Tensors

The term "tensor" has become deeply embedded in machine learning at least in part because of the famous package `tensorflow`, but what is a tensor?

Tensors are a multi-dimensional generalisation of vectors and matrices. A vector is 1D, a matrix 2D and so on.

There is a problem of terminology though. We say a vector of length d lives in $\R^d$, which is a d-dimensional space, so I don't want to use the term *dimension* again. Sometimes people use the term *rank* but that has a specific meaning for matrices, and hence we will avoid it as well. So let's use the term *order*.

So

- a scalar (a single number) is an *order-0* tensor.
- a vector is an *order-1* tensor.
- a matrix is an *order-2* tensor.

And we can increase the order as much as we like. Much as we might say A_{ij} is the (i, j)th element of a matrix, we could have A_{ijk} as the (i, j, k)th element of a order-3 tensor.

Most people don't use the term tensor unless the order is 3 or more.

We can talk about the *shape* or *size* of a tensor just as we could with a matrix. An $n \times m$ matrix has n rows and m columns and we might describe its size by the tuple (n, m). An order-3 tensor might have the size $n \times m \times k$, but its harder to talk about rows and columns now.

A simple example is a colour image. An image consists of $n \times m$ pixels. However in colour images, each pixel contains (at least) three values (typically red-green-blue intensities though there are other representations), so a colour image is typically a $n \times m \times 3$ tensor (or it might be a $3 \times n \times m$ tensor).

Lots of other data are naturally represented by tensors.

Another use is when your data (for instance images) are matrices (or tensors), but you have a set of k of them. Then you can store them all in a higher-order tensor.

Notation and conventions

We could just write T for a tensor, but that wouldn't be quite informative enough. We can't see its order.

Often we specify subscripts for the tensor to provide that information so $T_{\mu\nu\sigma}$ is an order-3 tensor. The indexes here are not referring to the particular indexed item, but are an abstract indication of the order of the tensor.

However, we can extend the idea of "rows" and "columns" to higher order by using subscripts and superscripts. We would write:

- a row vector as T_μ;
- a column vector as T^μ; and

- a (conventional) matrix as T^{ν}_{μ}.

This is useful in understanding how to combine (operate on) pairs of tensors. For instance, to add two tensors they should have compatible shape as indicated by the same number of super- and subscripts.

Thus although T_{ij} and T^{j}_{i} are of the same order, they are incompatible and we shouldn't add them.

This type of convention is often missing from computer packages that just treat each index the same way, but you might see it used paarticularly by physicists, so I wanted to introduce it.

Subtensors

We can generalise the idea that a row or column of a matrix is a vector.

I might write something like A_{i**} to mean the ith "row" of the order-3 tensor, which will be a matrix. Here I am using $*$ as a 'wild-card' that can stand for any element.

Or I could write A_{i*j} to extract a vector from the tensor.

I would call these *subtensors* of the tensor.

You can do these in some languages (Matlab and Python) using the `':'` as the wildcard.

In some Python packages, *e.g.,* tensorflow, these are supported through slices, but these are more general.

Slices

https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/tensor_slicing.ipynb

Transposition

Transposition of a matrix or vector is a standard operation in linear algebra that swaps rows for columns, e.g., $[A^T]_{ij} = A_{ji}$.

The concept could be generalised in various ways, but in `pytorch`, it just means swapping two indexes. So we specify, for instance in a matrix, that the 1st and 2nd indexes are swapped, *i.e.,* that $i \leftrightarrow j$.

We might write swapping the 1st and 2nd indices explicitly as $A^{T(1,2)}$ (but remember Python indexes start at 0 when writing code).

Other matrix operations such as determinants and so on can also be defined, but I won't go into details here.

Addition and subtraction

Adding and subtracting tensors generalises the standard matrix and vector element-wise addition operations.

So no surprises except that some computer code allows you do do things like $A + x$, *i.e.,* add a matrix to a vector. Typically this is performed by replicating the scalar into a matrix of the correct size. Technically this is incorrect, but it is an often-useful shorthand.

Multiplication and Einstein summation notation

We know how to multiply two vectors (using an standard dot or inner product) and we write it variously as

$$< \mathbf{u}, \mathbf{v} >= \mathbf{u} \cdot \mathbf{v} = \mathbf{u}^T \mathbf{v} = \sum_i u_i v_i.$$

Likewise matrix multiplication

$$[AB]_{ij} = \sum_k A_{ik} B_{ki}.$$

We can generalize to tensors where this operation is called a *contraction* because the order of the result is smaller (potentially) than the orders of the two inputs.

The core of Einstein summation notation is a generalisation and abbreviation for such operations: if the same symbol appears twice as an super- and subscript, we sum over that index.

So we now write

- a vector product as $X = U_j V^j$;
- a matrix-vector product as $U^i = A^i_j v^j$; and
- a matrix-matrix product as $C^i_j = A^i_k B^k_j$.

In contexts where superscripts aren't used (commonly in computer packages) the summation notation is just indicated by using the same index name, $e.g.$, $C_{ij} = A_{ik} B_{kj}$. Note though that it generalises, so

$$A_{ik} B_{jk} = [AB^T]_{ij}.$$

Implicitly tensors must have compatible shapes (or sizes).

Tensor products

Tensor products are not like standard matrix multiplication (see Kronecker multiplication for something closer). A tensor product takes two tensors and produces a new one $S \otimes T$.

This is very general, for instance, we can get things like

$$[AB]_{ijk\ell} = A_{ij} \otimes B_{k\ell},$$

where the order of the product is the sum of the orders of the two tensors. We don't seem to use this much in machine learning, but it is important to note the difference between this and a contraction.

Other stuff

We can generalise all the usual multi-d calculus to tensors but I won't go into it here.

Cross-products can be represented using tensors as well.

And there is a huge amount of other stuff, but mostly in machine learning, we use and manipulate tensors as a convenient means to contain multidimensional data, so you don't need to know much more for now.

Tensors in Python

Numpy

Tensors are just arrays of numbers with multiple axes. So they are easy to create in Python with `numpy`.

```
import numpy as np
A = np.array([[1, 2], [3, 4], [5, 6]])
print(A)
print('has order %d' %(A.ndim))
print('and has %d elements' %(A.size))
print('and has shape :', A.shape)
```

You use nested brackets to signify the blocks.

Note here that `size` reports the number of elements, not the shape.

You can also input the order of the tensor, even if you are only inputing one "row".

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr.shape)
```

Note that all the sizes except the last are 1.

You can index into the tensor as you might expect (remember Python indexes start at 0):

```
arr[0,0,0,0,3]
```

You can remove parts that have length 1 to obtain a lower-order tensor using `squeeze`, e.g.

```
arr.squeeze()
print(arr)
```

Here's a higher order example:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
print(arr.shape)
```

There is also `ndarray`, which gives you an alternative interface for creating such arrays, e.g.,

```
np.ndarray(shape=(2,2), dtype=float, order='F')
```

Try it!

You can do a *lot* more with tensors in Python. You can `reshape` them (for instance stack a matrix into a vector) or just directly `flatten` into a vector.

Pytorch

I've used `numpy` here, but in machine learning packages tensors are often first-class objects. So in `PyTorch` you have the same constructs with some fancy ways to work with them.

A `PyTorch` tensor is mostly the same as a `numpy` array, but `PyTorch` tensors can be used on a GPU, which makes life much easier for machine learning.

`PyTorch` tensors can hold all the usual numerical types (integers, floating points, or complex numbers) of various sizes.

We can create them directly as before, or from a `numpy` array, or with various other methods.

```
import torch
import math
import numpy as np
A = torch.tensor([[1., -1.], [1., -1.]])
B = torch.tensor(np.array([[1, 2, 3], [4, 5, 6]]))
C = torch.empty(3, 4)
D = torch.zeros(2, 3, 4)
E = torch.ones(1, 1, 2, dtype=torch.int16)
torch.manual_seed(1729)
F = torch.rand(2, 3, dtype=torch.float64)
```

Many of the data sets that come with the package will be automatically loaded as tensors.

You can index into these as if they were standard arrays and do many of the same operations, *e.g.,*

- `torch.shape`
- `torch.squeeze`

Most standard math works, and you can also do a few extra tricks like `torch.zeros(2, 2) + 1`, *i.e.,* we have added a scalar to a matrix (by replicating the scalar into a matrix of the same size). In general this is termed *broadcasting* and it follows a set of fairly complex rules that I won't describe here. Frankly, I think code is better (easier to read and more likely to avoid bugs) if you avoid using too many short cuts.

There are two forms of multiplication:

- the matrix style is `T @ S = T.matmul(S)`
- elementwise multiplication is `T * S = T.mul(S)`

This is a little weird for me. I wouldn't use either of those symbols that way.

Torch has an Einstein summation function `torch.einsum()`. It's docs are at

https://pytorch.org/docs/stable/generated/torch.einsum.html

Some people love doing their maths this way: https://rockt.github.io/2018/04/30/einsum

One quick example for matrix multiplication is

```
torch.einsum('ik, kj->ij', X, Y)
```

This says that the indexes of the two matrices are named (i, k) and (k, j) and the output should have indices (i, j) so we are implicitly summing over the index k, $i.e.$, we are doing matrix multiplication of X and Y.

It saves writing a big chunk of code like

```
for i in range(X.shape[0]):
    for j in range(Y.shape[1]):
        total = 0
        for k in range(X.shape[1]):
            total += X[i,k] * Y[k,j]
        Z[i,j] = total
```

and its potentially much faster because (hopefully) it is super optimised.

It can also be used for permutation and transposition with `torch.einsum('ij->ji', X)` or

summation of columns with `torch.einsum('ij->j', X)` as well as many other operations with pairs of tensors.

Links

- https://hadrienj.github.io/posts/Deep-Learning-Book-Series-2.1-Scalars-Vectors-Matrices-and-Tensors/
- https://www.w3schools.com/python/numpy/numpy*array*shape.asp
- https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html
- https://www.tensorflow.org/guide/tensor
- https://en.wikipedia.org/wiki/Einstein_notation
- http://www.iaeng.org/publication/WCE2010/WCE2010_pp1824-1828.pdf
- http://www.iaeng.org/publication/WCE2010/WCE2010_pp1838-1841.pdf
- http://www.iaeng.org/publication/WCE2010/WCE2010_pp1829-1833.pdf
- https://pytorch.org/docs/stable/tensors.html
- https://pytorch.org/tutorials/beginner/introyt/tensors*deeper*tutorial.html
- https://pytorch.org/docs/stable/generated/torch.einsum.html
- https://rockt.github.io/2018/04/30/einsum
- https://towardsdatascience.com/einsum-an-underestimated-function-99ca96e2942e