1.5. High dimension tensors

François Fleuret

https://fleuret.org/ee559/

Feb 7, 2020
A tensor can be of several types:

- `torch.float16, torch.float32, torch.float64`,
- `torch.uint8`,
- `torch.int8, torch.int16, torch.int32, torch.int64`

and can be located either in the CPU’s or in a GPU’s memory.

Operations with tensors stored in a certain device’s memory are done by that device. We will come back to that later.
```python
>>> x = torch.zeros(1, 3)
>>> x.dtype, x.device
(torch.float32, device(type='cpu'))
>>> x = x.long()
>>> x.dtype, x.device
(torch.int64, device(type='cpu'))
>>> x = x.to('cuda')
>>> x.dtype, x.device
(torch.int64, device(type='cuda', index=0))
```
2d tensor \((e.g.\) grayscale image)
3d tensor (e.g. rgb image)
4d tensor (e.g. sequence of rgb images)
Here are some examples from the vast library of tensor operations:

**Creation**
- `torch.empty(*size, ...)`
- `torch.zeros(*size, ...)`
- `torch.full(size, value, ...)`
- `torch.tensor(sequence, ...)`
- `torch.eye(n, ...)`
- `torch.from_numpy(ndarray)`

**Indexing, Slicing, Joining, Mutating**
- `torch.Tensor.view(*size)`
- `torch.cat(inputs, dimension=0)`
- `torch.chunk(tensor, nb_chunks, dim=0)`
- `torch.split(tensor, split_size, dim=0)`
- `torch.index_select(input, dim, index, out=None)`
- `torch.t(input, out=None)`
- `torch.transpose(input, dim0, dim1, out=None)`

**Filling**
- `Tensor.fill_(value)`
- `torch.bernoulli_(proba)`
- `torch.normal_([mu, [std]])`
Pointwise math
- torch.abs(input, out=None)
- torch.add()
- torch.cos(input, out=None)
- torch.sigmoid(input, out=None)
- (+ many operators)

Math reduction
- torch.dist(input, other, p=2, out=None)
- torch.mean()
- torch.norm()
- torch.std()
- torch.sum()

BLAS and LAPACK Operations
- torch.eig(a, eigenvectors=False, out=None)
- torch.lstsq(B, A, out=None)
- torch.inverse(input, out=None)
- torch.mm(mat1, mat2, out=None)
- torch.mv(mat, vec, out=None)
\[ x = \text{torch.tensor}(\begin{bmatrix} 1 & 3 & 0 \\ 2 & 4 & 6 \end{bmatrix}) \]
```
x = torch.tensor([[1, 3, 0],
                  [2, 4, 6]])
```

```
x.view(-1)
```
\[
x = \text{torch.tensor([[1, 3, 0], [2, 4, 6]])}
\]

\[
x.\text{view}(3, -1)
\]
\[ x = \text{torch.tensor([[1, 3, 0], \[2, 4, 6\]])} \]
\[ x = \text{torch.tensor}([ [ 1, 3, 0 ], [ 2, 4, 6 ] ]) \]

\[ x.\text{view}(1, 2, 3).\text{expand}(3, 2, 3) \]
\[
x = \text{torch.tensor}([[\begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix}],
\begin{bmatrix} 3 & 0 & 3 \\ 0 & 3 & 0 \end{bmatrix}])
\]

\[
x[0:1, :, :]
\]
\[ x = \text{torch.tensor}([ \begin{bmatrix} 1, & 2, & 1 \\ 2, & 1, & 2 \end{bmatrix}, \begin{bmatrix} 3, & 0, & 3 \\ 0, & 3, & 0 \end{bmatrix} ]) \]

\[ x[:, :, 0:2] \]
\[
x = \text{torch.tensor([[ [1, 2, 1],
        [ 2, 1, 2 ] ],
        [ [3, 0, 3],
        [ 0, 3, 0 ]]])}
\]

\[x.\text{transpose}(0, 1)\]
\begin{align*}
x &= \text{torch.tensor}([\begin{bmatrix} 1, 2, 1 \\ 2, 1, 2 \\ 3, 0, 3 \\ 0, 3, 0 \end{bmatrix}]) \\
x &\quad\text{.transpose(0, 2)}
\end{align*}
\[ x = \text{torch.tensor}([ [ [ 1, 2, 1 ], [ 2, 1, 2 ] ], [ [ 3, 0, 3 ], [ 0, 3, 0 ] ] ]) \]

\[ x.\text{transpose}(1, 2) \]
PyTorch offers simple interfaces to standard image data-bases.

```python
import torch, torchvision
cifar = torchvision.datasets.CIFAR10('./cifar10/', train = True, download = True)
x = torch.from_numpy(cifar.data).permute(0, 3, 1, 2).float() / 255
print(x.dtype, x.size(), x.min().item(), x.max().item())
```
PyTorch offers simple interfaces to standard image data-bases.

```python
import torch, torchvision
cifar = torchvision.datasets.CIFAR10('./cifar10/', train = True, download = True)
x = torch.from_numpy(cifar.data).permute(0, 3, 1, 2).float() / 255
print(x.dtype, x.size(), x.min().item(), x.max().item())
```

prints

```
Files already downloaded and verified
torch.float32 torch.Size([50000, 3, 32, 32]) 0.0 1.0
```
PyTorch offers simple interfaces to standard image data-bases.

```python
import torch, torchvision
cifar = torchvision.datasets.CIFAR10('./cifar10/', train = True, download = True)
x = torch.from_numpy(cifar.data).permute(0, 3, 1, 2).float() / 255
print(x.dtype, x.size(), x.min().item(), x.max().item())
```

prints

```
Files already downloaded and verified
torch.float32 torch.Size([50000, 3, 32, 32]) 0.0 1.0
```

Françoise Fleuret
# Narrows to the first images, converts to float

\[ x = x[:48] \]

# Saves these samples as a single image

```
torchvision.utils.save_image(x, 'cifar-4x12.png',
                      nrow = 12, pad_value = 1.0)
```
# Switches the row and column indexes
x.transpose_(2, 3)
torchvision.utils.save_image(x, 'cifar-4x12-rotated.png',
nrow = 12, pad_value = 1.0)
# Kills the green and blue channels
x[:, 1:3].fill_(0)
torchvision.utils.save_image(x, 'cifar-4x12-rotated-and-red.png',
                             nrow = 12, pad_value = 1.0)
Broadcasting and dimension naming
Broadcasting automagically expands dimensions by replicating coefficients, when it is necessary to perform operations that are “intuitively reasonable”.

```python
>>> x = torch.empty(100, 4).normal_(2)
>>> x.mean(0)
tensor([2.0476, 2.0133, 1.9109, 1.8588])
>>> x -= x.mean(0) # This should not work!
>>> x.mean(0)
tensor([-4.0531e-08, -4.4703e-07, -1.3471e-07, 3.5763e-09])
```
Broadcasting automagically expands dimensions by replicating coefficients, when it is necessary to perform operations that are “intuitively reasonable”.

For instance:

```python
>>> x = torch.empty(100, 4).normal_(2)
>>> x.mean(0)
tensor([2.0476, 2.0133, 1.9109, 1.8588])
>>> x -= x.mean(0) # This should not work!
>>> x.mean(0)
tensor([-4.0531e-08, -4.4703e-07, -1.3471e-07, 3.5763e-09])
```
Precisely, broadcasting proceeds as follows:

1. If one of the tensors has fewer dimensions than the other, it is reshaped by adding as many dimensions of size 1 as necessary in the front; then

2. for every dimension mismatch, **if one of the two tensors is of size one**, it is expanded along this axis by replicating coefficients.

If there is a tensor size mismatch for one of the dimension and neither of them is one, the operation fails.
A = torch.tensor([[1.], [2.], [3.], [4.]])
B = torch.tensor([[5., -5., 5., -5., 5.]])
C = A + B
A = torch.tensor([[1.], [2.], [3.], [4.]])
B = torch.tensor([[5., -5., 5., -5., 5.]])
C = A + B
A = torch.tensor([[1.], [2.], [3.], [4.]])
B = torch.tensor([[5., -5., 5., -5., 5.]])
C = A + B

Broadcasted:

\[
\begin{array}{ccccc}
1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 & 4 \\
\end{array}
\]

\[
\begin{array}{ccccc}
5 & -5 & 5 & -5 & 5 \\
5 & -5 & 5 & -5 & 5 \\
5 & -5 & 5 & -5 & 5 \\
5 & -5 & 5 & -5 & 5 \\
\end{array}
\]

\[
\begin{array}{cccccc}
6 & -4 & 6 & -4 & 6 \\
7 & -3 & 7 & -3 & 7 \\
8 & -2 & 8 & -2 & 8 \\
9 & -1 & 9 & -1 & 9 \\
\end{array}
\]

C = A + B
To deal with complex operations, PyTorch provides a dimension naming mechanism:

```python
>>> seq = torch.empty(100, 3, 1024, names = [ 'n', 'c', 't' ]).normal_()
>>> seq.mean('t').size()
torch.Size([100, 3])
>>> time_first = seq.align_to('n', 't', 'c')
>>> time_first.size()
torch.Size([100, 1024, 3])
>>> array = seq.flatten([ 'c', 't' ], 'i')
>>> array.size()
torch.Size([100, 3072])
>>> array.names
('n', 'i')
```
To deal with complex operations, PyTorch provides a dimension naming mechanism:

```python
>>> seq = torch.empty(100, 3, 1024, names = [ 'n', 'c', 't' ]).normal_()
>>> seq.mean('t').size()
torch.Size([100, 3])
>>> time_first = seq.align_to('n', 't', 'c')
>>> time_first.size()
torch.Size([100, 1024, 3])
>>> array = seq.flatten([ 'c', 't' ], 'i')
>>> array.size()
torch.Size([100, 3072])
>>> array.names
('n', 'i')
```
To deal with complex operations, PyTorch provides a dimension naming mechanism:

```python
>>> seq = torch.empty(100, 3, 1024, names=['n', 'c', 't']).normal_()
>>> seq.mean('t').size()
torch.Size([100, 3])
>>> time_first = seq.align_to('n', 't', 'c')
>>> time_first.size()
torch.Size([100, 1024, 3])
>>> array = seq.flatten(['c', 't'], 'i')
>>> array.size()
torch.Size([100, 3072])
>>> array.names
('n', 'i')
```
The end