JAX Quickstart

Contents

- Multiplying Matrices
- Using `jit()` to speed up functions
- Taking derivatives with `grad()`
- Auto-vectorization with `vmap()`

Open in Colab

JAX is NumPy on the CPU, GPU, and TPU, with great automatic differentiation for high-performance machine learning research.

With its updated version of Autograd, JAX can automatically differentiate native Python and NumPy code. It can differentiate through a large subset of Python's features, including loops, ifs, recursion, and closures, and it can even take derivatives of derivatives of derivatives. It supports reverse-mode as well as forward-mode differentiation, and the two can be composed arbitrarily to any order.

What's new is that JAX uses XLA to compile and run your NumPy code on accelerators, like GPUs and TPUs. Compilation happens under the hood by default, with library calls getting just-in-time compiled and executed. But JAX even lets you just-in-time compile your own Python functions into XLA-optimized kernels using a one-function API. Compilation and automatic differentiation can be composed arbitrarily, so you can express sophisticated algorithms and get maximal performance without having to leave Python.

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

Multiplying Matrices

We'll be generating random data in the following examples. One big difference between NumPy and JAX is how you generate random numbers. For more details, see Common Gotchas in JAX.

```python
key = random.PRNGKey(0)
x = random.normal(key, (10,))
print(x)
```

```
[-0.3721109   0.26423115 -0.18252768 -0.7368197  -0.44030377 -0.1521442
 -0.67135346 -0.5908641   0.73168886  0.5673026 ]
```

Let’s dive right in and multiply two big matrices.

```
size = 3000
x = random.normal(key, (size, size), dtype=jnp.float32)
%timeit jnp.dot(x, x.T).block_until_ready()  # runs on the GPU
```

```
13.5 ms ± 1.89 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

We added that `block_until_ready` because JAX uses asynchronous execution by default (see Asynchronous dispatch).

JAX NumPy functions work on regular NumPy arrays.

```
import numpy as np
x = np.random.normal(size=(size, size)).astype(np.float32)
%timeit jnp.dot(x, x.T).block_until_ready()
```

```
80 ms ± 30.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

That’s slower because it has to transfer data to the GPU every time. You can ensure that an NDArray is backed by device memory using **`device_put()`**.

```
from jax import device_put

x = np.random.normal(size=(size, size)).astype(np.float32)
x = device_put(x)
%timeit jnp.dot(x, x.T).block_until_ready()
```

```
15.8 ms ± 113 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

The output of **`device_put()`** still acts like an NDArray, but it only copies values back to the CPU when they’re needed for printing, plotting, saving to disk, branching, etc. The behavior of **`device_put()`** is equivalent to the function `jit(lambda x: x)`, but it’s faster.

If you have a GPU (or TPU!) these calls run on the accelerator and have the potential to be much faster than on CPU. See Is JAX faster than NumPy? for more comparison of performance characteristics of NumPy and JAX

JAX is much more than just a GPU-backed NumPy. It also comes with a few program transformations that are useful when writing numerical code. For now, there are three main ones:

- **`jit()`**, for speeding up your code
- **`grad()`**, for taking derivatives
- **`vmap()`**, for automatic vectorization or batching.

Let’s go over these, one-by-one. We’ll also end up composing these in interesting ways.

Using `jit()` to speed up functions

JAX runs transparently on the GPU or TPU (falling back to CPU if you don’t have one). However, in the above example, JAX is dispatching kernels to the GPU one operation at a time. If we have a sequence of operations, we can use the `@jit` decorator to compile multiple operations together using XLA. Let’s try that.

```
def selu(x, alpha=1.67, lmbda=1.05):
  return lmbda * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)

x = random.normal(key, (1000000,))
%timeit selu(x).block_until_ready()
```

```
1.07 ms ± 261 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

We can speed it up with `@jit`, which will jit-compile the first time `selu` is called and will be cached thereafter.

```
selu_jit = jit(selu)
%timeit selu_jit(x).block_until_ready()
```

```
127 µs ± 1.43 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Taking derivatives with `grad()`

In addition to evaluating numerical functions, we also want to transform them. One transformation is automatic differentiation. In JAX, just like in Autograd, you can compute gradients with the `grad()` function.

```
def sum_logistic(x):
  return jnp.sum(1.0 / (1.0 + jnp.exp(-x)))

x_small = jnp.arange(3.)
derivative_fn = grad(sum_logistic)
print(derivative_fn(x_small))
```

```
[0.25       0.19661194 0.10499357]
```

Let’s verify with finite differences that our result is correct.

```
def first_finite_differences(f, x):
  eps = 1e-3
  return jnp.array([(f(x + eps * v) - f(x - eps * v)) / (2 * eps)
                   for v in jnp.eye(len(x))])


print(first_finite_differences(sum_logistic, x_small))
```

```
[0.24998187 0.1965761  0.10502338]
```

Taking derivatives is as easy as calling `grad()`. `grad()` and `jit()` compose and can be mixed arbitrarily. In the above example we jitted `sum_logistic` and then took its derivative. We can go further:

```
print(grad(jit(grad(jit(grad(sum_logistic)))))(1.0))
```

```
-0.0353256
```

For more advanced autodiff, you can use `jax.vjp()` for reverse-mode vector-Jacobian products and `jax.jvp()` for forward-mode Jacobian-vector products. The two can be composed arbitrarily with one another, and with other JAX transformations. Here's one way to compose them to make a function that efficiently computes full Hessian matrices:

```
from jax import jacfwd, jacrev
def hessian(fun):
  return jit(jacfwd(jacrev(fun)))
```

Auto-vectorization with `vmap()`

JAX has one more transformation in its API that you might find useful: `vmap()`, the vectorizing map. It has the familiar semantics of mapping a function along array axes, but instead of keeping the loop on the outside, it pushes the loop down into a function's primitive operations for better performance. When composed with `jit()`, it can be just as fast as adding the batch dimensions by hand.

We're going to work with a simple example, and promote matrix-vector products into matrix-matrix products using `vmap()`. Although this is easy to do by hand in this specific case, the same technique can apply to more complicated functions.

```
mat = random.normal(key, (150, 100))
batched_x = random.normal(key, (10, 100))

def apply_matrix(v):
  return jnp.dot(mat, v)
```

Given a function such as `apply_matrix`, we can loop over a batch dimension in Python, but usually the performance of doing so is poor.

```
def naively_batched_apply_matrix(v_batched):
  return jnp.stack([apply_matrix(v) for v in v_batched])

print('Naively batched')
%timeit naively_batched_apply_matrix(batched_x).block_until_ready()
```

```
Naively batched
3.12 ms ± 176 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

We know how to batch this operation manually. In this case, `jnp.dot` handles extra batch dimensions transparently.

```
@jit
def batched_apply_matrix(v_batched):
  return jnp.dot(v_batched, mat.T)

print('Manually batched')
%timeit batched_apply_matrix(batched_x).block_until_ready()
```

```
Manually batched
45.6 µs ± 5.03 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

However, suppose we had a more complicated function without batching support. We can use `vmap()` to add batching support automatically.

```
@jit
def vmap_batched_apply_matrix(v_batched):
  return vmap(apply_matrix)(v_batched)

print('Auto-vectorized with vmap')
%timeit vmap_batched_apply_matrix(batched_x).block_until_ready()
```

```
Auto-vectorized with vmap
48.3 µs ± 1.06 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Of course, `vmap()` can be arbitrarily composed with `jit()`, `grad()`, and any other JAX transformation.

This is just a taste of what JAX can do. We're really excited to see what you do with it!