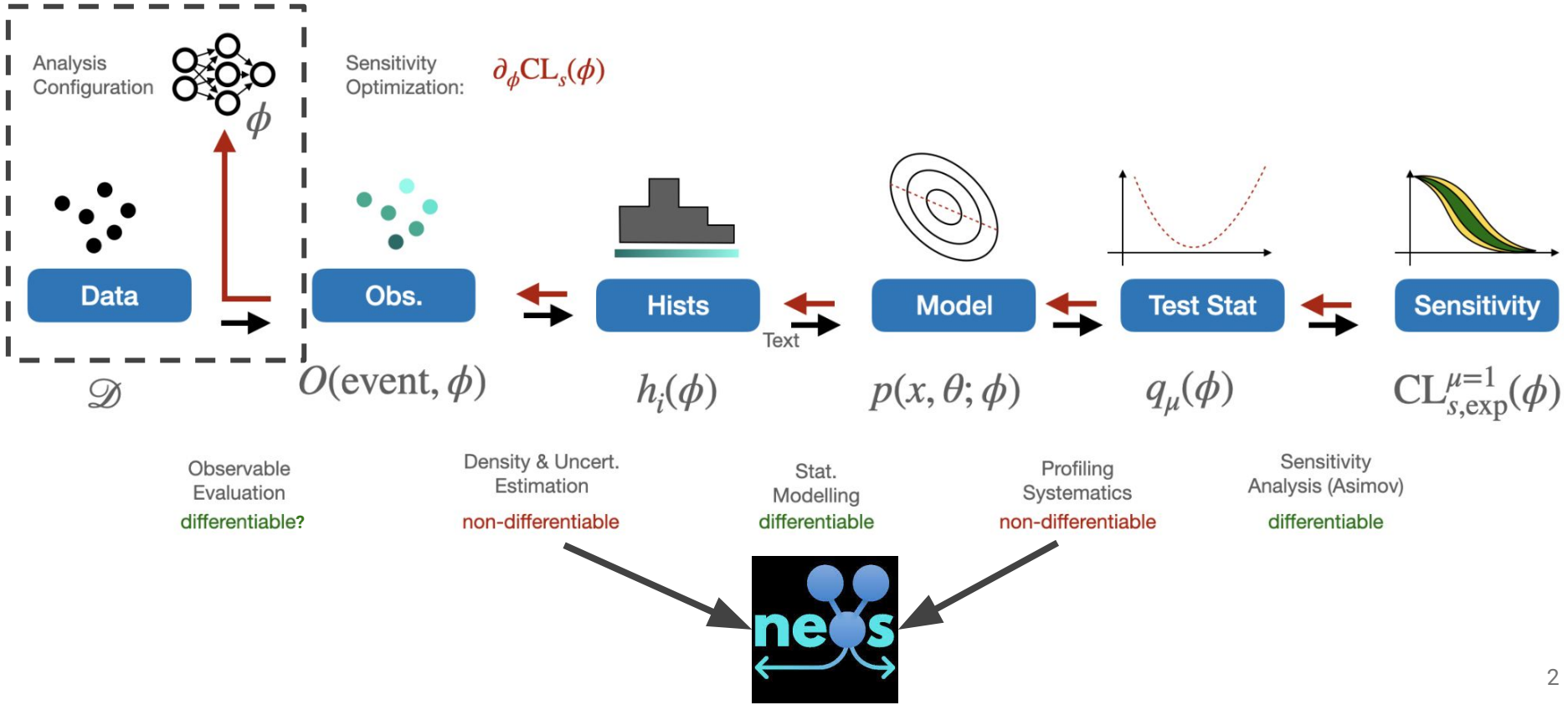
Differentiating Awkward Arrays Using JAX

Jim Pivarski
IRIS - HEP Mentor
Princeton University

Lukas Heinrich
IRIS - HEP Mentor
CERN

Anish Biswas
IRIS - HEP Fellow
Manipal Institute Of Technology

The IRIS-HEP Grand Analysis Challenge



Defining the goals of this project

- "*JAX is NumPy on the CPU, GPU, and TPU, with great automatic differentiation for high-performance machine learning research.*"
- One of the goals of the project, was to figure out whether or not we could use JAX to enable differentiation of functions involving Awkward Arrays, and this becomes a good problem since Awkward Arrays are by nature not rectilinear data buffers.
- Can the resulting architecture handle the most commonly used differentiation use-cases?
- Can we do better than writing another auto-differentiation library specifically for Awkward Arrays?

Awkward Array

Device Arrays ⇔ Awkward Arrays

- Even before we start discussing about how we can enable JAX to find gradients of functions containing Awkward Arrays, we need to set up an interoperability standard between JAX and Awkward Arrays.
- JAX exposes a JAX Class Type called `DeviceArray`, which is used as the primary data type for differentiation.
- We are currently using `dlpack` to achieve zero-copy interoperability with JAX `DeviceArrays`. Converting between them is as simple as calling the `ak.to_jax()` or `ak.from_jax()`.
- `dlpack` is an improvement even if JAX was not involved. It is a more general/systematic way of sharing arrays than what we had been using which was `__cuda_array_interface__`.
- As a syntactic sugar, ak.Array also accepts JAX `DeviceArrays` directly.

```
ak.Array(jax.numpy.array([1,2,3,4,5]))
```

```
<Array [1, 2, 3, 4, 5] type='5 * int64'>
```

```
jax.config.update("jax_enable_x64", True)
jax_array = jax.numpy.array([[1, 2], [3, 4], [5, 6]])
ak.from_jax(jax_array)
```

```
<Array [[1, 2], [3, 4], [5, 6]] type='3 * 2 * int64'>
```

```
ak.Array(jax_array)
```

```
<Array [[1, 2], [3, 4], [5, 6]] type='3 * var * int64'>
```

```
ak.to_jax(ak.Array(jax_array))
```

```
DeviceArray([[1, 2],
             [3, 4],
             [5, 6]], dtype=int64)
```

What are ufuncs?

- Numpy Universal functions that allow operations on any class that implements the `array_ufunc` class.
- These functions include square roots, trigonometric functions etc.

```
def func(x):
    return np.square(x[:-1])
```

```
from math import pi
listoffsetarray = ak.Array([[1., 2., 3.], [], [4., 5.]])
listoffsetarray_tangent = ak.Array([[0.0, 1.0, 0.0], [], [0.0, 0.0]])
```

```
jvp_value, jvp_grad = jax.jvp(func, (listoffsetarray, ), (listoffsetarray_tangent, ))
```

```
jvp_value
```

```
<Array [[1, 4, 9], []] type='2 * var * float32'>
```

```
jvp_grad
```

```
<Array [[0, 4, 0], []] type='2 * var * float32'>
```

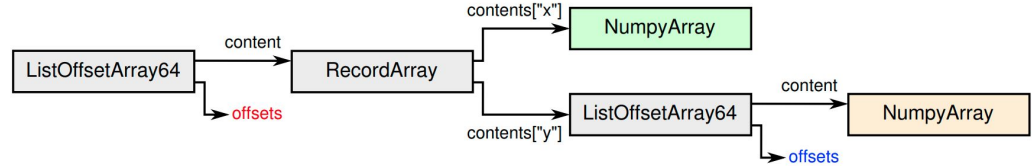
A Prelude to JAX Technicalities

- JAX does a number of fantastic things behind the scenes, so if you'd like to read about that in more details, you should check out the Autodidax Documentation of JAX.
- However, I'll try to stick to what is absolutely prerequisite to understand the "why" of the slides following this.
- JAX uses `PyTrees` as a way to expose JAX transformations for other Python Data Containers. To implement this, all we need to do is tell JAX how to break this complex Python Data Container into linear buffers, and also how to re-assemble them back into this Python Data Container structure.
- These linear buffers, then get converted into JAX `Tracer` objects, which keep a track of all the computations that is happening to the data buffer. As such, you can catch such Tracers flowing freely through your functions midway through any computation. This might also cause complications since, many functions which are not native to JAX, might fail to correctly treat these Tracers in the same way as they would treat their Python Data Containers. Example: Slicing of Awkward Arrays.

How does JAX interact with Awkward Arrays?

Flatten the Awkward Array → Handle Slices and ufuncs → Unflatten the Awkward Array

```
ak.Array([
        [{"x": 1,  "y": [11]},
         {"x": 4,  "y": [12, 22]},
         {"x": 9,  "y": [13, 23, 33]}],
        [],
        [{"x": 16, "y": [14, 24, 34, 44]}]
])
```




```
ak.Array([
        [{"x": *, "y": [*]},
         {"x": *, "y": [*, *]},
         {"x": *, "y": [*, *, *]}],
        [],
        [{"x": *, "y": [*, *, *, *]}]
])
```

Awkward Array

Let’s take an example!

```
import awkward as ak
import jax
import numpy as np

jax.config.update("jax_platform_name", "cpu")

ak.jax.register()
```

```
def func(x):
    return np.power(x[[2, 2, 0], ::-1], 3)
```

```
listoffsetarray = ak.Array([[1., 2., 3.], [], [4., 5.]])
listoffsetarray_tangent = ak.Array([[0.0, 1.0, 0.0], [], [0.0, 0.0]])
```

```
jvp_value, jvp_grad = jax.jvp(func, (listoffsetarray, ), (listoffsetarray_tangent, ))
```

```
jvp_value
```

```
<Array [[125, 64], [125, 64], [27, 8, 1]] type='3 * var * float32'>
```

```
jvp_grad
```

```
<Array [[0, 0], [0, 0], [0, 12, 0]] type='3 * var * float32'>
```

View of the Array before and after the slice.

Before the Slice:

```
listoffsetarray.layout
```

```
<ListOffsetArray64>
    <offsets><Index64 i="[0 3 3 5]" offset="0" length="4" at="0x0000029c3770"/></offsets>
    <content><NumpyArray format="d" shape="5" data="1 2 3 4 5" at="0x000002a1bdb0"/></content>
</ListOffsetArray64>
```

After the Slice:

```
listoffsetarray[[2, 2, 0], ::-1].layout
```

```
<ListOffsetArray64>
    <offsets><Index64 i="[0 2 4 7]" offset="0" length="4" at="0x000002aa7420"/></offsets>
    <content><NumpyArray format="d" shape="7" data="5 4 5 4 3 2 1" at="0x000002bb6720"/></content>
</ListOffsetArray64>
```

Before the slice, the linear buffer of `` `[1, 2, 3, 4, 5]` `` had a **JAX `Tracer` associated with it**. It now becomes important to **slice this `Tracer`**(since JAX passes these `Tracers` objects, throughout the function computation and it needs to know how our function maps **`A -> B`**) in such a way that it **starts representing our Array after the slice**. Since, slices can be complex in Awkward Arrays and most of them don't map onto linear buffers, we make use of **`Identities`**.

Enter Identities

```
ak.to_list(listoffsetarray)
```

```
[[1.0, 2.0, 3.0], [], [4.0, 5.0]]
```

```
np.asarray(listoffsetarray.layout.content.identities)
```

```
array([[0, 0],
       [0, 1],
       [0, 2],
       [2, 0],
       [2, 1]], dtype=int64)
```

```
ak.to_list(listoffsetarray[[2, 2, 0], ::-1])
```

```
[[5.0, 4.0], [5.0, 4.0], [3.0, 2.0, 1.0]]
```

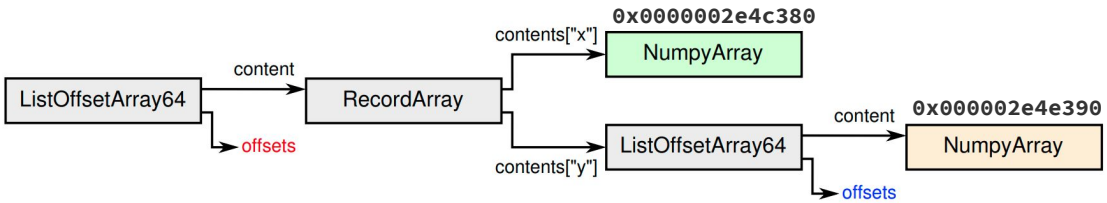
```
np.asarray(listoffsetarray[[2, 2, 0], ::-1].layout.content.identities)
```

```
array([[2, 1],
       [2, 0],
       [2, 1],
       [2, 0],
       [0, 2],
       [0, 1],
       [0, 0]], dtype=int64)
```

What if there are multiple Tracers?

- There can arise a case where there are more than one children buffers. An easy example is a Record Array.
- What happens when we have more than one buffer which results in multiple Tracer objects as well? How do we differentiate between these two buffers?
- We need to figure out a way to track which buffer the slice is acting on. Sometimes, a slice might make a new copy which combines elements from two or more buffers.

```
ak.Array([
      [{"x": 1,  "y": [11]},
       {"x": 4,  "y": [12, 22]},
       {"x": 9,  "y": [13, 23, 33]}],
      [],
      [{"x": 16, "y": [14, 24, 34, 44]}]
])
```



What if there are multiple Tracers?

The Solution:

- If the slices don't create a new copy, keep a dictionary which maps those `NumpyArray pointers` to the index in the buffer list.

  `HashMap(ptrs -> index) -> {0x0000002e4c380: 0, 0x000002e4e390: 1}`

```
array["x"].layout
```

```
<NumpyArray format="d" shape="3" data="1.1 2.2 3.3" at="0x000002e4c380">
    <Identities32 ref="8" fieldloc="0: x" width="1" offset="0" length="3" at="0x000002cfd280"/>
</NumpyArray>
```

- If the slices do create a new copy, use `identities` to collect all `Tracers` and make the new linear buffer.

```
array["x", [0, 2]].layout
```

```
<NumpyArray format="d" shape="2" data="1.1 3.3" at="0x0000027d6400">
    <Identities32 ref="8" fieldloc="0: x" width="1" offset="0" length="2" at="0x0000027ee8d0"/>
</NumpyArray>
```

What JAX functions do we support?

- For now, `jax.jvp` and `jax.jit` are tested extensively.
- `jax.vjp` or reverse mode differentiation is in an experimental stage. The limitations are posted in more detail as a documentation on the www.awkward-array.org website.

```
def func(array):
    return np.sin(array[::-1] ** 2)
```

```
listoffsetarray = ak.Array([[1.0, 2.0, 3.0], [], [4.0, 5.0]])
listoffsetarray_tangent = ak.Array([[0.0, 1.0, 0.0], [], [0.0, 0.0]])
```

```
jit_value = jax.jit(func)(listoffsetarray)
```

```
jit_value
```

```
<Array [[-0.288, -0.132], ... -0.757, 0.412]] type='3 * var * float64'>
```

Future Work

- Awkward Scalars are Python numbers, while JAX scalars are 0-dimensional arrays. There has to be a notion of a scalar in the Awkward Array library to support reverse mode differentiation using JAX. Currently the only way is to generate the scalar in a way that `jax.vjp` works correctly is in the form of an Awkward Array using slicing, like `array[0:1]`.
- Specialized functions / reducers, like `ak.sum()` and `ak.prod()` must be implemented in terms of JAX primitives to have function containing such reducers to be correctly differentiable.
- Writing higher-level utility functions, like `ak.jax.jacobian`, `ak.jax.grad`, `ak.jax.jacfwd`, which is built from elementary auto-differentiation functions i.e `jax.vjp` and `jax.jvp`.
- Eventually, we'd like to have more `ak.*` functions, to be defined in terms of JAX primitives, which would allow for a lot more operations in the functions, without having to worry about some trivial operation driving all the gradients to 0.

THANK YOU!


trickarcher


anishbiswas271@gmail.com