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?

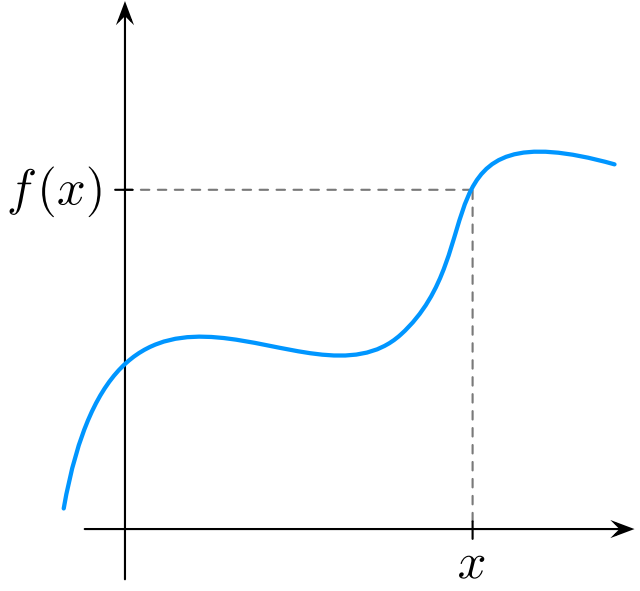
Demo!

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

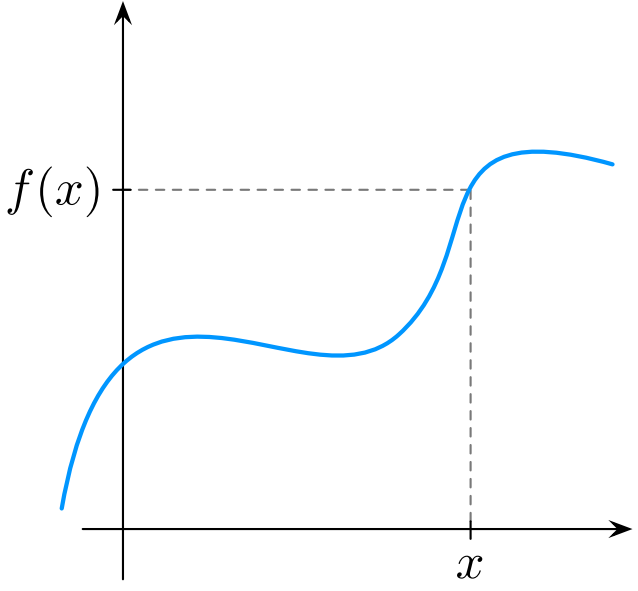
$$f : \mathbb{R}^n \to \mathbb{R}^m$$



Math and notation

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

$$\partial f : \mathbb{R}^n \to (\mathbb{R}^n \to \mathbb{R}^m)$$

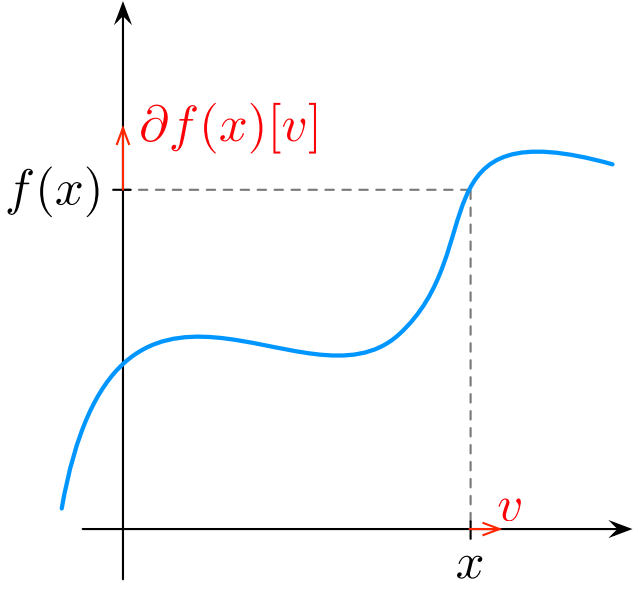


Math and notation

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

$$\partial f : \mathbb{R}^n \to (\mathbb{R}^n \to \mathbb{R}^m)$$

$$\partial f(x) : \mathbb{R}^n \to \mathbb{R}^m$$



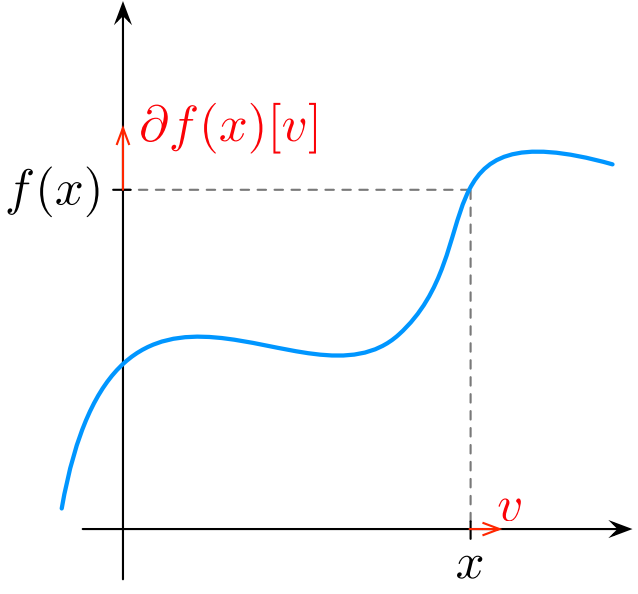
Math and notation

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

$$\partial f : \mathbb{R}^n \to (\mathbb{R}^n \to \mathbb{R}^m)$$

$$\partial f(x) : \mathbb{R}^n \to \mathbb{R}^m$$

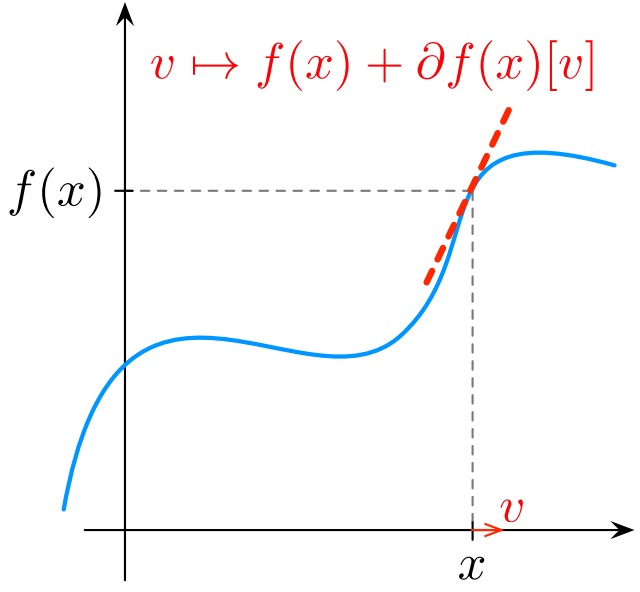
$$\partial f(x) \in \mathbb{R}^{m \times n}$$



Math and notation

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

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



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

$$y = f(x)$$

$$\frac{\partial y}{\partial x} = \partial f$$

$$\left.\frac{\partial y}{\partial x}\right|_{x=a} = \partial f(a)$$

Math and notation: gradients

$\nabla f(x)$ is the vector such that

$$\langle \nabla f(x), v \rangle = \partial f(x)[v]$$

Math and notation: chain rule

$$f = g \circ h \qquad\qquad f : \mathbb{R}^n \to \mathbb{R}^m$$

Math and notation: chain rule

$$f = g \circ h \qquad f : \mathbb{R}^n \to \mathbb{R}^m$$

$$h : \mathbb{R}^n \to \mathbb{R}^p \qquad g : \mathbb{R}^p \to \mathbb{R}^m$$

Math and notation: chain rule

$$f = g \circ h \qquad f : \mathbb{R}^n \to \mathbb{R}^m$$

$$h : \mathbb{R}^n \to \mathbb{R}^p \qquad g : \mathbb{R}^p \to \mathbb{R}^m$$

$$\partial f(x) = \partial g(h(x)) \circ \partial h(x)$$

Math and notation: chain rule

$$f = g \circ h \qquad f : \mathbb{R}^n \to \mathbb{R}^m$$

$$h : \mathbb{R}^n \to \mathbb{R}^p \qquad g : \mathbb{R}^p \to \mathbb{R}^m$$

$$\partial f(x) = \partial g(h(x)) \circ \partial h(x)$$

$$\partial f(x)[v] = \partial g(h(x))[\,\partial h(x)[v]\,]$$

Math and notation: chain rule

$$f = g \circ h \qquad f : \mathbb{R}^n \to \mathbb{R}^m$$

$$h : \mathbb{R}^n \to \mathbb{R}^p \qquad g : \mathbb{R}^p \to \mathbb{R}^m$$

$$\left.\frac{\partial y}{\partial x}\right|_{x=a} = \left.\frac{\partial y}{\partial z}\right|_{z=h(a)} \left.\frac{\partial z}{\partial x}\right|_{x=a}$$

$$z = h(x)$$

$$y = g(z)$$

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. $v \mapsto \partial f(x) v \quad \text{or} \quad v \mapsto \partial f(x)[v]$

   JVP / push-forward / forward-mode

   build Jacobian one column at a time

The two linear maps of AD

1. $v \mapsto \partial f(x) v \quad \text{or} \quad v \mapsto \partial f(x)[v]$

   JVP / push-forward / forward-mode

   build Jacobian one column at a time

2. $w^\mathsf{T} \mapsto w^\mathsf{T} \partial f(x) \quad \text{or} \quad w^\mathsf{T} \mapsto \partial f(x)^\mathsf{T}[w^\mathsf{T}]$

   VJP / pull-back / reverse-mode

   build Jacobian one row at a time

The two linear maps of AD

1. $v \mapsto \partial f(x) v \quad \text{or} \quad v \mapsto \partial f(x)[v]$

The two linear maps of AD

1. $v \mapsto \partial f(x) v \quad \text{or} \quad v \mapsto \partial f(x)[v]$

   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. $v \mapsto \partial f(x) v \quad \text{or} \quad v \mapsto \partial f(x)[v]$

   Let's say you have

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

scalar parameter

   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. $v \mapsto \partial f(x) v \quad \text{or} \quad v \mapsto \partial f(x)[v]$

   Let's say you have

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

scalar parameter

   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. $w^\mathsf{T} \mapsto w^\mathsf{T} \partial f(x) \quad \text{or} \quad w^\mathsf{T} \mapsto \partial f(x)^\mathsf{T} [\, w^\mathsf{T} \,]$

The two linear maps of AD

2. $w^\mathsf{T} \mapsto w^\mathsf{T} \partial f(x) \quad \text{or} \quad w^\mathsf{T} \mapsto \partial f(x)^\mathsf{T}[\, w^\mathsf{T}\,]$

Let's say you have

$$w^\mathsf{T} = \frac{\partial \ell}{\partial y}$$

and you want to compute

$$u^\mathsf{T} = \frac{\partial \ell}{\partial x} \quad \text{where} \quad y = f(x)$$

The two linear maps of AD

2. $w^\mathsf{T} \mapsto w^\mathsf{T} \partial f(x) \quad \text{or} \quad w^\mathsf{T} \mapsto \partial f(x)^\mathsf{T}[\, w^\mathsf{T}\,]$

Let's say you have

$$w^\mathsf{T} = \frac{\partial \ell}{\partial y}$$

scalar loss value

and you want to compute

$$u^\mathsf{T} = \frac{\partial \ell}{\partial x} \quad \text{where} \quad y = f(x)$$

The two linear maps of AD

2. $w^\mathsf{T} \mapsto w^\mathsf{T} \partial f(x) \quad \text{or} \quad w^\mathsf{T} \mapsto \partial f(x)^\mathsf{T}[\, w^\mathsf{T}\,]$

Let's say you have

$$w^\mathsf{T} = \frac{\partial \ell}{\partial y}$$

scalar loss value

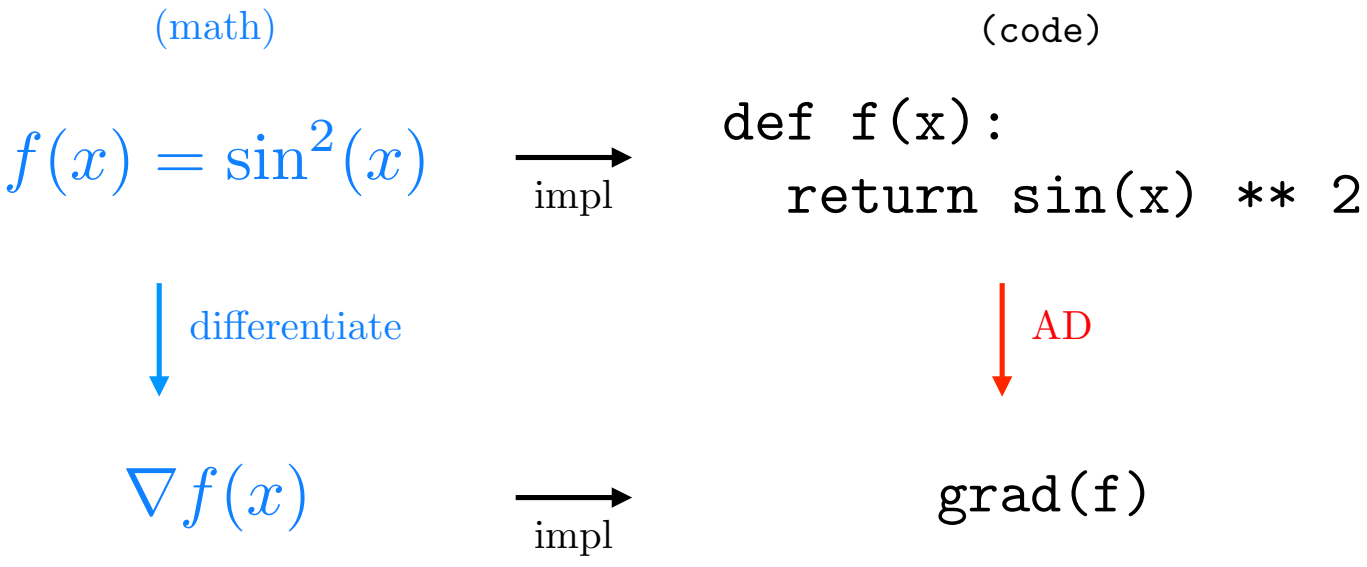
and you want to compute

$$u^\mathsf{T} = \frac{\partial \ell}{\partial x} \quad \text{where} \quad y = f(x)$$

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

So $\partial f(x)^\mathsf{T}$ "pulls back" sensitivity information.

From math to code



Two kinds of function: primitive and composite

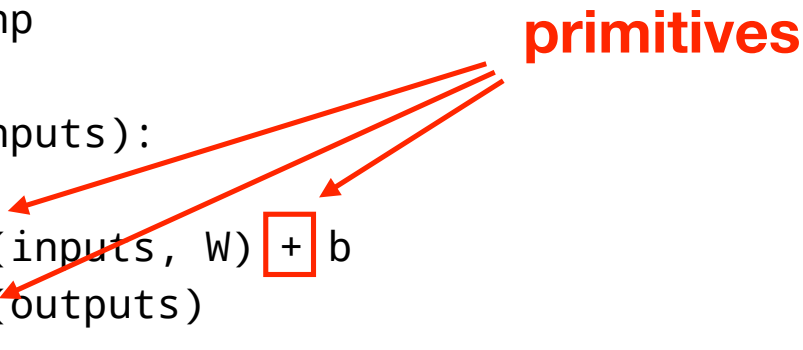
```
import jax.numpy as jnp

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

```
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

```
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
```

composite

Two kinds of function: primitive and composite

```
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

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

```
jvp :: (a -> b) -> (a, T a) -> (b, T b)
```

JVPs in code: types

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

```
jvp :: (a -> b) -> (a, T a) -> (b, T b)
```

given a function

JVPs in code: types

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

```
jvp :: (a -> b) -> (a, T a) -> (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 -> b) -> (a, T a) -> (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 -> b) -> (a, T a) -> (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 -> b) -> (a, T a) -> (b, T b)
```

JVPs in code: abstract version

```
jvp ::  (a -> b) -> (a, T a) -> (b, T b)


jvp sin (x, x_dot) =
  let y = sin x
      y_dot = (cos x) * x_dot
  in (y, y_dot)
```

JVPs in code: abstract version

```
jvp ::  (a -> b) -> (a, T a) -> (b, T b)


jvp (f . g) (x, x_dot) =
  let (y, y_dot) = jvp g (x, x_dot)
      (z, z_dot) = jvp f (y, y_dot)
  in (z, z_dot)
```

JVPs in code: Python version (sketch)

```
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)

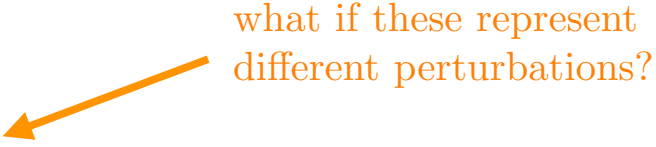
```
DualNumber = namedtuple('DualNumber',
                        ['primal', 'tangent'])
```

```
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)

```
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 ...
```



Demo!

Linearization in code: types

$$x \mapsto (f(x),\ \partial f(x))$$

Linearization in code: types

$$x \mapsto (f(x),\ \partial f(x))$$

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

Linearization in code: types

$$x \mapsto (f(x),\ \partial f(x))$$

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

given a function

Linearization in code: types

$$x \mapsto (f(x),\ \partial f(x))$$

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

given a function

and a primal input

Linearization in code: types

$$x \mapsto (f(x),\ \partial f(x))$$

```
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

$$x \mapsto (f(x),\ \partial f(x))$$

```
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

`--o` means "*linear* function"

Linearization in code: abstract version

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

$$x \mapsto (f(x),\ \partial f(x)^{\mathsf{T}})$$

VJPs in code: types

$$x \mapsto (f(x),\ \partial f(x)^{\mathsf{T}})$$

```
vjp ::  (a -> b) -> a -> (b, CT b --o CT a)
```

VJPs in code: types

$$x \mapsto (f(x),\ \partial f(x)^\mathsf{T})$$

```
vjp ::  (a -> b) -> a -> (b, CT b --o CT a)
```

given a function

VJPs in code: types

$$x \mapsto (f(x),\ \partial f(x)^\mathsf{T})$$

```
vjp ::  (a -> b) -> a -> (b, CT b --o CT a)
```

given a function

and a primal input

VJPs in code: types

$$x \mapsto (f(x),\ \partial f(x)^\mathsf{T})$$

```
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

$$x \mapsto (f(x),\ \partial f(x)^\mathsf{T})$$

```
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

$$\texttt{CT a} \equiv \texttt{a}$$

VJPs in code: abstract version

```
vjp ::  (a -> b) -> a -> (b, CT b --o CT a)
```

VJPs in code: abstract version

```
vjp ::  (a -> b) -> a -> (b, CT b --o CT a)


vjp sin x =
  let y = sin x
      sin_vjp = lambda y_bar: (cos x) * y_bar
  in (y, sin_vjp)
```

VJPs in code: abstract version

```
vjp ::  (a -> b) -> a -> (b, CT b --o CT a)


vjp (f . g) x =
  let (y, g_vjp) = vjp g x
      (z, f_vjp) = vjp f y
  in (z, g_vjp . f_vjp)
```

VJPs in code: Python version

https://github.com/mattjj/autodidact

http://videolectures.net/deeplearning2017_johnson_automatic_differentiation/

The two linear maps of AD

```
jvp ::  (a -> b) -> (a, T a) -> (b, T b)
```

build Jacobian one column at a time

costs $\mathcal{O}(1)$ times the FLOPs, and $\sim$2x memory

The two linear maps of AD

```
jvp :: (a -> b) -> (a, T a) -> (b, T b)
```

build Jacobian one column at a time

costs $\mathcal{O}(1)$ times the FLOPs, and $\sim$2x memory

```
vjp :: (a -> b) -> a -> (b, CT b --o CT a)
```

build Jacobian one row at a time

costs $\mathcal{O}(1)$ times the FLOPs, and $\mathcal{O}(\text{depth})$ memory

What about `grad`?

What about `grad`?

$$\ell : \mathbb{R}^{10^9} \to \mathbb{R}$$

What about `grad`?

$$\ell : \mathbb{R}^{10^9} \to \mathbb{R} \qquad \partial \ell(\theta) \in \mathbb{R}^{1 \times 10^9}$$

What about `grad`?

$$\ell : \mathbb{R}^{10^9} \to \mathbb{R} \qquad \partial\ell(\theta) \in \mathbb{R}^{1 \times 10^9}$$

```
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

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

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

Differentiating solvers and optimizers

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

$$g(x^*,\, a_0) = 0$$

Differentiating solvers and optimizers

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

$$g(x^*(a),\ a) = 0 \qquad \forall a \text{ near } a_0$$

Differentiating solvers and optimizers

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

$$g(x^*(a),\, a) = 0 \qquad \forall a \text{ near } a_0$$

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

Differentiating solvers and optimizers

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

$$g(x^*(a),\ a) = 0 \qquad \forall a \text{ near } a_0$$

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

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

Differentiating solvers and optimizers

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

$$g(x^*(a),\ a) = 0 \qquad \forall a \text{ near } a_0$$

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

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

$$w^\mathsf{T}\partial x^*(a) = -\left(w^\mathsf{T}\partial_0 g(x^*, a)^{-1}\right)\partial_1 g(x^*, a)$$

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

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

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

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

Differentiating iterative fixed points

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

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

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

$$= u^\mathsf{T}B$$

$$\text{where } u^\mathsf{T} = w^\mathsf{T}(I - A)^{-1}$$

Differentiating iterative fixed points

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

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

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

$$= u^\mathsf{T}B$$

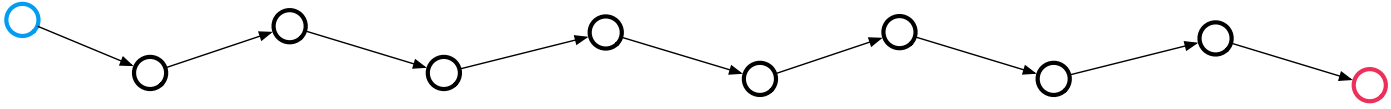
where $u^\mathsf{T} = w^\mathsf{T} + u^\mathsf{T}A$

backward pass is
a (linear) fixed point!

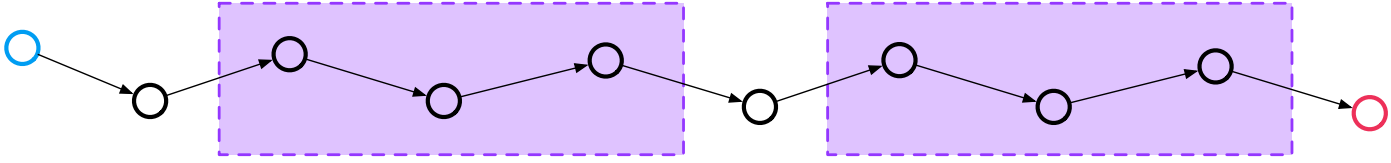
Demo!

Differentiating long iterations: checkpoints

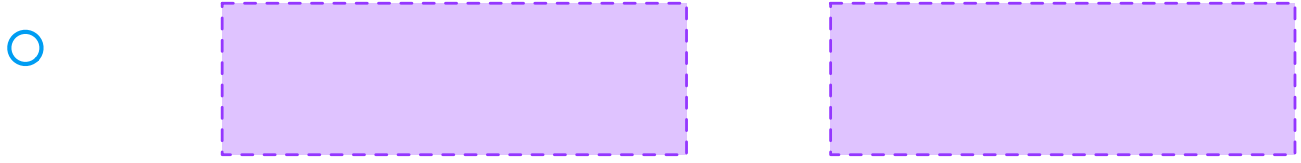
Differentiating long iterations: checkpoints



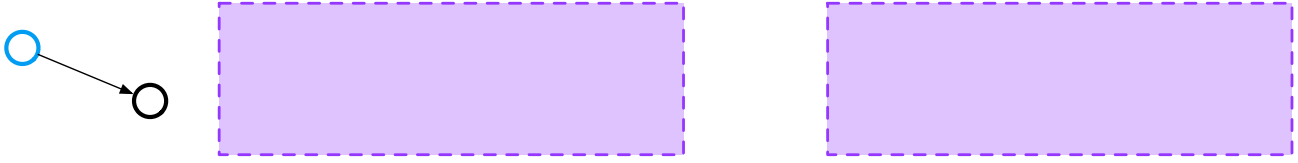
Differentiating long iterations: checkpoints



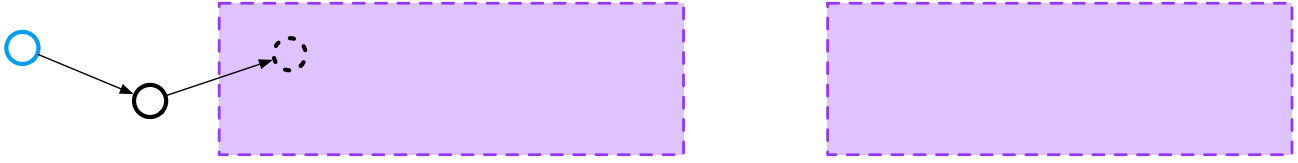
Differentiating long iterations: checkpoints



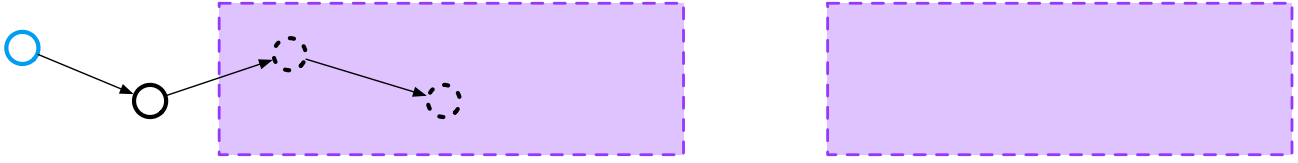
Differentiating long iterations: checkpoints



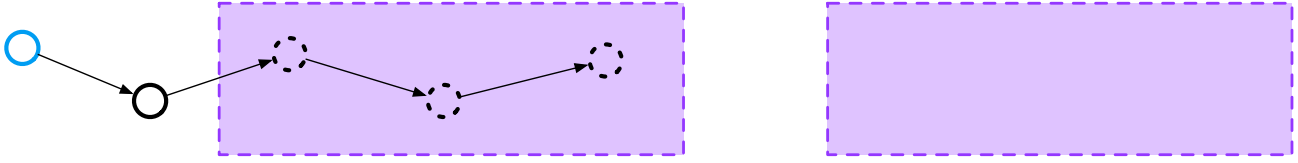
Differentiating long iterations: checkpoints



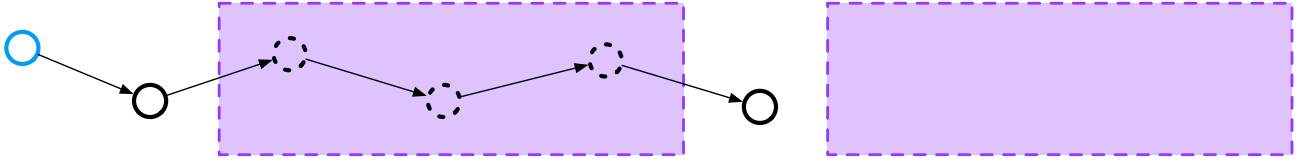
Differentiating long iterations: checkpoints



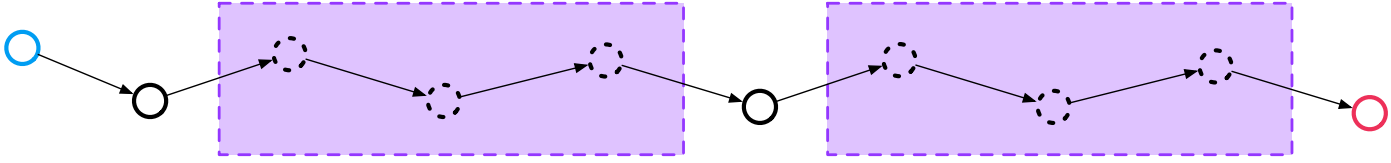
Differentiating long iterations: checkpoints



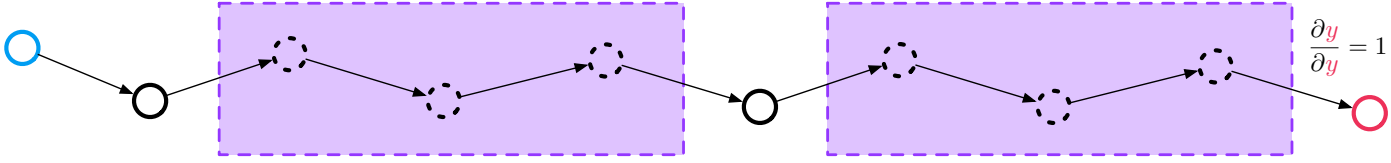
Differentiating long iterations: checkpoints



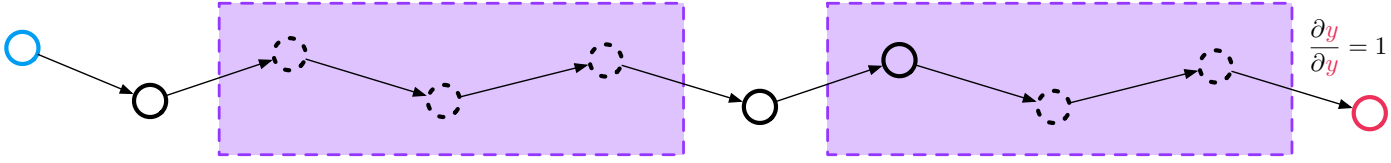
Differentiating long iterations: checkpoints



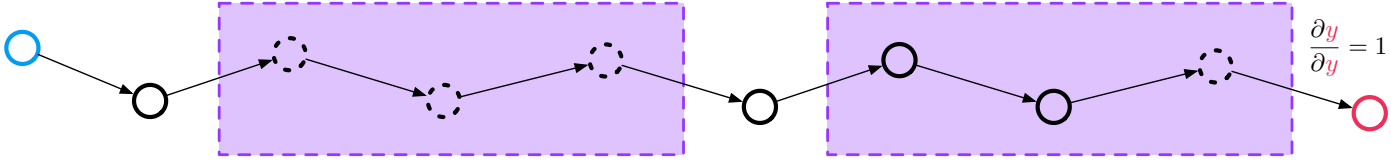
Differentiating long iterations: checkpoints



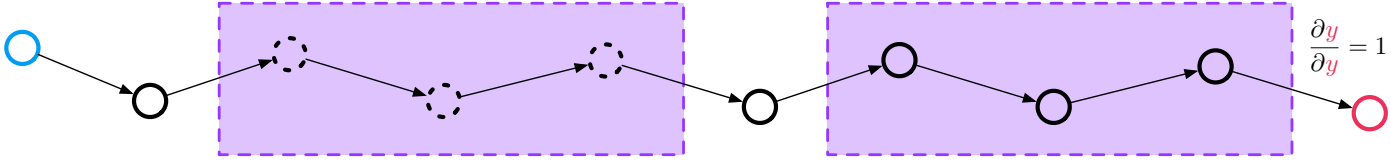
Differentiating long iterations: checkpoints



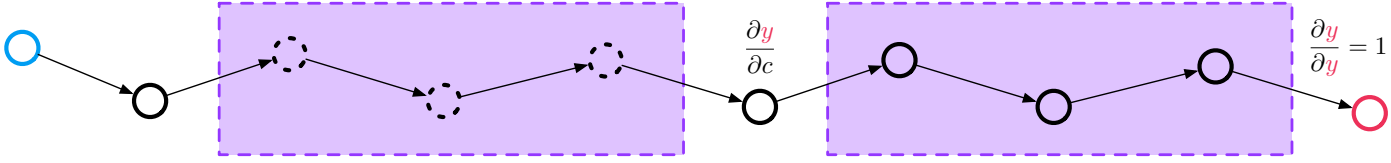
Differentiating long iterations: checkpoints



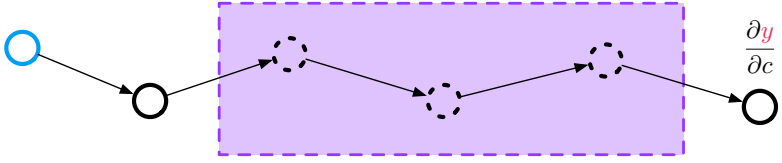
Differentiating long iterations: checkpoints



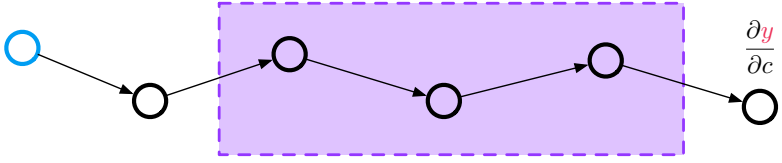
Differentiating long iterations: checkpoints



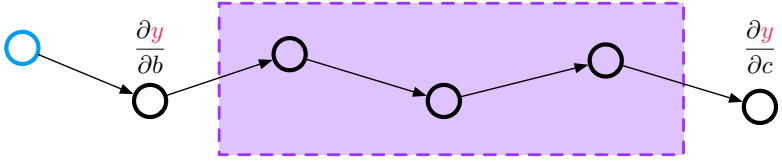
Differentiating long iterations: checkpoints



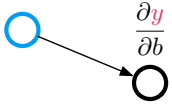
Differentiating long iterations: checkpoints



Differentiating long iterations: checkpoints



Differentiating long iterations: checkpoints



Demo!

Differentiating long iterations: checkpoints

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

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.

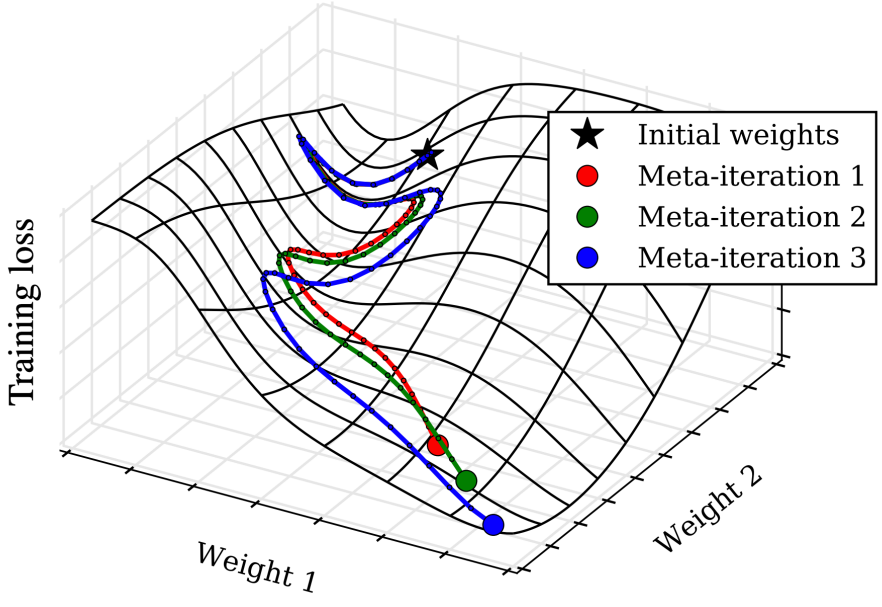
Differentiating long iterations: checkpoints

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



Maclaurin*, Duvenaud*, and Adams. *Gradient-based hyperparameter optimization through reversible learning*. https://arxiv.org/pdf/1502.03492.pdf ICML 2015.

Differentiating long iterations: reversibility

Algorithm 1 Stochastic gradient descent with momentum

input: initial $\mathbf{w}_1$, decays $\boldsymbol{\gamma}$, learning rates $\boldsymbol{\alpha}$, loss function $L(\mathbf{w}, \boldsymbol{\theta}, t)$
initialize $\mathbf{v}_1 = \mathbf{0}$
for $t = 1$ **to** T **do**
    $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ ▷ evaluate gradient
    $\mathbf{v}_{t+1} = \gamma_t \mathbf{v}_t - (1 - \gamma_t)\mathbf{g}_t$ ▷ update velocity
    $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \mathbf{v}_t$ ▷ update position
end for
output trained parameters $\mathbf{w}_T$

Maclaurin*, Duvenaud*, and Adams. *Gradient-based hyperparameter optimization through reversible learning*. https://arxiv.org/pdf/1502.03492.pdf ICML 2015.

Differentiating long iterations: reversibility

Algorithm 1 Stochastic gradient descent with momentum

input: initial $\mathbf{w}_1$, decays $\boldsymbol{\gamma}$, learning rates $\boldsymbol{\alpha}$, loss function $L(\mathbf{w}, \boldsymbol{\theta}, t)$
initialize $\mathbf{v}_1 = \mathbf{0}$
for $t = 1$ **to** T **do**
    $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ ▷ evaluate gradient
    $\mathbf{v}_{t+1} = \gamma_t \mathbf{v}_t - (1 - \gamma_t)\mathbf{g}_t$ ▷ update velocity
    $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \mathbf{v}_t$ ▷ update position
end for
output trained parameters $\mathbf{w}_T$

Algorithm 2 Reverse-mode differentiation of SGD

input: $\mathbf{w}_T$, $\mathbf{v}_T$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$, train loss $L(\mathbf{w}, \boldsymbol{\theta}, t)$, loss $f(\mathbf{w})$
initialize $d\mathbf{v} = \mathbf{0}$, $d\boldsymbol{\theta} = \mathbf{0}$, $d\alpha_t = \mathbf{0}$, $d\gamma = \mathbf{0}$
initialize $d\mathbf{w} = \nabla_{\mathbf{w}} f(\mathbf{w}_T)$
for $t = T$ **counting down to** 1 **do**
    $d\alpha_t = d\mathbf{w}^\mathsf{T} \mathbf{v}_t$
    $\mathbf{w}_{t-1} = \mathbf{w}_t - \alpha_t \mathbf{v}_t$ } exactly reverse
    $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ } gradient descent
    $\mathbf{v}_{t-1} = [\mathbf{v}_t + (1 - \gamma_t)\mathbf{g}_t]/\gamma_t$ } operations
    $d\mathbf{v} = d\mathbf{v} + \alpha_t d\mathbf{w}$
    $d\gamma_t = d\mathbf{v}^\mathsf{T}(\mathbf{v}_t + \mathbf{g}_t)$
    $d\mathbf{w} = d\mathbf{w} - (1 - \gamma_t) d\mathbf{v} \nabla_{\mathbf{w}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
    $d\boldsymbol{\theta} = d\boldsymbol{\theta} - (1 - \gamma_t) d\mathbf{v} \nabla_{\boldsymbol{\theta}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
    $d\mathbf{v} = \gamma_t d\mathbf{v}$
end for
output gradient of $f(\mathbf{w}_T)$ w.r.t $\mathbf{w}_1$, $\mathbf{v}_1$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}$

Maclaurin*, Duvenaud*, and Adams. *Gradient-based hyperparameter optimization through reversible learning*. https://arxiv.org/pdf/1502.03492.pdf ICML 2015.

Differentiating long iterations: reversibility

Algorithm 1 Stochastic gradient descent with momentum

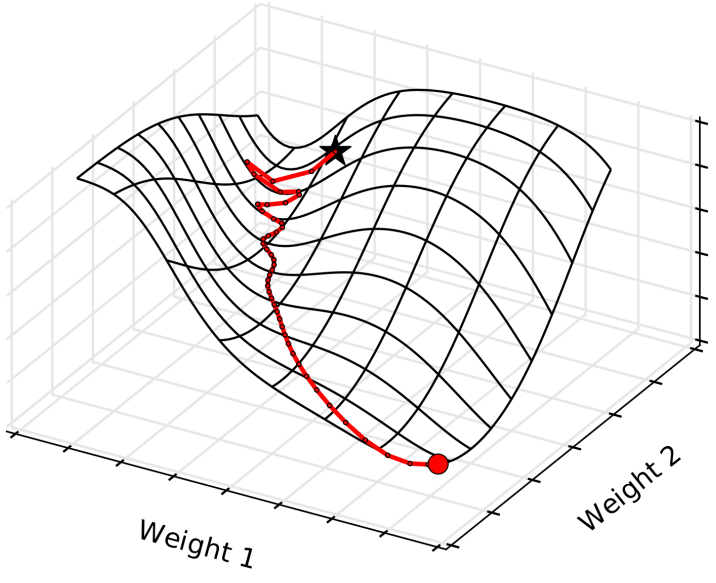
input: initial $\mathbf{w}_1$, decays $\boldsymbol{\gamma}$, learning rates $\boldsymbol{\alpha}$, loss function $L(\mathbf{w}, \boldsymbol{\theta}, t)$
initialize $\mathbf{v}_1 = \mathbf{0}$
for $t = 1$ **to** T **do**
  $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ ▷ evaluate gradient
  $\mathbf{v}_{t+1} = \gamma_t \mathbf{v}_t - (1 - \gamma_t)\mathbf{g}_t$ ▷ update velocity
  $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \mathbf{v}_t$ ▷ update position
end for
output trained parameters $\mathbf{w}_T$

Algorithm 2 Reverse-mode differentiation of SGD

input: $\mathbf{w}_T$, $\mathbf{v}_T$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$, train loss $L(\mathbf{w}, \boldsymbol{\theta}, t)$, loss $f(\mathbf{w})$
initialize $d\mathbf{v} = \mathbf{0}$, $d\boldsymbol{\theta} = \mathbf{0}$, $d\alpha_t = \mathbf{0}$, $d\gamma = \mathbf{0}$
initialize $d\mathbf{w} = \nabla_{\mathbf{w}} f(\mathbf{w}_T)$
for $t = T$ **counting down to** 1 **do**
  $d\alpha_t = d\mathbf{w}^\mathsf{T} \mathbf{v}_t$
  $\mathbf{w}_{t-1} = \mathbf{w}_t - \alpha_t \mathbf{v}_t$ } exactly reverse
  $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ } gradient descent
  $\mathbf{v}_{t-1} = [\mathbf{v}_t + (1 - \gamma_t)\mathbf{g}_t]/\gamma_t$ } operations
  $d\mathbf{v} = d\mathbf{v} + \alpha_t d\mathbf{w}$
  $d\gamma_t = d\mathbf{v}^\mathsf{T}(\mathbf{v}_t + \mathbf{g}_t)$
  $d\mathbf{w} = d\mathbf{w} - (1 - \gamma_t) d\mathbf{v} \nabla_{\mathbf{w}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
  $d\boldsymbol{\theta} = d\boldsymbol{\theta} - (1 - \gamma_t) d\mathbf{v} \nabla_{\boldsymbol{\theta}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
  $d\mathbf{v} = \gamma_t d\mathbf{v}$
end for
output gradient of $f(\mathbf{w}_T)$ w.r.t $\mathbf{w}_1$, $\mathbf{v}_1$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}$

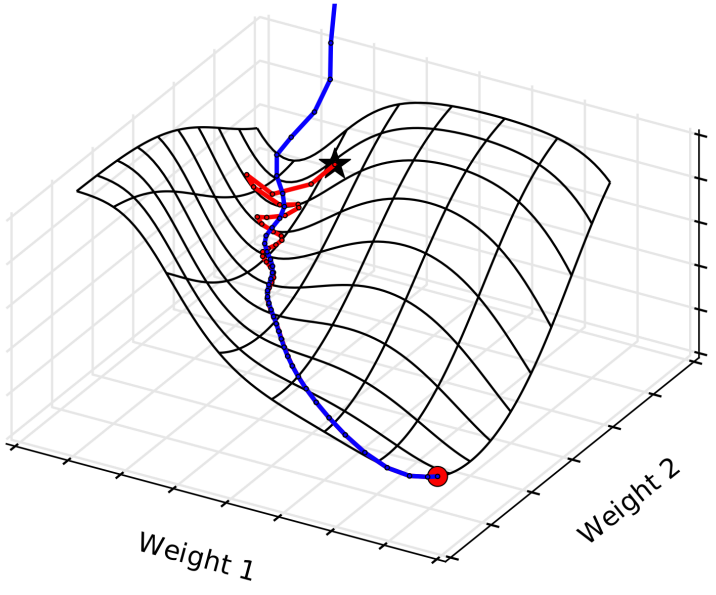
Maclaurin*, Duvenaud*, and Adams. *Gradient-based hyperparameter optimization through reversible learning*. https://arxiv.org/pdf/1502.03492.pdf ICML 2015.

Differentiating long iterations: reversibility



Maclaurin*, Duvenaud*, and Adams. *Gradient-based hyperparameter optimization through reversible learning*. https://arxiv.org/pdf/1502.03492.pdf ICML 2015.

Differentiating long iterations: reversibility



Maclaurin*, Duvenaud*, and Adams. *Gradient-based hyperparameter optimization through reversible learning*. https://arxiv.org/pdf/1502.03492.pdf ICML 2015.

Differentiating long iterations: reversibility

Algorithm 1 Stochastic gradient descent with momentum

input: initial $\mathbf{w}_1$, decays $\boldsymbol{\gamma}$, learning rates $\boldsymbol{\alpha}$, loss function $L(\mathbf{w}, \boldsymbol{\theta}, t)$
initialize $\mathbf{v}_1 = \mathbf{0}$
for $t = 1$ **to** T **do**
  $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ ▷ evaluate gradient
  $\mathbf{v}_{t+1} = \gamma_t \mathbf{v}_t - (1 - \gamma_t)\mathbf{g}_t$ ▷ update velocity
  $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \mathbf{v}_t$ ▷ update position
end for
output trained parameters $\mathbf{w}_T$

Algorithm 2 Reverse-mode differentiation of SGD

input: $\mathbf{w}_T$, $\mathbf{v}_T$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$, train loss $L(\mathbf{w}, \boldsymbol{\theta}, t)$, loss $f(\mathbf{w})$
initialize $d\mathbf{v} = \mathbf{0}$, $d\boldsymbol{\theta} = \mathbf{0}$, $d\alpha_t = \mathbf{0}$, $d\gamma = \mathbf{0}$
initialize $d\mathbf{w} = \nabla_{\mathbf{w}} f(\mathbf{w}_T)$
for $t = T$ **counting down to** 1 **do**
  $d\alpha_t = d\mathbf{w}^\mathsf{T} \mathbf{v}_t$
  $\mathbf{w}_{t-1} = \mathbf{w}_t - \alpha_t \mathbf{v}_t$ } exactly reverse
  $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ } gradient descent
  $\mathbf{v}_{t-1} = [\mathbf{v}_t + (1 - \gamma_t)\mathbf{g}_t]/\gamma_t$ } operations
  $d\mathbf{v} = d\mathbf{v} + \alpha_t d\mathbf{w}$
  $d\gamma_t = d\mathbf{v}^\mathsf{T}(\mathbf{v}_t + \mathbf{g}_t)$
  $d\mathbf{w} = d\mathbf{w} - (1 - \gamma_t) d\mathbf{v} \nabla_{\mathbf{w}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
  $d\boldsymbol{\theta} = d\boldsymbol{\theta} - (1 - \gamma_t) d\mathbf{v} \nabla_{\boldsymbol{\theta}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
  $d\mathbf{v} = \gamma_t d\mathbf{v}$
end for
output gradient of $f(\mathbf{w}_T)$ w.r.t $\mathbf{w}_1$, $\mathbf{v}_1$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}$

Maclaurin*, Duvenaud*, and Adams. *Gradient-based hyperparameter optimization through reversible learning*. https://arxiv.org/pdf/1502.03492.pdf ICML 2015.

Differentiating long iterations: reversibility

Algorithm 1 Stochastic gradient descent with momentum

input: initial $\mathbf{w}_1$, decays $\boldsymbol{\gamma}$, learning rates $\boldsymbol{\alpha}$, loss function $L(\mathbf{w}, \boldsymbol{\theta}, t)$
initialize $\mathbf{v}_1 = \mathbf{0}$
for $t = 1$ **to** T **do**
  $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ ▷ evaluate gradient
  $\mathbf{v}_{t+1} = \gamma_t \mathbf{v}_t - (1 - \gamma_t)\mathbf{g}_t$ ▷ update velocity
  $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_t \mathbf{v}_t$ ▷ update position
end for
output trained parameters $\mathbf{w}_T$

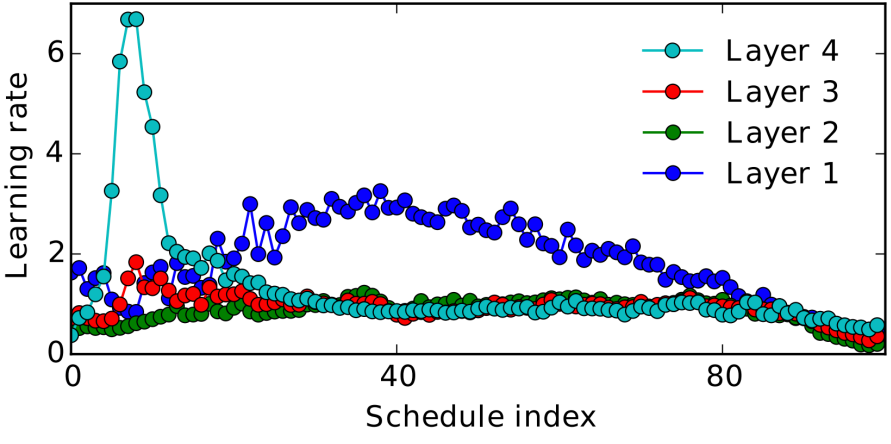
finite precision means information loss!

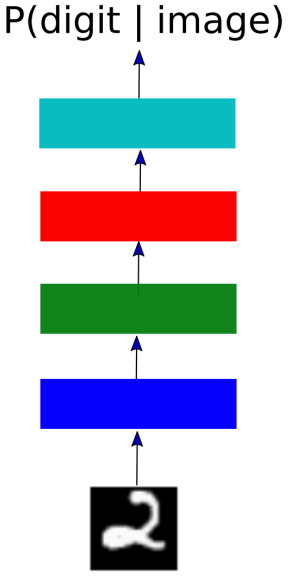
Algorithm 2 Reverse-mode differentiation of SGD

input: $\mathbf{w}_T$, $\mathbf{v}_T$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$, train loss $L(\mathbf{w}, \boldsymbol{\theta}, t)$, loss $f(\mathbf{w})$
initialize $d\mathbf{v} = \mathbf{0}$, $d\boldsymbol{\theta} = \mathbf{0}$, $d\alpha_t = \mathbf{0}$, $d\gamma = \mathbf{0}$
initialize $d\mathbf{w} = \nabla_{\mathbf{w}} f(\mathbf{w}_T)$
for $t = T$ **counting down to** 1 **do**
  $d\alpha_t = d\mathbf{w}^\mathsf{T} \mathbf{v}_t$
  $\mathbf{w}_{t-1} = \mathbf{w}_t - \alpha_t \mathbf{v}_t$ } exactly reverse
  $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$ } gradient descent
  $\mathbf{v}_{t-1} = [\mathbf{v}_t + (1 - \gamma_t)\mathbf{g}_t]/\gamma_t$ } operations
  $d\mathbf{v} = d\mathbf{v} + \alpha_t d\mathbf{w}$
  $d\gamma_t = d\mathbf{v}^\mathsf{T}(\mathbf{v}_t + \mathbf{g}_t)$
  $d\mathbf{w} = d\mathbf{w} - (1 - \gamma_t) d\mathbf{v} \nabla_{\mathbf{w}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
  $d\boldsymbol{\theta} = d\boldsymbol{\theta} - (1 - \gamma_t) d\mathbf{v} \nabla_{\boldsymbol{\theta}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
  $d\mathbf{v} = \gamma_t d\mathbf{v}$
end for
output gradient of $f(\mathbf{w}_T)$ w.r.t $\mathbf{w}_1$, $\mathbf{v}_1$, $\boldsymbol{\gamma}$, $\boldsymbol{\alpha}$ and $\boldsymbol{\theta}$

Maclaurin*, Duvenaud*, and Adams. *Gradient-based hyperparameter optimization through reversible learning*. https://arxiv.org/pdf/1502.03492.pdf ICML 2015.

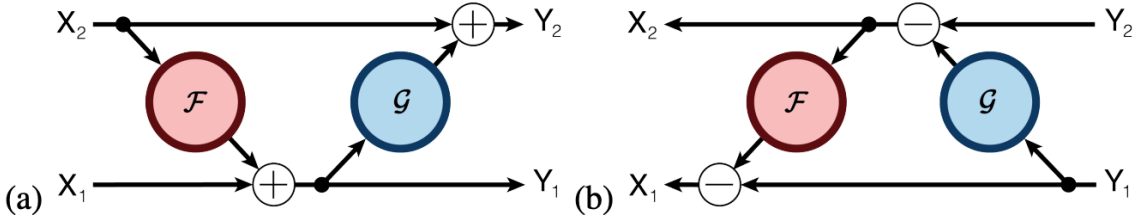
Differentiating long iterations: reversibility





Maclaurin*, Duvenaud*, and Adams. *Gradient-based hyperparameter optimization through reversible learning*. https://arxiv.org/pdf/1502.03492.pdf ICML 2015.

Differentiating long iterations: reversibility



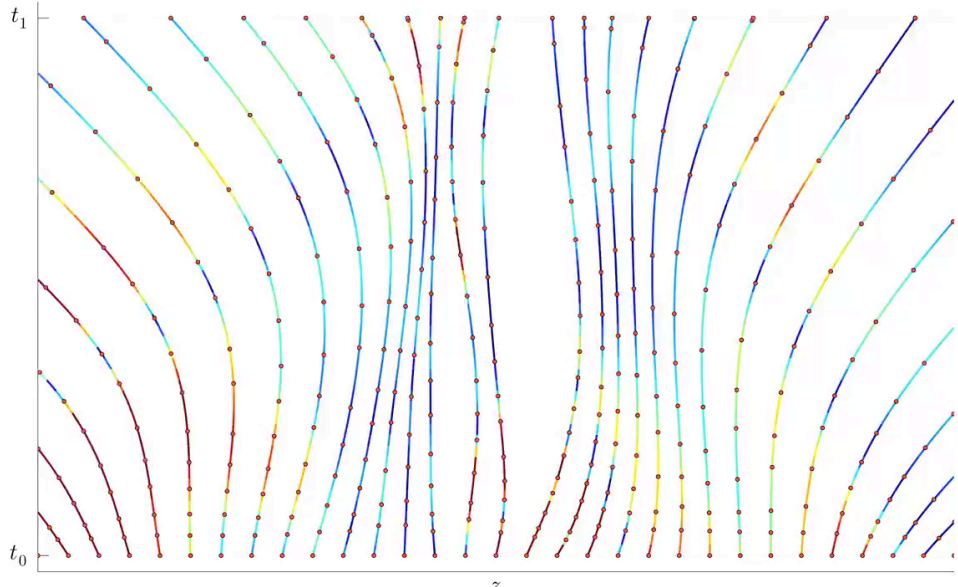

```
from jax import reversible
```


Gomez, Ren, Urtasun, and Grosse. *The reversible residual network: backpropagation without storing activations*. https://arxiv.org/pdf/1707.04585.pdf NeurIPS 2017.

Kitaev, Kaiser, and Levskaya. *Reformer: the efficient Transformer*. https://arxiv.org/pdf/2001.04451.pdf ICLR 2019.

Very high-order AD with `jet`

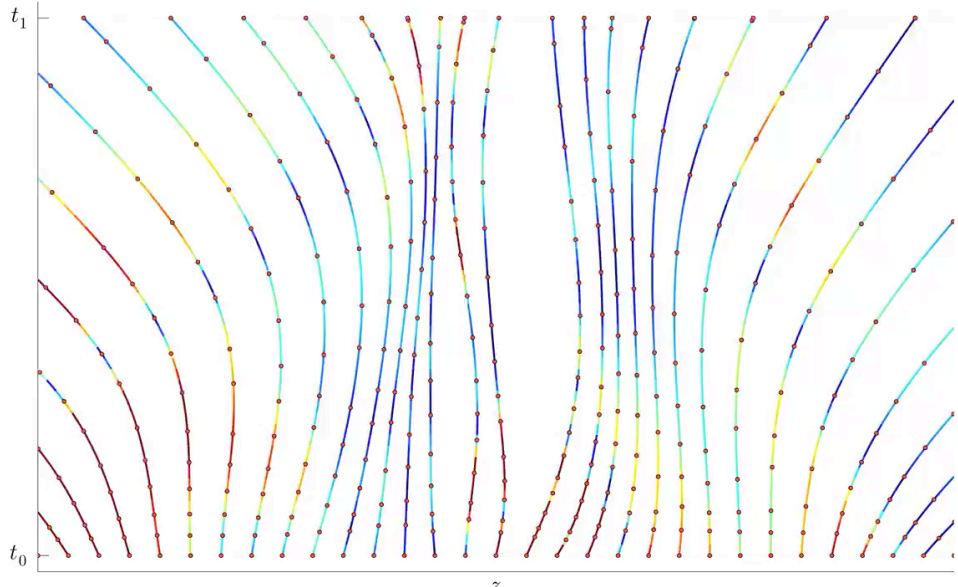
Neural ODEs can be slow at test time...



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 `jet`

$$f = g \circ h$$

$$\text{given } (h(x),\ \partial h(x)[v])$$
$$\text{compute } (f(x),\ \partial f(x)[v])$$

$$\text{using } g \text{ and } \partial g$$

Very high-order AD with `jet`

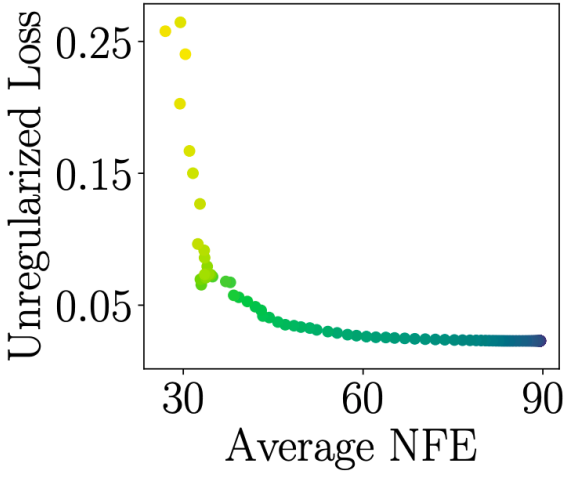
$$f = g \circ h$$

$$\text{given } (h(x),\ \partial h(x)[v],\ \tfrac{1}{2}\partial^2 h(x)[v,v],\ \cdots,\ \tfrac{1}{K!}\partial^K h(x)[v,\ldots,v])$$

$$\text{compute } (f(x),\ \partial f(x)[v],\ \tfrac{1}{2}\partial^2 f(x)[v,v],\ \cdots,\ \tfrac{1}{K!}\partial^K f(x)[v,\ldots,v])$$

$$\text{using } g,\ \partial g,\ \partial^2 g,\ \ldots,\ \text{and } \partial^K g$$

Very high-order AD with `jet`



(a) MNIST Classification

Kelly*, Bettencourt*, **MJJ**, and Duvenaud. *Learning differential equations that are easy to solve*. https://arxiv.org/pdf/2007.04504.pdf NeurIPS 2020, to appear.

Very high-order AD with `jet`

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

Thanks!