# Einstein Summation Convention in TensorFlow and NumPy

*One of the most useful power tools that many practitioners do not know about.*

Linear algebra is mostly about the properties of linear mappings between vector spaces. We will only consider finite-dimensional vector spaces here.

Introducing bases for these vector spaces, we can describe a linear mapping with a matrix, where the $(j, k)$-entry tells us "how much contribution to the output-space-basis-vector $j$ direction we get from input-space-basis-vector $k$". Matrices carry two indices.

In Machine Learning, just as in Physics, Differential Geometry, and other branches of Mathematics, quantities naturally occur that carry more than two indices, but also have some linear operations defined on them. For example, in Machine Learning, we may have an array of numbers carrying four indices $(b, y, x, c)$ where $(y, x)$ are the row- and column-coordinate of an image, $c$ is the number of the color-channel (such as red/green/blue), and $b$ labels the example in a collection of images (a 'batch') used in a single Machine Learning training step. Now, converting all the images in a batch to grayscale would be an example of a linear transformation from a $(b, y, x, c)$-indexed array to a $(b, y, x)$-indexed array.

Matrices are often thought of as numbers written in some two-dimensional layout, so such an object then could be interpreted as numbers written in some four-dimensional layout. **This way to see higher-rank indexed objects is generally not helpful.**

There is a better mental model that most information technology practitioners are already familiar with, which interprets such a multi-indexed quantity as some specific sort of SQL table. We note that we can also conveniently re-interpret operations such as matrix/vector multiplication, scalar products, or matrix/matrix multiplication in terms of SQL queries. Let us consider a very simple example where we start from a vector $\vec{u} = (u_x, u_y, u_z)$ that measures the position of a toy helicopter relative to the controller in miles for the horizontal $(u_x, u_y)$-cordinates, and in feet for the $u_z$ altitude-coordinate. We can convert this to a metric vector $\vec{m} = (m_x, m_y, m_z)$ that measures everything in meters by using a simple linear mapping - in this toy example even with a diagonal matrix. In normal linear algebra notation, this would read as:

$$\vec{m} = M \cdot \vec{u}, \quad \text{or:} \quad m_j = \sum_{k=0}^{2} M_{jk} u_k$$

with matrix $M = \text{diag}(1609.34, 1609.34, 0.3048)$.

Translated to SQL, this would roughly read as follows (this actually is an executable example that can be copy-pasted into a unix shell, if sqlite3 is installed). While SQL allows us to express this as well as other (more complex) such operations in a nice and orderly fashion as its relational model nicely matches the coordinate approach to tensor arithmetics, it also is a bit verbose, so this is quite a mouthful. Let us hence pay attention to the part that matters, which is the particular combination of SUM/WHERE/GROUP BY.

```
sqlite3 <<EOM

create table ImperialDistanceVector (Idx int, Value float);
create table MetricDistanceVector (Idx int, Value float);
create table MetricFromImperial (IdxMetric int, IdxImperial int, Value float);

-- Populate the Metric-From-Imperial constant table.
insert into MetricFromImperial (IdxMetric, IdxImperial, value)
values (0, 0, 1609.34), (1, 1, 1609.34), (2, 2, 0.3048);

-- Matrix-Vector Multiplication as a SQL statement.
select
  MetricFromImperial.IdxMetric,
  sum(MetricFromImperial.Value * VecImperial.Value)
from
  MetricFromImperial,
```

```
  -- An In-Place vector.
  (select Idx, value from ImperialDistanceVector  -- hack to get column names.
   union
   select * from (values (0, 5.0), (1, 10.0), (2, 100.0))
   ) as VecImperial
where
  VecImperial.Idx = MetricFromImperial.IdxImperial
group by MetricFromImperial.IdxMetric;

EOM

# Produces:
#
# 0|8046.7
# 1|16093.4
# 2|30.48
```

Thinking about multi-index arrays as SQL relational tables is more helpful for developing an intuition about 'multilinear algebra' than trying to visualize some (say) 4-dimensional coordinate-scheme.

Let us stick with the 4-indices (or, 'rank-4') example of a batch of RGB images of the same size. How would we map every image to grayscale? We would somehow use a vector of brightness-values for red, green, and blue to do a weighted-sum over the last index. Specifically, we have: `gray_value = 0.2989 * red_value + 0.5870 * green_value + 0.1140 * blue_value`.

While there is the convenience function `tf.dot()` / `numpy.dot()` for this sort of operation, let us for now see how this looks like with a more general approach. This is roughly how this operation might be translated to SQL form:

```
sqlite3 <<EOM
create table ImageBatch (NumImage int,
                         YCoord int,
                         XCoord int,
                         Color int,
                         Value float);
create table ColorBrightness (Color int, Value int);

insert into ColorBrightness (Color, Value)
values (0, 0.2989), (1, 0.5870), (2, 0.1140);

-- Demo Image 0 has 2 pixels, one orange at (y=10, x=20),
-- and one red at (y=10, x=25).
insert into ImageBatch (NumImage, YCoord, XCoord, Color, Value)
values (0, 10, 20, 0, 0.5), (0, 10, 20, 1, 0.5),  -- Orange pixel.
       (0, 10, 25, 0, 1.0);  -- Red pixel.

select NumImage, YCoord, XCoord, sum(ImageBatch.Value * ColorBrightness.Value)
from ImageBatch, ColorBrightness
where ColorBrightness.Color = ImageBatch.Color
group by NumImage, YCoord, XCoord;

EOM
# Produces:
# 0|10|20|0.44295
# 0|10|25|0.2989
```

Of course, we are only using SQL as a mental model here. Doing multilinear operations like this in SQL would be prohibitively inefficient.

Again, such multilinear-operations-as-a-SQL-statement look quite verbose, but they follow a simple pattern. Let

us focus on the case only where we combine two tables. Each table has some index-columns and a `Value`-column carrying a floating point number. We have some list of output-columns that we GROUP BY, and for the given output-column-index-values, we identify all elements that have matching indexing, and then sum over all the table-entry combinations that contribute 'in a suitable way', where 'summing in a suitable way' is done by SUM() and a WHERE() clause that specifies how the not-output-column-indices have to be matched up.

Can we make this less verbose? If everything follows this pattern, and we work with multi-index arrays that do not have "column names", we first have to introduce names for their indices. Then, we have to say what to sum over, and also what the output-indices are, and what their order is. Every index that is not an output-index then is an index that gets summed over.

The current approach used by `tf.einsum()` and `numpy.einsum()` is:

- Every index gets represented by a letter.
- An 'arrow' `->` separates the (comma-separated) index-groups for the input-array(s) to the left of the array from the output-array index-group to the right.
- For each index-letter, we iterate over all possible values. For each particular assignment of indices to index-letters that we see in this iteration, the product of the input tensor's values (if there is more than one input tensor) for the particular index-combination gets added into the (accumulator-)bucket(s) specified by the index-combination to the right of the '->' arrow.
- Any index-letter can show up multiple times on the input side, but at most once on the output side.
- (Note that a tensor with 0 indices has 1 value, since $N^0 = 1$.)

The name 'einsum' refers to the Einstein Summation Convention, which, however, is actually somewhat different from what happens here. According to the Einstein Summation Convention, each index that gets summed over has to occur exactly twice, and in some disciplines such as Relativity, there are additional rules about index-placement. These extra rules make sure that every properly-written formula "makes sense relativistically", that is, every observer sees the same physical laws. Quite often, we want to do some "generalized Einstein summation" for which there are no restrictions that would correspond to this requirement.

So, we could express the batched-to-grayscale conversion as:

```
batched_grayscale_images = tf.einsum('byxc,c->byx',
                                     batched_images,
                                     rgb_brightness)
```

Or, when working with `numpy` arrays, using `numpy.einsum()` in place of `tf.einsum()`.

Let us look at some common (multi)linear algebra operations:

- Matrix-Vector Multiplication: `tf.einsum('ij,j->i', mat, vec)`
- Matrix-Transposed-times-Vector: `tf.einsum('ji,j->i', mat, vec)`
- Scalar Product: `tf.einsum('i,i->', vec1, vec2)`
- Matrix Multiplication: `tf.einsum('ij,jk->ik', mat1, mat2)`
- Batched-matrix times matrix: `tf.einsum('bij,jk->bik', batched_mat, mat)`
- Batched-matrix times batched-matrix: `tf.einsum('bij,bjk->bik', batched_mat1, batched_mat2)`
- Matrix trace: `tf.einsum('ii,->', mat)`
- Batched-matrix trace: `tf.einsum('bii,->b', mat)`
- Matrix-times-matrix-transposed: `tf.einsum('ij,kj->ik', mat1, mat2)`
- Batched matrix L2 norm-squared: `tf.einsum('bij,bij->b', mat1, mat1)`
- 3-dimensional determinant: `tf.einsum('ijk,i,j,k->', eps3d, v1, v2, v3)`

The last example shows that we can actually take products of more than two arrays. Here, `eps3d` would be a 3-index ('rank-3') array which has entries +1 whenever indices (i,j,k) are a cyclic permutation of (0,1,2) and entries -1 if (i,j,k) are a cyclic permutation of (2,1,0). This way of computing determinants likely is not efficient in multiple ways. Note in particular that, with more than two factors, einsum would have to solve a 'SQL query planning' problem to do the summation efficiently - which as far as I am aware it does not do yet. In the practical examples, we will only consider products involving two objects.