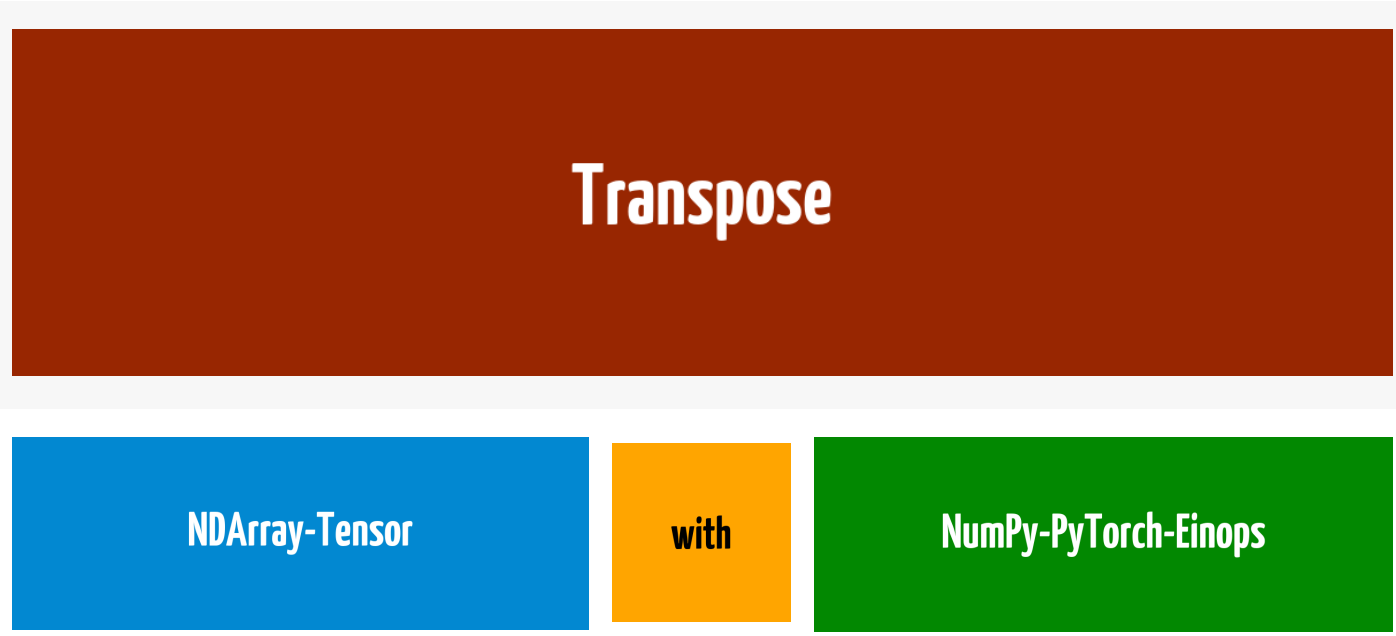
AUTHOR: Sugato Ray

- **LinkedIn**: https://www.linkedin.com/in/sugatoray/
- **Twitter**: https://twitter.com/sugatoray
- **GitHub**: https://github.com/sugatoray



Introduction

Say, you have a `4D` tensor/ndarray: `x`.

NumPy

👉 If you use **NumPy**, `numpy.transpose(x, (1, 0, 2, 3))` allows you to swap dimensions even for a multidimensional array. Here it changes dims `(0, 1, 2, 3)` to `(1, 0, 2, 3)`.

👉 Numpy also has two other methods called `numpy.swapaxes(x, 0, 1)` and `numpy.moveaxes(x, 0, 1)`.

All three `numpy` methods above will give you the same output.

PyTorch

👉 Similarly, if you use **PyTorch**, `torch.transpose(x, 0, 3)` will swap dims `0` and `3`. Alternatively, you can use `torch.Tensor.transpose()`: `x.transpose(0, 3)`.

👉 Alternatively, if you have heard of Einstein notation, there is a much simpler and more verbose notation that could not only do transposition, but a lot of other operations.

Einsum with NumPy or PyTorch

- `torch.einsum()` or `numpy.einsum()` could be used for that purpose.

Einops + PyTorch

👉 Or, you can use the **Einops** library to do the same with `einops.rearrange()` function.

A. With **NumPy**

```
import numpy as np

x = np.arange(120).reshape((2, 3, 4, 5))
x.shape
```

```
(2, 3, 4, 5)
```

A.1. Transposition using `numpy.einsum()`

- Docs: https://numpy.org/doc/stable/reference/generated/numpy.einsum.html#numpy.einsum

```
x.shape, "-->", np.einsum("ijkl->jikl", x).shape
```

```
((2, 3, 4, 5), '-->', (3, 2, 4, 5))
```

A.2. Transposition using `numpy.transpose()`

- Docs: https://numpy.org/doc/stable/reference/generated/numpy.transpose.html#numpy.transpose

```
x.shape, "-->", np.transpose(x, axes=(1, 0, 2, 3)).shape
```

```
((2, 3, 4, 5), '-->', (3, 2, 4, 5))
```

Observe that using `numpy.transpose()` or `x.transpose()` is **NOT** the same as using `x.T` (shown in the cell below), as for a multi-dimensional array just calling transpose on it is a bit ambiguous. You aren't really telling `numpy` which axes to apply tranposition on!

This is why, in my opinion, although `numpy.transpose()` is a bit verbose, it is more self-explanatory and provides the user the option to specify the target set of dimensions (axes) to work with.

```
x.shape, "-->", x.T.shape
```

```
((2, 3, 4, 5), '-->', (5, 4, 3, 2))
```

The code in the cell above is equivalent to the code in the next cell.

```
x.shape, "-->", np.transpose(x, axes=(3, 2, 1, 0)).shape
```

```
((2, 3, 4, 5), '-->', (5, 4, 3, 2))
```

A.3. Transposition using `numpy.swapaxes()`

- Docs: https://numpy.org/doc/stable/reference/generated/numpy.swapaxes.html#numpy.swapaxes

```
x.shape, "-->", np.swapaxes(x, 0, 1).shape
```

```
((2, 3, 4, 5), '-->', (3, 2, 4, 5))
```

A.4. Transposition using `numpy.moveaxis()`

- Docs: https://numpy.org/doc/stable/reference/generated/numpy.moveaxis.html#numpy.moveaxis

```
x.shape, "-->", np.moveaxis(x, 0, 1).shape
```

```
((2, 3, 4, 5), '-->', (3, 2, 4, 5))
```

B. With **PyTorch**

```
import torch

y = torch.from_numpy(x)
tuple(y.shape)
```

```
(2, 3, 4, 5)
```

B.1. Transposition using `torch.einsum()`

- Docs: https://pytorch.org/docs/master/generated/torch.einsum.html

```
tuple(y.shape), "-->", tuple(torch.einsum("ijkl->jikl", y).shape)
```

```
((2, 3, 4, 5), '-->', (3, 2, 4, 5))
```

B.2. Transposition using `torch.transpose()`

- Docs: https://pytorch.org/docs/master/generated/torch.transpose.html

> Note the difference b/w `numpy.transpose(x, axes=(1, 0, 2, 3))` and `torch.transpose(y, 1, 0)`.

```
tuple(y.shape), "-->", tuple(torch.transpose(y, 1, 0).shape)
```

```
((2, 3, 4, 5), '-->', (3, 2, 4, 5))
```

```
tuple(y.shape), "-->", tuple(y.mT.shape) # use y.mT instead of y.T ; otherwise u
```

```
((2, 3, 4, 5), '-->', (2, 3, 5, 4))
```

But using `torch.transpose()` gives you the option to swap non-adjascent dimensions as well.

Example:

- dims: `(0, 1, 2, 3) --> (2, 1, 0, 3)`
- shape: `(2, 3, 4, 5) --> (4, 3, 2, 5)`

```
tuple(y.shape), "-->", tuple(torch.transpose(y, 2, 0).shape)
```

```
((2, 3, 4, 5), '-->', (4, 3, 2, 5))
```

B.3a. Transposition using `torch.swapaxes()`

- Docs: https://pytorch.org/docs/master/generated/torch.swapaxes.html

```
tuple(y.shape), "-->", tuple(torch.swapaxes(y, 0, 1).shape)
```

```
((2, 3, 4, 5), '-->', (3, 2, 4, 5))
```

B.3b. Transposition using `torch.swapdims()`

- Docs: https://pytorch.org/docs/master/generated/torch.swapdims.html

```
tuple(y.shape), "-->", tuple(torch.swapdims(y, 0, 1).shape)
```

```
((2, 3, 4, 5), '-->', (3, 2, 4, 5))
```

B.4a. Transposition using `torch.moveaxis()`

- Docs: https://pytorch.org/docs/master/generated/torch.moveaxis.html

```
tuple(y.shape), "-->", tuple(torch.moveaxis(y, 0, 1).shape)
```

```
((2, 3, 4, 5), '-->', (3, 2, 4, 5))
```

B.4b. Transposition using `torch.movedim()`

- Docs: https://pytorch.org/docs/master/generated/torch.movedim.html

```
tuple(y.shape), "-->", tuple(torch.movedim(y, 0, 1).shape)
```

```
((2, 3, 4, 5), '-->', (3, 2, 4, 5))
```

C. With **Einops**

The intuitive but minimalistic API of `einops` ships out three operations: `rearrange`, `reduce` and `repeat`.

The three operations, as shown in `einops` tutorial, cover:

- stacking
- reshape
- transposition
- squeeze/unsqueeze
- repeat
- tile
- concatenate
- view
- numerous reductions

Source: https://github.com/arogozhnikov/einops

```
%%capture
! pip install -Uqq einops
```

```
from einops import rearrange
from einops.layers.torch import Rearrange # See in einpos docs to know how to us
```

Now, that we have seen how to use `einsum()` in both `numpy` and `torch`, you will find the following treatment quite intuitive.

Transposition using `einpos.rearrange()`

- Docs:
  - GitHub: https://github.com/arogozhnikov/einops
  - Examples: http://einops.rocks/pytorch-examples.html

```
1 tuple(y.shape), "-->", tuple(rearrange(y, 'i j k l -> j i k l').shape)
```

```
((2, 3, 4, 5), '-->', (3, 2, 4, 5))
```

In fact, you can even use **words** instead of single-characters as indices (see below).

```
1 tuple(y.shape), "-->", tuple(rearrange(y, 'time channel h w -> channel time h w'
```

```
((2, 3, 4, 5), '-->', (3, 2, 4, 5))
```

```
1
```