The Autodiff Cookbook

Contents

alexbw@, mattjj@

JAX has a pretty general automatic differentiation system. In this notebook, we'll go through a whole bunch of neat autodiff ideas that you can cherry pick for your own work, starting with the basics.

```
import jax.numpy as jnp
from jax import grad, jit, vmap
from jax import random

key = random.PRNGKey(0)
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun
for more info.)
```

Gradients

Starting with `grad`

You can differentiate a function with `grad`:

```
grad_tanh = grad(jnp.tanh)
print(grad_tanh(2.0))
```

```
0.070650816
```

`grad` takes a function and returns a function. If you have a Python function `f` that evaluates the mathematical function f, then `grad(f)` is a Python function that evaluates the mathematical function ∇f. That means `grad(f)(x)` represents the value $\nabla f(x)$.

Since `grad` operates on functions, you can apply it to its own output to differentiate as many times as you like:

```
print(grad(grad(jnp.tanh))(2.0))
print(grad(grad(grad(jnp.tanh)))(2.0))
```

```
-0.13621868
0.25265405
```

Let's look at computing gradients with `grad` in a linear logistic regression model. First, the setup:

```
def sigmoid(x):
    return 0.5 * (jnp.tanh(x / 2) + 1)

# Outputs probability of a label being true.
def predict(W, b, inputs):
    return sigmoid(jnp.dot(inputs, W) + b)

# Build a toy dataset.
inputs = jnp.array([[0.52, 1.12,  0.77],
                    [0.88, -1.08, 0.15],
                    [0.52, 0.06, -1.30],
                    [0.74, -2.49, 1.39]])
targets = jnp.array([True, True, False, True])

# Training loss is the negative log-likelihood of the training examples.
def loss(W, b):
    preds = predict(W, b, inputs)
    label_probs = preds * targets + (1 - preds) * (1 - targets)
    return -jnp.sum(jnp.log(label_probs))

# Initialize random model coefficients
key, W_key, b_key = random.split(key, 3)
W = random.normal(W_key, (3,))
b = random.normal(b_key, ())
```

Use the `grad` function with its `argnums` argument to differentiate a function with respect to positional arguments.

```
# Differentiate `loss` with respect to the first positional argument:
W_grad = grad(loss, argnums=0)(W, b)
print('W_grad', W_grad)

# Since argnums=0 is the default, this does the same thing:
W_grad = grad(loss)(W, b)
print('W_grad', W_grad)

# But we can choose different values too, and drop the keyword:
b_grad = grad(loss, 1)(W, b)
print('b_grad', b_grad)

# Including tuple values
W_grad, b_grad = grad(loss, (0, 1))(W, b)
print('W_grad', W_grad)
print('b_grad', b_grad)
```

```
W_grad [-0.16965583 -0.8774644  -1.4901346 ]
W_grad [-0.16965583 -0.8774644  -1.4901346 ]
b_grad -0.29227245
W_grad [-0.16965583 -0.8774644  -1.4901346 ]
b_grad -0.29227245
```

This `grad` API has a direct correspondence to the excellent notation in Spivak's classic *Calculus on Manifolds* (1965), also used in Sussman and Wisdom's *Structure and Interpretation of Classical Mechanics* (2015) and their *Functional Differential Geometry* (2013). Both books are open-access. See in particular the "Prologue" section of *Functional Differential Geometry* for a defense of this notation.

Essentially, when using the `argnums` argument, if `f` is a Python function for evaluating the mathematical function f, then the Python expression `grad(f, i)` evaluates to a Python function for evaluating $\partial_i f$.

Differentiating with respect to nested lists, tuples, and dicts

Differentiating with respect to standard Python containers just works, so use tuples, lists, and dicts (and arbitrary nesting) however you like.

```
def loss2(params_dict):
    preds = predict(params_dict['W'], params_dict['b'], inputs)
    label_probs = preds * targets + (1 - preds) * (1 - targets)
    return -jnp.sum(jnp.log(label_probs))

print(grad(loss2)({'W': W, 'b': b}))
```

```
{'W': Array([-0.16965583, -0.8774644 , -1.4901346 ], dtype=float32), 'b':
Array(-0.29227245, dtype=float32)}
```

You can register your own container types to work with not just `grad` but all the JAX transformations (`jit`, `vmap`, etc.).

Evaluate a function and its gradient using `value_and_grad`

Another convenient function is `value_and_grad` for efficiently computing both a function's value as well as its gradient's value:

```
from jax import value_and_grad
loss_value, Wb_grad = value_and_grad(loss, (0, 1))(W, b)
print('loss value', loss_value)
print('loss value', loss(W, b))
```

```
loss value 3.0519385
loss value 3.0519385
```

Checking against numerical differences

A great thing about derivatives is that they're straightforward to check with finite differences:

```
# Set a step size for finite differences calculations
eps = 1e-4

# Check b_grad with scalar finite differences
b_grad_numerical = (loss(W, b + eps / 2.) - loss(W, b - eps / 2.)) / eps
print('b_grad_numerical', b_grad_numerical)
print('b_grad_autodiff', grad(loss, 1)(W, b))

# Check W_grad with finite differences in a random direction
key, subkey = random.split(key)
vec = random.normal(subkey, W.shape)
unitvec = vec / jnp.sqrt(jnp.vdot(vec, vec))
W_grad_numerical = (loss(W + eps / 2. * unitvec, b) - loss(W - eps / 2. *
unitvec, b)) / eps
print('W_dirderiv_numerical', W_grad_numerical)
print('W_dirderiv_autodiff', jnp.vdot(grad(loss)(W, b), unitvec))
```

```
b_grad_numerical -0.29325485
b_grad_autodiff -0.29227245
W_dirderiv_numerical -0.2002716
W_dirderiv_autodiff -0.19909117
```

JAX provides a simple convenience function that does essentially the same thing, but checks up to any order of differentiation that you like:

```
from jax.test_util import check_grads
check_grads(loss, (W, b), order=2)  # check up to 2nd order derivatives
```

Hessian-vector products with `grad`-of-`grad`

One thing we can do with higher-order `grad` is build a Hessian-vector product function. (Later on we'll write an even more efficient implementation that mixes both forward- and reverse-mode, but this one will use pure reverse-mode.)

A Hessian-vector product function can be useful in a truncated Newton Conjugate-Gradient algorithm for minimizing smooth convex functions, or for studying the curvature of neural network training objectives (e.g. 1, 2, 3, 4).

For a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$ with continuous second derivatives (so that the Hessian matrix is symmetric), the Hessian at a point $x \in \mathbb{R}^n$ is written as $\partial^2 f(x)$. A Hessian-vector product function is then able to evaluate

$$v \mapsto \partial^2 f(x) \cdot v$$

for any $v \in \mathbb{R}^n$.

The trick is not to instantiate the full Hessian matrix: if n is large, perhaps in the millions or billions in the context of neural networks, then that might be impossible to store.

Luckily, `grad` already gives us a way to write an efficient Hessian-vector product function. We just have to use the identity

$$\partial^2 f(x) v = \partial [x \mapsto \partial f(x) \cdot v] = \partial g(x),$$

where $g(x) = \partial f(x) \cdot v$ is a new scalar-valued function that dots the gradient of f at x with the vector v. Notice that we're only ever differentiating scalar-valued functions of vector-valued arguments, which is exactly where we know `grad` is efficient.

In JAX code, we can just write this:

```
def hvp(f, x, v):
    return grad(lambda x: jnp.vdot(grad(f)(x), v))(x)
```

This example shows that you can freely use lexical closure, and JAX will never get perturbed or confused.

We'll check this implementation a few cells down, once we see how to compute dense Hessian matrices. We'll also write an even better version that uses both forward-mode and reverse-mode.

Jacobians and Hessians using `jacfwd` and `jacrev`

You can compute full Jacobian matrices using the `jacfwd` and `jacrev` functions:

```
from jax import jacfwd, jacrev

# Isolate the function from the weight matrix to the predictions
f = lambda W: predict(W, b, inputs)

J = jacfwd(f)(W)
print("jacfwd result, with shape", J.shape)
print(J)

J = jacrev(f)(W)
print("jacrev result, with shape", J.shape)
print(J)
```

```
jacfwd result, with shape (4, 3)
[[ 0.05981758  0.12883787  0.08857603]
 [ 0.04015916 -0.04928625  0.00684531]
 [ 0.12188288  0.01406341 -0.3047072 ]
 [ 0.00140431 -0.00472531  0.00263782]]
jacrev result, with shape (4, 3)
[[ 0.05981757  0.12883787  0.08857603]
 [ 0.04015916 -0.04928625  0.00684531]
 [ 0.12188289  0.01406341 -0.3047072 ]
 [ 0.00140431 -0.00472531  0.00263782]]
```

These two functions compute the same values (up to machine numerics), but differ in their implementation: `jacfwd` uses forward-mode automatic differentiation, which is more efficient for "tall" Jacobian matrices, while `jacrev` uses reverse-mode, which is more efficient for "wide" Jacobian matrices. For matrices that are near-square, `jacfwd` probably has an edge over `jacrev`.

You can also use `jacfwd` and `jacrev` with container types:

```
def predict_dict(params, inputs):
    return predict(params['W'], params['b'], inputs)

J_dict = jacrev(predict_dict)({'W': W, 'b': b}, inputs)
for k, v in J_dict.items():
    print("Jacobian from {} to logits is".format(k))
    print(v)
```

```
Jacobian from W to logits is
[[ 0.05981757  0.12883787  0.08857603]
 [ 0.04015916 -0.04928625  0.00684531]
 [ 0.12188289  0.01406341 -0.3047072 ]
 [ 0.00140431 -0.00472531  0.00263782]]
Jacobian from b to logits is
[0.11503381 0.04563541 0.23439017 0.00189771]
```

For more details on forward- and reverse-mode, as well as how to implement `jacfwd` and `jacrev` as efficiently as possible, read on!

Using a composition of two of these functions gives us a way to compute dense Hessian matrices:

```python
def hessian(f):
    return jacfwd(jacrev(f))

H = hessian(f)(W)
print("hessian, with shape", H.shape)
print(H)
```

```
hessian, with shape (4, 3, 3)
[[[ 0.02285465  0.04922541  0.03384247]
  [ 0.04922541  0.10602397  0.07289147]
  [ 0.03384247  0.07289147  0.05011288]]

 [[-0.03195215  0.03921401 -0.00544639]
  [ 0.03921401 -0.04812629  0.00668421]
  [-0.00544639  0.00668421 -0.00092836]]

 [[-0.01583708 -0.00182736  0.03959271]
  [-0.00182736 -0.00021085  0.00456839]
  [ 0.03959271  0.00456839 -0.09898177]]

 [[-0.00103524  0.00348343 -0.00194457]
  [ 0.00348343 -0.01172127  0.0065432 ]
  [-0.00194457  0.0065432  -0.00365263]]]
```

This shape makes sense: if we start with a function $f : \mathbb{R}^n \to \mathbb{R}^m$, then at a point $x \in \mathbb{R}^n$ we expect to get the shapes

- $f(x) \in \mathbb{R}^m$, the value of f at x,
- $\partial f(x) \in \mathbb{R}^{m \times n}$, the Jacobian matrix at x,
- $\partial^2 f(x) \in \mathbb{R}^{m \times n \times n}$, the Hessian at x,

and so on.

To implement `hessian`, we could have used `jacfwd(jacrev(f))` or `jacrev(jacfwd(f))` or any other composition of the two. But forward-over-reverse is typically the most efficient. That's because in the inner Jacobian computation we're often differentiating a function wide Jacobian (maybe like a loss function $f : \mathbb{R}^n \to \mathbb{R}$), while in the outer Jacobian computation we're differentiating a function with a square Jacobian (since $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$), which is where forward-mode wins out.

How it's made: two foundational autodiff functions

Jacobian-Vector products (JVPs, aka forward-mode autodiff)

JAX includes efficient and general implementations of both forward- and reverse-mode automatic differentiation. The familiar `grad` function is built on reverse-mode, but to explain the difference in the two modes, and when each can be useful, we need a bit of math background.

JVPs in math

Mathematically, given a function $f : \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian of f evaluated at an input point $x \in \mathbb{R}^n$, denoted $\partial f(x)$, is often thought of as a matrix in $\mathbb{R}^m \times \mathbb{R}^n$:

$$\partial f(x) \in \mathbb{R}^{m \times n}.$$

But we can also think of $\partial f(x)$ as a linear map, which maps the tangent space of the domain of f at the point x (which is just another copy of $\mathbb{R}^n$) to the tangent space of the codomain of f at the point $f(x)$ (a copy of $\mathbb{R}^m$):

$$\partial f(x) : \mathbb{R}^n \to \mathbb{R}^m.$$

This map is called the pushforward map of f at x. The Jacobian matrix is just the matrix for this linear map in a standard basis.

If we don't commit to one specific input point x, then we can think of the function ∂f as first taking an input point and returning the Jacobian linear map at that input point:

$$\partial f : \mathbb{R}^n \to \mathbb{R}^n \to \mathbb{R}^m.$$

In particular, we can uncurry things so that given input point $x \in \mathbb{R}^n$ and a tangent vector $v \in \mathbb{R}^n$, we get back an output tangent vector in $\mathbb{R}^m$. We call that mapping, from (x, v) pairs to output tangent vectors, the *Jacobian-vector product*, and write it as

$$(x, v) \mapsto \partial f(x) v$$

JVPs in JAX code

Back in Python code, JAX's `jvp` function models this transformation. Given a Python function that evaluates f, JAX's `jvp` is a way to get a Python function for evaluating $(x, v) \mapsto (f(x), \partial f(x) v)$.

```
from jax import jvp

# Isolate the function from the weight matrix to the predictions
f = lambda W: predict(W, b, inputs)

key, subkey = random.split(key)
v = random.normal(subkey, W.shape)

# Push forward the vector `v` along `f` evaluated at `W`
y, u = jvp(f, (W,), (v,))
```

In terms of Haskell-like type signatures, we could write

```
jvp :: (a -> b) -> a -> T a -> (b, T b)
```

where we use `T a` to denote the type of the tangent space for `a`. In words, `jvp` takes as arguments a function of type `a -> b`, a value of type `a`, and a tangent vector value of type `T a`. It gives back a pair consisting of a value of type `b` and an output tangent vector of type `T b`.

The `jvp`-transformed function is evaluated much like the original function, but paired up with each primal value of type `a` it pushes along tangent values of type `T a`. For each primitive numerical operation that the original function would have applied, the `jvp`-transformed function executes a "JVP rule" for that primitive that both evaluates the primitive on the primals and applies the primitive's JVP at those primal values.

That evaluation strategy has some immediate implications about computational complexity: since we evaluate JVPs as we go, we don't need to store anything for later, and so the memory cost is independent of the depth of the computation. In addition, the FLOP cost of the `jvp`-transformed function is about 3x the cost of just evaluating the function (one unit of work for evaluating the original function, for example `sin(x)`; one unit for linearizing, like `cos(x)`; and one unit for applying the linearized function to a vector, like `cos_x * v`). Put another way, for a fixed primal point x, we can evaluate $v \mapsto \partial f(x) \cdot v$ for about the same marginal cost as evaluating f.

That memory complexity sounds pretty compelling! So why don't we see forward-mode very often in machine learning?

To answer that, first think about how you could use a JVP to build a full Jacobian matrix. If we apply a JVP to a one-hot tangent vector, it reveals one column of the Jacobian matrix, corresponding to the nonzero entry we fed in. So we can build a full Jacobian one column at a time, and to get each column costs about the same as one function evaluation. That will be efficient for functions with “tall” Jacobians, but inefficient for “wide” Jacobians.

If you’re doing gradient-based optimization in machine learning, you probably want to minimize a loss function from parameters in $\mathbb{R}^n$ to a scalar loss value in $\mathbb{R}$. That means the Jacobian of this function is a very wide matrix: $\partial f(x) \in \mathbb{R}^{1\times n}$, which we often identify with the Gradient vector $\nabla f(x) \in \mathbb{R}^n$. Building that matrix one column at a time, with each call taking a similar number of FLOPs to evaluate the original function, sure seems inefficient! In particular, for training neural networks, where f is a training loss function and n can be in the millions or billions, this approach just won’t scale.

To do better for functions like this, we just need to use reverse-mode.

Vector-Jacobian products (VJPs, aka reverse-mode autodiff)

Where forward-mode gives us back a function for evaluating Jacobian-vector products, which we can then use to build Jacobian matrices one column at a time, reverse-mode is a way to get back a function for evaluating vector-Jacobian products (equivalently Jacobian-transpose-vector products), which we can use to build Jacobian matrices one row at a time.

VJPs in math

Let’s again consider a function $f : \mathbb{R}^n \to \mathbb{R}^m$. Starting from our notation for JVPs, the notation for VJPs is pretty simple:

$$(x, v) \mapsto v\partial f(x),$$

where v is an element of the cotangent space of f at x (isomorphic to another copy of $\mathbb{R}^m$). When being rigorous, we should think of v as a linear map $v : \mathbb{R}^m \to \mathbb{R}$, and when we write $v\partial f(x)$ we mean function composition $v \circ \partial f(x)$, where the types work out because $\partial f(x) : \mathbb{R}^n \to \mathbb{R}^m$. But in the common case we can identify v with a vector in $\mathbb{R}^m$ and use the two almost interchangeably, just like we might sometimes flip between “column vectors” and “row vectors” without much comment.

With that identification, we can alternatively think of the linear part of a VJP as the transpose (or adjoint conjugate) of the linear part of a JVP:

$$(x, v) \mapsto \partial f(x)^\mathsf{T} v.$$

For a given point x, we can write the signature as

$$\partial f(x)^\mathsf{T} : \mathbb{R}^m \to \mathbb{R}^n.$$

The corresponding map on cotangent spaces is often called the pullback of f at x. The key for our purposes is that it goes from something that looks like the output of f to something that looks like the input of f, just like we might expect from a transposed linear function.

VJPs in JAX code

Switching from math back to Python, the JAX function `vjp` can take a Python function for evaluating f and give us back a Python function for evaluating the VJP $(x, v) \mapsto (f(x), v^\mathsf{T}\partial f(x))$.

```
from jax import vjp

# Isolate the function from the weight matrix to the predictions
f = lambda W: predict(W, b, inputs)

y, vjp_fun = vjp(f, W)

key, subkey = random.split(key)
u = random.normal(subkey, y.shape)

# Pull back the covector `u` along `f` evaluated at `W`
v = vjp_fun(u)
```

In terms of Haskell-like type signatures, we could write

```
vjp :: (a -> b) -> a -> (b, CT b -> CT a)
```

where we use `CT a` to denote the type for the cotangent space for `a`. In words, `vjp` takes as arguments a function of type `a -> b` and a point of type `a`, and gives back a pair consisting of a value of type `b` and a linear map of type `CT b -> CT a`.

This is great because it lets us build Jacobian matrices one row at a time, and the FLOP cost for evaluating $(x, v) \mapsto (f(x), v^\mathsf{T}\partial f(x))$ is only about three times the cost of evaluating f. In particular, if we want the gradient of a function $f : \mathbb{R}^n \to \mathbb{R}$, we can do it in just one call. That's how `grad` is efficient for gradient-based optimization, even for objectives like neural network training loss functions on millions or billions of parameters.

There's a cost, though: though the FLOPs are friendly, memory scales with the depth of the computation. Also, the implementation is traditionally more complex than that of forward-mode, though JAX has some tricks up its sleeve (that's a story for a future notebook!).

For more on how reverse-mode works, see [this tutorial video from the Deep Learning Summer School in 2017](#).

Vector-valued gradients with VJPs

If you're interested in taking vector-valued gradients (like `tf.gradients`):

```
from jax import vjp

def vgrad(f, x):
  y, vjp_fn = vjp(f, x)
  return vjp_fn(jnp.ones(y.shape))[0]

print(vgrad(lambda x: 3*x**2, jnp.ones((2, 2))))
```

```
[[6. 6.]
 [6. 6.]]
```

Hessian-vector products using both forward- and reverse-mode

In a previous section, we implemented a Hessian-vector product function just using reverse-mode (assuming continuous second derivatives):

```
def hvp(f, x, v):
    return grad(lambda x: jnp.vdot(grad(f)(x), v))(x)
```

That's efficient, but we can do even better and save some memory by using forward-mode together with reverse-mode.

Mathematically, given a function $f : \mathbb{R}^n \to \mathbb{R}$ to differentiate, a point $x \in \mathbb{R}^n$ at which to linearize the function, and a vector $v \in \mathbb{R}^n$, the Hessian-vector product function we want is

$$(x, v) \mapsto \partial^2 f(x) v$$

Consider the helper function $g : \mathbb{R}^n \to \mathbb{R}^n$ defined to be the derivative (or gradient) of f, namely $g(x) = \partial f(x)$. All we need is its JVP, since that will give us

$$(x, v) \mapsto \partial g(x) v = \partial^2 f(x) v.$$

We can translate that almost directly into code:

```
from jax import jvp, grad

# forward-over-reverse
def hvp(f, primals, tangents):
  return jvp(grad(f), primals, tangents)[1]
```

Even better, since we didn't have to call `jnp.dot` directly, this `hvp` function works with arrays of any shape and with arbitrary container types (like vectors stored as nested lists/dicts/tuples), and doesn't even have a dependence on `jax.numpy`.

Here's an example of how to use it:

```
def f(X):
  return jnp.sum(jnp.tanh(X)**2)

key, subkey1, subkey2 = random.split(key, 3)
X = random.normal(subkey1, (30, 40))
V = random.normal(subkey2, (30, 40))

ans1 = hvp(f, (X,), (V,))
ans2 = jnp.tensordot(hessian(f)(X), V, 2)

print(jnp.allclose(ans1, ans2, 1e-4, 1e-4))
```

```
True
```

Another way you might consider writing this is using reverse-over-forward:

```
# reverse-over-forward
def hvp_revfwd(f, primals, tangents):
  g = lambda primals: jvp(f, primals, tangents)[1]
  return grad(g)(primals)
```

That's not quite as good, though, because forward-mode has less overhead than reverse-mode, and since the outer differentiation operator here has to differentiate a larger computation than the inner one, keeping forward-mode on the outside works best:

```
# reverse-over-reverse, only works for single arguments
def hvp_revrev(f, primals, tangents):
  x, = primals
  v, = tangents
  return grad(lambda x: jnp.vdot(grad(f)(x), v))(x)


print("Forward over reverse")
%timeit -n10 -r3 hvp(f, (X,), (V,))
print("Reverse over forward")
%timeit -n10 -r3 hvp_revfwd(f, (X,), (V,))
print("Reverse over reverse")
%timeit -n10 -r3 hvp_revrev(f, (X,), (V,))

print("Naive full Hessian materialization")
%timeit -n10 -r3 jnp.tensordot(hessian(f)(X), V, 2)
```

```
Forward over reverse
6.17 ms ± 126 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)
Reverse over forward
10.2 ms ± 4.22 ms per loop (mean ± std. dev. of 3 runs, 10 loops each)
Reverse over reverse
14.7 ms ± 7.18 ms per loop (mean ± std. dev. of 3 runs, 10 loops each)
Naive full Hessian materialization
71.3 ms ± 759 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)
```

Composing VJPs, JVPs, and `vmap`

Jacobian-Matrix and Matrix-Jacobian products

Now that we have `jvp` and `vjp` transformations that give us functions to push-forward or pull-back single vectors at a time, we can use JAX's `vmap` transformation to push and pull entire bases at once. In particular, we can use that to write fast matrix-Jacobian and Jacobian-matrix products.

```
# Isolate the function from the weight matrix to the predictions
f = lambda W: predict(W, b, inputs)

# Pull back the covectors `m_i` along `f`, evaluated at `W`, for all `i`.
# First, use a list comprehension to loop over rows in the matrix M.
def loop_mjp(f, x, M):
    y, vjp_fun = vjp(f, x)
    return jnp.vstack([vjp_fun(mi) for mi in M])

# Now, use vmap to build a computation that does a single fast matrix-matrix
# multiply, rather than an outer loop over vector-matrix multiplies.
def vmap_mjp(f, x, M):
    y, vjp_fun = vjp(f, x)
    outs, = vmap(vjp_fun)(M)
    return outs

key = random.PRNGKey(0)
num_covecs = 128
U = random.normal(key, (num_covecs,) + y.shape)

loop_vs = loop_mjp(f, W, M=U)
print('Non-vmapped Matrix-Jacobian product')
%timeit -n10 -r3 loop_mjp(f, W, M=U)

print('\nVmapped Matrix-Jacobian product')
vmap_vs = vmap_mjp(f, W, M=U)
%timeit -n10 -r3 vmap_mjp(f, W, M=U)

assert jnp.allclose(loop_vs, vmap_vs), 'Vmap and non-vmapped Matrix-Jacobian
Products should be identical'
```

```
Non-vmapped Matrix-Jacobian product
153 ms ± 472 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)

Vmapped Matrix-Jacobian product
5.93 ms ± 65.8 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)
```

```
def loop_jmp(f, W, M):
    # jvp immediately returns the primal and tangent values as a tuple,
    # so we'll compute and select the tangents in a list comprehension
    return jnp.vstack([jvp(f, (W,), (mi,))[1] for mi in M])

def vmap_jmp(f, W, M):
    _jvp = lambda s: jvp(f, (W,), (s,))[1]
    return vmap(_jvp)(M)

num_vecs = 128
S = random.normal(key, (num_vecs,) + W.shape)

loop_vs = loop_jmp(f, W, M=S)
print('Non-vmapped Jacobian-Matrix product')
%timeit -n10 -r3 loop_jmp(f, W, M=S)
vmap_vs = vmap_jmp(f, W, M=S)
print('\nVmapped Jacobian-Matrix product')
%timeit -n10 -r3 vmap_jmp(f, W, M=S)

assert jnp.allclose(loop_vs, vmap_vs), 'Vmap and non-vmapped Jacobian-Matrix
products should be identical'
```

```
Non-vmapped Jacobian-Matrix product
295 ms ± 1.07 ms per loop (mean ± std. dev. of 3 runs, 10 loops each)

Vmapped Jacobian-Matrix product
3.64 ms ± 66 µs per loop (mean ± std. dev. of 3 runs, 10 loops each)
```

The implementation of `jacfwd` and `jacrev`

Now that we've seen fast Jacobian-matrix and matrix-Jacobian products, it's not hard to guess how to write `jacfwd` and `jacrev`. We just use the same technique to push-forward or pull-back an entire standard basis (isomorphic to an identity matrix) at once.

```
from jax import jacrev as builtin_jacrev

def our_jacrev(f):
    def jacfun(x):
        y, vjp_fun = vjp(f, x)
        # Use vmap to do a matrix-Jacobian product.
        # Here, the matrix is the Euclidean basis, so we get all
        # entries in the Jacobian at once.
        J, = vmap(vjp_fun, in_axes=0)(jnp.eye(len(y)))
        return J
    return jacfun

assert jnp.allclose(builtin_jacrev(f)(W), our_jacrev(f)(W)), 'Incorrect
reverse-mode Jacobian results!'
```

```
from jax import jacfwd as builtin_jacfwd

def our_jacfwd(f):
    def jacfun(x):
        _jvp = lambda s: jvp(f, (x,), (s,))[1]
        Jt =vmap(_jvp, in_axes=1)(jnp.eye(len(x)))
        return jnp.transpose(Jt)
    return jacfun

assert jnp.allclose(builtin_jacfwd(f)(W), our_jacfwd(f)(W)), 'Incorrect
forward-mode Jacobian results!'
```

Interestingly, Autograd couldn't do this. Our implementation of reverse-mode `jacobian` in Autograd had to pull back one vector at a time with an outer-loop `map`. Pushing one vector at a time through the computation is much less efficient than batching it all together with `vmap`.

Another thing that Autograd couldn't do is `jit`. Interestingly, no matter how much Python dynamism you use in your function to be differentiated, we could always use `jit` on the linear part of the computation. For example:

```
def f(x):
    try:
        if x < 3:
            return 2 * x ** 3
        else:
            raise ValueError
    except ValueError:
        return jnp.pi * x

y, f_vjp = vjp(f, 4.)
print(jit(f_vjp)(1.))
```

```
(Array(3.1415927, dtype=float32, weak_type=True),)
```

Complex numbers and differentiation

JAX is great at complex numbers and differentiation. To support both holomorphic and non-holomorphic differentiation, it helps to think in terms of JVPs and VJPs.

Consider a complex-to-complex function $f : \mathbb{C} \to \mathbb{C}$ and identify it with a corresponding function $g : \mathbb{R}^2 \to \mathbb{R}^2$,

```
def f(z):
  x, y = jnp.real(z), jnp.imag(z)
  return u(x, y) + v(x, y) * 1j

def g(x, y):
  return (u(x, y), v(x, y))
```

That is, we've decomposed $f(z) = u(x, y) + v(x, y)i$ where $z = x + yi$, and identified $\mathbb{C}$ with $\mathbb{R}^2$ to get g.

Since g only involves real inputs and outputs, we already know how to write a Jacobian-vector product for it, say given a tangent vector $(c, d) \in \mathbb{R}^2$, namely

$$\begin{bmatrix} \partial_0 u(x, y) & \partial_1 u(x, y) \\ \partial_0 v(x, y) & \partial_1 v(x, y) \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix}.$$

To get a JVP for the original function f applied to a tangent vector $c + di \in \mathbb{C}$, we just use the same definition and identify the result as another complex number,

$$\partial f(x + yi)(c + di) = \begin{bmatrix} 1 & i \end{bmatrix} \begin{bmatrix} \partial_0 u(x, y) & \partial_1 u(x, y) \\ \partial_0 v(x, y) & \partial_1 v(x, y) \end{bmatrix} \begin{bmatrix} c \\ d \end{bmatrix}.$$

That's our definition of the JVP of a $\mathbb{C} \to \mathbb{C}$ function! Notice it doesn't matter whether or not f is holomorphic: the JVP is unambiguous.

Here's a check:

```
def check(seed):
  key = random.PRNGKey(seed)

  # random coeffs for u and v
  key, subkey = random.split(key)
  a, b, c, d = random.uniform(subkey, (4,))

  def fun(z):
    x, y = jnp.real(z), jnp.imag(z)
    return u(x, y) + v(x, y) * 1j

  def u(x, y):
    return a * x + b * y

  def v(x, y):
    return c * x + d * y

  # primal point
  key, subkey = random.split(key)
  x, y = random.uniform(subkey, (2,))
  z = x + y * 1j

  # tangent vector
  key, subkey = random.split(key)
  c, d = random.uniform(subkey, (2,))
  z_dot = c + d * 1j

  # check jvp
  _, ans = jvp(fun, (z,), (z_dot,))
  expected = (grad(u, 0)(x, y) * c +
              grad(u, 1)(x, y) * d +
              grad(v, 0)(x, y) * c * 1j+
              grad(v, 1)(x, y) * d * 1j)
  print(jnp.allclose(ans, expected))
```

```
check(0)
check(1)
check(2)
```

```
True
True
True
```

What about VJPs? We do something pretty similar: for a cotangent vector $c + di \in \mathbb{C}$ we define the VJP of f as

$$(c+di)^* \, \partial f(x+yi) = \begin{bmatrix} c & -d \end{bmatrix} \begin{bmatrix} \partial_0 u(x,y) & \partial_1 u(x,y) \\ \partial_0 v(x,y) & \partial_1 v(x,y) \end{bmatrix} \begin{bmatrix} 1 \\ -i \end{bmatrix}.$$

What's with the negatives? They're just to take care of complex conjugation, and the fact that we're working with covectors.

Here's a check of the VJP rules:

```
def check(seed):
  key = random.PRNGKey(seed)

  # random coeffs for u and v
  key, subkey = random.split(key)
  a, b, c, d = random.uniform(subkey, (4,))

  def fun(z):
    x, y = jnp.real(z), jnp.imag(z)
    return u(x, y) + v(x, y) * 1j

  def u(x, y):
    return a * x + b * y

  def v(x, y):
    return c * x + d * y

  # primal point
  key, subkey = random.split(key)
  x, y = random.uniform(subkey, (2,))
  z = x + y * 1j

  # cotangent vector
  key, subkey = random.split(key)
  c, d = random.uniform(subkey, (2,))
  z_bar = jnp.array(c + d * 1j)  # for dtype control

  # check vjp
  _, fun_vjp = vjp(fun, z)
  ans, = fun_vjp(z_bar)
  expected = (grad(u, 0)(x, y) * c +
              grad(v, 0)(x, y) * (-d) +
              grad(u, 1)(x, y) * c * (-1j) +
              grad(v, 1)(x, y) * (-d) * (-1j))
  assert jnp.allclose(ans, expected, atol=1e-5, rtol=1e-5)
```

```
check(0)
check(1)
check(2)
```

What about convenience wrappers like `grad`, `jacfwd`, and `jacrev`?

For $\mathbb{R} \to \mathbb{R}$ functions, recall we defined `grad(f)(x)` as being `vjp(f, x)[1](1.0)`, which works because applying a VJP to a `1.0` value reveals the gradient (i.e. Jacobian, or derivative). We can do the same thing for $\mathbb{C} \to \mathbb{R}$ functions: we can still use `1.0` as the cotangent vector, and we just get out a complex number result summarizing the full Jacobian:

```
def f(z):
  x, y = jnp.real(z), jnp.imag(z)
  return x**2 + y**2

z = 3. + 4j
grad(f)(z)
```

```
Array(6.-8.j, dtype=complex64)
```

For general $\mathbb{C} \to \mathbb{C}$ functions, the Jacobian has 4 real-valued degrees of freedom (as in the 2x2 Jacobian matrices above), so we can't hope to represent all of them within a complex number. But we can for holomorphic functions! A holomorphic function is precisely a $\mathbb{C} \to \mathbb{C}$ function with the special property that its derivative can be represented as a single complex number. (The Cauchy-Riemann equations ensure that the above 2x2 Jacobians have the special form of a scale-and-rotate matrix in the complex plane, i.e. the action of a single complex number under multiplication.) And we can reveal that one complex number using a single call to `vjp` with a covector of `1.0`.

Because this only works for holomorphic functions, to use this trick we need to promise JAX that our function is holomorphic; otherwise, JAX will raise an error when `grad` is used for a complex-output function:

```
def f(z):
  return jnp.sin(z)

z = 3. + 4j
grad(f, holomorphic=True)(z)
```

```
Array(-27.034946-3.8511534j, dtype=complex64, weak_type=True)
```

All the `holomorphic=True` promise does is disable the error when the output is complex-valued. We can still write `holomorphic=True` when the function isn't holomorphic, but the answer we get out won't represent the full Jacobian. Instead, it'll be the Jacobian of the function where we just discard the imaginary part of the output:

```
def f(z):
  return jnp.conjugate(z)

z = 3. + 4j
grad(f, holomorphic=True)(z)  # f is not actually holomorphic!
```

```
Array(1.-0.j, dtype=complex64, weak_type=True)
```

There are some useful upshots for how `grad` works here:

1. We can use `grad` on holomorphic $\mathbb{C} \to \mathbb{C}$ functions.
2. We can use `grad` to optimize $f : \mathbb{C} \to \mathbb{R}$ functions, like real-valued loss functions of complex parameters `x`, by taking steps in the direction of the conjugate of `grad(f)(x)`.

3. If we have an $\mathbb{R} \to \mathbb{R}$ function that just happens to use some complex-valued operations internally (some of which must be non-holomorphic, e.g. FFTs used in convolutions) then `grad` still works and we get the same result that an implementation using only real values would have given.

In any case, JVPs and VJPs are always unambiguous. And if we wanted to compute the full Jacobian matrix of a non-holomorphic $\mathbb{C} \to \mathbb{C}$ function, we can do it with JVPs or VJPs!

You should expect complex numbers to work everywhere in JAX. Here's differentiating through a Cholesky decomposition of a complex matrix:

```
A = jnp.array([[5.,    2.+3j,    5j],
               [2.-3j,   7.,  1.+7j],
               [-5j,  1.-7j,    12.]])

def f(X):
    L = jnp.linalg.cholesky(X)
    return jnp.sum((L - jnp.sin(L))**2)

grad(f, holomorphic=True)(A)
```

```
Array([[-0.7534182  +0.j       , -3.0509028 -10.940544j,
         5.9896846  +3.542303j],
       [-3.0509028 +10.940544j, -8.904491   +0.j       ,
        -5.1351523  -6.559373j],
       [ 5.9896846  -3.542303j, -5.1351523  +6.559373j,
         0.01320427 +0.j       ]], dtype=complex64)
```

More advanced autodiff

In this notebook, we worked through some easy, and then progressively more complicated, applications of automatic differentiation in JAX. We hope you now feel that taking derivatives in JAX is easy and powerful.

There's a whole world of other autodiff tricks and functionality out there. Topics we didn't cover, but hope to in an "Advanced Autodiff Cookbook" include:

- Gauss-Newton Vector Products, linearizing once
- Custom VJPs and JVPs
- Efficient derivatives at fixed-points
- Estimating the trace of a Hessian using random Hessian-vector products.
- Forward-mode autodiff using only reverse-mode autodiff.
- Taking derivatives with respect to custom data types.

- Checkpointing (binomial checkpointing for efficient reverse-mode, not model snapshotting).
- Optimizing VJPs with Jacobian pre-accumulation.
