Design Doc: Backend-agnostic in Funsor

First Edit: 2020-01-13 - Last Edit: 2020-...
Authors: Fritz Obermeyer, Eli Bingham, Neeraj Pradhan, Du Phan

Objective

Each tensor framework has its own strengths and weaknesses. We want to make Funsor backend-agnostic to be able to use the suitable frameworks for some specific problems.

Another motivation is PPL such as NumPyro needs to use Funsor to support models with discrete latent variables.

Design Overview

Currently, the issue https://github.com/pyro-ppl/funsor/issues/207 tracks the progress on this project. So far, there are three approaches:

Approach 1

For each backend, we implement the corresponding set of modules: `tensor`, `gaussian`, `cnf`, `distributions`,... The file structure will then look like

```
funsor/
  ops.py
  domains.py
  terms.py
  ...
  torch/
    tensor.py
    gaussian.py
```

```
    distributions.py
    ...
  numpy/
    tensor.py
    gaussian.py
    distributions.py
    ...
```

We could in principle add an auto-import snippet in funsor/__init__.py like

```
FUNSOR_BACKEND = os.environ.get("FUNSOR_BACKEND", "")
if FUNSOR_BACKEND == "torch":
    from .torch import *
elif FUNSOR_BACKEND == "numpy":
    from .numpy import *
```

Advantages:

- We just need to change several import statements to use the corresponding functionalities.

Disadvantages:

- Most codes are duplicated.
- The overhead to maintain the same api/mechanism for different backends is large.

Approach 2

Having a unified class `Tensor`, subclass it into `TorchTensor` and `NumpyTensor` (currently, the corresponding class names are `Tensor` and `Array`). Replace all PyTorch operators `torch.foo(*)` or `*.foo()` where * is a numeric tensor by `terms.foo(*)` or `*.foo()`, where * is a funsor Tensor, in the implementation of `gaussian`, `cnf`,...

Advantages:

- Can use the same import statements.
- Less duplicated code.

Disadvantages:

- Need to rewrite the logics of `gaussian`, `cnf`,... because in the current implementation, many computations are applied directly to the data (`torch.Tensor`) of input funsors.
- Implement `terms.foo(*)` or `*.foo()` for a funsor is non-trivial.
- This would incur interpreter overhead for all low-level numerical math, and would require wrapping all low-level math with an @eager or @eager_only interpretation context.

Approach 3

Same as 2 but keep the current implementations of `gaussian`, `cnf`,... Replace all PyTorch operators `torch.foo(*)` or `*.foo()` by `ops.foo(*)` where * is a numeric tensor.

Advantages:

- Can use the same import statements.
- Less duplicated code.
- No need to rewrite the logics of `gaussian`, `cnf`,...
- Implement `ops.foo(*)` for a numeric tensor is quite straightforward.

Disadvantages:

- ...

We can also mix approaches 2 and 3 together, depending on specific functions.

Approach 4

Create a unified low-level wrapper layer around torch and numpy. Make funsor.Tensor agnostic to which backend it uses. The file structure will then look like

```
funsor/
  ops.py
  domains.py
  terms.py
  ...
  backends/
    torch/
      tensor.py
      distributions.py
    numpy/
      tensor.py
      distributions.py
  tensor.py  # agnostic
  gaussian.py
```

We could in principle add a `funsor.set_backend()` similar to `torch.set_default_tensor_type()`. In particular, we will never need to work with both backends at the same time, so a global config is fine.

We might be able to use a global import to set backend (as in approach 1)?

```
# in funsor/__init__.py
if FUNSOR_BACKEND == "torch":
    from .torch import distributions, tensor
elif FUNSOR_BACKEND == "numpy":
    from .torch import distributions, tensor
```

Advantages:

- There is clear separation of the two abstraction mechanisms
- funsor.Tensor code is shared across backends
- does not incur interpreter overhead for low-level numerics

Disadvantages:

- Requires rewriting funsor.Tensor
- Incurs coding overhead when creating new funsor.torch.function()s
- Distributions implementations still differ across backend

Detailed Design

In approaches 2 and 3, currently, instead of having a unified class `Tensor`, subclass it into `TorchTensor` and `NumpyTensor`, we implemented `Tensor` and `Array` for torch and numpy backends separately. Either way, we need a mechanism to create a new `Tensor` because in funsor, many functions involve constructing new Tensors:

- If we have a unified class `Tensor`, then we can replace all Tensor constructions in `gaussian`, `cnf`, `...` by a method *.new_tensor(...) (for example).

- In https://github.com/pyro-ppl/funsor/pull/302, dispatches of a function call `Tensor(...)` to `TorchTensor`, `NumpyTensor` are used depending on input tensor types of those constructions. This is not elegant but will keep the current Tensor construction statements the same as before.

...

Plan

Make basic functionalities backend-agnostic.

In approaches 2 and 3, it is hard to recognize which statements in the current implementations are numeric ops. So we need to gradually add `numpy` tests and make corresponding changes.

Replace numpy by jax.numpy.

This is to prepare for later steps.

Make sample mechanism backend-agnostic.

We will need to import numpyro at this step.

Make inference backend-agnostic.

…

Possible coding steps

1. Remove funsor.Tensor and logic from torch.py but keep op registrations
2. Remove funsor.Array class from numpy.py but keep op registrations
3. Make a funsor/tensor.py with a new backend-agnostic Tensor class
4. Refactor funsor/gaussian.py to be backend-agnostic (similar to 3 and Du's current PR)