NumPy Broadcasting & JAX

Lecture 07

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 in 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 custom binary format.

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

```
1 new_x = np.load("data/x.npy")
2 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()`,

```
1 with open("data/mtcars.csv") as file:
2     mtcars = np.genfromtxt(file, delimiter=",", skip_header=True)
3
4 mtcars
```

```
array([[  6.   , 160.   , 110.   ,   3.9  ,   2.62 ,  16.46 ,   0.   ,   1.   ,   4.
       [  6.   , 160.   , 110.   ,   3.9  ,   2.875,  17.02 ,   0.   ,   1.   ,   4.
       [  4.   , 108.   ,  93.   ,   3.85 ,   2.32 ,  18.61 ,   1.   ,   1.   ,   4.
       [  6.   , 258.   , 110.   ,   3.08 ,   3.215,  19.44 ,   1.   ,   0.   ,   3.
       [  8.   , 360.   , 175.   ,   3.15 ,   3.44 ,  17.02 ,   0.   ,   0.   ,   3.
       [  6.   , 225.   , 105.   ,   2.76 ,   3.46 ,  20.22 ,   1.   ,   0.   ,   3.
       [  8.   , 360.   , 245.   ,   3.21 ,   3.57 ,  15.84 ,   0.   ,   0.   ,   3.
       [  4.   , 146.7  ,  62.   ,   3.69 ,   3.19 ,  20.   ,   1.   ,   0.   ,   4.
       [  4.   , 140.8  ,  95.   ,   3.92 ,   3.15 ,  22.9  ,   1.   ,   0.   ,   4.
       [  6.   , 167.6  , 123.   ,   3.92 ,   3.44 ,  18.3  ,   1.   ,   0.   ,   4.
       [  6.   , 167.6  , 123.   ,   3.92 ,   3.44 ,  18.9  ,   1.   ,   0.   ,   4.
       [  8.   , 275.8  , 180.   ,   3.07 ,   4.07 ,  17.4  ,   0.   ,   0.   ,   3.
       [  8.   , 275.8  , 180.   ,   3.07 ,   3.73 ,  17.6  ,   0.   ,   0.   ,   3.
       [  8.   , 275.8  , 180.   ,   3.07 ,   3.78 ,  18.   ,   0.   ,   0.   ,   3.
       [  8.   , 472.   , 205.   ,   2.93 ,   5.25 ,  17.98 ,   0.   ,   0.   ,   3.
       [  8.   , 460.   , 215.   ,   3.   ,   5.424,  17.82 ,   0.   ,   0.   ,   3.
```

```
[  8.   , 440.   , 230.   ,   3.23 ,   5.345,  17.42 ,   0.   ,   0.   ,   3.
[  4.   ,  78.7  ,  66.   ,   4.08 ,   2.2  ,  19.47 ,   1.   ,   1.   ,   4.
[  4.   ,  75.7  ,  52.   ,   4.93 ,   1.615,  18.52 ,   1.   ,   1.   ,   4.
```

Broadcasting

Broadcasting

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

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

```
1 x * 2
```

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

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

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

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

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

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

```
ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

Simplicity & Efficiency

Broadcast code is usually shorter / simpler and it can make the calculation more efficient,

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

```
1 %timeit -n 1000 x * 2
```

```
11.2 µs ± 302 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
1 %timeit -n 1000 x * np.array([2])
```

```
11.4 µs ± 197 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
1 %timeit -n 1000 x * y
```

```
45.7 µs ± 838 ns per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
1 %timeit -n 1000 x * np.array([2]).repeat(1e5)
```

```
72.7 µs ± 830 ns per loop (mean ± std. dev. of 7 runs, 1,000 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?

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

```
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
```

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

```
ValueError: operands could not be broadcast
```

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

A fix

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

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

```
1 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)

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

```
1  x
```

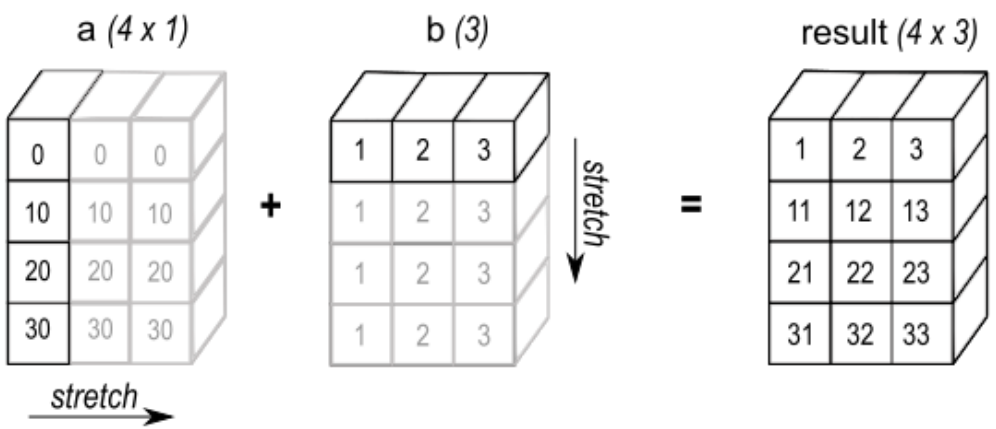
```
array([[ 0],
       [10],
       [20],
       [30]])
```

```
1  x+y
```

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

```
1  y
```

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



Unintended broadcasting

A common Broadcasting pitfalls is accidental broadcasting where operations succeed but produce unexpected results,

```
data = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])
row_means = data.mean(axis=1)  # mean of each row: [2, 5, 8]
```

Now we'd like to substract each row's mean from that row.

```
data - row_means
```

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

This "worked" but gave the wrong answer. The `(3,)` array broadcast across columns, not rows.

```
data - row_means.reshape(3, 1)
```

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

1d, row, and column vectors

Easy to get confused between `(n,)`, `(n,1)`, and `(1,n)` shapes' behavior,

```
1 v = np.array([1, 2, 3])
2 v.shape
```

```
(3,)
```

```
1 v_col = v.reshape(3, 1)
2 v_col.shape
```

```
(3, 1)
```

```
1 v_row = v.reshape(1, 3)
2 v_row.shape
```

```
(1, 3)
```

```
1 m = np.ones((3, 3))
```

```
1 m + v
```

```
array([[2., 3., 4.],
       [2., 3., 4.],
       [2., 3., 4.]])
```

```
1 m + v_col
```

```
array([[2., 2., 2.],
       [3., 3., 3.],
       [4., 4., 4.]])
```

```
1 m + v_row
```

```
array([[2., 3., 4.],
       [2., 3., 4.],
       [2., 3., 4.]])
```

All three succeed but `v` and `v_row` broadcast across rows while `v_col` broadcasts across columns.

Size explosion

Broadcasting doesn't copy data during the operation, but the *result* can be much larger than expected.

```
a = np.zeros((1000, 1))      # ~8 KB
b = np.zeros((1, 1000))      # ~8 KB
c = a + b                    # ~8 MB
c.shape
```

```
(1000, 1000)
```

You can use `np.broadcast_shapes()` to quick check the result shape before operating:

```
np.broadcast_shapes(a.shape, b.shape)
```

```
(1000, 1000)
```

```
np.broadcast_shapes((10000, 1), (1, 10000))
```

```
(10000, 10000)
```

Broadcasting with matrix multiplication

The `@` operator (and `np.matmul()`) has special broadcasting rules:

- The last two dimensions are treated as matrices and must be compatible for matrix multiplication (inner dimensions must match)
- Broadcasting applies only to the "batch" dimensions (all dimensions except the last two)

```
1 A = np.ones((3, 2, 4))
2 B = np.ones((4, 5))
3 (A @ B).shape
```

```
(3, 2, 5)
```

```
1 A = np.ones((2, 1, 2, 4))
2 B = np.ones((3, 4, 5))
3 (A @ B).shape
```

```
(2, 3, 2, 5)
```

```
1 A = np.ones((3, 2, 4))
2 B = np.ones((3, 4, 5))
3 (A @ B).shape
```

```
(3, 2, 5)
```

```
1 A = np.ones((2, 2, 4))
2 B = np.ones((3, 4, 5))
3 A @ B
```

```
ValueError: operands could not be broadcast toge
```

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()`.

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

```
d.shape
```

```
(1000, 3)
```

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

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

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

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

Let's 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,

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

```
1 x[:] = y
2 x
```

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

```
1 x[...] = y
2 x
```

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

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

```
ValueError: could not broadcast input array from sh
```

```
1 x[:] = z.reshape((3,1))
2 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.

```
1 import jax
2 jax.__version__
```

```
'0.9.0'
```

A Brief History

JAX is just one framework in a long history going back almost 20 years. These frameworks all share a common approach: representing computations as a graph of operations.

This enables automatic differentiation (via backpropagation) and optimization/compilation for efficient execution on GPUs or other specialized hardware.

- Theano (2007) - U of Montreal
- TensorFlow (2015) - Google Brain
- PyTorch (2016) - Facebook
- JAX (2018) - Google

JAX & NumPy

```
1 import numpy as np
2
3 x_np = np.arange(6).reshape(2, 3)
4 x_np
```

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

```
1 np.sum(x_np ** 2)
```

```
np.int64(55)
```

```
1 type(x_np)
```

```
numpy.ndarray
```

```
1 import jax.numpy as jnp
2
3 x_jnp = jnp.arange(6).reshape(2, 3)
4 x_jnp
```

```
Array([[0, 1, 2],
       [3, 4, 5]], dtype=int32)
```

```
1 jnp.sum(x_jnp ** 2)
```

```
Array(55, dtype=int32)
```

```
1 type(x_jnp)
```

```
jaxlib._jax.ArrayImpl
```

Compatibility

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

```
Array([[ 0.     ,  0.9093 , -0.7568 ],
       [-0.27942,  0.98936, -0.54402]], dtype=float32)
```

```
1 type(y_mix)
```

```
jaxlib._jax.ArrayImpl
```

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

```
Array([[ 0.     ,  0.9093 , -0.7568 ],
       [-0.27942,  0.98936, -0.54402]], dtype=float32)
```

```
1 type(y_mix)
```

```
jaxlib._jax.ArrayImpl
```

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

```
array([[ 0.     ,  0.9093 , -0.7568 ],
       [-0.27942,  0.98936, -0.54402]])
```

```
1 type(y_mix)
```

```
numpy.ndarray
```

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*

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

```
TypeError: JAX arrays are immutable and do not support in-place item assignment. Instead of x[idx] = y, use
```

- Related to the above, JAX does not support in-place operations - these functions now create and return a copy of the array

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

```
1 y
```

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

```
1 x
```

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

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

```
False
```

- The default JAX array dtypes are 32 bits not 64 bits (i.e. `float32` not `float64` and `int32` not `int64`)

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

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

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

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

```
1 #| warning: true
2 jnp.array([1, 2, 3], dtype=jnp.float64)
```

```
UserWarning: Explicitly requested dtype 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.
jnp.array([1, 2, 3], dtype=jnp.float64)

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

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

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

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

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

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

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

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

```
1 jax.devices()
```

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

```
1 x.devices()
```

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

```
1 x.sharding
```

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

- 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 this lecture.

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).

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

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

```
1 np.random.seed(0)
2 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 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 as their first argument.

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

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

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

```
Array(1.1031, dtype=float32)
```

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

```
Array(1.1031, dtype=float32)
```

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

```
Array([ 1.1031 ,  0.86306, -0.33868], dtype=float32)
```

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

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.

```
1 key11, key12 = jax.random.split(key)
2 f"{key=}"
3 f"{key11=}"
4 f"{key12=}"
```

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

```
1 key21, key22 = jax.random.split(key)
2 f"{key=}"
3 f"{key21=}"
4 f"{key22=}"
```

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

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

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

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

```
Array([ 1.1031 ,  0.86306, -0.33868], dtype=float32)
```

```
jax.random.normal(key11, shape=(3,))
jax.random.normal(key12, shape=(3,))
```

```
Array([ 1.49837, -1.47306, -2.08758], dtyp
```

```
jax.random.normal(key21, shape=(3,))
jax.random.normal(key22, shape=(3,))
```

```
Array([ 1.49837, -1.47306, -2.08758], dtyp
```

```
jax.random.normal(key3[0], shape=(3,))
jax.random.normal(key3[1], shape=(3,))
jax.random.normal(key3[2], shape=(3,))
```

```
Array([ 0.32401,  1.3939 , -1.17673], dtype=float32)
```

JAX & jit

Just-in-time compilation

JAX's `jit()` function compiles a function using XLA (Accelerated Linear Algebra), which can significantly speed up execution by optimizing the computation graph and reducing Python overhead.

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

x = np.linspace(-10, 10, int(1e6))
%timeit y = SELU_np(x)
```

```
2.7 ms ± 34.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

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

x = jnp.linspace(-10, 10, 1000000)
%timeit -r 3 y = SELU_jnp(x)
```

```
342 µs ± 721 ns per loop (mean ± std. dev. of 3 runs, 1,000 loops each)
```

```
SELU_jnp_jit = jax.jit(SELU_jnp)
%timeit -r 3 y = SELU_jnp_jit(x)
```

```
160 µs ± 544 ns per loop (mean ± std. dev. of 3 runs, 10,000 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.

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

```
f(3.)
```

```
9.0
```

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

```
Array(6., dtype=float32, weak_t
```

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

```
Array(2., dtype=float32, weak_t
```

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

```
g(1.)
```

```
Array(0.36788, dtype=float32, w
```

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

```
Array(-0.36788, dtype=float32,
```

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

```
Array(0.36788, dtype=float32, w
```

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

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

```
Array(0., dtype=float32, weak_t
```

```
h(2.)
```

```
Array(2., dtype=float32, weak_t
```

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

```
Array(0., dtype=float32, weak_t
```

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

```
Array(1., dtype=float32, weak_t
```

Aside - `vmap()`

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

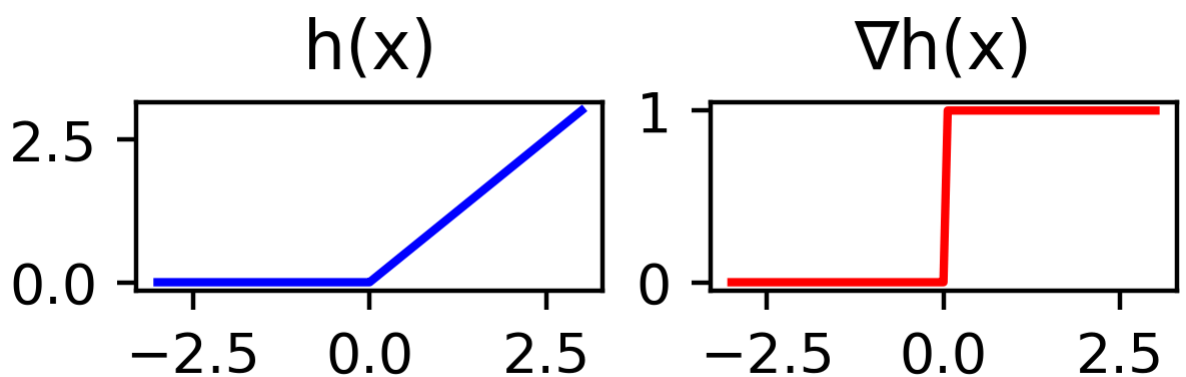
```
1 x = jnp.linspace(-3,3,101)
2 y = h(x)
3 y_grad = jax.grad(h)(x)
```

```
TypeError: Gradient only defined for scalar-output functions. Output had shape: (101,).
```

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

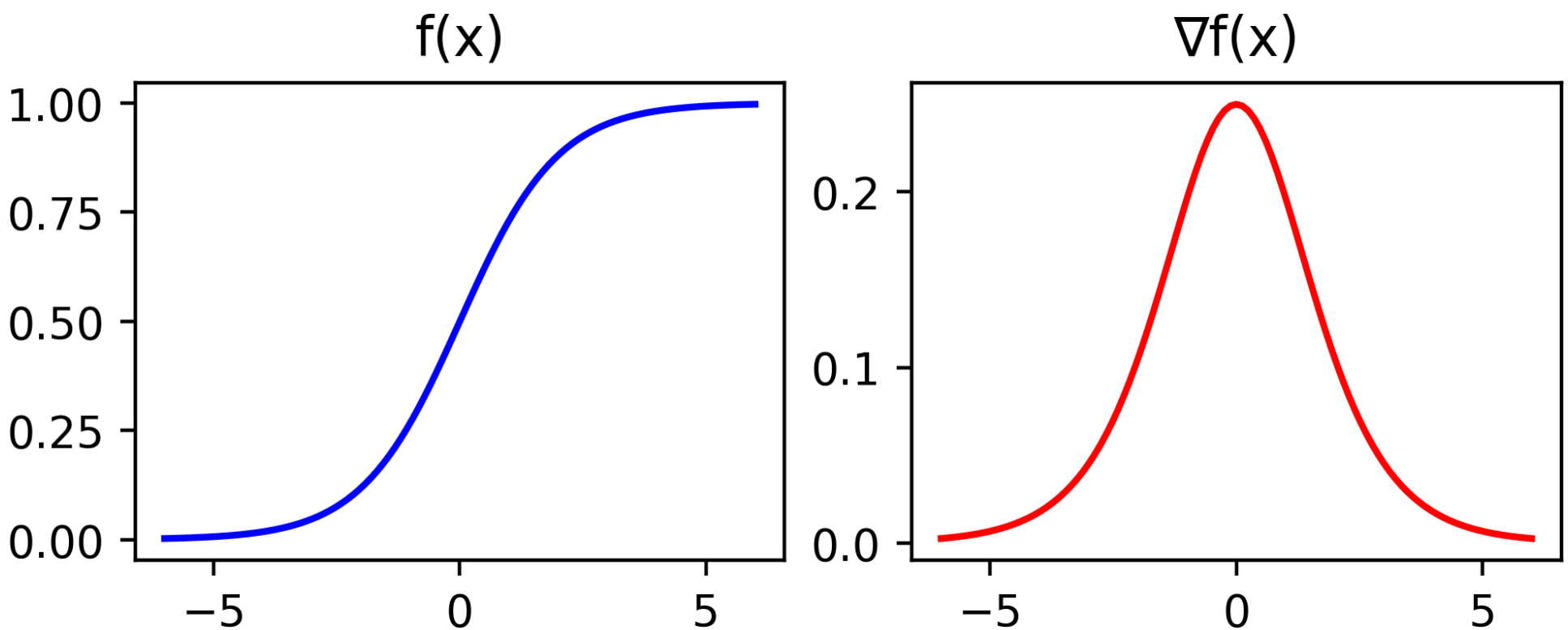
```
h_grad = jax.vmap(
  jax.grad(h)
)
y_grad = h_grad(x); y_grad
```

```
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., 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=float32)
```



Another quick example

```
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)
```



JAX & GPUs

Installation

JAX requires different packages depending on your hardware.

With uv:

```
# CPU only
uv add jax

# NVIDIA GPU (CUDA)
uv add jax[cuda13] # match the cuda version

# Apple Silicon (Metal)
uv add jax-metal
```

The GPU versions require appropriate drivers and libraries (CUDA toolkit for NVIDIA, Metal for Apple Silicon).

GPU Acceleration

One of JAX's main advantages is seamless GPU support. Once installed correctly, JAX automatically uses available GPUs without requiring code changes.

```
1 jax.devices()
```

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

To check which backend JAX is using:

```
1 jax.default_backend()
```

```
'cpu'
```

Arrays are automatically placed on the default device, but you can explicitly control placement:

```
1 # Place array on specific device
2 gpu = jax.devices('gpu')[0]
3 x_gpu = jax.device_put(x, gpu)
```

JAX performance (GPU)

```
1 key = jax.random.PRNGKey(1234)
2 x_jnp = jax.random.normal(key, (1000,1000))
3 x_np = np.array(x_jnp)
4 x_jnp.device
```

```
CudaDevice(id=0)
```

```
1 %timeit y = x_np @ x_np
```

```
1.55 ms ± 118 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
1 %timeit y = x_jnp @ x_jnp
```

```
99.8 µs ± 517 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
1 %timeit y = 3*x_np + x_np
```

```
152 µs ± 612 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
1 %timeit y = 3*x_jnp + x_jnp
```

```
59.4 µs ± 41.5 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```