Automatic differentiation

Matthew J Johnson (mattjj@google.com)
Deep Learning Summer School
Montreal 2017

Dougal Maclaurin    David Duvenaud    Ryan P Adams
Our awesome new world
Our awesome new world

- TensorFlow, Stan, Theano, Edward, PyTorch, MinPy
- Only need to specify forward model
- Autodiff + optimization / inference done for you
Our awesome new world

- TensorFlow, Stan, Theano, Edward, PyTorch, MinPy
- Only need to specify forward model
- Autodiff + optimization / inference done for you

- loops? branching? recursion? closures? data structures?
Our awesome new world

- TensorFlow, Stan, Theano, Edward, PyTorch, MinPy
- Only need to specify forward model
- Autodiff + optimization / inference done for you

- loops? branching? recursion? closures? data structures?
- debugger?
Our awesome new world

- TensorFlow, Stan, Theano, Edward, PyTorch, MinPy
- Only need to specify forward model
- Autodiff + optimization / inference done for you

- loops? branching? recursion? closures? data structures?
- debugger?
- a second compiler/interpreter to satisfy
Our awesome new world

- TensorFlow, Stan, Theano, Edward, PyTorch, MinPy
- Only need to specify forward model
- Autodiff + optimization / inference done for you

- loops? branching? recursion? closures? data structures?
- debugger?
- a second compiler/interpreter to satisfy
- a new mini-language to learn
Autograd

- [github.com/hips/autograd](https://github.com/hips/autograd)
  - differentiates native Python code
  - handles most of Numpy + Scipy
  - loops, branching, recursion, closures
  - arrays, tuples, lists, dicts, classes, …
  - derivatives of derivatives
  - a one-function API
  - small and easy to extend
import autograd.numpy as np
import autograd.numpy.random as npr
from autograd import grad

def predict(weights, inputs):
    for W, b in weights:
        outputs = np.dot(inputs, W) + b
        inputs = np.tanh(outputs)
    return outputs

def init_params(scale, sizes):
    return [(npr.randn(m, n) * scale, npr.randn(n) * scale)
            for m, n in zip(sizes[:-1], sizes[1:])]

def logprob_func(weights, inputs, targets):
    preds = predict(weights, inputs)
    return np.sum((preds - targets)**2)

gradient_func = grad(logprob_func)
import autograd.numpy as np
from autograd import grad
import matplotlib.pyplot as plt

x = np.linspace(-7, 7, 200)
plt.plot(x, np.tanh(x),
         x, grad(np.tanh)(x),
         x, grad(grad(np.tanh))(x),
         x, grad(grad(grad(np.tanh)))(x),
         x, grad(grad(grad(grad(np.tanh))))(x),
         x, grad(grad(grad(grad(grad(np.tanh)))))(x))
from autograd import grad, jacobian

def hessian(fun, argnum=0):
    return jacobian(jacobian(fun, argnum), argnum)

def hvp(fun):
    def grad_dot_vector(arg, vector):
        return np.dot(grad(fun)(arg), vector)
    return grad(grad_dot_vector)

\[ \nabla^2 f(x) \cdot v = \nabla_x (\nabla_x f(x) \cdot v) \n\]
Black-box inference in a tweet

Ryan Adams @ryan_p_adams · 7 Nov 2015
@DavidDuvenaud
def elbo(p, lp, D, N):
    v=exp(p[D:])
    s=randn(N,D)*sqrt(v)+p[:D]
    return mvn.entropy(0, diag(v))+mean(lp(s))
gf = grad(elbo)
1. Jacobians and the chain rule
   • Forward and reverse accumulation
2. Autograd’s implementation
   • Fully closed tracing autodiff in Python
3. Advanced autodiff techniques
   • Checkpointing, forward from reverse, differentiating optima and fixed points
Tutorial goals

1. Jacobians and the chain rule
   - Forward and reverse accumulation

2. Autograd’s implementation
   - Fully closed tracing autodiff in Python

3. Advanced autodiff techniques
   - Checkpointing, forward from reverse, differentiating optima and fixed points
\[ F : \mathbb{R}^n \to \mathbb{R} \]
$F : \mathbb{R}^n \rightarrow \mathbb{R}$

$F : \begin{array}{c}
\mathbf{x} \\
\in \mathbb{R}^n
\end{array} \quad \rightarrow \quad \begin{array}{c}
y \\
\in \mathbb{R}
\end{array}$
\[ F : \mathbb{R}^n \to \mathbb{R} \]

\[ F = D \circ C \circ B \circ A \]
\[ F : \mathbb{R}^n \rightarrow \mathbb{R} \]

\[ F : \begin{array}{c} x \in \mathbb{R}^n \\ y \in \mathbb{R} \end{array} \]

\[ F = D \circ C \circ B \circ A \]

\[ y = F(x) = D(C(B(A(x)))) \]
\( F : \mathbb{R}^n \to \mathbb{R} \)

\[
F = D \circ C \circ B \circ A \quad \quad \quad y = F(\mathbf{x}) = D(C(B(A(\mathbf{x}))))
\]

\[
y = D(c), \quad c = C(b), \quad b = B(a), \quad a = A(\mathbf{x})
\]
\[ y = D(c), \quad c = C(b), \quad b = B(a), \quad a = A(x) \]
\[ y = D(c), \quad c = C(b), \quad b = B(a), \quad a = A(x) \]

\[ F'(x) = \frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix} \]
\[ y = D(c), \quad c = C(b), \quad b = B(a), \quad a = A(x) \]

\[ F'(x) = \frac{\partial y}{\partial x} = \left[ \frac{\partial y}{\partial x_1} \cdots \frac{\partial y}{\partial x_n} \right] \]

\[ F'(x) = \frac{\partial y}{\partial c} \quad \frac{\partial c}{\partial b} \quad \frac{\partial b}{\partial a} \quad \frac{\partial a}{\partial x} \]
\[ y = D(c), \quad c = C(b), \quad b = B(a), \quad a = A(x) \]

\[ F'(x) = \frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix} \]

\[ F'(x) = \frac{\partial y}{\partial c} \quad \frac{\partial c}{\partial b} \quad \frac{\partial b}{\partial a} \quad \frac{\partial a}{\partial x} \]

\[ \frac{\partial y}{\partial c} = D'(c) \]
\[ y = D(c), \quad c = C(b), \quad b = B(a), \quad a = A(x) \]

\[ F'(x) = \frac{\partial y}{\partial x} = \left[ \frac{\partial y}{\partial x_1} \quad \ldots \quad \frac{\partial y}{\partial x_n} \right] \]

\[ F'(x) = \frac{\partial y}{\partial c} \quad \frac{\partial c}{\partial b} \quad \frac{\partial b}{\partial a} \quad \frac{\partial a}{\partial x} \]

\[ \frac{\partial y}{\partial c} = D'(c) \quad \frac{\partial c}{\partial b} = C'(b) \]
\[ \frac{\partial y}{\partial c} = D'(c), \quad \frac{\partial c}{\partial b} = C'(b), \quad \frac{\partial b}{\partial a} = B'(a), \quad \frac{\partial a}{\partial x} = A'(x) \]
\[ F'(x) = \frac{\partial y}{\partial c} \left( \frac{\partial c}{\partial b} \left( \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \right) \right) \]
\[ F'(x) = \frac{\partial y}{\partial c} \left( \frac{\partial c}{\partial b} \left( \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \right) \right) \]

\[ \frac{\partial b}{\partial x} = \begin{bmatrix} \frac{\partial b_1}{\partial x_1} & \ldots & \frac{\partial b_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial b_m}{\partial x_1} & \ldots & \frac{\partial b_m}{\partial x_n} \end{bmatrix} \]
\[ F'(x) = \frac{\partial y}{\partial c} \left( \frac{\partial c}{\partial b} \left( \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \right) \right) \]

\[
\frac{\partial b}{\partial x} = \begin{bmatrix}
\frac{\partial b_1}{\partial x_1} & \cdots & \frac{\partial b_1}{\partial x_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial b_m}{\partial x_1} & \cdots & \frac{\partial b_m}{\partial x_n}
\end{bmatrix}
\]

Forward accumulation
\[ F'(x) = \frac{\partial y}{\partial c} \left( \frac{\partial c}{\partial b} \left( \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \right) \right) \]

\[ \frac{\partial b}{\partial x} = \begin{bmatrix} \frac{\partial b_1}{\partial x_1} & \cdots & \frac{\partial b_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial b_m}{\partial x_1} & \cdots & \frac{\partial b_m}{\partial x_n} \end{bmatrix} \]

Forward accumulation

\[ F'(x) = \left( \left( \frac{\partial y}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \right) \frac{\partial a}{\partial x} \right) \]

\[ F'(x) = \frac{\partial y}{\partial c} \left( \frac{\partial c}{\partial b} \left( \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \right) \right) \]

\[ \frac{\partial b}{\partial x} = \begin{bmatrix} \frac{\partial b_1}{\partial x_1} & \cdots & \frac{\partial b_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial b_m}{\partial x_1} & \cdots & \frac{\partial b_m}{\partial x_n} \end{bmatrix} \]

Forward accumulation

\[ F'(x) = \left( \left( \frac{\partial y}{\partial c} \frac{\partial c}{\partial b} \right) \frac{\partial b}{\partial a} \right) \frac{\partial a}{\partial x} \]

\[ \frac{\partial y}{\partial b} = \begin{bmatrix} \frac{\partial y}{\partial b_1} & \cdots & \frac{\partial y}{\partial b_m} \end{bmatrix} \]
\[ F'(x) = \frac{\partial y}{\partial c} \left( \frac{\partial c}{\partial b} \left( \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \right) \right) \]

\[ \frac{\partial b}{\partial x} = \begin{bmatrix} \frac{\partial b_1}{\partial x_1} & \cdots & \frac{\partial b_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial b_m}{\partial x_1} & \cdots & \frac{\partial b_m}{\partial x_n} \end{bmatrix} \]

Forward accumulation

\[ F'(x) = \left( \left( \frac{\partial y}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \right) \frac{\partial a}{\partial x} \right) \]

\[ \frac{\partial y}{\partial b} = \begin{bmatrix} \frac{\partial y}{\partial b_1} & \cdots & \frac{\partial y}{\partial b_m} \end{bmatrix} \]

Reverse accumulation
\[ F'(x) \ v = \frac{\partial y}{\partial c} \ \frac{\partial c}{\partial b} \ \frac{\partial b}{\partial a} \ \frac{\partial a}{\partial x} \ v \]
\[ F'(x) \, v = \frac{\partial y}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \, v \]

\[ F'(x) \, v = \frac{\partial y}{\partial c} \left( \frac{\partial c}{\partial b} \left( \frac{\partial b}{\partial a} \left( \frac{\partial a}{\partial x} \, v \right) \right) \right) \]
\[
F'(x) v = \begin{bmatrix}
\frac{\partial y}{\partial c} & \frac{\partial y}{\partial b} & \frac{\partial y}{\partial a}
\end{bmatrix}
\begin{bmatrix}
\frac{\partial c}{\partial x} & \frac{\partial b}{\partial x} & \frac{\partial a}{\partial x}
\end{bmatrix}
v
\]

\[
F'(x) v = \frac{\partial y}{\partial c} \left( \frac{\partial c}{\partial b} \left( \frac{\partial b}{\partial a} \left( \frac{\partial a}{\partial x} v \right) \right) \right)
\]

Forward accumulation \iff Jacobian-vector products
Build Jacobian one column at a time
\[ F'(x) \ v = \ \frac{\partial y}{\partial c} \ \frac{\partial c}{\partial b} \ \frac{\partial b}{\partial a} \ \frac{\partial a}{\partial x} \ v \]

\[ F'(x) \ v = \ \frac{\partial y}{\partial c} \left( \frac{\partial c}{\partial b} \left( \frac{\partial b}{\partial a} \left( \frac{\partial a}{\partial x} \ v \right) \right) \right) \]

Forward accumulation \ \leftrightarrow \ \text{Jacobian-vector products}

Build Jacobian one column at a time

\[ F'(x) = \ \frac{\partial y}{\partial c} \left( \frac{\partial c}{\partial b} \left( \frac{\partial b}{\partial a} \left( \frac{\partial a}{\partial x} \ \frac{\partial x}{\partial x} \right) \right) \right) \]
\[ \nu^T F'(x) = \nu^T \frac{\partial y}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \]
\[ v^T F'(x) = v^T \frac{\partial y}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \]

\[ v^T F'(x) = \left( \left( \left( v^T \frac{\partial y}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \right) \right) \right) \]
\[ v^T F'(x) = v^T \frac{\partial y}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \]

\[ v^T F'(x) = \left( \left( \left( \left( v^T \frac{\partial y}{\partial c} \right) \frac{\partial c}{\partial b} \right) \frac{\partial b}{\partial a} \right) \frac{\partial a}{\partial x} \right) \]

Reverse accumulation \iff vector-Jacobian products

Build Jacobian one row at a time
\[ v^T F'(x) = v^T \frac{\partial y}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \]

\[ v^T F'(x) = \left( \left( \left( v^T \frac{\partial y}{\partial c} \right) \frac{\partial c}{\partial b} \right) \frac{\partial b}{\partial a} \right) \frac{\partial a}{\partial x} \]

Reverse accumulation $\iff$ vector-Jacobian products

Build Jacobian one row at a time

\[ F'(x) = \left( \left( \left( \frac{\partial y}{\partial y} \frac{\partial y}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \right) \right) \right) \]
Forward and reverse accumulation

- **Forward accumulation**
  - Jacobian-vector products
  - "push-forward"
  - build Jacobian matrix one column at a time

- **Reverse accumulation**
  - vector-Jacobian products
  - "pull-back"
  - build Jacobian matrix one row at a time
Non-chain composition
Non-chain composition

\[ y = F(x_1, x_2) \]
Non-chain composition

Fan-in

\[ y = F(x_1, x_2) \]

\[ \frac{\partial y}{\partial x_1} = F_1'(x_1, x_2) \]

\[ \frac{\partial y}{\partial x_2} = F_2'(x_1, x_2) \]
Non-chain composition

Fan-in

\[ y = F(x_1, x_2) \]

\[ \frac{\partial y}{\partial x_1} = F'(x_1, x_2) \]

\[ \frac{\partial y}{\partial x_2} = F'_2(x_1, x_2) \]

Fan-out

\[ G(x) = \begin{bmatrix} x \\ x \end{bmatrix} = \begin{bmatrix} I \\ I \end{bmatrix} x \]
Non-chain composition

**Fan-in**

\[
y = F(x_1, x_2)
\]

\[
\frac{\partial y}{\partial x_1} = F'(x_1, x_2)
\]

\[
\frac{\partial y}{\partial x_2} = F'_2(x_1, x_2)
\]

**Fan-out**

\[
G(x) = \begin{bmatrix} x \\ x \end{bmatrix} = \begin{bmatrix} I \\ I \end{bmatrix} x
\]

\[
G'(x) = \begin{bmatrix} I \\ I \end{bmatrix} \\
\begin{bmatrix} v_1^T \\ v_2^T \end{bmatrix} = \begin{bmatrix} I \\ I \end{bmatrix} = v_1^T + v_2^T
\]
1. Jacobians and the chain rule
   • Forward and reverse accumulation
2. Autograd’s implementation
   • Fully closed tracing autodiff in Python
3. Advanced autodiff techniques
   • Checkpointing, forward from reverse, differentiating optima and fixed points
1. Read and generate source code ahead-of-time
   - source and target language could be Python
   - or a “computation graph” language (TensorFlow)

2. Monitor function execution at runtime
1. Read and generate source code ahead-of-time
   - source and target language could be Python
   - or a “computation graph” language (TensorFlow)

2. **Monitor function execution at runtime**
1. Tracing the composition of primitive functions
2. Defining a vector-Jacobian product (VJP) operator for each primitive
3. Composing VJPs backward
Autograd’s ingredients

1. Tracing the composition of primitive functions
2. Defining a vector-Jacobian product (VJP) operator for each primitive
3. Composing VJPs backward
numpy.sum
primitive

autograd.np.sum

np.sum
Node $a$

- Value: $a$
- Function: $F$
- Parents: $[x]$

```
primitive

autograd.numpy.sum

numpy.sum
```
Node \(\mathbf{a}\)

- **value:** \(\mathbf{a}\)
- **function:** \(F\)
- **parents:** \([x]\)

Diagram:

- **primitive**
  - **autograd.numpy.sum**
    - **unbox**
      - **a**
        - **numpy.sum**
Node $\tilde{a}$

- **value:** $a$
- **function:** $F$
- **parents:** $[\tilde{x}]$

Node $\tilde{b}$

- **value:** $b$
- **function:** `anp.sum`
- **parents:** $[\tilde{a}]$

**primitive**

- **value:** $\text{autograd.numpy.sum}$
- **function:**
- **parents:** $[\tilde{a}, \tilde{x}]$

**Nodes:**

- **unbox**
- **box**
- **numpy.sum**
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)
```python
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),
```
class primitive(object):
    def __call__(self, *args, **kwargs):
        argvals = list(args)
        progenitors = set()
        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))
                progenitors.update(arg.progenitors & active_progenitors)

        result_value = self.fun(*argvals, **kwargs)
        return new_node(result_value, (self, args, kwargs, parents), progenitors)
class Node:
__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:
    def __call__(self, *args, **kwargs):
        argvals = list(args)
        progenitors = set()
        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))
            progenitors.update(arg.progenitors & active_progenitors)
        result_value = self.fun(*argvals, **kwargs)
        return new_node(result_value, (self, args, kwargs, parents), progenitors)

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

\[ x \]
\( a = A(x) \)
\[ a = A(x) \]
\[ b = B(a) \]
\[ b = B(a) \]
\[ c = C(b) \]
\[ a = A(x) \]
\[ x \]
\( b = B(a) \)
No control flow!
Autograd’s ingredients

1. Tracing the composition of primitive functions
2. Defining a vector-Jacobian product (VJP) operator for each primitive
3. Composing VJPs backward
$x \rightarrow a = A(x)$
\[
\frac{\partial y}{\partial a} = a = A(x)
\]
\[ \frac{\partial y}{\partial x} = ? \]

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

\[
\begin{array}{c}
\circ \quad x \\
\circ \quad a = A(x)
\end{array}
\]
\[ \frac{\partial y}{\partial x} = \frac{\partial y}{\partial a} \cdot A'(x) \]
\[
\frac{\partial y}{\partial x} = \frac{\partial y}{\partial a} \cdot A'(x) \quad \frac{\partial y}{\partial a}
\]
```python
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 = self.vspace(value)

class primitive(object):
    def __call__(self, *args, **kwargs):
        argvals = list(args)
        progenitors = set()
        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))
                progenitors.update(arg.progenitors & active_progenitors)

        result_value = self.fun(*argvals, **kwargs)
        return new_node(result_value, (self, args, kwargs, parents), progenitors)

    def forward_pass(self, 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

anp.sinh.defvjp(lambda g, ans, vs, gvs, x: g * anp.cosh(x))
anp.cosh.defvjp(lambda g, ans, vs, gvs, x: g * anp.sinh(x))
anp.tanh.defvjp(lambda g, ans, vs, gvs, x: g / anp.cosh(x)**2)
anp.cross.defvjp(lambda g, ans, vs, gvs, a, b, axisa=-1, axisb=-1, axisc=-1, axis=None:
    anp.cross(b, g, axisb, axisc, axisa, axis), argnum=0)

def grad_sort(g, ans, vs, gvs, x, axis=-1, kind='quicksort', order=None):
    sort_perm = anp.argsort(x, axis, kind, order)
    return unpermuter(g, sort_perm)
anp.sort.defvjp(grad_sort)
```
1. Tracing the composition of primitive functions
2. Defining a vector-Jacobian product (VJP) operator for each primitive
3. Composing VJPs backward
\[ a = A(x) \]
\[ b = B(a) \]
\[ c = C(b) \]
\[ y = D(c) \]
\( a = A(x) \)

\( b = B(a) \)

\( c = C(b) \)

\( \frac{\partial y}{\partial y} = 1 \)

\( y = D(c) \)
\[ x = A(x) \]
\[ b = B(a) \]
\[ c = C(b) \]
\[ \frac{\partial y}{\partial y} = 1 \]
$a = A(x)$

$b = B(a)$

$c = C(b)$

$\frac{\partial y}{\partial b}$

$\frac{\partial y}{\partial c}$

$\frac{\partial y}{\partial y} = 1$

$y = D(c)$
\[ a = A(x) \]
\[ b = B(a) \]
\[ c = C(b) \]
\[ y = D(c) \]
\[ \frac{\partial y}{\partial b} \]
\[ \frac{\partial y}{\partial c} \]
\[ \frac{\partial y}{\partial y} = 1 \]
\[ \frac{\partial y}{\partial x} = \frac{\partial y}{\partial a} b = B(a) \]

\[ \frac{\partial y}{\partial b} c = C(b) \]

\[ \frac{\partial y}{\partial c} = 1 \]

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

\[ \frac{\partial y}{\partial x} = 1 \]

\[ \frac{\partial y}{\partial c} \]

\[ \text{start_node} \quad \quad \quad \text{end_node} \]

\[ x \quad \quad \quad y = D(c) \]
\[
\begin{align*}
\frac{\partial y}{\partial b} & \quad \frac{\partial y}{\partial c} \\
\frac{\partial y}{\partial y} & = 1 \\
\end{align*}
\]

\[
\begin{align*}
\text{start_node} & \quad x \\
a & = A(x) \\
b & = B(a) \\
c & = C(b) \\
y & = D(c) \\
\text{end_node} & \\
\end{align*}
\]
\[ \frac{\partial y}{\partial a} \] 

\[ \frac{\partial y}{\partial b} \] 

\[ \frac{\partial y}{\partial c} \] 

\[ \frac{\partial y}{\partial y} = 1 \] 

\[ b = B(a) \] 

\[ c = C(b) \] 

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

\[ x = A(x) \quad b = B(a) \quad c = C(b) \quad y = D(c) \]
\[ \frac{\partial y}{\partial y} = 1 \]

\[
\begin{align*}
    y &= D(c) \\
    c &= C(b) \\
    b &= B(a) \\
    a &= A(x)
\end{align*}
\]

\[\text{start_node} \quad x \quad \text{end_node}\]
def backward_pass(g, end_node, start_node):
    outgrads = {end_node : (g, False)}
    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 = outgrads.pop(node)
        function, args, kwargs, parents = node.recipe
        for argnum, parent in parents:
            outgrad = function.vjp(argnum, cur_outgrad[0], node,
                                    parent.vspace, node.vspace, args, kwargs)
            assert_vspace_match(outgrad, parent.vspace, function)
            outgrads[parent] = add_outgrads(parent.vspace, outgrads.get(parent),
                                             outgrad)

    return cur_outgrad[0]
def grad(fun, argnum=0):
    def gradfun(*args,**kwargs):
        args = list(args)
        args[argnum] = safe_type(args[argnum])
        vjp, ans = make_vjp(fun, argnum)(*args, **kwargs)
        return vjp(vspace(getval(ans)).ones())
    return gradfun

make_vjp = grad

def make_vjp(fun, argnum=0):
    def vjp_maker(*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.")
            def vjp(g): return start_node.vspace.zeros()
        else:
            def vjp(g): return backward_pass(g, end_node, start_node)
        return vjp, end_node
    return vjp_maker
Autograd’s ingredients

1. Tracing the composition of primitive functions
   Node, primitive, forward_pass

2. Defining a vector-Jacobian product (VJP) operator for each primitive
   defvjp

3. Composing VJPs backward
   backward_pass, make_vjp, grad
Tradeoffs in forward vs reverse
Tradeoffs in forward vs reverse

- Reverse-mode requires tracing a program’s execution
  - Memory cost scales like depth of program
  - Checkpointing can trade off time and memory
Tradeoffs in forward vs reverse

• Reverse-mode requires tracing a program’s execution
  • Memory cost scales like depth of program
  • Checkpointing can trade off time and memory

• Forward-mode evaluates a JVP with constant memory overhead
  • But requires $n$ calls to form Jacobian of $F : \mathbb{R}^n \rightarrow \mathbb{R}$
  • Autograd forward-mode by @j-towns: github.com/BB-UCL/autograd-forward
Tradeoffs in forward vs reverse

- Reverse-mode requires tracing a program’s execution
  - Memory cost scales like depth of program
  - Checkpointing can trade off time and memory
- Forward-mode evaluates a JVP with constant memory overhead
  - But requires $n$ calls to form Jacobian of $F : \mathbb{R}^n \rightarrow \mathbb{R}$
  - Autograd forward-mode by @j-towns: github.com/BB-UCL/autograd-forward
- Can use both together (in autograd!) for mixed-mode
Tutorial goals

1. Jacobians and the chain rule
   - Forward and reverse accumulation

2. Autograd’s implementation
   - Fully closed tracing autodiff in Python

3. Advanced autodiff techniques
   - Checkpointing, forward from reverse, differentiating optima and fixed points
Checkpointing
Checkpointing
Checkpointing
Checkpointing
Checkpointing
Checkpointing
Checkpointing
Checkpointing
Checkpointing
Checkpointing
Checkpointing

\[ \frac{\partial y}{\partial y} = 1 \]
Checkpointing

\[ \frac{\partial y}{\partial y} = 1 \]
Checkpointing
Checkpointing

\[ \frac{\partial y}{\partial c} \]

\[ \frac{\partial y}{\partial y} = 1 \]
Checkpointing
Checkpointing

\[ \frac{\partial y}{\partial c} \]
Checkpointing
Checkpointing
def checkpoint(fun):
    """Returns a checkpointed version of ‘fun’, where intermediate values computed during the forward pass of ‘fun’ are discarded and then recomputed for the backward pass. Useful to trade off time and memory."""
    def wrapped_grad(argnum, g, ans, vs, gvs, args, kwargs):
        return make_vjp(fun, argnum)(*args, **kwargs)[0](g)
    wrapped = primitive(fun)
    wrapped.vjp = wrapped_grad
    return wrapped
Getting forward from reverse

```python
def make_jvp(fun, argnum=0):
    def jvp_maker(*args, **kwargs):
        vjp, y = make_vjp(fun, argnum)(*args, **kwargs)
        vjp_vjp, _ = make_vjp(vjp)(vspace(getval(y)).zeros())  # dummy vals
        return vjp_vjp  # vjp_vjp is just jvp by linearity
    return jvp_maker
```
Getting forward from reverse

```python
def make_jvp(fun, argnum=0):
    def jvp_maker(*args, **kwargs):
        vjp, y = make_vjp(fun, argnum)(*args, **kwargs)
        vjp_vjp, _ = make_vjp(vjp)(vspace(getval(y)).zeros())  # dummy vals
        return vjp_vjp  # vjp_vjp is just jvp by linearity
    return jvp_maker
```
Getting forward from reverse

```python
def make_jvp(fun, argnum=0):
    def jvp_maker(*args, **kwargs):
        vjp, y = make_vjp(fun, argnum)(*args, **kwargs)
        vjp_vjp, _ = make_vjp(vjp)(vspace(getval(y)).zeros())  # dummy vals
        return vjp_vjp  # vjp_vjp is just jvp by linearity
    return jvp_maker
```

$$J^T v$$
Getting forward from reverse

```python
def make_jvp(fun, argnum=0):
    def jvp_maker(*args, **kwargs):
        vjp, y = make_vjp(fun, argnum)(*args, **kwargs)
        vjp_vjp, _ = make_vjp(vjp)(vspace(getval(y)).zeros())  # dummy vals
        return vjp_vjp  # vjp_vjp is just jvp by linearity
    return jvp_maker
```
import tensorflow as tf

def fwd_gradients(ys, xs, d_xs):
    v = tf.placeholder(ys.dtype, shape=ys.get_shape())  # dummy variable
    g = tf.gradients(ys, xs, grad_ys=v)
    return tf.gradients(g, v, grad_ys=d_xs)
Solutions, optima, and fixed points
Solutions, optima, and fixed points

\[ x^*(a) = \underset{x}{\arg \min} \ f(a, x) \]

\[ \nabla x^*(a) = ? \]
Solutions, optima, and fixed points

\[ x^*(a) = \arg \min_x f(a, x) \]

\[ \nabla x^*(a) = ? \]

solve \( g(a, x) = 0 \) for \( x \)

\[ g(a, x^*(a)) = 0 \]

\[ \nabla x^*(a) = ? \]
The implicit function theorem

\[ g(a, x^*(a)) = 0 \]
The implicit function theorem

\[ g(a, x^*(a)) = 0 \]

\[ \nabla_a g(a, x^*) + \nabla x^*(a) \nabla_x g(a, x^*) = 0 \]
The implicit function theorem

\[ g(a, x^*(a)) = 0 \]

\[ \nabla_a g(a, x^*) + \nabla x^*(a) \nabla_x g(a, x^*) = 0 \]

\[ \nabla x^*(a) = -\nabla_a g(a, x^*) \nabla_x g(a, x^*)^{-1} \]
The implicit function theorem

\[ g(a, x^*(a)) = 0 \]

\[ \nabla_a g(a, x^*) + \nabla x^*(a) \nabla_x g(a, x^*) = 0 \]

\[ \nabla x^*(a) = -\nabla_a g(a, x^*) \nabla_x g(a, x^*)^{-1} \]

differentiate solutions / optima \quad \leftrightarrow \quad \text{solve linearized systems}
The implicit function theorem

\[ g(a, x^*(a)) = 0 \]

\[ \nabla_a g(a, x^*) + \nabla x^*(a) \nabla_x g(a, x^*) = 0 \]

\[ \nabla x^*(a) = -\nabla_a g(a, x^*) \nabla_x g(a, x^*)^{-1} \]

Differentiate solutions / optima \iff\ solve linearized systems

Automatically generate a linear solver from the forward solver?
Differentiating fixed points
Differentiating fixed points

\[ x^*(a) \] solves \[ x = f(a, x) \] for \[ x \]
Differentiating fixed points

\[ x^*(a) \] solves \( x = f(a, x) \) for \( x \)

from autograd import primitive
from functools import partial

@primitive
def fixed_point(f, a, init, converged, max_iter):
    update = partial(f, a)
    current, prev = update(init), init
    for _ in xrange(max_iter):
        if converged(current, prev): break
        current, prev = update(current), current
    else:
        print 'fixed point iteration limit reached'
    return current
Differentiating fixed points

\[ a \]

\[ x_{\text{init}} \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \cdots \rightarrow x_{n-2} \rightarrow x_{n-1} \rightarrow x_n \]
Differentiating fixed points

\[ x_{\text{init}} \to x_1 \to x_2 \to x_3 \to \cdots \to x_{n-2} \to x_{n-1} \to x_n \]

\[ a \]

\[ n \to \infty \]

\[ x^* = x_n = x_{n-1} = x_{n-2} = \cdots \]
Differentiating fixed points

```
from autograd import primitive, make_vjp, make_tuple
from autograd.util import flatten

def grad_fixed_point(g_fp, fp, vs, gvs, f, a, init, converged, max_iter):
    vjp, _ = make_vjp(lambda args: f(*args))(make_tuple(a, fp))
    g_a_flat, unflatten = flatten(vs.zeros())
    for _ in xrange(max_iter):
        if normsq(flatten(g)[0]) < 1e-6: break
        term, g = vjp(g)
        g_a_flat = g_a_flat + flatten(term)[0]
    else:
        print 'backward fixed point iteration limit reached'
    return unflatten(g_a_flat)
```
Differentiating fixed points

- Inherits structure from forward iteration
  - Forward is Newton $\Rightarrow$ reverse requires only one step
  - Forward is block coordinate descent $\Rightarrow$ reverse is block Gauss-Seidel
- May be preferable to decouple forward and reverse
  - Then choose any linear solver for implicit linearized system
  - Can reuse dual variables from forward solver
def make_hvp(fun, argnum=0):
    """Builds a function for evaluating the Hessian-vector product at a point, which may be useful when evaluating many Hessian-vector products at the same point while caching the results of the forward pass."""
    def hvp_maker(*args, **kwargs):
        return make_vjp(grad(fun, argnum), argnum)(*args, **kwargs)[0]
    return hvp_maker

def make_ggnvp(f, g=lambda x: 1./2*np.sum(x**2, axis=-1), f_argnum=0):
    """Builds a function for evaluating generalized-Gauss-Newton-vector products at a point. Slightly more expensive than mixed-mode."""
    def ggnvp_maker(*args, **kwargs):
        f_vjp, f_x = make_vjp(f, f_argnum)(*args, **kwargs)
        g_hvp, grad_g_x = make_vjp(grad(g))(f_x)
        f_vjp_vjp, _ = make_vjp(f_vjp)(vspace(getval(grad_g_x)).zeros())
        def ggnvp(v): return f_vjp(g_hvp(f_vjp_vjp(v)))
        return ggnvp
    return ggnvp_maker
Thanks!

• github.com/hips/autograd