Instruction count microbenchmarks

Quick start

To run the benchmark:

```
# From pytorch root
cd benchmarks/instruction_counts
python main.py
```

Currently `main.py` contains a very simple threadpool (so that run time isn't unbearably onerous) and simply prints the results. These components will be upgraded in subsequent PRs.

To define a new benchmark:

- `TimerArgs`: Low level definition which maps directly to `torch.utils.benchmark.Timer`
- `GroupedStmts`: Benchmark a snippet. (Python, C++, or both) Can automatically generate TorchScript and autograd variants.
- `GroupedModules`: Like `GroupedStmts`, but takes `nn.Module`s
- `GroupedVariants`: Benchmark-per-line to define many related benchmarks in a single code block.

Architecture

Benchmark definition.

One primary goal of this suite is to make it easy to define semantically related clusters of benchmarks. The crux of this effort is the `GroupedBenchmark` class, which is defined in `core/api.py`. It takes a definition for a set of related benchmarks, and produces one or more concrete cases. It's helpful to see an example to understand how the machinery works. Consider the following benchmark:

```
# `GroupedStmts` is an alias of `GroupedBenchmark.init_from_stmts`
benchmark = GroupedStmts(
    py_stmt=r"y = x * w",
    cpp_stmt=r"auto y = x * w;",

    setup=GroupedSetup(
        py_setup="""
            x = torch.ones((4, 4))
            w = torch.ones((4, 4), requires_grad=True)
        """,
        cpp_setup="""
            auto x = torch::ones((4, 4));
            auto w = torch::ones((4, 4));
            w.set_requires_grad(true);
        """,
    ),

    signature="f(x, w) -> y",
    torchscript=True,
    autograd=True,
),
```

It is trivial to generate Timers for the eager forward mode case (ignoring `num_threads` for now):

```
Timer(
    stmt=benchmark.py_fwd_stmt,
    setup=benchmark.setup.py_setup,
)

Timer(
    stmt=benchmark.cpp_fwd_stmt,
    setup=benchmark.setup.cpp_setup,
    language="cpp",
)
```

Moreover, because `signature` is provided we know that creation of `x` and `w` is part of setup, and the overall comptation uses `x` and `w` to produce `y`. As a result, we can derive TorchScript'd and AutoGrad variants as well. We can deduce that a TorchScript model will take the form:

```
@torch.jit.script
def f(x, w):
    # Paste `benchmark.py_fwd_stmt` into the function body.
    y = x * w
    return y  # Set by `-> y` in signature.
```

And because we will want to use this model in both Python and C++, we save it to disk and load it as needed. At this point Timers for TorchScript become:

```
Timer(
    stmt="""
        y = jit_model(x, w)
    """,
    setup=""",
        # benchmark.setup.py_setup
        # jit_model = torch.jit.load(...)
        # Warm up jit_model
    """,
)

Timer(
    stmt="""
        std::vector<torch::jit::IValue> ivalue_inputs(
            torch::jit::IValue({x}),
            torch::jit::IValue({w})
        );
        auto y = jit_model.forward(ivalue_inputs);
    """,
    setup="""
        # benchmark.setup.cpp_setup
        # jit_model = torch::jit::load(...)
        # Warm up jit_model
    """,
)
```

While nothing above is particularly complex, there is non-trivial bookkeeping (managing the model artifact, setting up IValues) which if done manually would be rather bug-prone and hard to read.

The story is similar for autograd: because we know the output variable (`y`) and we make sure to assign it when calling TorchScript models, testing AutoGrad is as simple as appending `y.backward()` (or `y.backward();` in C++) to the stmt of the forward only variant. Of course this requires that `signature` be provided, as there is nothing special about the name `y` .

The logic for the manipulations above is split between `core/api.py` (for generating `stmt` based on language, Eager/TorchScript, with or without AutoGrad) and `core/expand.py` (for larger, more expansive generation). The benchmarks themselves are defined in `definitions/standard.py` . The current set is chosen to demonstrate the various model definition APIs, and will be expanded when the benchmark runner infrastructure is better equipped to deal with a larger run.

Benchmark execution.

Once `expand.materialize` has flattened the abstract benchmark definitions into `TimerArgs` , they can be sent to a worker (`worker/main.py`) subprocess to execution. This worker has no concept of the larger benchmark suite; `TimerArgs` is a one-to-one and direct mapping to the `torch.utils.benchmark.Timer` instance that the worker instantiates.