NumPy & JAX NumPy:
Numerical Computing with Python

Key Differences and JAX Superpowers

Familiar API, Powerful New Engine

- You know NumPy: The foundation of Python scientific computing (`ndarray`, rich function library).
- You've seen JAX: High-performance numerical computing, especially for ML research.
- `jax.numpy` is designed to feel like NumPy, but better.

NumPy/PyTorch = Eager, JAX = JIT Compiled

- NumPy/PyTorch (Default): Operations run immediately as Python encounters them. Easy debugging, intuitive flow.
- JAX: Uses `jax.jit` for Just-In-Time compilation via XLA (Accelerated Linear Algebra).
    - **Tracing**: JAX traces the function once for given input shapes/types.
    - **Optimization**: XLA optimizes and compiles the traced operations into efficient kernels (often fused).
    - **Execution**: Subsequent calls with compatible inputs use the fast, compiled code.

NumPy = Mutable, JAX = Immutable

NumPy: Arrays can be changed in-place. Standard Python behavior.

```
# NumPy Example
import numpy as np
a_np = np.arange(4.)
print("Original:", a_np)
a_np[0] = 100.0 # Modify in-place
print("Modified:", a_np)
# Output:
# Original: [0. 1. 2. 3.]
# Modified: [100. 1.   2.   3.]
```

JAX: Arrays cannot be changed in-place. Returns a new array. Functional style.

```
# JAX Example
import jax.numpy as jnp
a_jnp = jnp.arange(4.)
# a_jnp[0] = 100.0 # <-- This causes a TypeError!

# Use the .at[].set() syntax (or add, min, max...)
b_jnp = a_jnp.at[0].set(100.0) # New array
print(f'{a_jnp is b_jnp = }')
# Output:
# a_jnp is b_jnp = False
```

Google

Subtle Differences in Memory Handling

NumPy: Operations like **transpose()**, `reshape()`, slicing often return views (sharing underlying data).

- Memory efficient, but changes through a view affect the original.

JAX: Equivalent operations typically return copies.

- Consistent with immutability.
- Seems less memory efficient?
  But: `jax.jit()` often optimizes away intermediate copies.
  - "buffer donation"

JAX approach eliminates lurking side-effects

Explicit PRNG Keys are Key!

NumPy: Uses a global random state. Easy to use, but tricky for reproducibility in complex/parallel code.

JAX: Requires explicit PRNG keys. Ensures reproducibility. Functional style.

```
# NumPy RNG
np.random.seed(0)
print(np.random.normal()) # Call 1
print(np.random.normal()) # Call 2
# Output:
# 1.764052345967664
# 0.4001572083672233
```

```
# JAX RNG
from jax import random
key = random.PRNGKey(0) # Initial key
print(random.normal(key))
print(random.normal(key))
# Output:
# 1.6226422
# 1.6226422
```

Write for One, Run for Many

- **NumPy**: Relies on broadcasting and manually writing vectorized operations. Can be tricky for complex functions.
- **JAX**: `jax.vmap()` automatically transforms a function written for single data points to operate over batches/axes.

Write for One, Run for Many

```python
import jax
import jax.numpy as jnp
from jax import vmap

def predict(W, b, x):
    return jnp.dot(W, x) + b

W = jnp.ones((3, 4))
b = jnp.ones(3)
batch_x = jnp.ones((10, 4)) # Batch of 10 data points

# Apply vmap() to predict the whole batch without a Python loop
batch_predict = vmap(predict, in_axes=(None, None, 0))

# Result is shape (10, 3) - one prediction per input in the batch
batch_result = batch_predict(W, b, batch_x)
```

Google

Pytrees

- Tree-like nested Python containers (**`lists`**, **`tuples`**, **`dicts`**) holding JAX arrays (or other values) as "leaves"
- Ubiquitous in JAX for parameters, metrics, optimizer states and etc.
- Essential for initializing complex structures, applying updates, aggregating results

```
params = {
    "layer1":{
        "w":[1, 1],
        "b":2
    },
    "layer2":{
        "w":3,
        "b":4
    }
}
```

Working with pytrees

- Most JAX functions (`jit`, `grad`, `vmap`, optimizers) operate transparently over pytrees
- `jax.tree.map()` works similarly to Python `map()`, but operates over pytrees

```
params = {
    "layer1":{
        "w":[1, 1],
        "b":2
    },
    "layer2":{
        "w":3,
        "b":4
    }
}

jax.tree.map(lambda x: x*2, params)
# {'layer1': {'b': 4, 'w': [2, 2]}, 'layer2': {'b': 8, 'w': 6}}
```

JAX Strength: Explicit Parallelism (`shard_map`)

aka “shmap”

- **Manual Control**: Provides explicit, manual control over multi-device parallelism, complementing `jit's` automatic partitioning.
- **SPMD Approach**: You write the code from a device-local perspective (Single-Program Multiple-Data).

JAX Strength: Explicit Parallelism (`shard_map`)

aka “shmap”

- **Explicit Communication**: Requires users to explicitly write collective communication operations (e.g., all_gather, psum) needed between devices/shards.
- **Expressive & Debuggable**: Offers more expressiveness and can work eagerly, aiding debugging.

JAX Strength: `shard_map()` (aka "shmap")

```python
import jax
import jax.numpy as jnp

# jax.P is also a direct alias of PartitionSpec
from jax.sharding import Mesh, PartitionSpec as P

mesh = jax.make_mesh((8,), ('x',))

@jax.jit
def f_elementwise(x):
  return 2 * jnp.sin(x) + 1
```

Google

JAX Strength: `shard_map()` (aka "shmap")

```
f_elementwise_sharded = jax.shard_map(f_elementwise,# From previous
    mesh=mesh, in_specs=P('x'), out_specs=P('x'))

arr = jnp.arange(32)
sharded = f_elementwise_sharded(arr)
sharded
```

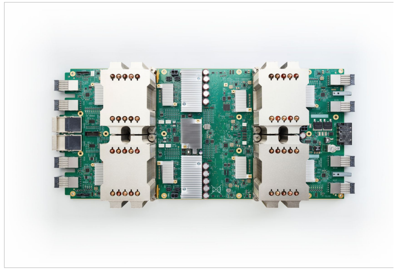
```
Array([ 1.        ,  2.682942  ,  2.818595  ,  1.28224   , -0.513605  ,
       -0.9178486 ,  0.44116902,  2.3139732 ,  2.9787164 ,  1.824237  ,
       -0.08804226, -0.99998045, -0.07314587,  1.840334  ,  2.9812148 ,
        2.3005757 ,  0.42419338, -0.92279494, -0.50197446,  1.2997544 ,
        2.8258905 ,  2.6733112 ,  0.98229736, -0.69244087, -0.81115675,
        0.7352965 ,  2.525117  ,  2.912752  ,  1.5418116 , -0.32726777,
       -0.97606325,  0.19192469], dtype=float32)
```

JAX Strength: `shard_map()` (aka “shmap”)

```
jax.debug.visualize_array_sharding(sharded)
```

CPU, GPU, TPU - Write Once, Run Anywhere (Fast!)

- **NumPy**: Primarily CPU-bound. GPU/TPU require other libraries (CuPy, Numba, etc.) often with code changes.
- **JAX**: Built on XLA. Runs seamlessly on GPUs and TPUs without changing your `jax.numpy` code.
  - JAX detects available hardware.
  - `jax.jit()` compiles code optimized for the specific accelerator.
- **Benefit**: Massive speedups for large-scale computation (deep learning, physics simulations) with minimal effort. Just run your script on a machine with the hardware!



Hardware Portability: JAX v PyTorch v TF Failure Rates

Comparison of TPU and GPU Failure and Success Rates						
	GPUs			TPUs		
	Success	Failure		Success	Failure	
	Pass	Partial	Complete	Pass	Partial	Complete
TensorFlow	78%	8%	14%	71%	15%	14%
PyTorch	92%	3%	5%	57%	27%	17%
JAX	98%	0%	2%	97%	0%	3%

Source:
[The Grand Illusion: The Myth of Software Portability and Implications for ML Progress (Cohere/MIT Sept 2023)]

Key Differences at a Glance

Feature	NumPy	JAX NumPy	Why It Matters
Mutability	Mutable (in-place)	**Immutable**	Functional style, JAX transforms
Execution	Eager	**JIT Compiled** (via XLA)	Performance (esp. accelerators)
In-place Ops	`a[i] = x`	`a.at[i].set(x)` (new array)	Immutability requirement
Views/Copies	Often Views	Typically Copies	JIT optimizes copies away
RNG	Global State (`np.random`)	**Explicit Keys** (`jax.random`)	Reproducibility, parallelism
Autodiff (`grad`)	External Libs	**Built-in (`jax.grad`)**	Foundational for ML
Auto-vectorize (`vmap`)	Manual / Broadcasting	**Built-in (`jax.vmap`)**	Easier batching
Hardware	CPU (mostly)	**CPU, GPU, TPU**	Performance scaling

Learning Resources

Code Exercises, Quick References, and Slides

- https://goo.gle/learning-jax

More videos for learning JAX

- https://goo.gle/learn-jax-videos

Learn JAX

Community and Docs

Community:

- https://goo.gle/jax-community

Docs

- JAX AI Stack: https://jaxstack.ai
- JAX: https://jax.dev
- Flax NNX: https://flax.readthedocs.io