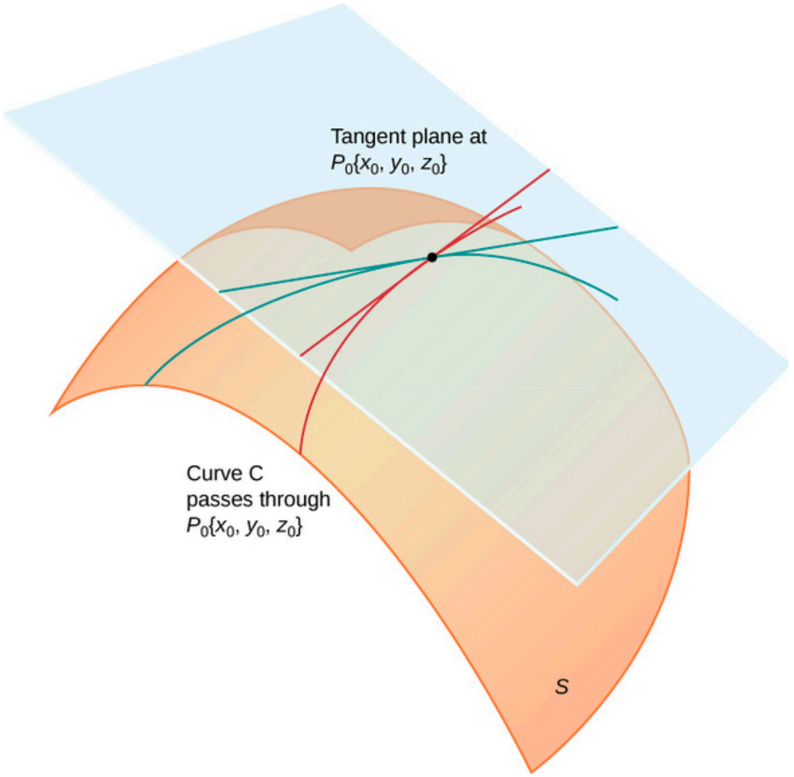
Reverse mode automatic differentiation

UCSD CSE 291 Differentiable Programming

Tzu-Mao Li

Recall: forward mode

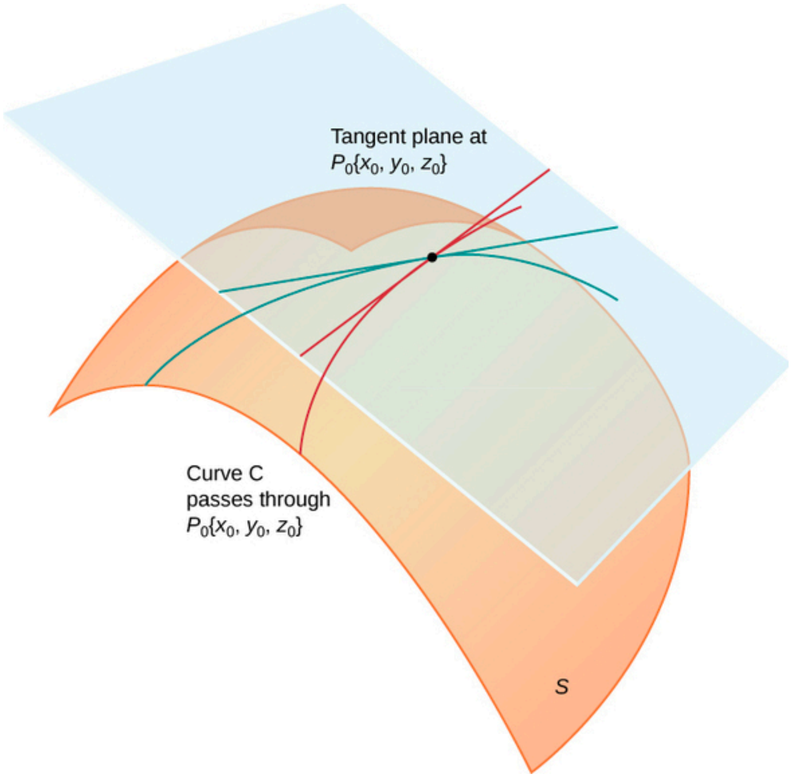
$$f(x_0 + dx) \approx f(x_0) + Df(x_0)(dx)$$



Recall: forward mode

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

$$f(x_0 + dx) \approx f(x_0) + Df(x_0)(dx)$$



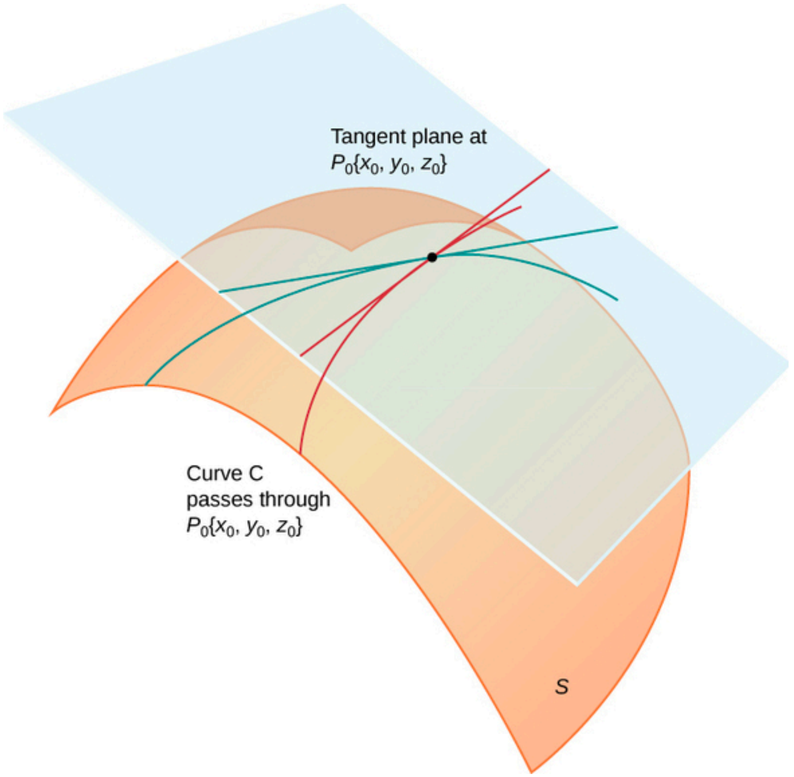
Recall: forward mode

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

$$f(x_0 + dx) \approx f(x_0) + Df(x_0)(dx)$$

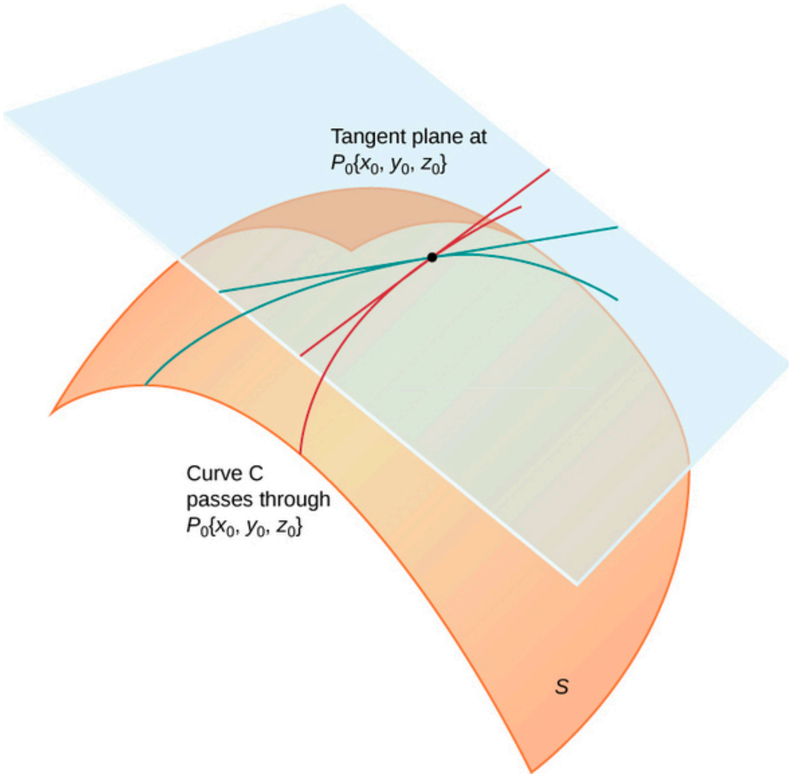
$$Df(x_0) = \text{Jacobian}$$

$$Df(x_0) : \mathbb{R}^n \to \mathbb{R}^m \text{ (linear)}$$



Recall: forward mode

$$f(x_0 + dx) \approx f(x_0) + Df(x_0)(dx)$$



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

$Df(x_0) =$ Jacobian

$Df(x_0) : \mathbb{R}^n \to \mathbb{R}^m$ (linear)

$Df(x_0)(dx) =$ total derivative

$$Df : \mathbb{R}^n \to \left(\mathbb{R}^n \to \mathbb{R}^m\right)$$

Recall:

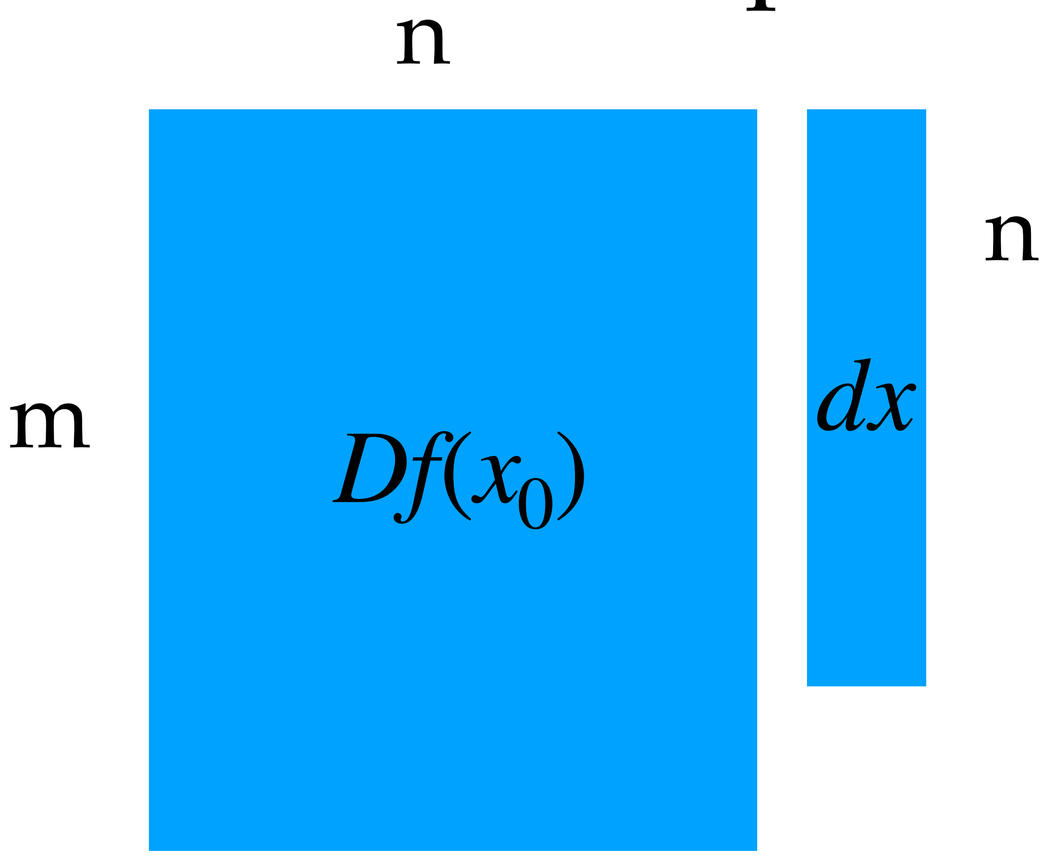
total derivative = Jacobian-vector product

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

$Df(x_0)$ = Jacobian

$Df(x_0) : \mathbb{R}^n \to \mathbb{R}^m$ (linear)

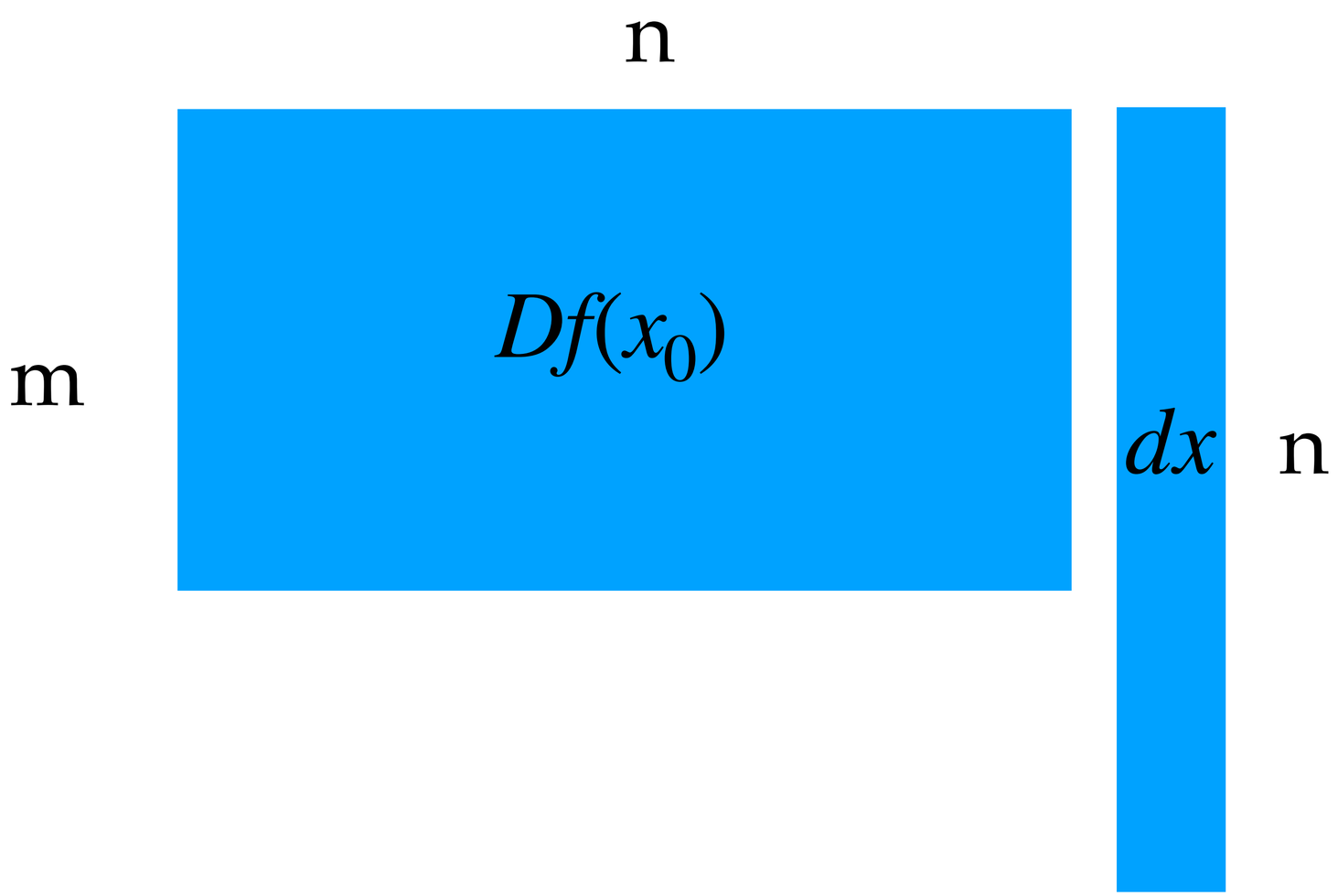
$Df(x_0)(dx)$ = total derivative



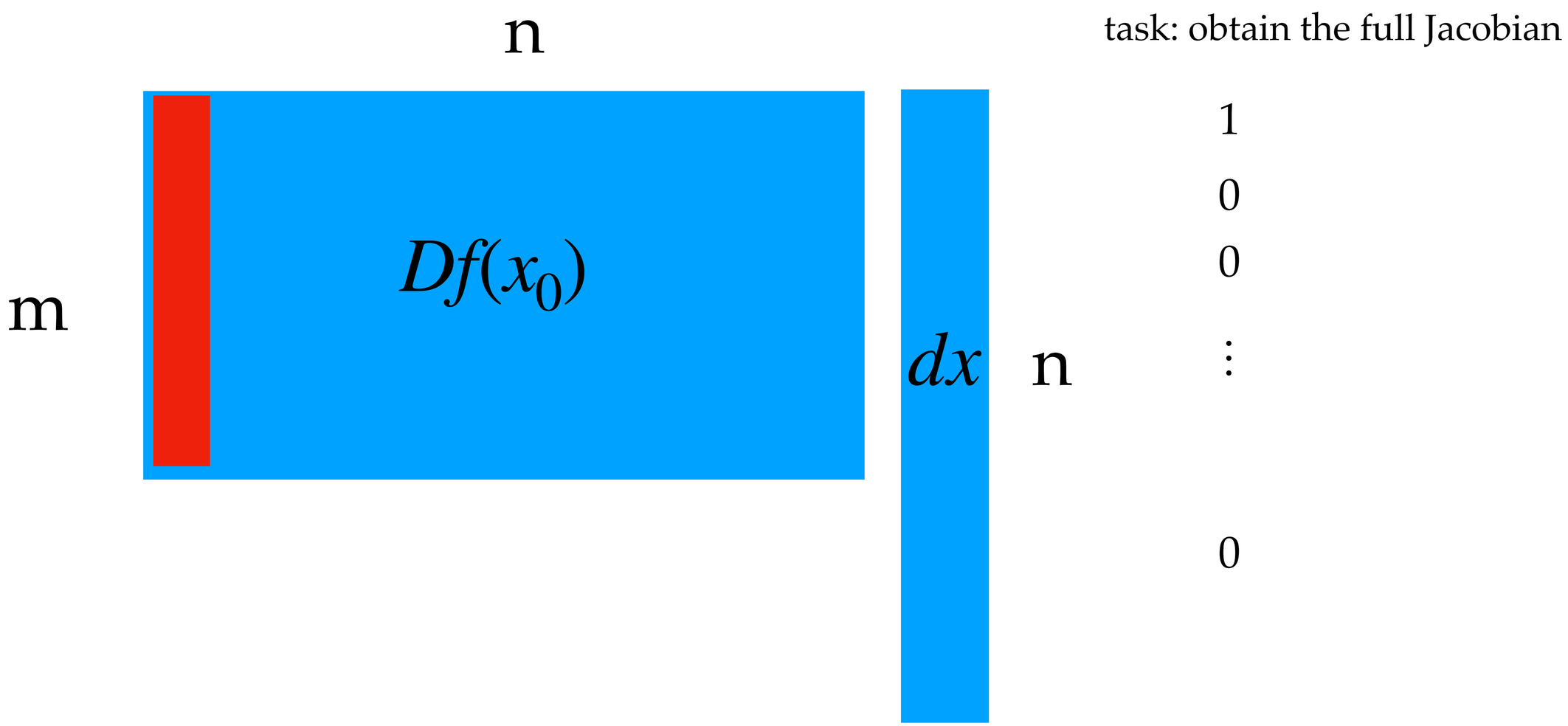
forward mode outputs m numbers each time

Forward mode is inefficient when the input dimension is large

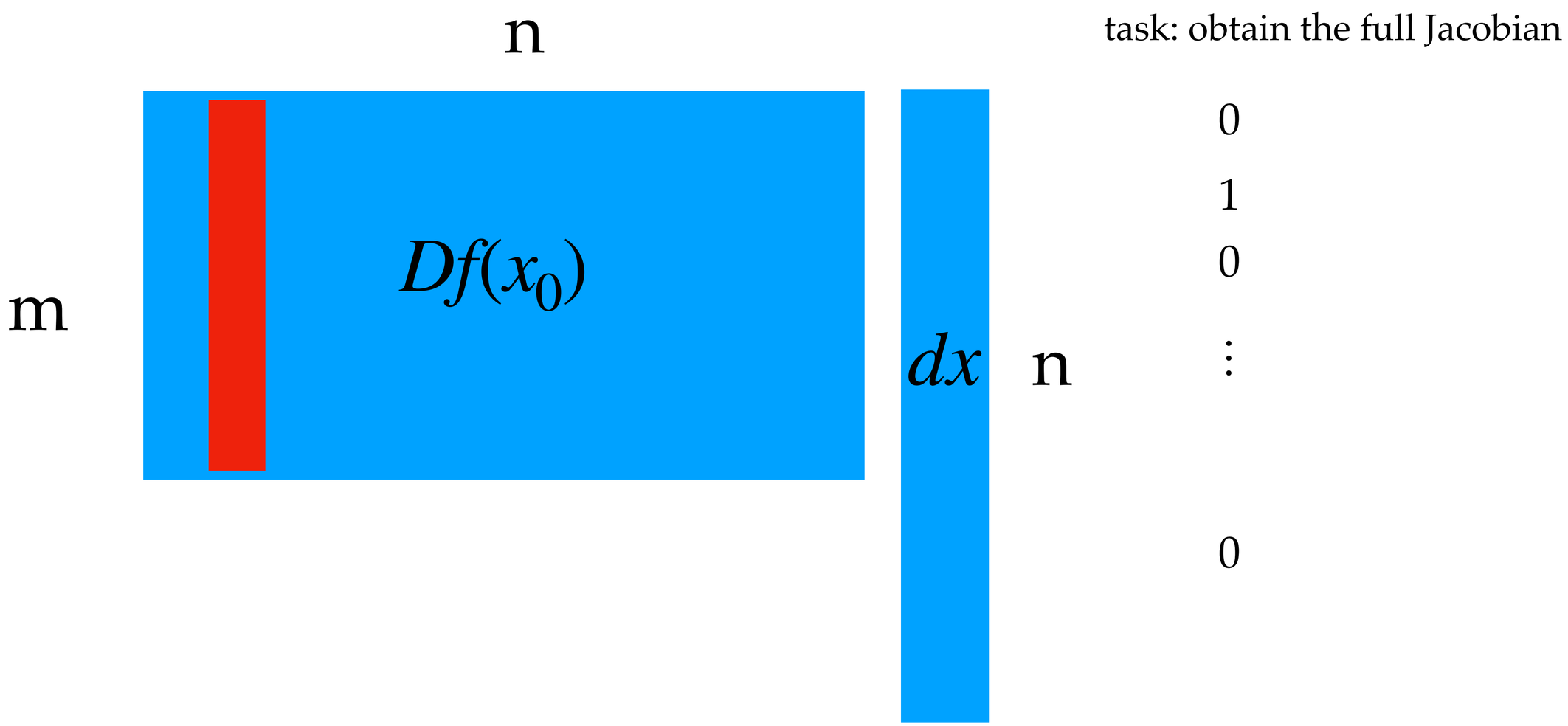
task: obtain the full Jacobian



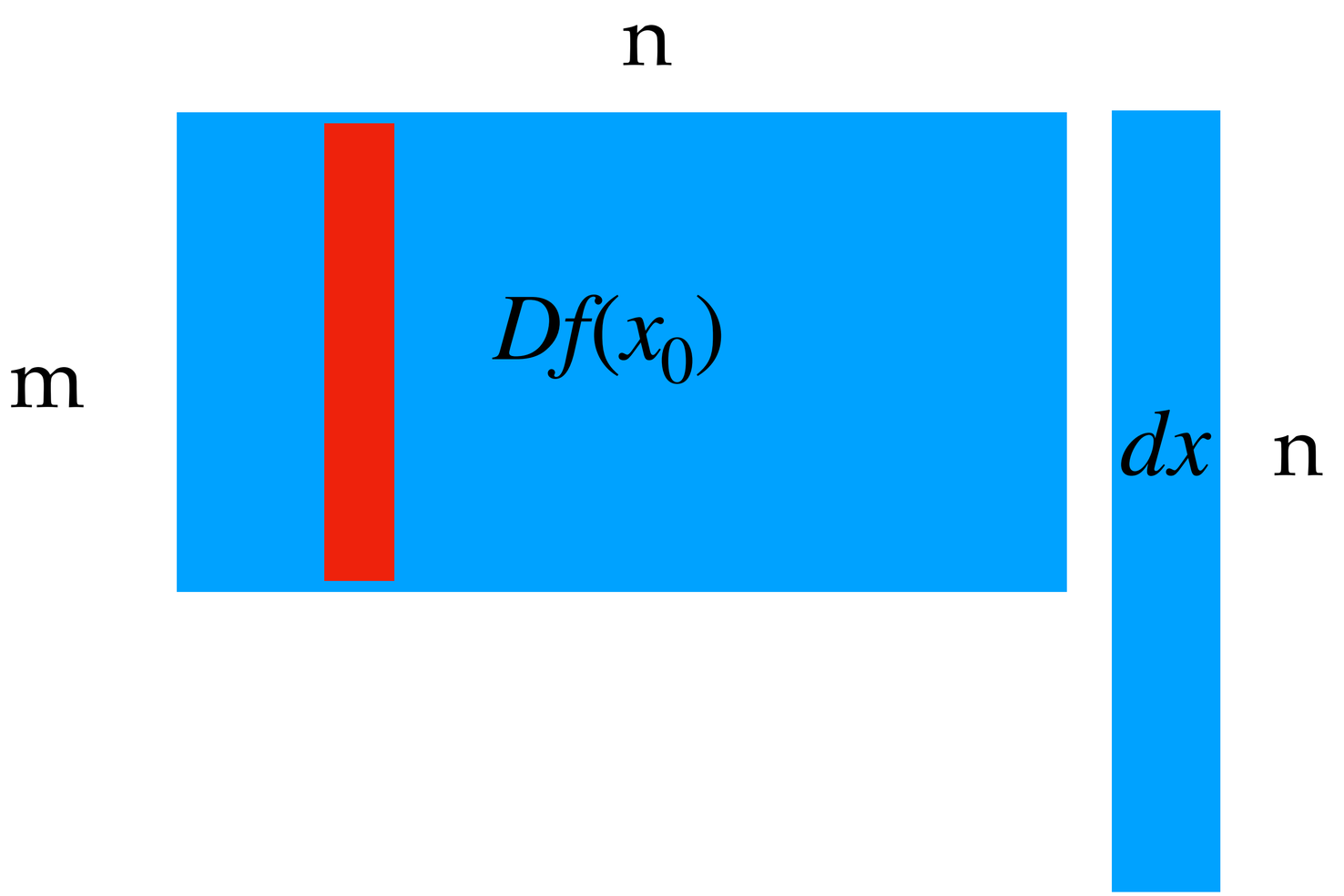
Forward mode is inefficient when the input dimension is large



Forward mode is inefficient when the input dimension is large



Forward mode is inefficient when the input dimension is large



task: obtain the full Jacobian

$$\begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

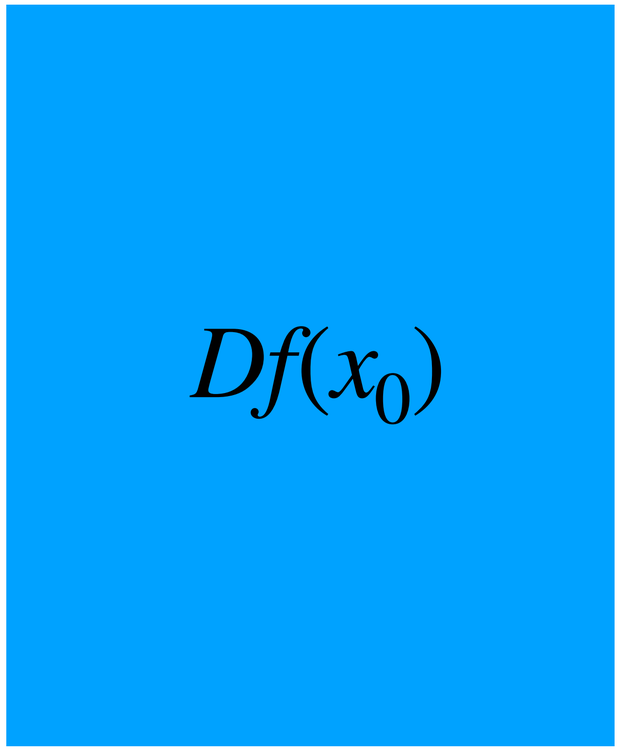
getting the full Jacobian takes n executions!

Reverse mode automatic differentiation to the rescue

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

n

m



forward mode takes **n** executions to compute the full Jacobian

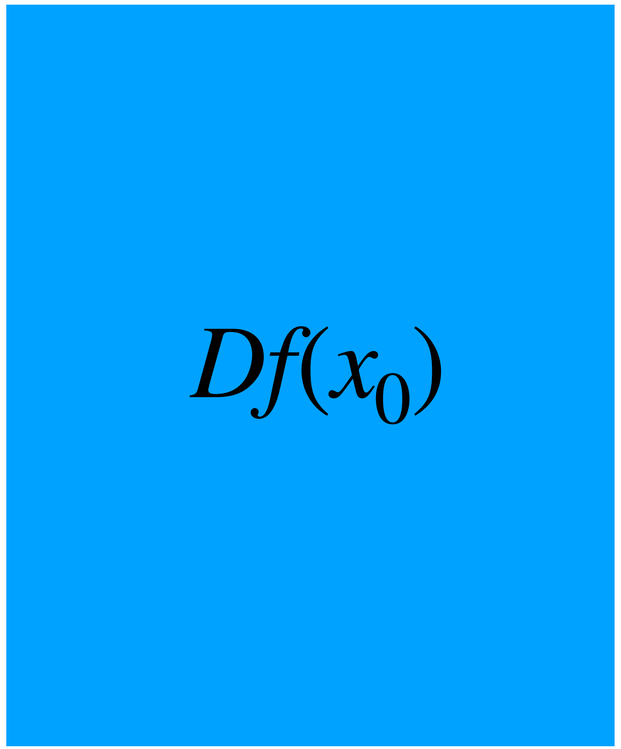
reverse mode takes **m** executions to compute the full Jacobian

Reverse mode automatic differentiation to the rescue

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

n

m



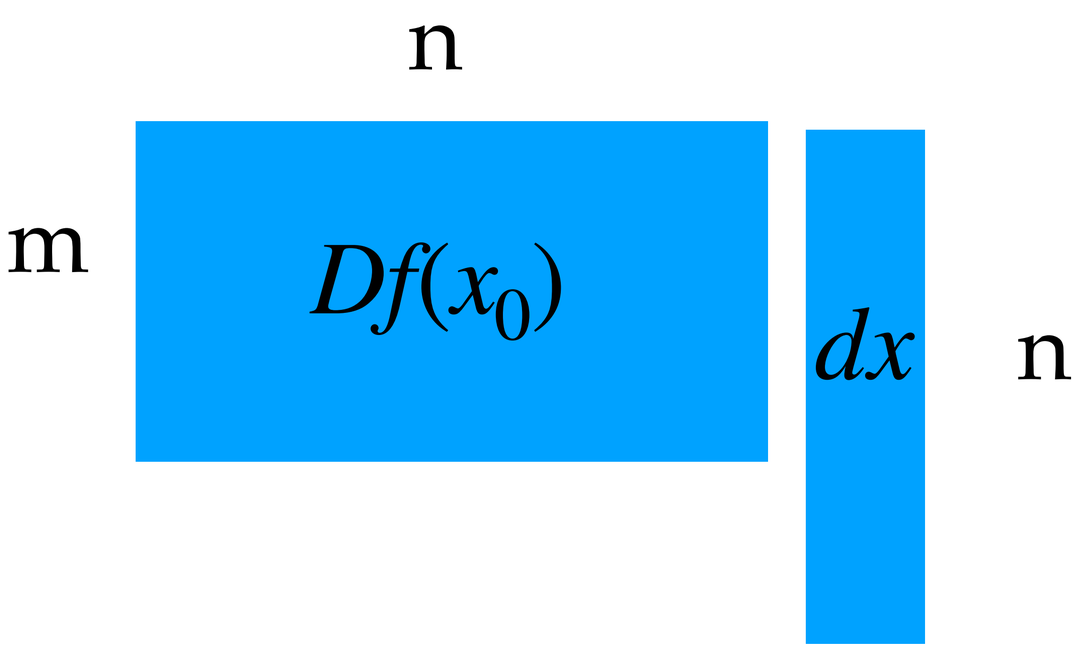
forward mode takes **n** executions to compute the full Jacobian

reverse mode takes **m** executions to compute the full Jacobian

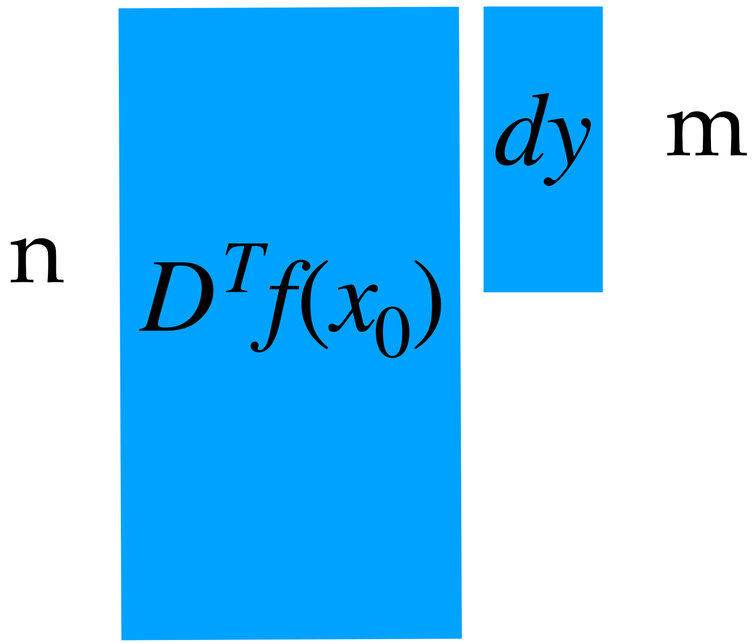
m=1 -> reverse mode computes the gradient of a function in one pass!!

Idea of reverse mode: transpose the Jacobian matrix

forward mode

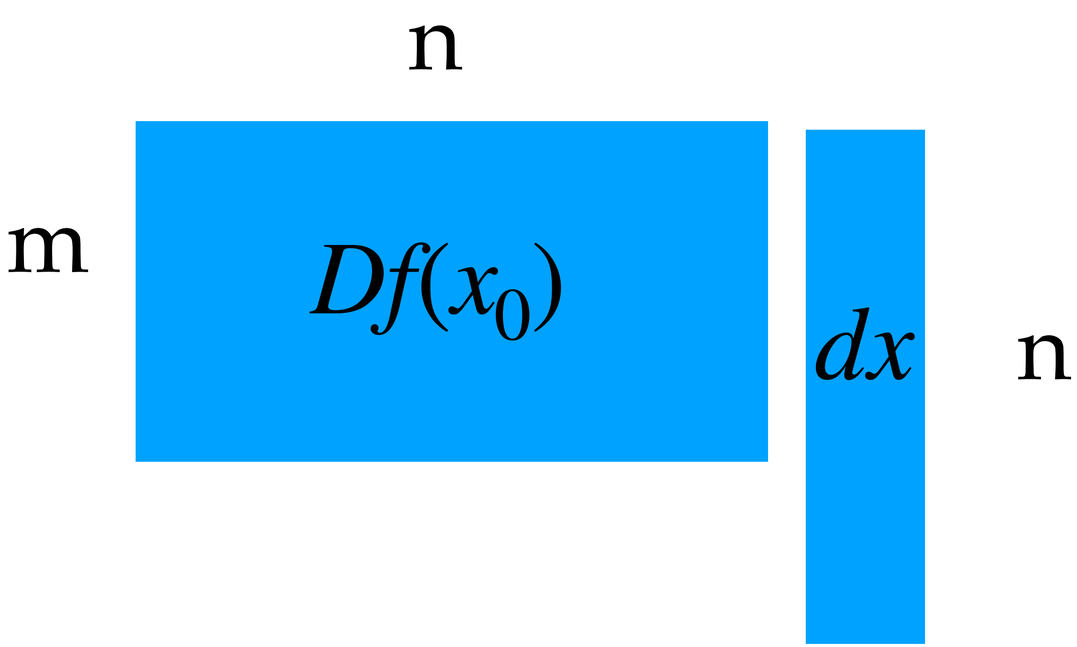


reverse mode



Idea of reverse mode: transpose the Jacobian matrix

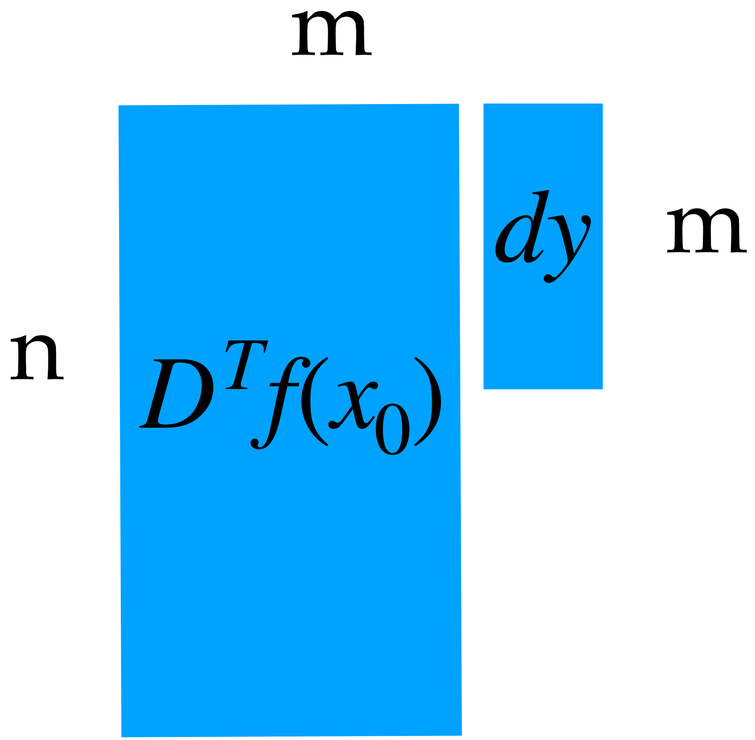
forward mode



outputs **m** number at once

n executions to compute the full Jacobian

reverse mode



outputs **n** number at once

m executions to compute the full Jacobian

Idea of reverse mode: transpose the Jacobian matrix

$$Df(x_0) = \text{Jacobian}$$

$$Df(x_0) : \mathbb{R}^n \to \mathbb{R}^m \text{ (linear)}$$

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

Idea of reverse mode: transpose the Jacobian matrix

$$Df(x_0) = \text{Jacobian}$$

$$Df(x_0) : \mathbb{R}^n \to \mathbb{R}^m \text{ (linear)}$$

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

$$D^T f(x_0) = \text{transposed Jacobian}$$

$$D^T f(x_0) : \mathbb{R}^m \to \mathbb{R}^n \text{ (linear)}$$

Idea of reverse mode: transpose the Jacobian matrix

$$Df(x_0) = \text{Jacobian}$$

$$Df(x_0) : \mathbb{R}^n \to \mathbb{R}^m \text{ (linear)}$$

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

$$D^T f(x_0) = \text{transposed Jacobian}$$

$$D^T f(x_0) : \mathbb{R}^m \to \mathbb{R}^n \text{ (linear)}$$

$$Df(x_0)(dx) = \text{total derivative}$$

$$D^T f(x_0)(dy) = \text{adjoint derivative}$$

Let's look at an example

forward mode

```
f(x, y):
   a = g(x, y)
   b = h(a)
   return b
```

```
Df(x, dx, y, dy):
   a, da = Dg(x, dx, y, dy)
   b, db = Dh(a, da)
   return b, db
```

Let's look at an example

reverse mode

```
f(x, y):
   a = g(x, y)
   b = h(a)
   return b
```

```
DTf(x, y, db):
      …
      return dx, dy
```

Let's look at an example

```
f(x, y):
   a = g(x, y)
   b = h(a)
   return b
```

```
DTf(x, y, db):
   …
   return dx, dy
```

```
DTg(x, y, da) -> dx, dy
```

```
DTh(a, db) -> da
```

Let's look at an example

```
f(x, y):
   a = g(x, y)
   b = h(a)
   return b
```

```
DTf(x, y, db):

   …
   dx, dy = DTg(x, y, da)
   return dx, dy
```

```
DTh(a, db) -> da
```

Let's look at an example

```
f(x, y):
   a = g(x, y)
   b = h(a)
   return b
```

```
DTf(x, y, db):

   …
   da = DTh(a, db)
   dx, dy = DTg(x, y, da)
   return dx, dy
```

Let's look at an example

```
f(x, y):
   a = g(x, y)
   b = h(a)
   return b
```

```
DTf(x, y, db):

   …
   b = h(a)
   da = DTh(a, db)
   dx, dy = DTg(x, y, da)
   return dx, dy
```

Let's look at an example

```
f(x, y):
   a = g(x, y)
   b = h(a)
   return b
```

```
DTf(x, y, db):
   a = g(x, y)
   b = h(a)
   da = DTh(a, db)
   dx, dy = DTg(x, y, da)
   return dx, dy
```

Let's look at an example

```
f(x, y):
   a = g(x, y)
   b = h(a)
   return b
```

```
DTf(x, y, db):
   a = g(x, y)
   b = h(a)
   da = DTh(a, db)
   dx, dy = DTg(x, y, da)
   return dx, dy
```

reverse mode automatic differentiation

Let's look at an example

```
f(x, y):
    a = g(x, y)
    b = h(a)
    return b
```

```
DTf(x, y, db):
    a = g(x, y)
    b = h(a)
    da = DTh(a, db)
    dx, dy = DTg(x, y, da)
    return dx, dy
```

execute the primal code in normal order

Let's look at an example

```
f(x, y):
   a = g(x, y)
   b = h(a)
   return b
```

```
DTf(x, y, db):
   a = g(x, y)
   b = h(a)
   da = DTh(a, db)
   dx, dy = DTg(x, y, da)
   return dx, dy
```

execute the adjoints in reverse

The inventor of reverse mode autodiff: Seppo Linnainmaa

(his master thesis!)

1970

Seppo Linnainmaa

ALGORITMIN KUMULATIIVINEN PYÖRISTYSVIRHE

YKSITTÄISTEN PYÖRISTYSVIRHEIDEN TAYLOR-KEHITELMÄNÄ

Pro gradu-tutkielma · ohjaaja professori M.Tienari

"The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors"

See Griewank's article for more history

Who Invented the Reverse Mode of Differentiation?

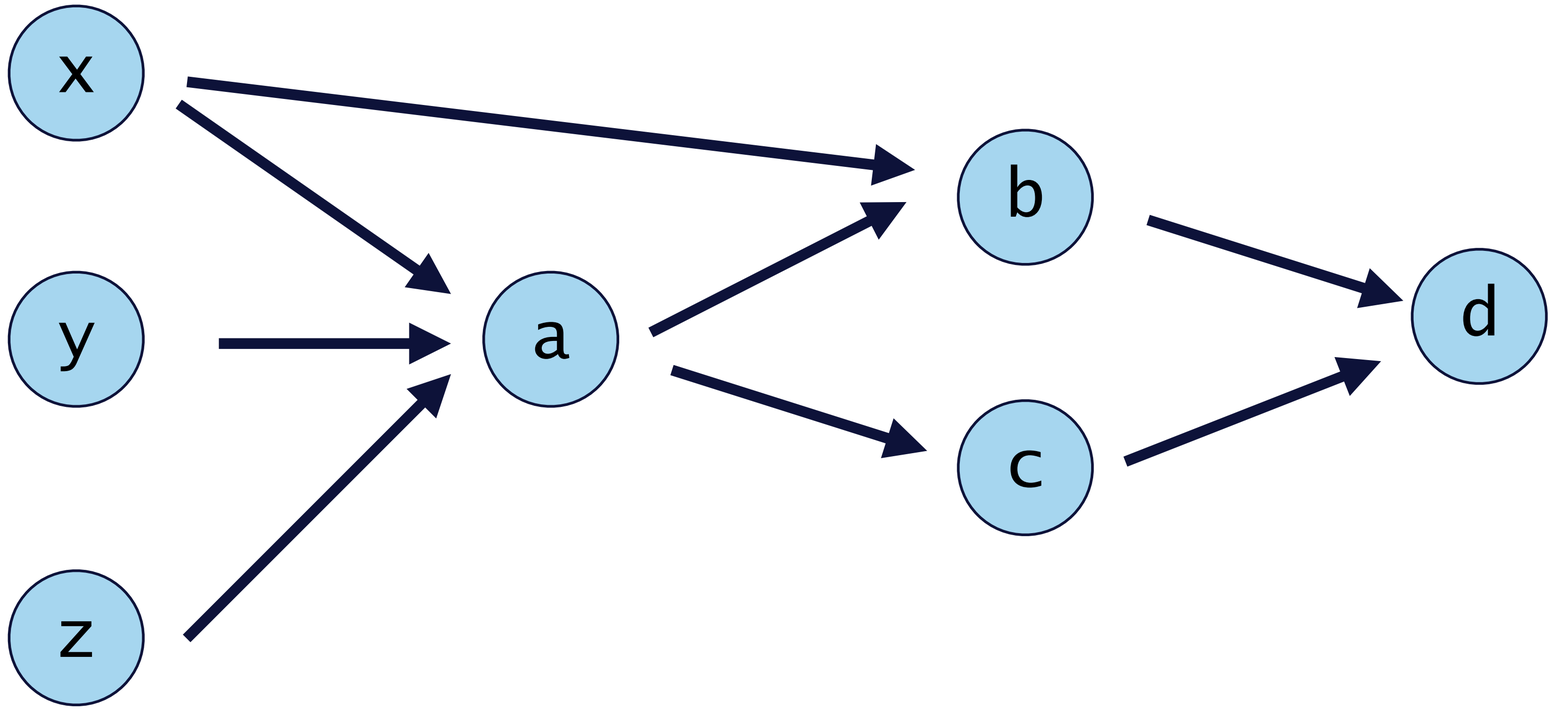
Andreas Griewank

2010 Mathematics Subject Classification: 05C85, 49M99, 65D25, 68Q17
Keywords and Phrases: Adjoints, gradient evaluation, round-off estimation, program reversal

https://www.math.uni-bielefeld.de/documenta/vol-ismp/52_griewank-andreas-b.pdf

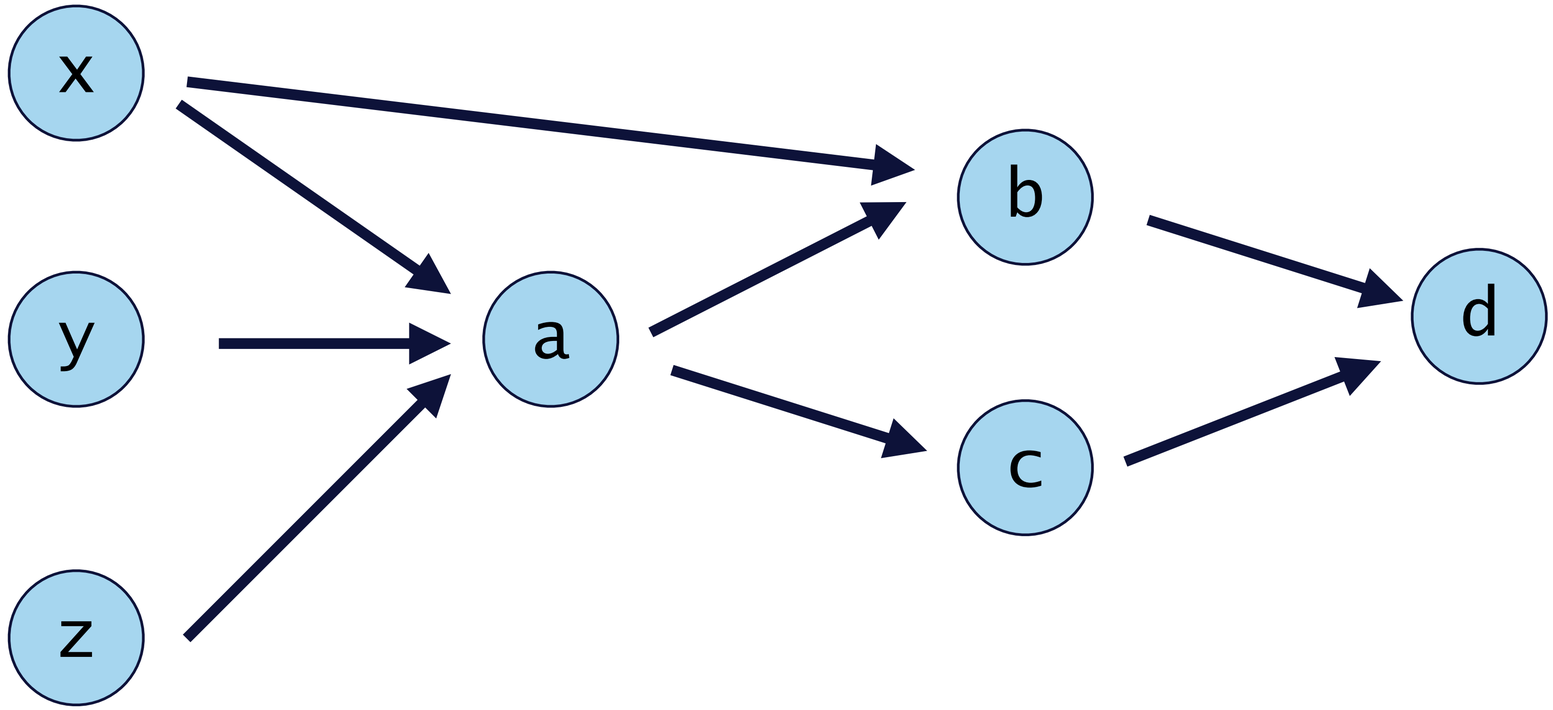
Another perspective: computational graph

```
f(x, y, z):
    a = g(x, y, z)
    b = h(x, a)
    c = i(a)
    d = j(b, c)
    return d
```



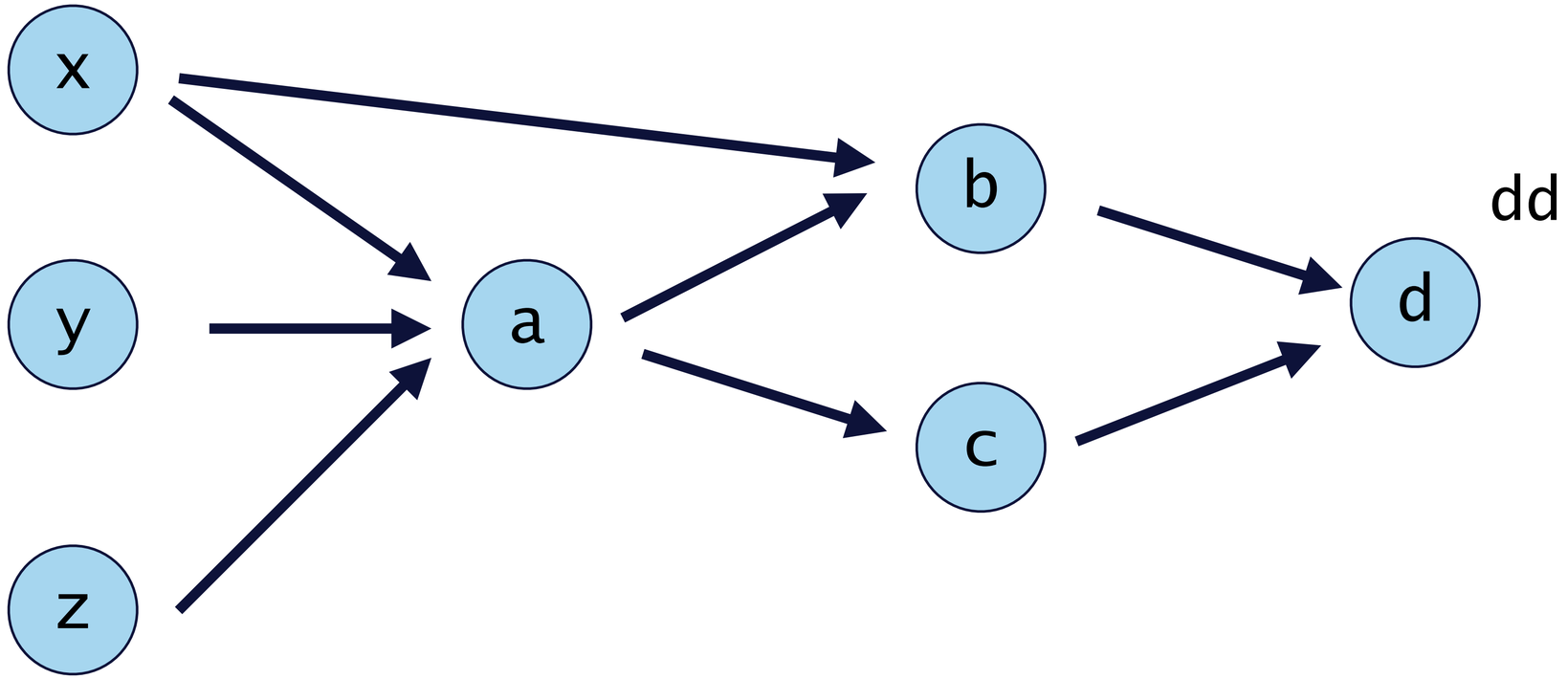
Another perspective: computational graph

```
DTf(x, y, z, dd):
   a = g(x, y, z)
   b = h(x, a)
   c = i(a)
   d = j(b, c)
   …
   return dx, dy, dz
```



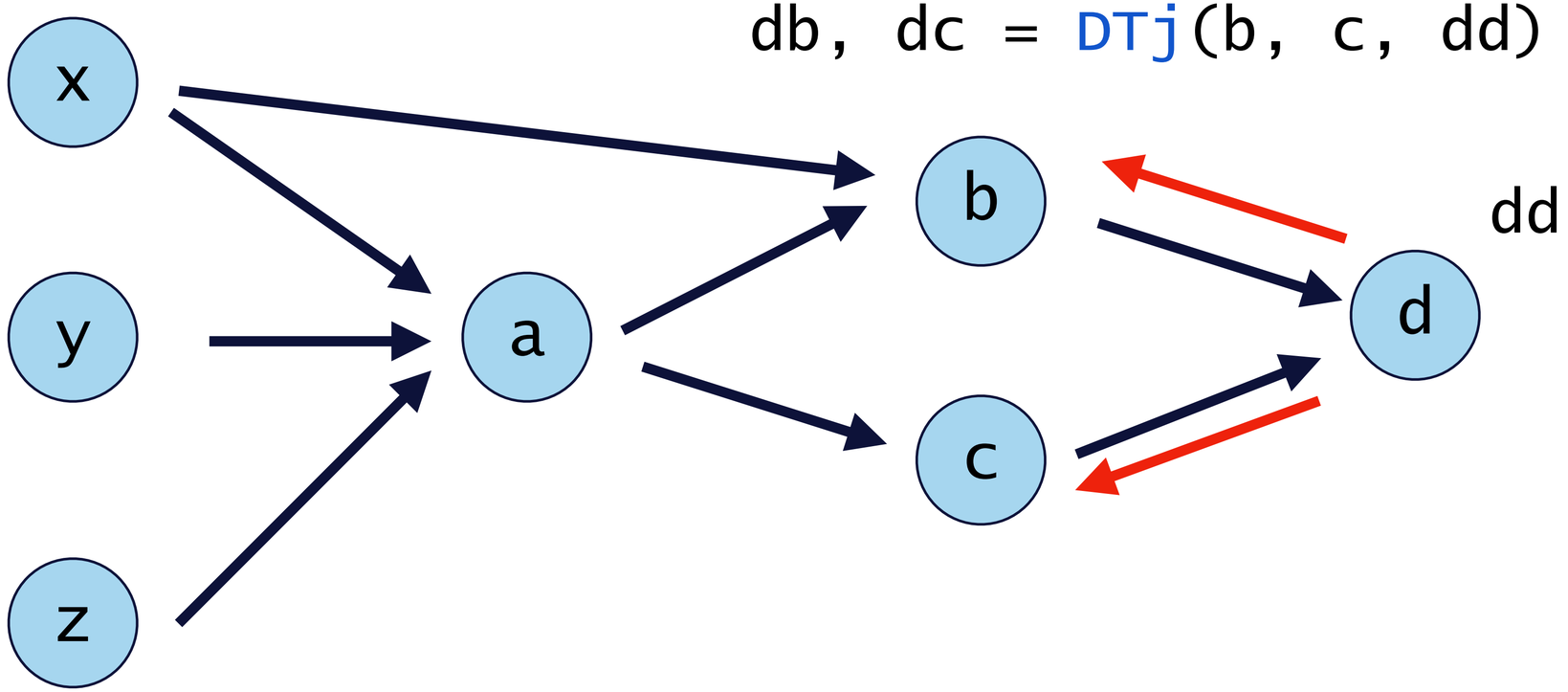
Another perspective: computational graph

```
DTf(x, y, z, dd):
   a = g(x, y, z)
   b = h(x, a)
   c = i(a)
   d = j(b, c)
   …
   return dx, dy, dz
```



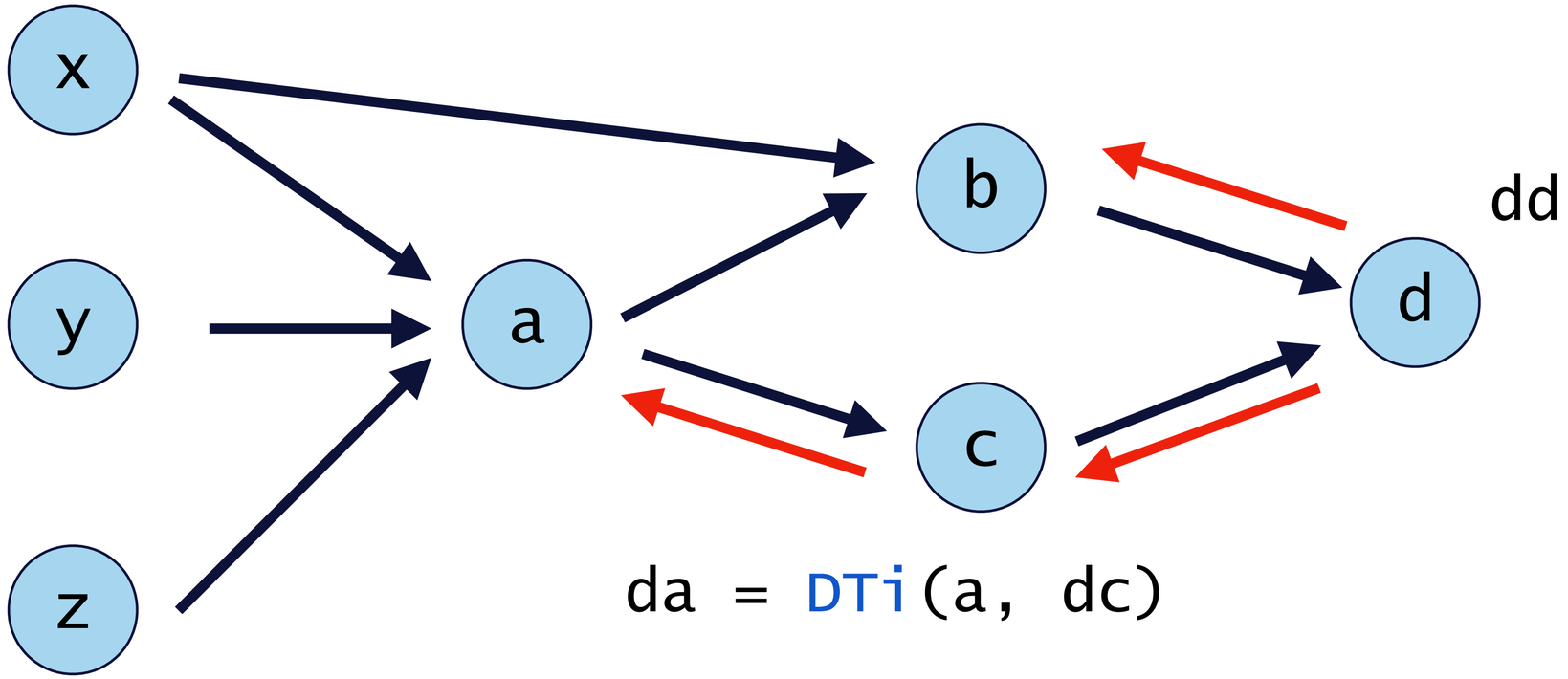
Another perspective: computational graph

```
DTf(x, y, z, dd):
   a = g(x, y, z)
   b = h(x, a)
   c = i(a)
   d = j(b, c)
   db,dc=DTj(b,c,dd)
   …
   return dx, dy, dz
```



Another perspective: computational graph

```
DTf(x, y, z, dd):
   a = g(x, y, z)
   b = h(x, a)
   c = i(a)
   d = j(b, c)
   db,dc=DTj(b,c,dd)
   da = DTi(a,dc)
   …
   return dx, dy, dz
```

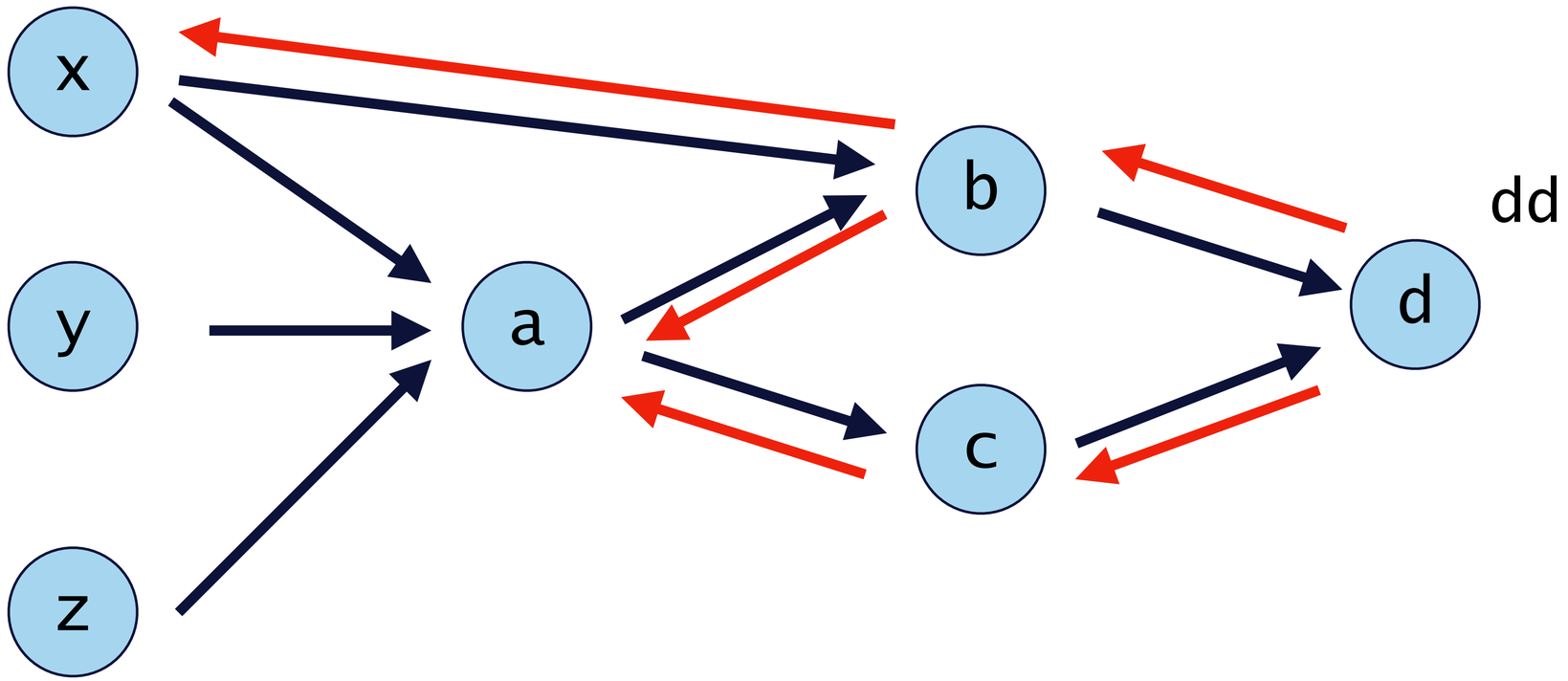


Another perspective: computational graph

```
DTf(x, y, z, dd):
   a = g(x, y, z)
   b = h(x, a)
   c = i(a)
   d = j(b, c)
   db,dc=DTj(b,c,dd)
   da = DTi(a,dc)
   dx,da+=DTh(x,a,db)
   …
   return dx, dy, dz
```

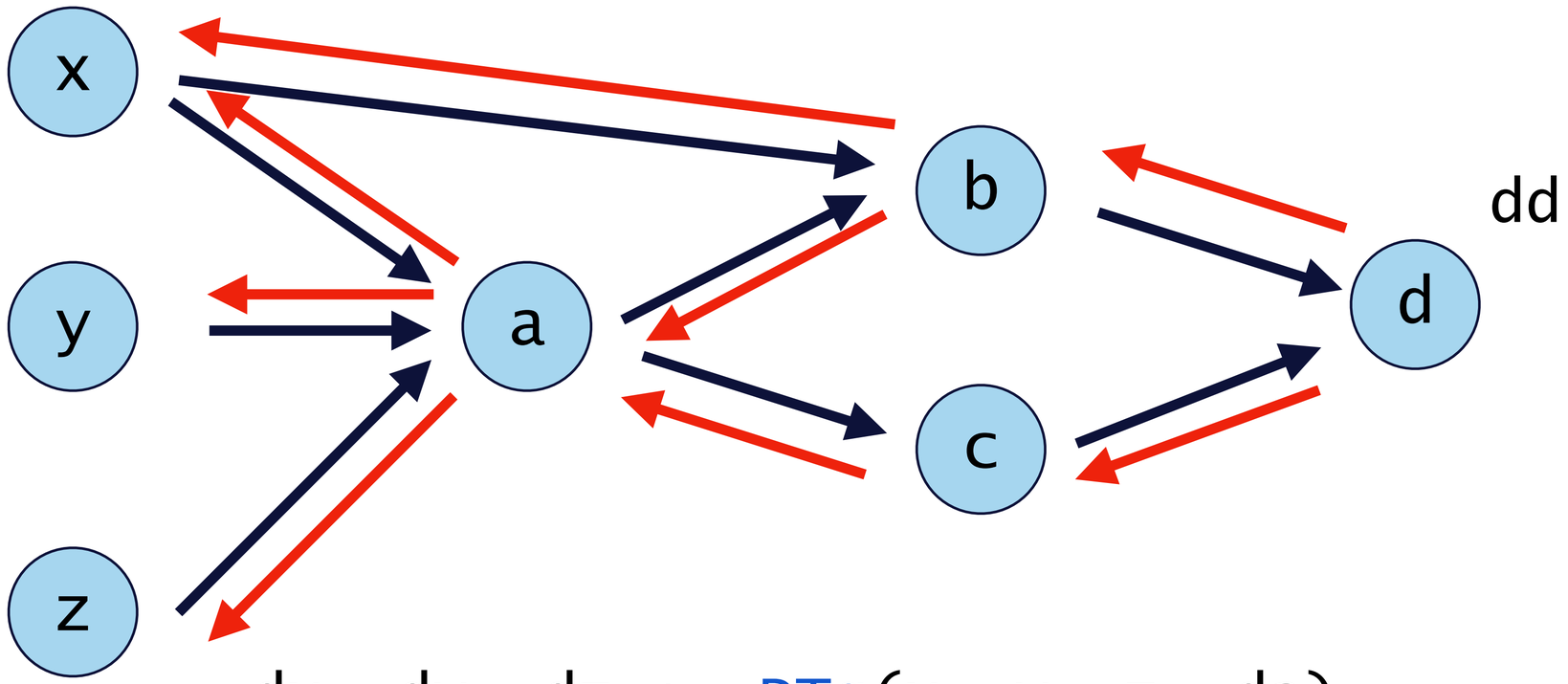
Quiz: why +=?

```
dx, da += DTh(x, a, db)
```



Another perspective: computational graph

```
DTf(x, y, z, dd):
   a = g(x, y, z)
   b = h(x, a)
   c = i(a)
   d = j(b, c)
   db,dc=DTj(b,c,dd)
   da = DTi(a,dc)
   dx,da+=DTh(x,a,db)
   dx,dy,dz+=DTg(…)
   return dx, dy, dz
```

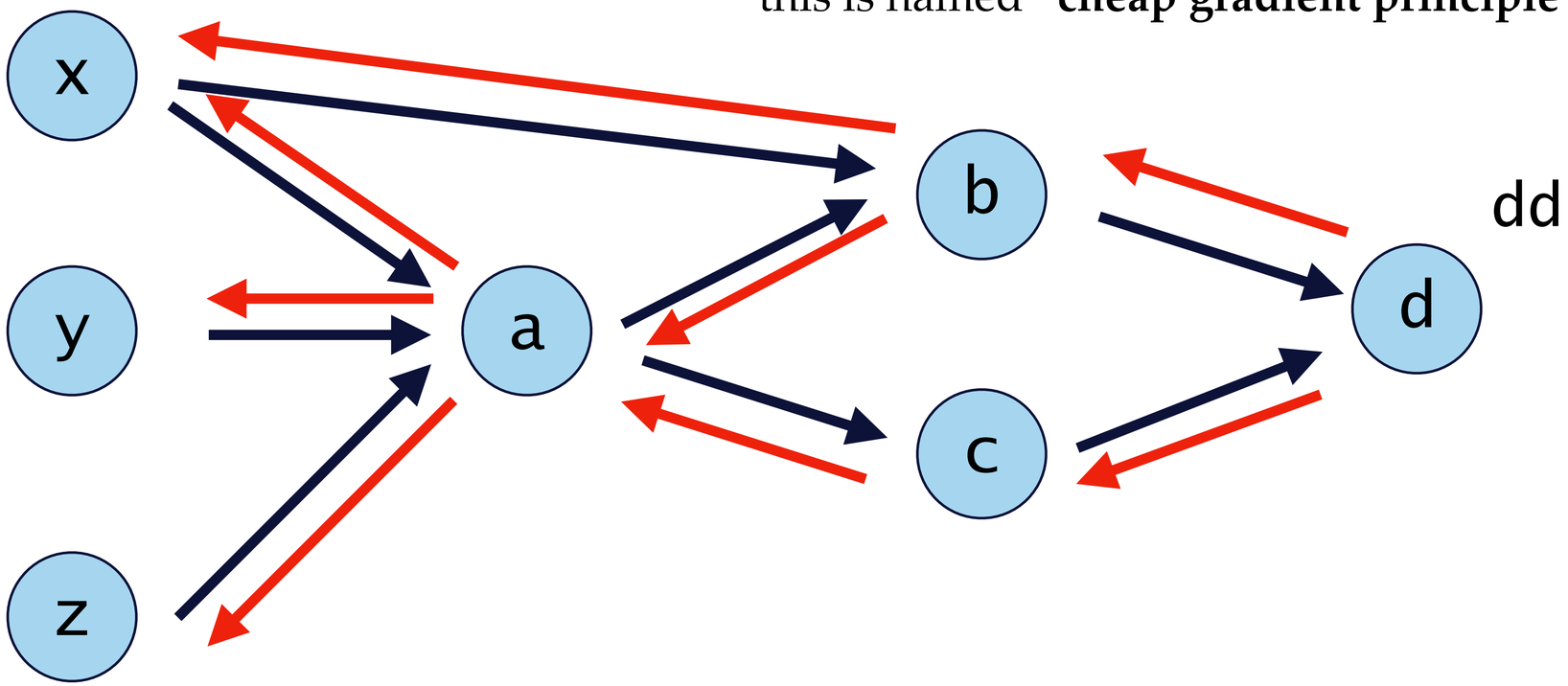



```
dx, dy, dz += DTg(x, y, z, da)
```

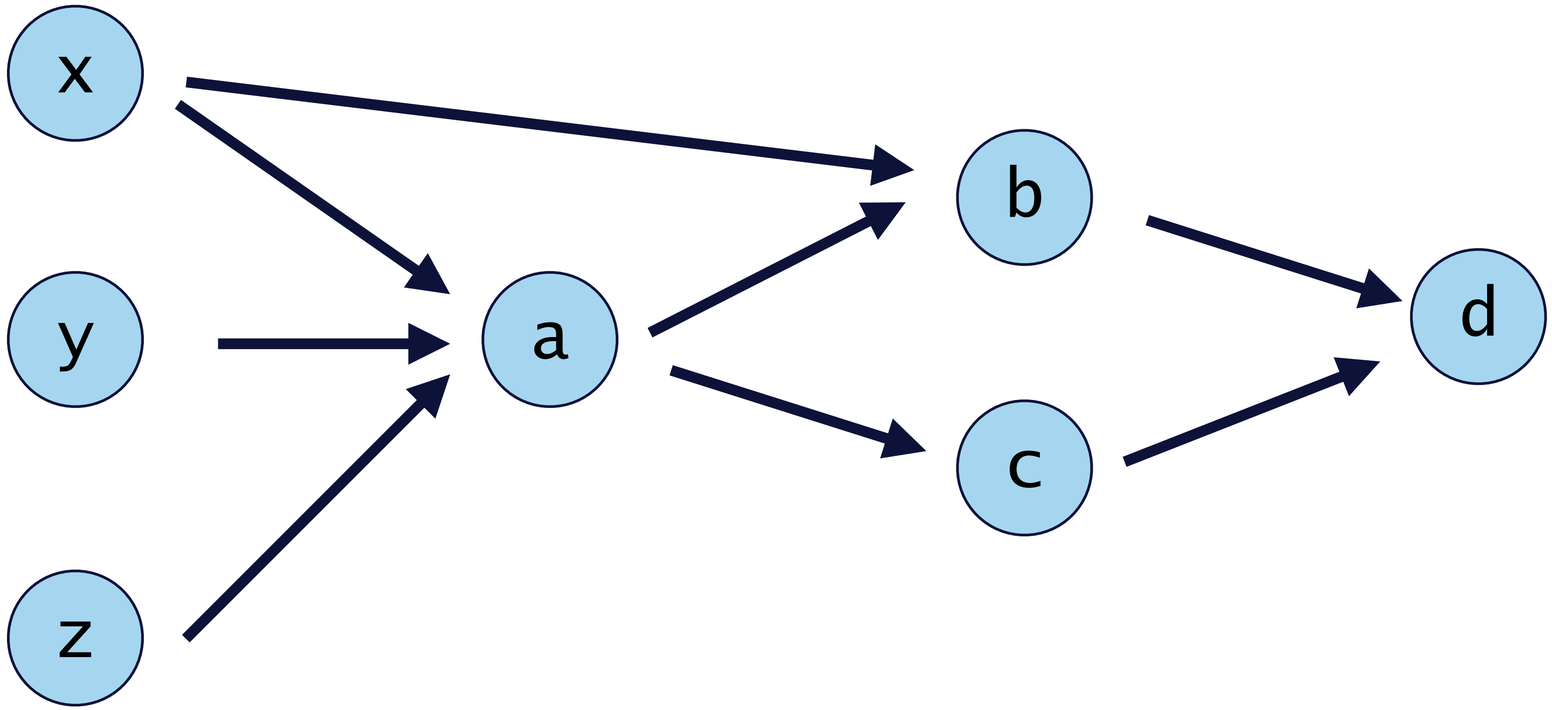
Complexity analysis

each edge of the computational graph will only be traversed once
-> reverse mode adjoint has the same time complexity as the primal function!!

this is named "**cheap gradient principle**" by Griewank

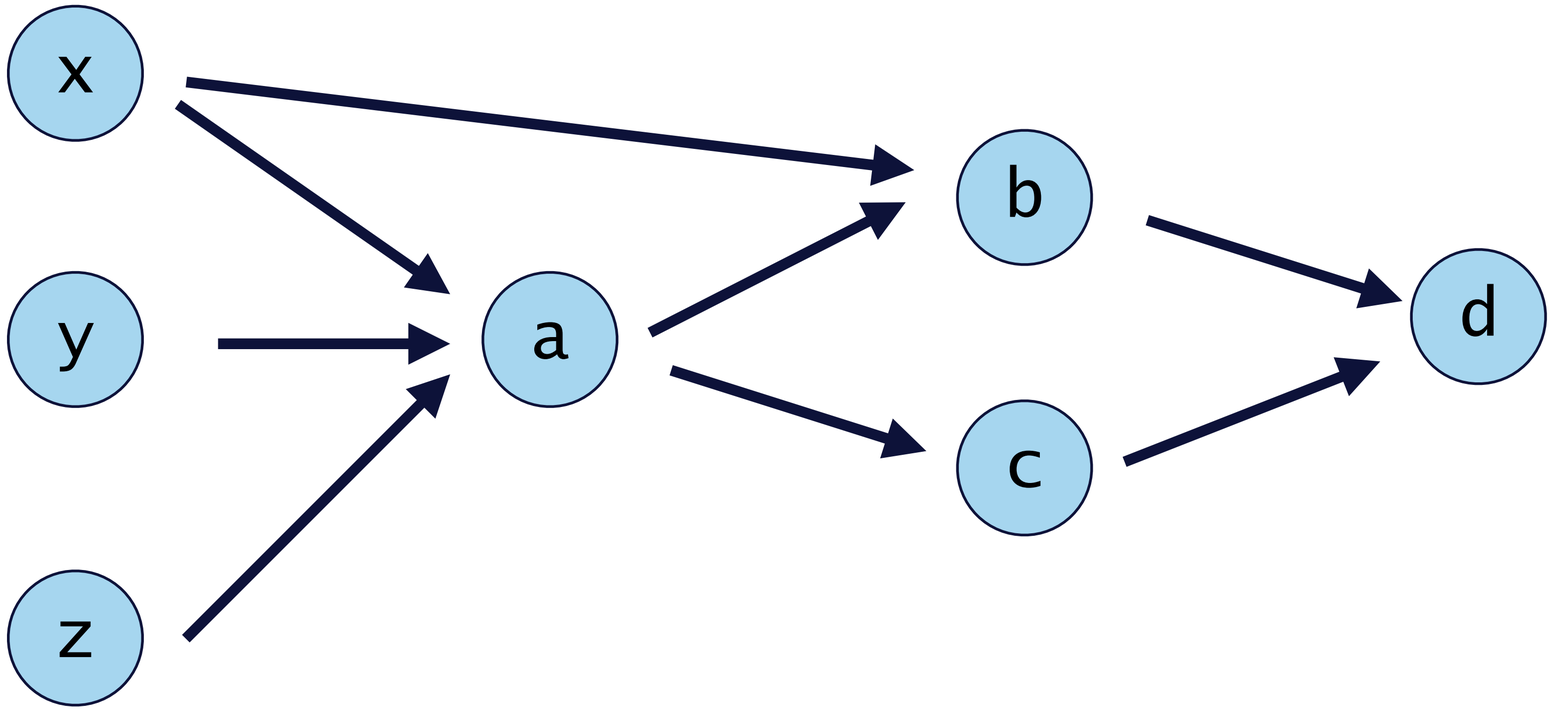


Automatic differentiation = path counting problem



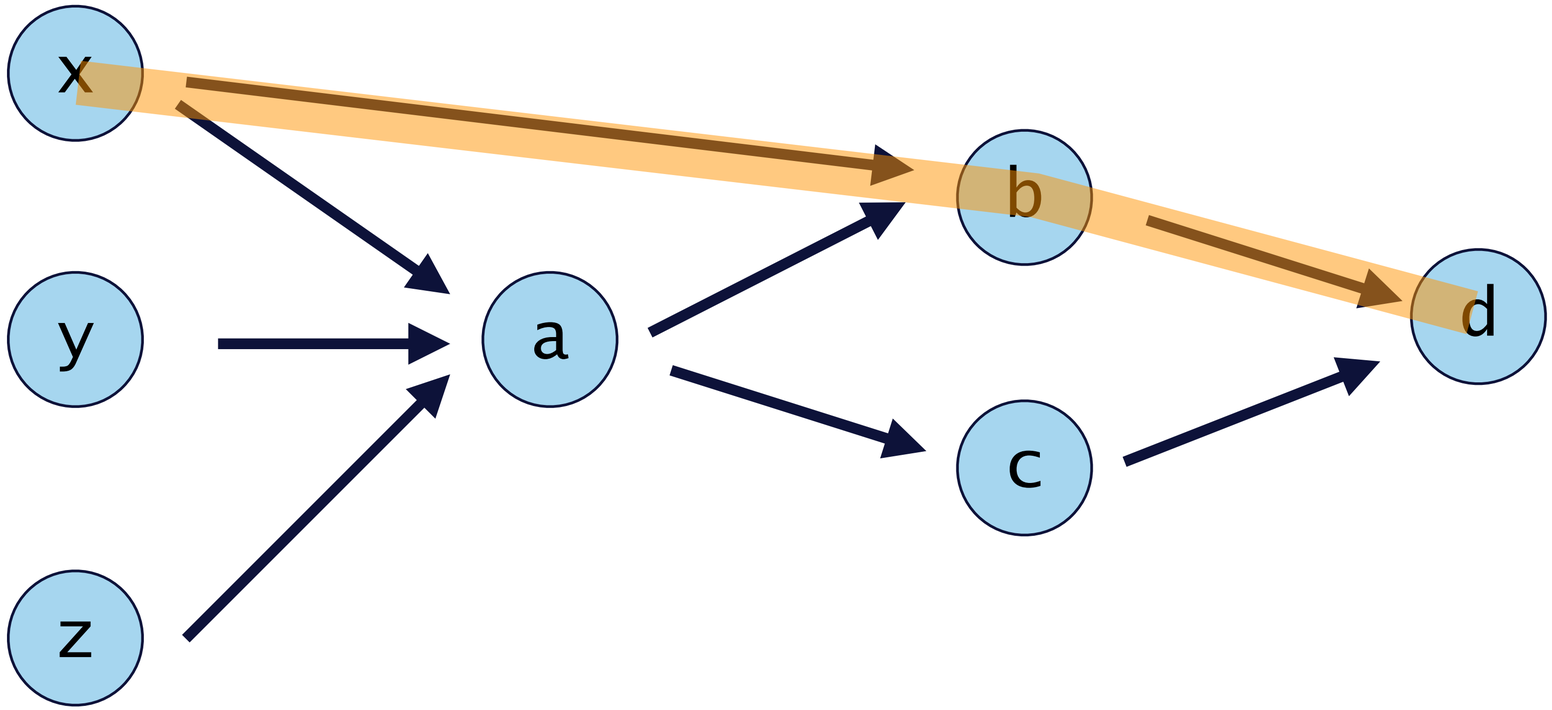
Automatic differentiation = path counting problem

$$\frac{\partial d}{\partial x}$$



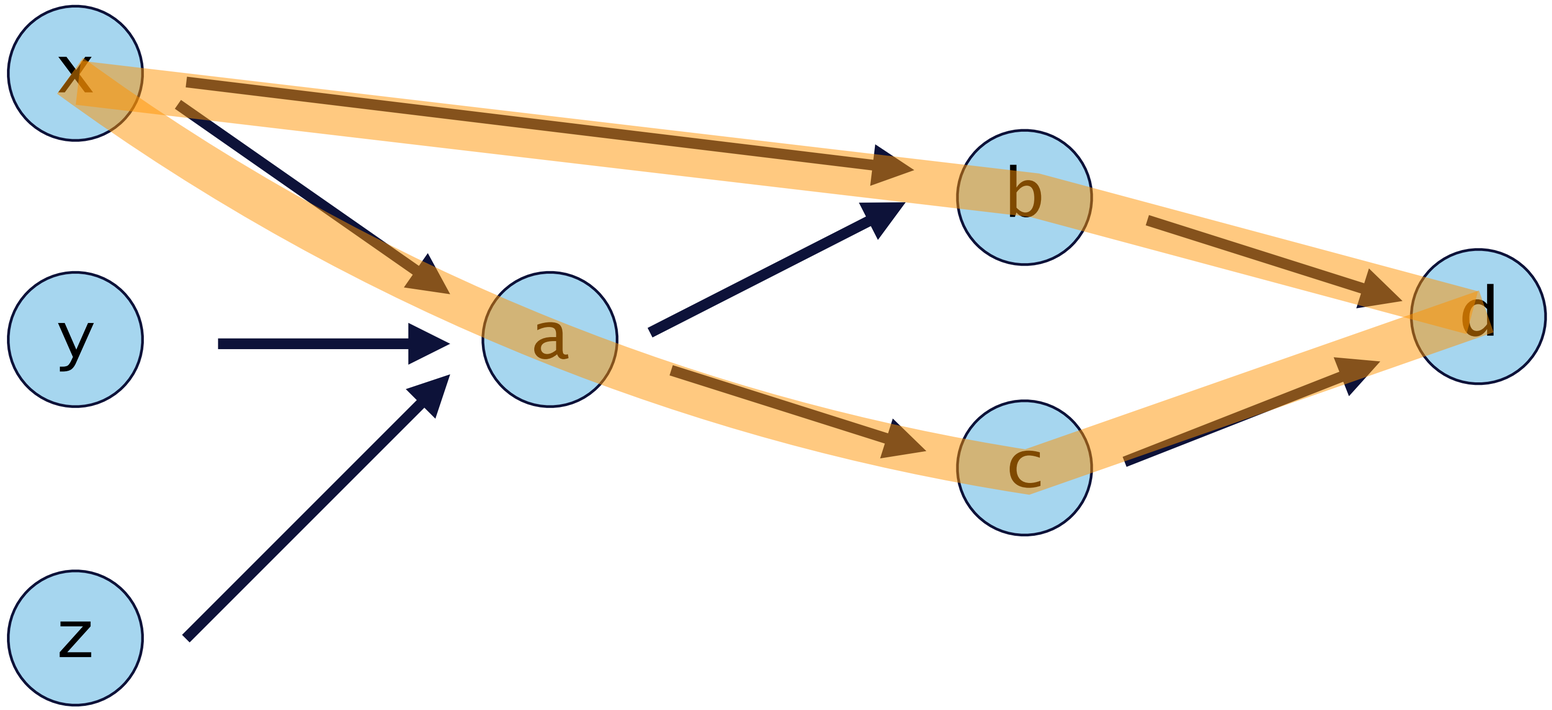
Automatic differentiation = path counting problem

$$\frac{\partial d}{\partial x} = \frac{\partial d}{\partial b}\frac{\partial b}{\partial x} + \cdots$$



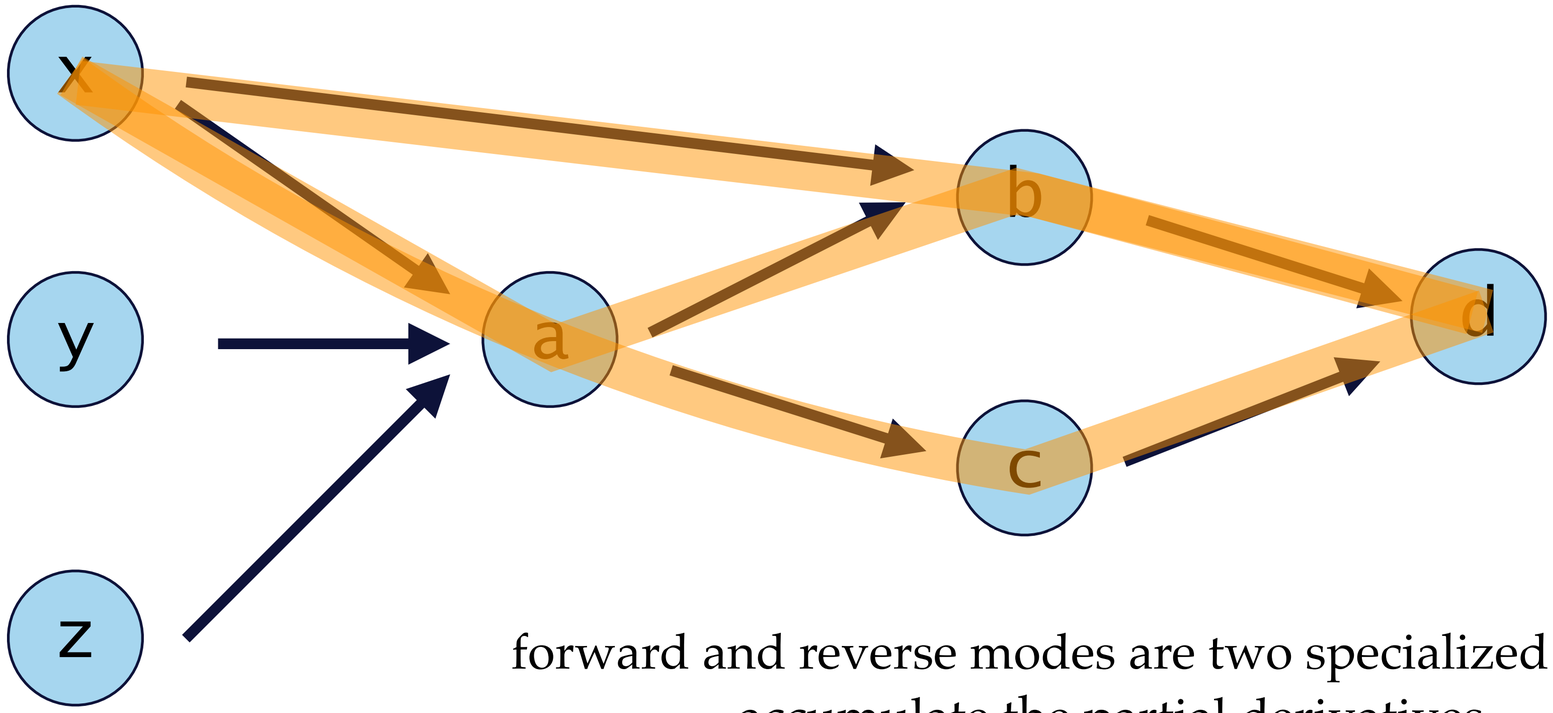
Automatic differentiation = path counting problem

$$\frac{\partial d}{\partial x} = \frac{\partial d}{\partial b}\frac{\partial b}{\partial x} + \frac{\partial d}{\partial c}\frac{\partial c}{\partial a}\frac{\partial a}{\partial x} + \cdots$$



Automatic differentiation = path counting problem

$$\frac{\partial d}{\partial x} = \frac{\partial d}{\partial b}\frac{\partial b}{\partial x} + \frac{\partial d}{\partial c}\frac{\partial c}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial d}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}$$



forward and reverse modes are two specialized way to accumulate the partial derivatives

Let's look at more examples

```
f(x, y):
    a = x + y
    b = -a
    return b
```

Let's look at more examples

```
f(x, y):
    a = x + y
    b = -a
    return b
```

```
DTf(x, y, db):
```

Let's look at more examples

```
f(x, y):
    a = x + y
    b = -a
    return b
```

```
DTf(x, y, db):
    a = x + y
    b = -a
```

Let's look at more examples

```
f(x, y):
    a = x + y
    b = -a
    return b
```

```
DTf(x, y, db):
    a = x + y
    b = -a
    da = -db
```

Let's look at more examples

```
f(x, y):
   a = x + y
   b = -a
   return b
```

```
DTf(x, y, db):
   a = x + y
   b = -a
   da = -db
   dx = da
   dy = da
```

Let's look at more examples

```
f(x, y):
    a = x + y
    b = -a
    return b
```

```
DTf(x, y, db):
    a = x + y
    b = -a
    da = -db
    dx = da
    dy = da
    return dx, dy
```

Let's look at even more examples

```
f(x, y):
  a = x * y
  b = a / y
  return b
```

```
DTf(x, y, db):
```

Let's look at even more examples

```
f(x, y):
   a = x * y
   b = a / y
   return b
```

```
DTf(x, y, db):
   a = x * y
   b = a / y
```

Let's look at even more examples

```
f(x, y):
   a = x * y
   b = a / y
   return b
```

```
DTf(x, y, db):
   a = x * y
   b = a / y
   da = db / y
   dy = -db * a / y^2
```

Let's look at even more examples

```
f(x, y):
    a = x * y
    b = a / y
    return b
```

```
DTf(x, y, db):
    a = x * y
    b = a / y
    da = db / y
    dy = -db * a / y^2
    dx = da * y
    dy += da * x
```

Let's look at even more examples

```
f(x, y):
    a = x * y
    b = a / y
    return b
```

```
DTf(x, y, db):
    a = x * y
    b = a / y
    da = db / y
    dy = -db * a / y^2
    dx = da * y
    dy += da * x
    return dx, dy
```

One more example

```
f(x, y):
    a = x * y
    a = a / y
    return a
```

One more example

```
f(x, y):
    a = x * y
    a = a / y
    return a
```

```
DTf(x, y, da):
    a = x * y
    a = a / y
    da = da / y
    dy = -da * a / y^2
    dx = da * y
    dy += da * x
    return dx, dy
```

is this correct?

Dealing with side effects in reverse mode

```
x = g(x, y0, y1, …)
```

Dealing with side effects in reverse mode

option 1: rewrite the assignment to remove the side effect
("Single Static Assignment" form)

```
x = g(x, y0, y1, …)
```

```
x1 = g(x0, y0, y1, …)
```

see https://en.wikipedia.org/wiki/Static_single-assignment_form for more details

Dealing with side effects in reverse mode

option 1: rewrite the assignment to remove the side effect
("Single Static Assignment" form)

```
arr[i] = g(arr)
```

???

SSA rewrites can be made complicated when there are arrays involved

Dealing with side effects in reverse mode

option 2: what we implement in HW2

```
x = g(x, y0, y1, …)
```

(x can be an array access)

Dealing with side effects in reverse mode

```
x = g(x, y0, y1, …)
```

(x can be an array access)

option 2: what we implement in HW2

step 1: in forward pass, push x into a stack before calling g

```
stack.push(x)
x = g(x, y0, y1, …)
```

Dealing with side effects in reverse mode

```
x = g(x, y0, y1, …)
```

(x can be an array access)

option 2: what we implement in HW2

step 1: in forward pass, push x into a stack before calling g

```
stack.push(x)
x = g(x, y0, y1, …)
```

step 2: in reverse pass, pop x from the stack

```
x = stack.pop()
```

Dealing with side effects in reverse mode

```
x = g(x, y0, y1, …)
```

(x can be an array access)

option 2: what we implement in HW2

step 1: in forward pass, push x into a stack before calling g

```
stack.push(x)
x = g(x, y0, y1, …)
```

step 2: in reverse pass, pop x from the stack

```
x = stack.pop()
```

step 3: compute the adjoints

```
dx_, y0_, dy1_, … = DTg(x, y0, y1, …, dx)
```

Dealing with side effects in reverse mode

```
x = g(x, y0, y1, …)
```

(x can be an array access)

option 2: what we implement in HW2

step 1: in forward pass, push x into a stack before calling g

```
stack.push(x)
x = g(x, y0, y1, …)
```

step 2: in reverse pass, pop x from the stack

```
x = stack.pop()
```

step 3: compute the adjoints

```
dx_, y0_, dy1_, … = DTg(x, y0, y1, …, dx)
```

step 4: zero dx

```
dx = 0
```

Dealing with side effects in reverse mode

```
x = g(x, y0, y1, …)
```

(x can be an array access)

option 2: what we implement in HW2

step 1: in forward pass, push x into a stack before calling g

```
stack.push(x)
x = g(x, y0, y1, …)
```

step 2: in reverse pass, pop x from the stack

```
x = stack.pop()
```

step 3: compute the adjoints

```
dx_, y0_, dy1_, … = DTg(x, y0, y1, …, dx)
```

step 4: zero dx

```
dx = 0
```

step 5: accumulate adjoints

```
dx, dy0, dy1, … += dx_, dy0_, dy1_, …
```

Backpropagation = reverse mode

```
f(x, w0, w1, w2):
  l0_ = FC(x, w0)
  l0 = ReLU(l0_)
  l1_ = FC(l0, w1)
  l1 = ReLU(l1_)
  l2 = FC(l1, w2)
  return l2
```

Backpropagation = reverse mode

```
f(x, w0, w1, w2):
  l0_ = FC(x, w0)
  l0 = ReLU(l0_)
  l1_ = FC(l0, w1)
  l1 = ReLU(l1_)
  l2 = FC(l1, w2)
  return l2
```

```
DTf(x, w0, w1, w2, dl2):
  l0_ = FC(x, w0)
  l0 = ReLU(l0_)
  l1_ = FC(l0, w1)
  l1 = ReLU(l1_)
  l2 = FC(l1, w2)

  dl1, dw2 = DTFC(l1, w2, dl2)
  dl1_ = DTReLU(l1_, dl1)
  dl0, dw1 = DTFC(l0, w1, dl1_)
  dl0_ = DTReLU(l0_, dl0)
  dx, dw0 = DTFC(x, w0, dl0_)
  return dx, dw0, dw1, dw2
```

Backpropagation = reverse mode

```
f(x, w0, w1, w2):
  l0_ = FC(x, w0)
  l0 = ReLU(l0_)
  l1_ = FC(l0, w1)
  l1 = ReLU(l1_)
  l2 = FC(l1, w2)
  return l2
```

```
DTf(x, w0, w1, w2, dl2):
  l0_ = FC(x, w0)
  l0 = ReLU(l0_)
  l1_ = FC(l0, w1)
  l1 = ReLU(l1_)
  l2 = FC(l1, w2)

  dl1, dw2 = DTFC(l1, w2, dl2)
  dl1_ = DTReLU(l1_, dl1)
  dl0, dw1 = DTFC(l0, w1, dl1_)
  dl0_ = DTReLU(l0_, dl0)
  dx, dw0 = DTFC(x, w0, dl0_)
  return dx, dw0, dw1, dw2
```

Backpropagation = reverse mode

```
FC(x, w):
  out = float[x.len]
  for i:
    for j:
      out[i] += w[i, j] * x[j]
  return out
```

Backpropagation = reverse mode

```
FC(x, w):
  out = float[x.len]
  for i:
    for j:
      out[i] += w[i, j] * x[j]
  return out

DTFC(x, w, dout):
  // allocate dx, dw
  for i:
    for j:
      dw[i, j] += dout[i] * x[j]
      dx[j] += dout[i] * w[i, j]
  return dx, dw
```

(since matrix-vector multiplication is linear, we don't need the forward pass)

(outer product between `dout` and `x`)

(matrix-vector multiplication between `w` and `dout`)