompositions (primtorch, aten to aten): these are reimplementations of existing operators in PyTorch in terms of other operators, and need to be functionally equivalent to the originals. This is a strictly harder problem than meta/fake tensors, since you have to do all the metadata from meta/fake tensors plus actually doing the compute
    - Backend correctness: Even though decompositions simplify the set of kernels backends have to implement, this simplified set still needs to be implemented correctly and not miscompiled when there are more complicated compositions
    - Caching: We wish to make compiled graphs that can be reused even if some shapes, eg. batch size, changes, but in some situations we will make assumptions based on the concrete value of a size. We must invalidate the trace if these assumptions change in the future. Failing to do so results in a soundness error.
    - Non-goal: bitwise identical numeric results to eager mode. Instead, we will only guarantee that the results are either as accurate or more accurate when compared to the double precision calculation, up to some tolerance.
- PT2 will have **good user experience**. This is a bit intangible but there are a number of things we mean by this.
    - Users can write their code in eager style, but transparently make use of a compiler. This is enabled by **torchdynamo**.
        - Unlike prior work with lazy tensors, users do not have to rewrite their models to make use of a lazy device (e.g., ltc or xla); their existing code can be used as-is with a compiler.
        - Unlike prior work with TorchScript, users do not have to rewrite their models to be TorchScriptable; our system can gracefully fallback whenever there are Python language features it does not support.
        - Unlike prior work with FX, our compiler integration will work with training.
    - PT2 will not infinitely recompile when given **dynamic shapes**. The number of recompilations we do is constant (not linear or exponential in program size).
    - The amount of compilation that is happening will be legible to the user, so they have the ability to trade off compilation time and performance
    - Compilation time will be linear in program size, not quadratic or exponential.
- PT2 will **work with distributed training**. We don't necessarily seek to improve distributed performance--however, as it turns out, we did see gains!
    - DDP and FSDP will continue to work and perform well when dynamo+backend is applied to the model.  This is thought to cover the majority of external use cases.
    - Scaling Efficiency
        - Scaling efficiency improvement is a non-goal
        - Scaling efficiency may be worsened due to single-node perf improvements with fixed/unchanged communications perf
        - Scaling efficiency **will not** be worsened due to breaking existing communication optimizations such as overlapping, stream usage, etc
- PT2 will provide infrastructure for a **no python export mode** for edge and performance sensitive serving cases. The PT2 team won't drive this end to end stack, but we will keep

a feedback loop with the teams in charge of this and ensure the components we build are reusable in these situations.
    - Dynamic shapes will be used to trace user code without baking in specific shapes.

# P1 Goals

These goals are not necessary for the PT2.0 initial release; but we may work on them to the extent they enable our P0 goals. They constitute our underlying strategy for how we get to the P0 goals.

- PT2 will unlock the ability for more of PyTorch's codebase to be **written in Python**. You no longer have to trade off the simplicity of Python with the performance of C++; you can write Python and get the same performance you would have gotten in C++.
    - More of PyTorch being written in Python means it will be easier for external users to hack on / remix internals. C++ that "looks like Python" can just be Python (so users can see them in backtraces and edit them without compiling PyTorch from source). Fused kernels can be written in decomposed form and modified without suffering from performance cliffs.
    - When planning new features to be built on PyTorch, the default choice should no longer be to write them in C++; instead, there should be a strong preference to write them in Python, with the understanding that in many circumstances PyTorch's compiler integration can eliminate Python overhead and the GIL.
- PT2 will provide shared infrastructure such as shape **propagation and decompositions** that can be reused by both our internal backend as well as external compiler backends targeting PyTorch. This will unlock a compiler community that is larger than just our internal compiler team.
- PT2 will provide a **foundational set of operators** that both eager and graph backends can target to hit maximum coverage (canonical ATen operators + PrimTorch). For example, to write a new functorch transform, it would be possible to bootstrap minimal (albeit inefficient) functionality using the prim set.

# Constraints

- We are not doing a big bang rewrite of PyTorch from scratch. In some sense, PT2 is not a project; it is a way to focus existing workstreams into a marketing moment and a north star for where PyTorch as a whole should be headed.