Autograd’s implementation

[github.com/hips/autograd](https://github.com/hips/autograd)

Dougal Maclaurin, David Duvenaud, Matt Johnson

- differentiates native Python code
- handles most of Numpy + Scipy
- loops, branching, recursion, closures
- arrays, tuples, lists, dicts...
- derivatives of derivatives
- a one-function API!
autodiff implementation options
A. direct specification of computation graph
B. source code inspection
C. monitoring function execution
ingredients:
1. tracing composition of primitive functions
2. vector-Jacobian product for each primitive
3. composing VJPs backward
ingredients:
1. tracing composition of primitive functions
2. vector-Jacobian product for each primitive
3. composing VJPs backward
numpy.sum
primitive

autograd.numpy.sum

numpy.sum
Node a

value: a
function: F
parents: [x]

primitive

autograd.numpy.sum

numpy.sum
primitive

Node \(\tilde{a}\)

value: \(a\)
function: \(F\)
parents: \([\tilde{x}]\)

Node \(\tilde{b}\)

value: \(b\)
function: \(\text{anp.sum}\)
parents: \([\tilde{a}]\)

\[
\begin{align*}
\text{unbox} & \quad \rightarrow \quad \text{numpy.sum} \\
\text{numpy.sum} & \quad \rightarrow \quad \text{box}
\end{align*}
\]
class Node(object):
    __slots__ = ['value', 'recipe', 'progenitors', 'vspace']

    def __init__(self, value, recipe, progenitors):
        self.value = value
        self.recipe = recipe
        self.progenitors = progenitors
        self.vspace = vspace(value)
class primitive(object):
    def __call__(self, *args, **kwargs):
        argvals = list(args)

        parents = []
        for argnum, arg in enumerate(args):
            if isnode(arg):
                argvals[argnum] = arg.value
                if argnum in self.zero_vjps: continue
            parents.append((argnum, arg))

        result_value = self.fun(*argvals, **kwargs)
        return new_node(result_value, (self, args, kwargs, parents), )
def forward_pass(fun, args, kwargs, argnum=0):
    args = list(args)
    start_node = new_progenitor(args[argnum])
    args[argnum] = start_node
    active_progenitors.add(start_node)
    end_node = fun(*args, **kwargs)
    active_progenitors.remove(start_node)
    return start_node, end_node
start_node

x
\[ a = A(x) \]
\[ a = A(x) \]

\[ b = B(a) \]
$b = B(a)$

$a = A(x)$

$c = C(b)$
$x = A(x)$

$b = B(a)$

$c = C(b)$

$y = D(c)$
No control flow!
ingredients:
1. tracing composition of primitive functions
2. vector-Jacobian product for each primitive
3. composing VJPs backward
\[ a = A(x) \]
\[ a = A(x) \]
\frac{\partial y}{\partial x} = \ ? \\
\frac{\partial y}{\partial a} \\
x \rightarrow a = A(x)
\[
\frac{\partial y}{\partial x} = \frac{\partial y}{\partial a} \cdot \frac{\partial a}{\partial x}
\]

\[a = A(x)\]
\[
\frac{\partial y}{\partial x} = \frac{\partial y}{\partial a} \cdot A'(x) \quad \frac{\partial y}{\partial a}
\]

\[
x \quad a = A(x)
\]
```

```
ingredients:
1. tracing composition of primitive functions
2. vector-Jacobian product for each primitive
3. composing VJPs backward
\[
\begin{aligned}
    x & \rightarrow a = A(x) \\
    & \rightarrow b = B(a) \\
    & \rightarrow c = C(b) \\
    & \rightarrow y = D(c)
\end{aligned}
\]
\( a = A(x) \)
\( b = B(a) \)
\( c = C(b) \)
\( y = D(c) \)
\( \frac{\partial y}{\partial y} = 1 \)
\[ a = A(x) \]
\[ b = B(a) \]
\[ c = C(b) \]
\[ \frac{\partial y}{\partial y} = 1 \]
\[ y = D(c) \]
The image contains a diagram with labeled nodes and annotations. The nodes are connected by arrows indicating the flow of the process. The nodes and their annotations are as follows:

- **start_node** (x): \[ a = A(x) \]
- **middle_node 1**: \[ b = B(a) \]
- **middle_node 2**: \[ c = C(b) \]
- **end_node**: \[ y = D(c), \frac{\partial y}{\partial y} = 1 \]

The diagram illustrates a chain of operations, starting with \( x \), passing through \( a \), \( b \), and \( c \), leading to \( y \). The notation \( \frac{\partial y}{\partial b} \) and \( \frac{\partial y}{\partial c} \) indicates partial derivatives along the path.
\[
\frac{\partial y}{\partial a} = A(x) \quad \frac{\partial y}{\partial b} = B(a) \quad \frac{\partial y}{\partial c} = C(b) \quad \frac{\partial y}{\partial y} = 1
\]

\[a = A(x)\quad b = B(a)\quad c = C(b)\quad y = D(c)\]
\[
\begin{align*}
\frac{\partial y}{\partial x} &= x \\
\frac{\partial y}{\partial a} &= a = A(x) \\
\frac{\partial y}{\partial b} &= b = B(a) \\
\frac{\partial y}{\partial c} &= c = C(b) \\
\frac{\partial y}{\partial y} &= y = D(c) = 1
\end{align*}
\]
higher-order autodiff just works: the backward pass can itself be traced
\[
\begin{align*}
\frac{\partial y}{\partial y} &= 1 \\
\begin{align*}
\text{start_node} &\quad x \\
\quad &\quad a = A(x) \\
\quad &\quad b = B(a) \\
\quad &\quad c = C(b) \\
\quad &\quad y = D(c) \\
\end{align*}
\end{align*}
\]
\[ a = A(x) \]
\[ b = B(a) \]
\[ c = C(b) \]
\[ y = D(c) \]
\[
\frac{\partial y}{\partial b} = \frac{\partial y}{\partial c}
\]
\[
\frac{\partial y}{\partial y} = 1
\]

Diagram:

- Start node: \( x = A(x) \)
- \( a = A(x) \)
- \( b = B(a) \)
- \( c = C(b) \)
- End node: \( y = D(c) \)
\[ \frac{\partial y}{\partial x} = 1 \]

\[ x = A(x) \]

\[ a = B(a) \]

\[ b = C(b) \]

\[ y = D(c) \]
\[ \frac{\partial y}{\partial y} = 1 \]

\[ a = A(x) \]

\[ b = B(a) \]

\[ c = C(b) \]

\[ y = D(c) \]
```python
def backward_pass(g, end_node, start_node):
    outgrads = defaultdict(list)
    outgrads[end_node] = [g]
    assert_vspace_match(outgrads[end_node][0], end_node.vspace, None)
    for node in toposort(end_node, start_node):
        if node not in outgrads: continue
        cur_outgrad = vsum(node.vspace, *outgrads[node])
        function, args, kwargs, parents = node.recipe
        for argnum, parent in parents:
            outgrad = function.vjp(argnum, cur_outgrad, node,
                                    parent.vspace, node.vspace, args, kwargs)
            outgrads[parent].append(outgrad)
            assert_vspace_match(outgrad, parent.vspace, function)
    return cur_outgrad
```
```python
def grad(fun, argnum=0):
    @attach_name_and_doc(fun, argnum, 'Gradient')
    def gradfun(*args, **kwargs):
        vjp, _ = make_vjp(fun, argnum)(*args, **kwargs)
        return vjp(1.0)
    return gradfun

def make_vjp(fun, argnum=0):
    def vjp(*args, **kwargs):
        start_node, end_node = forward_pass(fun, args, kwargs, argnum)
        if not isnode(end_node) or start_node not in end_node.progenitors:
            warnings.warn("Output seems independent of input.")
        return lambda g: start_node.vspace.zeros(), end_node
    return lambda g: backward_pass(g, end_node, start_node), end_node
return vjp
```
ingredients:

1. tracing composition of primitive functions
   Node, primitive, forward_pass

2. vector-Jacobian product for each primitive
   defvjp

3. composing VJPs backward
   backward_pass, make_vjp, grad
what’s the point? easy to extend!
- develop autograd!
- forward mode
- log joint densities from sampler programs