Understanding NumPy's einsum

Asked 8 years, 6 months ago Modified 8 months ago Viewed 143k times

336

How does `np.einsum` work?

Given arrays `A` and `B`, their matrix multiplication followed by transpose is computed using `(A @ B).T`, or equivalently, using:

```
np.einsum("ij, jk -> ki", A, B)
```

python arrays numpy multidimensional-array numpy-einsum

Share Follow

edited Jul 30, 2022 at 8:51

Mateen Ulhaq
23.5k 16 91 132

asked Sep 28, 2014 at 21:33

Lance Strait
3,871 4 17 18

8 Answers

Sorted by: Highest score (default)

644

(Note: this answer is based on a short blog post about `einsum` I wrote a while ago.)

What does `einsum` do?

Imagine that we have two multi-dimensional arrays, `A` and `B`. Now let's suppose we want to...

- *multiply* `A` with `B` in a particular way to create new array of products; and then maybe
- *sum* this new array along particular axes; and then maybe
- *transpose* the axes of the new array in a particular order.

+100

There's a good chance that `einsum` will help us do this faster and more memory-efficiently than combinations of the NumPy functions like `multiply`, `sum` and `transpose` will allow.

How does `einsum` work?

Here's a simple (but not completely trivial) example. Take the following two arrays:

```
[ 4,  5,  6,  7],
[ 8,  9, 10, 11]])
```

We will multiply `A` and `B` element-wise and then sum along the rows of the new array. In "normal" NumPy we'd write:

```
>>> (A[:, np.newaxis] * B).sum(axis=1)
array([ 0, 22, 76])
```

So here, the indexing operation on `A` lines up the first axes of the two arrays so that the multiplication can be broadcast. The rows of the array of products are then summed to return the answer.

Now if we wanted to use `einsum` instead, we could write:

```
>>> np.einsum('i,ij->i', A, B)
array([ 0, 22, 76])
```

The *signature* string `'i,ij->i'` is the key here and needs a little bit of explaining. You can think of it in two halves. On the left-hand side (left of the `->`) we've labelled the two input arrays. To the right of `->`, we've labelled the array we want to end up with.

Here is what happens next:

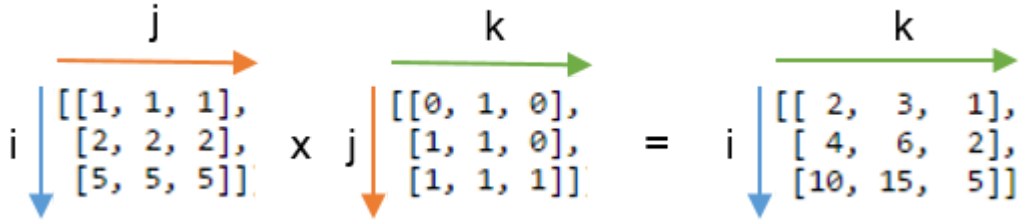
- `A` has one axis; we've labelled it `i`. And `B` has two axes; we've labelled axis 0 as `i` and axis 1 as `j`.
- By **repeating** the label `i` in both input arrays, we are telling `einsum` that these two axes should be **multiplied** together. In other words, we're multiplying array `A` with each column of array `B`, just like `A[:, np.newaxis] * B` does.
- Notice that `j` does not appear as a label in our desired output; we've just used `i` (we want to end up with a 1D array). By **omitting** the label, we're telling `einsum` to **sum** along this axis. In other words, we're summing the rows of the products, just like `.sum(axis=1)` does.

That's basically all you need to know to use `einsum`. It helps to play about a little; if we leave both labels in the output, `'i,ij->ij'`, we get back a 2D array of products (same as `A[:, np.newaxis] * B`). If we say no output labels, `'i,ij->`, we get back a single number (same as doing `(A[:, np.newaxis] * B).sum()`).

The great thing about `einsum` however, is that it does not build a temporary array of products first; it just sums the products as it goes. This can lead to big savings in memory use.

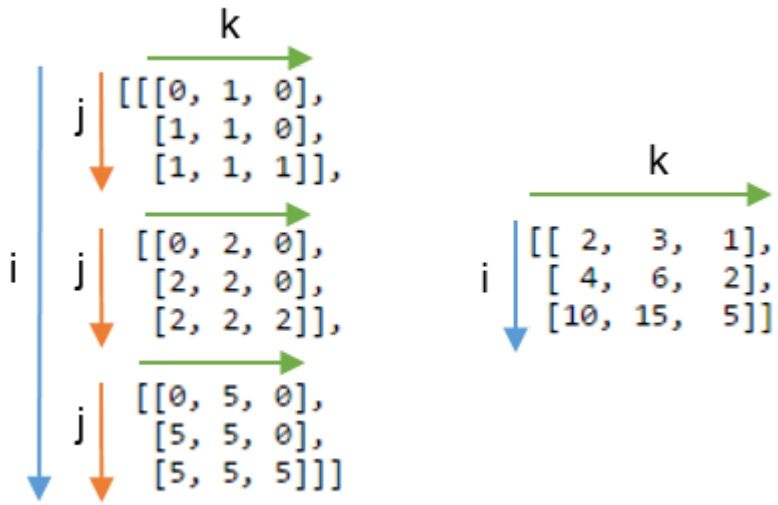
```
A = array([[1, 1, 1],
           [2, 2, 2],
           [5, 5, 5]])

B = array([[0, 1, 0],
           [1, 1, 0],
           [1, 1, 1]])
```

We will compute the dot product using `np.einsum('ij,jk->ik', A, B)`. Here's a picture showing the labelling of the `A` and `B` and the output array that we get from the function:



You can see that label `j` is repeated - this means we're multiplying the rows of `A` with the columns of `B`. Furthermore, the label `j` is not included in the output - we're summing these products. Labels `i` and `k` are kept for the output, so we get back a 2D array.

It might be even clearer to compare this result with the array where the label `j` is *not* summed. Below, on the left you can see the 3D array that results from writing `np.einsum('ij,jk->ijk', A, B)` (i.e. we've kept label `j`):



Summing axis `j` gives the expected dot product, shown on the right.

Some exercises

Let A and B be two 1D arrays with the same length. For example, `A = np.arange(10)` and `B = np.arange(5, 15)`.

- The sum of `A` can be written:

```
np.einsum('i->', A)
```

- Element-wise multiplication, `A * B`, can be written:

```
np.einsum('i,i->i', A, B)
```

- The inner product or dot product, `np.inner(A, B)` or `np.dot(A, B)`, can be written:

```
np.einsum('i,i->', A, B) # or just use 'i,i'
```

- The outer product, `np.outer(A, B)`, can be written:

```
np.einsum('i,j->ij', A, B)
```

For 2D arrays, `C` and `D`, provided that the axes are compatible lengths (both the same length or one of them of has length 1), here are a few examples:

- The trace of `C` (sum of main diagonal), `np.trace(C)`, can be written:

```
np.einsum('ii', C)
```

- Element-wise multiplication of `C` and the transpose of `D`, `C * D.T`, can be written:

```
np.einsum('ij,ji->ij', C, D)
```

- Multiplying each element of `C` by the array `D` (to make a 4D array), `C[:, :, None, None] * D`, can be written:

```
np.einsum('ij,kl->ijkl', C, D)
```

Share Follow

edited Aug 15, 2021 at 13:06

Ivan
32.9k 7 50 94

answered Nov 10, 2015 at 23:10

Alex Riley
166k 45 260 236

4 Very nice explanation, thanks. "Notice that i does not appear as a label in our desired output"-- doesn't it? – Ian Hincks Sep 7, 2016 at 20:14

1 Thanks @IanHincks! That looks like a typo; I've corrected it now. – Alex Riley Sep 18, 2016 at 13:40

2 Very good answer. It's also worth noting that `ij,jk` could work by itself (without the arrows) to form the matrix multiplication. But it seems like for clarity it's best to put the arrows and then the output dimensions

2 @Peaceful: this is one of those occasions where it's difficult to choose the right word! I feel "column" fits a bit better here since `A` is of length 3, the same as the length of the columns in `B` (whereas rows of `B` have length 4 and cannot be multiplied element-wise by `A`). – Alex Riley Jan 15, 2017 at 10:47

4 Note that omitting the `->` affects the semantics: "In implicit mode, the chosen subscripts are important since the axes of the output are reordered alphabetically. This means that `np.einsum('ij', a)` doesn't affect a 2D array, while `np.einsum('ji', a)` takes its transpose." – BallpointBen Dec 31, 2019 at 13:18

82

Grasping the idea of `numpy.einsum()` is very easy if you understand it intuitively. As an example, let's start with a simple description involving *matrix multiplication*.

To use `numpy.einsum()`, all you have to do is to pass the so-called *subscripts string* as an argument, followed by your *input arrays*.

Let's say you have two 2D arrays, `A` and `B`, and you want to do matrix multiplication. So, you do:

```
np.einsum("ij, jk -> ik", A, B)
```

Here the *subscript string* `ij` corresponds to array `A` while the *subscript string* `jk` corresponds to array `B`. Also, the most important thing to note here is that the *number of characters* in each *subscript string* **must** match the dimensions of the array (i.e., two chars for 2D arrays, three chars for 3D arrays, and so on). And if you repeat the chars between *subscript strings* (`j` in our case), then that means you want the `ein` *sum* to happen along those dimensions. Thus, they will be sum-reduced (i.e., that dimension will be *gone*).

The *subscript string* after this `->` symbol represent the dimensions of our resultant array. If you leave it empty, then everything will be summed and a scalar value is returned as the result. Else the resultant array will have dimensions according to the *subscript string*. In our example, it'll be `ik`. This is intuitive because we know that for the matrix multiplication to work, the number of columns in array `A` has to match the number of rows in array `B` which is what is happening here (i.e., we encode this knowledge by repeating the char `j` in the *subscript string*)

Here are some more examples illustrating the use/power of `np.einsum()` in implementing some common *tensor* or *nd-array* operations, succinctly.

Inputs

```
# a vector
In [197]: vec
Out[197]: array([0, 1, 2, 3])

# an array
```

```
       [21, 22, 23, 24],
       [31, 32, 33, 34],
       [41, 42, 43, 44]])

# another array
In [199]: B
Out[199]:
array([[1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3],
       [4, 4, 4, 4]])
```

1) Matrix multiplication (similar to `np.matmul(arr1, arr2)`)

```
In [200]: np.einsum("ij, jk -> ik", A, B)
Out[200]:
array([[130, 130, 130, 130],
       [230, 230, 230, 230],
       [330, 330, 330, 330],
       [430, 430, 430, 430]])
```

2) Extract elements along the main-diagonal (similar to `np.diag(arr)`)

```
In [202]: np.einsum("ii -> i", A)
Out[202]: array([11, 22, 33, 44])
```

3) Hadamard product (i.e. element-wise product of two arrays) (similar to `arr1 * arr2`)

```
In [203]: np.einsum("ij, ij -> ij", A, B)
Out[203]:
array([[ 11,  12,  13,  14],
       [ 42,  44,  46,  48],
       [ 93,  96,  99, 102],
       [164, 168, 172, 176]])
```

4) Element-wise squaring (similar to `np.square(arr)` or `arr ** 2`)

```
In [210]: np.einsum("ij, ij -> ij", B, B)
Out[210]:
array([[ 1,  1,  1,  1],
       [ 4,  4,  4,  4],
       [ 9,  9,  9,  9],
       [16, 16, 16, 16]])
```

5) Trace (i.e. sum of main-diagonal elements) (similar to `np.trace(arr)`)

```
In [217]: np.einsum("ii -> ", A)
Out[217]: 110
```

```
In [221]: np.einsum("ij -> ji", A)
Out[221]:
array([[11, 21, 31, 41],
       [12, 22, 32, 42],
       [13, 23, 33, 43],
       [14, 24, 34, 44]])
```

7) Outer Product (of vectors) (similar to `np.outer(vec1, vec2)`)

```
In [255]: np.einsum("i, j -> ij", vec, vec)
Out[255]:
array([[0, 0, 0, 0],
       [0, 1, 2, 3],
       [0, 2, 4, 6],
       [0, 3, 6, 9]])
```

8) Inner Product (of vectors) (similar to `np.inner(vec1, vec2)`)

```
In [256]: np.einsum("i, i -> ", vec, vec)
Out[256]: 14
```

9) Sum along axis 0 (similar to `np.sum(arr, axis=0)`)

```
In [260]: np.einsum("ij -> j", B)
Out[260]: array([10, 10, 10, 10])
```

10) Sum along axis 1 (similar to `np.sum(arr, axis=1)`)

```
In [261]: np.einsum("ij -> i", B)
Out[261]: array([ 4,  8, 12, 16])
```

11) Batch Matrix Multiplication

```
In [287]: BM = np.stack((A, B), axis=0)

In [288]: BM
Out[288]:
array([[[11, 12, 13, 14],
        [21, 22, 23, 24],
        [31, 32, 33, 34],
        [41, 42, 43, 44]],

       [[ 1,  1,  1,  1],
        [ 2,  2,  2,  2],
        [ 3,  3,  3,  3],
        [ 4,  4,  4,  4]]])
```

```
# batch matrix multiply using einsum
In [292]: BMM = np.einsum("bij, bjk -> bik", BM, BM)

In [293]: BMM
Out[293]:
array([[[1350, 1400, 1450, 1500],
        [2390, 2480, 2570, 2660],
        [3430, 3560, 3690, 3820],
        [4470, 4640, 4810, 4980]],

       [[  10,   10,   10,   10],
        [  20,   20,   20,   20],
        [  30,   30,   30,   30],
        [  40,   40,   40,   40]]])

In [294]: BMM.shape
Out[294]: (2, 4, 4)
```

12) Sum along axis 2 (similar to `np.sum(arr, axis=2)`)

```
In [330]: np.einsum("ijk -> ij", BM)
Out[330]:
array([[ 50,  90, 130, 170],
       [  4,   8,  12,  16]])
```

13) Sum all the elements in array (similar to `np.sum(arr)`)

```
In [335]: np.einsum("ijk -> ", BM)
Out[335]: 480
```

14) Sum over multiple axes (i.e. marginalization)

(similar to `np.sum(arr, axis=(axis0, axis1, axis2, axis3, axis4, axis6, axis7))`)

```
# 8D array
In [354]: R = np.random.standard_normal((3,5,4,6,8,2,7,9))

# marginalize out axis 5 (i.e. "n" here)
In [363]: esum = np.einsum("ijklmnop -> n", R)

# marginalize out axis 5 (i.e. sum over rest of the axes)
In [364]: nsum = np.sum(R, axis=(0,1,2,3,4,6,7))

In [365]: np.allclose(esum, nsum)
Out[365]: True
```

15) *Double Dot Products* (similar to **np.sum(hadamard-product)** cf. **3**)

```
In [772]: A
```

```
In [773]: B
Out[773]:
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])

In [774]: np.einsum("ij, ij -> ", A, B)
Out[774]: 124
```

16) *2D and 3D array multiplication*

Such a multiplication could be very useful when solving linear system of equations (**Ax = b**) where you want to verify the result.

```
# inputs
In [115]: A = np.random.rand(3,3)
In [116]: b = np.random.rand(3, 4, 5)

# solve for x
In [117]: x = np.linalg.solve(A, b.reshape(b.shape[0], -1)).reshape(b.shape)

# 2D and 3D array multiplication :)
In [118]: Ax = np.einsum('ij, jkl', A, x)

# indeed the same!
In [119]: np.allclose(Ax, b)
Out[119]: True
```

On the contrary, if one has to use `np.matmul()` for this verification, we have to do couple of `reshape` operations to achieve the same result like:

```
# reshape 3D array `x` to 2D, perform matmul
# then reshape the resultant array to 3D
In [123]: Ax_matmul = np.matmul(A, x.reshape(x.shape[0], -1)).reshape(x.shape)

# indeed correct!
In [124]: np.allclose(Ax, Ax_matmul)
Out[124]: True
```

Bonus: Read more math here : Einstein-Summation and definitely here: Tensor-Notation

Share Follow

edited Mar 29, 2022 at 22:33

answered Dec 25, 2017 at 7:04

kmario23

55.7k 13 156 148

1 After understanding einsum technique I went on to write many articles based on it. For reference - towardsdatascience.com/... – MSS Jan 3, 2022 at 8:55

22

When reading einsum equations, I've found it the most helpful to just be able to mentally boil them down to their imperative versions.

Let's start with the following (imposing) statement:

```
C = np.einsum('bhwi,bhwj->bij', A, B)
```

Working through the punctuation first we see that we have two 4-letter comma-separated blobs - `bhwi` and `bhwj`, before the arrow, and a single 3-letter blob `bij` after it. Therefore, the equation produces a rank-3 tensor result from two rank-4 tensor inputs.

Now, let each letter in each blob be the name of a range variable. The position at which the letter appears in the blob is the index of the axis that it ranges over in that tensor. The imperative summation that produces each element of C, therefore, has to start with three nested for loops, one for each index of C.

```
for b in range(...):
    for i in range(...):
        for j in range(...):
            # the variables b, i and j index C in the order of their appearance in the
equation
            C[b, i, j] = ...
```

So, essentially, you have a `for` loop for every output index of C. We'll leave the ranges undetermined for now.

Next we look at the left-hand side - are there any range variables there that *don't* appear on the *right-hand* side? In our case - yes, `h` and `w`. Add an inner nested `for` loop for every such variable:

```
for b in range(...):
    for i in range(...):
        for j in range(...):
            C[b, i, j] = 0
            for h in range(...):
                for w in range(...):
                    ...
```

Inside the innermost loop we now have all indices defined, so we can write the actual summation and the translation is complete:

```
# three nested for-loops that index the elements of C
for b in range(...):
    for i in range(...):
```

```
            C[b, i, j] = 0

            # two nested for-loops for the two indexes that don't appear on the right-
hand side
            for h in range(...):
                for w in range(...):
                    # Sum! Compare the statement below with the original einsum formula
                    # 'bhwi,bhwj->bij'

                    C[b, i, j] += A[b, h, w, i] * B[b, h, w, j]
```

If you've been able to follow the code thus far, then congratulations! This is all you need to be able to read einsum equations. Notice in particular how the original einsum formula maps to the final summation statement in the snippet above. The for-loops and range bounds are just fluff and that final statement is all you really need to understand what's going on.

For the sake of completeness, let's see how to determine the ranges for each range variable. Well, the range of each variable is simply the length of the dimension(s) which it indexes. Obviously, if a variable indexes more than one dimension in one or more tensors, then the lengths of each of those dimensions have to be equal. Here's the code above with the complete ranges:

```
# C's shape is determined by the shapes of the inputs
# b indexes both A and B, so its range can come from either A.shape or B.shape
# i indexes only A, so its range can only come from A.shape, the same is true for j and
B
assert A.shape[0] == B.shape[0]
assert A.shape[1] == B.shape[1]
assert A.shape[2] == B.shape[2]
C = np.zeros((A.shape[0], A.shape[3], B.shape[3]))
for b in range(A.shape[0]): # b indexes both A and B, or B.shape[0], which must be the
same
    for i in range(A.shape[3]):
        for j in range(B.shape[3]):
            # h and w can come from either A or B
            for h in range(A.shape[1]):
                for w in range(A.shape[2]):
                    C[b, i, j] += A[b, h, w, i] * B[b, h, w, j]
```

Share Follow

answered Jan 22, 2020 at 11:35

Stefan Dragnev
13.9k 6 48 52

12

Another view on `np.einsum`

Most answers here explain by example, I thought I'd give an additional point of view.

You can see `einsum` as a generalized matrix summation operator. The string given contains the subscripts which are labels representing axes. I like to think of it as your operation definition. The

2. axis size equality between inputs.

Let's take the initial example: `np.einsum('ij,jk->ki', A, B)`. Here the constraints ***1.*** translates to `A.ndim == 2` and `B.ndim == 2`, and ***2.*** to `A.shape[1] == B.shape[0]`.

As you will see later down, there are other constraints. For instance:

3. labels in the output subscript must not appear more than once.
4. labels in the output subscript must appear in the input subscripts.

When looking at `ij,jk->ki`, you can think of it as:

> **which components from the input arrays will contribute to component `[k, i]` of the output array**.

The subscripts contain the exact definition of the operation for each component of the output array.

We will stick with operation `ij,jk->ki`, and the following definitions of `A` and `B`:

```
>>> A = np.array([[1,4,1,7], [8,1,2,2], [7,4,3,4]])
>>> A.shape
(3, 4)

>>> B = np.array([[2,5], [0,1], [5,7], [9,2]])
>>> B.shape
(4, 2)
```

The output, `Z`, will have a shape of `(B.shape[1], A.shape[0])` and could naively be constructed in the following way. Starting with a blank array for `Z`:

```
Z = np.zeros((B.shape[1], A.shape[0]))
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        for k range(B.shape[0]):
           Z[k, i] += A[i, j]*B[j, k] # ki <- ij*jk
```

`np.einsum` is about accumulating contributions in the output array. Each `(A[i,j], B[j,k])` pair is seen contributing to each `Z[k, i]` component.

You might have noticed, it looks extremely similar to how you would go about computing general matrix multiplications...

Minimal implementation

Here is a minimal implementation of `np.einsum` in Python. This should help understand what is really going on under the hood.

As we go along I will keep referring to the previous example. Defining `inputs` as `[A, B]`.

`np.einsum` can actually take more than two inputs. In the following, we will focus on the general case: *n* inputs and *n* input subscripts. The main goal is to find the domain of iteration, *i.e.* the cartesian product of all our ranges.

We can't rely on manually writing `for` loops, simply because we don't know how many there will be. The main idea is this: we need to find all unique labels (I will use `key` and `keys` to refer to them), find the corresponding array shape, then create ranges for each one, and *compute* the product of the ranges using `itertools.product` to get the domain of study.

index	**keys**	**constraints**	**sizes**	**ranges**
1	`'i'`	`A.shape[0]`	3	`range(0, 3)`
2	`'j'`	`A.shape[1] == B.shape[0]`	4	`range(0, 4)`
0	`'k'`	`B.shape[1]`	2	`range(0, 2)`

The domain of study is the cartesian product: `range(0, 2) x range(0, 3) x range(0, 4)`.

1. Subscripts processing:

```
>>> expr = 'ij,jk->ki'
>>> qry_expr, res_expr = expr.split('->')
>>> inputs_expr = qry_expr.split(',')
>>> inputs_expr, res_expr
(['ij', 'jk'], 'ki')
```

2. Find the unique keys (*labels*) in the input subscripts:

```
>>> keys = set([(key, size) for keys, input in zip(inputs_expr, inputs)
                for key, size in list(zip(keys, input.shape))])
{('i', 3), ('j', 4), ('k', 2)}
```

   We should be checking for constraints (as well as in the output subscript)! Using `set` is a bad idea but it will work for the purpose of this example.

3. Get the associated sizes (used to initialize the output array) and construct the ranges (used to create our domain of iteration):

```
>>> sizes = dict(keys)
{'i': 3, 'j': 4, 'k': 2}

>>> ranges = [range(size) for _, size in keys]
```

4. We need an *list* containing the keys (*labels*):

```
>>> to_key = sizes.keys()
['k', 'i', 'j']
```

5. Compute the cartesian product of the `range`s

```
>>> domain = product(*ranges)
```

   Note: `[itertools.product][1]` returns an iterator which gets *consumed* over time.

6. Initialize the output tensor as:

```
>>> res = np.zeros([sizes[key] for key in res_expr])
```

7. We will be looping over `domain`:

```
>>> for indices in domain:
...     pass
```

   For each iteration, `indices` will contain the values on each axis. In our example, that would provide `i`, `j`, and `k` as a *tuple*: `(k, i, j)`. For each input (`A` and `B`) we need to determine which component to fetch. That's `A[i, j]` and `B[j, k]`, yes! However, we don't have variables `i`, `j`, and `k`, literally speaking.

   We can zip `indices` with `to_key` to create a mapping between each key (*label*) and its current value:

```
>>> vals = dict(zip(to_key, indices))
```

   To get the coordinates for the output array, we use `vals` and loop over the keys: `[vals[key] for key in res_expr]`. However, to use these to index the output array, we need to wrap it with `tuple` and `zip` to separate the indices along each axis:

```
>>> res_ind = tuple(zip([vals[key] for key in res_expr]))
```

   Same for the input indices (although there can be several):

```
>>> inputs_ind = [tuple(zip([vals[key] for key in expr])) for expr in inputs_expr]
```

8. We will use a `itertools.reduce` to compute the product of all contributing components:

```
>>> def reduce_mult(L):
...     return reduce(lambda x, y: x*y, L)
```

9. Overall the loop over the domain looks like:

```
...                     for expr in inputs_expr]
...
...     res[res_ind] += reduce_mult([M[i] for M, i in zip(inputs, inputs_ind)])
```

```
>>> res
array([[70., 44., 65.],
       [30., 59., 68.]])
```

That's pretty close to what `np.einsum('ij,jk->ki', A, B)` returns!

Share Follow

edited Jul 1, 2021 at 22:23

answered Feb 2, 2021 at 9:53

Ivan
32.9k 7 50 94

3 Great effort, do you know btw, were the "ein" term came from? – prosti Feb 2, 2021 at 17:06

3 Thank you! It's from Albert ***Ein***stein himself! – Ivan Feb 2, 2021 at 18:26

8

I found NumPy: The tricks of the trade (Part II) instructive

> We use -> to indicate the order of the output array. So think of 'ij, i->j' as having left hand side (LHS) and right hand side (RHS). Any repetition of labels on the LHS computes the product element wise and then sums over. By changing the label on the RHS (output) side, we can define the axis in which we want to proceed with respect to the input array, i.e. summation along axis 0, 1 and so on.

```
import numpy as np

>>> a
array([[1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])
>>> b
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> d = np.einsum('ij, jk->ki', a, b)
```

Notice there are three axes, i, j, k, and that j is repeated (on the left-hand-side). `i,j` represent rows and columns for `a`. `j,k` for `b`.

In order to calculate the product and align the `j` axis we need to add an axis to `a`. (`b` will be broadcast along(?) the first axis)

```
a[i, j, k]
   b[j, k]

>>> c = a[:,:,np.newaxis] * b
>>> c
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 0,  2,  4],
        [ 6,  8, 10],
        [12, 14, 16]],

       [[ 0,  3,  6],
        [ 9, 12, 15],
        [18, 21, 24]]])
```

`j` is absent from the right-hand-side so we sum over `j` which is the second axis of the 3x3x3 array

```
>>> c = c.sum(1)
>>> c
array([[ 9, 12, 15],
       [18, 24, 30],
       [27, 36, 45]])
```

Finally, the indices are (alphabetically) reversed on the right-hand-side so we transpose.

```
>>> c.T
array([[ 9, 18, 27],
       [12, 24, 36],
       [15, 30, 45]])

>>> np.einsum('ij, jk->ki', a, b)
array([[ 9, 18, 27],
       [12, 24, 36],
       [15, 30, 45]])
>>>
```

Share Follow

edited Nov 24, 2019 at 7:02

answered Sep 29, 2014 at 12:17

wwii

23k 7 37 77

NumPy: The tricks of the trade (Part II) seems to require an invite from the site owner as well as a Wordpress account – Tejas Shetty Nov 24, 2019 at 6:47

Lets make 2 arrays, with different, but compatible dimensions to highlight their interplay

```
array([[0, 1, 2],
       [3, 4, 5]])


In [44]: B=np.arange(12).reshape(3,4)
Out[44]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Your calculation, takes a 'dot' (sum of products) of a (2,3) with a (3,4) to produce a (4,2) array. `i` is the 1st dim of `A`, the last of `C`; `k` the last of `B`, 1st of `C`. `j` is 'consumed' by the summation.

```
In [45]: C=np.einsum('ij,jk->ki',A,B)
Out[45]:
array([[20, 56],
       [23, 68],
       [26, 80],
       [29, 92]])
```

This is the same as `np.dot(A,B).T` - it's the final output that's transposed.

To see more of what happens to `j`, change the `C` subscripts to `ijk`:

```
In [46]: np.einsum('ij,jk->ijk',A,B)
Out[46]:
array([[[ 0,  0,  0,  0],
        [ 4,  5,  6,  7],
        [16, 18, 20, 22]],

       [[ 0,  3,  6,  9],
        [16, 20, 24, 28],
        [40, 45, 50, 55]]])
```

This can also be produced with:

```
A[:,:,None]*B[None,:,:]
```

That is, add a `k` dimension to the end of `A`, and an `i` to the front of `B`, resulting in a (2,3,4) array.

`0 + 4 + 16 = 20`, `9 + 28 + 55 = 92`, etc; Sum on `j` and transpose to get the earlier result:

```
np.sum(A[:,:,None] * B[None,:,:], axis=1).T

# C[k,i] = sum(j) A[i,j (,k) ] * B[(i,)  j,k]
```

217k 14 224 346

1

Once get familiar with the dummy index (the common or repeating index) and the summation along the dummy index in the Einstein Summation (einsum), the output `->` shaping is easy. Hence focus on:

1. Dummy index, the common index `j` in `np.einsum("ij,jk->ki", a, b)`
2. Summation along the dummy index `j`

Dummy index

For `einsum("...", a, b)`, element wise multiplication always happens in-between matrices `a` and `b` regardless there are common indices or not. We can have `einsum('xy,wz', a, b)` which has no common index in the subscripts `'xy,wz'`.

If there is a common index, as `j` in `"ij,jk->ki"`, then it is called a **dummy index** in the Einstein Summation.

- Einstein Summation

> An index that is summed over is a summation index, in this case "i". It is also called a dummy index since any symbol can replace "i" without changing the meaning of the expression provided that it does not collide with index symbols in the same term.

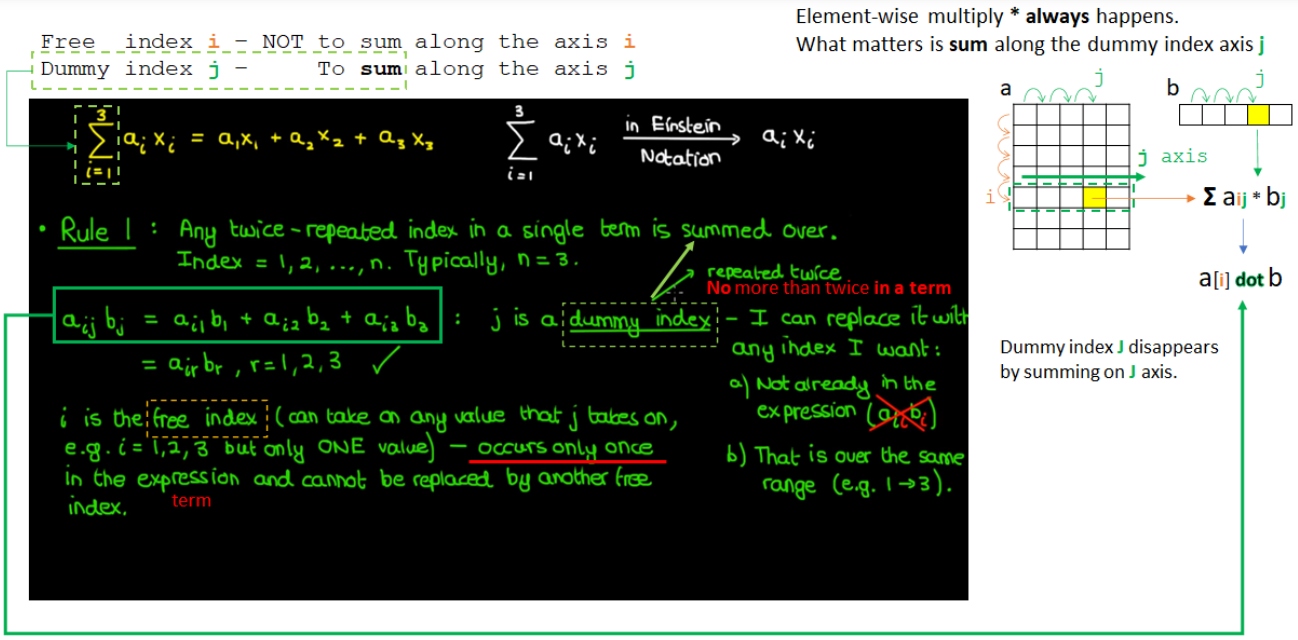
Summation along the dummy index

For `np.einsum("ij,j", a, b)` of the **green rectangle** in the diagram, `j` is the dummy index. The element-wise multiplication `a[i][j] * b[j]` is summed up along the `j` axis as `Σ ( a[i][j] * b[j] )`.

$$\sum_{j}(a_{ij} * b_{j})$$

It is a **dot product** `np.inner(a[i], b)` for each `i`. Here being specific with `np.inner()` and avoiding `np.dot` as it is not strictly a mathematical **dot product** operation.

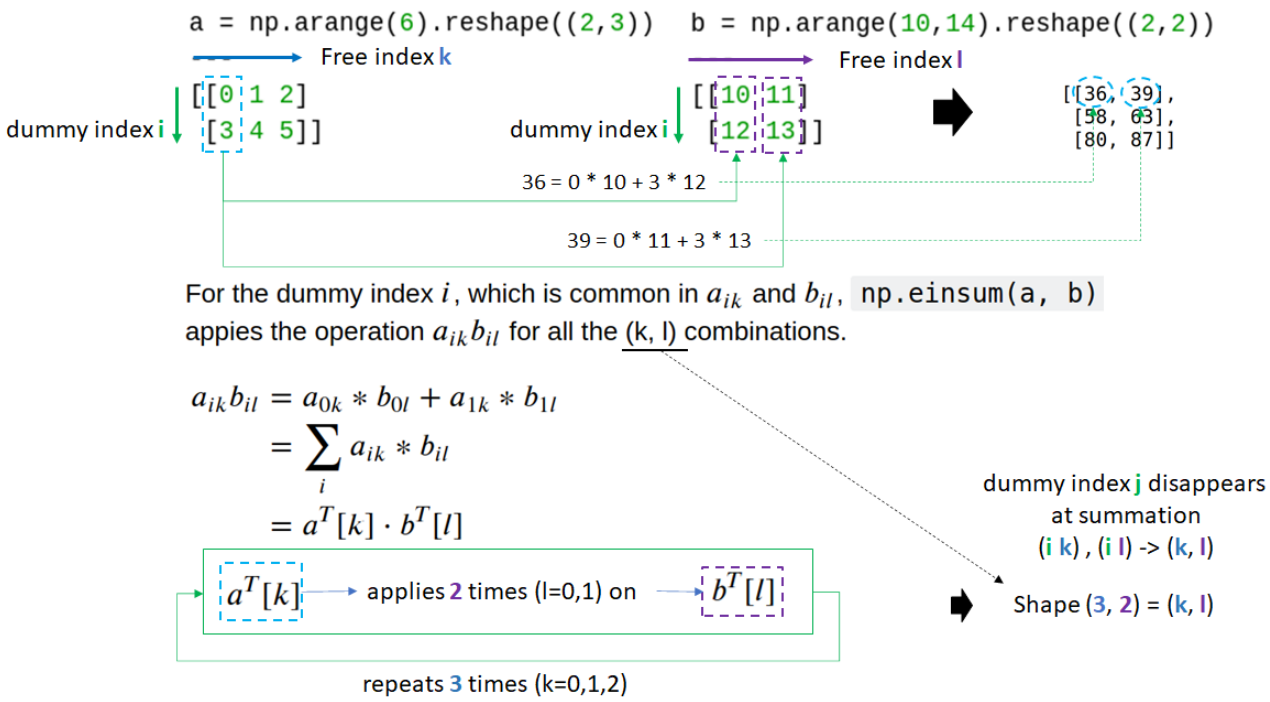
- Einstein Summation Convention: an Introduction



The dummy index can appear anywhere as long as the rules (please see the youtube for details) are met.

For the dummy index `i` in `np.einsum("ik,il", a, b)`, it is a row index of the matrices `a` and `b`, hence a column from `a` and that from `b` are extracted to generate the **dot product**s.

```
c = np.einsum("ik,il", a, b)
```



Output form

Because the summation occurs along the **dummy index**, the dummy index disappears in the

`np.einsum("... -> <shape>")` to specify the output form by the **output subscript labels** with `->` identifier.

See the **explicit mode** in numpy.einsum for details.

> In explicit mode the output can be directly controlled by specifying output subscript labels. This requires the identifier `'->'` as well as the list of output subscript labels. This feature increases the flexibility of the function since summing can be disabled or forced when required. The call `np.einsum('i->', a)` is like `np.sum(a, axis=-1)`, and `np.einsum('ii->i', a)` is like `np.diag(a)`. The difference is that einsum does not allow broadcasting by default. Additionally `np.einsum('ij,jh->ih', a, b)` directly specifies the order of the output subscript labels and therefore returns matrix multiplication, unlike the example above in implicit mode.

Without a dummy index

An example for having no dummy index in the einsum.

1. A term (subscript Indices, e.g. `"ij"`) selects an element in each array.
2. Each left-hand side element is applied on the element on the right-hand side for element-wise multiplication (hence multiplication always happens).

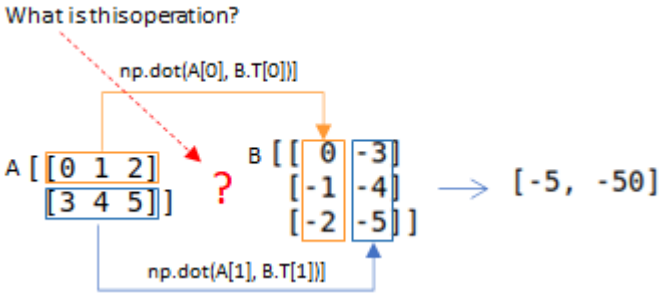
`a` has shape (2,3) each element of which is applied to `b` of shape (2,2). Hence it creates a matrix of shape `(2,3,2,2)` without no summation as `(i,j)`, `(k.l)` are all free indices.

```
# ----------------------------------------------------------------------------
# For np.einsum("ij,kl", a, b)
# 1-1: Term "ij" or (i,j), two free indices, selects selects an element a[i][j].
# 1-2: Term "kl" or (k,l), two free indices, selects selects an element b[k][l].
# 2:   Each a[i][j] is applied on b[k][l] for element-wise multiplication a[i][j] *
b[k,l]
# ----------------------------------------------------------------------------
# for (i,j) in a:
#    for(k,l) in b:
#        a[i][j] * b[k][l]
np.einsum("ij,kl", a, b)

array([[[[ 0,  0],
         [ 0,  0]],

        [[10, 11],
         [12, 13]],

        [[20, 22],
         [24, 26]]],
```

```
       [[40, 44],
        [48, 52]],

       [[50, 55],
        [60, 65]]])
```

Examples

dot products from matrix A rows and matrix B columns




```
A = np.matrix('0 1 2; 3 4 5')
B = np.matrix('0 -3; -1 -4; -2 -5');
np.einsum('ij,ji->i', A, B)

# Same with
np.diagonal(np.matmul(A,B))
(A*B).diagonal()
---
[ -5 -50]
[ -5 -50]
[[ -5 -50]]
```

Share Follow

edited Feb 16, 2021 at 0:44

answered Feb 15, 2021 at 10:39

mon
17.2k 18 102 184

I think the simplest example is in [tensorflow docs](#)

0

There are four steps to convert your equation to einsum notation. Lets take this equation as an example `C[i,k] = sum_j A[i,j] * B[j,k]`

1. First we drop the variable names. We get `ik = sum_j ij * jk`
2. We drop the `sum_j` term as it is implicit. We get `ik = ij * jk`
3. We replace `*` with `,`. We get `ik = ij, jk`
4. The output is on the RHS and is separated with `->` sign. We get `ij, jk -> ik`

Here are some more examples from the docs

```
# Matrix multiplication
einsum('ij,jk->ik', m0, m1)  # output[i,k] = sum_j m0[i,j] * m1[j, k]

# Dot product
einsum('i,i->', u, v)  # output = sum_i u[i]*v[i]

# Outer product
einsum('i,j->ij', u, v)  # output[i,j] = u[i]*v[j]

# Transpose
einsum('ij->ji', m)  # output[j,i] = m[i,j]

# Trace
einsum('ii', m)  # output[j,i] = trace(m) = sum_i m[i, i]

# Batch matrix multiplication
einsum('aij,ajk->aik', s, t)  # out[a,i,k] = sum_j s[a,i,j] * t[a, j, k]
```

Share Follow

edited Jul 8, 2020 at 5:03

answered Jul 8, 2020 at 4:50

Souradeep Nanda

3,076 2 35 44