Deep learning

1.6. Tensor internals

François Fleuret
https://fleuret.org/dlc/
A tensor is a view of a [part of a] storage, which is a low-level 1d vector.

```python
>>> x = torch.zeros(2, 4)
>>> x.storage()
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
  0.0
[torch.FloatStorage of size 8]
>>> q = x.storage()
>>> q[4] = 1.0
>>> x
  tensor([[ 0., 0., 0., 0.],
           [ 1., 0., 0., 0.]]
```

Notes

A storage is a series of values contiguous in memory and can be seen as a low-level 1d vector, and each tensor has a storage attached to it.

As we have previously seen before, some functions return a tensor sharing the same underlying storage with an argument, and modifying one of this tensors will also modify the other one.
The first coefficient of a tensor is the one at `storage_offset()` in `storage()`.

Incrementing index $k$ by 1 move by `stride(k)` elements in the storage.

```python
>>> q = torch.arange(0., 20.).storage()
>>> x = torch.empty(0).set_(q, storage_offset = 5, size = (3, 2), stride = (4, 1))
>>> x
tensor([[ 5.,  6.],
        [ 9., 10.],
        [13., 14.]])
```

### Notes

The offset is the location of the first coefficient of the tensor in the storage.

The stride is the number of memory elements that should be skipped to go one element the following one in that dimension.

On the illustration, tensor $x$ starts at element 5 of the storage.

We move along dimension 1 by jumping of 1 element in memory:

- $x[1, 1]$ is “1 element after” $x[0, 0]$;
- $x[2, 1]$ is “1 element after” $x[1, 0]$;
- $x[1, 2]$ is “1 element after” $x[0, 1]$;
- etc.

We move along dimension 0 by jumping of 4 elements in memory:

- $x[0, 1]$ is “4 elements after” $x[0, 0]$;
- $x[2, 1]$ is “4 elements after” $x[1, 0]$;
- etc.
We can explicitly create different “views” of the same storage

```python
>>> n = torch.linspace(1, 4, 4)
>>> n
tensor([ 1., 2., 3., 4.])
>>> torch.tensor(0.).set_(n.storage(), 1, (3, 3), (0, 1))
tensor([[ 2., 3., 4.],
        [ 2., 3., 4.],
        [ 2., 3., 4.]])
>>> torch.tensor(0.).set_(n.storage(), 1, (2, 4), (1, 0))
tensor([[ 2., 2., 2., 2.],
        [ 3., 3., 3., 3.]])
```

This is in particular how transpositions and broadcasting are implemented.

```python
>>> x = torch.empty(100, 100)
>>> x.stride()
(100, 1)
>>> y = x.t()
>>> y.stride()
(1, 100)
```

---

Notes

Replication is achieved with a stride of 0.

The main idea of functions like `view`, `narrow`, `transpose`, etc. and of operations involving broadcasting is to never replicate data in memory, but to “play” with the offsets and strides of the underlying storage.
This organization explains the following (maybe surprising) error

```python
>>> x = torch.empty(100, 100)
>>> x.t().view(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: invalid argument 2: view size is not compatible with
input tensor's size and stride (at least one dimension spans across
two contiguous subspaces). Call .contiguous() before .view()
```

`x.t()` shares `x`'s storage and cannot be “flattened” to 1d.

This can be fixed with `contiguous()`, which returns a contiguous version of the tensor, making a copy if needed.

The function `reshape()` combines `view()` and `contiguous()`. 