Topics

1. Array Summation
2. Array Multiplication
3. Array Broadcasting
4. Numpy’s Einsum Function (Einstein Summation)*

* Main focus
Summing of Multidimensional Arrays

```python
>>> import numpy as np
>>> a2 = np.array(range(9)).reshape(3,3)
>>> a3 = np.array(range(8)).reshape(2,2,2)
>>> a2
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> a3
array([[[0, 1],
         [2, 3]],
       [[[4, 5],
          [6, 7]]])

>>> a2.sum()  # Sum entire array
36
>>> a3.sum()  # Sum entire array
28
```

**Summing with axis = m**

=> Move along axis m and add up values, thus removing axis m.

```python
axis = 0
axis = 1
axis = 2
```

```python
>>> a2new0 = np.zeros(3)
>>> a2new0
array([0., 0., 0.])
>>> for j in range(3):
...     for i in range(3):
...         a2new0[j] += a2[i,j]
...
>>> a2new0
array([ 9., 12., 15.])
>>> a2.sum(axis=0)
array([ 9, 12, 15])
```

```python
>>> a3new0 = np.zeros((2,2))
>>> a3new0
array([[0., 0.],
        [0., 0.]])
>>> for j in range(2):
...     for k in range(2):
...         for i in range(2):
...             a3new0[j,k] += a3[i,j,k]
...
>>> a3new0
array([[ 4.,  6.],
        [ 8., 10.]])
>>> a3.sum(axis=0)
array([[ 4,  6],
        [ 8, 10]])
```

```python
>>> a2new1 = np.zeros(3)
>>> for i in range(3):
...     for j in range(3):
...         a2new1[i] += a2[i,j]
...
>>> a2new1
array([ 3., 12., 21.])
>>> a2.sum(axis=1)
array([ 3, 12, 21])
```

```python
>>> a3new1 = np.zeros((2,2))
>>> for i in range(2):
...     for k in range(2):
...         for j in range(2):
...             a3new1[i,k] += a3[i,j,k]
...
>>> a3new1
array([[ 2.,  4.],
        [10., 12.]])
>>> a3.sum(axis=1)
array([[ 2,  4],
        [10, 12]])
```
Summing with Einsum

```python
a = np.array(range(27)).reshape(3,3,3)
```

```python
>>> np.einsum('ijk->ij',a)
array([[ 3, 12, 21],
       [30, 39, 48],
       [57, 66, 75]])
```

```python
>>> np.einsum('ijk->ik',a)
array([[ 9, 12, 15],
       [36, 39, 42],
       [63, 66, 69]])
```

```python
>>> np.einsum('ijk->jk',a)
array([[27, 30, 33],
       [36, 39, 42],
       [45, 48, 51]])
```

Think of it as compressing along one or more dimensions

Think of it as **compressing** along one or more dimensions

Original array has 3 dimensions

Final array has 2 dimensions, with the dimension denoted by k summed over and removed.

Sum over **kth** dimension

A(i,j,0) + A(i,j,1) + ... + A(i,j,k_{max})

A(0,j,0) + A(0,j,1) + ... + A(0,j,k_{max}) +
A(1,j,0) + A(1,j,1) + ... + A(1,j,k_{max}) +
...
A(i_{max},j,0) + A(i_{max},j,1) + ... + A(i_{max},j,k_{max})
Many ways to multiply two matrices

```python
>>> x = np.array(range(4)).reshape(2,2)
>>> x
array([[0, 1],
       [2, 3]])
```

```python
>>> y = 0.1 * np.array(range(6)).reshape(2,3)
>>> y
array([[ 0.,  0.1,  0.2],
       [ 0.3,  0.4,  0.5]])
```

```python
>>> np.matmul(x, y)
array([[0.3, 0.4, 0.5],
       [0.9, 1.4, 1.9]])
```

```python
>>> np.dot(x, y)
array([[0.3, 0.4, 0.5],
       [0.9, 1.4, 1.9]])
```

```python
>>> np.einsum('ij,jk->ik', x, y)
array([[0.3, 0.4, 0.5],
       [0.9, 1.4, 1.9]])
```
Dissecting Einsum Multi-Array Operations

\[ \text{np.einsum}(\text{'ij,jk->ik'}, x, y) \]

**Dimensions of x**

**Dimensions of y**

**Dimensions of product**

Repetition of “j” => complete rows of \( x \) will undergo element-wise multiplication with complete columns of \( y \). Thus, the length of \( x \)'s axis=1 must equal the length of \( y \)'s axis=0.

Absence of “j” in the product array => the element-wise products will be summed (thus completing the dot product of \( x \)'s rows with \( y \)'s columns).

If these element-wise products are not to be summed, the einsum will add a third dimension to the result. Since it is also “j”, we know that it’s the same size as each of the original vectors (\( x \)'s rows and \( y \)'s columns).

\[ \text{np.einsum}(\text{'ij,jk->ijk'}, x, y) \]

\[ \text{np.einsum}(\text{'ij,jk->ikj'}, x, y) \]

or

\[ \text{np.einsum}(\text{'ij,jk->ikj'}, x, y) \]

>>> \( x \)
array([[0, 1],
       [2, 3]])

>>> \( y \)
array([[0., 0.1, 0.2],
       [0.3, 0.4, 0.5]])

\[ \text{np.einsum}(\text{'ij,jk->ijk'}, x, y) \]
array([[0., 0. , 0. ],
       [0.3, 0.4, 0.5],
       [0. , 0.2, 0.4],
       [0.9, 1.2, 1.5]])

\[ \text{np.einsum}(\text{'ij,jk->ikj'}, x, y) \]
array([[0. , 0.3],
       [0. , 0.4],
       [0. , 0.5],
       [0.9, 1.2, 1.5]])
The element-wise product vectors are \([0, 0.3], [0, 0.4], [0, 0.5] \ldots [0.4, 1.5]\)

Note how these vectors appear along axis = 1 here ...

\[
>>> \text{np.einsum('ij,jk->ijk',x,y)}
\]

array([[[0.0, 0.0, 0.0],
        [0.3, 0.4, 0.5]],
       [[0.0, 0.2, 0.4],
        [0.9, 1.2, 1.5]]])

\[
>>> \text{np.einsum('ij,jk->ikj',x,y)}
\]

array([[[0.0, 0.3],
        [0.0, 0.4],
        [0.0, 0.5]]],
       [[0.0, 0.9],
        [0.2, 1.2],
        [0.4, 1.5]]))

... along axis = 2 here ...

\[
>>> \text{np.einsum('ij,jk->jik',x,y)}
\]

array([[[0.0, 0.0, 0.0],
        [0.3, 0.4, 0.5]],
       [[0.0, 0.2, 0.4],
        [0.9, 1.2, 1.5]]])

... and along axis = 0 here ...

\[
>>> \text{np.einsum('ij,jk->ijk',x,y)}
\]

array([[[0.0, 0.0, 0.0],
        [0.3, 0.4, 0.5]],
       [[0.0, 0.9],
        [0.2, 1.2],
        [0.4, 1.5]]])

... along axis = 2 here ...

\[
>>> \text{np.einsum('ij,jk->ikj',x,y)}
\]

array([[[0.0, 0.3],
        [0.0, 0.4],
        [0.0, 0.5]]],
       [[0.0, 0.9],
        [0.2, 1.2],
        [0.4, 1.5]]))

... and along axis = 0 here ...

\[
>>> \text{np.einsum('ij,jk->jik',x,y)}
\]

array([[[0.0, 0.0, 0.0],
        [0.3, 0.4, 0.5]],
       [[0.0, 0.9],
        [0.2, 1.2],
        [0.4, 1.5]]])

... along axis = 2 here ...

\[
>>> \text{np.einsum('ij,jk->ijk',x,y)}
\]

array([[[0.0, 0.3],
        [0.0, 0.4],
        [0.0, 0.5]]],
       [[0.0, 0.9],
        [0.2, 1.2],
        [0.4, 1.5]]))
Handling Many Dimensions

>>> a = np.array(range(32)).reshape(2,2,2,2,2)
>>> a
array([[[[ 0,  1],
         [ 2,  3]],
        [[ 4,  5],
         [ 6,  7]]],
       [[[ 8,  9],
         [10, 11]],
        [[12, 13],
         [14, 15]]],
       [[[16, 17],
         [18, 19]],
        [[20, 21],
         [22, 23]]],
       [[[24, 25],
         [26, 27]],
        [[28, 29],
         [30, 31]]])

>>> b = np.array([0,0,1,1]).reshape(2,2)
>>> b
array([[0, 0],
       [1, 1]])

>>> c = np.array(range(4)).reshape(2,2)
>>> c
array([[0, 1],
       [2, 3]])

>>> np.dot(b,c)
array([[0, 0],
       [2, 4]])

>>> np.dot(c,b)
array([[1, 1],
       [3, 3]])

Pre-multiply by b
This broadcasts array b (due to ellipsis: ‘…’) to make it compatible with array a before doing the inner matrix multiplications.

>>> np.einsum('...ef,abcfk->abcek',b,a)
array([[[[ 0,  0],
         [ 2,  4]],
        [[ 0,  0],
         [10, 12]]],
       [[[ 0,  0],
         [18, 20]],
        [[ 0,  0],
         [26, 28]]],
       [[[ 0,  0],
         [34, 36]],
        [[ 0,  0],
         [42, 44]]],
       [[[ 0,  0],
         [50, 52]],
        [[ 0,  0],
         [58, 60]]])

Post-multiply by b
Ellipses lets us skip writing all of a's dimensions.

>>> np.einsum('...ef,fk->...ek',a,b)
array([[[ 1,  1],
         [ 3,  3]],
       [[[ 9,  9],
         [11, 11]],
       [[[13, 13],
         [15, 15]]],
       [[[17, 17],
         [19, 19]],
        [[21, 21],
         [23, 23]]],
       [[[25, 25],
         [27, 27]],
        [[29, 29],
         [31, 31]]])

Simple examples of pre- and post-multiplying by b

Create arrays a, b and c

Read about broadcasting in the numpy manual:
numpy.org/doc/stable/user/basics.broadcasting.html

All these give same result:
np.einsum('...ef,...fk->...ek',b,a)
gives the same result.

Implicit mode
Copy each element of $A_2$ $k_{\text{max}}$ times to fill the new dimension (k).

$k_{\text{max}} = 3$

Appending an extra index.

After choosing values for i and j, the value for k will not matter => copying was done correctly.
Inserting an index in the middle.

* Insertions of lower-order dimensions (higher in index tree) → repetition of longer sequences from the original array.
import numpy as np
import functools as FTOOL

def array_broadcast(a, k, ksize):
    dims = a.shape; newdims = list(dims)
    newdims.insert(k, ksize)
    aflat = list(a.flatten())  # pick sequences from this.
    a2flat = []  # Append subgroups to this and then reshape it in the end.
    num_buds = FTOOL.reduce((lambda a, b: a*b), [1] + newdims[0:k])
    subsize = int(len(aflat) / num_buds)  # size of a repeatable subtree
    for i in range(num_buds):
        flat_subtree = aflat[i * subsize:(i + 1) * subsize]
        a2flat.append(flat_subtree * ksize)  # Make ksize copies of the flat subtree
    return np.array(a2flat).reshape(newdims)

# Examples

>>> a
array([[ 0,  1,  2],
        [ 3,  4,  5]],
       [[ 6,  7,  8],
        [ 9, 10, 11]]])

>>> array_broadcast(a, 0, 2)
array([[ 0,  0,  0],
        [ 1,  1,  1],
        [ 2,  2,  2]],
       [[ 3,  3,  3],
        [ 4,  4,  4],
        [ 5,  5,  5]]])

>>> array_broadcast(a, 1, 3)
array([[ 0,  1,  2],
        [ 0,  1,  2],
        [ 0,  1,  2]],
       [[ 3,  4,  5],
        [ 3,  4,  5],
        [ 3,  4,  5]],
       [[ 6,  7,  8],
        [ 6,  7,  8],
        [ 6,  7,  8]],
       [[ 9, 10, 11],
        [ 9, 10, 11],
        [ 9, 10, 11]]])

>>> array_broadcast(a, 2, 4)
array([[ 0,  0,  0],
        [ 0,  0,  0],
        [ 0,  0,  0]],
       [[ 1,  1,  1],
        [ 1,  1,  1],
        [ 1,  1,  1]],
       [[ 2,  2,  2],
        [ 2,  2,  2],
        [ 2,  2,  2]],
       [[ 3,  3,  3],
        [ 3,  3,  3],
        [ 3,  3,  3]],
       [[ 4,  4,  4],
        [ 4,  4,  4],
        [ 4,  4,  4]],
       [[ 5,  5,  5],
        [ 5,  5,  5],
        [ 5,  5,  5]],
       [[ 6,  6,  6],
        [ 6,  6,  6],
        [ 6,  6,  6]],
       [[ 7,  7,  7],
        [ 7,  7,  7],
        [ 7,  7,  7]],
       [[ 8,  8,  8],
        [ 8,  8,  8],
        [ 8,  8,  8]],
       [[ 9,  9,  9],
        [ 9,  9,  9],
        [ 9,  9,  9]],
       [[10, 10, 10],
        [10, 10, 10],
        [10, 10, 10]],
       [[11, 11, 11],
        [11, 11, 11],
        [11, 11, 11]]])

>>> array_broadcast(a, 3, 3)
array([[ 0,  0,  0],
        [ 0,  0,  0],
        [ 0,  0,  0]],
       [[ 1,  1,  1],
        [ 1,  1,  1],
        [ 1,  1,  1]],
       [[ 2,  2,  2],
        [ 2,  2,  2],
        [ 2,  2,  2]],
       [[ 3,  3,  3],
        [ 3,  3,  3],
        [ 3,  3,  3]],
       [[ 4,  4,  4],
        [ 4,  4,  4],
        [ 4,  4,  4]],
       [[ 5,  5,  5],
        [ 5,  5,  5],
        [ 5,  5,  5]],
       [[ 6,  6,  6],
        [ 6,  6,  6],
        [ 6,  6,  6]],
       [[ 7,  7,  7],
        [ 7,  7,  7],
        [ 7,  7,  7]],
       [[ 8,  8,  8],
        [ 8,  8,  8],
        [ 8,  8,  8]],
       [[ 9,  9,  9],
        [ 9,  9,  9],
        [ 9,  9,  9]],
       [[10, 10, 10],
        [10, 10, 10],
        [10, 10, 10]],
       [[11, 11, 11],
        [11, 11, 11],
        [11, 11, 11]]])
Einsum’s Versatility

Matrix multiplication can be broken down into its two main steps via two einsums.

```python
>>> z = np.einsum('ij,jk->jik',x,y)  # Step 1
array([[[0. , 0. , 0. ],
        [0. , 0.2, 0.4]],
        [[0.3, 0.4, 0.5],
        [0.9, 1.2, 1.5]]])
>>> np.einsum('jik->ik',z)  # Step 2
array([[0.3, 0.4, 0.5],
        [0.9, 1.4, 1.9]])
>>> np.einsum('abc->bc',z)  # Any letters work
array([[0.3, 0.4, 0.5],
        [0.9, 1.4, 1.9]])
```

**Letter restrictions**
1) Repeating letters => respective axes must have same length.
2) Relative alphabetic order affects some defaults.

```python
>>> np.einsum('kj,ji->ik',x,y)  # Implicit mode: no arrow
array([[0.3, 0.9],
        [0.4, 1.4],
        [0.5, 1.9]])
>>> np.einsum('kj,ji->ki',x,y)  # Explicit mode: uses arrow
array([[0.3, 0.9],
        [0.4, 1.4],
        [0.5, 1.9]])
```

With no arrow, einsum assumes:
- a) the duplicate j's entail full dot products, and
- b) the order of the remaining axes is alphabetic: (i,k), not (k,i)

With arrow, but nothing on its right, einsum assumes:
- a) dot products of duplicate j's
- b) sum over entire product array

```python
>>> np.einsum('kj,ji->ik',x,y)  # Implicit mode: no arrow
array([[0.3, 0.9],
        [0.4, 1.4],
        [0.5, 1.9]])
>>> np.einsum('kj,ji->ki',x,y)  # Explicit mode: uses arrow
array([[0.3, 0.9],
        [0.4, 1.4],
        [0.5, 1.9]])
```

Compute the inner product, pairing rows of x to rows of w.

```python
>>> w = y.transpose()  # Step 1
>>> w
array([[0. , 0.3],
        [0.1, 0.4],
        [0.2, 0.5]])
>>> x
array([[0, 1],
        [2, 3]])
>>> np.einsum('ij,kj->ikj',x,w)
array([[[0. , 0.3],
        [0. , 0.4],
        [0. , 0.5]],
        [[0. , 0.9],
        [0.2, 1.2],
        [0.4, 1.5]])
>>> np.einsum('ij,kj->ik',x,w)
array([[0.3, 0.4, 0.5],
        [0.9, 1.4, 1.9]])
```

With arrow, but nothing on its right, einsum assumes:
- a) dot products of duplicate j's
- b) sum over entire product array

```python
>>> np.einsum('kj,ji->',x,y)  # Implicit mode: no arrow
5.4
>>> np.einsum('ab-',x)  # Implicit mode: no arrow
array([[0, 1],
        [2, 3]])
>>> np.einsum('ab->',x)
6
```

```
>>> np.einsum('kj,ji->',x,y)  # Implicit mode: no arrow
5.4
>>> np.einsum('ab-',x)  # Implicit mode: no arrow
array([[0, 1],
        [2, 3]])
>>> np.einsum('ab->',x)
6
```

Explicit mode: uses arrow
Implicit mode: no arrow

Keith L. Downing  |  Numpy Array Operations: Supplementary Material
### Deep Array Modifications via Einsum

#### Normalizing Columns

```python
>>> x = np.array(range(9)).reshape(3,3)
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> colsums = np.einsum('ij->j',x)
>>> colsums
array([ 9, 12, 15])
>>> q = 1 / colsums
>>> q
array([0.11111111, 0.08333333, 0.06666667])
>>> norm_cols = np.einsum('ij,j->ij',x,q)
>>> norm_cols
array([[0.        , 0.08333333, 0.13333333],
        [0.33333333, 0.33333333, 0.33333333],
        [0.66666667, 0.58333333, 0.53333333]])
```

#### Normalizing Rows (without intermediate variables)

```python
>>> einsum('ij,i->ij',x,1/np.einsum('ij->i',x))
array([[0.        , 0.33333333, 0.66666667],
        [0.25      , 0.33333333, 0.41666667],
        [0.28571429, 0.33333333, 0.38095238]])
```

* Note: These do not account for divide-by-zero possibilities. But this does...

```python
>>> rsums = np.einsum('ij->i',x)
>>> np.einsum('ij,i->ij',x,np.where(rsums != 0, 1/rsums, 1))
```

#### Normalizing vectors along axis = 2 in a 4-d array

```python
>>> z = np.array(range(81)).reshape(3,3,3,3)
>>> z
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],
        [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],
        [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
>>> q = np.einsum('ijkl->ijl',z)
>>> q
array([[[ 9, 12, 15],
        [ 36,  39, 42],
        [ 63,  66,  69]],
        [[ 90,  93,  96],
        [117, 120, 123],
        [144, 147, 150]],
        [[171, 174, 177],
        [198, 201, 204],
        [225, 228, 231]])
```

(1) Create 4-d array

(2) Calculate vector sums along axis 2.

(3) Divide each element of an axis-2 vector by its vector's sum.

Ellipsis (...) tells einsum to broadcast the reciprocal vector sums across an additional axis of q, thus filling axis 2 with copies of those reciprocals.

#### Ellipsis (…) tells einsum to broadcast the reciprocal vector sums across an additional axis of q, thus filling axis 2 with copies of those reciprocals.

Note that q has only 3 dimensions

```python
>>> np.einsum('abcd,ab...d->abcd',z,1/q)
array([[[0.        , 0.08333333, 0.13333333],
        [0.33333333, 0.33333333, 0.33333333],
        [0.66666667, 0.58333333, 0.53333333]],
        [[0.25      , 0.25641026, 0.26190476],
        [0.33333333, 0.33333333, 0.33333333],
        [0.41666667, 0.41025641, 0.4047619 ]],
        [[0.32      , 0.32017544, 0.32034632],
        [0.33333333, 0.33333333, 0.33333333],
        [0.34666667, 0.34649123, 0.34632035]])
```
Feed-Forward Calculations for a Neural Network

```python
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

x = np.random.uniform(0,1,5)  # Random input case
w = np.random.uniform(-.1,.1,(5,8))  # the weights
y_bias = np.random.uniform(0,1,8) # biases for 2nd layer
y = sigmoid(np.einsum('a,ab->b',x,w) + y_bias)
```

Simple change to handle minibatches

```python
minbatch_size = 7
xnew = np.random.uniform(0,1,(minbatch_size,5))
ynew = sigmoid(np.einsum('ma,ab->mb',xnew,w) + y_bias)
```

The addition of the 7 x 8 result of einsum to the y_bias vector of size 8 forces numpy to broadcast the y_bias vector to fill an entire 7 x 8 array, where each row is a copy of the original vector. The sigmoid function is then applied individually to each element of the 7 x 8 array to produce the minibatch of outputs.
Einsum rules are normally easiest to understand at the high level using the basic interpretation rules:

- Letters that appear in more than one operand denote axes to be element-wise multiplied.
- Letters that appear on the left-hand-side but not on the right-hand-side denote summations of product vectors.

These should be viewed as **constraints**. For example, \( z = \text{np.einsum}(\text{'}ab,ab\rightarrow ab\text{',}x,y,) \) says that the columns of \( z \) are constrained to be the products of the columns of \( x \) and \( y \), and the rows of \( z \) are constrained to be the products of the rows of \( x \) and \( y \).

But it does NOT mean that einsum will take the products of \( x \)-\( y \) rows and the products of \( x \)-\( y \) columns and combine them in some way.

The computation that einsum (implicitly*) performs is essentially:

- Nested FOR loops iterating across all dimensions of all operands.
- Inside all of these loops: a single command that increments a cell in the output array by products from the input arrays.

The letters in the einsum rule are easily translated into the proper indices used in this single inner command !!

*Explicitly, einsum’s source code is not so simple, since it has to handle ANY rule.
np.einsum('ab, bc->ac', x, y) = standard matrix multiplication. It assumes x.shape[1] = y.shape[0].

Permuting the FOR loops does not change the result.

```
def mm(x,y):
    z = np.zeros((x.shape[0],y.shape[1]))
    for a in range(x.shape[0]):
        for c in range(y.shape[1]):
            for b in range(x.shape[1]):
                z[a,c] += x[a,b] * y[b,c]
    return z
```

```
def mm2(x,y):
    z = np.zeros((x.shape[0],y.shape[1]))
    for b in range(x.shape[1]):
        for a in range(x.shape[0]):
            for c in range(y.shape[1]):
                z[a,c] += x[a,b] * y[b,c]
    return z
```

np.einsum('ab, ab->ab', x, y) = Hammard product. It assumes x.shape = y.shape.

```
def hammard2(x,y):
    z = np.zeros(x.shape)
    for a in range(x.shape[0]):
        for b in range(x.shape[1]):
            z[a,b] += x[a,b] * y[a,b]
    return z
```

```
>>> x
array([[0, 1],
       [2, 3]])

>>> y
array([[0. , 0.1],
       [0.2, 0.3]])

>>> hammard2(x,y)
array([[ 0.2,  0.3],
       [ 0.6,  1.1]])
```

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

>>> mm(x,w)
array([[ 3.,  4.,  5.],
       [ 9., 14., 19.]])
```

```
>>> mm2(x,w)
array([[ 3.,  4.,  5.],
       [ 9., 14., 19.]])
```

def deepmm(x,y):
    shx = x.shape; shy = y.shape
    z = np.zeros(shx[0:4]+[shy[1]])
    for a in range(shx[0]):
        for b in range(shx[1]):
            for c in range(shx[2]):
                for d in range(shx[3]):
                    for f in range(shy[1]):
                        for e in range(shx[4]):
                            z[a,b,c,d,f] += x[a,b,c,d,e] * y[e,f]
    return z

Although it does not affect the result, this code is more easily understood if:

a) All of variables that appear on both the left and right sides of the rule are iterated in the outer FOR loops.

b) All variables that are being summed over (and thus do not appear on the right side) are iterated over in the inner FOR loops.

np.einsum('abcde,ef->abcdf', x, y) = Deep matrix multiplication.
It assumes x.shape[4] = y.shape[0].

def deepmm(x,y):
    shx = x.shape; shy = y.shape
    z = np.zeros(shx[0:4]+[shy[1]])
    for a in range(shx[0]):
        for b in range(shx[1]):
            for c in range(shx[2]):
                for d in range(shx[3]):
                    for f in range(shy[1]):
                        for e in range(shx[4]):
                            z[a,b,c,d,f] += x[a,b,c,d,e] * y[e,f]
    return z

Although it does not affect the result, this code is more easily understood if:

a) All of variables that appear on both the left and right sides of the rule are iterated in the outer FOR loops.

b) All variables that are being summed over (and thus do not appear on the right side) are iterated over in the inner FOR loops.

Gives same result, except all real values are integers

>>> x
array([[[[ 0,  1],
    [ 2,  3]],
   [[ 4,  5],
    [ 6,  7]]],
  [[ 8,  9],
   [10, 11]],
  [[12, 13],
   [14, 15]]])

>>> y
array([[0, 1],
   [2, 3]])

Gives same result, except all real values are integers

>>> np.einsum('abcde,ef->abcdf', x, y)