AMMI – Introduction to Deep Learning

1.5. High dimension tensors

François Fleuret
https://fleuret.org/ammi-2018/
Fri Nov 9 22:40:18 UTC 2018
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 on tensors located in 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.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, chunks, dim=0)[source]`
- `torch.split(tensor, split size, dim=0)[source]`
- `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.gels(B, A, out=None)
- torch.inverse(input, out=None)
- torch.mm(mat1, mat2, out=None)
- torch.mv(mat, vec, out=None)
\[
x = \text{torch.tensor([[1, 3, 0],
                      [12, 4, 6]])}
\]

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

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

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

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

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

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

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

$$x.\text{transpose}(0, 2)$$
```python
x = torch.tensor([[ [ 1, 2, 1 ],
                  [ 2, 1, 2 ] ],
                  [ [ 3, 0, 3 ],
                  [ 0, 3, 0 ] ]])

x.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.train_data).transpose(1, 3).transpose(2, 3).float()
x = x / 255
print(x.type(), 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.train_data).transpose(1, 3).transpose(2, 3).float()
x = x / 255
print(x.type(), x.size(), x.min().item(), x.max().item())

prints

Files already downloaded and verified
torch.FloatTensor 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.train_data).transpose(1, 3).transpose(2, 3).float()
x = x / 255
print(x.type(), x.size(), x.min().item(), x.max().item())
```

prints

Files already downloaded and verified
torch.FloatTensor torch.Size([50000, 3, 32, 32]) 0.0 1.0
# Narrows to the first images, converts to float
x = x.narrow(0, 0, 48).float()

# Saves these samples as a single image
torchvision.utils.save_image(x, 'cifar-4x12.png', nrow = 12)
# Switches the row and column indexes
x.transpose_(2, 3)
torchvision.utils.save_image(x, 'cifar-4x12-rotated.png', nrow = 12)
# Kills the green and blue channels
x.narrow(1, 1, 2).fill_(0)
torchvision.utils.save_image(x, 'cifar-4x12-rotated-and-red.png', nrow = 12)
Broadcasting
**Broadcasting** automagically expands dimensions by replicating coefficients, when it is necessary to perform operations.
**Broadcasting** automagically expands dimensions by replicating coefficients, when it is necessary to perform operations.

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)
>>> 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.

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)
>>> x.mean(0)
tensor([-4.0531e-08, -4.4703e-07, -1.3471e-07, 3.5763e-09])
```

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 mismatch, **if one of the two tensor 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
\begin{align*}
A &= \text{torch.tensor([[1.], [2.], [3.], [4.]])} \\
B &= \text{torch.tensor([[5., -5., 5., -5., 5.]])} \\
C &= A + B
\end{align*}
A = torch.tensor([[1.], [2.], [3.], [4.]])
B = torch.tensor([[5., -5., 5., -5., 5.]])
C = A + B
The end