torch.func

Functional Transforms in PyTorch

Toronto Machine Learning Summit 2024

@shagunsodhani

About Me

1. Tech Lead and Staff Research Engineer @ Meta AI

2. Training foundation models to develop adaptive neural interfaces

Agenda

1. Quick overview of PyTorch
2. Why torch.func
3. Overview of torch.func API
4. Where can you start using torch.func today
5. Gotachs to look out for
6. Questions and Answers

PyTorch

1. Open-source Machine Learning framework

2. Provides Numpy-like arrays with GPU acceleration

3. Enables training deep neural networks

4. Well-known for its ease of use

Why torch.func

Following use cases are tricky to do in PyTorch

1. computing per-sample-gradients
2. running model ensembles on a single machine
3. efficiently batching together tasks in the inner-loop of MAML-like algorithms
4. efficiently computing Jacobians and Hessians (with or without batching)

These can be supported by introducing composable function transforms

Why torch.func

Composable Function Transforms

Why torch.func

Composable Function Transforms

1. Higher-order function that accepts functions are input and returns function as output.

Why torch.func

Composable Function Transforms

1. Higher-order function that accepts functions are input and returns function as output.

2. Examples include auto-differentiation transforms, `grad(f)` that returns a function that computes the gradient of `f` or vectorization/batching transform, `vmap(f)` that returns a function that computes `f` over batches of inputs.

Why torch.func

Composable Function Transforms

These function transforms can compose with each other arbitrarily. For example, composing `vmap(grad(f))` computes per-sample-gradients!

Why torch.func

1. In general, having "pure" functions makes it easier to compose them.
    a. Functions that always produce the same output for the same input and have no side effects

2. In PyTorch, commonly used constructs like modules are stateful.

3. `torch.func` makes it easier to embrace the functional programming style, which in-turn simplifies some workflows easier.

API | `vmap`

Given a function `func` that runs on a single example, we can lift it to a function that can take batches of examples with `vmap(func)`

`vmap(func)` adds a dimension to all tensor operations in `func`

It can be invoked as `vmap(func)(*inputs)`

API | vmap

```
x: torch.Tensor = torch.randn(100)
y: torch.Tensor = torch.randn(100)
x_dot_y = torch.dot(x, y)
print(f"{x_dot_y=}")
```

```
x_dot_y=tensor(-3.0892)
```

API | vmap

```python
x: torch.Tensor = torch.randn(100)
y: torch.Tensor = torch.randn(100)
x_dot_y = torch.dot(x, y)
print(f"{x_dot_y=}")
```

```
x_dot_y=tensor(-3.0892)
```

```python
x: torch.Tensor = torch.randn(10, 100)
y: torch.Tensor = torch.randn(10, 100)
x_dot_y = torch.dot(x, y)
```

```
RuntimeError:1D tensors expected, but got 2D and 2D tensors
```

API | vmap

```python
x: torch.Tensor = torch.randn(10, 100)
y: torch.Tensor = torch.randn(10, 100)
x_dot_y = torch.ones(10)
for i in range(x.shape[0]):
    x_dot_y[i] = torch.dot(x[i], y[i])
print(f"{x_dot_y=}")
```

```
x_dot_y=tensor([  8.2551,  13.3984, -11.2821, -
         -7.5680,  -3.9727,  -0.6121])
```

API | vmap

```python
batched_dot_product = torch.func.vmap(torch.dot)
x_dot_y_using_vmap = batched_dot_product(x, y)
print(f"{x_dot_y_using_vmap=}")
```

```
x_dot_y_using_vmap=tensor([  8.2551,  13.3984, -11.282
         -7.5680,  -3.9727,  -0.6121])
```

API | vmap

```python
batched_dot_product = torch.func.vmap(torch.dot)
x_dot_y_using_vmap = batched_dot_product(x, y)
print(f"{x_dot_y_using_vmap=}")
```

```
x_dot_y_using_vmap=tensor([  8.2551,  13.3984, -11.282
        -7.5680,  -3.9727,  -0.6121])
```

```python
assert torch.allclose(x_dot_y, x_dot_y_using_vmap)
```

API | `grad`

`grad` operator helps computing gradients of `func` .

This operator can be nested to compute higher-order gradients.

API | grad

```
sin_x = lambda x: torch.sin(x)
grad_sin_x = torch.func.grad(sin_x)
x = torch.randn([])
assert torch.allclose(grad_sin_x(x), x.cos())
```

API | grad

```python
grad_grad_sin_x = torch.func.grad(grad_sin_x)
assert torch.allclose(grad_grad_sin_x(x), -x.sin())
```

API | `grad + vmap`

When composed with vmap, grad can be used to compute per-sample-gradients:

API | grad + vmap

```python
from torch.func import grad, vmap

batch_size, feature_size = 3, 5

def model(weights: torch.Tensor, feature_vec: torch.Tensor) -> torch.Tensor:
    return feature_vec.dot(weights).relu()

def compute_loss(weights: torch.Tensor, example: torch.Tensor, target: torch.Tensor) -> torch.Tensor:
    y = model(weights, example)
    return ((y - target) ** 2).mean()  # MSELoss'

weights = torch.randn(feature_size, requires_grad=True)
examples = torch.randn(batch_size, feature_size)
targets = torch.randn(batch_size)
```

API | grad + vmap

```python
inputs = (weights, examples, targets)
grad_of_loss = grad(compute_loss)
grad_of_loss_per_sample = vmap(grad_of_loss, in_dims=(None, 0, 0))

grad_weight_per_example = grad_of_loss_per_sample(*inputs)
print(grad_weight_per_example)
```

```
tensor([[-0.0000,  0.0000, -0.0000, -0.0000, -0.0000],
        [-0.0352,  0.0429,  0.0334, -0.0174,  0.0097],
        [ 0.0000, -0.0000, -0.0000,  0.0000, -0.0000]], grad_fn=<MulBackward0>)
```

API | `functional_call`

Performs a functional call on the module by replacing the module parameters and buffers with the provided ones.

API | functional_call

```
x = torch.randn(4, 3)
t = torch.randn(4, 3)

model = nn.Linear(3, 3)

params = dict(model.named_parameters())
y = functional_call(model, params, x)

assert torch.allclose(y, model(x))
```

API | functional_call

```python
def compute_loss(
    params: dict[str, torch.Tensor], x: torch.Tensor, t: torch.Tensor
) -> torch.Tensor:
    y = functional_call(model, params, x)
    return nn.functional.mse_loss(y, t)


grad_of_loss = grad(compute_loss)
grad_weights = grad_of_loss(dict(model.named_parameters()), x, t)
```

API | stack_module_state

```
def forward_call(params, buffers, data):
    return torch.func.functional_call(models[0], (params, buffers), data)

vmap_forward_call = vmap(forward_call, (0, 0, None))

params, buffers = torch.func.stack_module_state(models)

output = vmap_forward_call(params, buffers, data)

assert output.shape == (num_models, batch_size, out_features)
```

API

Other examples include

- vjp (vector jacobian product)
- jvp (jacobian vector product)
- hessian
- ...

Gotchas

Using PyTorch torch.no_grad together with grad.

Case 1: Using torch.no_grad inside a function:

```
>>> def f(x):
>>>     with torch.no_grad():
>>>         c = x ** 2
>>>     return x - c
```

In this case, grad(f)(x) will respect the inner torch.no_grad.

Case 2: Using grad inside torch.no_grad context manager:

```
>>> with torch.no_grad():
>>>     grad(f)(x)
```

In this case, grad will respect the inner torch.no_grad, but not the outer one. This is because grad is a “function transform”: its result should not depend on the result of a context manager outside of f.

Gotchas

1. Functions with side-effects / global effects can be problematic
2. `vmap` does not work with some inplace operations
3. `vmap` does not work with some data dependent conditionals
4. Batchnorm requires special handling
5. For more gotachs, checkout this

Thank you!

https://shagunsodhani.com/talks/

Toronto Machine Learning Summit 2024

@shagunsodhani