Pseudorandom numbers

Contents

> *If all scientific papers whose results are in doubt because of bad `rand()`s were to disappear from library shelves, there would be a gap on each shelf about as big as your fist.* - Numerical Recipes

In this section we focus on `jax.random` and pseudo random number generation (PRNG); that is, the process of algorithmically generating sequences of numbers whose properties approximate the properties of sequences of random numbers sampled from an appropriate distribution.

PRNG-generated sequences are not truly random because they are actually determined by their initial value, which is typically referred to as the `seed`, and each step of random sampling is a deterministic function of some `state` that is carried over from a sample to the next.

Pseudo random number generation is an essential component of any machine learning or scientific computing framework. Generally, JAX strives to be compatible with NumPy, but pseudo random number generation is a notable exception.

To better understand the difference between the approaches taken by JAX and NumPy when it comes to random number generation we will discuss both approaches in this section.

Random numbers in NumPy

Pseudo random number generation is natively supported in NumPy by the module. In NumPy, pseudo random number generation is based on a globa[illegible] which can be set to a deterministic initial condition using `numpy.random.seed()`.

```
import numpy as np
np.random.seed(0)
```

Repeated calls to NumPy's stateful pseudorandom number generators (PRNGs) mutate the global state and give a stream of pseudorandom numbers:

```
print(np.random.random())
print(np.random.random())
print(np.random.random())
```

```
0.5488135039273248
0.7151893663724195
0.6027633760716439
```

Underneath the hood, NumPy uses the Mersenne Twister PRNG to power its pseudorandom functions. The PRNG has a period of $2^{19937} - 1$ and at any point can be described by 624 32-bit unsigned ints and a position indicating how much of this "entropy" has been used up.

You can inspect the content of the state using the following command.

```
def print_truncated_random_state():
  """To avoid spamming the outputs, print only part of the state."""
  full_random_state = np.random.get_state()
  print(str(full_random_state)[:460], '...')

print_truncated_random_state()
```

```
('MT19937', array([2443250962, 1093594115, 1878467924, 2709361018, 1101979660,
       3904844661,  676747479, 2085143622, 1056793272, 3812477442,
       2168787041,  275552121, 2696932952, 3432054210, 1657102335,
       3518946594,  962584079, 1051271004, 3806145045, 1414436097,
       2032348584, 1661738718, 1116708477, 2562755208, 3176189976,
        696824676, 2399811678, 3992505346,  569184356, 2626558620,
        136797809, 4273176064,  296167901, 343 ...
```

The `state` is updated by each call to a random function:

```
np.random.seed(0)
print_truncated_random_state()
```

```
('MT19937', array([         0,          1, 1812433255, 1900727105, 1208447044,
       2481403966, 4042607538,  337614300, 3232553940, 1018809052,
       3202401494, 1775180719, 3192392114,  594215549,  184016991,
        829906058,  610491522, 3879932251, 3139825610,  297902587,
       4075895579, 2943625357, 3530655617, 1423771745, 2135928312,
       2891506774, 1066338622,  135451537,  933040465, 2759011858,
       2273819758, 3545703099, 2516396728, 127 ...
```

```
_ = np.random.uniform()
print_truncated_random_state()
```

```
('MT19937', array([2443250962, 1093594115, 1878467924, 2709361018, 1101979660,
       3904844661,  676747479, 2085143622, 1056793272, 3812477442,
       2168787041,  275552121, 2696932952, 3432054210, 1657102335,
       3518946594,  962584079, 1051271004, 3806145045, 1414436097,
       2032348584, 1661738718, 1116708477, 2562755208, 3176189976,
        696824676, 2399811678, 3992505346,  569184356, 2626558620,
        136797809, 4273176064,  296167901, 343 ...
```

NumPy allows you to sample both individual numbers, or entire vectors of numbers in a single function call. For instance, you may sample a vector of 3 scalars from a uniform distribution by doing:

```
np.random.seed(0)
print(np.random.uniform(size=3))
```

```
[0.5488135  0.71518937 0.60276338]
```

NumPy provides a *sequential equivalent guarantee*, meaning that sampling N numbers in a row individually or sampling a vector of N numbers results in the same pseudo-random sequences:

```
np.random.seed(0)
print("individually:", np.stack([np.random.uniform() for _ in range(3)]))

np.random.seed(0)
print("all at once: ", np.random.uniform(size=3))
```

```
individually: [0.5488135  0.71518937 0.60276338]
all at once:  [0.5488135  0.71518937 0.60276338]
```

Random numbers in JAX

JAX's random number generation differs from NumPy's in important ways, because NumPy's PRNG design makes it hard to simultaneously guarantee a number of desirable properties. Specifically, in JAX we want PRNG generation to be:

1. reproducible,
2. parallelizable,
3. vectorisable.

We will discuss why in the following. First, we will focus on the implications of a PRNG design based on a global state. Consider the code:

```
import numpy as np

np.random.seed(0)

def bar(): return np.random.uniform()
def baz(): return np.random.uniform()

def foo(): return bar() + 2 * baz()

print(foo())
```

```
1.9791922366721637
```

The function `foo` sums two scalars sampled from a uniform distribution.

The output of this code can only satisfy requirement #1 if we assume a predictable order of execution for `bar()` and `baz()`. This is not a problem in NumPy, which always evaluates code in the order defined by the Python interpreter. In JAX, however, this is more problematic: for efficient execution, we want the JIT compiler to be free to reorder, elide, and fuse various operations in the function we define. Further, when executing in multi-device environments, execution efficiency would be hampered by the need for each process to synchronize a global state.

Explicit random state

To avoid these issues, JAX avoids implicit global random state, and instead tracks state explicitly via a random `key`:

```
from jax import random

key = random.key(42)
print(key)
```

```
Array((), dtype=key<fry>) overlaying:
[ 0 42]
```

Note

This section uses the new-style typed PRNG keys produced by `jax.random.key()`, rather than the old-style raw PRNG keys produced by `jax.random.PRNGKey()`. For details, see JEP 9263: Typed keys & pluggable RNGs.

A key is an array with a special dtype corresponding to the particular PRNG implementation being used; in the default implementation each key is backed by a pair of `uint32` values.

The key is effectively a stand-in for NumPy's hidden state object, but we pass it explicitly to `jax.random()` functions. Importantly, random functions consume the key, but do not modify it: feeding the same key object to a random function will always result in the same sample being generated.

```
print(random.normal(key))
print(random.normal(key))
```

```
-0.028304616
-0.028304616
```

Reusing the same key, even with different `random` APIs, can result in correlated outputs, which is generally undesirable.

The rule of thumb is: never reuse keys (unless you want identical outputs). Reusing the same state will cause sadness and monotony, depriving the end user of lifegiving chaos.

JAX uses a modern Threefry counter-based PRNG that's splittable. That is, its design allows us to fork the PRNG state into new PRNGs for use with parallel stochastic generation. In order to generate different and independent samples, you must `split()` the key explicitly before passing it to a random function:

```
for i in range(3):
  new_key, subkey = random.split(key)
  del key  # The old key is consumed by split() -- we must never use it again.

  val = random.normal(subkey)
  del subkey  # The subkey is consumed by normal().

  print(f"draw {i}: {val}")
  key = new_key  # new_key is safe to use in the next iteration.
```

```
draw 0: 0.6057640314102173
draw 1: -0.21089035272598267
draw 2: -0.3948981463909149
```

(Calling `del` here is not required, but we do so to emphasize that the key should not be reused once consumed.)

`jax.random.split()` is a deterministic function that converts one `key` into several independent (in the pseudorandomness sense) keys. We keep one of the outputs as the `new_key`, and can safely use the unique extra key (called `subkey`) as input into a random function, and then discard it forever. If you wanted to get another sample from the normal distribution, you would split `key` again, and so on: the crucial point is that you never use the same key twice.

It doesn't matter which part of the output of `split(key)` we call `key`, and which we call `subkey`. They are all independent keys with equal status. The key/subkey naming convention is a typical usage pattern that helps track how keys are consumed: subkeys are destined for immediate consumption by random functions, while the key is retained to generate more randomness later.

Usually, the above example would be written concisely as

```
key, subkey = random.split(key)
```

which discards the old key automatically. It's worth noting that `split()` can create as many keys as you need, not just 2:

```
key, *forty_two_subkeys = random.split(key, num=43)
```

Lack of sequential equivalence

Another difference between NumPy's and JAX's random modules relates to the sequential equivalence guarantee mentioned above.

As in NumPy, JAX's random module also allows sampling of vectors of numbers. However, JAX does not provide a sequential equivalence guarantee, because doing so would interfere with the vectorization on SIMD hardware (requirement #3 above).

In the example below, sampling 3 values out of a normal distribution individually using three subkeys gives a different result to using giving a single key and specifying `shape=(3,)`:

```
key = random.key(42)
subkeys = random.split(key, 3)
sequence = np.stack([random.normal(subkey) for subkey in subkeys])
print("individually:", sequence)

key = random.key(42)
print("all at once: ", random.normal(key, shape=(3,)))
```

```
individually: [0.07592554 0.60576403 0.4323065 ]
all at once:  [-0.02830462  0.46713185  0.29570296]
```

The lack of sequential equivalence gives us freedom to write code more efficiently; for example, instead of generating `sequence` above via a sequential loop, we can use `jax.vmap()` to compute the same result in a vectorized manner:

```
import jax
print("vectorized:", jax.vmap(random.normal)(subkeys))
```

```
vectorized: [0.07592554 0.60576403 0.4323065 ]
```

Next Steps

For more information on JAX random numbers, refer to the documentation of the `jax.random` module. If you're interested in the details of the design of JAX's random number generator, see JAX PRNG Design.

Previous
Pytrees

Next
Introduction to parallel programming

latest