CS330 Automatic differentiation

Matt Johnson, mattjj@google.com, Sept 28 2020
Why study AD?
Why study AD?

1. You use it *every day!*
Why study AD?

1. You use it every day!

2. **DL revolution** = data + compute + architectures + AD
Why study **AD**?

1. You use it *every day*!

2. **DL revolution** = data + compute + architectures + **AD**

3. **Research frontiers** use advanced **AD**
   - iMAML, Neural ODEs, DEQs, OptNet, RevNets, Reformer, new optimizers…
Why study AD?

1. You use it every day!

2. DL revolution = data + compute + architectures + AD

3. Research frontiers use advanced AD
   - iMAML, Neural ODEs, DEQs, OptNet, RevNets, Reformer, new optimizers…

4. AD is interesting in its own right
What is autodiff?

Demo!
This lecture

- **Fundamentals**
  - Forward- and reverse-modes
  - Some sense for how they’re implemented

- **Advanced techniques**
  - Efficient AD of equation solvers, fixed-points, optimizers
  - Efficient AD of long-running iterative processes
  - Very high-order AD via jet
Math and notation

\[ f : \mathbb{R}^n \rightarrow \mathbb{R}^m \]
Math and notation

\[ f : \mathbb{R}^n \rightarrow \mathbb{R}^m \]

\[ \partial f : \mathbb{R}^n \rightarrow (\mathbb{R}^n \rightarrow \mathbb{R}^m) \]
Math and notation

\[ f : \mathbb{R}^n \rightarrow \mathbb{R}^m \]

\[ \partial f : \mathbb{R}^n \rightarrow (\mathbb{R}^n \rightarrow \mathbb{R}^m) \]

\[ \partial f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m \]
Math and notation

\[ f : \mathbb{R}^n \rightarrow \mathbb{R}^m \]

\[ \partial f : \mathbb{R}^n \rightarrow (\mathbb{R}^n \rightarrow \mathbb{R}^m) \]

\[ \partial f(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m \]

\[ \partial f(x) \in \mathbb{R}^{m \times n} \]
Math and notation

\[ f : \mathbb{R}^n \rightarrow \mathbb{R}^m \]

\[ f(x + v) = f(x) + \partial f(x)[v] + \mathcal{O}(\|v\|^2) \]

\[ v \mapsto f(x) + \partial f(x)[v] \]
Math and notation: comparison

\frac{\partial y}{\partial x}
Math and notation: comparison

\[ y = f(x) \]

\[ \frac{\partial y}{\partial x} \]
Math and notation: comparison

\[ y = f(x) \]

\[ \frac{\partial y}{\partial x} = \partial f \]
Math and notation: comparison

\[ y = f(x) \]

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

\[ \left. \frac{\partial y}{\partial x} \right|_{x=a} = \partial f(a) \]
\( \nabla f(x) \) is the vector such that

\[
\langle \nabla f(x), \ v \rangle = \partial f(x)[v]
\]
Math and notation: chain rule

\[ f = g \circ h \quad f : \mathbb{R}^n \rightarrow \mathbb{R}^m \]
Math and notation: chain rule

\[ f = g \circ h \]

\[ f : \mathbb{R}^n \rightarrow \mathbb{R}^m \]

\[ h : \mathbb{R}^n \rightarrow \mathbb{R}^p \]

\[ g : \mathbb{R}^p \rightarrow \mathbb{R}^m \]
Math and notation: chain rule

\[ f = g \circ h \quad f : \mathbb{R}^n \to \mathbb{R}^m \]
\[ h : \mathbb{R}^n \to \mathbb{R}^p \quad g : \mathbb{R}^p \to \mathbb{R}^m \]
\[ \partial f(x) = \partial g(h(x)) \circ \partial h(x) \]
Math and notation: chain rule

\[ f = g \circ h \quad f : \mathbb{R}^n \rightarrow \mathbb{R}^m \]

\[ h : \mathbb{R}^n \rightarrow \mathbb{R}^p \quad g : \mathbb{R}^p \rightarrow \mathbb{R}^m \]

\[ \partial f(x) = \partial g(h(x)) \circ \partial h(x) \]

\[ \partial f(x)[v] = \partial g(h(x))[\partial h(x)[v]] \]
Math and notation: chain rule

\[ f = g \circ h \]

\[ f : \mathbb{R}^n \rightarrow \mathbb{R}^m \]

\[ h : \mathbb{R}^n \rightarrow \mathbb{R}^p \]

\[ g : \mathbb{R}^p \rightarrow \mathbb{R}^m \]

\[ \frac{\partial y}{\partial x} \bigg|_{x=a} = \frac{\partial y}{\partial z} \bigg|_{z=h(a)} \frac{\partial z}{\partial x} \bigg|_{x=a} \]

\[ z = h(x) \]

\[ y = g(z) \]
Do we need a DAG rule?
Do we need a DAG rule?

\[ f(x) = (x, x) \]
Do we need a DAG rule?

\[ f(x) = (x, x) \]

\[ f(x) = \begin{bmatrix} I \\ I \end{bmatrix} x \]
The two linear maps of AD

1. $v \mapsto \partial f(x)v$ or $v \mapsto \partial f(x)[v]$

JVP / push-forward / forward-mode

build Jacobian one column at a time
The two linear maps of AD

1. \( v \mapsto \partial f(x)v \) or \( v \mapsto \partial f(x)[v] \)
   
   **JVP** / push-forward / forward-mode
   
   build Jacobian one **column** at a time

2. \( w^T \mapsto w^T \partial f(x) \) or \( w^T \mapsto \partial f(x)^T[w^T] \)
   
   **VJP** / pull-back / reverse-mode
   
   build Jacobian one **row** at a time
The two linear maps of AD

1. $v \mapsto \partial f(x)v$  or  $v \mapsto \partial f(x)[v]$
The two linear maps of AD

1. \( v \mapsto \partial f(x)v \) or \( v \mapsto \partial f(x)[v] \)

Let’s say you have

\[ v = \frac{\partial x}{\partial \theta} \]

and you want to compute

\[ w = \frac{\partial y}{\partial \theta} \quad \text{where} \quad y = f(x) \]
The two linear maps of AD

1. \( v \mapsto \partial f(x)v \) or \( v \mapsto \partial f(x)[v] \)

Let’s say you have

\[
v = \frac{\partial x}{\partial \theta}
\]

and you want to compute

\[
w = \frac{\partial y}{\partial \theta} \quad \text{where} \quad y = f(x)
\]
The two linear maps of AD

1. $v \mapsto \partial f(x)v$ or $v \mapsto \partial f(x)[v]$

Let’s say you have

$$v = \frac{\partial x}{\partial \theta}$$

and you want to compute

$$w = \frac{\partial y}{\partial \theta} \quad \text{where} \quad y = f(x)$$

$$w = \partial f(x)[v] = \frac{\partial y}{\partial x} \frac{\partial x}{\partial \theta}.$$

So $\partial f(x)$ “pushes forward” perturbation information.
The two linear maps of AD

2. \( w^T \mapsto w^T \partial f(x) \) or \( w^T \mapsto \partial f(x)^T [w^T] \)
The two linear maps of AD

2. $w^T \mapsto w^T \partial f(x)$ or $w^T \mapsto \partial f(x)^T [w^T]$

Let’s say you have

$$w^T = \frac{\partial l}{\partial y}$$

and you want to compute

$$u^T = \frac{\partial l}{\partial x} \quad \text{where} \quad y = f(x)$$
The two linear maps of AD

2. \( w^T \mapsto w^T \partial f(x) \) or \( w^T \mapsto \partial f(x)^T [w^T] \)

Let’s say you have

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

and you want to compute

\[
u^T = \frac{\partial \ell}{\partial x} \quad \text{where} \quad y = f(x)
\]
The two linear maps of AD

2. \( w^T \mapsto w^T \partial f(x) \) or \( w^T \mapsto \partial f(x)^T [w^T] \)

Let’s say you have

\[
w^T = \frac{\partial l}{\partial y}
\]

and you want to compute

\[
u^T = \frac{\partial l}{\partial x} \quad \text{where} \quad y = f(x)
\]

\[
u^T = \partial f(x)^T [w^T] = \frac{\partial l}{\partial y} \frac{\partial y}{\partial x}.
\]

So \( \partial f(x)^T \) “pulls back” sensitivity information.
From math to code

(math)

\[ f(x) = \sin^2(x) \]

impl

\[ \nabla f(x) \]

differentiate

impl

(code)

def f(x):
    return \sin(x) ** 2

grad(f)

AD
import jax.numpy as jnp

def predict(params, inputs):
    for W, b in params:
        outputs = jnp.dot(inputs, W) + b
        inputs = jnp.tanh(outputs)
    return outputs
import jax.numpy as jnp

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

primitives
Two kinds of function: primitive and composite

```python
import jax.numpy as jnp

def predict(params, inputs):
    for W, b in params:
        outputs = jnp.dot(inputs, W) + b
        inputs = jnp.tanh(outputs)
    return outputs
```
Two kinds of function: primitive and composite

```python
import jax.numpy as jnp

def predict(params, inputs):
    for W, b in params:
        outputs = jnp.dot(inputs, W) + b
        inputs = jnp.tanh(outputs)
    return outputs
```

For **primitive** functions we have a lookup table

For **composite** functions we use the chain rule
JVPs in code: types

$$(x, v) \mapsto (f(x), \partial f(x)[v])$$
JVPs in code: types

\[(x, v) \mapsto (f(x), \partial f(x)[v])\]

\[
\text{jvp} :: \ (a \to b) \to (a, T\ a) \to (b, T\ b)
\]
JVPs in code: types

\[(x, v) \mapsto (f(x), \partial f(x)[v])\]

\(\text{jvp} :: (\text{a} \to \text{b}) \to (\text{a}, \text{T a}) \to (\text{b}, \text{T b})\)

given a function
JVPs in code: types

\[(x, v) \mapsto (f(x), \partial f(x)[v])\]

\[
jvp :: (a \rightarrow b) \rightarrow (a, T a) \rightarrow (b, T b)
\]

given a function

a “primal” input, and

“tangent” input perturbation
JVPs in code: types

\[(x, v) \mapsto (f(x), \partial f(x)[v])\]

\[
jvp :: (a \to b) \to (a, T a) \to (b, T b)
\]

given a function

a “primal” input, and
“tangent” input perturbation

return a primal output, and
tangent output perturbation
JVPs in code: types

\[(x, v) \mapsto (f(x), \partial f(x)[v])\]

\[
jvp :: (a \to b) \to (a, T a) \to (b, T b)
\]

given a function

a “primal” input, and
“tangent” input perturbation

return a primal output, and
tangent output perturbation

\[T \ a \equiv a\]
JVPs in code: abstract version

\[
jvp :: (a \to b) \to (a, T a) \to (b, T b)
\]
JVPs in code: abstract version

\[
jvp :: (a \rightarrow b) \rightarrow (a, T a) \rightarrow (b, T b)
\]

\[
jvp \; \text{sin} \; (x, x\_dot) =
\]
\[
\begin{align*}
\text{let } & y = \text{sin} \; x \\
& y\_dot = (\cos \; x) \times x\_dot \\
\text{in } & (y, y\_dot)
\end{align*}
\]
JVPs in code: abstract version

\[ \text{jvp} :: (a \to b) \to (a, T\ a) \to (b, T\ b) \]

\[
\text{jvp} (f \ . \ g) (x, x_{\text{dot}}) = \\
\text{let } (y, y_{\text{dot}}) = \text{jvp} g (x, x_{\text{dot}}) \\
(z, z_{\text{dot}}) = \text{jvp} f (y, y_{\text{dot}}) \\
in (z, z_{\text{dot}})
\]
JVPs in code: Python version (sketch)

```python
DualNumber = namedtuple('DualNumber',
    ['primal', 'tangent'])

def sin(x):
    t = type(x)
    if t == DualNumber:
        return DualNumber(sin(x.primal),
                          mul(x.tangent, cos(x.primal)))
    elif ...

def neg(x):
    t = type(x)
    if t == DualNumber:
        return DualNumber(neg(x.primal),
                          neg(x.tangent))
    elif ...
```
JVPs in code: Python version (sketch)

```python
DualNumber = namedtuple('DualNumber',
    ['primal', 'tangent'])

def add(x, y):
    t = (type(x), type(y))
    if t == (DualNumber, DualNumber):
        return DualNumber(add(x.primal, y.primal),
                          add(x.tangent, y.tangent))
    elif ...
```
JVPs in code: Python version (sketch)

```python
DualNumber = namedtuple('DualNumber',
                         ['primal', 'tangent'])

def add(x, y):
    t = (type(x), type(y))
    if t == (DualNumber, DualNumber):
        return DualNumber(add(x.primal, y.primal),
                           add(x.tangent, y.tangent))
    elif ...
```

What if these represent different perturbations?
Demo!
Linearization in code: types

\[ x \mapsto (f(x), \partial f(x)) \]
Linearization in code: types

\[ x \mapsto (f(x), \partial f(x)) \]

\[
\text{lin} :: (a \to b) \to a \to (b, T a \to T b)
\]
Linearization in code: types

\[ x \mapsto (f(x), \partial f(x)) \]

\[
\text{lin :: } (a \to b) \to a \to (b, T a \to o T b)
\]

given a function
Linearization in code: types

\[ x \mapsto (f(x), \partial f(x)) \]

\textbf{lin} :: (a \to b) \to a \to (b, T \text{ a} \to o T \text{ b})

given a function

and a primal input
Linearization in code: types

\[ x \mapsto (f(x), \partial f(x)) \]

\[ \text{lin :: (a -> b) -> a -> (b, T a --> o T b)} \]

given a function

and a primal input

return a primal output, and

a linearized function
Linearization in code: types

\[ x \mapsto (f(x), \partial f(x)) \]

\[
\text{lin :: } (a \to b) \to a \to (b, T\ a \ --\o\ T\ b)
\]

given a function

and a primal input

return a primal output, and

a linearized function

\(--\o\) means “linear function”
Linearization in code: abstract version

\[ \text{lin} :: (a \to b) \to a \to (b, T a \to o T b) \]
Linearization in code: abstract version

lin :: (a -> b) -> a -> (b, T a --o T b)

lin sin x =
    (sin x, lambda x_dot: (cos x) * x_dot)
Linearization in code: abstract version

```
lin :: (a -> b) -> a -> (b, T a --o T b)

lin (f . g) x =
    let y, g_lin = lin g x
        z, f_lin = lin f y
    in (z, f_lin . g_lin)
```
VJPs in code: types

\[ x \mapsto (f(x), \partial f(x)^T) \]
VJPs in code: types

\[ x \mapsto (f(x), \partial f(x)^T) \]

\[
vjp :: (a \to b) \to a \to (b, CT b \rightarrow o CT a)\]
VJPs in code: types

\[ x \mapsto (f(x), \partial f(x)^T) \]

\[
vjp :: (a -> b) -> a -> (b, CT b --o CT a)
\]

given a function
VJPs in code: types

\[ x \mapsto (f(x), \partial f(x)^T) \]

\texttt{vjp :: (a -> b) -> a -> (b, CT b --o CT a)}

given a function

and a primal input
VJPs in code: types

\[ x \mapsto (f(x), \partial f(x)^T) \]

\[ \text{vjp :: (a -> b) -> a -> (b, CT b --o CT a)} \]

given a function

and a primal input

return a primal output, and
the transposed linearized function
VJPs in code: types

\[ x \mapsto (f(x), \partial f(x)^T) \]

\( vjp :: (a \to b) \to a \to (b, \text{CT} b \to \text{CT} a) \)

given a function

and a primal input

return a primal output, and
the transposed linearized function

\( \text{CT} \ a \equiv a \)
VJPs in code: abstract version

\[ \text{vjp} :: (a \rightarrow b) \rightarrow a \rightarrow (b, \text{CT} b \rightarrow \text{CT} a) \]
VJPs in code: abstract version

\[
\text{vjp :: (a -> b) -> a -> (b, CT b --o CT a)}
\]

\[
\text{vjp } \sin \ x = \\
\text{let } y = \sin x \\
\quad \sin_{\text{vjp}} = \lambda y_{\text{bar}}: (\cos x) * y_{\text{bar}} \\
\text{in (y, sin_{\text{vjp}})}
\]
VJPs in code: abstract version

\[ \text{vjp :: } (a \to b) \to a \to (b, \text{CT } b \dashrightarrow \text{CT } a) \]

\[ \text{vjp } (f \cdot g) \ x = \]

\[ \text{let } (y, g_{\text{vjp}}) = \text{vjp } g \ x \]
\[ (z, f_{\text{vjp}}) = \text{vjp } f \ y \]
\[ \text{in } (z, g_{\text{vjp}} \ . \ f_{\text{vjp}}) \]
VJPs in code: Python version

https://github.com/mattjj/autodidact

http://videolectures.net/deeplearning2017_johnson_automatic_differentiation/
The two linear maps of AD

\[ \text{jvp} :: (a \to b) \to (a, T a) \to (b, T b) \]

build Jacobian one \textit{column} at a time

costs \(O(1)\) times the FLOPs, and \(\sim 2x\) memory
The two linear maps of AD

\[
\text{jvp} :: (a \rightarrow b) \rightarrow (a, T\ a) \rightarrow (b, T\ b)
\]
build Jacobian one \textit{column} at a time
costs \(O(1)\) times the FLOPs, and \(\sim 2x\) memory

\[
\text{vjp} :: (a \rightarrow b) \rightarrow a \rightarrow (b, CT\ b --o CT\ a)
\]
build Jacobian one \textit{row} at a time
costs \(O(1)\) times the FLOPs, and \(O(\text{depth})\) memory
What about $\text{grad}$?
What about $\text{grad}$?

$$\ell : \mathbb{R}^{10^9} \rightarrow \mathbb{R}$$
What about \textbf{grad}?

\[ \ell : \mathbb{R}^{10^9} \rightarrow \mathbb{R} \quad \partial \ell(\theta) \in \mathbb{R}^{1 \times 10^9} \]
What about \texttt{grad}?

\[ \ell : \mathbb{R}^{10^9} \rightarrow \mathbb{R} \quad \partial \ell(\theta) \in \mathbb{R}^{1 \times 10^9} \]

```python
def grad(x):
    def gradfun(*args):
        _, f_vjp = vjp(f, *args)
        return f_vjp(1.)
    return gradfun
```
Demo!
Differentiating solvers and optimizers

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

\[ x^*(a) \triangleq \text{solve } g(x, a) = 0 \]
Differentiating solvers and optimizers

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

optimality conditions

\[ x^*(a) \triangleq \text{solve } g(x, a) = 0 \]
Differentiating solvers and optimizers

\[ x^*(a) \triangleq \text{solve } g(x, a) = 0 \]

\[ g(x^*, a_0) = 0 \]
Differentiating solvers and optimizers

\[ x^*(a) \triangleq \text{solve } g(x, a) = 0 \]

\[ g(x^*(a), a) = 0 \quad \forall a \text{ near } a_0 \]
Differentiating solvers and optimizers

\[ x^*(a) \triangleq \text{solve } g(x, a) = 0 \]

\[ g(x^*(a), a) = 0 \quad \forall a \text{ near } a_0 \]

\[ \partial_0 g(x^*(a), a) \partial x^*(a) + \partial_1 g(x^*(a), a) = 0 \]
Differentiating solvers and optimizers

\[ x^*(a) \triangleq \text{solve } g(x, a) = 0 \]

\[ g(x^*(a), a) = 0 \quad \forall a \text{ near } a_0 \]

\[ \partial_0 g(x^*(a), a) \partial x^*(a) + \partial_1 g(x^*(a), a) = 0 \]

\[ \partial x^*(a) = -\partial_0 g(x^*, a)^{-1} \partial_1 g(x^*, a) \]
Differentiating solvers and optimizers

\[ x^*(a) \triangleq \text{solve } g(x, a) = 0 \]

\[ g(x^*(a), a) = 0 \quad \forall a \text{ near } a_0 \]

\[ \partial_0 g(x^*(a), a) \partial x^*(a) + \partial_1 g(x^*(a), a) = 0 \]

\[ \partial x^*(a) = -\partial_0 g(x^*, a)^{-1} \partial_1 g(x^*, a) \]

\[ w^T \partial x^*(a) = - (w^T \partial_0 g(x^*, a)^{-1}) \partial_1 g(x^*, a) \]
Differentiating iterative fixed points

\[ x^*(a) \triangleq \text{solve } x = f(x, a) \]
Differentiating iterative fixed points

\[ x^*(a) \triangleq \text{solve } x = f(x, a) \]

\[ x^{k+1} = f(x^k, a) \]
Differentiating iterative fixed points

\[ x^*(a) \triangleq \text{solve } x = f(x, a) \]

\[ x^{k+1} = f(x^k, a) \]

\[ \partial x^*(a) = \partial_0 f(x^*(a), a) \partial x^*(a) + \partial_1 f(x^*(a), a) \]
Differentiating iterative fixed points

\[ x^*(a) \triangleq \text{solve } x = f(x, a) \]

\[ x^{k+1} = f(x^k, a) \]

\[ \partial x^*(a) = \partial_0 f(x^*(a), a) \partial x^*(a) + \partial_1 f(x^*(a), a) \]

\[ A \triangleq \partial_0 f(x^*(a), a) \]

\[ B \triangleq \partial_1 f(x^*(a), a) \]
Differentiating iterative fixed points

\[ x^*(a) \triangleq \text{solve } x = f(x, a) \]

\[ \partial x^*(a) = A \partial x^*(a) + B \]
Differentiating iterative fixed points

\[ x^*(a) \triangleq \text{solve } x = f(x, a) \]

\[ \partial x^*(a) = A \partial x^*(a) + B \]

\[ \partial x^*(a) = (I - A)^{-1} B \]
Differentiating iterative fixed points

\[ x^*(a) \triangleq \text{solve } x = f(x, a) \]

\[ \partial x^*(a) = A \partial x^*(a) + B \]

\[ w^T \partial x^*(a) = w^T (I - A)^{-1} B \]
Differentiating iterative fixed points

\[ x^*(a) \triangleq \text{solve } x = f(x, a) \]

\[ \partial x^*(a) = A \partial x^*(a) + B \]

\[ w^T \partial x^*(a) = w^T (I - A)^{-1} B \]

\[ = u^T B \]

where \( u^T = w^T (I - A)^{-1} \)
Differentiating iterative fixed points

\[ x^*(a) \triangleq \text{solve } x = f(x, a) \]

\[ \partial x^*(a) = A \partial x^*(a) + B \]

\[ w^\top \partial x^*(a) = w^\top (I - A)^{-1} B \]

\[ = u^\top B \]

where \( u^\top = w^\top + u^\top A \)  

backward pass is a (linear) fixed point!
Demo!
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints

\[ \frac{\partial y}{\partial y} = 1 \]
Differentiating long iterations: checkpoints

\[ \frac{\partial y}{\partial y} = 1 \]
Differentiating long iterations: checkpoints

\[ \frac{\partial y}{\partial y} = 1 \]
Differentiating long iterations: checkpoints

\[
\frac{\partial y}{\partial y} = 1
\]
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Differentiating long iterations: checkpoints
Demo!
Differentiating long iterations: checkpoints

Computing VJPs uses $\mathcal{O}(D)$ memory for depth $D$. 
Differentiating long iterations: checkpoints

Computing VJPs uses $O(D)$ memory for depth $D$.

With recursive checkpointing, only need $O(\log D)$ memory, at overhead of $O(\log D)$ times as much computation.
Differentiating long iterations: checkpoints

Computing VJPs uses $\mathcal{O}(D)$ memory for depth $D$.

With recursive checkpointing, only need $\mathcal{O}(\log D)$ memory, at overhead of $\mathcal{O}(\log D)$ times as much computation.

...can we ever get $\mathcal{O}(1)$ memory and $\mathcal{O}(1)$ overhead?
Differentiating long iterations: reversibility

Differentiating long iterations: reversibility

Algorithm 1 Stochastic gradient descent with momentum

1: \textbf{input:} initial $w_1$, decays $\gamma$, learning rates $\alpha$, loss function $L(w, \theta, t)$
2: initialize $v_1 = 0$
3: \textbf{for} $t = 1$ \textbf{to} $T$ \textbf{do}
4: \hspace{1em} $g_t = \nabla_w L(w_t, \theta, t)$ \hfill $\triangleright$ evaluate gradient
5: \hspace{1em} $v_{t+1} = \gamma_t v_t - (1 - \gamma_t) g_t$ \hfill $\triangleright$ update velocity
6: \hspace{1em} $w_{t+1} = w_t + \alpha_t v_t$ \hfill $\triangleright$ update position
7: \textbf{end for}
8: \textbf{output} trained parameters $w_T$

Differentiating long iterations: reversibility

Algorithm 1 Stochastic gradient descent with momentum

1: input: initial $w_1$, decays $\gamma$, learning rates $\alpha$, loss function $L(w, \theta, t)$
2: initialize $v_1 = 0$
3: for $t = 1$ to $T$ do
4:   $g_t = \nabla_w L(w_t, \theta, t)$ \> evaluate gradient
5:   $v_{t+1} = \gamma_t v_t - (1 - \gamma_t)g_t$ \> update velocity
6:   $w_{t+1} = w_t + \alpha_t v_t$ \> update position
7: end for
8: output trained parameters $w_T$

Algorithm 2 Reverse-mode differentiation of SGD

1: input: $w_T$, $v_T$, $\gamma$, $\alpha$, train loss $L(w, \theta, t)$, loss $f(w)$
2: initialize $dv = 0$, $d\theta = 0$, $d\alpha_t = 0$, $d\gamma = 0$
3: initialize $dw = \nabla_w f(w_T)$
4: for $t = T$ counting down to $1$ do
5:   $d\alpha_t = dw^T v_t$
6:   $w_{t-1} = w_t - \alpha_t v_t$
7:   $g_t = \nabla_w L(w_t, \theta, t)$ \> exactly reverse
8:   $v_{t-1} = [v_t + (1 - \gamma_t)g_t]/\gamma_t$ \> gradient descent
9:   $dv = dv + \alpha_t dw$ \> operations
10: $d\gamma_t = dv^T(v_t + g_t)$
11: $dw = dw - (1 - \gamma_t)dv\nabla_w \nabla_w L(w_t, \theta, t)$
12: $d\theta = d\theta - (1 - \gamma_t)dv\nabla_\theta \nabla_w L(w_t, \theta, t)$
13: $dv = \gamma_t dv$
14: end for
15: output gradient of $f(w_T)$ w.r.t $w_1$, $v_1$, $\gamma$, $\alpha$ and $\theta$

Differentiating long iterations: reversibility

Algorithm 1 Stochastic gradient descent with momentum

1: **input**: initial $w_1$, decays $\gamma$, learning rates $\alpha$, loss function $L(w, \theta, t)$
2: initialize $v_1 = 0$
3: for $t = 1$ to $T$ do
4:   $g_t = \nabla_w L(w_t, \theta, t)$ // evaluate gradient
5:   $v_{t+1} = \gamma_t v_t - (1 - \gamma_t) g_t$ // update velocity
6:   $w_{t+1} = w_t + \alpha_t v_t$ // update position
7: end for
8: **output** trained parameters $w_T$

Algorithm 2 Reverse-mode differentiation of SGD

1: **input**: $w_T, v_T, \gamma, \alpha$, train loss $L(w, \theta, t)$, loss $f(w)$
2: initialize $dv = 0, d\theta = 0, d\alpha_t = 0, d\gamma = 0$
3: initialize $dw = \nabla_w f(w_T)$
4: for $t = T$ counting down to 1 do
5:   $d\alpha_t = dw^T v_t$
6:   $w_{t-1} = w_t - \alpha_t v_t$
7:   $g_t = \nabla_w L(w_t, \theta, t)$
8:   $v_{t-1} = [v_t + (1 - \gamma_t) g_t]/\gamma_t$
9:   $dv = dv + \alpha_t dw$
10: $d\gamma_t = dv^T (v_t + g_t)$
11: $dw = dw - (1 - \gamma_t) dv \nabla_w \nabla_w L(w_t, \theta, t)$
12: $d\theta = d\theta - (1 - \gamma_t) dv \nabla_\theta \nabla_w L(w_t, \theta, t)$
13: $dv = \gamma_t dv$
14: end for
15: **output** gradient of $f(w_T)$ w.r.t $w_1, v_1, \gamma, \alpha$ and $\theta$

Differentiating long iterations: reversibility

Differentiating long iterations: reversibility

Differentiating long iterations: reversibility

Algorithm 1 Stochastic gradient descent with momentum
1: **input**: initial $w_1$, decays $\gamma$, learning rates $\alpha$, loss function $L(w, \theta, t)$
2: initialize $v_1 = 0$
3: for $t = 1$ to $T$ do
4:   $g_t = \nabla_w L(w_t, \theta, t)$ ▶ evaluate gradient
5:   $v_{t+1} = \gamma_t v_t - (1 - \gamma_t) g_t$ ▶ update velocity
6:   $w_{t+1} = w_t + \alpha_t v_t$ ▶ update position
7: end for
8: **output** trained parameters $w_T$

Algorithm 2 Reverse-mode differentiation of SGD
1: **input**: $w_T, v_T, \gamma, \alpha$, train loss $L(w, \theta, t)$, loss $f(w)$
2: initialize $dv = 0$, $d\theta = 0$, $d\alpha_t = 0$, $d\gamma = 0$
3: initialize $dw = \nabla_w f(w_T)$
4: for $t = T$ counting down to 1 do
5:   $d\alpha_t = dw^T v_t$
6:   $w_{t-1} = w_t - \alpha_t v_t$ \{ exactly reverse gradient descent operations \}
7:   $g_t = \nabla_w L(w_t, \theta, t)$
8:   $v_{t-1} = [v_t + (1 - \gamma_t)g_t]/\gamma_t$ \{ gradient descent operations \}
9:   $dv = dv + \alpha_t dw$
10: $d\gamma_t = dv^T (v_t + g_t)$
11: $dw = dw - (1 - \gamma_t) dv \nabla_w \nabla_w L(w_t, \theta, t)$
12: $d\theta = d\theta - (1 - \gamma_t) dv \nabla_\theta \nabla_w L(w_t, \theta, t)$
13: $dv = \gamma_t dv$
14: end for
15: **output** gradient of $f(w_T)$ w.r.t $w_1, v_1, \gamma, \alpha$ and $\theta$

Differentiating long iterations: reversibility

Algorithm 1 Stochastic gradient descent with momentum

1: **input:** initial $w_1$, decays $\gamma$, learning rates $\alpha$, loss function $L(w, \theta, t)$
2: initialize $v_1 = 0$
3: for $t = 1$ to $T$ do
4: \[ g_t = \nabla_w L(w_t, \theta, t) \] ▶ evaluate gradient
5: \[ v_{t+1} = \gamma_t v_t - (1 - \gamma_t) g_t \] ▶ update velocity
6: \[ w_{t+1} = w_t + \alpha_t v_t \] ▶ update position
7: end for
8: **output** trained parameters $w_T$

finite precision means information loss!

Algorithm 2 Reverse-mode differentiation of SGD

1: **input:** $w_T, v_T, \gamma, \alpha$, train loss $L(w, \theta, t)$, loss $f(w)$
2: initialize $dv = 0, d\theta = 0, d\alpha = 0, d\gamma = 0$
3: initialize $dw = \nabla_w f(w_T)$
4: for $t = T$ counting down to 1 do
5: \[ d\alpha_t = dw^T v_t \]
6: \[ w_{t-1} = w_t - \alpha_t v_t \] exactly reverse gradient descent operations
7: \[ g_t = \nabla_w L(w_t, \theta, t) \]
8: \[ v_{t-1} = [v_t + (1 - \gamma_t) g_t] / \gamma_t \]
9: \[ dv = dv + \alpha_t dw \]
10: \[ d\gamma_t = dv^T (v_t + g_t) \]
11: \[ dw = dw - (1 - \gamma_t) dv \nabla_w L(w_t, \theta, t) \]
12: \[ d\theta = d\theta - (1 - \gamma_t) dv \nabla_\theta L(w_t, \theta, t) \]
13: \[ dv = \gamma_t dv \]
14: end for
15: **output** gradient of $f(w_T)$ w.r.t $w_1, v_1, \gamma, \alpha$ and $\theta$

Differentiating long iterations: reversibility

Differentiating long iterations: reversibility

from jax import reversible


Very high-order AD with jet

Neural ODEs can be slow at test time...

Idea: speed them up by regularizing higher derivatives, learning ODEs that are easy to solve!
Very high-order AD with jet

Neural ODEs can be slow at test time...

Idea: speed them up by regularizing higher derivatives, learning ODEs that are easy to solve!
Very high-order AD with \textit{jet}

\[ f = g \circ h \]

given \((h(x), \partial h(x)[v])\)

compute \((f(x), \partial f(x)[v])\)

using \(g\) and \(\partial g\)
Very high-order AD with \texttt{jet}

\[ f = g \circ h \]

given \((h(x), \partial h(x)[v], \frac{1}{2} \partial^2 h(x)[v, v], \ldots, \frac{1}{K!} \partial^K h(x)[v, \ldots, v])\)

compute \((f(x), \partial f(x)[v], \frac{1}{2} \partial^2 f(x)[v, v], \ldots, \frac{1}{K!} \partial^K f(x)[v, \ldots, v])\)

using \(g, \partial g, \partial^2 g, \ldots, \) and \(\partial^K g\)
Very high-order AD with jet

Very high-order AD with jet

This project is a good demonstration of the composability of JAX transformations. JAX jitted the gradient of auto-minibatched ode solutions of jets of a neural net!

```
jit(vmap(grad(odeint(jet(neural_net)))))(x, θ)
```

https://twitter.com/DavidDuvenaud/status/1284181673496776706
Thanks!