JAX

Lecture 25

Dr. Colin Rundel

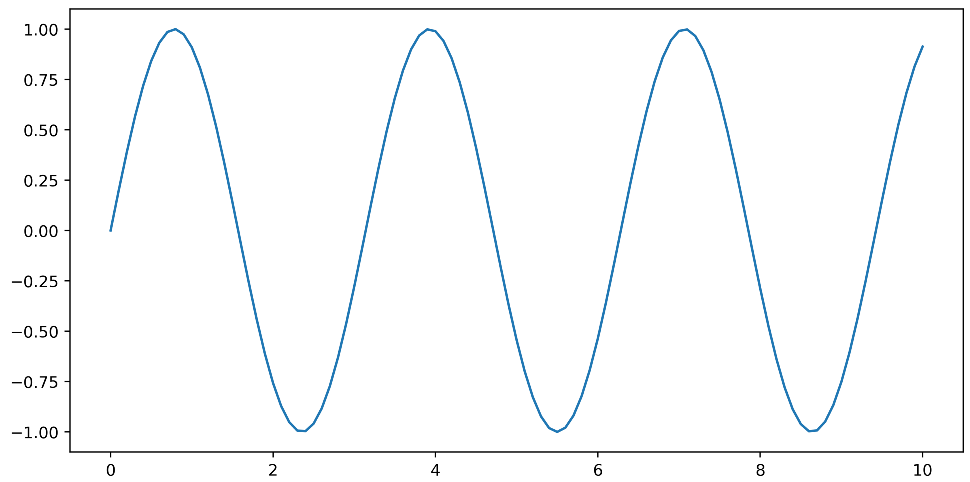
JAX

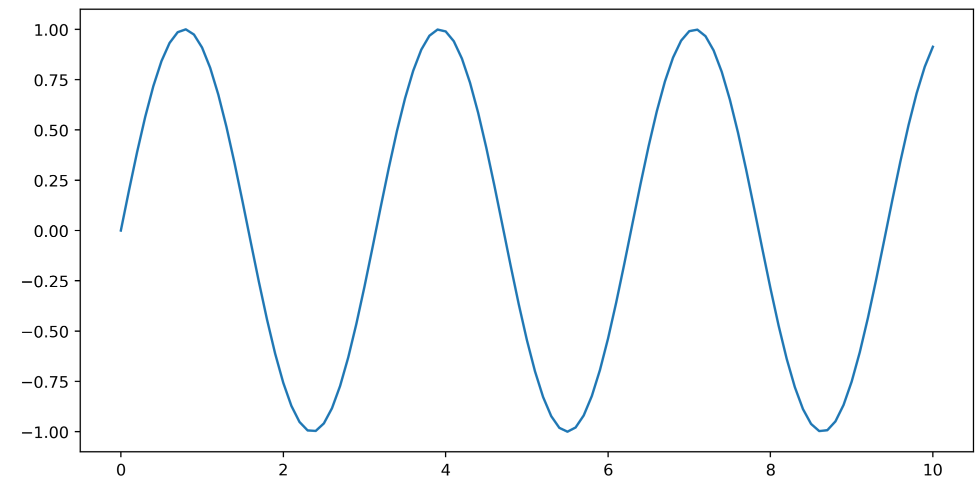
> JAX is NumPy on the CPU, GPU, and TPU, with great automatic differentiation for high-performance machine learning research.
>
> - JAX provides a NumPy-inspired interface for convenience (`jax.numpy`), can often be used as drop-in replacement
> - All JAX operations are implemented in terms of operations in XLA (Accelerated Linear Algebra compiler)
> - Supports sequential execution or JIT compilation
> - Updated autograd which can be used with native Python and NumPy functions

JAX & NumPy

```python
import numpy as np

x_np = np.linspace(0, 10, 101)
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, 10, 101)
y_jnp = 2 * jnp.sin(x_jnp) * jnp.cos(x_jnp)
plt.plot(x_jnp, y_jnp)
```






```python
type(x_np)
```

```
numpy.ndarray
```

```python
type(x_jnp)
```

```
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. ,  5.1,  5.2,  5.3,  5.4,  5.5
        5.6,  5.7,  5.8,  5.9,  6. ,  6.1,  6.2,  6.3
        6.4,  6.5,  6.6,  6.7,  6.8,  6.9,  7. ,  7.1
        7.2,  7.3,  7.4,  7.5,  7.6,  7.7,  7.8,  7.9
        8. ,  8.1,  8.2,  8.3,  8.4,  8.5,  8.6,  8.7
        8.8,  8.9,  9. ,  9.1,  9.2,  9.3,  9.4,  9.5
        9.6,  9.7,  9.8,  9.9, 10. ])
```

```
x_np.dtype
```

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

```
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. ,  5.1,  5.2,  5.3,  5.4,  5.5
        5.6,  5.7,  5.8,  5.9,  6. ,  6.1,  6.2,  6.3
        6.4,  6.5,  6.6,  6.7,  6.8,  6.9,  7. ,  7.1
        7.2,  7.3,  7.4,  7.5,  7.6,  7.7,  7.8,  7.9
        8. ,  8.1,  8.2,  8.3,  8.4,  8.5,  8.6,  8.7
        8.8,  8.9,  9. ,  9.1,  9.2,  9.3,  9.4,  9.5
        9.6,  9.7,  9.8,  9.9, 10. ], dtype=float32)
```

```
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, -0.69987, -0.82783, -0.92278, -0.98094,
       -0.99999, -0.97918, -0.91933, -0.82283, -0.69353,
       -0.53657, -0.35823, -0.1656 ,  0.03362,  0.23151,
        0.42017,  0.59207,  0.74038,  0.85916,  0.9437 ,
        0.99061,  0.99803,  0.96566,  0.89479,  0.78825,
        0.65029,  0.4864 ,  0.30312,  0.10775, -0.09191,
       -0.2879 , -0.47242, -0.63811, -0.77835, -0.88757,
       -0.9614 , -0.9969 , -0.99266, -0.94885, -0.8672 ,
       -0.75099, -0.60483, -0.43457, -0.24698, -0.04954,
        0.14988,  0.34331,  0.52307,  0.68196,  0.81367,
        0.91295], dtype=float32)
```

```
type(y_mix)
```

```
jaxlib.xla_extension.ArrayImpl
```

Aside - PRNG

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).

```
np.random.seed(0)
print("individually:", np.stack([np.random.uniform() for i in range(5)]))
```

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

```
np.random.seed(0)
print("all at once: ", np.random.uniform(size=5))
```

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

Parallelization & Sequential equivalence

Sequantial equivalence can be problematic in light of parallelization, 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 that a key be passed in.

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

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

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

```
Array(-0.5402, dtype=float32)
```

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

```
Array(-0.5402, dtype=float32)
```

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

```
Array([-0.01978, -0.0731 , -1.07825], dtype=float32)
```

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.

```
new_key1, sub_key1 = jax.random.split(key)
print(f"key     : {key}")
print(f"new_key1: {new_key1}")
print(f"sub_key1: {sub_key1}")
```

```
key     : [    0 1234]
new_key1: [2113592192 1902136347]
sub_key1: [603280156 445306386]
```

```
new_key2, sub_key2 = jax.random.split(key)
print(f"key     : {key}")
print(f"new_key2: {new_key2}")
print(f"sub_key2: {sub_key2}")
```

```
key     : [    0 1234]
new_key2: [2113592192 1902136347]
sub_key2: [603280156 445306386]
```

```
new_key3, *sub_keys3 = jax.random.split(key, num=3)
sub_keys3
```

```
[Array([1047329699,  140093922], dtype=uint32),
 Array([2907975018, 3484112841], dtype=uint32)]
```

JAX performance & jit

JAX performance

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

```
type(x_np)
```

```
numpy.ndarray
```

```
type(x_jnp)
```

```
jaxlib.xla_extension.ArrayImpl
```

```
x_np.shape
```

```
(1000, 1000)
```

```
x_jnp.shape
```

```
(1000, 1000)
```

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

```
9.56 ms ± 1.12 ms per loop (mean ± std. dev. of 7 rur
```

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

```
487 µs ± 7.72 µs per loop (mean ± std. dev. of 7 runs
```

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

```
3.38 ms ± 132 µs per loop (mean ± std. dev. of 7 runs
```

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

```
451 µs ± 20.1 µs per loop (mean ± std. dev. of 7 runs
```

```
%timeit y = (x_jnp @ x_jnp).block_until_ready()
```

```
3.77 ms ± 400 µs per loop (mean ± std. dev. of 7 runs
```

```
%timeit y = (3*x_jnp + x_jnp).block_until_ready(
```

```
509 µs ± 21.2 µs per loop (mean ± std. dev. of 7 runs
```

jit

```
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) -
```

```
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)
```

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

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

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

```
5.2 ms ± 172 µs per loop (mean ± std. dev. of 7 runs,
```

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

```
1.85 ms ± 47.5 µs per loop (mean ± std. dev. of 7 rur
```

```
%timeit y = SELU_np_jit(x).block_until_ready()
```

```
TracerArrayConversionError: The numpy.ndarray convers
The error occurred while tracing the function SELU_nj
See https://jax.readthedocs.io/en/latest/errors.html;
```

```
%timeit y = SELU_jnp_jit(x).block_until_ready()
```

```
363 µs ± 13 µs per loop (mean ± std. dev. of 7 runs,
```

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

autograd

Basics

Like with torch, 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_type
```

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

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

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

```
g(1.)
```

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

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

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

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

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

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

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

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

```
h(2.)
```

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

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

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

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

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

Aside - `vmap()`

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

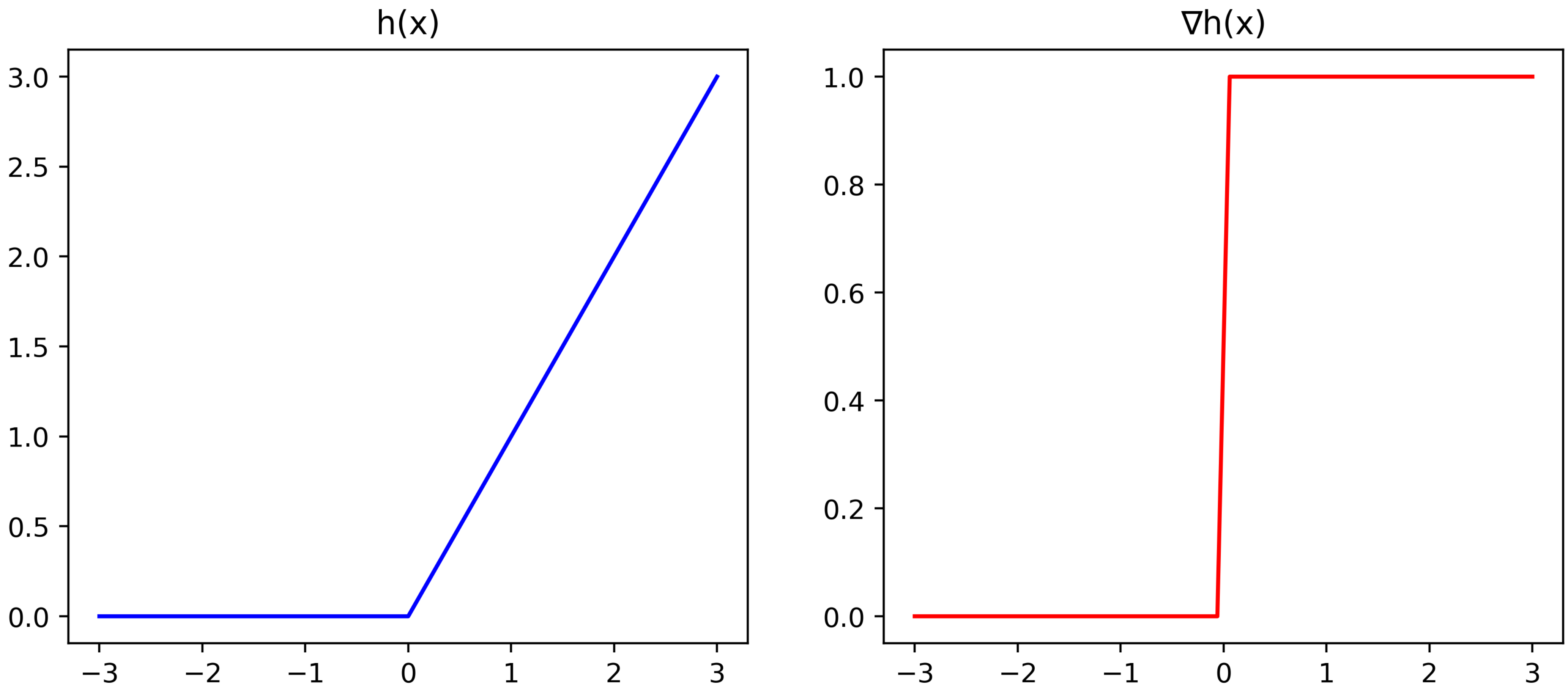
```
x = jnp.linspace(-3,3,101)
y = h(x)
y_grad = jax.grad(h)(x)
```

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

As mentiond on the previous slide - in order to calculate the gradient we need to apply it to a scalar valued function. We can transform our scalar function into a vectorized function using `vmap()`.

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

`vmap()` is significantly more powerful than just this as it allows for mapping over multiple input and output axes


h(x)
∇h(x)
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.8
0.6
0.4
0.2
−3
−2
−1
0
1
2
3

Regession example

```python
d = pd.read_csv("https://sta663-sp23.github.io/slides/data/ridge.
```

```
              y        x1        x2        x3        x4 x5
0   -0.151710  0.353658  1.633932  0.553257  1.415731  A
1    3.579895  1.311354  1.457500  0.072879  0.330330  B
2    0.768329 -0.744034  0.710362 -0.246941  0.008825  B
3    7.788646  0.806624 -0.228695  0.408348 -2.481624  B
4    1.394327  0.837430 -1.091535 -0.860979 -0.810492  A
..        ...       ...       ...       ...       ... ..
495 -0.204932 -0.385814 -0.130371 -0.046242  0.004914  A
496  0.541988  0.845885  0.045291  0.171596  0.332869  A
497 -1.402627 -1.071672 -1.716487 -0.319496 -1.163740  C
498 -0.043645  1.744800 -0.010161  0.422594  0.772606  A
499 -1.550276  0.910775 -1.675396  1.921238 -0.232189  B

[500 rows x 6 columns]
```

```python
X = jnp.array(
  pd.get_dummies(
    d.drop("y", axis=1)
  ).to_numpy(dtype = np.float3
)
X.shape
```

```
(500, 8)
```

```python
y = jnp.array(
  d.y.to_numpy(dtype = np.floa
)
y.shape
```

```
(500,)
```

Model & loss functions

```
def model(b, X=X):
  return X @ b

def reg_loss(b, λ=0., X=X, y=y, model=model):
  return jnp.mean((y - model(b,X).squeeze())**2)

def ridge_loss(b, λ=0., X=X, y=y, model=model):
  return jnp.mean((y - model(b,X).squeeze())**2) + λ * jnp.sum(b**2)

def lasso_loss(b, λ=0., X=X, y=y, model=model):
  return jnp.mean((y - model(b,X).squeeze())**2) + λ * jnp.sum(jnp.abs(b))
```

`grad()` of a multiargument function will take the gradient with respect to the first argument.

```
grad_reg_loss = jax.grad(reg_loss)
grad_ridge_loss = jax.grad(ridge_loss)
grad_lasso_loss = jax.grad(lasso_loss)
```

```python
key = jax.random.PRNGKey(1234)
b = jax.random.normal(key, (X.shape[1],1))
```

```python
grad_reg_loss(b)
```

```
Array([[-4.02278],
       [-4.46708],
       [ 0.08748],
       [ 5.57939],
       [-1.13703],
       [ 0.03037],
       [ 1.11083],
       [-0.30304]], dtype=float32)
```

```python
grad_ridge_loss(b, λ = 1)
```

```
Array([[-5.8394 ],
       [-5.74791],
       [ 0.68661],
       [ 5.13631],
       [-4.59595],
       [-0.35854],
       [ 3.54811],
       [-0.61224]], dtype=float32)
```

```python
grad_lasso_loss(b, λ = 1)
```

```
Array([[-5.02278],
       [-5.46708],
       [ 1.08748],
       [ 4.57939],
       [-2.13703],
       [-0.96963],
       [ 2.11083],
       [-1.30304]], dtype=float32)
```

sklearn

```
from sklearn.linear_model import LinearRegression

lm = LinearRegression(fit_intercept=False).fit(X,y)
lm.coef_
```

```
array([ 0.99505,  2.00762,  0.00232, -3.00088,  0.49329,
        0.10193, -0.29413,  1.00856], dtype=float32)
```

Fit implementation

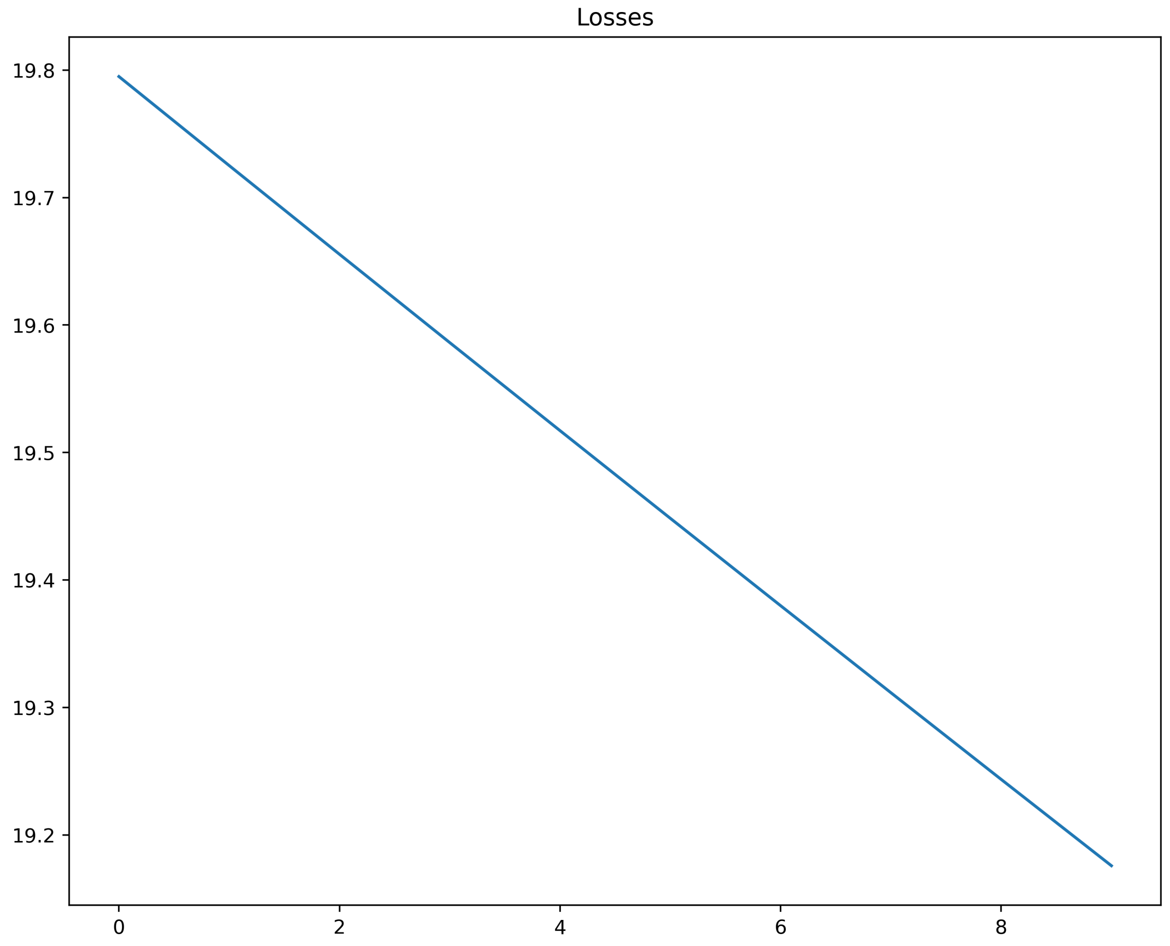
```
def fit(b, loss, λ=0., n=200, lr=0.001, X=X, y=y, model=model):
  val_grad = jax.value_and_grad(loss)

  losses = []
  for i in range(n):
    val, grad = val_grad(b, λ)
    losses.append(val.item())

    b -= lr * grad

  return (b, losses)
```

Linear regression

```
b = jax.random.normal(key, (X.shape[1],1))
b_hat, losses = fit(b, reg_loss, n=10)
```

```
b_hat
```

```
Array([[-0.86847],
       [-0.59607],
       [ 0.29865],
       [-0.27682],
       [-1.7181 ],
       [-0.19474],
       [ 1.20758],
       [-0.15159]], dtype=float32)
```

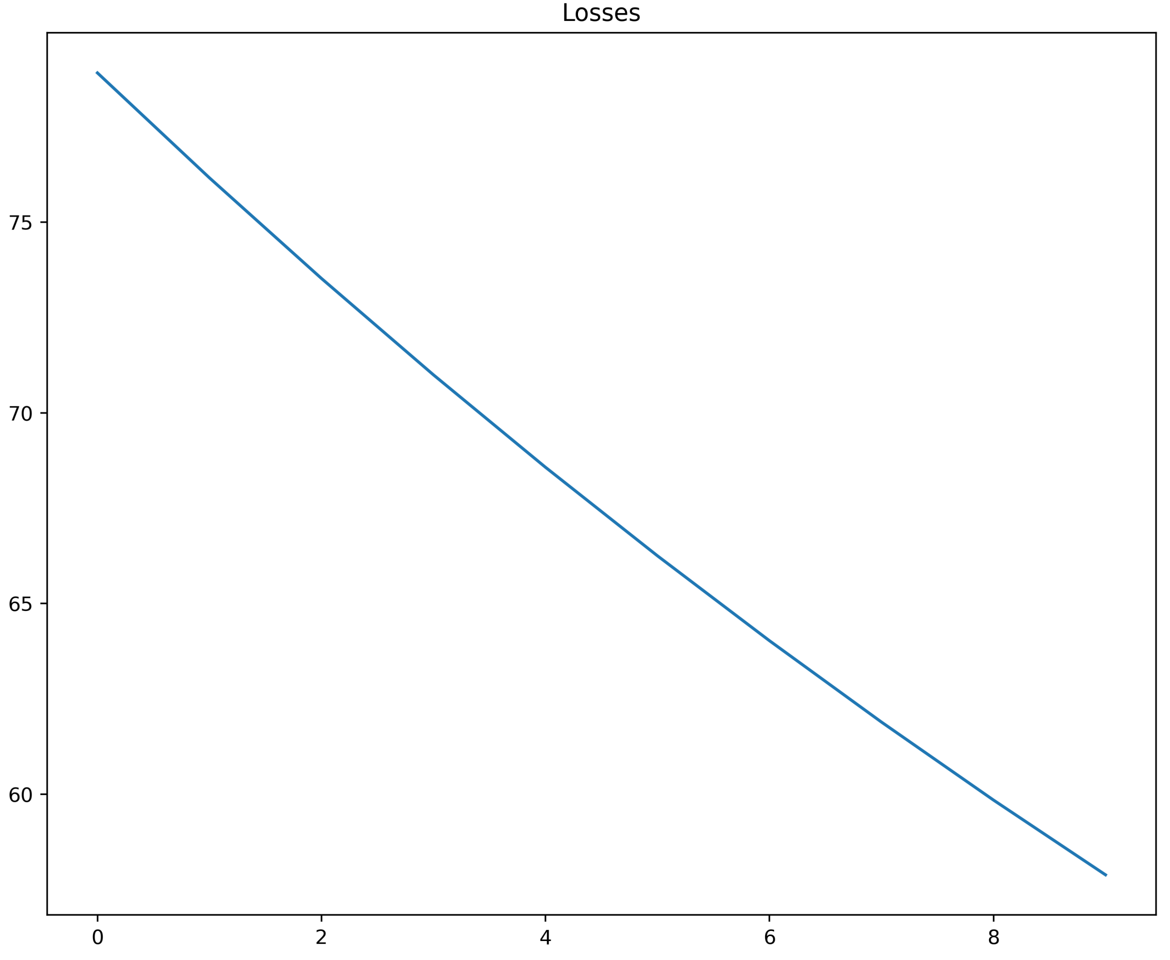


Ridge regression

```
b = jax.random.normal(key, (X.shape[1],1))
b_hat, losses = fit(b, ridge_loss, λ=10., n=10)
```

```
b_hat
```

```
Array([[-0.70718],
       [-0.48367],
       [ 0.24419],
       [-0.23206],
       [-1.40357],
       [-0.15921],
       [ 0.98619],
       [-0.12359]], dtype=float32)
```

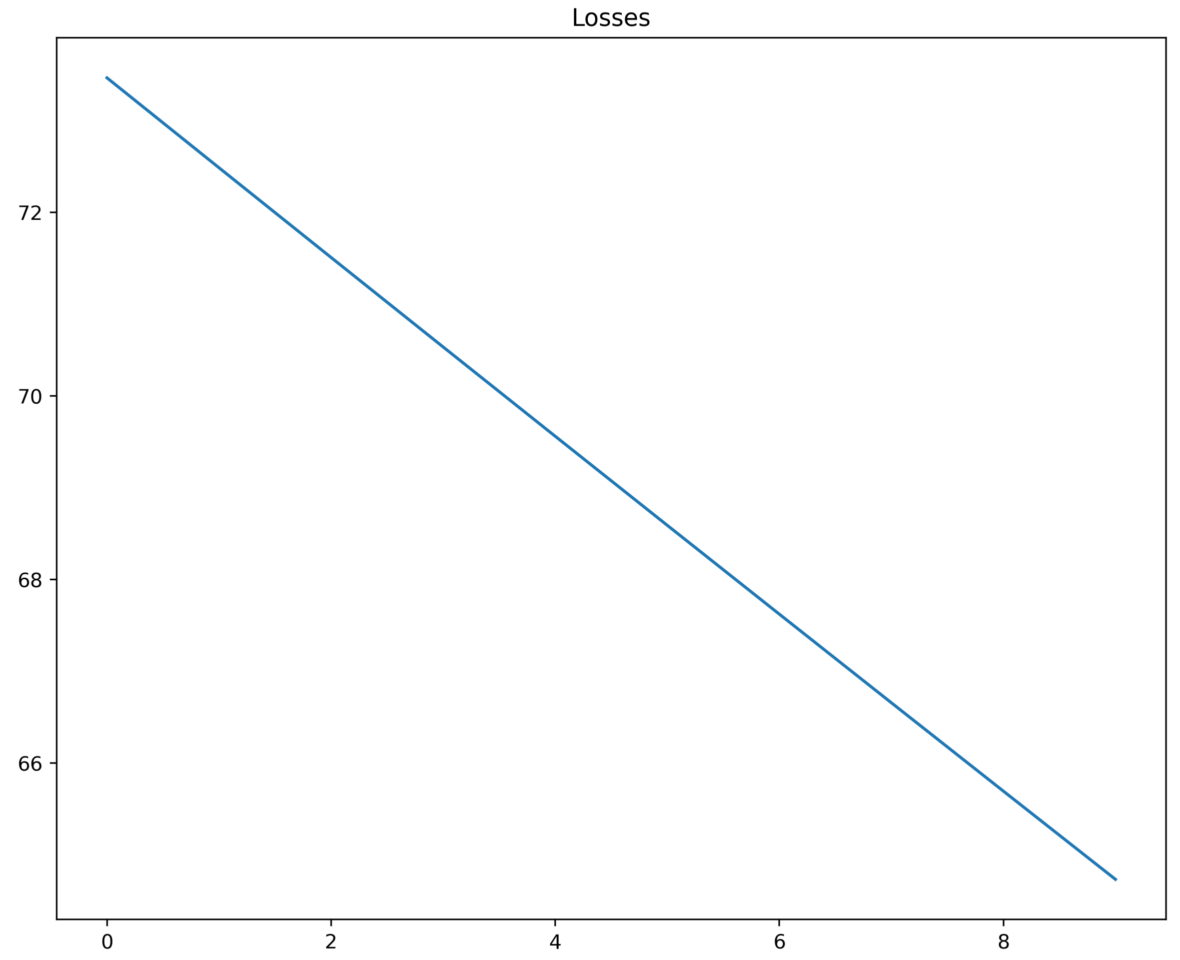


Lasso regression

```
b = jax.random.normal(key, (X.shape[1],1))
b_hat, losses = fit(b, lasso_loss, λ=10., n=10)
```

```
b_hat
```

```
Array([[-0.76927],
       [-0.49694],
       [ 0.19941],
       [-0.17782],
       [-1.61842],
       [-0.095  ],
       [ 1.10782],
       [-0.05166]], dtype=float32)
```



Jitting fit?

```
fit_jit = jax.jit(fit)
b_hat, losses = fit_jit(b, reg_loss, λ=10., n=10)
```

```
TypeError: Cannot interpret value of type <class 'function'> as an abstract array; it does not have a dtype
```

```
fit_jit = jax.jit(fit, static_argnames=["loss","λ","n","X","y","model"])
b_hat = fit_jit(b, reg_loss)
```

```
ConcretizationTypeError: Abstract tracer value encountered where concrete value is expected: Traced<ShapedAr
The problem arose with the `float` function. If trying to convert the data type of a value, try using `x.ast
The error occurred while tracing the function fit at /var/folders/ds/8sqz2v4d355btthn6r88kdc00000gn/T/ipyker

  operation a [35m:f32[500,1] [39m = dot_general[dimension_numbers=(([1], [0]), ([], []))] b c
    from line /var/folders/ds/8sqz2v4d355btthn6r88kdc00000gn/T/ipykernel_28187/530544497.py:2 (model)

  operation a [35m:f32[500] [39m = sub b c
    from line /var/folders/ds/8sqz2v4d355btthn6r88kdc00000gn/T/ipykernel_28187/530544497.py:5 (reg_loss)

  operation a [35m:f32[] [39m = div b c
    from line /var/folders/ds/8sqz2v4d355btthn6r88kdc00000gn/T/ipykernel_28187/530544497.py:5 (reg_loss)

  operation a [35m:f32[] [39m = div b c
    from line /var/folders/ds/8sqz2v4d355btthn6r88kdc00000gn/T/ipykernel_28187/530544497.py:5 (reg_loss)

  operation a [35m:f32[1,8] [39m = dot_general[dimension_numbers=(([0], [0]), ([], []))] b c
    from line /var/folders/ds/8sqz2v4d355btthn6r88kdc00000gn/T/ipykernel_28187/530544497.py:2 (model)
```

Simpler fit

```
def fit_simple(b, loss, λ=0., n=200, lr=0.001, X=X, y=y, model=model):
  grad = jax.grad(loss)

  for i in range(n):
    b -= lr * grad(b, λ)

  return b

b_hat = fit_simple(b, reg_loss)
```

```
fit_jit = jax.jit(fit_simple, static_argnames=["loss","λ","n","X","y","model"])
b_hat_jit = fit_jit(b, reg_loss)
```

Performance

```
%timeit b_hat = fit_simple(b, reg_loss, n=50)
```

```
92 ms ± 1.23 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit b_hat_jit = fit_jit(b, reg_loss, n=50)
```

```
66.7 µs ± 3.4 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit b_hat_jit = fit_jit(b, reg_loss, n=50).block_until_ready()
```

```
65.5 µs ± 470 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
b_hat
```

```
Array([[-0.25274],
       [ 0.12124],
       [ 0.26991],
       [-1.13775],
       [-1.50921],
       [-0.194  ],
       [ 1.01511],
       [-0.09929]], dtype=float32)
```

```
b_hat_jit
```

```
Array([[-0.25274],
       [ 0.12124],
       [ 0.26991],
       [-1.13775],
       [-1.50921],
       [-0.194  ],
       [ 1.01511],
       [-0.09929]], dtype=float32)
```

Pytrees

What is a pytrees?

> a pytree is a container of leaf elements and/or more pytrees. Containers include lists, tuples, and dicts. A leaf element is anything that's not a pytree, e.g. an array. In other words, a pytree is just a possibly-nested standard or user-registered Python container. If nested, note that the container types do not need to match. A single "leaf", i.e. a non-container object, is also considered a pytree.

Why do we need them?

> In machine learning, some places where you commonly find pytrees are:
>
> - Model parameters
> - Dataset entries

This helps us avoid functions with large argument lists and make it possible to vectorize / map more operations.

tree_map

JAX provides a number of built-in tools for working with / iterating over pytrees, `tree_map()` being the most commonly used,

```
list_of_lists = [
    [1, 2, 3],
    [1, 2],
    [1, 2, 3, 4]
]
```

```
jax.tree_map(
  lambda x: x**2,
  list_of_lists
)
```

```
[[1, 4, 9], [1, 4], [1, 4, 9, 16]]
```

```
jax.tree_map(
  lambda x,y: x+y,
  list_of_lists, list_of_lists
)
```

```
[[2, 4, 6], [2, 4], [2, 4, 6, 8]]
```

```
d = {
  'W': jnp.array([[1.,2.],[3.,4.],[5.,6.]]),
  'b': jnp.array([-1.,1.])
}
```

```
jax.tree_map(
  lambda p: (p-jnp.mean(p))/jnp.std(p),
  d
)
```

```
{'W': Array([[-1.46385, -0.87831],
        [-0.29277,  0.29277],
        [ 0.87831,  1.46385]], dtype=float32),
 'b': Array([-1.,  1.], dtype=float32)}
```

Nested trees

`tree_map()` will iterate and apply the desired function over *all* of the leaf elements while maintaining the structure of the pytree (similar to `rapply()` in R).

```
example_trees = [
    [1, 'a', object()],
    (1, (2, 3), ()),
    [1, {'k1': 2, 'k2': (3, 4)}, 5],
    {'a': 2, 'b': (2, 3)},
    jnp.array([1, 2, 3]),
]

jax.tree_map(type, example_trees)
```

```
[[int, str, object],
 (int, (int, int), ()),
 [int, {'k1': int, 'k2': (int, int)}, int],
 {'a': int, 'b': (int, int)},
 jaxlib.xla_extension.ArrayImpl]
```

FNN example - Parameter setup

```
def init_params(layer_widths, key):
  params = []
  for n_in, n_out in zip(layer_widths[:-1], layer_widths[1:]):
    key, new_key = jax.random.split(key)
    params.append(
      dict(
        W = jax.random.normal(new_key, shape=(n_in, n_out)) * np.sqrt(2/n_in),
        b = jnp.ones(shape=(n_out,))
      )
    )
  return params

key = jax.random.PRNGKey(1234)
params = init_params([1, 128, 128, 1], key)
```

```
jax.tree_map(lambda x: x.shape, params)
```

```
[{'W': (1, 128), 'b': (128,)},
 {'W': (128, 128), 'b': (128,)},
 {'W': (128, 1), 'b': (1,)}]
```

Model

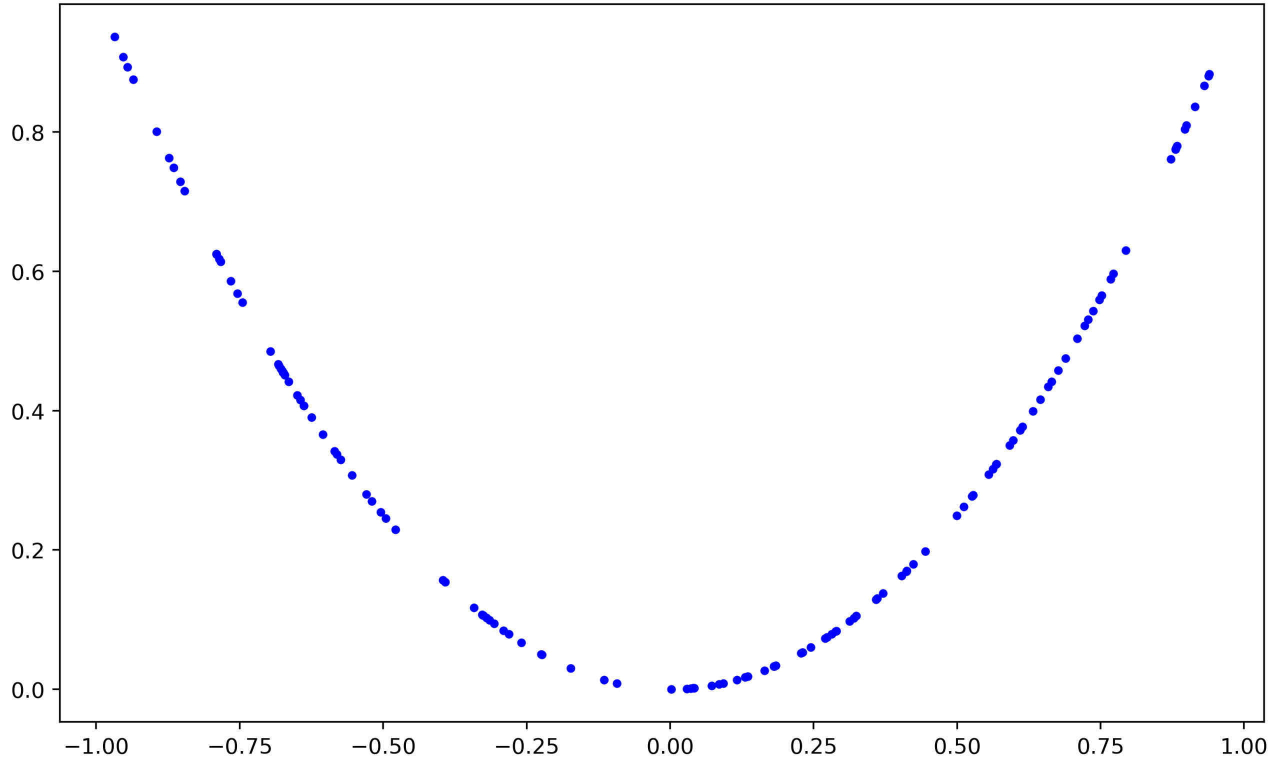
```python
from functools import partial

class model:
  def forward(self, params, x):
    *hidden, last = params
    for layer in hidden:
      x = x @ layer['W'] + layer['b']
      x = jax.nn.relu(x)
    return x @ last['W'] + last['b']

  def loss_fn(self, params, x, y):
    return jnp.mean((self.forward(params, x) - y) ** 2)

  @partial(jax.jit, static_argnames=['self', 'lr'])
  def step(self, params, x, y, lr=0.0001):
    grads = jax.grad(self.loss_fn)(params, x, y) # Note that since `params` is a pytree so will `grads`
    return jax.tree_map(
      lambda p, g: p - lr * g, params, grads
    )

  def fit(self, params, x, y, n = 1000):
    for i in range(n):
      params = self.step(params, x, y)
```

Data

```
key = jax.random.PRNGKey(12345)
x = jax.random.uniform(key, (128, 1), minval=-1., maxval=1.)
y = x**2
```

```
x.shape, y.shape
```

```
((128, 1), (128, 1))
```



Fitting

```
m = model()
```

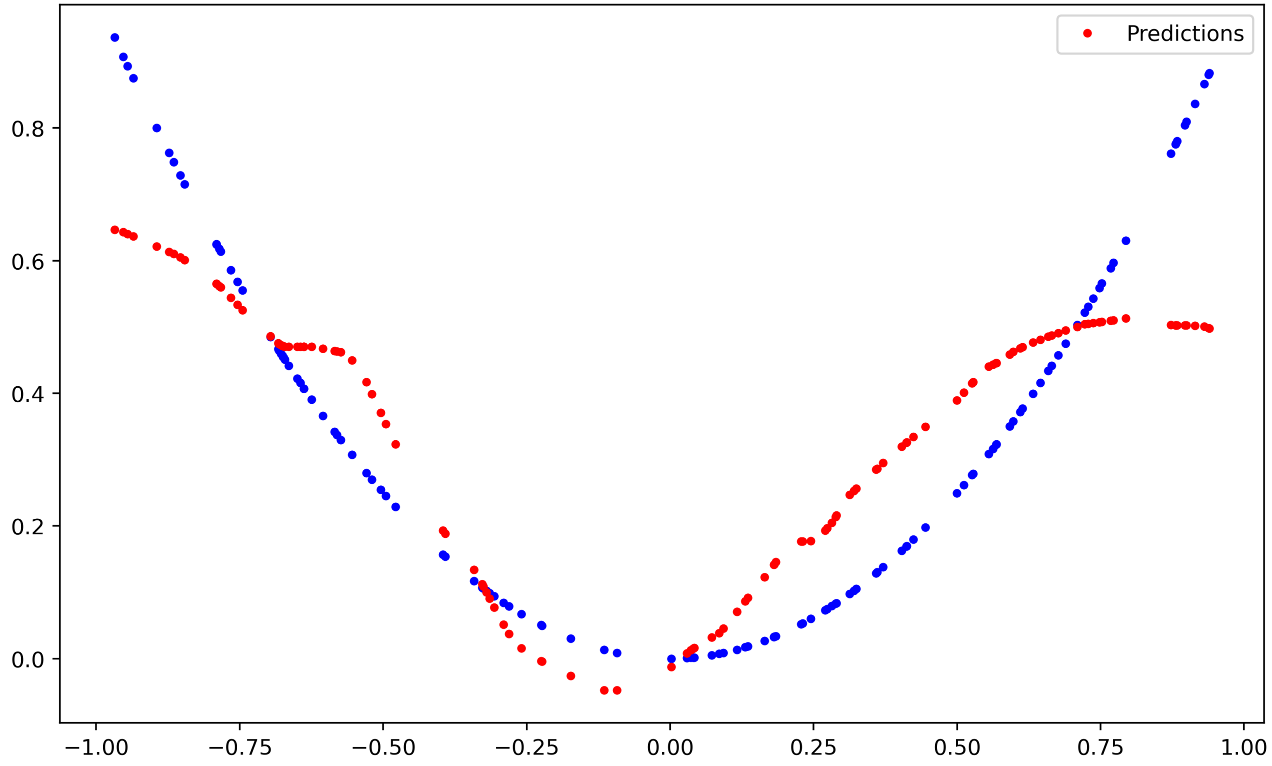
```
m.loss_fn(params, x, y)
```

```
Array(5.64001, dtype=float32)
```

```
params_fit = m.fit(params, x, y, n=1000)
m.loss_fn(params_fit, x, y)
```

```
Array(0.01788, dtype=float32)
```

```
y_hat = m.forward(params_fit, x)
```


Predictions
0.8
0.6
0.4
0.2
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

What next?

Additional Resources

There are a number of other libraries built on top of JAX that provide higher level interfaces for common tasks,

- Neural networks (torch-like interfaces)
  - flax - Google brain
  - haiku - DeepMind
  - equinox
- Bayesian models
  - BlackJAX - samplers for log-probability densities (optional backend for pymc)
- Other
  - Optax - gradient processing and optimization library (DeepMind)
  - Awesome-JAX - collection of JAX related links and resources