jax-summary-for-cms-away-day-25

June 13, 2025

1 JAX and its potential use to CMS

1.0.1 Sadie Bartholomew, CMS Away Day 2025

1.1 1. Summary of JAX

"A Python library for accelerator-oriented array computation and program transformation, designed for high-performance numerical computing and large-scale machine learning"

Quick links:

- Docs: https://docs.jax.dev/
- Codebase, issue tracker etc.: https://github.com/jax-ml/jax
- 'Awesome' listing of myriad appliations: https://github.com/n2cholas/awesome-jax

Key points:

- Made by 'Big Tech' (mostly Google, some Nvidia) and relatively new to the scene - a "nascent version" of JAX was described in a 2018 paper.
- The crux: *"brings Autograd and XLA (Accelerated Linear Algebra) together"* in order to provide *"a differentiable Numpy that runs on accelerators"*. Hence the name, **J**IT, **A**utoGrad, **X**LA.
- Overall, offers means for parallel array-based computation especially targeting accelerators like GPUs and TPUs (tensor PUs), rather than traditional CPU based computing. Fast due to JIT + XLA.
- Describes itself as *"an extensible system for composable function transformations at scale"* where *'composable function transformations'* are operations taking a function and returning a new one, that can be combined arbitrarily i.e. stacked, making it very functional in style.

Comparison to similar tools:

Similar to:	…such as:	…in that:
ML frameworks	PyTorch, TensorFlow, scikit-learn	it is often use to train neural networks. though has a more functional style for writing code in, as opposed to the ML frameworks which are more OO
Efficient high-level languages	Julia	you write high-level code that runs like low-level code
Array computing libraries	NumPy	it has a corresponding API (though extra APIs via 'layered API' available)

But *not* really similar to Dask, even though you might naively think so, because:

- Dask targets (multi-core) CPUs or clusters for task-level parallelism, whereas JAX targets GPU and TPU parallelism, so they are designed to target pretty different cases.
- Both are libraries focusing on efficiency, but JAX for compute-bound numeric computation (e.g. calculus, optimisation, etc.) whereas Dask is for data-bound data access, movement, and scheduling (e.g. filtering large datasets, out-of-core transforms).

Usage Installation from PyPI i.e. via `pip` but commands differ depending on the hardware you intend to use on. At its most basic:

[1]:
```
import jax
# help(jax)
```

A layered API Has three levels of API available, depending on what you want. Possibly most notable is the NumPy-matched API, see https://docs.jax.dev/en/latest/jax.numpy.html ("starting with JAX v0.4.32, jax.Array and jax.numpy are compatible with the Python Array API Standard" https://data-apis.org/array-api/latest/ except with regard to in-place updates).

[2]:
```
# 1. jax.numpy is a high-level wrapper that provides a familiar interface. So␣
↪instead of using:
import numpy as np
# you can do:
import jax.numpy as jnp
# help(jnp)
# and use `jnp` instead of `np`.
```

[3]:
```
# 2. jax.lax is a lower-level API that is stricter and often more powerful.
# https://docs.jax.dev/en/latest/jax.lax.html
from jax import lax
# help(lax)
# 3. All JAX operations are implemented in terms of operations in XLA - the␣
↪Accelerated Linear Algebra compiler.
```

[4]:
```
# A quick explore of the numpy-like API

# Create equivalent arrays
np_array = np.linspace(0, 1, 10)
print(np_array)
jnp_array = jnp.linspace(0, 1, 10)
print(jnp_array)

# Check equality - depending on what you mean...
print(np_array == jnp_array)
print(np.array_equal(np_array, jnp_array))
print(jnp.array_equal(np_array, jnp_array))
```

```
[0.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
[0.         0.11111111 0.22222222 0.33333334 0.44444445 0.5555556
 0.6666667  0.7777778  0.8888889  1.        ]
[ True  True  True  True  True  True  True  True  True  True]
False
True
```

[5]:
```
# Warning! A key difference is that JAX arrays are always immutable, unlike␣
 ↪numpy where you can change them e.g:
np_array[0] = -1
print(np_array)
```

```
[-1.         0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
```

[6]:
```
# Won't work with JAX arrays!
jnp_array[0] = -1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 2
      1 # Won't work with JAX arrays!
----> 2 jnp_array[0] = -1

File ~/miniconda3/envs/cf-env-312-numpy2/lib/python3.12/site-packages/jax/_src/
 ↪numpy/array_methods.py:596, in _unimplemented_setitem(self, i, x)
    592 def _unimplemented_setitem(self, i, x):
    593   msg = ("JAX arrays are immutable and do not support in-place item␣
 ↪assignment."
    594          " Instead of x[idx] = y, use x = x.at[idx].set(y) or another .
 ↪at[] method:"
    595          " https://docs.jax.dev/en/latest/_autosummary/jax.numpy.ndarra.
 ↪at.html")
--> 596   raise TypeError(msg.format(type(self)))
```

```
TypeError: JAX arrays are immutable and do not support in-place item assignment
 ↪Instead of x[idx] = y, use x = x.at[idx].set(y) or another .at[] method: https:
 ↪//docs.jax.dev/en/latest/_autosummary/jax.numpy.ndarray.at.html
```

[7]:
```
# Following the advice from the traceback, note you can do what you wanted
 ↪(though it requires creation of
# a new array object in a functional array update) via:
jnp_array = jnp_array.at[0].set(-1)
print(jnp_array)
```

```
[-1.          0.11111111  0.22222222  0.33333334  0.44444445  0.5555556
  0.6666667   0.7777778   0.8888889   1.        ]
```

[8]:
```
# Similar wrapper for SciPy! JAX-based implementations of SciPy API likewise is:
# https://docs.jax.dev/en/latest/jax.scipy.html
import jax.scipy as jscipy
help(jscipy)
```

```
Help on package jax.scipy in jax:

NAME
    jax.scipy

DESCRIPTION
    # Copyright 2018 The JAX Authors.
    #
    # Licensed under the Apache License, Version 2.0 (the "License");
    # you may not use this file except in compliance with the License.
    # You may obtain a copy of the License at
    #
    #     https://www.apache.org/licenses/LICENSE-2.0
    #
    # Unless required by applicable law or agreed to in writing, software
    # distributed under the License is distributed on an "AS IS" BASIS,
    # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    # See the License for the specific language governing permissions and
    # limitations under the License.

PACKAGE CONTENTS
    cluster (package)
    fft
    integrate
    interpolate (package)
    linalg
    ndimage
    optimize (package)
    signal
```

```
    sparse (package)
    spatial (package)
    special
    stats (package)

FILE
    /home/slb93/miniconda3/envs/cf-env-312-numpy2/lib/python3.12/site-
packages/jax/scipy/__init__.py
```

Short basic examples

A. Auto-diff with e.g. `jax.grad` and linear algebra with `jax.linalg` Typical applications of JAX in primitive form - some automatic differentation and linear algebra

A1. Basic scalar example, finding a gradient

[9]:
```
def scalar_fn(x):
    return x**3  # scalar output

grad_f = jax.grad(scalar_fn)  # 1st derivative: 3*x^2
second_grad_f = jax.grad(grad_f)  # 2nd derivative: 6*x

x = 2.0
print(second_grad_f(x))  # prints 12.0
```

```
12.0
```

A2. Vector examples: define a matrix A and vector b to play with

[10]:
```
A = jnp.array([[3.0, 2.0], [1.0, 4.0]])
b = jnp.array([7.0, 10.0])

# Solve the linear system Ax = b
x = jnp.linalg.solve(A, b)
print("Solution x is:", x)

# Define function that returns a scalar first element of Ax = b solution,␣
 ↪similar to above
def matrix_fn(A):
    return jnp.linalg.solve(A, b)[0]  # scalar output

# First derivative: grad of f
grad_f = jax.grad(matrix_fn)

# Second derivative: grad of the sum of grad_f outputs (to reduce to scalar)
second_grad_f = jax.grad(lambda A: jnp.sum(grad_f(A)))
```

```
grad_val = grad_f(A)
second_grad_val = second_grad_f(A)

print("First derivative shape:", grad_val.shape)  # Should be (2, 2)
print("First derivative:\n", grad_val)

print("Second derivative shape:", second_grad_val.shape)  # Also (2, 2) in this␣
 ↪case
print("Second derivative (gradient of summed gradient):\n", second_grad_val)
```

```
Solution x is: [0.8000001 2.3      ]
First derivative shape: (2, 2)
First derivative:
 [[-0.32000005 -0.9200001 ]
 [ 0.16000003  0.46000004]]
Second derivative shape: (2, 2)
Second derivative (gradient of summed gradient):
 [[ 0.29600003  0.38600004]
 [-0.10800001 -0.07800002]]
```

A3. Matrix operations and decompositions, using matrix A as an example:

[11]:
```
#   Matrix multiplication
C = jnp.matmul(A, A.T)
print("A * A.T is:\n", C)
#   Inverse
A_inv = jnp.linalg.inv(A)
print("Inverse is:\n", A_inv)
#   Determinant
det_A = jnp.linalg.det(A)
print("Determinant is:\n", det_A)
#   Singular Value Decomposition
U, S, Vh = jnp.linalg.svd(A)
print("SVD values are:\n", U, S, Vh)
#   Eigenvalues
eigvals = jnp.linalg.eigvals(A)
print("Eigenvalues are:\n", eigvals)
```

```
A * A.T is:
 [[13. 11.]
 [11. 17.]]
Inverse is:
 [[ 0.40000004 -0.20000002]
 [-0.10000001  0.3       ]]
Determinant is:
 10.0
SVD values are:
 [[-0.64074737 -0.76775175]
 [-0.7677517   0.64074737]] [5.116672  1.9543954] [[-0.5257311 -0.8506508]
```

```
 [-0.8506508  0.5257311]]
Eigenvalues are:
 [2.+0.j 5.+0.j]
```

A4. Adding JIT compilation to aim to speed it up

[12]:
```python
@jax.jit
def scalar_fn(x):
    return x**3

@jax.jit
def matrix_fn(A):
    return jnp.linalg.solve(A, b)[0]  # scalar output

# Same outputs, potential speed-up
```

Machine learning is mostly, under-the-hood, calculus (as I understand it) therefore the above snippets should demonstrate to you how JAX can be used to streamline and facilitate ML, etc.

1.1.1 B. Auto-vectorisation with `vmap`

Another key function is `jax.vmap` for automatic vectorisation. jax.vmap automatically vectorizes a function, letting you apply it in parallel across a batch of inputs — without writing explicit loops.

B1. Basic example - batch solving the Ax = b from above

[13]:
```python
# Batch of b vectors (e.g., 3 different right-hand sides)
B = jnp.array([
    [7.0, 10.0],
    [1.0, 1.0],
    [4.0, 5.0]
])

# Function to solve Ax = b for fixed A
def matrix_fn_A_fixed(b):
    return jnp.linalg.solve(A, b)

# Vectorize over the 0th axis of B
batched_solve = jax.vmap(matrix_fn_A_fixed)

X = batched_solve(B)  # Each row of X is a solution to Ax_i = b_i
print("Solutions for each b:\n", X)
```

```
Solutions for each b:
 [[0.8000001  2.3       ]
 [0.20000002 0.2       ]
 [0.6        1.1       ]]
```

1.2 2. JAX for CMS

From the above, I suspect JAX may be useful to us in these contexts:

- Designed for use on accelerated hardware, so anywhere we are doing work where we know the architecture will have GPUs (or TPUs)
- But for CPU-only systems it can still be useful since it offers efficient automatic differentiation
- As it is Python-based with no bindings or interfaces for other languages, it would be most suitable for use in Python codes we work with as opposed to anything in other languages

More specifically... ...it could potentially be used in Python code we are reponsible for or contribute to, as:

- a direct replacement for NumPy operations where mutation isn't required, *assuming* correct context and sufficient testing is done to be confident it results in speed-up rather than possible slowing;
- where efficient calculus and linear algebra operations on scalars, vectors or matrices/tensors are required;
- for any ML training we might want to do.

For instance, one concrete idea I had was that we could perhaps upgrade our methematical operation offerings in cf-python to:

- make our calculus operations such as derivative, gradient, laplacian, div and curl methods more efficient;
- add linear algebra data functionality - we know that folk would find this useful at least thinking towards use for ML and similar (thinking of David Case's CMS meeting talk a while back regarding PyTorch)
