idss_selfstudy_numerical_i_ch_4

January 14, 2021

---

1 Introduction to Data Science and Systems

1.1 Self-study: Arrays, numpy and vectorisaiton

1.1.1 Chapter 4: Arithmetic, broadcasting and aggregationű

University of Glasgow - material prepared by John H. Williamson* (adapted to IDSS by BSJ). $$

p>SyntaxError: unexpected character after line continuation character (, line 1)

*arg min

%< %R $$

---

[1]:
```
import IPython.display
IPython.display.HTML("""
<script>
  function code_toggle() {
    if (code_shown){
      $('div.input').hide('500');
      $('#toggleButton').val('Show Code')
    } else {
      $('div.input').show('500');
      $('#toggleButton').val('Hide Code')
    }
    code_shown = !code_shown
  }

  $( document ).ready(function(){
    code_shown=false;
    $('div.input').hide()
  });
</script>
<form action="javascript:code_toggle()"><input type="submit" id="toggleButton"␣
 ↪value="Show Code"></form>""")
```

[1]: <IPython.core.display.HTML object>

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from jhwutils.image_audio import play_sound, show_image, load_sound,␣
 ↪show_image_mpl, load_image_gray
from jhwutils.matrices import show_boxed_tensor_latex, print_matrix
%matplotlib inline
```

```
c:\python\Anaconda3_37_201907\lib\site-packages\IPython\kernel\__init__.py:13:
ShimWarning: The `IPython.kernel` package has been deprecated since IPython
4.0.You should import from ipykernel or jupyter_client instead.
  "You should import from ipykernel or jupyter_client instead.", ShimWarning)
```

---

2 Map: arithmetic on arrays

The major advantage of array representations is to be able to do arithmetic on arrays directly.

Basic arithmetic is computed **elementwise**. This means that a function is applied to each element of an array. There are a few different kind of element wise operations:

- single argument, like `np.tan()` or unary negative `(-x)`
- two argument, like `x+y` or `x-1` or `np.maximum(x,y)`
- and various other cases, like np.where(condition, true_values, false_values)‘

All of these work on arrays without any special syntax. We can simply write expressions using array variables.

```
x + y + 2  # if x and y are arrays, this just works
```

[3]:
```
x = np.array([1,2,3,4])
y = np.array([0,1,2,3])
print_matrix("x", x)
print_matrix("y", y)
```

$$x = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$$
$$y = \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}$$

[4]:
```
print_matrix("x+y", x+y)
print_matrix("x-y", x-y)
print_matrix("x*y", x*y)
print_matrix("x/y", x/y)
print_matrix("x^y", x**y)
```

$$x + y = \begin{bmatrix} 1 & 3 & 5 & 7 \end{bmatrix}$$
$$x - y = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}$$
$$x * y = \begin{bmatrix} 0 & 2 & 6 & 12 \end{bmatrix}$$

```
C:\python\Anaconda3_37_201812\lib\site-packages\ipykernel_launcher.py:4:
RuntimeWarning: divide by zero encountered in true_divide
  after removing the cwd from sys.path.
```

$x/y = \begin{bmatrix}\infty & 2.0 & 1.5 & 1.33333333333333\end{bmatrix}$
$x^y = \begin{bmatrix}1 & 2 & 9 & 64\end{bmatrix}$

[5]:
```
# examples with scalars and arrays
print_matrix("x+1", x+1)
print_matrix("2x", x*2)
print_matrix("1/x", 1/x)
print_matrix("x^2", x**2)
```

$x+1 = \begin{bmatrix}2 & 3 & 4 & 5\end{bmatrix}$
$2x = \begin{bmatrix}2 & 4 & 6 & 8\end{bmatrix}$
$1/x = \begin{bmatrix}1.0 & 0.5 & 0.333333333333333 & 0.25\end{bmatrix}$
$x^2 = \begin{bmatrix}1 & 4 & 9 & 16\end{bmatrix}$

3 An example: changing volume

If a sound is just an array of values, then we can use array operations to apply changes to a whole sound at once.

For example, scaling (multiplying the values) will change the volume.

[6]:
```
guitar = load_sound("sounds/guitar.wav")
play_sound(guitar)
```

```
C:\python\Anaconda3_37_201812\lib\site-packages\scipy\io\wavfile.py:273:
WavFileWarning: Chunk (non-data) not understood, skipping it.
  WavFileWarning)
```

```
<IPython.lib.display.Audio object>
```

[7]:
```
play_sound(guitar*0.2)
```

```
<IPython.lib.display.Audio object>
```

[8]:
```
play_sound(guitar*3)
```

```
<IPython.lib.display.Audio object>
```

[9]:
```
play_sound(np.tanh(guitar*300)*0.15) # distortion pedal
```

```
<IPython.lib.display.Audio object>
```

If we wanted to fade out the sound, we'd need to multiply *each element* by a different value (e.g. fading from 0.0 to 1.0).

[10]:
```
fade = np.linspace(1,0, len(guitar))
play_sound(guitar*fade)
```

```
<IPython.lib.display.Audio object>
```

3.1 Mixing sounds

Mixing sounds simply involves adding them (and possibly reducing the gain)

[11]:
```
sax = load_sound("sounds/sax.wav")
play_sound(sax)
```

```
<IPython.lib.display.Audio object>
```

[12]:
```
max_len = min(len(guitar), len(sax))
# nb: here I slice so that both are the length of the shortest sound
play_sound(sax[:max_len]+guitar[:max_len])
```

```
<IPython.lib.display.Audio object>
```

[13]:
```
play_sound(sax[:max_len]*fade[:max_len]+guitar[:max_len]*(1-fade[:max_len]))
```

```
<IPython.lib.display.Audio object>
```

3.2 Map

This is a special case of a **map**: the application of a function to each element of a sequence.

There are certain rules which dictate what operations can be applied together.

- For single argument operations, there is no problem; the operation is applied to each element of the array
- If there are more than two arguments, like in `x + y`, then `x` and `y` must have **compatible shapes**. This means it must be possible to pair each element of `x` with a corresponding element of `y`

3.2.1 Same shape

In the simplest case, `x` and `y` have the same shape; then the operation is applied to each pair of elements from `x` and `y` in sequence.

3.2.2 Not the same shape

If `x` and `y` aren't the same shape, it might seem like they cannot be added (or divided, or "maximumed"). However, NumPy provides **broadcasting rules** to allow arrays to be automatically expanded to allow operations between certain shapes of arrays.

3.2.3 Repeat until they match

The rule is simple; if the arrays don't match in size, but one array can be *tiled* to be the same size as the other, this tiling is done implicitly as the operation occurs. For example, adding a scalar to an array implicitly *tiles* the scalar to the size of the array, then adds the two arrays together (this is done much more efficiently internally than explicitly generating the array).

The easiest broadcasting rule is scalar arithmetic: `x+1` is valid for any array `x`, because NumPy **broadcasts** the 1 to make it the same shape as `x` and then adds them together, so that every element of `x` is paired with a 1.

Broadcasting always works for any scalar and any array, because a scalar can be repeated however many times necessary to make the operation work.

You can imagine that `x+1` is really `x + np.tile(1, x.shape)` which works the same, but is much less efficient:

[14]:
```
x = np.zeros((5,5))
x + 1
```

[14]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

[15]:
```
# same, but creates large temporary array
x + np.tile(1, x.shape)
```

[15]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

3.3 Broadcasting

So far we have seen: * **elementwise array arithmetic** (both sides of an operator have exactly the same shape) and * **scalar arithmetic** (one side of the operator is a scalar, and the other is an array).

This is part of a general pattern, which lets us very compactly write operations between arrays of different sizes, under some specific restrictions.

Broadcasting is the way in which arithmetic operations are done on arrays when the operands are of different shapes.

1. If the operands have the same number of dimensions, then they **must** have the same shape; operations are done elementwise. `y = x + x`
2. If one operand is an array with fewer dimensions than the other, then if the *last dimensions* of the first array match the shape as the second array, operations are well-defined. If we have a LHS of size (. . . ,j,k,l) and a RHS of (l) or (k,l) or (j,k,l) etc., then everything is OK.

This says for example that:

```
shape (2,2) * shape(2,) -> valid
shape (2,3,4) * shape(3,4) -> valid
```

```
shape (2,3,4) * shape(4,) -> valid

shape (2,3,4) * shape (2,4) -> invalid
shape (2,3,4) * shape(2) --> invalid
shape (2,3,4) * shape(8) --> invalid
```

Broadcasting is just automatic tiling When broadcasting, the array is *repeated* or tiling as needed to expand to the correct size, then the operation is applied. So adding a (2,3) array and a (3,) array means repeating the (3,) array into 2 identical rows, then adding to the (2,3) array.

---

[16]:
```
vec4 = np.array([1,2,3,4])
mat4 = np.zeros((4,4)) # 4x4 zeros
```

[17]:
```
mat3x4 = np.full((3,4), 8.0) # 3x4 filled with 8
vec3 = np.array([1,2,3])
vec4x = np.array([1,1,1,1])
```

[18]:
```
print((vec4+1))         # scalar (Rule 2)
print((vec4 + vec4x))   # elementwise (Rule 1)
```

```
[2 3 4 5]
[2 3 4 5]
```

[19]:
```
print((mat4 + mat4))    # elementwise (Rule 1)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

[20]:
```
# note that vec4 is repeated over the rows to make it the same size
print((mat4 + vec4))    # broadcasting: valid because RHS (vec4) has dimension␣
 ↪matching the last dimension of matt4
```

```
[[1. 2. 3. 4.]
 [1. 2. 3. 4.]
 [1. 2. 3. 4.]
 [1. 2. 3. 4.]]
```

[21]:
```
# note that the operation operates across *columns*, i.e the last dimension of␣
 ↪the array
print((mat3x4 + vec4)) # broadcasting
```

```
[[ 9. 10. 11. 12.]
 [ 9. 10. 11. 12.]
 [ 9. 10. 11. 12.]]
```

[22]:
```
# broadcasting also works on comparisons
mat4x = np.array([[1,2,3,4],
                  [4,5,6,7],
                  [8,9,10,11],
                  [12,13,14,15]])
print((mat4x>vec4))
# note this has compared [1,2,3,4] to each row of mat4x
```

```
[[False False False False]
 [ True  True  True  True]
 [ True  True  True  True]
 [ True  True  True  True]]
```

3.3.1 Invalid broadcasting examples

[23]:
```
print((mat3x4 + vec3)) # invalid: last dimensions don't match!
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-23-1b68fd0fbbe9> in <module>
----> 1 print((mat3x4 + vec3)) # invalid: last dimensions don't match!


ValueError: operands could not be broadcast together with shapes (3,4) (3,)
```

[24]:
```
print((mat4+vec3)) # invalid: last dimensions don't match
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-24-ce5edd0958c6> in <module>
----> 1 print((mat4+vec3)) # invalid: last dimensions don't match


ValueError: operands could not be broadcast together with shapes (4,4) (3,)
```

[25]:
```
print((mat4+mat3x4)) # invalid: arrays have same rank but different shape
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-25-964961da9c7b> in <module>
----> 1 print((mat4+mat3x4)) # invalid: arrays have same rank but different␣
↪shape


ValueError: operands could not be broadcast together with shapes (4,4)␣
↪(3,4)
```

3.4 Transposing in broadcasts

Transpose solves one of the problems you might have seen with broadcasting. Imagine we want to add a vector to every row of a matrix. This is easy:

[26]:
```
x = np.zeros((4,3)) # 4 rows, 3 columns
y = np.array([1,1,9]) # 3 element vector, applies to each row
print_matrix("x",x )
print_matrix("y",y)
# this will repeat the 3 element vector into 4 rows, then add
print_matrix("x+y",x+y)
```

$$x = \begin{bmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}$$

$$y = \begin{bmatrix} 1 & 1 & 9 \end{bmatrix}$$

$$x + y = \begin{bmatrix} 1.0 & 1.0 & 9.0 \\ 1.0 & 1.0 & 9.0 \\ 1.0 & 1.0 & 9.0 \\ 1.0 & 1.0 & 9.0 \end{bmatrix}$$

But how would we add a vector to each *column*? This would need a 4 element vector, and this cannot be added directly as it violates the broadcasting rules

[27]:
```
z = np.array([1,2,3,4])
print_matrix("x+z", x+z) # this can't work; a 4x3 and a 4 don't have matching␣
 ↪last dimensions
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-27-35d1b012d421> in <module>
  1 z = np.array([1,2,3,4])
```

```
----> 2 print_matrix("x+z", x+z) # this can't work; a 4x3 and a 4 don't have␣
↪matching last dimensions


ValueError: operands could not be broadcast together with shapes (4,3) (4,)
```

But the **transpose** of x is a 3x4 matrix, to which z can be added. The result is transposed, so we transpose it back:

[28]:
```
print_matrix("(x^T+z^T)^T", (x.T + z).T)
```

$$(x^T + z^T)^T = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 2.0 & 2.0 & 2.0 \\ 3.0 & 3.0 & 3.0 \\ 4.0 & 4.0 & 4.0 \end{bmatrix}$$

3.4.1 Tiling as broadcasting

Note again that `np.tile` repeats arrays explicitly, but we can get the same effect by broadcasting instead:

[29]:
```
# use tiling to repeat an array
x = np.array([1.0,2.0,3.0,4.0])
print_matrix("x", np.tile(x,(3,1)))
```

$$x = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \\ 1.0 & 2.0 & 3.0 & 4.0 \\ 1.0 & 2.0 & 3.0 & 4.0 \end{bmatrix}$$

[30]:
```
# same effect, using broadcasting with a 3x1 array of zeros
print_matrix("x", x + np.zeros((3,1)))
```

$$x = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \\ 1.0 & 2.0 & 3.0 & 4.0 \\ 1.0 & 2.0 & 3.0 & 4.0 \end{bmatrix}$$

4 Reduction

Reduction is the process of applying an operator or function with two arguments repeatedly to some sequence.

For example, if we reduce [1,2,3,4] with +, the result is `1+2+3+4 = 10`. If we reduce `[1,2,3,4]` with `*`, the result is `1*2*3*4 = 24`.

Reduction: stick an operator in between elements

```
1 2 3 4
5 6 7 8
```

Reduce on columns with "+":

```
1 + 2 + 3 + 4  =  10
5 + 6 + 7 + 8  =  26
```

Reduce on rows with "+":

```
1 2 3 4
+ + + +
5 6 7 8

=
6 8 10 12
```

Reduce on rows then columns:

```
1 + 2 + 3 + 4
+   +   +   +
5 + 6 + 7 + 8

=
6 + 8 + 10 + 12  = 36
```

Many operations can be expressed as reductions. These are **aggregate** operations.

`np.any` and `np.all` test if an array of Boolean values is all True or not all False (i.e. if any element is True). These are one kind of **aggregate function** – a function that processes an array and returns a single value which "summarises" the array in some way.

- `np.any` is the reduction with logical OR
- `np.all` is the reduction with logical AND
- `np.min` is the reduction with min(a,b)
- `np.max` is the reduction with max(a,b)
- `np.sum` is the reduction with +
- `np.prod` is the reduction with *

[31]:
```
print("any", np.any([True, False, False])) # true = True or False or False
print("all", np.all([True, False, False])) # false = True and False and False
```

```
any True
all False
```

[32]:
```
x = np.array([1,2,3,4,5,6]) # 1 + 2 + 3 + 4 + 5 + 6
print(np.sum(x))
```

```
21
```

[33]:
```
print(np.prod(x)) # 1 * 2 * 3 * 4 * 5 * 6 = 6!
```

```
720
```

[34]:
```
print(np.max(x)) # max(max(max(max(max(1,2), 3), 4), 5), 6)
```

```
6
```

Some functions are built on top of reductions: * `np.mean` is the sum divided by the number of elements reduced * `np.std` computes the standard deviation using the mean, then some element-wise arithmetic

[35]:
```
print(np.mean(x))
print(np.sum(x) / len(x)) # equivalent
```

```
3.5
3.5
```

By default, aggregate functions operate over the whole array, regardless of how many dimensions it has. This means reducing over the last axis, then reducing over the second last axis, and so on, until a single scalar remains. For example, `np.max(x)`, if `x` is a 2D array, will compute the reduction across columns and get the max for each row, then reduce over rows to get the max over the whole array.

We can specify the specific axes to reduce on using the `axes=` argument to any function that reduces.

[36]:
```
x = np.array([[1,2,3], [4,5,6], [7,8,9]])
print_matrix("x", x)
print("max(x)=", np.max(x)) # reduce on all axes
print_matrix("max_{0}(x)", np.max(x, axis=0)) # reduce on rows
print_matrix("max_{1}(x)", np.max(x, axis=1)) # reduce on columns
print("max_{0,1}(x)=", np.max(x, axis=(0,1))) # same as all axes in this case
```

$$x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

```
max(x)= 9
```

$$max_0(x) = \begin{bmatrix} 7 & 8 & 9 \end{bmatrix}$$
$$max_1(x) = \begin{bmatrix} 3 & 6 & 9 \end{bmatrix}$$

```
max_{0,1}(x)= 9
```

[37]:
```
print_matrix("\\text{mean}_0(x)", np.mean(x, axis=0)) # mean on rows
print_matrix("\\text{mean}_1(x)", np.mean(x, axis=1)) # mean on columns
```

$$\text{mean}_0(x) = \begin{bmatrix} 4.0 & 5.0 & 6.0 \end{bmatrix}$$
$$\text{mean}_1(x) = \begin{bmatrix} 2.0 & 5.0 & 8.0 \end{bmatrix}$$

5 Accumulation

The sum of an array is a single scalar value. The **cumulative sum** or **running sum** of an array is an array of the same size, which stores the result of summing up every element until that point.

This is almost the same as reduction, but we keep intermediate values during the computation, instead of collapsing to just the final result. The general process is called **accumulation** and it can be used with different operators.

For example, the accumulation of `[1,2,3,4]` with `+` is `[1, 1+2, 1+2+3, 1+2+3+4] = [1,3,6,10]`.

- `np.cumsum` is the accumulation of `+`

- `np.cumprod` is the accumulation of `*`
- `np.diff` is the accumulation of `-` (but note that it has one less output than input)

Accumulations operate on a single axis at a time, and you should specify this if you are using them on an array with more than one dimension (otherwise you will get the accumulation of flattened array).

[38]:
```
print_matrix("x", x)
print_matrix("\\text{cumsum}_0(x)", np.cumsum(x, axis=0)) # sum across rows
print_matrix("\\text{cumprod}_1(x)", np.cumprod(x, axis=1)) # product across␣
↪columns
print_matrix("\\text{diff}_0(x)", np.diff(x, axis=0)) # difference across rows
print_matrix("\\text{diff}_1(x)", np.diff(x, axis=1)) # difference across columns
```

$$x = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

$$\text{cumsum}_0(x) = \begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 12 & 15 & 18 \end{bmatrix}$$

$$\text{cumprod}_1(x) = \begin{bmatrix} 1 & 2 & 6 \\ 4 & 20 & 120 \\ 7 & 56 & 504 \end{bmatrix}$$

$$\text{diff}_0(x) = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \end{bmatrix}$$

$$\text{diff}_1(x) = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

- `np.gradient` is like `np.diff` but uses central differences to get same length output, and it computes the gradient over *every* axis and returns them all in a list. It is a very useful function in image processing.

[39]:
```
print_matrix("\\nabla x_0", np.gradient(x)[0])
print_matrix("\\nabla x_1", np.gradient(x)[1])
```

$$\nabla x_0 = \begin{bmatrix} 3.0 & 3.0 & 3.0 \\ 3.0 & 3.0 & 3.0 \\ 3.0 & 3.0 & 3.0 \end{bmatrix}$$

$$\nabla x_1 = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 \end{bmatrix}$$

5.1 Finding

There are functions which find **indices** that satisfy criteria. For example, the largest value along some axis.

- `np.argmax()` finds the index of the largest element
- `np.argmin()` finds the index of the smallest element

- `np.argsort()` finds the indices that would sort the array back into order
- `np.nonzero()` finds indices that are non-zero (or True, for Boolean arrays)

Finding indices is of great importance, because it allows us to cross-reference across axes or arrays. For example, we can find the row where some value is maximised (most wheat production) and then find the attribute which corresponds to it (the year when most wheat was produced).

[40]:
```
x = np.array([5,9,0,13,-8,7,2,8,0,-8])
print_matrix("x", x)
# note: argmin/max will tie break on the first occurence
print("argmin(x)=", np.argmin(x))
print("argmax(x)=", np.argmax(x))
print("nonzero(x)=", np.nonzero(x))
## argmin is almost the same as this
## but this can return *multiple* minimums, instead of the first
print(np.nonzero(x==np.min(x)))
```

$$x = \begin{bmatrix} 5 & 9 & 0 & 13 & -8 & 7 & 2 & 8 & 0 & -8 \end{bmatrix}$$

```
argmin(x)= 4
argmax(x)= 3
nonzero(x)= (array([0, 1, 3, 4, 5, 6, 7, 9], dtype=int64),)
(array([4, 9], dtype=int64),)
```

[41]:
```
# get the indices that would put x into order
print(np.argsort(x))
```

```
[4 9 2 8 6 0 5 7 1 3]
```

[42]:
```
# hey presto! sorted!
print(x[np.argsort(x)])
```

```
[-8 -8  0  0  2  5  7  8  9 13]
```

5.2 Selection

- **Slicing** can chop out rectangular sections of an array, including with regular gaps. `x[2:5, :]` selects rows 3-6 of `x`. `x[:,:,::-1]` selects all of `x`, but with the last axis reversed (e.g RGB colours -> BGR colours).
- **Specific indexing** We can index anywhere we could specify a slice range with a list instead. This allows quick tricks to rearrange elements. `x[[0,2,1], :]` will return an array with the first, third and second rows of `x`, in that order. `x[np.argsort(x)]` will sort `x`, because `np.argsort` returns the indices that would sort `x`.
- **Boolean indexing** or **masking**. if we index an array with a Boolean array instead of a slice range, we will get all of the elements where that array was True. This is particularly useful in assignments. We can, for example, write `x[x>5] = 0` to set all value of `x` that are greater than 5 to 0.

[ ]: