Differentiable Computation with Awkward Array and JAX

Fifth MODE workshop on Differentiable Programming for Experiment Design

Ianna Osborne (Princeton University), Saransh Chopra (University College London)

A quick introduction

“Generalist” Research Software Engineer at Advanced Research Computing Centre, University College London

- Autodiff and GPU support for cosmological software, kubernetes deployment for bioimaging software, and a knack for teaching/designing course materials
- Not a cosmologist, not a biologist, not a professor :(

Worked on Scikit-HEP as a fellow/intern in 2022 and 2024 (at CERN) under Princeton University (IRIS-HEP)

- **Autodiff support for Awkward**, GPU support for histogramming, symbolic support for vector, internal migrations, other features, …, still maintains some software
- Not a physicist :(

Developed mathematical theorem provers (type theory and formal verification) before this (theoretical CS and Math, now we’re getting there)

Graduated (2024) with a major in CS and Math from University of Delhi, India :)

Contributes to/maintains several open-source scientific software written in Python in free time

A quick introduction to Awkward Array

Awkward Array

Collision of particles (“events”) at HEP experiments produces enormous nested, jagged/ragged, vectorizable, and “struct” (each particle has several data points) like data.

Before Awkward, the Python array (NumPy and co) ecosystem could not handle or had very limited capability to handle such data (because of the raggedness of data).

Awkward Array is a library for nested, variable-sized data, including arbitrary-length lists, records, mixed types, and missing data, using NumPy-like idioms.

Under the hood, array operations are compiled and fast.

The library was built for particle physics, but it has now become a generic Scientific Python library.

A quick introduction to Awkward Array

Awkward Array

Given an array of lists of objects with `x`, `y` fields (with nested lists in the `y` field),

```
import awkward as ak

array = ak.Array([
    [{"x": 1.1, "y": [1]}, {"x": 2.2, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}],
    [],
    [{"x": 4.4, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}]
])
```

A quick introduction to Awkward Array

the following slices out the `y` values, drops the first element from each inner list, and runs NumPy's `np.square` function on everything that is left:

```
output = np.square(array["y", ..., 1:])
```

A quick introduction to Awkward Array

The result is

```
[
    [[], [4], [4, 9]],
    [],
    [[4, 9, 16], [4, 9, 16, 25]]
]
```

Other ragged data structures in Python

```
In [2]: nt = torch.nested.nested_tensor([torch.arange(12).reshape(
   ...:     2, 6), torch.arange(18).reshape(3, 6)], dtype=torch.float, device=device)
   ...: print(f"{nt=}")
/Users/saransh/Code/HEP/MODE24-awkward-jax/.venv/lib/python3.13/site-packages/torch/nested/__init__.p
y:250: UserWarning: The PyTorch API of nested tensors is in prototype stage and will change in the ne
ar future. We recommend specifying layout=torch.jagged when constructing a nested tensor, as this lay
out receives active development, has better operator coverage, and works with torch.compile. (Trigger
ed internally at /Users/runner/work/pytorch/pytorch/pytorch/aten/src/ATen/NestedTensorImpl.cpp:182.)
  return _nested.nested_tensor(
nt=nested_tensor([
  tensor([[ 0.,  1.,  2.,  3.,  4.,  5.],
          [ 6.,  7.,  8.,  9., 10., 11.]]),
  tensor([[ 0.,  1.,  2.,  3.,  4.,  5.],
          [ 6.,  7.,  8.,  9., 10., 11.],
          [12., 13., 14., 15., 16., 17.]])
])
```

Torch (experimental, lacks several awkward features)

Not developed anymore

Extensive library for columnar data; being actively developed; ak.to_arrow

Tensorflow (lacks several awkward features)

```
In [2]: digits = tf.ragged.constant([[3, 1, 4, 1], [], [5, 9, 2], [6], []])
   ...: words = tf.ragged.constant([["So", "long"], ["thanks", "for", "all", "the", "fish"]])

In [3]: digits, words
Out[3]:
(<tf.RaggedTensor [[3, 1, 4, 1], [], [5, 9, 2], [6], []]>,
 <tf.RaggedTensor [[b'So', b'long'], [b'thanks', b'for', b'all', b'the', b'fish']]>)
```

Had in v2, no support in v3 (open feature request)

Need for Automatic Differentiation

We know that we can use gradient based methods in particle physics to -

- check if an event originated from signal or background processes
- find cut positions
- …

But what if we can use gradient based methods on any part of the pipeline and use any loss function for optimisation? Optimise any parameter with respect to any goal?

Or, what if we can treat the entire pipeline as a single optimisation problem?

Simpson, N. (2023). Data Analysis in High-Energy Physics as a Differentiable Program. Lund University.

Need for Automatic Differentiation

> tools are often optimized for performance on a particular task
> that is several steps removed from the ultimate physical goal of
> searching for a new particle or testing a new physical theory.

> sensitivity to high-level physics questions must account for systematic uncertainties,
> which involve a nonlinear trade-off between the typical machine learning performance
> metrics and the systematic uncertainty estimates

Guest, D., Cranmer, K., & Whiteson, D. (2018). Deep Learning and its Application to LHC Physics. Ann. Rev. Nucl. Part. Sci., 68, 161–181.

Need for Automatic Differentiation

relaxed

Provides differentiable ("relaxed") versions of common operations in high-energy physics.

Based on `jax`. Where possible, function APIs try to mimic their commonly used counterparts, e.g. fitting and hypothesis testing in `pyhf`.

(Simpson, 2023) introduced a way to make an entire HEP analysis pipeline differentiable by breaking it down into chunks and using chain rule.

$$\frac{\partial \text{objective}}{\partial \varphi} = \frac{\partial \text{objective}}{\partial \text{likelihood}} \times \frac{\partial \text{likelihood}}{\partial \text{model parameters}} \times \frac{\partial \text{model parameters}}{\partial \text{cut values}} \times \dots$$

The thesis even experimented with differentiating through discrete things, like histograms and profile likelihood ratios.

But it left out one thing, differentiating through ragged arrays and operations on ragged arrays.

Also see:

Differentiable Programming in the Scikit-HEP Ecosystem

9 Jun 2025, 16:00
25m
OAC conference center, Kolymbari, Crete, Greece.

Talk | Methods and tools | Methods and tools

neos

neural end-to-end-optimised summary statistics

Simpson, N. (2023). Data Analysis in High-Energy Physics as a Differentiable Program. Lund University.

Differentiating through ragged data structures

```
In [2]: nt = torch.nested.nested_tensor([torch.arange(12).reshape(
   ...:     2, 6), torch.arange(18).reshape(3, 6)], dtype=torch.float, device=device)
   ...: print(f"{nt=}")
/Users/saransh/Code/HEP/MODE24-awkward-jax/.venv/lib/python3.13/site-packages/torch/nested/__init__.p
y:250: UserWarning: The PyTorch API of nested tensors is in prototype stage and will change in the ne
ar future. We recommend specifying layout=torch.jagged when constructing a nested tensor, as this lay
out receives active development, has better operator coverage, and works with torch.compile. (Trigger
ed internally at /Users/runner/work/pytorch/pytorch/pytorch/aten/src/ATen/NestedTensorImpl.cpp:182.)
  return _nested.nested_tensor(
nt=nested_tensor([
  tensor([[ 0.,  1.,  2.,  3.,  4.,  5.],
          [ 6.,  7.,  8.,  9., 10., 11.]]),
  tensor([[ 0.,  1.,  2.,  3.,  4.,  5.],
          [ 6.,  7.,  8.,  9., 10., 11.],
          [12., 13., 14., 15., 16., 17.]])
])
```

Torch (not differentiable)

Not developed anymore

Not differentiable

Tensorflow (differentiable using gradient tape)

```
In [2]: digits = tf.ragged.constant([[3, 1, 4, 1], [], [5, 9, 2], [6], []])
   ...: words = tf.ragged.constant([["So", "long"], ["thanks", "for", "all", "the", "fish"]])

In [3]: digits, words
Out[3]:
(<tf.RaggedTensor [[3, 1, 4, 1], [], [5, 9, 2], [6], []]>,
 <tf.RaggedTensor [[b'So', b'long'], [b'thanks', b'for', b'all', b'the', b'fish']]>)
```

Had in v2, no support in v3 (open feature request)

A quick introduction to JAX

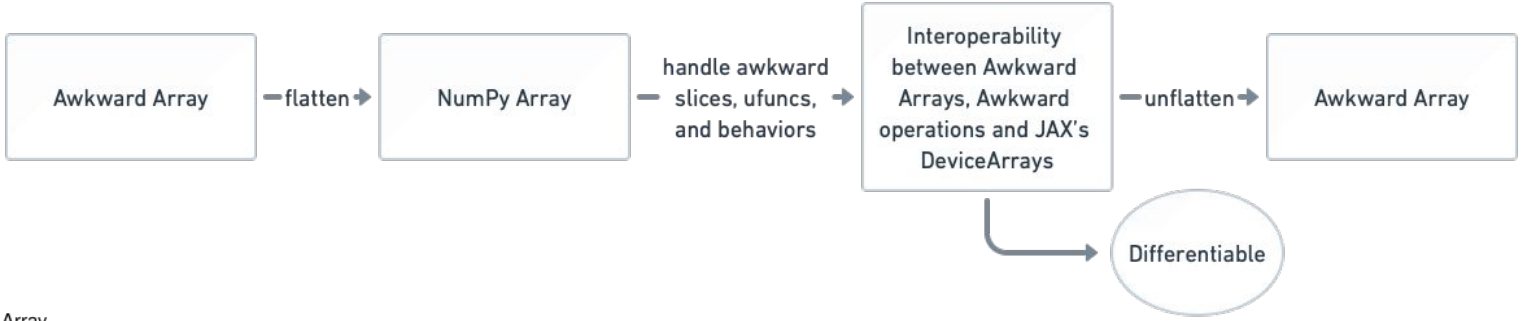
JAX is Autograd and XLA, brought together for high-performance numerical computing. JAX has a high-level NumPy compatible (`jax.numpy`) module that is array API compliant.

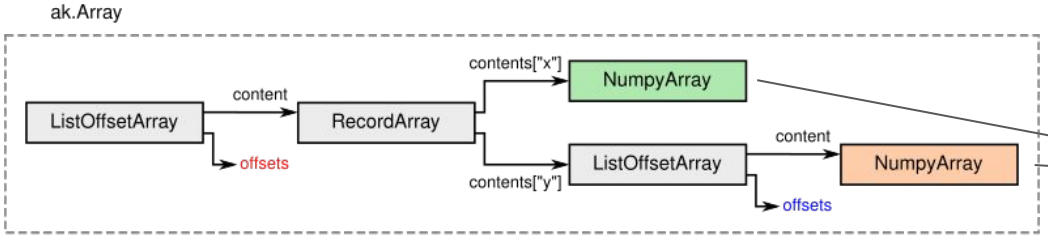
Unlike TensorFlow, we do not need a gradient tape or a meta-language within Python to monitor executions.

JAX has no support for ragged arrays, but one can make custom Python data containers compatible with JAX by registering a way to flatten and unflatten them (registering `pytree` nodes).

JAX follows the functional programming paradigm!

Differentiating Awkward arrays (internals)





represents the following (color-coded):

```
[
    [{"x": 1.1, "y": [1]}, {"x": 2.2, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}],
    [],
    [{"x": 4.4, "y": [3, 2]}, {"x": 5.5, "y": [3]}]
]
```

Differentiating Awkward arrays (example)

```
In [1]: import jax
   ...: jax.config.update("jax_platform_name", "cpu")

In [2]: import awkward as ak
   ...: ak.jax.register_and_check()
```

Differentiating Awkward arrays (example)

```
In [3]: def reverse_sum(array):
   ...:     return ak.sum(array[::-1], axis=0)
   ...:

In [4]: array = ak.Array([[1.0, 2.0, 3.0], [], [4.0, 5.0]], backend="jax")

In [5]: reverse_sum(array)
Out[5]: <Array [5.0, 7.0, 3.0] type='3 * float64'>
```

Differentiating Awkward arrays (example)

```
In [6]: tangent = ak.Array([[0.0, 0.0, 0.0], [], [0.0, 1.0]], backend="jax")

In [7]: value_jvp, jvp_grad = jax.jvp(reverse_sum, (array,), (tangent,))

In [8]: value_jvp
Out[8]: <Array [5.0, 7.0, 3.0] type='3 * float64'>

In [9]: assert value_jvp.to_list() == reverse_sum(array).to_list()

In [10]: jvp_grad
Out[10]: <Array [0.0, 1.0, 0.0] type='3 * float64'>
```

Differentiating Awkward arrays (example)

```
In [11]: value_vjp, func_vjp = jax.vjp(reverse_sum, array)

In [12]: assert value_vjp.to_list() == reverse_sum(array).to_list()

In [13]: cotanget = ak.Array([0., 1., 0.], backend="jax")

In [14]: func_vjp(value_vjp)
Out[14]: (<Array [[5.0, 7.0, 3.0], [], [5.0, 7.0]] type='3 * var * float64'>,)
```

Awkward Arrays and JAX (support and peculiarities)

`jax.jvp`, `jax.vjp`, and `jax.grad` functions are supported on Awkward arrays **only** with

- any combination of ufunc operations like `x + y`
- all reducers like `ak.sum()`
  - using the internal `jax.ops` API and custom “segment” reducers
- any slicing like `x[1:]`

Only automatic differentiation is supported and

- JIT compilation is not (or cannot be) supported - use Numba for JIT compilation!
- GPU/TPU scaling is not supported (or has not been experimented/tested with) - use the CUDA backend for GPU support!

Ongoing work

Regular bug fixes

Refactoring to keep the backend's functionality consistent with other backends

Feature additions to keep the backend's functionality consistent with other backends

Some integration efforts (support automatic differentiation in the Analysis Grand Challenge) -

Future plans

More robust tests and test coverage for the JAX backend

Better support for other JAX functionalities, such as jacobian

Keep the backend's functionality consistent with other backends

More integration efforts - support automatic differentiation in the Analysis Grand Challenge

Thank you!