NumPy Broadcasting & JAX

Lecture 06

Dr. Colin Rundel

Basic file IO

Reading and writing ndarrays

We will not spend much time on this as most data you will encounter is more likely to be a tabular format (e.g. data frame) and tools like Pandas are more appropriate.

For basic saving and loading of NumPy arrays there are the `save()` and `load()` functions which use a built in binary format.

```python
x = np.arange(1e5)
np.save("data/x.npy", x)
```

```python
new_x = np.load("data/x.npy")
np.all(x == new_x)
```

```
np.True_
```

Additional functions for saving (`savez()`, `savez_compressed()`, `savetxt()`) exist for saving multiple arrays or saving a text representation of an array.

Reading delimited data

While not particularly recommended, if you need to read delimited (csv, tsv, etc.) data into a NumPy array you can use `genfromtxt()`,

```python
with open("data/mtcars.csv") as file:
    mtcars = np.genfromtxt(file, delimiter=",", skip_header=True)

mtcars
```

```
array([[  6.   , 160.   , 110.   ,   3.9  ,   2.62 ,
         16.46 ,   0.   ,   1.   ,   4.   ,   4.   ],
       [  6.   , 160.   , 110.   ,   3.9  ,   2.875,
         17.02 ,   0.   ,   1.   ,   4.   ,   4.   ],
       [  4.   , 108.   ,  93.   ,   3.85 ,   2.32 ,
         18.61 ,   1.   ,   1.   ,   4.   ,   1.   ],
       [  6.   , 258.   , 110.   ,   3.08 ,   3.215,
         19.44 ,   1.   ,   0.   ,   3.   ,   1.   ],
       [  8.   , 360.   , 175.   ,   3.15 ,   3.44 ,
         17.02 ,   0.   ,   0.   ,   3.   ,   2.   ],
       [  6.   , 225.   , 105.   ,   2.76 ,   3.46 ,
         20.22 ,   1.   ,   0.   ,   3.   ,   1.   ],
       [  8.   , 360.   , 245.   ,   3.21 ,   3.57 ,
         15.84 ,   0.   ,   0.   ,   3.   ,   4.   ],
       [  4.   , 146.7  ,  62.   ,   3.69 ,   3.19 ,
```

Broadcasting

Broadcasting

This is an approach for deciding how to generalize operations between arrays with differing shapes.

```python
x = np.array([1, 2, 3])
```

```python
x * 2
```

```
array([2, 4, 6])
```

```python
x * np.array([2,2,2])
```

```
array([2, 4, 6])
```

```python
x * np.array([2])
```

```
array([2, 4, 6])
```

Efficiency

Using broadcasts can be more efficient as it does not copy the broadcast data,

```python
x = np.arange(1e5)
y = np.array([2]).repeat(1e5)
```

```python
%timeit x * 2
```

```
13.1 µs ± 297 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```python
%timeit x * np.array([2])
```

```
18.3 µs ± 96.4 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```python
%timeit x * y
```

```
61.4 µs ± 465 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```python
%timeit x * np.array([2]).repeat(1e5)
```

```
99.2 µs ± 1.12 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Rules for Broadcasting

> When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are compatible when
>
> 1. they are equal, or
> 2. one of them is 1
>
> If these conditions are not met, a `ValueError: operands could not be broadcast together` exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the size that is not 1 along each axis of the inputs.

Example

Why does the code on the left work but not the code on the right?

```python
x = np.arange(12).reshape((4,3)); x
```

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

```python
x + np.array([1,2,3])
```

```
array([[ 1,  3,  5],
       [ 4,  6,  8],
       [ 7,  9, 11],
       [10, 12, 14]])
```

```python
x = np.arange(12).reshape((3,4)); x
```

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

```python
x + np.array([1,2,3])
```

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

```
x    (2d array): 4 x 3
y    (1d array):     3
----------------------
x+y  (2d array): 4 x 3
```

```
x    (2d array): 3 x 4
y    (1d array):     3
----------------------
x+y  (2d array): Error
```

A fix

```python
x = np.arange(12).reshape((3,4)); x
```

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

```python
x + np.array([1,2,3]).reshape(3,1)
```

```
array([[ 1,  2,  3,  4],
       [ 6,  7,  8,  9],
       [11, 12, 13, 14]])
```

```
x    (2d array): 3 x 4
y    (2d array): 3 x 1
----------------------
x+y  (2d array): 3 x 4
```

Examples (2)

```python
x = np.arange(12).reshape((4,3))
y = 1
x+y
```

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
x    (2d array): 4 x 3
y    (1d array):     1
----------------------
x+y  (2d array): 4 x 3
```

```python
x = np.arange(12).reshape((4,3))
y = np.array([1,2,3])
x+y
```

```
array([[ 1,  3,  5],
       [ 4,  6,  8],
       [ 7,  9, 11],
       [10, 12, 14]])
```

```
x    (2d array): 4 x 3
y    (1d array):     3
----------------------
x+y  (2d array): 4 x 3
```

Examples (3)

```python
x = np.array([0,10,20,30]).reshape((4,1))
y = np.array([1,2,3])
```

```python
x
```

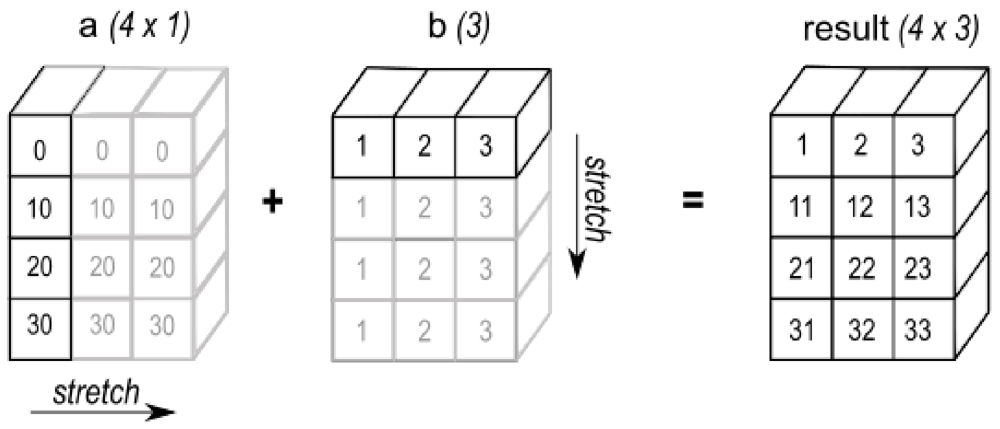
```
array([[ 0],
       [10],
       [20],
       [30]])
```

```python
y
```

```
array([1, 2, 3])
```

```python
x+y
```

```
array([[ 1,  2,  3],
       [11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```



Exercise 1

For each of the following combinations determine what the resulting dimension will be using broadcasting

- A [128 x 128 x 3] + B [3]
- A [8 x 1 x 6 x 1] + B [7 x 1 x 5]
- A [2 x 1] + B [8 x 4 x 3]
- A [3 x 1] + B [15 x 3 x 5]
- A [3] + B [4]

Demo 1 - Standardization

Below we generate a data set with 3 columns of random normal values. Each column has a different mean and standard deviation which we can check with `mean()` and `std()`.

```python
rng = np.random.default_rng(1234)
d = rng.normal(
  loc=[-1,0,1],
  scale=[1,2,3],
  size=(1000,3)
)
```

```python
d.shape
```

```
(1000, 3)
```

```python
d.mean(axis=0)
```

```
array([-1.02944, -0.01396,  1.01242])
```

```python
d.std(axis=0)
```

```
array([0.99675, 2.03223, 3.10625])
```

Lets use broadcasting to standardize all three columns to have mean 0 and standard deviation 1.

Broadcasting and assignment

In addition to arithmetic operators, broadcasting can be used with assignment via array indexing,

```
x = np.arange(12).reshape((3,4))
y = -np.arange(4)
z = -np.arange(3)
```

```
x[:] = y
x
```

```
array([[ 0, -1, -2, -3],
       [ 0, -1, -2, -3],
       [ 0, -1, -2, -3]])
```

```
x[...] = y
x
```

```
array([[ 0, -1, -2, -3],
       [ 0, -1, -2, -3],
       [ 0, -1, -2, -3]])
```

```
x[:] = z
```

```
ValueError: could not broadcast input
array from shape (3,) into shape (3,4)
```

```
x[:] = z.reshape((3,1))
x
```

```
array([[ 0,  0,  0,  0],
       [-1, -1, -1, -1],
       [-2, -2, -2, -2]])
```

JAX

JAX

> JAX is a library for array-oriented numerical computation (à la NumPy), with automatic differentiation and JIT compilation to enable high-performance machine learning research.
>
> - JAX provides a unified NumPy-like interface to computations that run on CPU, GPU, or TPU, in local or distributed settings.
> - JAX features built-in Just-In-Time (JIT) compilation via Open XLA, an open-source machine learning compiler ecosystem.
> - JAX functions support efficient evaluation of gradients via its automatic differentiation transformations.
> - JAX functions can be automatically vectorized to efficiently map them over arrays representing batches of inputs.

```python
import jax
jax.__version__
```

```
'0.5.0'
```

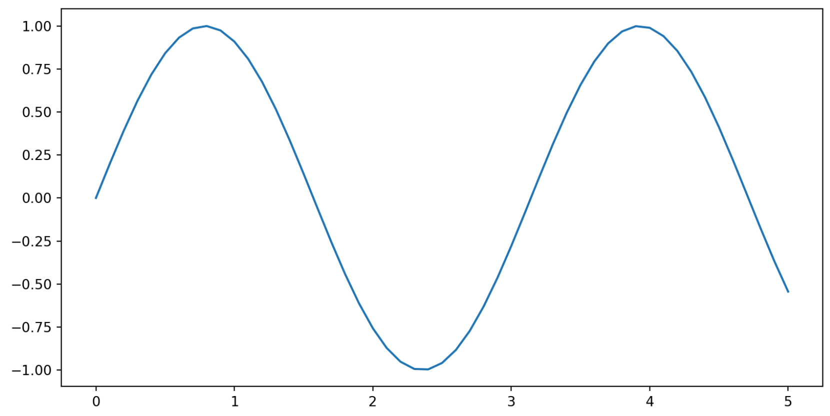
JAX & NumPy

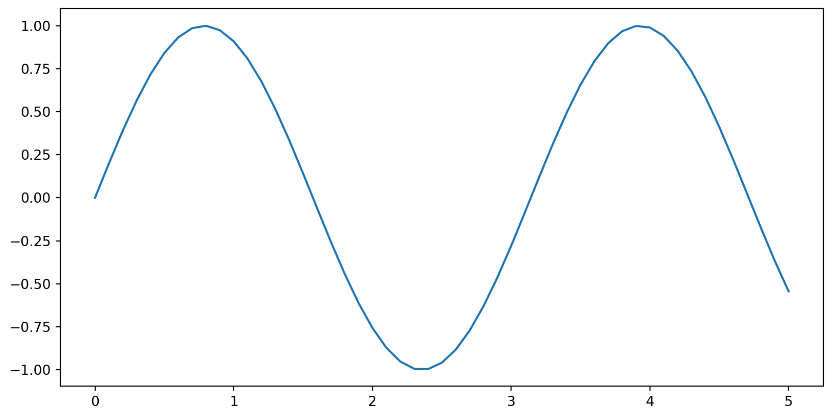
```python
import numpy as np

x_np = np.linspace(0, 5, 51)
y_np = 2 * np.sin(x_np) * np.cos(x_np)
plt.plot(x_np, y_np)
```

```python
import jax.numpy as jnp

x_jnp = jnp.linspace(0, 5, 51)
y_jnp = 2 * jnp.sin(x_jnp) * jnp.cos(x
plt.plot(x_jnp, y_jnp)
```





```python
type(x_np)
```

```
<class 'numpy.ndarray'>
```

```python
type(x_jnp)
```

```
<class 'jaxlib.xla_extension.ArrayImpl'>
```

```
x_np
```

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6,
0.7, 0.8, 0.9,
       1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6,
1.7, 1.8, 1.9,
       2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6,
2.7, 2.8, 2.9,
       3. , 3.1, 3.2, 3.3, 3.4, 3.5, 3.6,
3.7, 3.8, 3.9,
       4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6,
4.7, 4.8, 4.9,
       5. ])
```

```
x_jnp
```

```
Array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6,
0.7, 0.8, 0.9,
       1. , 1.1, 1.2, 1.3, 1.4, 1.5, 1.6,
1.7, 1.8, 1.9,
       2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6,
2.7, 2.8, 2.9,
       3. , 3.1, 3.2, 3.3, 3.4, 3.5, 3.6,
3.7, 3.8, 3.9,
       4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6,
4.7, 4.8, 4.9,
       5. ], dtype=float32)
```

```
x_np.dtype
```

```
dtype('float64')
```

```
x_jnp.dtype
```

```
dtype('float32')
```

Compatibility

```
y_mix = 2 * np.sin(x_jnp) * jnp.cos(x_np); y_mix
```

```
Array([ 0.      ,  0.19867,  0.38942,  0.56464,  0.71736,
        0.84147,  0.93204,  0.98545,  0.99957,  0.97385,
        0.9093 ,  0.8085 ,  0.67546,  0.5155 ,  0.33499,
        0.14112, -0.05837, -0.25554, -0.44252, -0.61186,
       -0.7568 , -0.87158, -0.9516 , -0.99369, -0.99616,
       -0.95892, -0.88345, -0.77276, -0.63127, -0.4646 ,
       -0.27942, -0.08309,  0.11655,  0.31154,  0.49411,
        0.65699,  0.79367,  0.89871,  0.96792,  0.99854,
        0.98936,  0.94073,  0.8546 ,  0.7344 ,  0.58492,
        0.41212,  0.22289,  0.02478, -0.17433, -0.36648,
       -0.54402], dtype=float32)
```

```
type(y_mix)
```

```
<class 'jaxlib.xla_extension.ArrayImpl'>
```

JAX Arrays

As we've just seen a JAX array is very similar to a numpy array but there are some important differences.

- JAX arrays are immutable

```
x = jnp.array([3, 2, 1])
x[0] = 2
```

```
TypeError: JAX arrays are immutable and do not support in-place item assignment.
Instead of x[idx] = y, use x = x.at[idx].set(y) or another .at[] method:
https://jax.readthedocs.io/en/latest/_autosummary/jax.numpy.ndarray.at.html
```

- because of the above JAX does not support inplace operations - the result will be a copy

```
y = x.sort()
y
```

```
Array([1, 2, 3], dtype=int32)
```

```
x
```

```
Array([3, 2, 1], dtype=int32)
```

```
np.shares_memory(x,y)
```

```
False
```

- The default JAX array dtype is `float32` not `float64`

```
jnp.array([1, 2, 3])
```

```
Array([1, 2, 3], dtype=int32)
```

```
jnp.array([1, 2, 3], dtype=jnp.float64)
```

```
UserWarning: Explicitly requested dtype <class 'jax.numpy.float64'> requested in
array is not available, and will be truncated to dtype float32. To enable more
dtypes, set the jax_enable_x64 configuration option or the JAX_ENABLE_X64 shell
environment variable. See https://github.com/jax-ml/jax#current-gotchas for more.

Array([1., 2., 3.], dtype=float32)
```

64-bit dtypes can be enabled by setting `jax_enable_x64=True` in the JAX configuration.

```
jax.config.update("jax_enable_x64", True)
```

```
jnp.array([1, 2, 3])
```

```
Array([1, 2, 3], dtype=int64)
```

```
jnp.array([1., 2., 3.])
```

```
Array([1., 2., 3.], dtype=float64)
```

- JAX arrays are allocated to one *or more* devices

```python
jax.devices()
```

```
[CpuDevice(id=0)]
```

```python
x.devices()
```

```
{CpuDevice(id=0)}
```

```python
x.sharding
```

```
SingleDeviceSharding(device=CpuDevice(id=0
memory_kind=unpinned_host)
```

- Using JAX interactively allows for the use of standard Python control flow (`if`, `while`, `for`, etc.) but this is not supported for some of JAX's more advanced operations (e.g. `jit` and `grad`)

  There are replacements for most of these constructs in JAX, but they are beyond the scope of today.

Random number generation

JAX vs NumPy

Pseudo random number generation in JAX is a bit different than with NumPy - the latter depends on a global state that is updated each time a random function is called.

NumPy's PRNG guarantees something called sequential equivalence which amounts to sampling N numbers sequentially is the same as sampling N numbers at once (e.g. a vector of length N).

```python
np.random.seed(0)
f"individually: {np.stack([np.random.uniform() for i in range(5)])}"
```

```
'individually: [0.54881 0.71519 0.60276 0.54488 0.42365]'
```

```python
np.random.seed(0)
f"at once: {np.random.uniform(size=5)}"
```

```
'at once: [0.54881 0.71519 0.60276 0.54488 0.42365]'
```

Parallelization & sequential equivalence

Sequential equivalence can be problematic in when using parallelization across multiple devices, consider the following code:

```
np.random.seed(0)

def bar():
  return np.random.uniform()

def baz():
  return np.random.uniform()

def foo():
  return bar() + 2 * baz()
```

How do we guarantee that we get consistent results if we don’t know the order that `bar()` and `baz()` will run?

PRNG keys

JAX makes use of *random keys* which are just a fancier version of random seeds - all of JAX’s random functions require a key be passed in.

```
key = jax.random.PRNGKey(1234); key
```

```
Array([   0, 1234], dtype=uint32)
```

```
jax.random.normal(key)
```

```
Array(-0.5402, dtype=float64)
```

```
jax.random.normal(key)
```

```
Array(-0.5402, dtype=float64)
```

```
jax.random.normal(key, shape=(3,))
```

```
Array([-0.5402 ,  0.43958, -0.01978], dtype=float64)
```

Note that JAX does not provide a sequential equivalence guarantee - this is so that it can support vectorization for the generation of PRN.

Splitting keys

Since a key is essentially a seed we do not want to reuse them (unless we want an identical output). Therefore to generate multiple different PRN we can split a key to deterministically generate two (or more) new keys.

```python
key11, key12 = jax.random.split(key)
f"{key=}"
```

```
'key=Array([   0, 1234], dtype=uint32)'
```

```python
f"{key11=}"
```

```
'key11=Array([1264997412, 2518116175],
dtype=uint32)'
```

```python
f"{key12=}"
```

```
'key12=Array([2877103387, 1697627890],
dtype=uint32)'
```

```python
key21, key22 = jax.random.split(key)
f"{key=}"
```

```
'key=Array([   0, 1234], dtype=uint32)'
```

```python
f"{key21=}"
```

```
'key21=Array([1264997412, 2518116175],
dtype=uint32)'
```

```python
f"{key22=}"
```

```
'key22=Array([2877103387, 1697627890],
dtype=uint32)'
```

```python
key3 = jax.random.split(key, num=3)
key3
```

```
Array([[1264997412, 2518116175],
       [2877103387, 1697627890],
       [2113592192,  603280156]], dtype=uint32)
```

```python
jax.random.normal(key, shape=(3,))
```

```
Array([-0.5402 ,  0.43958, -0.01978], dtype=float64)
```

```python
jax.random.normal(key11, shape=(3,))
```

```
Array([ 1.24104,  0.12019, -2.2399 ],
dtype=float64)
```

```python
jax.random.normal(key12, shape=(3,))
```

```
Array([ 0.07627, -1.3035 ,  0.86524],
dtype=float64)
```

```python
jax.random.normal(key21, shape=(3,))
```

```
Array([ 1.24104,  0.12019, -2.2399 ],
dtype=float64)
```

```python
jax.random.normal(key22, shape=(3,))
```

```
Array([ 0.07627, -1.3035 ,  0.86524],
dtype=float64)
```

```python
jax.random.normal(key3[0], shape=(3,))
```

```
Array([ 1.24104,  0.12019, -2.2399 ], dtype=float64)
```

```python
jax.random.normal(key3[1], shape=(3,))
```

```
Array([ 0.07627, -1.3035 ,  0.86524], dtype=float64)
```

```python
jax.random.normal(key3[2], shape=(3,))
```

```
Array([-0.02894,  1.05075, -2.22082], dtype=float64)
```

JAX & jit

JAX performance

```python
key = jax.random.PRNGKey(1234)
x_jnp = jax.random.normal(key, (1000,1000))
x_np = np.array(x_jnp)
```

```python
%timeit y = x_np @ x_np
```

```
1.09 ms ± 92.4 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```python
%timeit y = x_jnp @ x_jnp
```

```
3.42 ms ± 122 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```python
%timeit y = 3*x_np + x_np
```

```
514 µs ± 41 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```python
%timeit y = 3*x_jnp + x_jnp
```

```
413 µs ± 24.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

jit

```python
def SELU_np(x, α=1.67, λ=1.05):
  "Scaled Exponential Linear Unit"
  return λ * np.where(x > 0, x, α * np.exp(
```

```python
x = np.arange(1e6)
%timeit y = SELU_np(x)
```

```
4.08 ms ± 80 µs per loop (mean ± std. dev.
of 7 runs, 100 loops each)
```

```python
SELU_np_jit = jax.jit(SELU_np)
```

```python
%timeit y = SELU_np_jit(x)
```

```
TracerArrayConversionError: The
numpy.ndarray conversion method
__array__() was called on traced array
with shape float32[1000000]
```

```python
def SELU_jnp(x, α=1.67, λ=1.05):
  "Scaled Exponential Linear Unit"
  return λ * jnp.where(x > 0, x, α * jnp.ex
```

```python
x = jnp.arange(1e6)
%timeit y = SELU_jnp(x)
```

```
1.58 ms ± 68.8 µs per loop (mean ± std.
dev. of 7 runs, 1,000 loops each)
```

```python
SELU_jnp_jit = jax.jit(SELU_jnp)
```

```python
%timeit y = SELU_jnp_jit(x)
```

```
418 µs ± 13 µs per loop (mean ± std. dev.
of 7 runs, 1,000 loops each)
```

jit limitations

When it works the jit tool is fantastic, but it does have a number of limitations,

- Must use pure functions (no side effects)
- Must primarily use JAX functions
  - e.g. use `jnp.minimum()` not `np.minimum()` or `min()`
- Must generally avoid conditionals / control flow
- Issues around concrete values when tracing (static values)
- Check performance - there are not always gains + there is the initial cost of compilation

Automatic differentiation

Basics

The `grad()` function takes a numerical function, returning a scalar, and returns a function for calculating the gradient of that function.

```python
def f(x):
  return x**2
```

```python
f(3.)
```

```
9.0
```

```python
jax.grad(f)(3.)
```

```
Array(6., dtype=float64,
weak_type=True)
```

```python
jax.grad(
  jax.grad(f)
)(3.)
```

```
Array(2., dtype=float64,
weak_type=True)
```

```python
def g(x):
  return jnp.exp(-x)
```

```python
g(1.)
```

```
Array(0.36788, dtype=float64,
weak_type=True)
```

```python
jax.grad(g)(1.)
```

```
Array(-0.36788, dtype=float64,
weak_type=True)
```

```python
jax.grad(
  jax.grad(g)
)(1.)
```

```
Array(0.36788, dtype=float64,
weak_type=True)
```

```python
def h(x):
  return jnp.maximum(0,x)
```

```python
h(-2.)
```

```
Array(0., dtype=float64,
weak_type=True)
```

```python
h(2.)
```

```
Array(2., dtype=float64,
weak_type=True)
```

```python
jax.grad(h)(-2.)
```

```
Array(0., dtype=float64,
weak_type=True)
```

```python
jax.grad(h)(2.)
```

```
Array(1., dtype=float64,
weak_type=True)
```

Aside - `vmap()`

I would like to plot `h()` and `jax.grad(h)()` - lets see what happens,

```
x = jnp.linspace(-3,3,101)
y = h(x)
y_grad = jax.grad(h)(x)
```

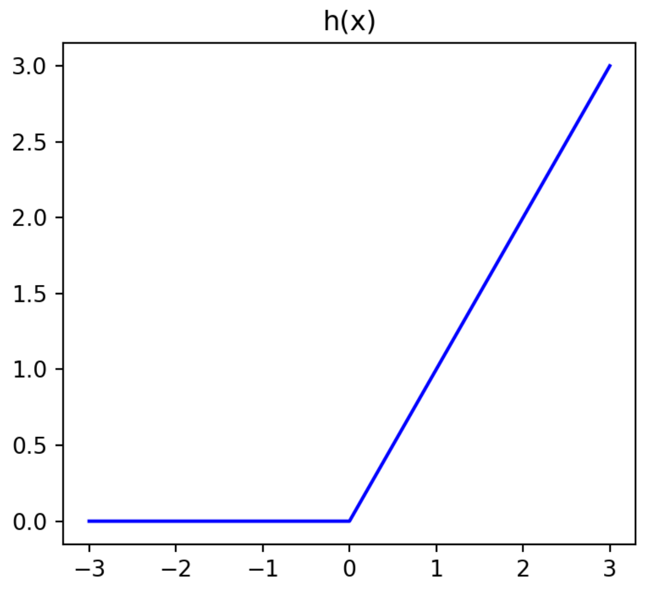
```
TypeError: Gradient only defined for scalar-output functions. Output had shape:
(101,).
--------------------
For simplicity, JAX has removed its internal frames from the traceback of the
following exception. Set JAX_TRACEBACK_FILTERING=off to include these.
```

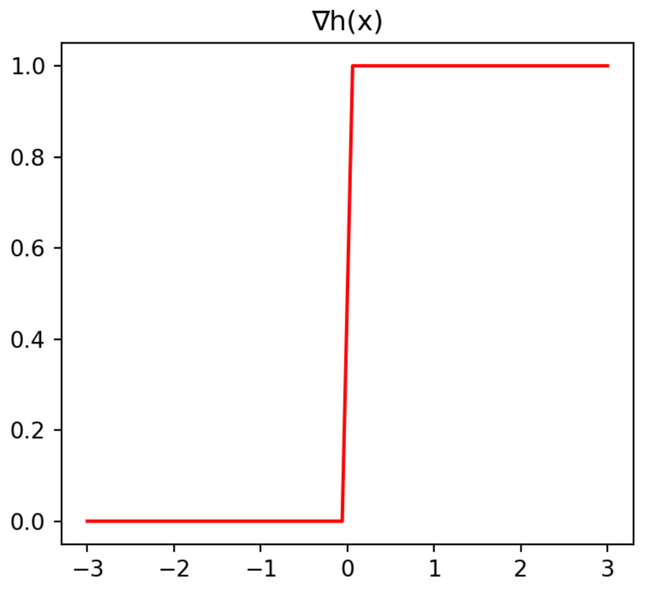
As just mentiond, we can only calculate the gradient for scalar valued functions. However, we can transform our scalar function into a vectorized function using `vmap()`.

```
h_grad = jax.vmap(
  jax.grad(h)
)
y_grad = h_grad(x)
```

```
y_grad
```

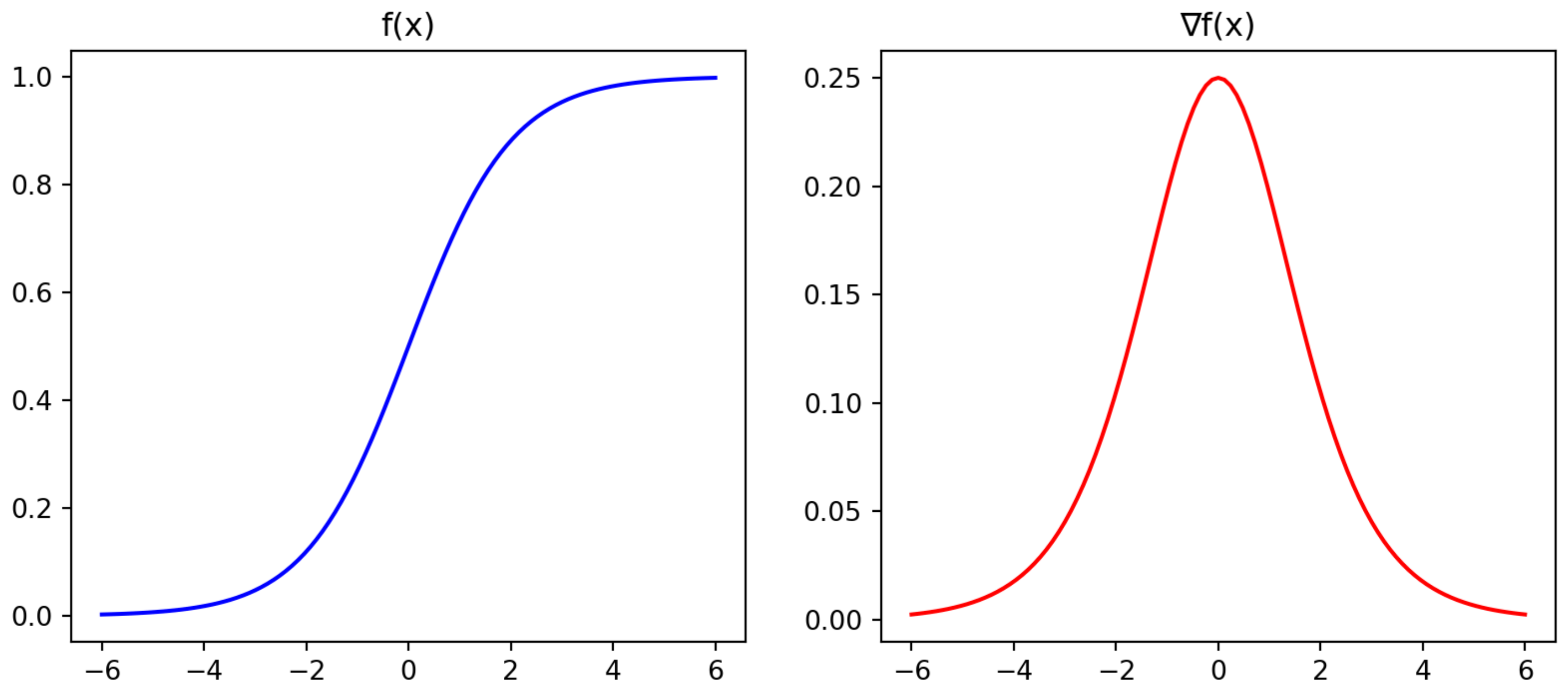
```
Array([0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
       0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
       0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
       0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
       0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ,
       0.5, 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. , 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. ], dtype=float64)
```





Another quick example

```python
x = jnp.linspace(-6,6,101)
f = lambda x: 0.5 * (jnp.tanh(x / 2) + 1)
y = f(x)
y_grad = jax.vmap(jax.grad(f))(x)
```


f(x)
∇f(x)
1.0
0.8
0.6
0.4
0.2
0.0
0.25
0.20
0.15
0.10
0.05
0.00
−6
−4
−2
0
2
4
6