Deep learning

1.5. High dimension tensors

François Fleuret

https://fleuret.org/dlc/

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.

Notes

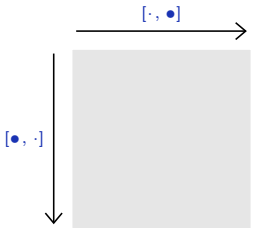
All the coefficients in a given tensor are of the same type, which can be either an integer or floating point value of a certain precision.

```
>>> 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))
```

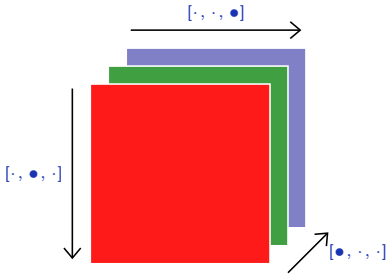

Notes

The default type of tensor values is `torch.float32`, and the default computing device is the CPU.
The data type of the tensor can be accessed with `dtype` and the device on which the tensor lies with `device`.
When casting a tensor to a new type (for instance here with `x = x.long()`), a copy is actually made. If the type is already adequate, a reference to the same tensor is returned.
It is a best practice to define the device that is going to be used once for all at the beginning of a program, and use the method `to(device)` to move the data to the target device.

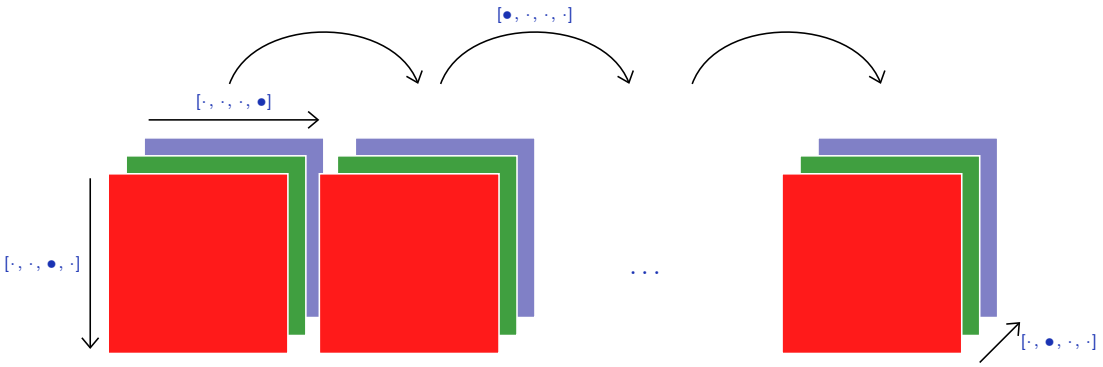
2d tensor (e.g. grayscale image)



3d tensor (e.g. rgb image)



4d tensor (e.g. sequence of rgb images)



Notes

In these figures, the • marker denotes the index of the dimension corresponding to the drawn axis, and · denotes the other dimensions.
A 2d tensor can be seen as a grayscale image: the first index is the row, and the second index the column.
A 3d tensor can be viewed as a RGB image. The standard in PyTorch is to have the channel index first. For instance, a CIFAR10 image is of size $3 \times 32 \times 32$.
A 4d tensor can be seen as a sequence of multi-channel images. For instance, given a mini-batch `batch` of 10 CIFAR10 images is of size $10 \times 3 \times 32 \times 32$,

- the 5th image can be accessed as `batch[4]`;
- the blue channel (3rd) of the 7th image can be accessed with `batch[6, 2]` or `batch[6, 2, :, :]`.

Here are a few examples from the immense 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)[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

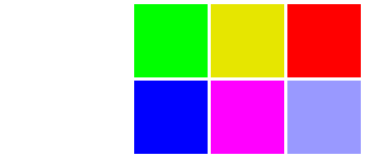
- `torch.abs(input, out=None)`
- `torch.add()`
- `torch.cos(input, out=None)`
- `torch.sigmoid(input, out=None)`

Math reduction

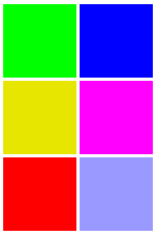
- `torch.dist(input, other, p=2, out=None)`
- `torch.mean()`
- `torch.norm()`
- `torch.std()`
- `torch.sum()`

BLAS and LAPACK Operations

- `torch.linalg(a)`
- `torch.linalg(A, B)`
- `torch.inverse(input, out=None)`
- `torch.mm(mat1, mat2, out=None)`
- `torch.mv(mat, vec, out=None)`

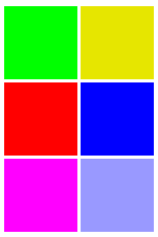


```
x = torch.tensor([ [ 1, 3, 0 ],
                   [ 2, 4, 6 ] ])
```



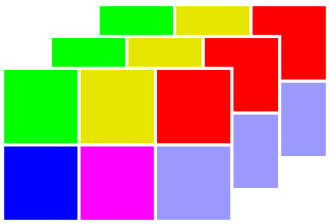
x.t()

x.view(-1)



x.view(3, -1)

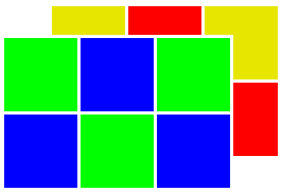
x[:, 1:3]



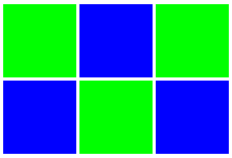
x.view(1, 2, 3).expand(3, 2, 3)

Notes

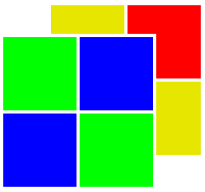
`t()` can be applied to a 2d tensor and simply transpose the indices, as a classical matrix transpose.
`view()` unfolds the tensor in a different shape. Using `-1` for one of the dimension computes the proper value to match the number of coefficients with the original tensor.
Here, `x.view(1, 2, 3).expand(3, 2, 3)` can also be achieved with `x.unsqueeze(0).expand(3, 2, 3)`.
`unsqueeze` adds a dimension of size 1 at the specified position.



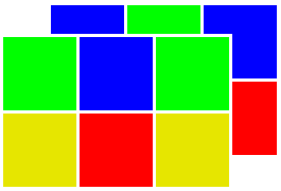
```
x = torch.tensor([ [ [ 1, 2, 1 ],
                     [ 2, 1, 2 ] ],
                   [ [ 3, 0, 3 ],
                     [ 0, 3, 0 ] ] ])
```



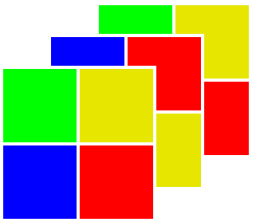
x[0:1, :, :]



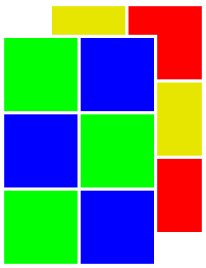
x[:, :, 0:2]



x.transpose(0, 1)



x.transpose(0, 2)



x.transpose(1, 2)

Notes

Transposing two dimensions of a tensor can also be done by specifying the two dimensions as input: `transpose(dim0, dim1)`. This is of course applicable to tensors of greater than two dimensions.

For efficiency reasons, different tensors can share the same data and **modifying one will modify the others**. By default do not make the assumption that two tensors refer to different data in memory.

```
>>> a = torch.full((2, 3), 1)
>>> a
tensor([[1, 1, 1],
        [1, 1, 1]])
>>> b = a.view(-1)
>>> b
tensor([1, 1, 1, 1, 1, 1])
>>> a[1, 1] = 2
>>> a
tensor([[1, 1, 1],
        [1, 2, 1]])
>>> b
tensor([1, 1, 1, 1, 2, 1])
>>> b[0] = 9
>>> a
tensor([[9, 1, 1],
        [1, 2, 1]])
>>> b
tensor([9, 1, 1, 1, 2, 1])
```


Notes

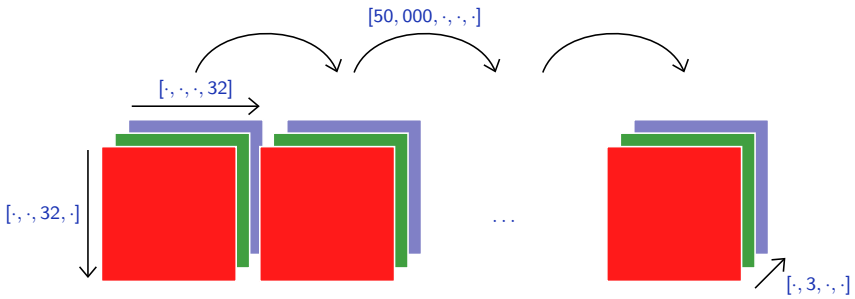
Note that many operations returns a new tensor which shares the same underlying storage as the original tensor, so changing the values of one will change the other as well: `view`, `transpose`, `squeeze`, `unsqueeze`, `expand`, `permute`, etc.
We will come back later to the underlying representation of a tensor that allows that.

PyTorch offers simple interfaces to standard image databases.

```
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
```



Notes

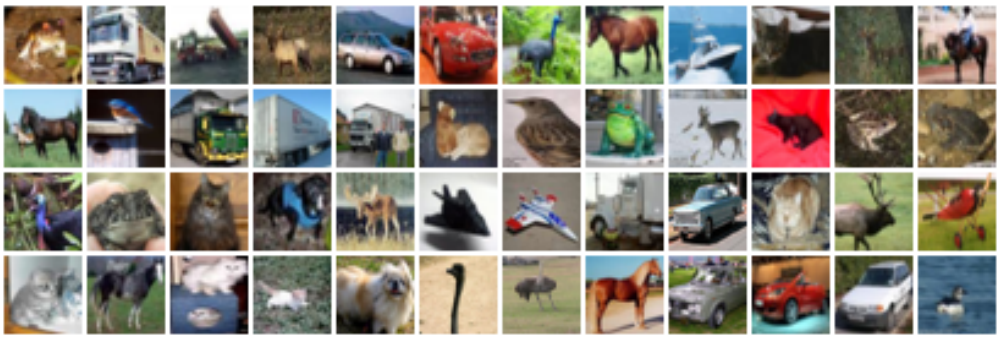
Note that there are different storage conventions between some libraries used by PyTorch (pillow and NumPy) and PyTorch itself:

- loading the images yields a tensor of shape $50000 \times 32 \times 32 \times 3$, but
- PyTorch works with the channel dimension as the second one: $50000 \times 3 \times 32 \times 32$.

This change is made with `permute(0, 3, 1, 2)` which means that we want dimension 3 of the original tensor to lie at the second position of the new tensor.
In the original tensor, accessing pixel $(5, 9)$ of the first image `cifar.data[0, 5, 9]` returns `[122 82 44]`, because the last dimension is the number of channels.
Once the permutation is done, `x[n, 0, :, :]` allows to access the red channel of image n.
If we don't put `float()/255`, then we can have:

- `x[0, 0, 5, 9]` returns `tensor(122, dtype=torch.uint8)`
- `x[0, 1, 5, 9]` returns `tensor(82, dtype=torch.uint8)`
- `x[0, 2, 5, 9]` returns `tensor(44, dtype=torch.uint8)`

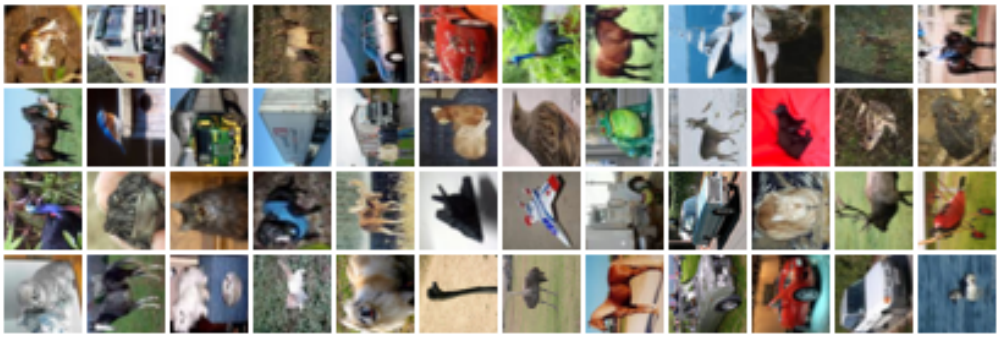
```
# Narrows to the first images, converts to float
x = x[:48]

# Saves these samples as a single image
torchvision.utils.save_image(x, 'cifar-4x12.png',
                             nrow = 12, pad_value = 1.0)
```



Notes

`x[:48]` returns the first 48 images.

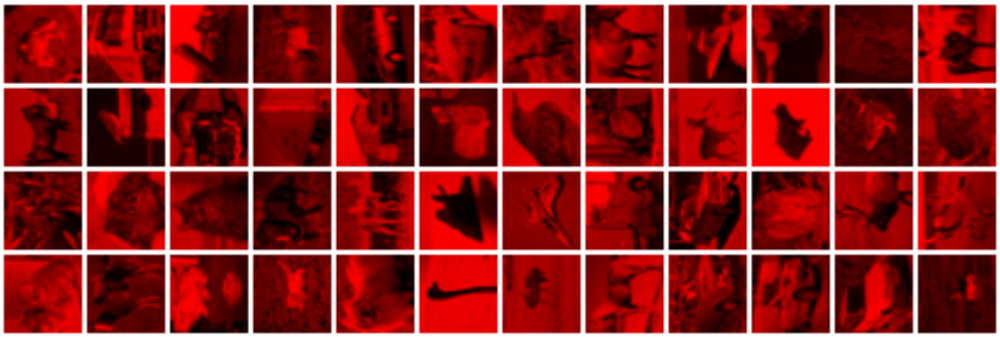
```
# Switches the row and column indexes
x.transpose_(2, 3)
torchvision.utils.save_image(x, 'cifar-4x12-rotated.png',
                             nrow = 12, pad_value = 1.0)
```



Notes

Since the data follows the standard PyTorch "channel first" convention, transposing dimensions 2 and 3 (that is the 3rd and the fourth) exchanges the height and width of the images.
Remember that functions ending with an underscore operate in-place.

```
# 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)
```



Notes

Here, we set all the values of the green and blue channels to zero (channels 1 and 2 respectively).

Broadcasting and Einstein summations

Broadcasting automagically expands dimensions by replicating coefficients, when it is necessary to perform operations that are "intuitively reasonable".

For instance:

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

Notes

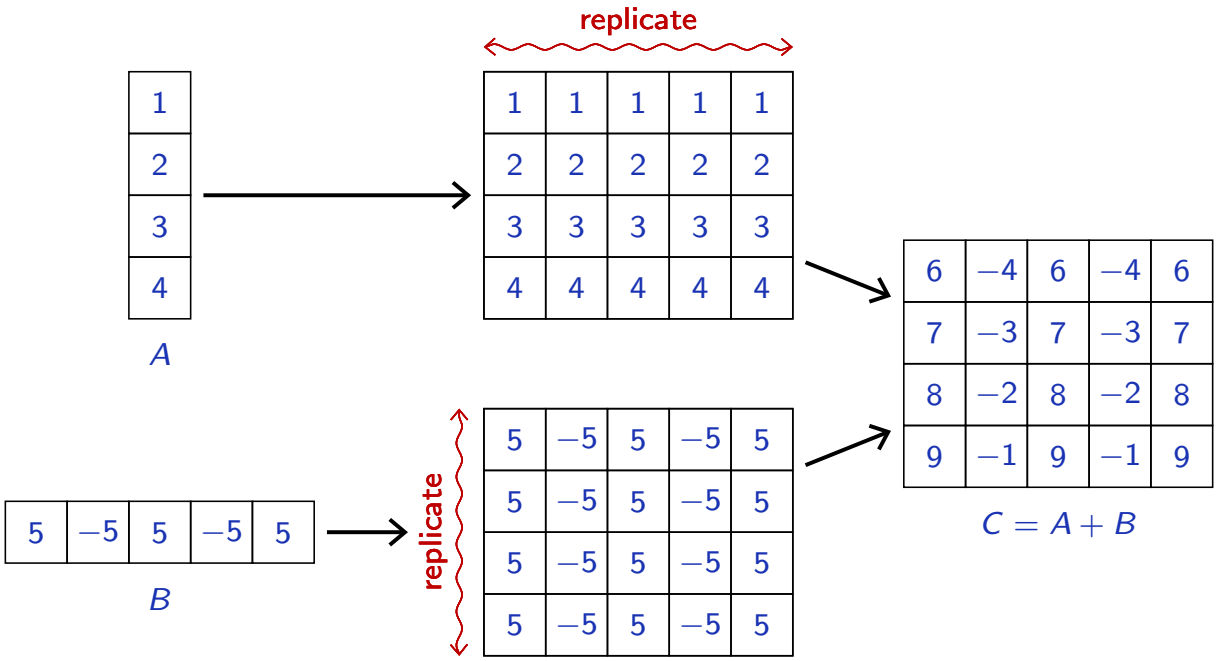
Broadcasting is a mechanism taken from NumPy which expands the proper dimensions of size 1 to perform operations on tensors/arrays of different dimensions.
In the example above, considering that a $N \times D$ tensor is a list of N vectors of dimension D, we want to compute the mean vector. So, here, starting from a tensor of dimension `(100, 4)`, the mean along dimension 0 yields a tensor with 4 values of size `(4,)`, one for each column.
It is quite natural to substract a vector to a *series* of vectors. For instance here, it seems reasonable to subtract the mean vector to all the vectors of `x`, but since the dimensions are respectively `(4,)` and `(100, 4)`, the operation cannot be done.
To allow it, the "broadcasting" mechanism creates [implicitely] a matrix of size `(100, 4)` by replicating the row 100 times.

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
```



Broadcasted

Notes

In the example,

- `A` is of size `(4,1)`
- `B` is of size `(1,5)`

Following the procedure of the previous slide,

1. Both tensors have two dimensions;
2. Then, for each of the two dimensions:
   - On dimension 0, `A` has 4 rows, while `B` has 1. Therefore, `B` is expanded along this dimension by replicating its row 4 times. The "new" `B` is of size `(4,5)`.
   - On dimension 1, `A` has 1 column, while `B` has 5. Therefore, `A` is expanded along this dimension by replicating its column 5 times. The "new" `A` is of size `(4,5)`.
   - The operation can be perform on these two tensors of size `(4,5)`.

Note that all this is transparent and that no copy is actually made.

A powerful generic tool for complex tensorial operations is the **Einstein summation convention**. It provides a concise way of describing dimension re-ordering and summing of component-wise products along some of them.

`torch.einsum` takes as argument a string describing the operation, the tensors to operate on, and returns a tensor.

The operation string is a comma-separated list of indexing, followed by the indexing for the result.

Summations are executed on all indexes not appearing in the result indexing.

For instance, we can formulate that way the standard matrix product:

$$\mathbb{R}^{A\times B} \times \mathbb{R}^{B\times C} \to \mathbb{R}^{A\times C}$$

$$\forall i,k,\ m_{i,k} = \sum_j p_{i,j} q_{j,k}$$

The summation is done along `j` since it does not appear after the `->`.

```
>>> p = torch.rand(2, 5)
>>> q = torch.rand(5, 4)
>>> torch.einsum('ij,jk->ik', p, q)
tensor([[2.0833, 1.1046, 1.5220, 0.4405],
        [2.1338, 1.2601, 1.4226, 0.8641]])
>>> p@q
tensor([[2.0833, 1.1046, 1.5220, 0.4405],
        [2.1338, 1.2601, 1.4226, 0.8641]])
```

Matrix-vector product:

$$\mathbb{R}^{A\times B} \times \mathbb{R}^{B} \to \mathbb{R}^{A}$$

$$\forall i, k,\ w_i = \sum_j m_{i,j} v_j$$

```
w = torch.einsum('ij,j->i', m, v)
```

Hadamard (component-wise) product:

$$\mathbb{R}^{A\times B} \times \mathbb{R}^{A\times B} \to \mathbb{R}^{A\times B}$$

$$\forall i, j,\ m_{i,j} = p_{i,j} q_{i,j}$$

```
m = torch.einsum('ij,ij->ij', p, q)
```

Extracting the diagonal:

$$\mathbb{R}^{D\times D} \to \mathbb{R}^{D}$$

$$\forall i, k,\ v_i = m_{i,i}$$

```
v = torch.einsum('ii->i', m)
```

Batch matrix product:

$$\mathbb{R}^{N\times A\times B} \times \mathbb{R}^{N\times B\times C} \to \mathbb{R}^{N\times A\times C}$$

$$\forall n, i, k,\ \ m_{n,i,k} = \sum_j p_{n,i,j} q_{n,j,k}$$

```
m = torch.einsum('nij,njk->nik', p, q)
```

Batch trace:

$$\mathbb{R}^{N\times D\times D} \to \mathbb{R}^{N}$$

$$\forall n,\ \ t_n = \sum_i m_{n,i,i}$$

```
t = torch.einsum('nii->n', m)
```

Tri-linear product along a channel:

$$\mathbb{R}^{N\times C\times T} \times \mathbb{R}^{N\times C\times T} \times \mathbb{R}^{N\times C\times T} \to \mathbb{R}^{N\times T}$$

$$\forall n, t,\ \ m_{n,t} = \sum_c p_{n,c,t} q_{n,c,t} r_{n,c,t}$$

```
m = torch.einsum('nct,nct,nct->nt', p, q, r)
```