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

\[
x.\text{t()}
\]

\[
x.\text{view}(-1)
\]

\[
x.\text{view}(3, -1)
\]

\[
x[:, 1:3]
\]

\[
x.\text{view}(1, 2, 3).\text{expand}(3, 2, 3)
\]

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

\[
x[0:1, :, :]
\]

\[
x[:, :, 0:2]
\]

\[
x.\text{transpose}(0, 1)
\]

\[
x.\text{transpose}(0, 2)
\]

\[
x.\text{transpose}(1, 2)
\]
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())
```

prints

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

![Diagram of tensors and images](image)

# Narrows to the first images, converts to float

```python
x = x[:48]
```

# Saves these samples as a single image

```python
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 Einstein summations**

**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, but it does!
>>> 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.

$A = \text{torch.tensor([[1.], [2.], [3.], [4.]]})$
$B = \text{torch.tensor([[5., -5., 5., -5., 5.]]})$
$C = A + B$

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 \\
4 & 4 & 4 & 4 \\
\end{array}
\]
\[
\begin{array}{cccc}
5 & -5 & 5 & -5 \\
5 & -5 & 5 & -5 \\
5 & -5 & 5 & -5 \\
5 & -5 & 5 & -5 \\
\end{array}
\]

Broadcasted
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:

\[
\mathbb{R}^{A \times B} \times \mathbb{R}^{B \times C} \rightarrow \mathbb{R}^{A \times C}
\]

\[\forall i, k, \quad m_{i,k} = \sum_j p_{i,j} q_{j,k}\]

\[m = \text{torch.einsum}('ij,jk->ik', p, q)\]

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

```python
>>> 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]])
```

```python
>>> p@q
tensor([[2.0833, 1.1046, 1.5220, 0.4405],
        [2.1338, 1.2601, 1.4226, 0.8641]])
```
Matrix-vector product:

\[ \mathbb{R}^{A \times B} \times \mathbb{R}^B \rightarrow \mathbb{R}^A \]

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

\[ w = \text{torch.einsum}'ij,j->i', m, v] \]

Hadamard (component-wise) product:

\[ \mathbb{R}^{A \times B} \times \mathbb{R}^{A \times B} \rightarrow \mathbb{R}^{A \times B} \]

\[ \forall i, j, m_{i,j} = p_{i,j} q_{i,j} \]

\[ m = \text{torch.einsum}'ij,ij->ij', p, q] \]

Extracting the diagonal:

\[ \mathbb{R}^{D \times D} \rightarrow \mathbb{R}^D \]

\[ \forall i, k, v_i = m_{i,i} \]

\[ v = \text{torch.einsum}'ii->i', m] \]

Batch matrix product:

\[ \mathbb{R}^{N \times A \times B} \times \mathbb{R}^{N \times B \times C} \rightarrow \mathbb{R}^{N \times A \times C} \]

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

\[ m = \text{torch.einsum}'nij,njk->nik', p, q] \]

Batch trace:

\[ \mathbb{R}^{N \times D \times D} \rightarrow \mathbb{R}^N \]

\[ \forall n, t_n = \sum_i m_{n,i,i} \]

\[ t = \text{torch.einsum}'nii->n', m] \]

Tri-linear product along a channel:

\[ \mathbb{R}^{N \times C \times T} \times \mathbb{R}^{N \times C \times T} \times \mathbb{R}^{N \times C \times T} \rightarrow \mathbb{R}^{N \times T} \]

\[ \forall n, t, m_{n,t} = \sum_c p_{n,c,t} q_{n,c,t} f_{n,c,t} \]

\[ m = \text{torch.einsum}'nct,nct,nct->nt', p, q, r] \]