**JAX: Fast as PyTorch, Simple as NumPy**

[[Hylke C. Donker](https://medium.com/@hylke.donker?source=post_page-----a0c14893a738--------------------------------)](https://medium.com/@hylke.donker?source=post_page-----a0c14893a738--------------------------------)

[Hylke C. Donker](https://medium.com/@hylke.donker?source=post_page-----a0c14893a738--------------------------------)

·

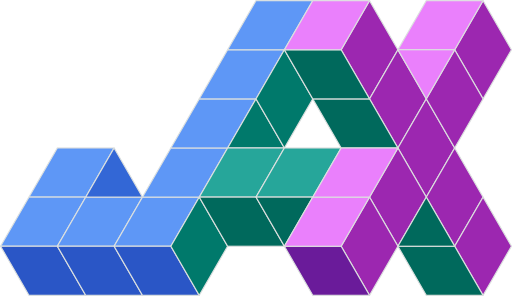
[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fea1bfe4db7a8&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40hylke.donker%2Fjax-fast-as-pytorch-simple-as-numpy-a0c14893a738&user=Hylke+C.+Donker&userId=ea1bfe4db7a8&source=post_page-ea1bfe4db7a8----a0c14893a738---------------------post_header-----------)

4 min read

·

Jul 6

120



JAX logo. Google LLC, Public domain, via Wikimedia Commons.

[JAX](https://jax.readthedocs.io/) is a new competitor of TensorFlow and PyTorch. JAX emphasises simplicity without sacrificing speed and scalability. Since JAX requires less boiler plate code, programs are shorter, closer to the math, and thus easier to understand.

**TL;DR**:

* 🐍 Access NumPy functions using import jax.numpy and SciPy functions with import jax.scipy.
* 🔥 Speed up with just-in-time compilation by decorating with @jax.jit.
* ∇ Take derivatives using jax.grad.
* ➡️ Vectorise with jax.vmap and parallelise across devices with jax.pmap.

This post is a shortened version of a talk I gave last spring at [PyGrunn 11](https://pygrunn.org/" \t "_blank), one of the largest Python conferences in Europe.

**Functional Programming**

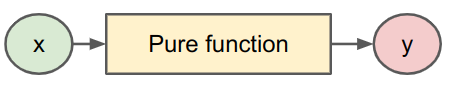


Fig. 1: A pure function looks like a mathematical function. Image by Author.

JAX follows a functional programming philosophy. This means that your functions must be self-contained, or *pure*: [side effects](https://en.wikipedia.org/wiki/Side_effect_(computer_science)) are not allowed. Essentially, a pure function looks like a mathematical function (Fig. 1). Input comes in, something comes out, but there is no communication with the outside world.

**❌ Example #1**

The following snippet is an example that **is not** functionally pure.

import jax.numpy as jnp  
  
bias = jnp.array(0)  
def impure\_example(x):  
 total = x + bias  
 return total

Notice the bias outside impure\_example. During compilation (see below), bias may be cached and therefore no longer reflect changes to bias.

**✅ Example #2**

Here is an example that **is** pure.

def pure\_example(x, weights, bias):  
 activation = weights @ x + bias  
 return activation

Here, pure\_example is self-contained: all parameters are passed as arguments.

**🎰 Deterministic Samplers**

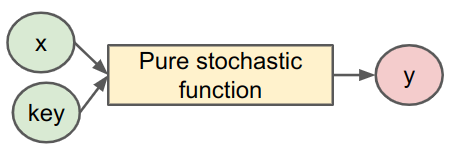


Fig. 2: Random functions require a pseudo random number generator key. Image by Author.

In computers, true randomness does not exist. Instead, libraries such as NumPy and TensorFlow track a pseudo-random number state to generate “random” samples. A direct consequence of functional programming is that random functions work different. Since a global state is no longer allowed, you need to explicitly pass in a pseudo random number generator (PRNG) key every time you sample a random number (Fig. 2).

import jax  
  
key = jax.random.PRNGKey(42)  
u = jax.random.uniform(key)

Moreover, you are responsible for advancing the “random state” for any subsequent calls.

key = jax.random.PRNGKey(43)  
# Split off and consume subkey.  
key, subkey = jax.random.split(key)  
u = jax.random.uniform(subkey)  
# Split off and consume second subkey.  
key, subkey = jax.random.split(key)  
u = jax.random.uniform(subkey)  
  
..

**🔥**jit

You can speed up your code by just-in-time compiling your JAX instructions. For example, to compile your scaled exponential linear units (SELU) function, use the NumPy functions from jax.numpy and add the jax.jit decorator to the function as follows:

from jax import jit  
  
@jit  
def selu(x, α=1.67, λ=1.05):  
 return λ \* jnp.where(x > 0, x, α \* jnp.exp(x) - α)

Under the hood, JAX traces your instructions and converts it into [jaxpr](https://jax.readthedocs.io/en/latest/jaxpr.html" \t "_blank). This allows the accelerated linear algebra (XLA) compiler to make very efficient optimised code for your accelerator.

**∇ grad**

One of the most powerful features of JAX is that you can easily take gradients. With jax.grad, you define a new function that is the symbolic derivative.

from jax import grad  
  
def f(x):  
 return x + 0.5 \* x\*\*2  
  
df\_dx = grad(f)  
d2f\_dx2 = grad(grad(f))

As you can see in the example, you’re not limited to first order derivatives. You can take the *n*-th order derivative by simply chaining the grad function *n* times in sequence.

**➡️ vmap and pmap**

Matrix multiplications require serious mental gymnastics to get all the batch dimensions right. JAX’s vectorise-map function vmap alleviates this burden by vectorising your function. Basically, every code chunk that applies a function f element-wise is a candidate to be replaced by vmap. Let’s look at an example.

To compute the linear function:

def linear(x):  
 return weights @ x

across a batch of examples **[*x***₁, ***x***₂,..], we could naively (without vmap) implement it as follows:

def naively\_batched\_linear(X\_batched):  
 return jnp.stack([linear(x) for x in X\_batched])

Instead, by vectorising linearwith vmap we can compute the entire batch in one go:

def vmap\_batched\_linear(X\_batched):  
 return vmap(linear)(X\_batched)

**Bonus**: To distribute your workload across accelerators, you can play the same game: replace vmapwith pmap and your computation will scale out across multiple devices.

To learn more, I recommend Laurence Moroney’s introductory video. For further reading, take a look at the [JAX docs](https://jax.readthedocs.io/).