# What does CPS have to do with deep learning?

Jeffrey Mark Siskind, qobi@purdue.edu

PURDUE
UNIVERSITY

Amazon, Wednesday 13 June 2018

Joint work with Barak Avrum Pearlmutter

# Barak and my Work

# Barak and my Work

AD in functional programs.

# Barak and my Work

AD in functional programs.

AD is easier in functional programs.

# Part I

# Forward Mode

$$f = f_1 \circ \cdots \circ f_n$$

# Forward Mode

$$f = f_1 \circ \cdots \circ f_n$$
$$\mathcal{J}(f)(x_0) = \mathcal{J}(f_n)(x_{n-1}) \times \cdots \times \mathcal{J}(f_1)(x_0)$$

# Forward Mode

$$
\begin{aligned}
f &= f_1 \circ \cdots \circ f_n \\
\mathcal{J}(f)(x_0) &= \mathcal{J}(f_n)(x_{n-1}) \times \cdots \times \mathcal{J}(f_1)(x_0) \\
\acute{x}_n &= \mathcal{J}(f)(x_0) \times \acute{x}_0
\end{aligned}
$$

# Forward Mode

$$
\begin{aligned}
f &= f_1 \circ \cdots \circ f_n \\
\mathcal{J}(f)(x_0) &= \mathcal{J}(f_n)(x_{n-1}) \times \cdots \times \mathcal{J}(f_1)(x_0) \\
\acute{x}_n &= \mathcal{J}(f)(x_0) \times \acute{x}_0
\end{aligned}
$$

$$
\begin{aligned}
x_1 &= f_1(x_0) \\
\acute{x}_1 &= \mathcal{J}(f_1)(x_0) \times \acute{x}_0 \\
&\vdots \\
x_n &= f_n(x_{n-1}) \\
\acute{x}_n &= \mathcal{J}(f_n)(x_{n-1}) \times \acute{x}_{n-1}
\end{aligned}
$$

# Reverse Mode

$$f = f_1 \circ \cdots \circ f_n$$

# Reverse Mode

$$f = f_1 \circ \cdots \circ f_n$$
$$\mathcal{J}(f)(x_0)^\top = \mathcal{J}(f_1)(x_0)^\top \times \cdots \times \mathcal{J}(f_n)(x_{n-1})^\top$$

# Reverse Mode

$$
\begin{aligned}
f &= f_1 \circ \cdots \circ f_n \\
\mathcal{J}(f)(x_0)^\top &= \mathcal{J}(f_1)(x_0)^\top \times \cdots \times \mathcal{J}(f_n)(x_{n-1})^\top \\
\grave{x}_0 &= \mathcal{J}(f)(x_0)^\top \times \grave{x}_n
\end{aligned}
$$

# Reverse Mode

$$
\begin{aligned}
f &= f_1 \circ \cdots \circ f_n \\
\mathcal{J}(f)(x_0)^\top &= \mathcal{J}(f_1)(x_0)^\top \times \cdots \times \mathcal{J}(f_n)(x_{n-1})^\top \\
\grave{x}_0 &= \mathcal{J}(f)(x_0)^\top \times \grave{x}_n
\end{aligned}
$$

$$
\begin{aligned}
x_1 &= f_1(x_0) \\
&\vdots \\
x_n &= f_n(x_{n-1}) \\
\grave{x}_{n-1} &= \mathcal{J}(f_n)(x_{n-1}) \times \grave{x}_n \\
&\vdots \\
\grave{x}_0 &= \mathcal{J}(f_1)(x_0) \times \grave{x}_1
\end{aligned}
$$

# Forward Mode by Overloading

$$
\begin{aligned}
x_1 &= f_1(x_0) \\
\acute{x}_1 &= \mathcal{J}(f_1)(x_0) \times \acute{x}_0 \\
&\vdots \\
x_n &= f_n(x_{n-1}) \\
\acute{x}_n &= \mathcal{J}(f_n)(x_{n-1}) \times \acute{x}_{n-1}
\end{aligned}
$$

# Forward Mode by Overloading

$$
\begin{aligned}
x_1 &= f_1(x_0) \\
\acute{x}_1 &= \mathcal{J}(f_1)(x_0) \times \acute{x}_0 \\
&\vdots \\
x_n &= f_n(x_{n-1}) \\
\acute{x}_n &= \mathcal{J}(f_n)(x_{n-1}) \times \acute{x}_{n-1}
\end{aligned}
$$

$$
x_i = f_i(x_{i-1})
$$

# Forward Mode by Overloading

$$
\begin{aligned}
x_1 &= f_1(x_0) \\
\acute{x}_1 &= \mathcal{J}(f_1)(x_0) \times \acute{x}_0 \\
&\vdots \\
x_n &= f_n(x_{n-1}) \\
\acute{x}_n &= \mathcal{J}(f_n)(x_{n-1}) \times \acute{x}_{n-1}
\end{aligned}
$$

$$
\begin{aligned}
x_i &= f_i(x_{i-1}) \\
\langle x_i, \acute{x}_i \rangle &= \langle f_i(x_{i-1}), \mathcal{J}(f_i)(x_{i-1}) \times \acute{x}_{i-1} \rangle
\end{aligned}
$$

# Forward Mode by Overloading

$$
\begin{aligned}
x_1 &= f_1(x_0) \\
\acute{x}_1 &= \mathcal{J}(f_1)(x_0) \times \acute{x}_0 \\
&\vdots \\
x_n &= f_n(x_{n-1}) \\
\acute{x}_n &= \mathcal{J}(f_n)(x_{n-1}) \times \acute{x}_{n-1}
\end{aligned}
$$

$$
\begin{aligned}
x_i &= f_i(x_{i-1}) \\
\langle x_i, \acute{x}_i \rangle &= \langle f_i(x_{i-1}), \mathcal{J}(f_i)(x_{i-1}) \times \acute{x}_{i-1} \rangle \\
\overrightarrow{x_i} &= \overrightarrow{f_i}\,(\overrightarrow{x_{i-1}})
\end{aligned}
$$

# Implementation of Forward Mode by Overloading—I

```
(define-structure dual-number primal tangent)

(set! original+ +)

(define (+ x y)
 (dual-number
  (original+ (primal x) (primal y))
  (original+ (tangent x) (tangent y))))

(define (derivative f x)
 (tangent (f (dual-number x 1))))
```

# Implementation of Forward Mode by Overloading—II

```
(set! original+ +)

(define (+ x y)
 (if (dual-number? x)
     (dual-number
       (original+ (primal x) (primal y))
       (original+ (tangent x) (tangent y)))
     (original+ x y)))
```

# Implementation of Forward Mode by Overloading—III

```
(set! original+ +)

(define (+ x y)
 (if (dual-number? x)
     (dual-number
       (+ (primal x) (primal y))
       (+ (tangent x) (tangent y)))
     (original+ x y)))

(define (derivative2 f x)
 (tangent
  (tangent
   (f (dual-number
       (dual-number x 1)
       (dual-number 1 0))))))
```

# Implementation of Forward Mode by Overloading—IV

```
(define +0 +)
(define (+1 x y)
 (dual-number
  (+0 (primal x) (primal y))
  (+0 (tangent x) (tangent y))))
(define (+2 x y)
 (dual-number
  (+1 (primal x) (primal y))
  (+1 (tangent x) (tangent y))))
⋮
(f0 x)
(tangent (f1 (dual-number x 1)))
(tangent
 (tangent
  (f2 (dual-number
       (dual-number x 1) (dual-number 1 0)))))
```

# Implementation of Forward Mode by Overloading—V

```
(define +0 +)

(define (+1 xp xt yp yt)
 (values
  (+0 xp yp)
  (+0 xt yt)))

(define (+2 xpp xpt xtp xtt ypp ypt ytp ytt)
 (let-values ((zpp zpt (+1 xpp xpt ypp ypt))
              (ztp ttt (+1 xtp xtt ytp xtt)))
  (values zpp zpt ztp ztt)))
⋮
```

# Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((derivative f) x) (tangent (f (make-bundle x 1))))
```

# Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((derivative f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))
```

# Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((derivative f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(derivative f)
```

# Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((derivative f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(derivative f)
(derivative (derivative f))
(derivative (lambda (x) ... (derivative (lambda (y) ...) ...) ...) ...)
```

# Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((derivative f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(derivative f)
(derivative (derivative f))
(derivative (lambda (x) ... (derivative (lambda (y) ...) ...) ...) ...)
```

# Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((derivative f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(derivative f)
(derivative (derivative f))
(derivative (lambda (x) ... (derivative (lambda (y) ...) ...) ...) ...)
```

Convenient

# Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((derivative f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(derivative f)
(derivative (derivative f))
(derivative (lambda (x) ... (derivative (lambda (y) ...) ...) ...) ...)
```

Convenient but slow

# Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((derivative f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(derivative f)
(derivative (derivative f))
(derivative (lambda (x) ... (derivative (lambda (y) ...) ...) ...) ...)
```

Convenient but slow

# Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define +
 (let ((+ +))
  (lambda (x1 x2)
   (make-bundle (+ (primal x1) (primal x2))
                (+ (tangent x1) (tangent x2))))))

(define *
 (let ((+ +) (* *))
  (lambda (x1 x2)
   (make-bundle (* (primal x1) (primal x2))
                (+ (* (primal x1) (tangent x2))
                   (* (tangent x1) (primal x2)))))))

(define ((derivative f) x) (tangent (f (make-bundle x 1))))

(define (f x) (* 2 (* x (* x x))))

(derivative f)
(derivative (derivative f))
(derivative (lambda (x) ... (derivative (lambda (y) ...) ...) ...) ...)
```

Convenient but slow

# Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define ((derivative f) x)
 (fluid-let ((+ (lambda (x1 x2)
                  (make-bundle (+ (primal x1) (primal x2))
                               (+ (tangent x1) (tangent x2)))))
             (* (lambda (x1 x2)
                  (make-bundle (* (primal x1) (primal x2))
                               (+ (* (primal x1) (tangent x2))
                                  (* (tangent x1) (primal x2)))))))
  (tangent (f (make-bundle x 1)))))
```

```
(define (f x) (* 2 (* x (* x x))))

(derivative f)
(derivative (derivative f))
(derivative (lambda (x) ... (derivative (lambda (y) ...) ...) ...) ...)
```

Convenient but slow

# Dynamic Overloading: SCMUTILS

```
(define-structure bundle primal tangent)
(define (primal p) (if (bundle? p) (bundle-primal p) p))
(define (tangent p) (if (bundle? p) (bundle-tangent p) 0))

(define ((derivative f) x)
 (fluid-let ((+ (lambda (x1 x2)
                  (make-bundle (+ (primal x1) (primal x2))
                               (+ (tangent x1) (tangent x2)))))
             (* (lambda (x1 x2)
                  (make-bundle (* (primal x1) (primal x2))
                               (+ (* (primal x1) (tangent x2))
                                  (* (tangent x1) (primal x2)))))))
  (tangent (f (make-bundle x 1)))))
```

```
(define (f x) (* 2 (* x (* x x))))

(derivative f)
(derivative (derivative f))
(derivative (lambda (x) ... (derivative (lambda (y) ...) ...) ...) ...)
```

Convenient but slow

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end
```

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end

function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end

function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

Fast

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end

function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

Fast but inconvenient

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)                                    AD_TOP = f
double precision x, f
f = 2.0d0*x*x*x
end

function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

Fast but inconvenient

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end

function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

```
AD_TOP = f
AD_IVARS = x
AD_DVARS = f
```

Fast but inconvenient

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end

function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

```
AD_TOP = f
AD_IVARS = x
AD_DVARS = f
```

Fast but inconvenient

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end

function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

```
AD_TOP = f
AD_IVARS = x
AD_DVARS = f
```

Fast but inconvenient

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)
double precision x, f
f = 2.0d0*x*x*x
end
```

```
AD_TOP = f
AD_IVARS = x
AD_DVARS = f
```

```
function gf(x, gx, gresult)
double precision x, gx, gf, gresult
gf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
end
```

```
AD_TOP = gf
AD_IVARS = x, gx
AD_DVARS = gf, gresult
```

Fast but inconvenient

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)                                    AD_TOP = f
double precision x, f                            AD_IVARS = x
f = 2.0d0*x*x*x                                  AD_DVARS = f
end

function gf(x, gx, gresult)                      AD_TOP = gf
double precision x, gx, gf, gresult              AD_IVARS = x, gx
gf = 2.0d0*x*x*x                                 AD_DVARS = gf, gresult
gresult = 6.0d0*x*x*gx
end

function ggf(x, gx, gx, ggx, gresult, ggresult, gresult)
double precision x, gx, gx, ggx, ggf, gresult, gresult, ggresult
ggf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
gresult = 6.0d0*x*x*gx
ggresult = 6.0d0*x*x*ggx+12.0d0*x*gx*gx
end
```

Fast but inconvenient

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)                                    AD_TOP = f
double precision x, f                            AD_IVARS = x
f = 2.0d0*x*x*x                                  AD_DVARS = f
end

function gf(x, gx, gresult)                      AD_TOP = gf
double precision x, gx, gf, gresult              AD_IVARS = x, gx
gf = 2.0d0*x*x*x                                 AD_DVARS = gf, gresult
gresult = 6.0d0*x*x*gx
end

function ggf(x, gx, gx, ggx, gresult, ggresult, gresult)
double precision x, gx, gx, ggx, ggf, gresult, gresult, ggresult
ggf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
gresult = 6.0d0*x*x*gx
ggresult = 6.0d0*x*x*ggx+12.0d0*x*gx*gx
end
```

Fast but inconvenient

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)                                    AD_TOP = f
double precision x, f                            AD_IVARS = x
f = 2.0d0*x*x*x                                  AD_DVARS = f
end

function gf(x, gx, gresult)                      AD_TOP = gf
double precision x, gx, gf, gresult              AD_IVARS = x, gx
gf = 2.0d0*x*x*x                                 AD_DVARS = gf, gresult
gresult = 6.0d0*x*x*gx
end

function ggf(x, gx, gx, ggx, gresult, ggresult, gresult)
double precision x, gx, gx, ggx, ggf, gresult, gresult, ggresult
ggf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
gresult = 6.0d0*x*x*gx
ggresult = 6.0d0*x*x*ggx+12.0d0*x*gx*gx
end
```

Fast but inconvenient

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)                                    AD_TOP = f
double precision x, f                            AD_IVARS = x
f = 2.0d0*x*x*x                                  AD_DVARS = f
end

function gf(x, gx, gresult)                      AD_TOP = gf
double precision x, gx, gf, gresult              AD_IVARS = x, gx
gf = 2.0d0*x*x*x                                 AD_DVARS = gf, gresult
gresult = 6.0d0*x*x*gx
end

function ggf(x, gx, gx, ggx, gresult, ggresult, gresult)
double precision x, gx, gx, ggx, ggf, gresult, gresult, ggresult
ggf = 2.0d0*x*x*x
gresult = 6.0d0*x*x*gx
gresult = 6.0d0*x*x*gx
ggresult = 6.0d0*x*x*ggx+12.0d0*x*gx*gx
end
```

Fast but inconvenient

# Preprocessor: ADIFOR and TAPENADE

```
function f(x)                                    AD_TOP = f
double precision x, f                            AD_IVARS = x
f = 2.0d0*x*x*x                                  AD_DVARS = f
end

function gf(x, gx, gresult)                      AD_TOP = gf
double precision x, gx, gf, gresult              AD_IVARS = x, gx
gf = 2.0d0*x*x*x                                 AD_DVARS = gf, gresult
gresult = 6.0d0*x*x*gx                           AD_PREFIX = h
end

function hgf(x, hx, gx, hgx, gresult, hgresult, hresult)
double precision x, hx, gx, hgx, hgf, hresult, gresult, hgresult
hgf = 2.0d0*x*x*x
hresult = 6.0d0*x*x*hx
gresult = 6.0d0*x*x*gx
hgresult = 6.0d0*x*x*hgx+12.0d0*x*gx*hx
end
```

Fast but inconvenient

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...
```

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...
```

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...
```

Slow and inconvenient

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...

template <typename T>
T f(T x) {return 2*x*x*x;}
T x;
```

Slow and inconvenient

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...

template <typename T>
T f(T x) {return 2*x*x*x;}
T x;
```

Slow and inconvenient

# Static Overloading: FADBAD++

```
double f(double x) {return 2*x*x*x;}
double x;
... f(x) ...

F<double> f(F<double> x) {return 2*x*x*x;}
F<double> x;
x.diff(0, 1);
... f(x).d(0) ...

F<F<double> > f(F<F<double> > x) {return 2*x*x*x;}
F<F<double> > x;
x.diff(0, 1);
x.diff(0, 1).diff(0,1);
... f(x).d(0).d(0) ...

template <typename T>
T f(T x) {return 2*x*x*x;}
T x;
```

Slow and inconvenient

# Implementation of Reverse Mode by Overloading

```
(define-structure tape value operation argments)

(set! original+ +)

(define (+ x y)
 (if (tape? x)
     (tape (+ (value x) (value y))
           '+
           (list (arguments x) (arguments y)))
     (original+ x y)))
```

# Reverse Mode

$$
\begin{aligned}
x_1 &= f_1(x_0) \\
&\vdots \\
x_n &= f_n(x_{n-1}) \\
\grave{x}_{n-1} &= \mathcal{J}(f_n)(x_{n-1}) \times \grave{x}_n \\
&\vdots \\
\grave{x}_0 &= \mathcal{J}(f_1)(x_0) \times \grave{x}_1
\end{aligned}
$$

# Implementation of Reverse Mode by Transformation—I

```
subroutine sqr(x, y)
  y = x * x
  end

subroutine l2(x1, y1, x2, y2, r)
  t1 = x2 - x1
  sqr(t1, t2)
  t3 = y2 - y1
  sqr(t3, t4)
  r = t2 + t4
  end
```

# Implementation of Reverse Mode by Transformation—II

```
subroutine sqrf(xp, yp)
  push(xp)
  yp = xp * xp
  end

subroutine l2f(x1p, y1p, x2p, y2p, rp)
  t1p = x2p - x1p
  sqr(t1p, t2p)
  t3p = y2p - y1p
  sqr(t3p, t4p)
  rp = t2p + t4p
  end
```

# Implementation of Reverse Mode by Transformation—III

```
subroutine sqrr(xc, yc)
  pop(xp)
  xc = yc * xp
  xc += xp * yc
  end

subroutine l2r(x1c, y1c, x2c, y2c, rc)
  t2c = rc
  t4c = rc
  sqrr(t3c, t4c)
  y2c = -t3c
  y1c = t3c
  sqrr(t1c, t2c)
  x2c = -t1c
  x1c = t1c
  end
```

# Key Idea

Migrate reflective source-to-source transformation
from run time to compile time
with abstract interpretation

# Traditional AD by Source-to-Source Transformation

## Preprocessor at Compile Time

```
function g(x)
   return x+1
end


function f(x)
   return 2*g(x)
end



... derivative(f, 3) ...
```

# Traditional AD by Source-to-Source Transformation

## Preprocessor at Compile Time

```
function g(x)
   return x+1
end


function f(x)
   return 2*g(x)
end


local y, y_tangent = f_forward(3, 1)
... y_tangent ...
```

# Traditional AD by Source-to-Source Transformation

## Preprocessor at Compile Time

```
function g(x)
   return x+1
end


function f_forward(x, x_tangent)
   local y, y_tangent = g_forward(x, x_tangent)
   return return 2*y, 2*y_tangent
end

local y, y_tangent = f_forward(3, 1)
... y_tangent ...
```

# Traditional AD by Source-to-Source Transformation

## Preprocessor at Compile Time

```
function g_forward(x, x_tangent)
   local y, y_tangent = x, x_tangent
   return x+1, x_tangent
end

function f_forward(x, x_tangent)
   local y, y_tangent = g_forward(x, x_tangent)
   return return 2*y, 2*y_tangent
end

local y, y_tangent = f_forward(3, 1)
... y_tangent ...
```

# Source-to-Source Transformation at Run Time

## Reflection

```
function f(x)
   return 2*g(x)
end
```

# Source-to-Source Transformation at Run Time

## Reflection

```
function f(x)
   return 2*g(x)
end

code(f)
```

# Source-to-Source Transformation at Run Time

## Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"
```

--

# Source-to-Source Transformation at Run Time

## Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end")
```

--

# Source-to-Source Transformation at Run Time

## Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end") ==> "function f_forward(x, x_tangent)
                         local y, y_tangent = g_forward(x, x_tangent)
                         return return 2*y, 2*y_tangent
                      end"
```

--

# Source-to-Source Transformation at Run Time

## Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end") ==> "function f_forward(x, x_tangent)
                         local y, y_tangent = g_forward(x, x_tangent)
                         return return 2*y, 2*y_tangent
                      end"

compile("function f_forward(x, x_tangent)
            local y, y_tangent = g_forward(x, x_tangent)
            return return 2*y, 2*y_tangent
         end")
```

--

# Source-to-Source Transformation at Run Time

## Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end") ==> "function f_forward(x, x_tangent)
                         local y, y_tangent = g_forward(x, x_tangent)
                         return return 2*y, 2*y_tangent
                      end"

compile("function f_forward(x, x_tangent)
            local y, y_tangent = g_forward(x, x_tangent)
            return return 2*y, 2*y_tangent
         end") ==> f_forward
```

--

# Source-to-Source Transformation at Run Time

## Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end") ==> "function f_forward(x, x_tangent)
                         local y, y_tangent = g_forward(x, x_tangent)
                         return return 2*y, 2*y_tangent
                      end"

compile("function f_forward(x, x_tangent)
            local y, y_tangent = g_forward(x, x_tangent)
            return return 2*y, 2*y_tangent
         end") ==> f_forward

called_by(f)
```

--

# Source-to-Source Transformation at Run Time

## Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end") ==> "function f_forward(x, x_tangent)
                         local y, y_tangent = g_forward(x, x_tangent)
                         return return 2*y, 2*y_tangent
                      end"

compile("function f_forward(x, x_tangent)
            local y, y_tangent = g_forward(x, x_tangent)
            return return 2*y, 2*y_tangent
         end") ==> f_forward

called_by(f) ==> {g}
```

--

# Source-to-Source Transformation at Run Time

## Reflection

```
function f(x)
   return 2*g(x)
end

code(f) ==> "function f(x)
                return 2*g(x)
             end"

transform("function f(x)
              return 2*g(x)
           end") ==> "function f_forward(x, x_tangent)
                         local y, y_tangent = g_forward(x, x_tangent)
                         return return 2*y, 2*y_tangent
                      end"

compile("function f_forward(x, x_tangent)
            local y, y_tangent = g_forward(x, x_tangent)
            return return 2*y, 2*y_tangent
         end") ==> f_forward

called_by(f) ==> {g}

function derivative(f, x)
   for g in called_by(f) do compile(transform(code(g))) end
   local y, y_tangent = compile(transform(code(f)))(x, 1)
   return y_tangent
end
--
```

# But How Can We Make This Efficient?

```
while not converged() do
   x = x-eta*derivative(f, x)
end
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add(x, y)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... add(x, y) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add(x, y)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = DOUBLE, y = DOUBLE
... add(x, y) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add(x, y)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = DOUBLE, y = DOUBLE
... add(DOUBLE, DOUBLE) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_1(DOUBLE, DOUBLE)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = DOUBLE, y = DOUBLE
... add_1(DOUBLE, DOUBLE) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_1(DOUBLE, DOUBLE)
   if DOUBLE=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = DOUBLE, y = DOUBLE
... add_1(DOUBLE, DOUBLE) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_1(DOUBLE, DOUBLE)
   if false then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = DOUBLE, y = DOUBLE
... add_1(DOUBLE, DOUBLE) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_1(DOUBLE, DOUBLE)



      return scalar_add(x, y)

end

local x = DOUBLE, y = DOUBLE
... add_1(DOUBLE, DOUBLE) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_1(DOUBLE, DOUBLE)



      return scalar_add(x, y)

end

local x = 3, y = 4
... scalar_add(x, y) ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_1(DOUBLE, DOUBLE)



      return scalar_add(x, y)

end

local x = 3, y = 4
... x+y ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... add(x, y) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add(x, y)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... x+y ...
local x = ARRAY, y = ARRAY
... add(x, y) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add(x, y)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... x+y ...
local x = ARRAY, y = ARRAY
... add(ARRAY, ARRAY) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_2(ARRAY, ARRAY)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... x+y ...
local x = ARRAY, y = ARRAY
... add_2(ARRAY, ARRAY) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_2(ARRAY, ARRAY)
   if ARRAY=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... x+y ...
local x = ARRAY, y = ARRAY
... add_2(ARRAY, ARRAY) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_2(ARRAY, ARRAY)
   if true then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... x+y ...
local x = ARRAY, y = ARRAY
... add_2(ARRAY, ARRAY) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add_2(ARRAY, ARRAY)

      return vector_add(x, y)



end

local x = 3, y = 4
... x+y ...
local x = ARRAY, y = ARRAY
... add_2(ARRAY, ARRAY) ...
```

# Abstract Interpretation aka (Polyvariant) Flow Analysis

```
function scalar_add(x, y)
   return x+y
end

function vector_add(x, y)
   local n = x:size(1)
   local z = torch.Tensor(n)
   for i = 1, n do
      z[i] = x[i]+y[i]
   end
   return z
end

function add(x, y)
   if x:type()=="torch.Tensor" then
      return vector_add(x, y)
   else
      return scalar_add(x, y)
   end
end

local x = 3, y = 4
... x+y ...
local x = torch.Tensor(5):zeros(), y = torch.Tensor(5):zeros()
... vector_add(x, y) ...
```

# A Single Powerful Optimization

```
{x = e1, y = e2}.x
```

# A Single Powerful Optimization

$$\texttt{\{x = e1, y = e2\}.x} \rightsquigarrow \texttt{e1}$$

# A Single Powerful Optimization

$$\texttt{\{x = e1, y = e2\}.x} \rightsquigarrow \texttt{e1}$$

- can eliminate storage allocation

# A Single Powerful Optimization

```
{x = e1, y = e2}.x ⇝ e1
```

- can eliminate storage allocation
- can eliminate storage reclamation

# A Single Powerful Optimization

$$\texttt{\{x = e1, y = e2\}.x} \rightsquigarrow \texttt{e1}$$

- can eliminate storage allocation
- can eliminate storage reclamation
- can eliminate storage writes

# A Single Powerful Optimization

$$\texttt{\{x = e1, y = e2\}.x} \rightsquigarrow \texttt{e1}$$

- can eliminate storage allocation
- can eliminate storage reclamation
- can eliminate storage writes
- can eliminate storage reads

# A Single Powerful Optimization

```
{x = e1, y = e2}.x ⇝ e1
```

- can eliminate storage allocation
- can eliminate storage reclamation
- can eliminate storage writes
- can eliminate storage reads
- can eliminate dead code

# The Kind of Code People Write in Dynamic Languages

```
function map(f, x)
   y = torch.Tensor(x:size(1))
   for i = 1, x:size(1) do
      y[i] = f(x[i])
   end
   return y
end

function reduce(g, i, x)
   y = i
   for i = 1, x:size(1) do
      y = g(y, x[i])
   end
   return y
end

reduce(function(x, y) return x+y end,
       0,
       map(function(x) return x*x end, torch.Tensor({u, v, w, x, y})))
```

--

# The Kind of Code People Write in Dynamic Languages

```
function map(f, x)
   y = torch.Tensor(x:size(1))
   for i = 1, x:size(1) do
      y[i] = f(x[i])
   end
   return y
end

function reduce(g, i, x)
   y = i
   for i = 1, x:size(1) do
      y = g(y, x[i])
   end
   return y
end

reduce(function(x, y) return x+y end,
       0,
       map(function(x) return x*x end, torch.Tensor({u, v, w, x, y})))

u*u + v*v + w*w + x*x + y*y

--
```

# Key Idea

You need this anyway
to compile dynamic languages efficiently

# Same mechanism can support AD

# Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end
```

```
function derivative(g, x)
   local y, y_tangent = compile(transform(code(g)))(x, 1)

   return y_tangent
end
```

```
... derivative(f, 3) ...
```

# Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end
```

```
function derivative_1(g, x)
   local y, y_tangent = compile(transform(code(g)))(x, 1)

   return y_tangent
end
```

```
... derivative_1(FUNCTION_F, 3) ...
```

# Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end
```

```
function derivative_1(FUNCTION_F, x)
   local y, y_tangent = compile(transform(code(FUNCTION_F)))(x, 1)

   return y_tangent
end

... derivative_1(FUNCTION_F, 3) ...
```

# Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end
```

```
function derivative_1(FUNCTION_F, x)
   local y, y_tangent = compile(transform("function f(x)
                                              return 2*x
                                           end"))(x, 1)

   return y_tangent
end
```

```
... derivative_1(FUNCTION_F, 3) ...
```

# Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end
```

```
function derivative_1(FUNCTION_F, x)
   local y, y_tangent = compile("function f_forward(x, x_tangent)
                                    local y, y_tangent = 2*x, 2*x_tangent
                                    return y, y_tangent
                                 end")(x, 1)
   return y_tangent
end
```

```
... derivative_1(FUNCTION_F, 3) ...
```

# Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end

function f_forward(x, x_tangent)
   local y, y_tangent = 2*x, 2*x_tangent
   return y, y_tangent
end

function derivative_1(FUNCTION_F, x)
   local y, y_tangent = f_forward(x, 1)



   return y_tangent
end


... derivative_1(FUNCTION_F, 3) ...
```

# Migrating Reflective AD from Run Time to Compile Time

```
function f(x)
   return 2*x
end

function f_forward(x, x_tangent)
   local y, y_tangent = 2*x, 2*x_tangent
   return y, y_tangent
end

function derivative(g, x)
   local y, y_tangent = compile(transform(code(g)))(x, 1)



   return y_tangent
end

local y, y_tangent = f_forward(x, 1)
... y_tangent ...
```

# A Single Powerful Optimization

# A Single Powerful Optimization

- separates AD from optimization

# A Single Powerful Optimization

- separates AD from optimization
- allows simple formulation of AD transforms

# A Single Powerful Optimization

- separates AD from optimization
- allows simple formulation of AD transforms
  (forward mode is 28 lines; reverse mode is 155 lines)

# A Single Powerful Optimization

- separates AD from optimization
- allows simple formulation of AD transforms
  (forward mode is 28 lines; reverse mode is 155 lines)
- tape is a data structure (in the language)

# A Single Powerful Optimization

- separates AD from optimization
- allows simple formulation of AD transforms
  (forward mode is 28 lines; reverse mode is 155 lines)
- tape is a data structure (in the language)
- many AD optimizations (like TBR) fall out

# A Single Powerful Optimization

- separates AD from optimization
- allows simple formulation of AD transforms
  (forward mode is 28 lines; reverse mode is 155 lines)
- tape is a data structure (in the language)
- many AD optimizations (like TBR) fall out
- makes it easier to get it right

# A Single Powerful Optimization

- separates AD from optimization
- allows simple formulation of AD transforms
  (forward mode is 28 lines; reverse mode is 155 lines)
- tape is a data structure (in the language)
- many AD optimizations (like TBR) fall out
- makes it easier to get it right
- makes it easier to get it to nest

# Essence of Forward Transform

$$
\begin{array}{rcl}
\overrightarrow{c} & \rightsquigarrow & \overrightarrow{\mathcal{J}}\ c \\
\overrightarrow{\lambda x.e} & \rightsquigarrow & \lambda\, \overrightarrow{x}\, .\, \overrightarrow{e} \\
\overrightarrow{e_1\ e_2} & \rightsquigarrow & \overrightarrow{e_1}\ \overrightarrow{e_2} \\
\overrightarrow{\textbf{letrec}\ x_1 = e_1; \ldots; x_n = e_n\ \textbf{in}\ e} & \rightsquigarrow & \textbf{letrec}\ \overrightarrow{x_1} = \overrightarrow{e_1}; \ldots; \overrightarrow{x_n} = \overrightarrow{e_n}\ \textbf{in}\ \overrightarrow{e} \\
\overrightarrow{e_1, e_2} & \rightsquigarrow & \overrightarrow{e_1}\ \overrightarrow{,}\ \overrightarrow{e_2}
\end{array}
$$

# Essence of Reverse Transform

$$
\begin{array}{rcl}
\overleftarrow{x = c} & \rightsquigarrow & \overleftarrow{x} = \overleftarrow{\mathcal{J}}\, c \\
\overleftarrow{x_1 = x_2} & \rightsquigarrow & \overleftarrow{x_1} = \overleftarrow{x_2} \\
\overleftarrow{x = \lambda x.e} & \rightsquigarrow & \overleftarrow{x} = \lambda x.e \\
\overleftarrow{x = x_1\ x_2} & \rightsquigarrow & \overleftarrow{x}, \overleftarrow{\bar{x}} = \overleftarrow{x_1}\ \overleftarrow{x_2} \\
\overleftarrow{x = x_1, x_2} & \rightsquigarrow & \overleftarrow{x} = \overleftarrow{x_1}, \overleftarrow{x_2}
\end{array}
$$

$$
\begin{array}{rcl}
\overline{x_1 = x_2} & \rightsquigarrow & \overleftarrow{x_2} \mathrel{+}= \overleftarrow{x_1} \\
\overline{x = \lambda x.e} & \rightsquigarrow & \overleftarrow{\lambda x.e} \mathrel{+}= \overleftarrow{x} \\
\overline{x = x_1\ x_2} & \rightsquigarrow & \overleftarrow{x_1}, \overleftarrow{x_2} \mathrel{+}= \overleftarrow{\bar{x}}\ \overleftarrow{x} \\
\overline{x = x_1, x_2} & \rightsquigarrow & \overleftarrow{x_1}, \overleftarrow{x_2} \mathrel{+}= \overleftarrow{x}
\end{array}
$$

$$
\overleftarrow{\lambda x.\mathbf{let}\ b_1; \ldots; b_n \mathbf{in}\ y} \quad \rightsquigarrow \quad \lambda \overleftarrow{x}.\mathbf{let}\ \overleftarrow{b_1}; \ldots; \overleftarrow{b_n} \mathbf{in}\ \overleftarrow{y}, \lambda \overleftarrow{y}.\mathbf{let}\ \overline{b_n}; \ldots; \overline{b_1} \mathbf{in}\ \overleftarrow{x}
$$

# Game Theory

| | | $B$ | | | | |
|---|---|---|---|---|---|---|
| | | $b_1$ | $\dots$ | $b_j$ | $\dots$ | $b_n$ |
| | $a_1$ | | | | | |
| | $\vdots$ | | $\ddots$ | $\vdots$ | | |
| $A$ | $a_i$ | | $\dots$ | $\textsc{Payoff}(a_i, b_j)$ | $\dots$ | |
| | $\vdots$ | | | $\vdots$ | $\ddots$ | |
| | $a_m$ | | | | | |

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ.

# Game Theory

| | | $B$ | | | | |
|---|---|---|---|---|---|---|
| | | $b_1$ | $\dots$ | $b_j$ | $\dots$ | $b_n$ |
| | $a_1$ | | | | | |
| | $\vdots$ | | $\ddots$ | $\vdots$ | | |
| $A$ | $a_i$ | | $\dots$ | $\textsc{Payoff}(a_i, b_j)$ | $\dots$ | |
| | $\vdots$ | | | $\vdots$ | $\ddots$ | |
| | $a_m$ | | | | | |

$$\max_{a \in A} \min_{b \in B} \textsc{Payoff}(a, b)$$

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ.

# Game Theory

| | | $\mathbb{R}^n$ | |
|---|---|---|---|
| | $\dots$ | $\mathbf{b}$ | $\dots$ |
| $\mathbb{R}^m$ $\mathbf{a}$ | $\ddots$ | $\vdots$ | |
| | $\dots$ | $\textsc{Payoff}(\mathbf{a}, \mathbf{b})$ | $\dots$ |
| | | $\vdots$ | $\ddots$ |

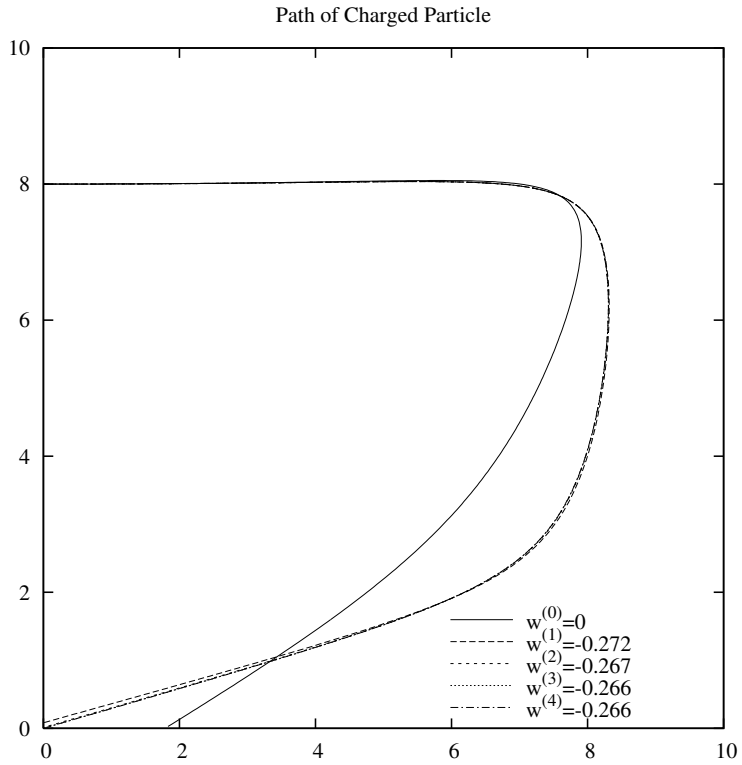
$$\max_{\mathbf{a} \in \mathbb{R}^m} \min_{\mathbf{b} \in \mathbb{R}^n} \textsc{Payoff}(\mathbf{a}, \mathbf{b})$$

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ.

# Code

```
(letrec ((loop
          (lambda (i r)
           (if (zero? i)
               r
               (loop (- i 1)
                     (let* ((start (list (real 1) (real 1)))
                            (f (lambda (x1 y1 x2 y2)
                                 (- (+ (sqr x1) (sqr y1))
                                    (+ (sqr x2) (sqr y2)))))
                            ((list x1* y1*)
                             (multivariate-argmin-F
                              (lambda ((list x1 y1))
                               (multivariate-max-F
                                (lambda ((list x2 y2)) (f x1 y1 x2 y2))
                                start))
                              start))
                            ((list x2* y2*)
                             (multivariate-argmax-F
                              (lambda ((list x2 y2)) (f x1* y1* x2 y2))
                              start)))
                       (list (list (write-real x1*) (write-real y1*))
                             (list (write-real x2*) (write-real y2*)))))))))
 (loop (real 1000) (list (list (real 0) (real 0)) (list (real 0) (real 0)))))
```

# Cathode Ray Tubes




$$
\begin{aligned}
\text{potential: } p(\mathbf{x}; w) &= \|\mathbf{x} - (10, 10 - w)\|^{-1} + \|\mathbf{x} - (10, 0)\|^{-1} \\
\ddot{\mathbf{x}}(t) &= -\nabla_{\mathbf{x}} p(\mathbf{x})|_{\mathbf{x}=\mathbf{x}(t)} \\
\dot{\mathbf{x}}(t + \Delta t) &= \dot{\mathbf{x}}(t) + \Delta t \ddot{\mathbf{x}}(t) \\
\mathbf{x}(t + \Delta t) &= \mathbf{x}(t) + \Delta t \dot{\mathbf{x}}(t) \\
\text{When: } x_1(t + \Delta t) &\leq 0 \\
\text{let: } \Delta t_f &= -x_1(t)/\dot{x}_1(t) \\
t_f &= t + \Delta t_f \\
\mathbf{x}(t_f) &= \mathbf{x}(t) + \Delta t_f \dot{\mathbf{x}}(t) \\
\text{Error: } E(w) &= x_0(t_f)^2 \\
\text{Find:} & \quad \operatorname*{argmin}_w E(w)
\end{aligned}
$$

Sprague, C. S. and George, R. H. (1939). *Cathode Ray Deflecting Electrode*. US Patent 2,161,437.

George, R. H. (1940). *Cathode Ray Tube*. US Patent 2,222,942.

# Code

```
(define (naive-euler w)
 (let* ((charges
         (list (list (real 10) (- (real 10) w)) (list (real 10) (real 0))))
        (x-initial (list (real 0) (real 8)))
        (xdot-initial (list (real 0.75) (real 0)))
        (delta-t (real 1e-1))
        (p (lambda (x)
             ((reduce + (real 0))
              ((map (lambda (c) (/ (real 1) (distance x c)))) charges)))))
  (letrec ((loop (lambda (x xdot)
                   (let* ((xddot (k*v (real -1) ((gradient-F p) x)))
                          (x-new (v+ x (k*v delta-t xdot))))
                     (if (positive? (list-ref x-new 1))
                         (loop x-new (v+ xdot (k*v delta-t xddot)))
                         (let* ((delta-t-f (/ (- (real 0) (list-ref x 1))
                                              (list-ref xdot 1)))
                                (x-t-f (v+ x (k*v delta-t-f xdot))))
                           (sqr (list-ref x-t-f 0)))))))))
   (loop x-initial xdot-initial))))

(letrec ((loop
          (lambda (i r)
            (if (zero? i)
                r
                (loop (- i 1)
                      (let* ((w0 (real 0))
                             ((list w*)
                              (multivariate-argmin-F
                               (lambda ((list w)) (naive-euler w)) (list w0))))
                        (write-real w*)))))))
 (loop (real 1000) (real 0)))
```

# Probabilistic Lambda Calculus

$P =$ **if** $x_0$ **then** $0$ **else if** $x_1$ **then** $1$ **else** $2$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

# Probabilistic Lambda Calculus

$P =$ **if** $x_0$ **then** $0$ **else if** $x_1$ **then** $1$ **else** $2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\quad$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$

$\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\quad$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

# Probabilistic Lambda Calculus

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\quad$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$

$\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\quad$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

$\Pr(\mathcal{E}(P) = 0 | p_0, p_1) = p_0$

$\Pr(\mathcal{E}(P) = 1 | p_0, p_1) = (1 - p_0)p_1$

$\Pr(\mathcal{E}(P) = 2 | p_0, p_1) = (1 - p_0)(1 - p_1)$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

# Probabilistic Lambda Calculus

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0$ $\quad$ $\Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$

$\Pr(x_1 \mapsto \textbf{true}) = p_1$ $\quad$ $\Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

$$\Pr(\mathcal{E}(P) = 0 | p_0, p_1) = p_0$$
$$\Pr(\mathcal{E}(P) = 1 | p_0, p_1) = (1 - p_0)p_1$$
$$\Pr(\mathcal{E}(P) = 2 | p_0, p_1) = (1 - p_0)(1 - p_1)$$

$$\prod_{v \in \{0,1,2,2\}} \Pr(\mathcal{E}(P) = v | p_0, p_1) = p_0(1 - p_0)^3 p_1(1 - p_1)^2$$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

# Probabilistic Lambda Calculus

$P = \textbf{if } x_0 \textbf{ then } 0 \textbf{ else if } x_1 \textbf{ then } 1 \textbf{ else } 2$

$\Pr(x_0 \mapsto \textbf{true}) = p_0 \qquad \Pr(x_0 \mapsto \textbf{false}) = 1 - p_0$
$\Pr(x_1 \mapsto \textbf{true}) = p_1 \qquad \Pr(x_1 \mapsto \textbf{false}) = 1 - p_1$

$$\Pr(\mathcal{E}(P) = 0 | p_0, p_1) = p_0$$
$$\Pr(\mathcal{E}(P) = 1 | p_0, p_1) = (1 - p_0)p_1$$
$$\Pr(\mathcal{E}(P) = 2 | p_0, p_1) = (1 - p_0)(1 - p_1)$$

$$\prod_{v \in \{0,1,2,2\}} \Pr(\mathcal{E}(P) = v | p_0, p_1) = p_0(1 - p_0)^3 p_1(1 - p_1)^2$$

$$\operatorname*{argmax}_{p_0, p_1} \prod_{v \in \{0,1,2,2\}} \Pr(\mathcal{E}(P) = v | p_0, p_1) = \left\langle \frac{1}{4}, \frac{1}{3} \right\rangle$$

Koller, D., McAllester, D. , and Pfeffer, A. (1997). *Effective Bayesian Inference for Stochastic Programs*. Proceedings of the 14th National Conference on Artificial Intelligence (AAAI), pp. 740–7.

# Probabilistic Prolog

```
p(0).
p(X):-q(X).
q(1).
q(2).
```

# Probabilistic Prolog

$\Pr(\texttt{p(0).}) = p_0$
$\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$
$\Pr(\texttt{q(1).}) = p_1$
$\Pr(\texttt{q(2).}) = 1 - p_1$

# Probabilistic Prolog

$\Pr(\texttt{p(0).}) = p_0$
$\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$
$\Pr(\texttt{q(1).}) = p_1$
$\Pr(\texttt{q(2).}) = 1 - p_1$

$\Pr(\texttt{?-p(0).}) = p_0$
$\Pr(\texttt{?-p(1).}) = (1 - p_0)p_1$
$\Pr(\texttt{?-p(2).}) = (1 - p_0)(1 - p_1)$

# Probabilistic Prolog

$\Pr(\texttt{p(0).}) = p_0$
$\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$
$\Pr(\texttt{q(1).}) = p_1$
$\Pr(\texttt{q(2).}) = 1 - p_1$

$\Pr(\texttt{?-p(0).}) = p_0$
$\Pr(\texttt{?-p(1).}) = (1 - p_0)p_1$
$\Pr(\texttt{?-p(2).}) = (1 - p_0)(1 - p_1)$

$$\prod_{q \in \{\texttt{p(0)},\texttt{p(1)},\texttt{p(2)},\texttt{p(2)}\}} \Pr(\texttt{?-}q\texttt{.}) = p_0(1 - p_0)^3 p_1(1 - p_1)^2$$

# Probabilistic Prolog

$\Pr(\texttt{p(0).}) = p_0$
$\Pr(\texttt{p(X):-q(X).}) = 1 - p_0$
$\Pr(\texttt{q(1).}) = p_1$
$\Pr(\texttt{q(2).}) = 1 - p_1$

$\Pr(\texttt{?-p(0).}) = p_0$
$\Pr(\texttt{?-p(1).}) = (1 - p_0)p_1$
$\Pr(\texttt{?-p(2).}) = (1 - p_0)(1 - p_1)$

$$\prod_{q \in \{\texttt{p(0)},\texttt{p(1)},\texttt{p(2)},\texttt{p(2)}\}} \Pr(\texttt{?-}q\texttt{.}) = p_0(1 - p_0)^3 p_1(1 - p_1)^2$$

$$\operatorname*{argmax}_{p_0,p_1} \prod_{q \in \{\texttt{p(0)},\texttt{p(1)},\texttt{p(2)},\texttt{p(2)}\}} \Pr(\texttt{?-}q\texttt{.}) = \left\langle \frac{1}{4}, \frac{1}{3} \right\rangle$$

# Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
          (map-tagged-distribution
           (lambda (value) (value tagged-distribution))
           (evaluate (application-callee expression) environment))))))
```

# Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
          (map-tagged-distribution
           (lambda (value) (value tagged-distribution))
           (evaluate (application-callee expression) environment))))))
```

# Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
          (map-tagged-distribution
           (lambda (value) (value tagged-distribution))
           (evaluate (application-callee expression) environment))))))
```

# Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
          (map-tagged-distribution
           (lambda (value) (value tagged-distribution))
           (evaluate (application-callee expression) environment))))))
```

# Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
          (map-tagged-distribution
           (lambda (value) (value tagged-distribution))
           (evaluate (application-callee expression) environment))))))
```

# Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
          (map-tagged-distribution
           (lambda (value) (value tagged-distribution))
           (evaluate (application-callee expression) environment))))))
```

# Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
          (map-tagged-distribution
           (lambda (value) (value tagged-distribution))
           (evaluate (application-callee expression) environment))))))
```

# Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
          (map-tagged-distribution
           (lambda (value) (value tagged-distribution))
           (evaluate (application-callee expression) environment)))))))
```

# Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
          (map-tagged-distribution
           (lambda (value) (value tagged-distribution))
           (evaluate (application-callee expression) environment))))))
```

# Probabilistic Lambda Calculus

```
(define (evaluate expression environment)
 (cond
  ((constant-expression? expression)
   (singleton-tagged-distribution
    (constant-expression-value expression)))
  ((variable-access-expression? expression)
   (lookup-value
    (variable-access-expression-variable expression) environment))
  ((lambda-expression? expression)
   (singleton-tagged-distribution
    (lambda (tagged-distribution)
     (evaluate
      (lambda-expression-body expression)
      (cons (make-binding (lambda-expression-variable expression)
                          tagged-distribution)
            environment)))))
  (else (let ((tagged-distribution
               (evaluate (application-argument expression)
                         environment)))
          (map-tagged-distribution
           (lambda (value) (value tagged-distribution))
           (evaluate (application-callee expression) environment))))))
```

# Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
    (map-reduce
     *
     1.0
     (lambda (value)
      (likelihood value tagged-distribution))
     '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
    (map-reduce
     *
     1.0
     (lambda (value)
      (likelihood value tagged-distribution))
     '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list  Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                          Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                          ...)))))
    (map-reduce
     *
     1.0
     (lambda (value)
      (likelihood value tagged-distribution))
     '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
    (map-reduce
     *
     1.0
     (lambda (value)
      (likelihood value tagged-distribution))
     '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
    (map-reduce
     *
     1.0
     (lambda (value)
      (likelihood value tagged-distribution))
     '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
    (map-reduce
     *
     1.0
     (lambda (value)
      (likelihood value tagged-distribution))
     '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0   Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1   Pr(x1 ↦ false) = 1 − p1
                         ...))))
    (map-reduce
     *
     1.0
     (lambda (value)
      (likelihood value tagged-distribution))
     '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0  Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1  Pr(x1 ↦ false) = 1 − p1
                         ...))))
    (map-reduce
     *
     1.0
     (lambda (value)
      (likelihood value tagged-distribution))
     '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0  Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1  Pr(x1 ↦ false) = 1 − p1
                         ...))))
    (map-reduce
     *
     1.0
     (lambda (value)
      (likelihood value tagged-distribution))
     '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Lambda Calculus

```
(gradient-ascent
 (lambda (p)
  (let ((tagged-distribution
         (evaluate if x0 then 0 else if x1 then 1 else 2
                   (list Pr(x0 ↦ true) = p0  Pr(x0 ↦ false) = 1 − p0
                         Pr(x1 ↦ true) = p1  Pr(x1 ↦ false) = 1 − p1
                         ...))))
    (map-reduce
     *
     1.0
     (lambda (value)
      (likelihood value tagged-distribution))
     '(0 1 2 2))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

# Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

# Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

# Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

# Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

# Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

# Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

# Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses))))))))
   clauses)))
```

# Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses))))))))
   clauses)))
```

# Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

# Probabilistic Prolog

```
(define (proof-distribution term clauses)
 (let ((offset ...))
  (map-reduce
   append
   '()
   (lambda (clause)
    (let ((clause (alpha-rename clause offset)))
     (let loop ((p (clause-p clause))
                (substitution (unify term (clause-term clause)))
                (terms (clause-terms clause)))
      (if (boolean? substitution)
          '()
          (if (null? terms)
              (list (make-double p substitution))
              (map-reduce
               append
               '()
               (lambda (double)
                (loop (* p (double-p double))
                      (append substitution (double-substitution double))
                      (rest terms)))
               (proof-distribution
                (apply-substitution substitution (first terms)) clauses)))))))
   clauses)))
```

# Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 − p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 − p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 − p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 − p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 − p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 − p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 − p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 − p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 − p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 − p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 − p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 − p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
 '(0.5 0.5)
 1000.0
 0.1)
```

# Probabilistic Prolog

```
(gradient-ascent
 (lambda (p)
  (let ((clauses (list Pr(p(0).) = p0
                       Pr(p(X):-q(X).) = 1 − p0
                       Pr(q(1).) = p1
                       Pr(q(2).) = 1 − p1)))
   (map-reduce
    *
    1.0
    (lambda (query)
     (likelihood (proof-distribution query clauses)))
    '(p(0) p(1) p(2) p(2)))))
'(0.5 0.5)
1000.0
0.1)
```

# Generated Code

```
static void f2679(double a_f2679_0,double a_f2679_1,double a_f2679_2,double a_f2679_3){
  int t272381=((a_f2679_2==0.)?0:1);
  double t272406;
  double t272405;
  double t272404;
  double t272403;
  double t272402;
  if((t272381==0)){
    double t272480=(1.-a_f2679_0);
    double t272572=(1.-a_f2679_1);
    double t273043=(a_f2679_0+0.);
    double t274185=(t272480*a_f2679_1);
    double t274426=(t274185+0.);
    double t275653=(t272480*t272572);
    double t275894=(t275653+0.);
    double t277121=(t272480*t272572);
    double t277362=(t277121+0.);
    double t277431=(t277362*1.);
    double t277436=(t275894*t277431);
    double t277441=(t274426*t277436);
    double t277446=(t273043*t277441);
    ...
    double t1777107=(t1774696+t1715394);
    double t1777194=(0.-t1745420);
    double t1778533=(t1777194+t1419700);
    t272406=a_f2679_0;
    t272405=a_f2679_1;
    t272404=t277446;
    t272403=t1778533;
    t272402=t1777107;}
  else {...}
  r_f2679_0=t272406;
  r_f2679_1=t272405;
  r_f2679_2=t272404;
  r_f2679_3=t272403;
  r_f2679_4=t272402;}
```

# Benchmarks

| | | backprop | | |
|---|---|---|---|---|
| | | Fs | Fv | R |
| VLAD | STALIN∇ | 1.00 | ■ | 1.00 |
| FORTRAN | ADIFOR | 15.51 | 3.35 | ■ |
| | TAPENADE | 14.97 | 5.97 | 6.86 |
| C | ADIC | 22.75 | 5.61 | ■ |
| C++ | ADOL-C | 12.16 | 5.79 | 32.77 |
| | CPPAD | 54.74 | ■ | 29.24 |
| | FADBAD++ | 132.31 | 46.01 | 60.71 |
| ML | MLTON | 95.20 | ■ | 39.90 |
| | OCAML | 202.01 | ■ | 156.93 |
| | SML/NJ | 181.93 | ■ | 102.89 |
| HASKELL | GHC | ■ | ■ | ■ |
| SCHEME | BIGLOO | 743.26 | ■ | 360.07 |
| | CHICKEN | 1626.73 | ■ | 1125.24 |
| | GAMBIT | 671.54 | ■ | 379.63 |
| | IKARUS | 279.59 | ■ | 165.16 |
| | LARCENY | 1203.34 | ■ | 511.54 |
| | MIT SCHEME | 2446.33 | ■ | 1113.09 |
| | MZC | 1318.60 | ■ | 754.47 |
| | MZSCHEME | 1364.14 | ■ | 772.10 |
| | SCHEME->C | 597.67 | ■ | 280.93 |
| | SCMUTILS | 5889.26 | ■ | ■ |
| | STALIN | 435.82 | ■ | 281.27 |

# Damned Benchmarks

| | | particle | | | | saddle | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | FF | FR | RF | RR | FF | FR | RF | RR |
| VLAD | STALIN∇ | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| FORTRAN | ADIFOR | 2.05 | ■ | ■ | ■ | 5.44 | ■ | ■ | ■ |
| | TAPENADE | 5.51 | ■ | ■ | ■ | 8.09 | ■ | ■ | ■ |
| C | ADIC | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| C++ | ADOL-C | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| | CPPAD | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ |
| | FADBAD++ | 93.32 | ■ | ■ | ■ | 60.67 | ■ | ■ | ■ |
| ML | MLTON | 78.13 | 111.27 | 45.95 | 32.57 | 114.07 | 146.28 | 12.27 | 10.58 |
| | OCAML | 217.03 | 415.64 | 352.06 | 261.38 | 291.26 | 407.67 | 42.39 | 50.21 |
| | SML/NJ | 153.01 | 226.84 | 270.63 | 192.13 | 271.84 | 299.76 | 25.66 | 23.89 |
| HASKELL | GHC | 209.44 | ■ | ■ | ■ | 247.57 | ■ | ■ | ■ |
| SCHEME | BIGLOO | 627.78 | 855.70 | 275.63 | 187.39 | 1004.85 | 1076.73 | 105.24 | 89.23 |
| | CHICKEN | 1453.06 | 2501.07 | 821.37 | 1360.00 | 2276.69 | 2964.02 | 225.73 | 252.87 |
| | GAMBIT | 578.94 | 879.39 | 356.47 | 260.98 | 958.73 | 1112.70 | 89.99 | 89.23 |
| | IKARUS | 266.54 | 386.21 | 158.63 | 116.85 | 424.75 | 527.57 | 41.27 | 42.34 |
| | LARCENY | 964.18 | 1308.68 | 360.68 | 272.96 | 1565.53 | 1508.39 | 126.44 | 112.82 |
| | MIT SCHEME | 2025.23 | 3074.30 | 790.99 | 609.63 | 3501.21 | 3896.88 | 315.17 | 295.67 |
| | MZC | 1243.08 | 1944.00 | 740.31 | 557.45 | 2135.92 | 2434.05 | 194.49 | 187.53 |
| | MZSCHEME | 1309.82 | 1926.77 | 712.97 | 555.28 | 2371.35 | 2690.64 | 224.61 | 219.29 |
| | SCHEME->C | 582.20 | 743.00 | 270.83 | 208.38 | 910.19 | 913.66 | 82.93 | 69.87 |
| | SCMUTILS | 4462.83 | ■ | ■ | ■ | 7651.69 | ■ | ■ | ■ |
| | STALIN | 364.08 | 547.73 | 399.39 | 295.00 | 543.68 | 690.64 | 63.96 | 52.93 |

# Statistics

| | | probabilistic-lambda-calculus | | probabilistic-prolog | |
|---|---|---|---|---|---|
| | | F | R | F | R |
| VLAD | STALIN∇ | 1.00 | 1.00 | 1.00 | 1.00 |
| FORTRAN | ADIFOR | ■ | ■ | ■ | ■ |
| | TAPENADE | ■ | ■ | ■ | ■ |
| C | ADIC | ■ | ■ | ■ | ■ |
| C++ | ADOL-C | ■ | ■ | ■ | ■ |
| | CPPAD | ■ | ■ | ■ | ■ |
| | FADBAD++ | ■ | ■ | ■ | ■ |
| ML | MLTON | 129.11 | 114.88 | 848.45 | 507.21 |
| | OCAML | 249.40 | 499.43 | 1260.83 | 1542.47 |
| | SML/NJ | 234.62 | 258.53 | 2505.59 | 1501.17 |
| HASKELL | GHC | ■ | ■ | ■ | ■ |
| SCHEME | BIGLOO | 983.12 | 1016.50 | 12832.92 | 7918.21 |
| | CHICKEN | 2324.54 | 3040.44 | 44891.04 | 24634.44 |
| | GAMBIT | 1033.46 | 1107.26 | 26077.48 | 14262.70 |
| | IKARUS | 497.48 | 517.89 | 8474.57 | 4845.10 |
| | LARCENY | 1658.27 | 1606.44 | 25411.62 | 14386.61 |
| | MIT SCHEME | 4130.88 | 3817.57 | 87772.39 | 49814.12 |
| | MZC | 2294.93 | 2346.13 | 57472.76 | 31784.38 |
| | MZSCHEME | 2721.35 | 2625.21 | 60269.37 | 33135.06 |
| | SCHEME->C | 811.37 | 803.22 | 10605.32 | 5935.56 |
| | SCMUTILS | 7699.14 | ■ | 83656.17 | ■ |
| | STALIN | 956.47 | 1994.44 | 15048.42 | 16939.28 |

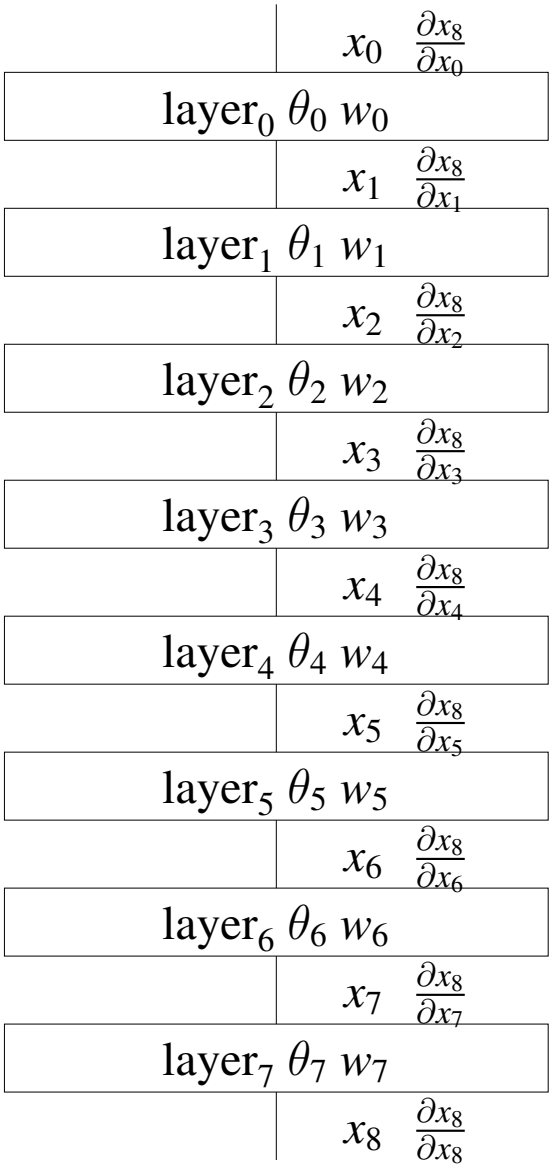
# Take-Home Message

Powerful and efficient AD can be attained by:

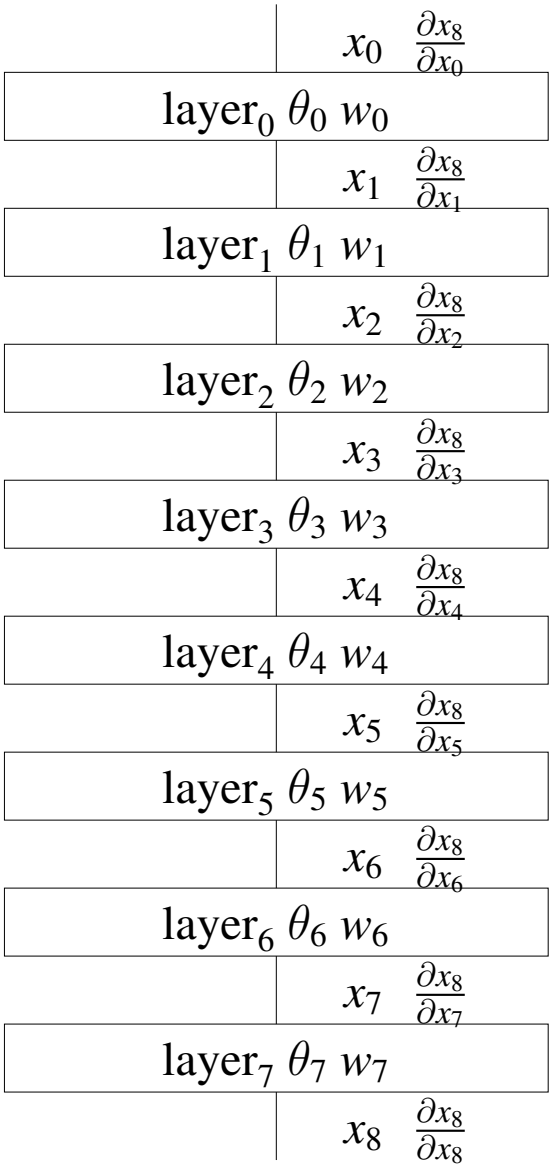
- integrating AD into compiler
- formulating AD as one of many compiler transformations
- using abstract interpretation to migrate AD transformation from run time to compile time

# Part II

# A Neural Network



# A Neural Network is a (Functional) Program




$$
\begin{aligned}
&\text{net}\ [\theta_0, \ldots, \theta_7]\ [w_0, \ldots, w_7]\ x_0 \triangleq \\
&\quad \textbf{let}\ \ x_1 = \text{layer}_0\ \theta_0\ w_0\ x_0 \\
&\qquad\quad x_2 = \text{layer}_1\ \theta_1\ w_1\ x_1 \\
&\qquad\quad x_3 = \text{layer}_2\ \theta_2\ w_2\ x_2 \\
&\qquad\quad x_4 = \text{layer}_3\ \theta_3\ w_3\ x_3 \\
&\qquad\quad x_5 = \text{layer}_4\ \theta_4\ w_4\ x_4 \\
&\qquad\quad x_6 = \text{layer}_5\ \theta_5\ w_5\ x_5 \\
&\qquad\quad x_7 = \text{layer}_6\ \theta_6\ w_6\ x_6 \\
&\qquad\quad x_8 = \text{layer}_7\ \theta_7\ w_7\ x_7 \\
&\quad \textbf{in}\ x_8
\end{aligned}
$$

# A (Functional) Program

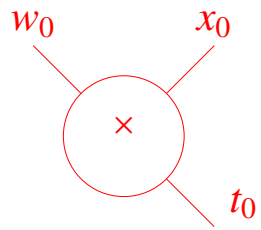
$$
\begin{array}{l}
f\ [w_0, w_1]\ [x_0, x_1] \stackrel{\triangle}{=} \\
\quad \textbf{let} \quad t_0 = w_0 \times x_0 \\
\qquad\quad t_1 = w_1 \times x_1 \\
\qquad\quad y = t_0 + t_1 \\
\quad \textbf{in}\ y
\end{array}
$$

# A (Functional) Program is a (Neural) Network

$$
\begin{array}{l}
f\ [w_0, w_1]\ [x_0, x_1] \stackrel{\triangle}{=} \\
\quad \textbf{let} \quad t_0 = w_0 \times x_0 \\
\qquad\quad\ \ t_1 = w_1 \times x_1 \\
\qquad\quad\ \ y = t_0 + t_1 \\
\quad \textbf{in}\ y
\end{array}
$$

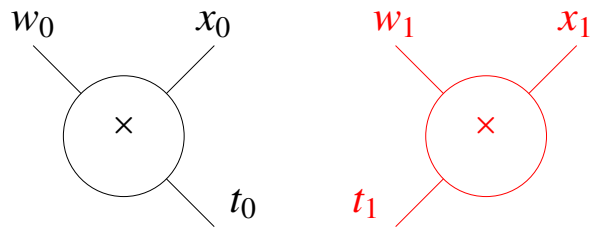
# A (Functional) Program is a (Neural) Network

$$
\begin{array}{l}
f\ [w_0, w_1]\ [x_0, x_1] \triangleq \\
\quad \textbf{let} \quad t_0 = w_0 \times x_0 \\
\qquad\quad t_1 = w_1 \times x_1 \\
\qquad\quad y = t_0 + t_1 \\
\quad \textbf{in}\ y
\end{array}
$$



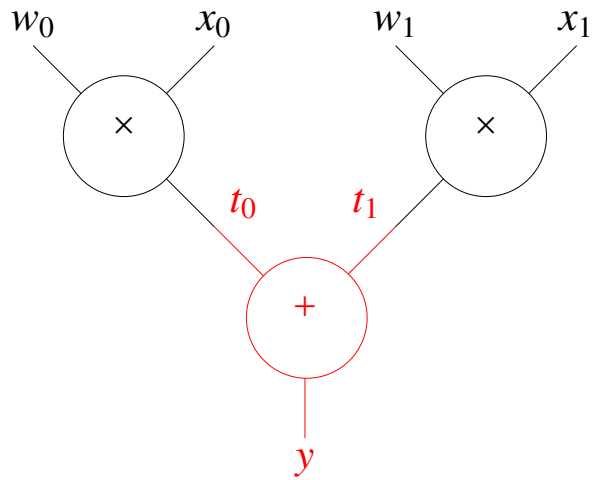
# A (Functional) Program is a (Neural) Network

$$
\begin{array}{l}
f\ [w_0, w_1]\ [x_0, x_1] \stackrel{\triangle}{=} \\
\quad \textbf{let} \quad t_0 = w_0 \times x_0 \\
\qquad\quad t_1 = w_1 \times x_1 \\
\qquad\quad y = t_0 + t_1 \\
\quad \textbf{in}\ y
\end{array}
$$



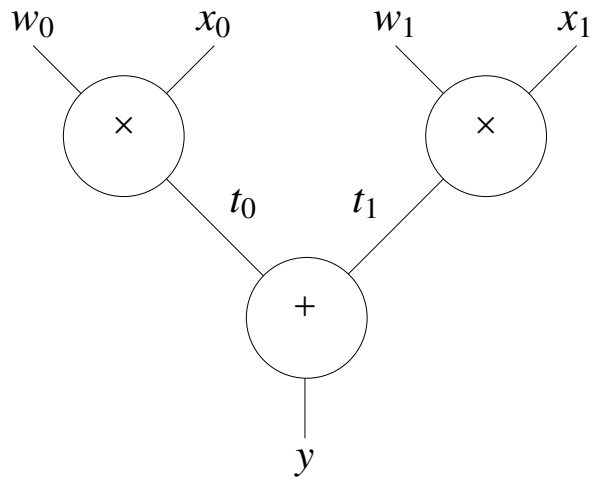
# A (Functional) Program is a (Neural) Network

$$
\begin{array}{l}
f\ [w_0, w_1]\ [x_0, x_1] \triangleq \\
\quad \textbf{let} \quad t_0 = w_0 \times x_0 \\
\qquad\quad t_1 = w_1 \times x_1 \\
\qquad\quad y = t_0 + t_1 \\
\quad \textbf{in}\ y
\end{array}
$$



# A (Functional) Program is a (Neural) Network

$$
\begin{array}{l}
f\ [w_0, w_1]\ [x_0, x_1] \triangleq \\
\quad \textbf{let} \quad t_0 = w_0 \times x_0 \\
\qquad\quad\ \ t_1 = w_1 \times x_1 \\
\qquad\quad\ \ y = t_0 + t_1 \\
\quad \textbf{in}\ y
\end{array}
$$



# Some Observations

# Some Observations

- Deep learning network ‘frameworks’ are domain specific (functional) programming languages.

# Some Observations

- Deep learning network ‘frameworks’ are domain specific (functional) programming languages.
- A deep neural network is a long running (functional) program.

# Some Observations

- Deep learning network ‘frameworks’ are domain specific (functional) programming languages.
- A deep neural network is a long running (functional) program.
- Can perform backpropagation on (functional) programs

# Some Observations

- Deep learning network ‘frameworks’ are domain specific (functional) programming languages.
- A deep neural network is a long running (functional) program.
- Can perform backpropagation on (functional) programs by having an execution of the program generate a network.

# Some Observations

- Deep learning network ‘frameworks’ are domain specific (functional) programming languages.
- A deep neural network is a long running (functional) program.
- Can perform backpropagation on (functional) programs
  by having an execution of the program generate a network.
  This is called reverse-mode automatic differentiation (AD).

# A (Brief) History of Backpropagation aka Reverse-Mode AD

# A (Brief) History of Backpropagation aka Reverse-Mode AD

D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature, 323:533–536, 1986.

# A (Brief) History of Backpropagation aka Reverse-Mode AD

D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature, 323:533–536, 1986.

B. Speelpenning, *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

# A (Brief) History of Backpropagation aka Reverse-Mode AD

D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature, 323:533–536, 1986.

B. Speelpenning, *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

P.J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Harvard University, 1974.

# A (Brief) History of Backpropagation aka Reverse-Mode AD

D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature, 323:533–536, 1986.

B. Speelpenning, *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

P.J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Harvard University, 1974.

G.M. Ostrovskii, Y.M. Volin, and W.W. Borisov, *Über die Berechnung von Ableitungen, Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg*, 13(4):382–384, 1971.

# A (Brief) History of Backpropagation aka Reverse-Mode AD

D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature, 323:533–536, 1986.

B. Speelpenning, *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

P.J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Harvard University, 1974.

G.M. Ostrovskii, Y.M. Volin, and W.W. Borisov, *Über die Berechnung von Ableitungen, Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg*, 13(4):382–384, 1971.

S. Linnainmaa, *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors* (in Finnish), Department of Computer Science, University of Helsinki, 1970.

# A (Brief) History of Backpropagation aka Reverse-Mode AD

D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature, 323:533–536, 1986.

B. Speelpenning, *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

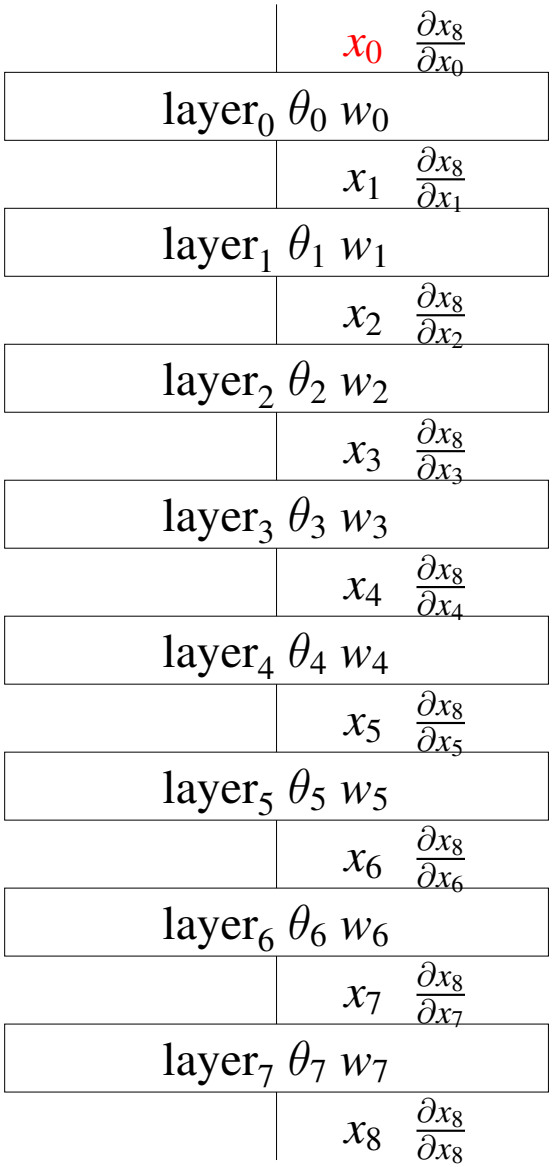
P.J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Harvard University, 1974.

G.M. Ostrovskii, Y.M. Volin, and W.W. Borisov, *Über die Berechnung von Ableitungen, Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg*, 13(4):382–384, 1971.

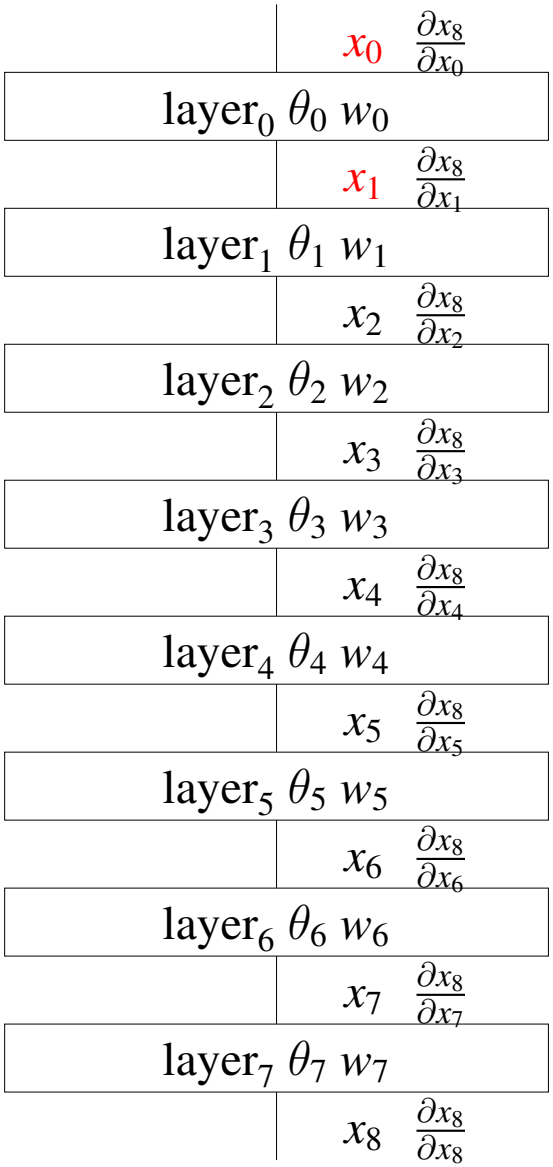
S. Linnainmaa, *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors* (in Finnish), Department of Computer Science, University of Helsinki, 1970.

A.E. Bryson, Jr. and Y.-C. Ho, *Applied optimal control*, Blaisdell, 1969.

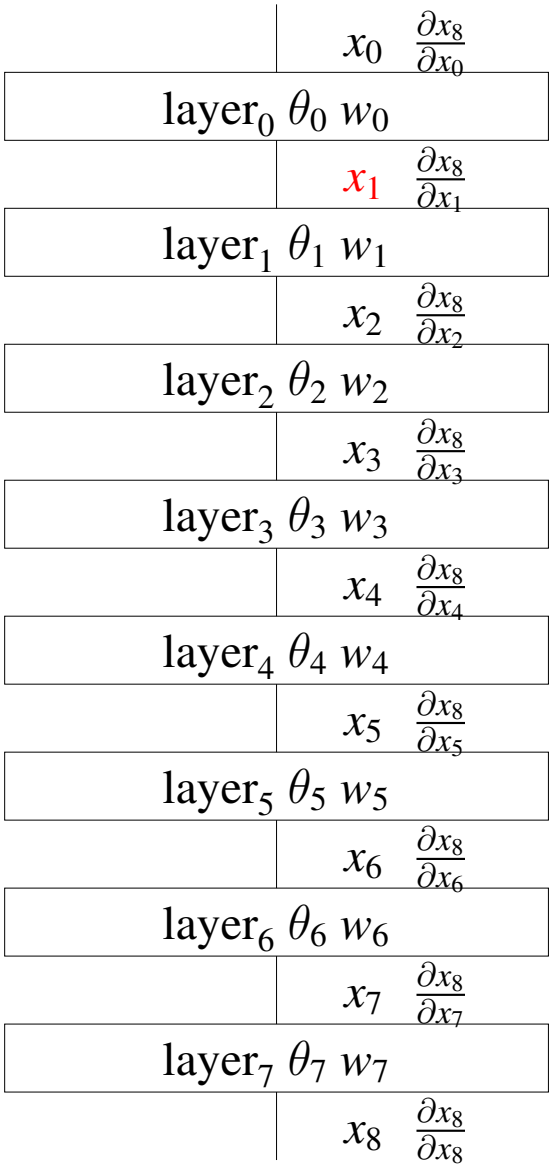
# Evaluating a Neural Network



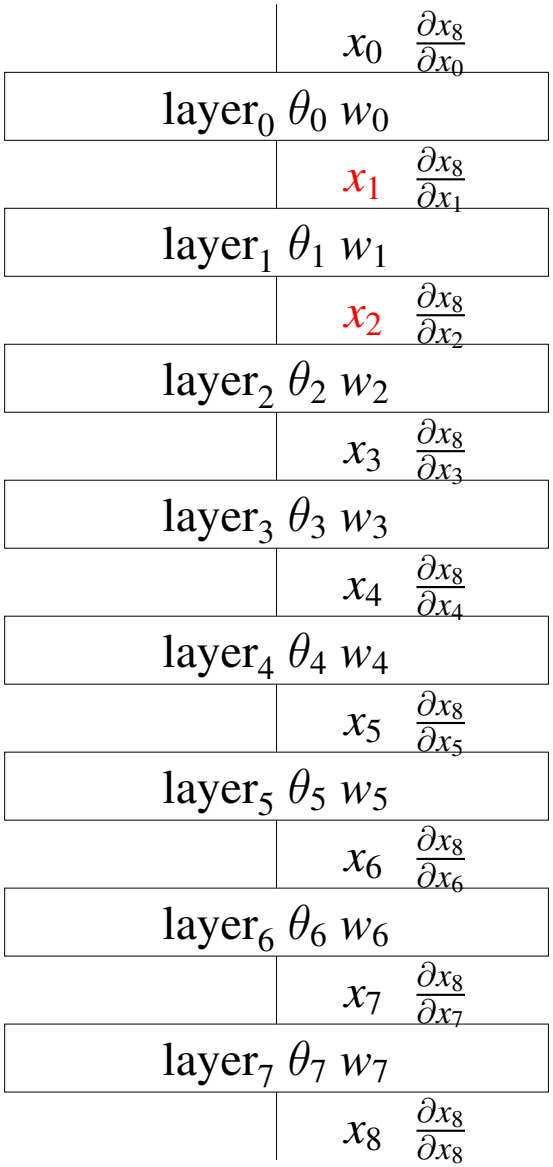
# Evaluating a Neural Network



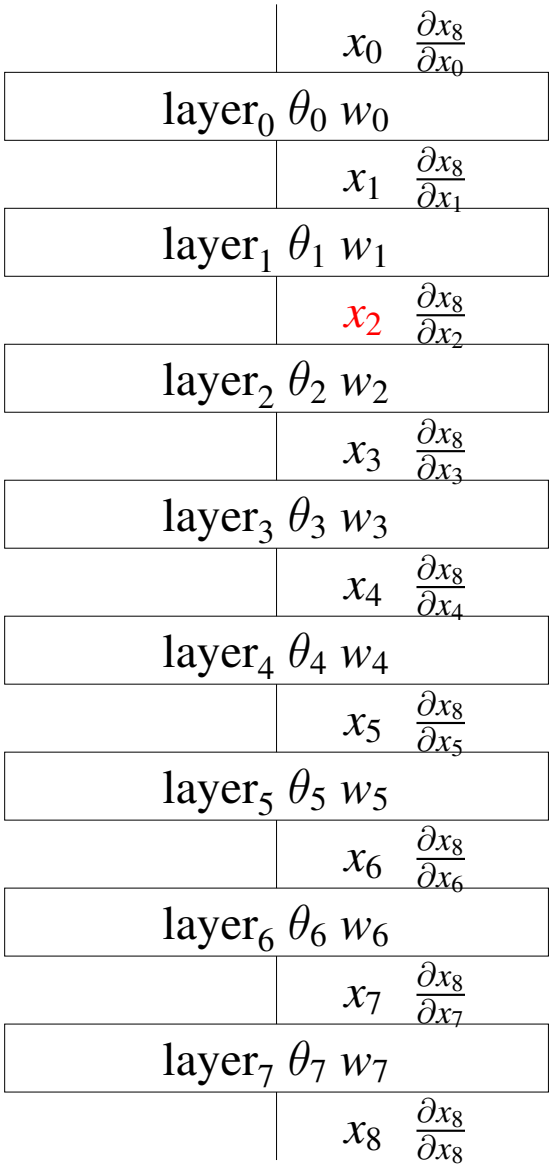
# Evaluating a Neural Network



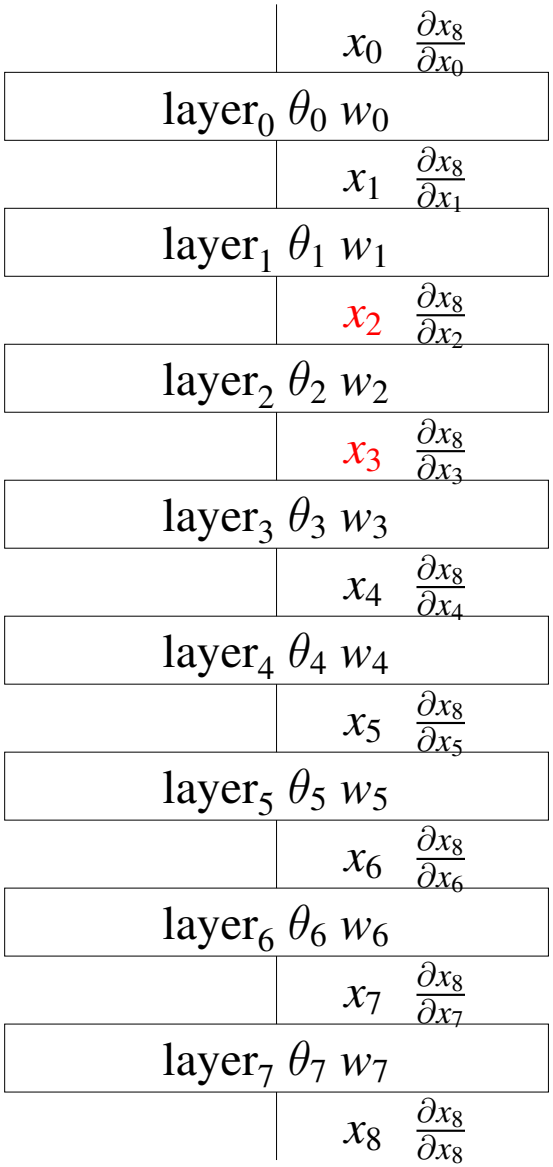
# Evaluating a Neural Network



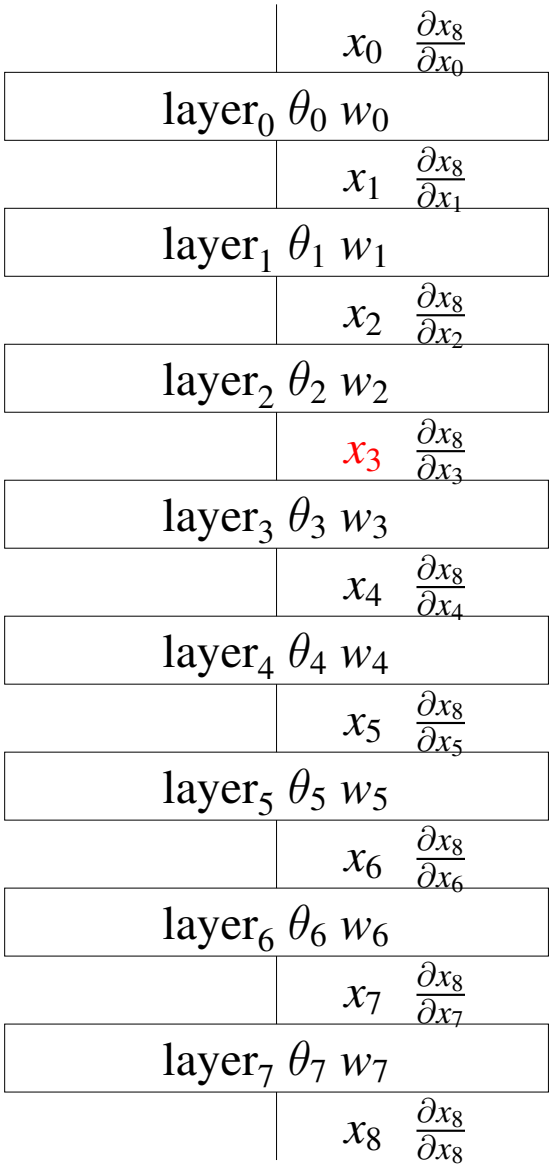
# Evaluating a Neural Network



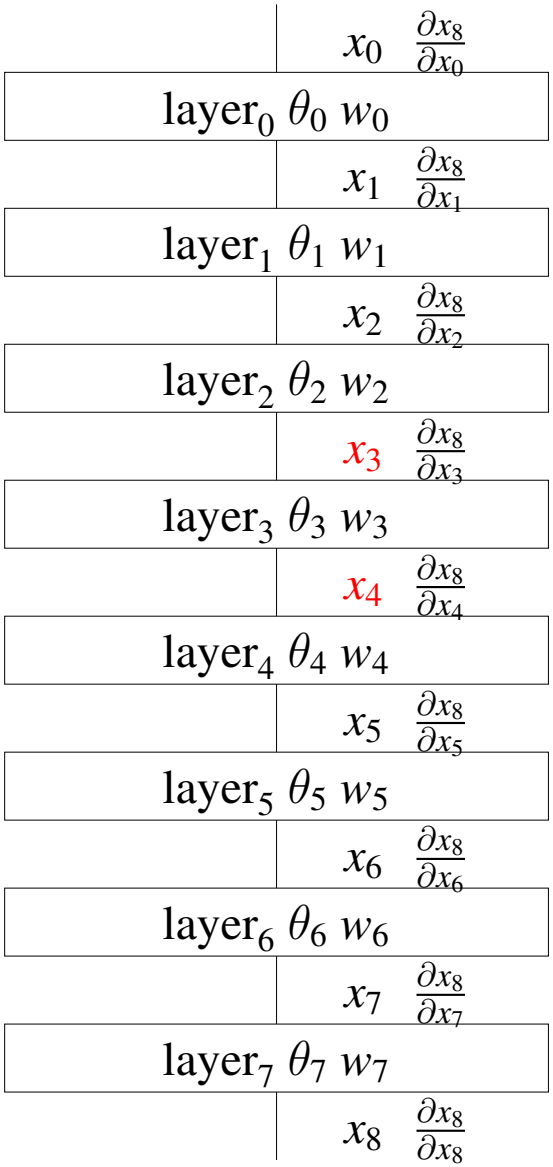
# Evaluating a Neural Network



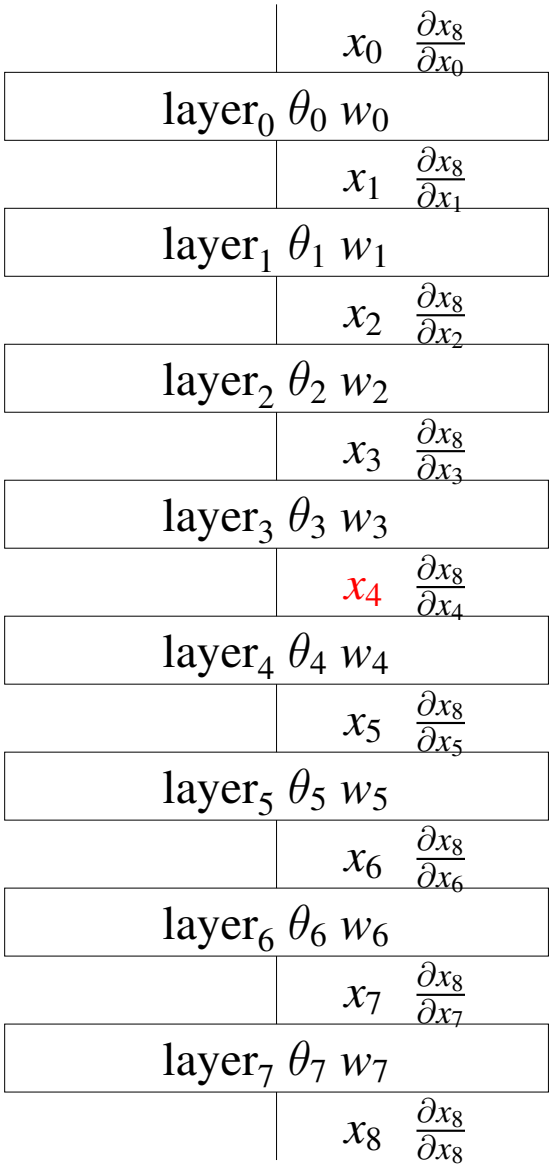
# Evaluating a Neural Network



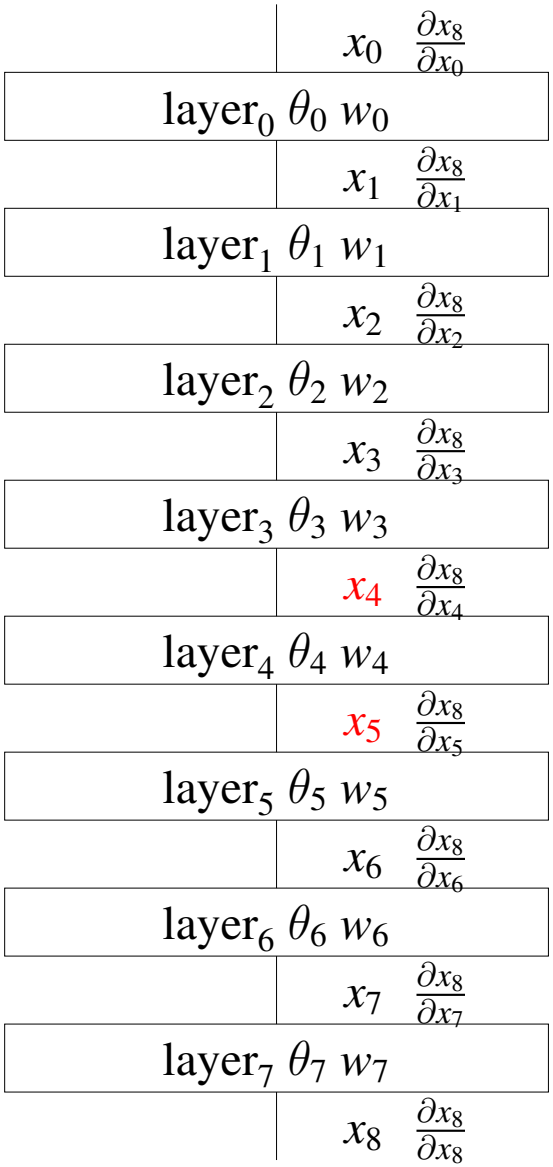
# Evaluating a Neural Network



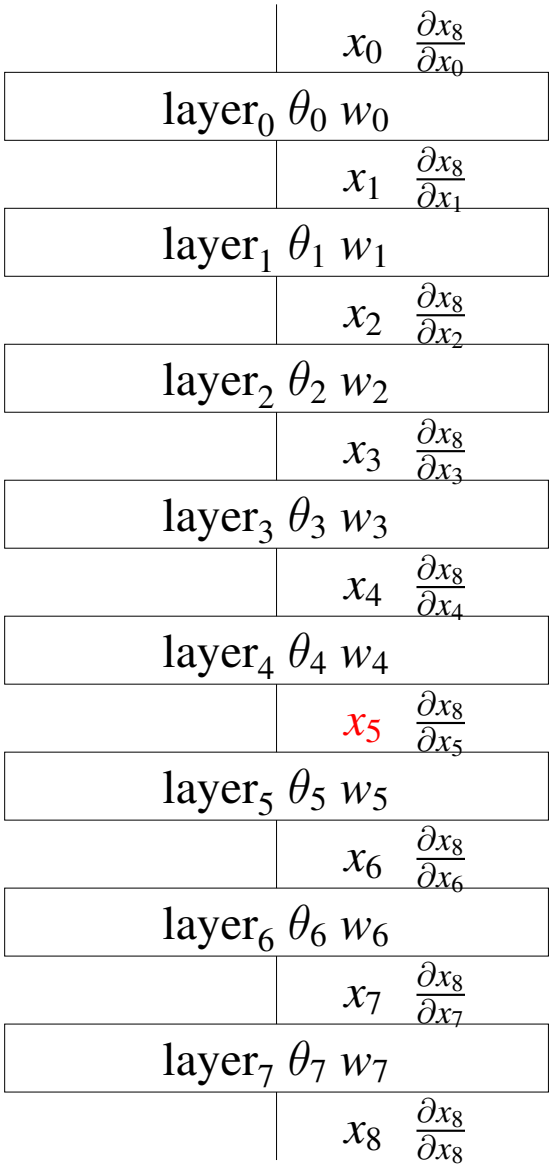
# Evaluating a Neural Network



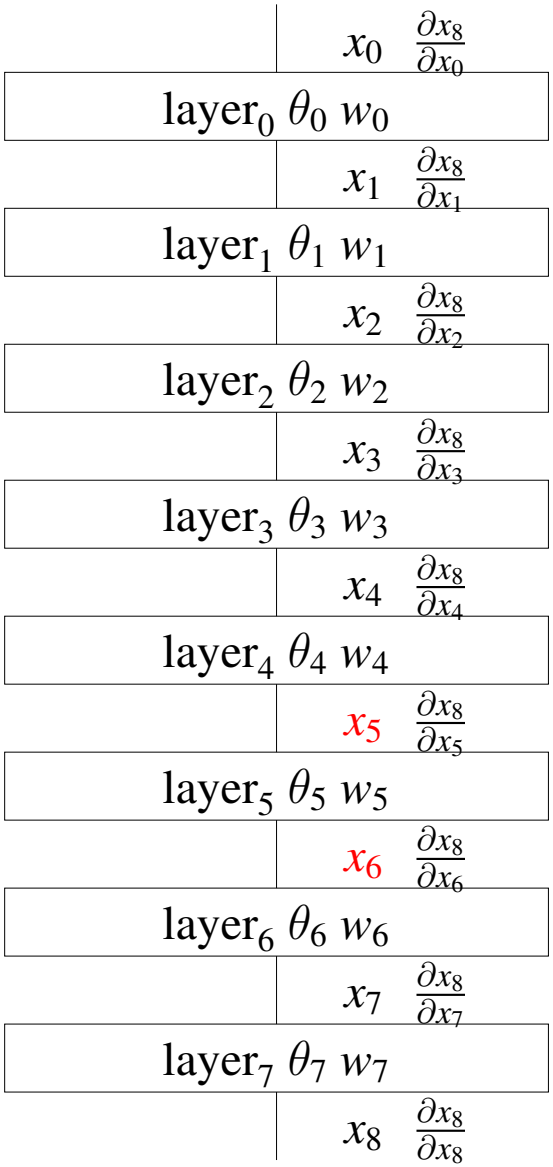
# Evaluating a Neural Network



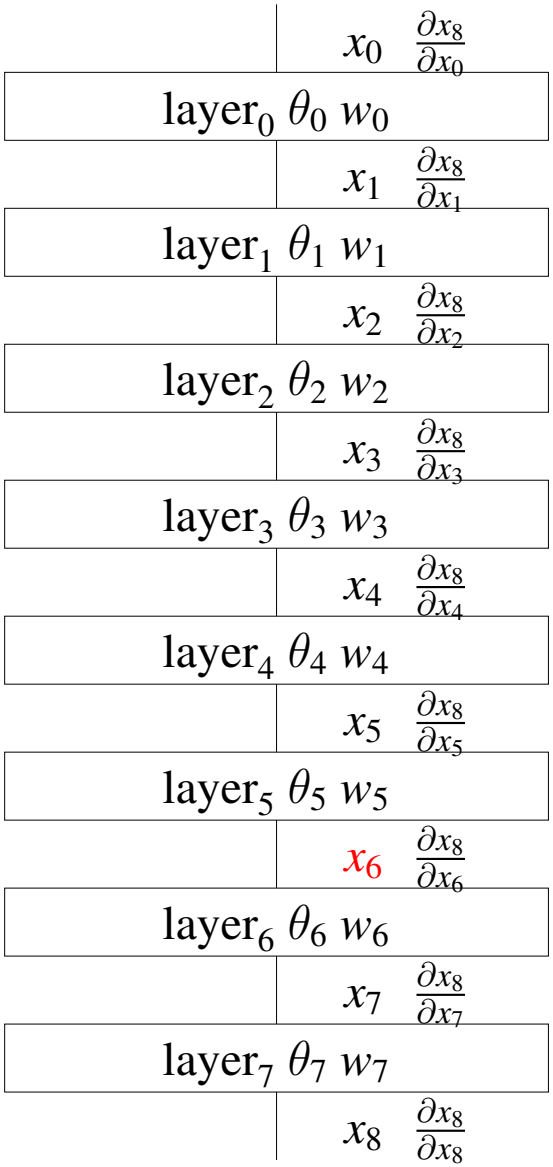
# Evaluating a Neural Network



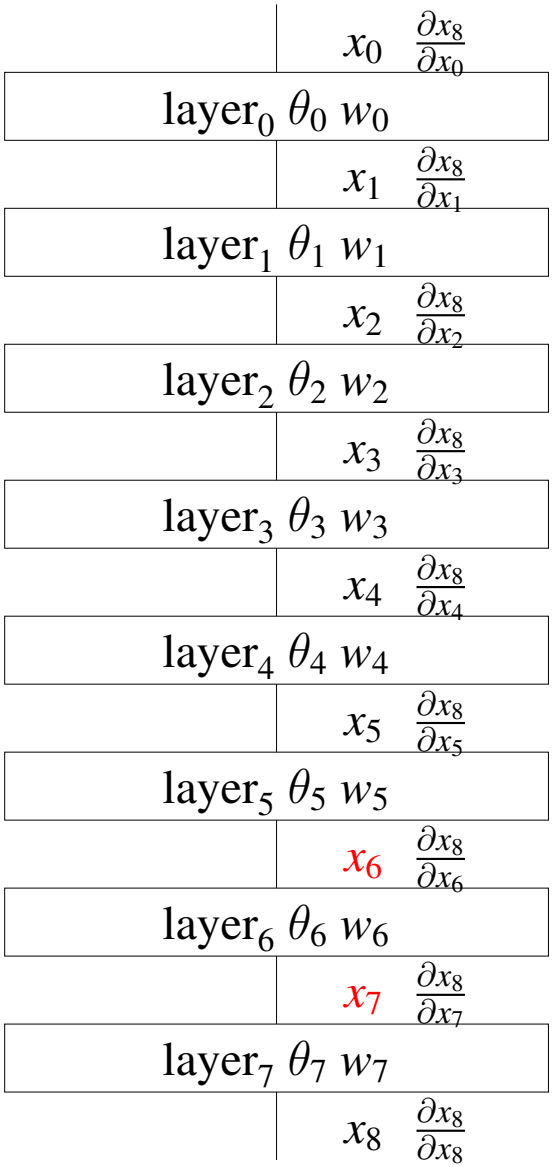
# Evaluating a Neural Network



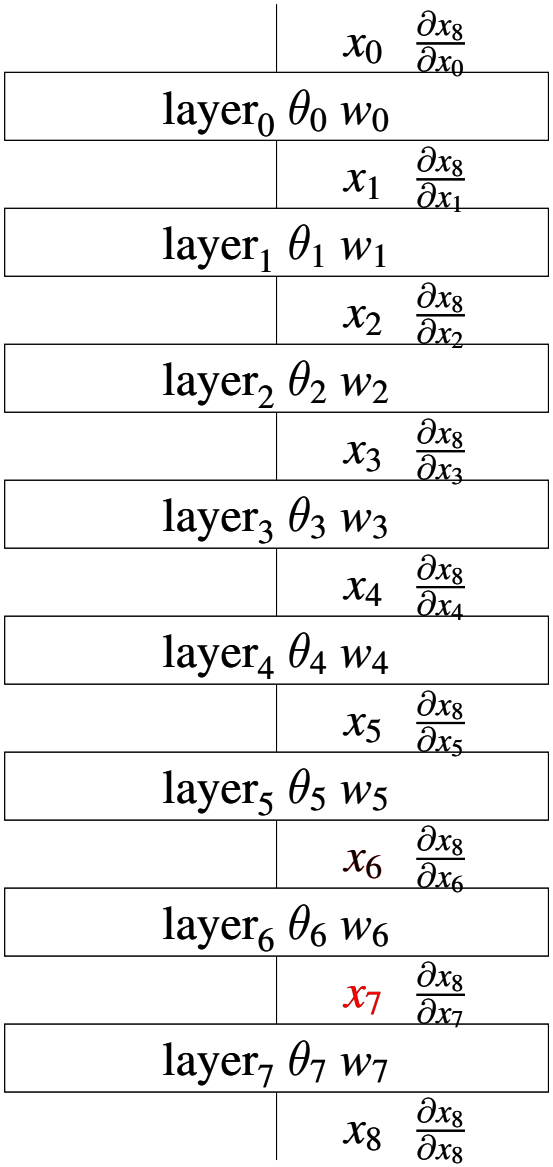
# Evaluating a Neural Network



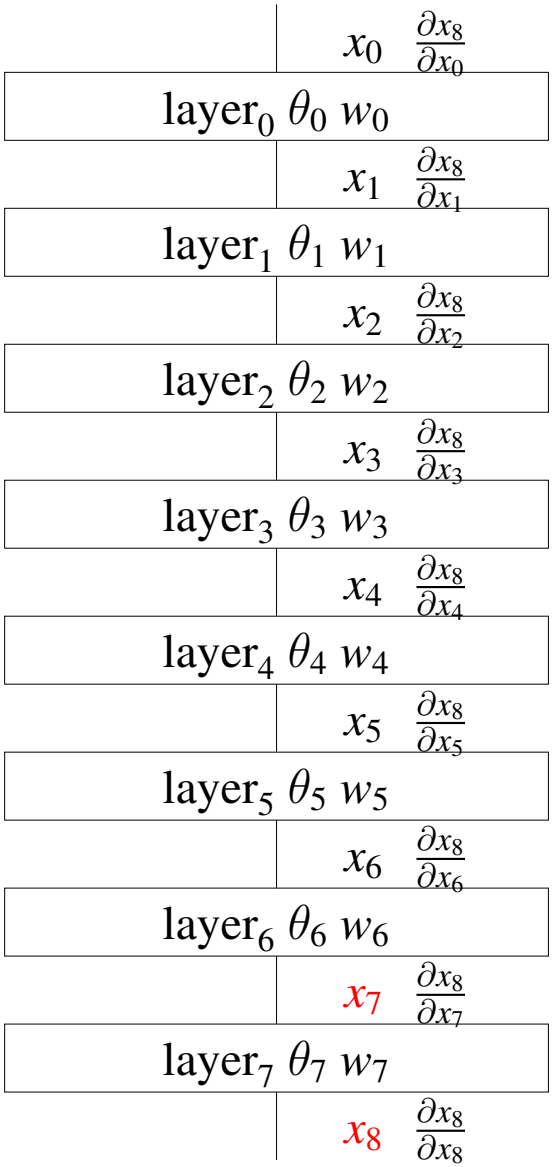
# Evaluating a Neural Network



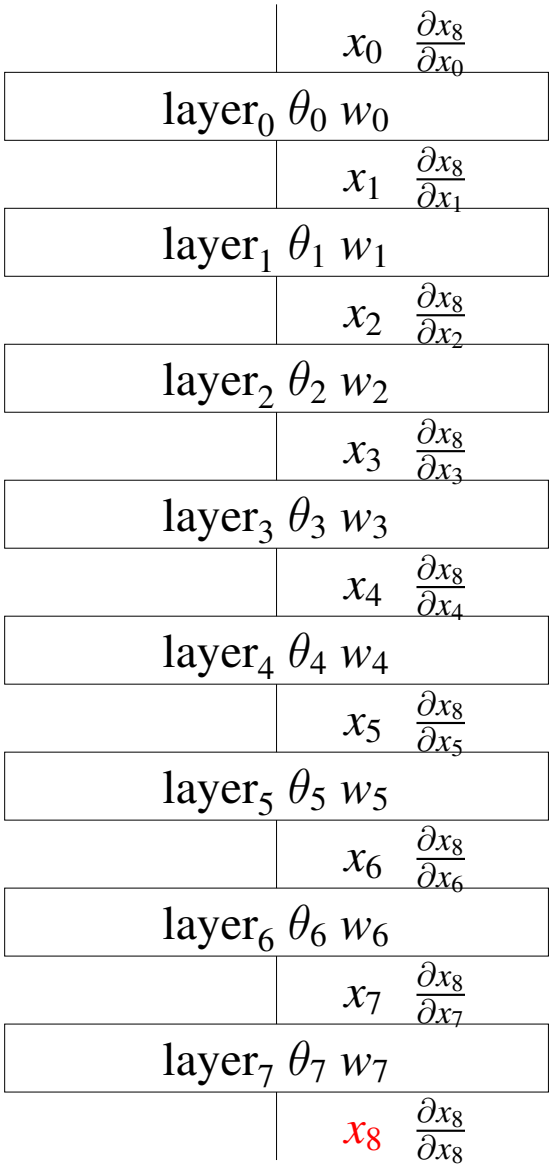
# Evaluating a Neural Network



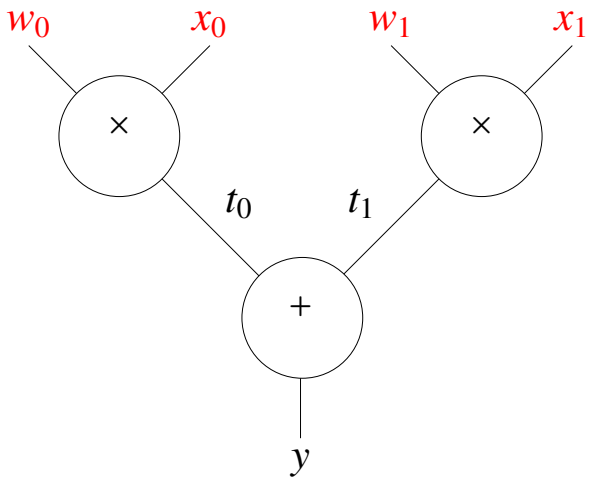
# Evaluating a Neural Network



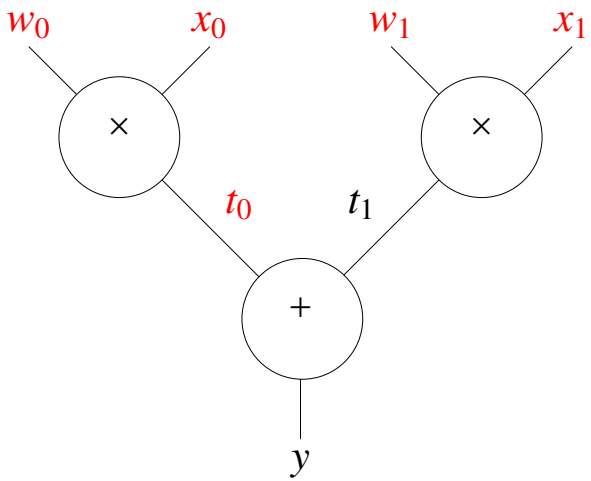
# Evaluating a Neural Network



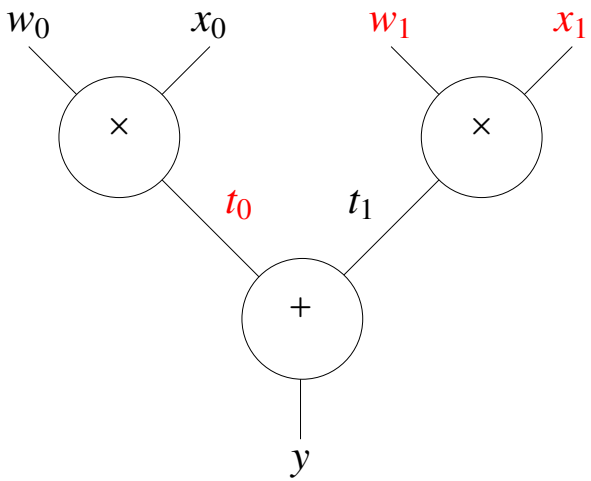
# Evaluating a Network



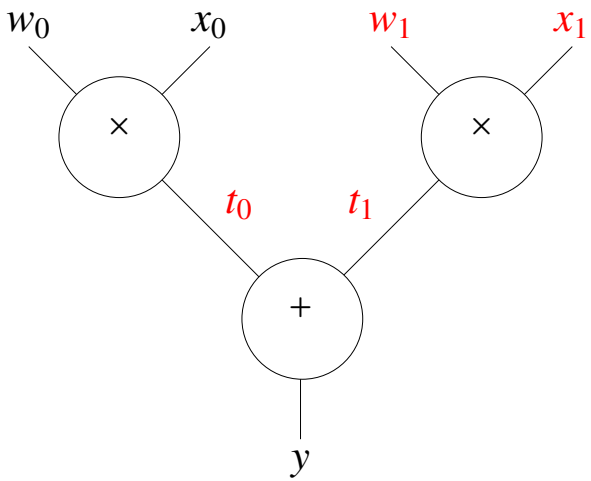
# Evaluating a Network



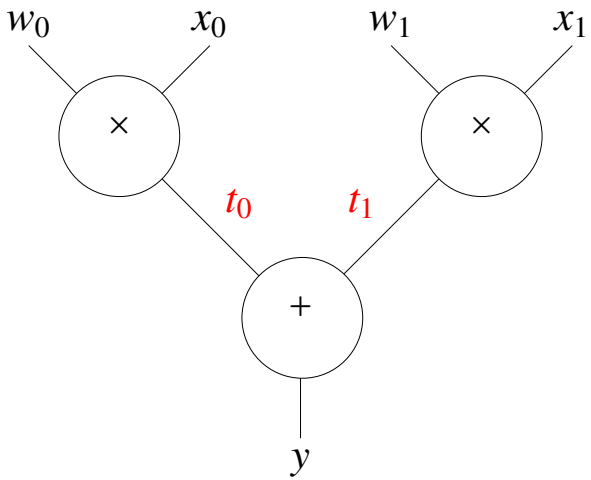
# Evaluating a Network



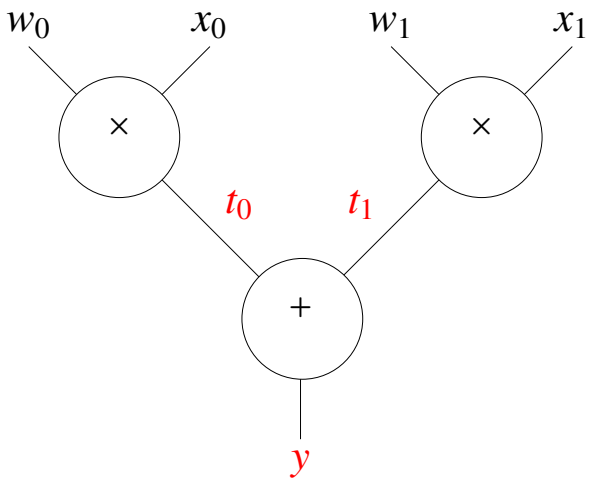
# Evaluating a Network



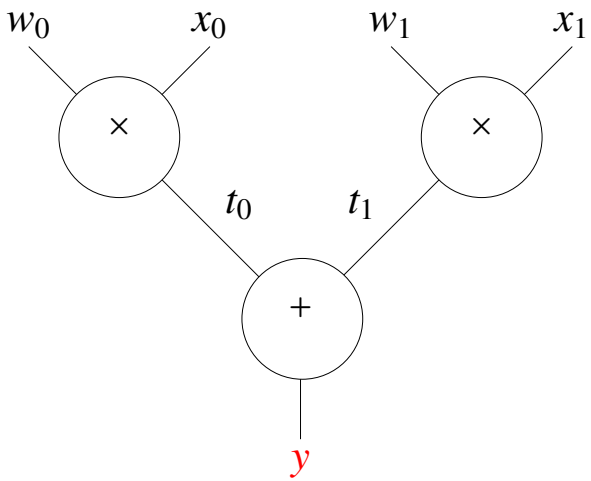
# Evaluating a Network



# Evaluating a Network



# Evaluating a Network



# Some Observations

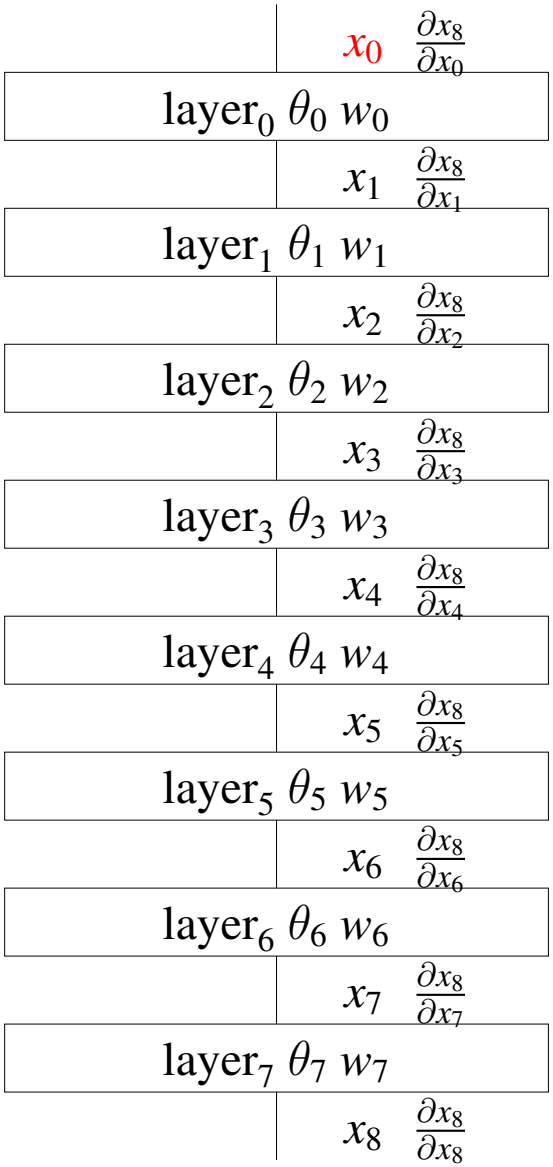
# Some Observations

- Only need to store live variables.

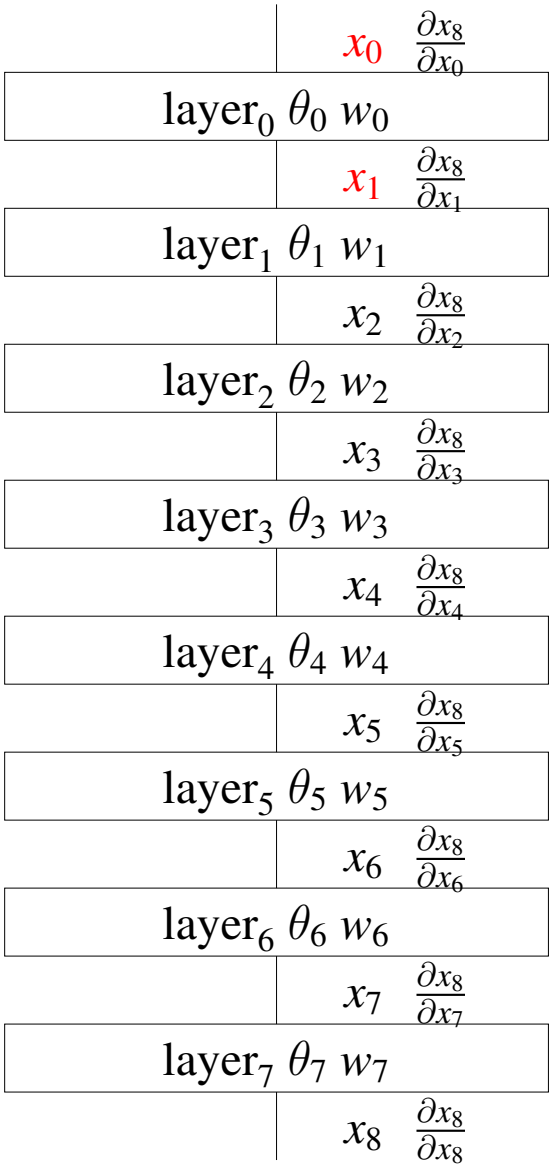
# Some Observations

- Only need to store live variables.
- Most deep learning frameworks store all intermediate variables to allow subsequent backpropagation.

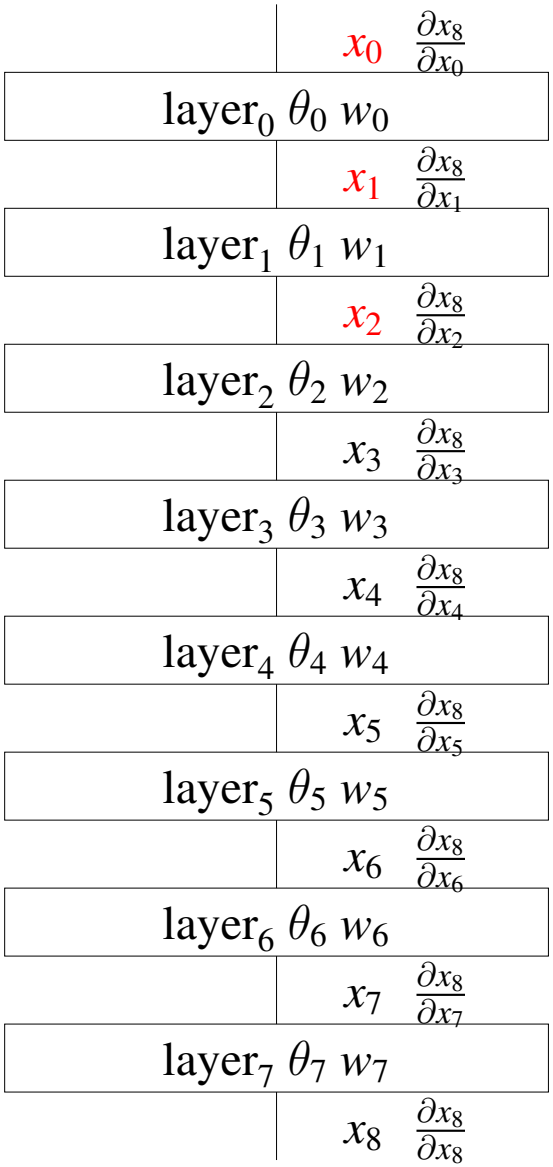
# Evaluating a Neural Network to Allow Subsequent Backpropagation



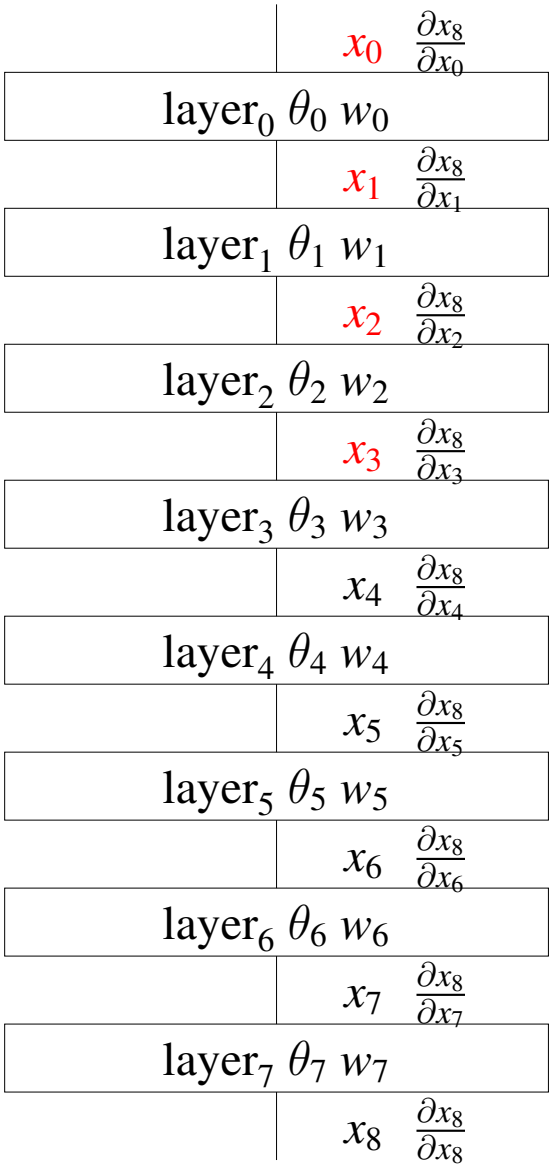
# Evaluating a Neural Network to Allow Subsequent Backpropagation



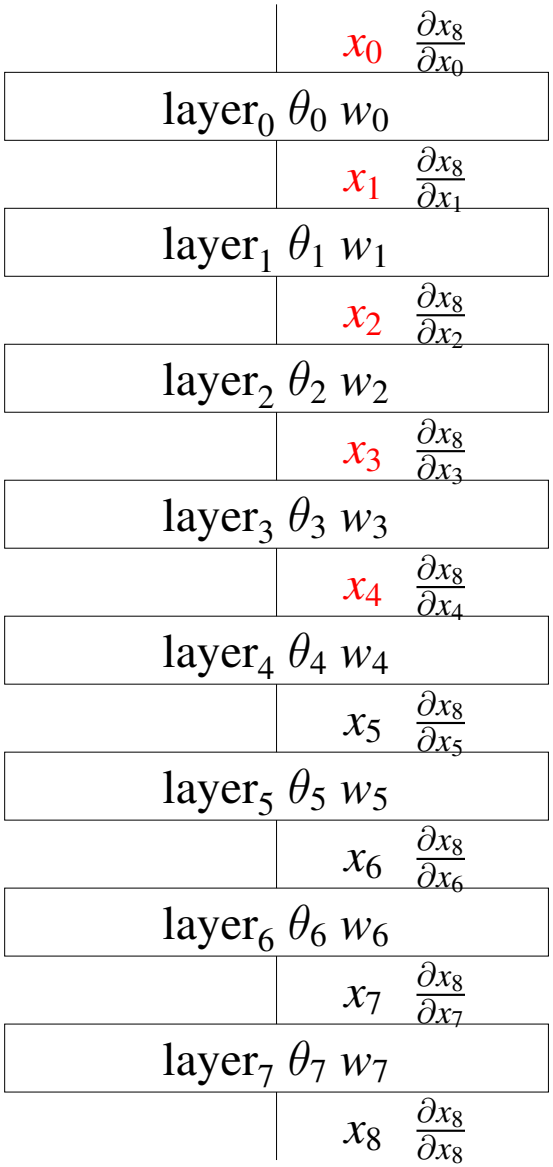
# Evaluating a Neural Network to Allow Subsequent Backpropagation



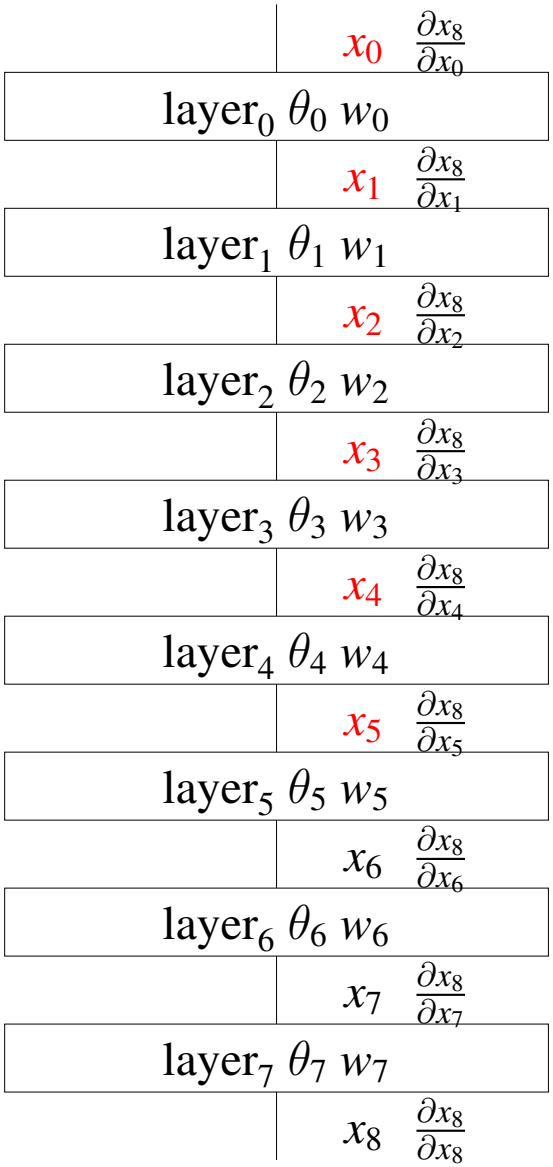
# Evaluating a Neural Network to Allow Subsequent Backpropagation



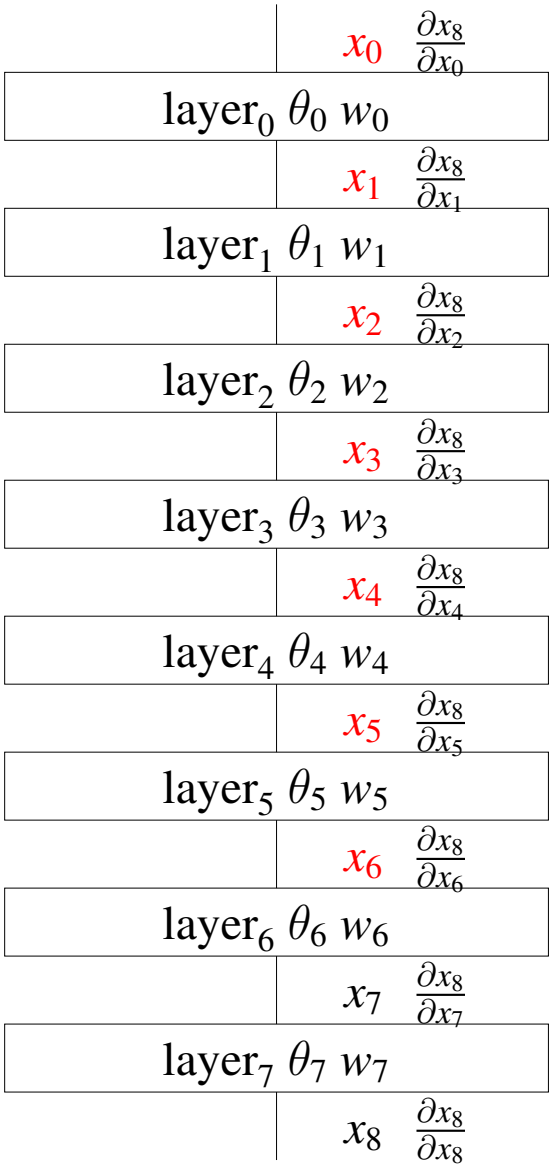
# Evaluating a Neural Network to Allow Subsequent Backpropagation



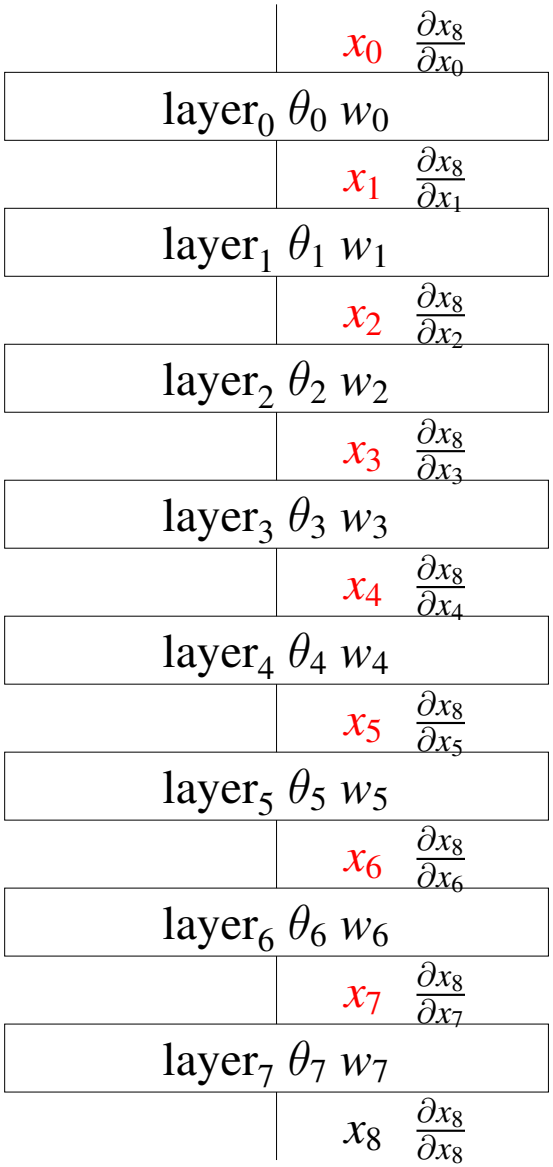
# Evaluating a Neural Network to Allow Subsequent Backpropagation



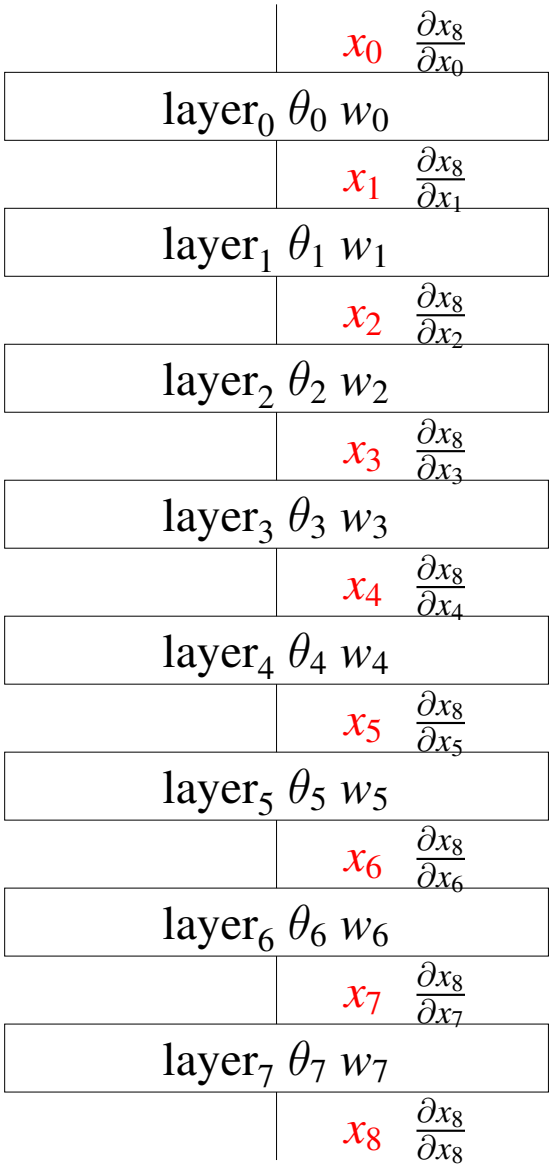
# Evaluating a Neural Network to Allow Subsequent Backpropagation



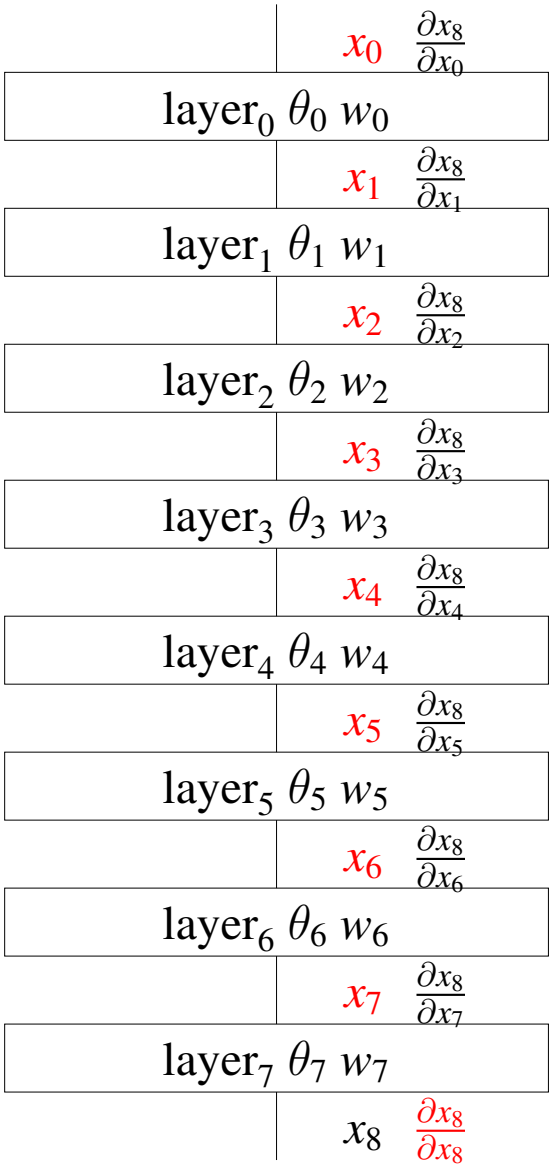
# Evaluating a Neural Network to Allow Subsequent Backpropagation



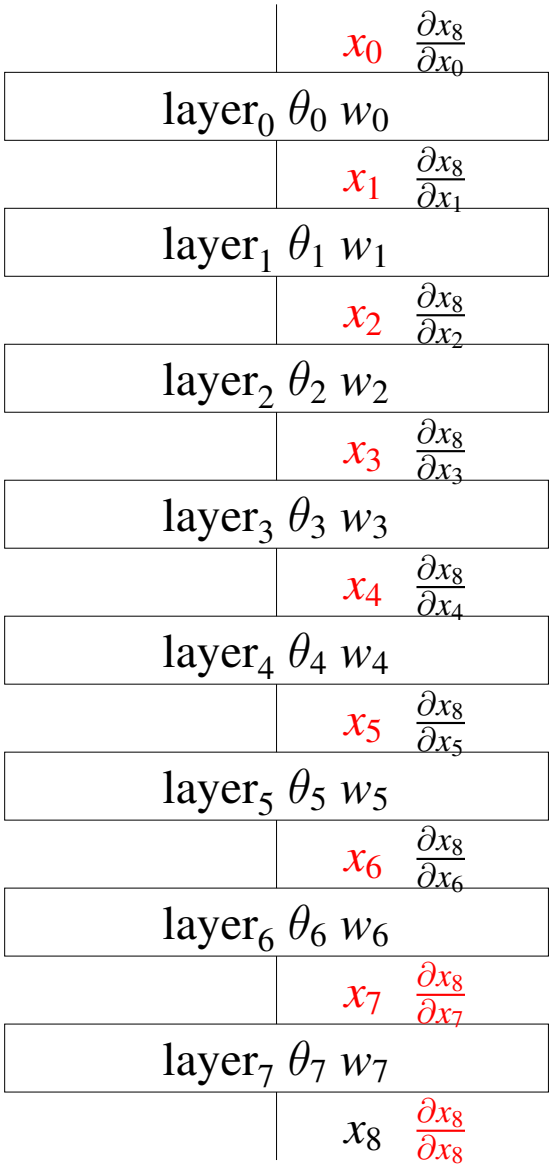
# Evaluating a Neural Network to Allow Subsequent Backpropagation



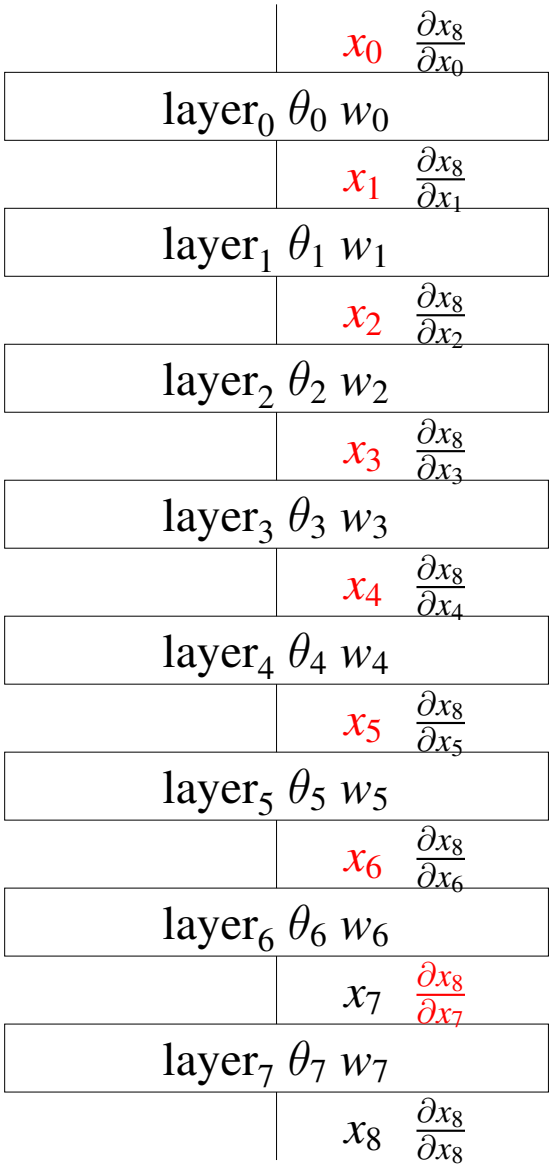
# Evaluating a Neural Network to Allow Subsequent Backpropagation



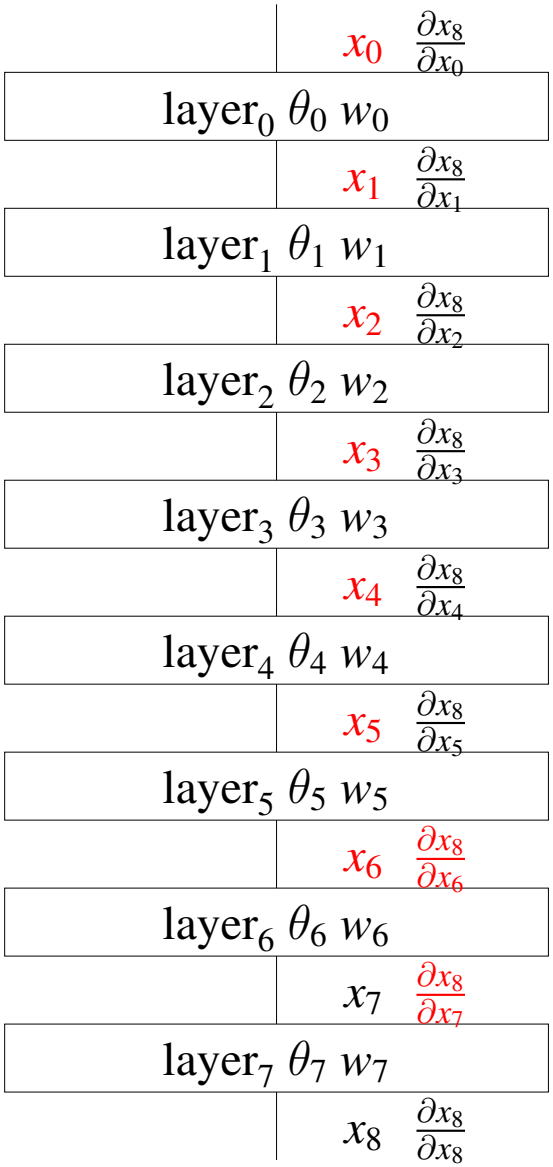
# Evaluating a Neural Network to Allow Subsequent Backpropagation



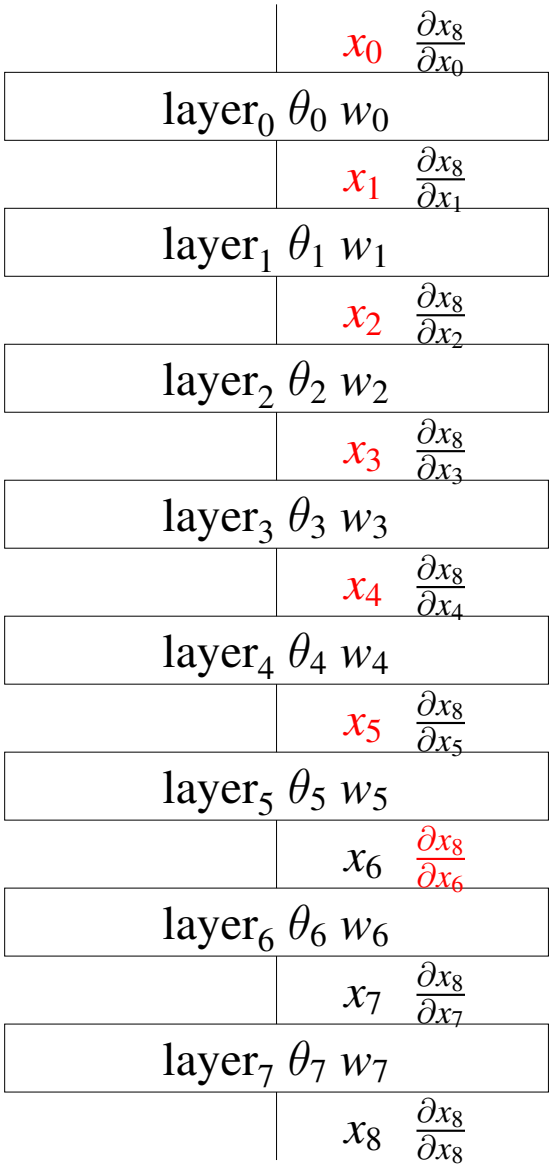
# Evaluating a Neural Network to Allow Subsequent Backpropagation



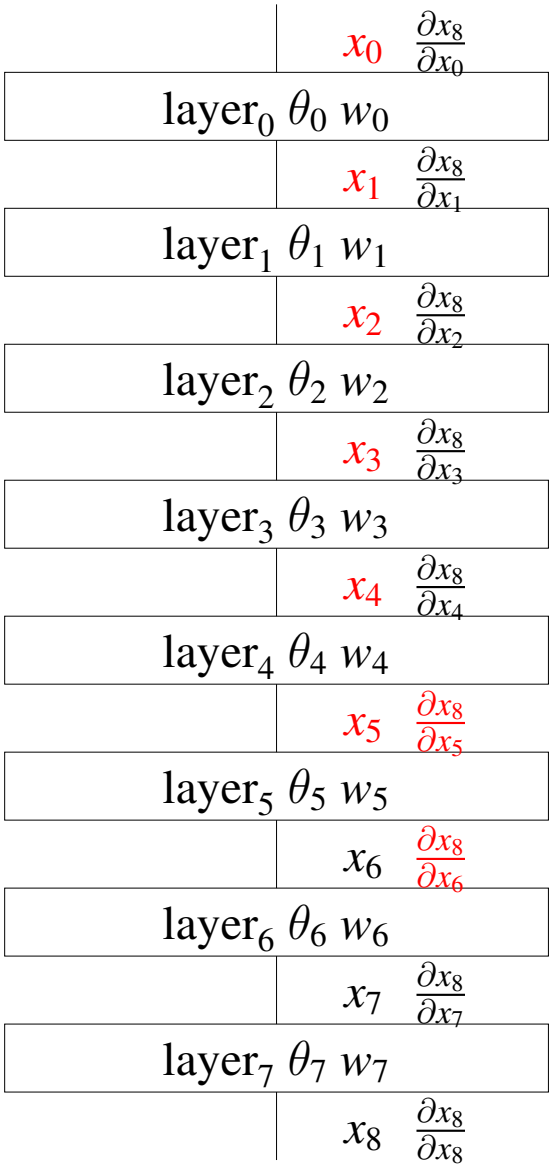
# Evaluating a Neural Network to Allow Subsequent Backpropagation



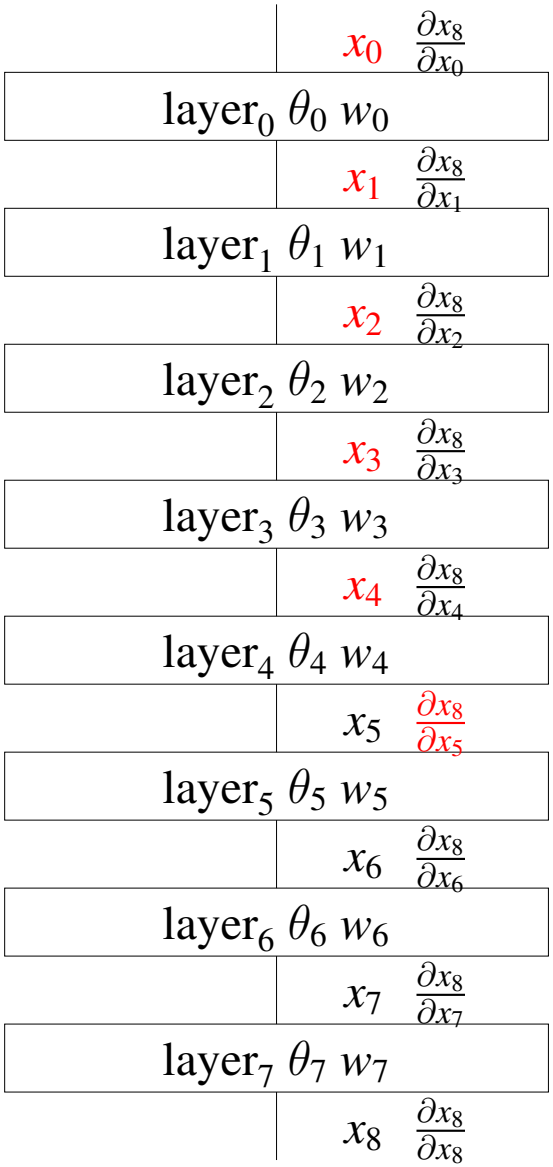
# Evaluating a Neural Network to Allow Subsequent Backpropagation



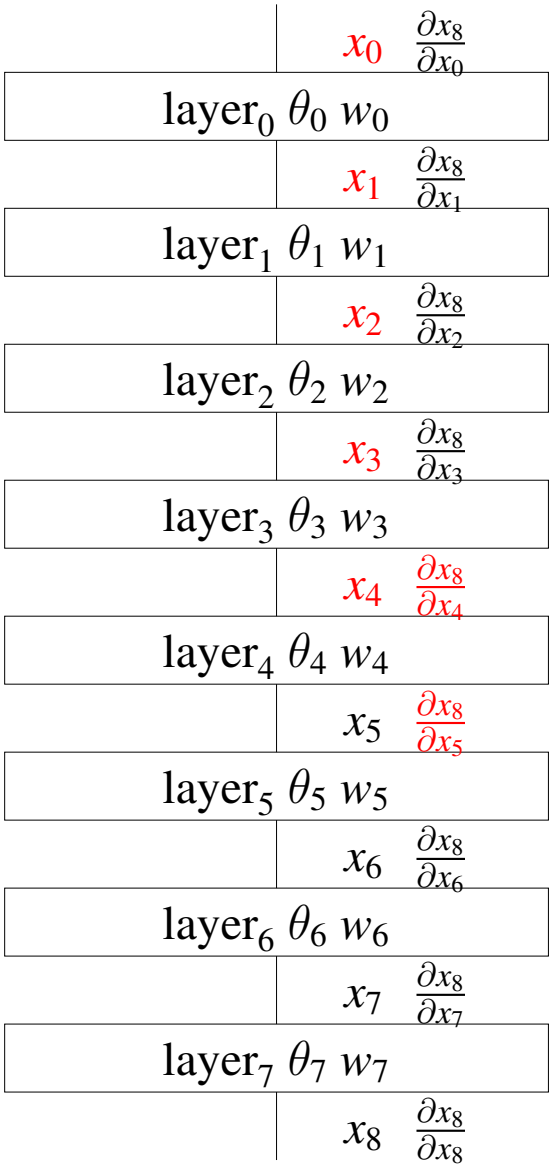
# Evaluating a Neural Network to Allow Subsequent Backpropagation



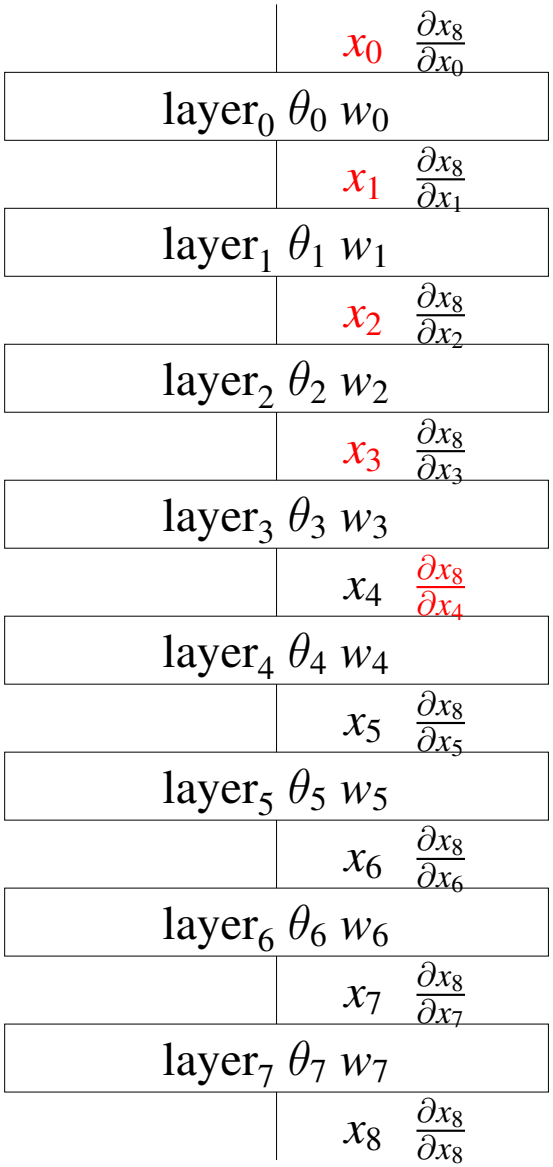
# Evaluating a Neural Network to Allow Subsequent Backpropagation



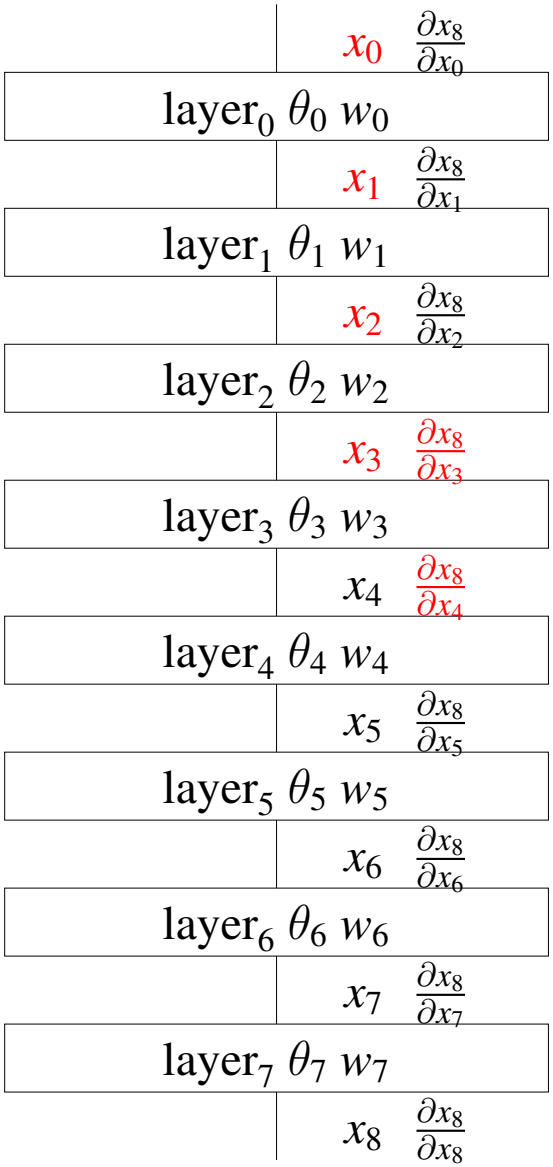
# Evaluating a Neural Network to Allow Subsequent Backpropagation



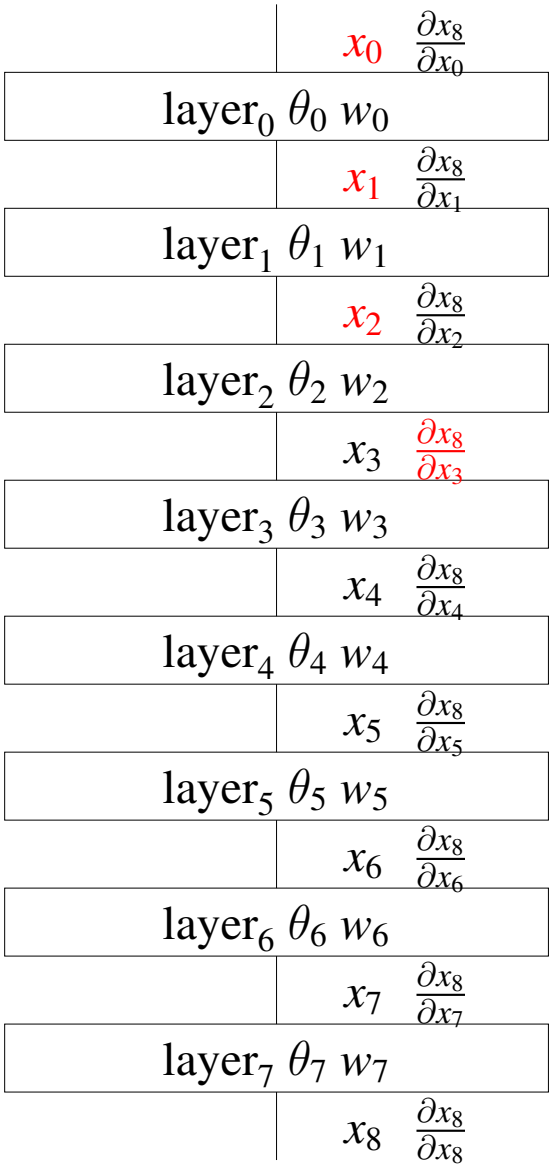
# Evaluating a Neural Network to Allow Subsequent Backpropagation



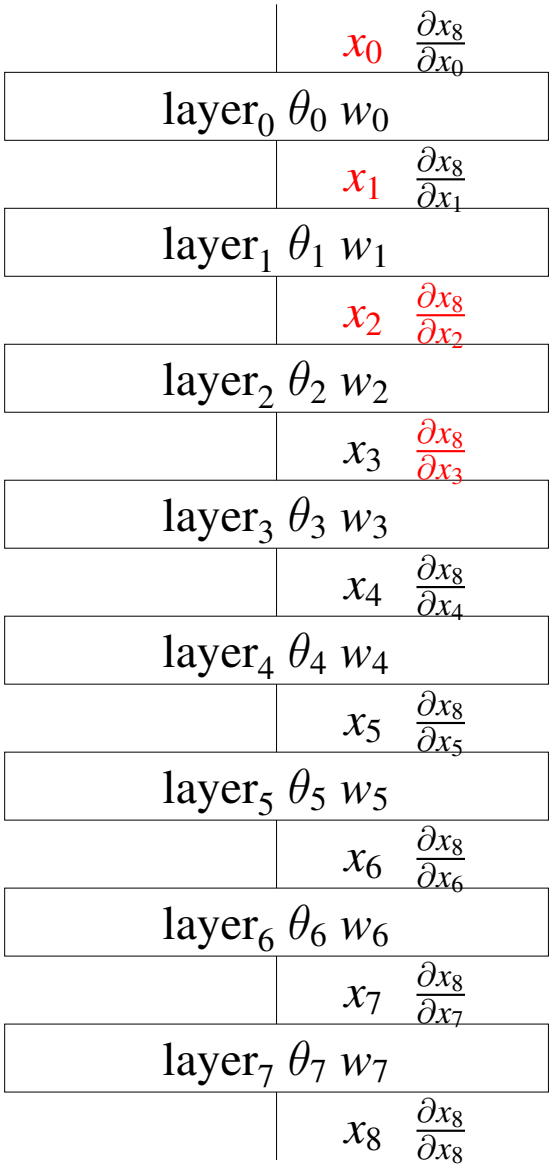
# Evaluating a Neural Network to Allow Subsequent Backpropagation



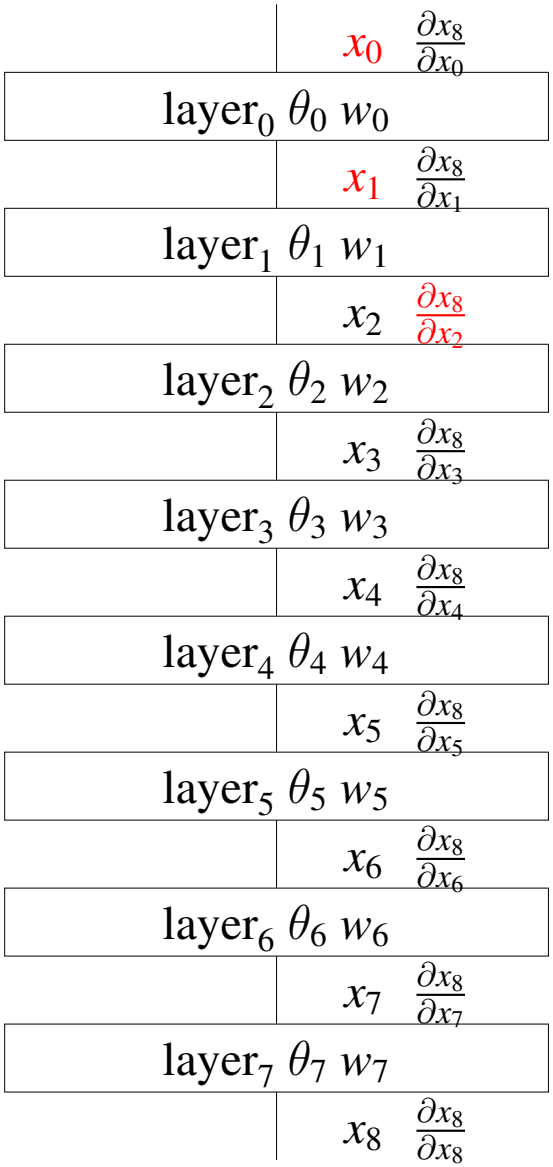
# Evaluating a Neural Network to Allow Subsequent Backpropagation



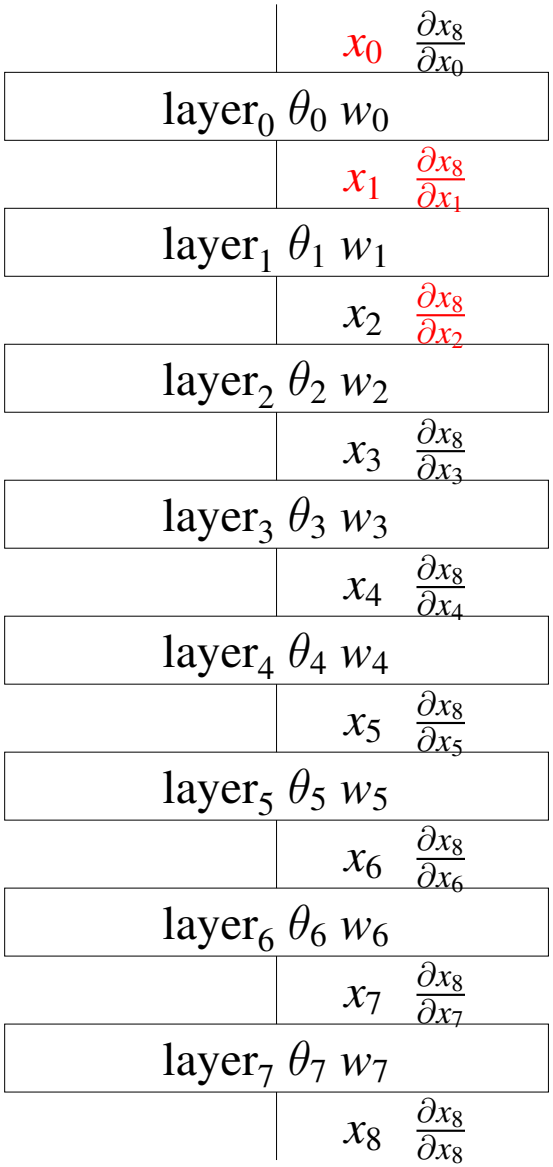
# Evaluating a Neural Network to Allow Subsequent Backpropagation



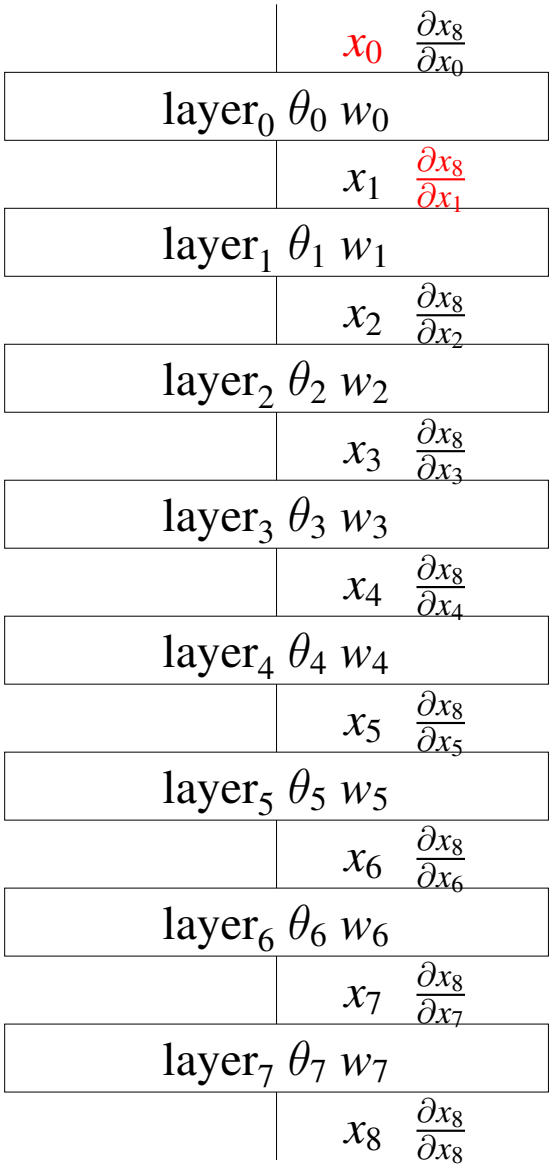
# Evaluating a Neural Network to Allow Subsequent Backpropagation



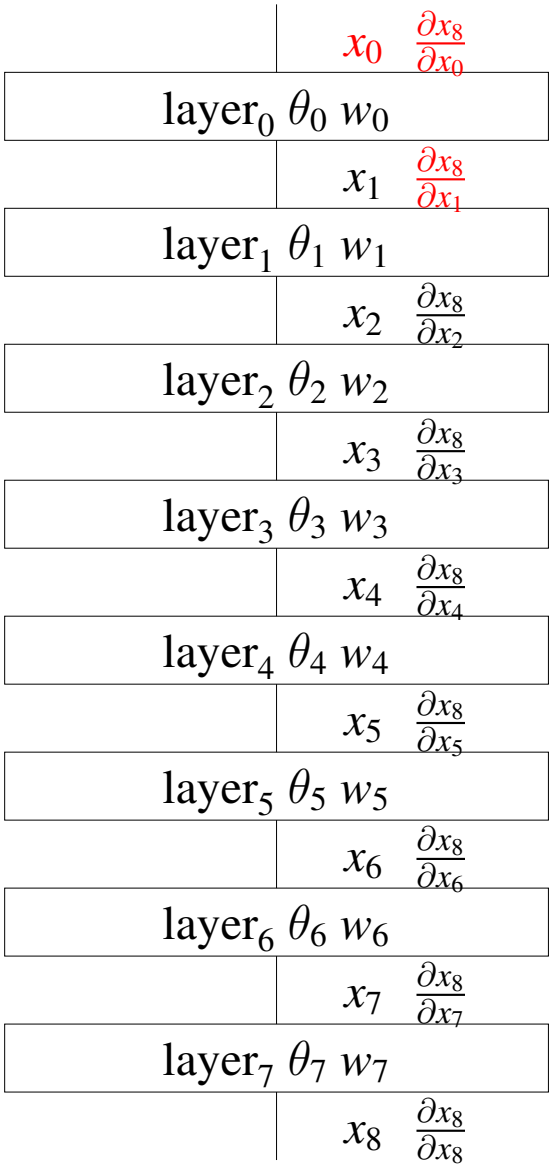
# Evaluating a Neural Network to Allow Subsequent Backpropagation



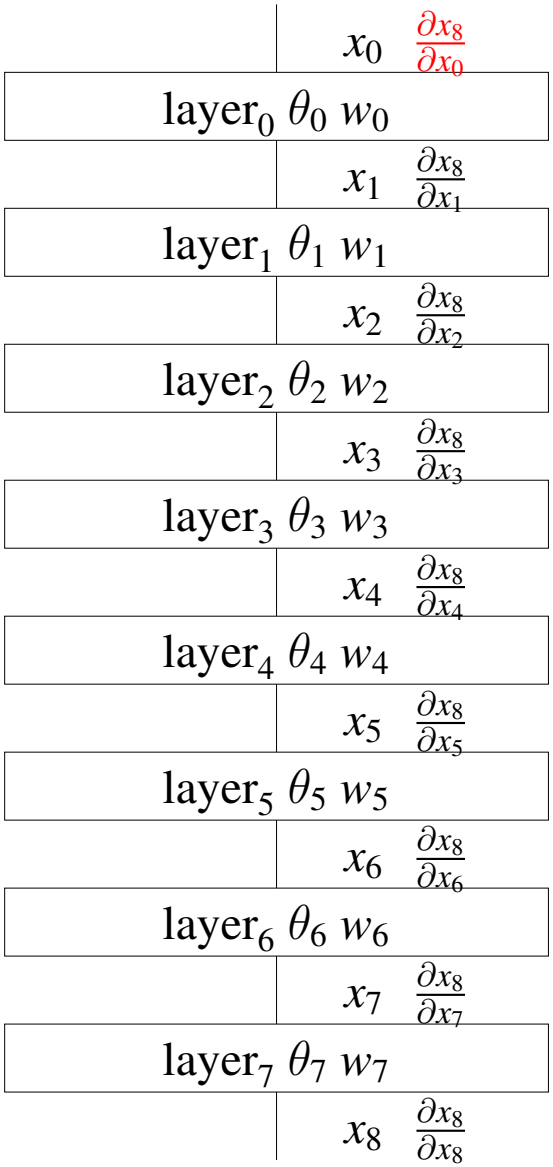
# Evaluating a Neural Network to Allow Subsequent Backpropagation



# Evaluating a Neural Network to Allow Subsequent Backpropagation



# Evaluating a Neural Network to Allow Subsequent Backpropagation



# Some Observations

# Some Observations

- Only need to store live variables from forward pass until they are used in reverse pass.

# Some Observations

- Only need to store live variables from forward pass until they are used in reverse pass.
- Only need to store live variables during reverse pass.

# Some Observations

- Only need to store live variables from forward pass until they are used in reverse pass.
- Only need to store live variables during reverse pass.
- Most deep learning frameworks store all intermediate forward and reverse pass variables for simplicity of implementation.

# Some Observations

- Only need to store live variables from forward pass until they are used in reverse pass.
- Only need to store live variables during reverse pass.
- Most deep learning frameworks store all intermediate forward and reverse pass variables for simplicity of implementation.
- It doesn't matter because storage use is dominated by maximal use.

# Some Observations

- Only need to store live variables from forward pass until they are used in reverse pass.
- Only need to store live variables during reverse pass.
- Most deep learning frameworks store all intermediate forward and reverse pass variables for simplicity of implementation.
- It doesn't matter because storage use is dominated by maximal use.
- Maximal use is proportional to the depth of the network *i.e.*, the running time of the program.

# Complexity of Reverse-Mode AD

# Complexity of Reverse-Mode AD

- If running time of primal is $O(t)$

# Complexity of Reverse-Mode AD

- If running time of primal is $O(t)$
  and primal has maximal live storage $O(w)$

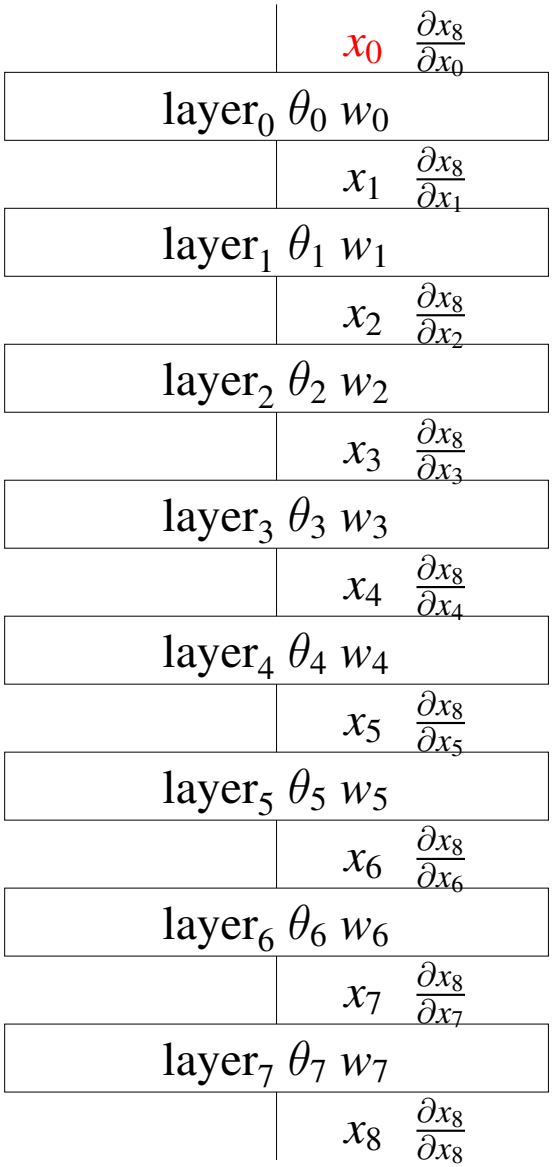
# Complexity of Reverse-Mode AD

- If running time of primal is $O(t)$
  and primal has maximal live storage $O(w)$
- then reverse mode takes $O(wt)$ space

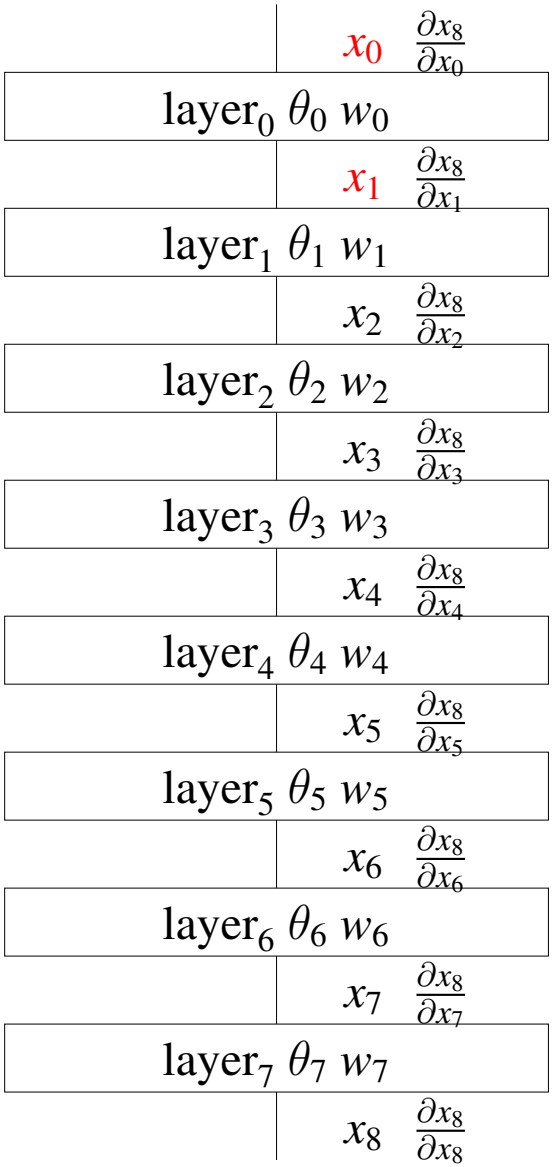
# Complexity of Reverse-Mode AD

- If running time of primal is $O(t)$
  and primal has maximal live storage $O(w)$
- then reverse mode takes $O(wt)$ space
  and $O(t)$ time.

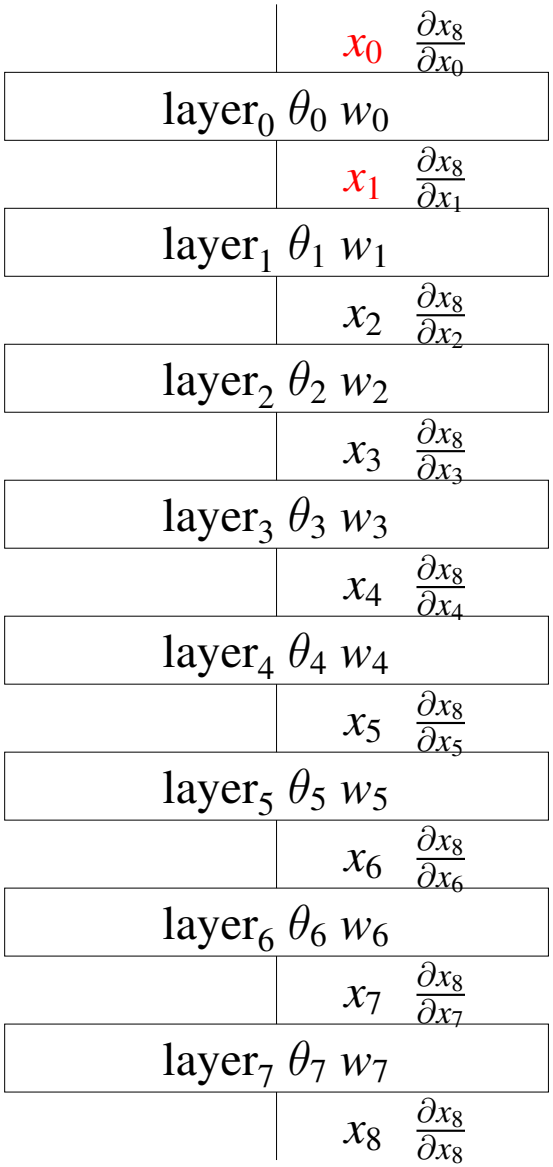
# Backpropagation in a Neural Network with Checkpointing



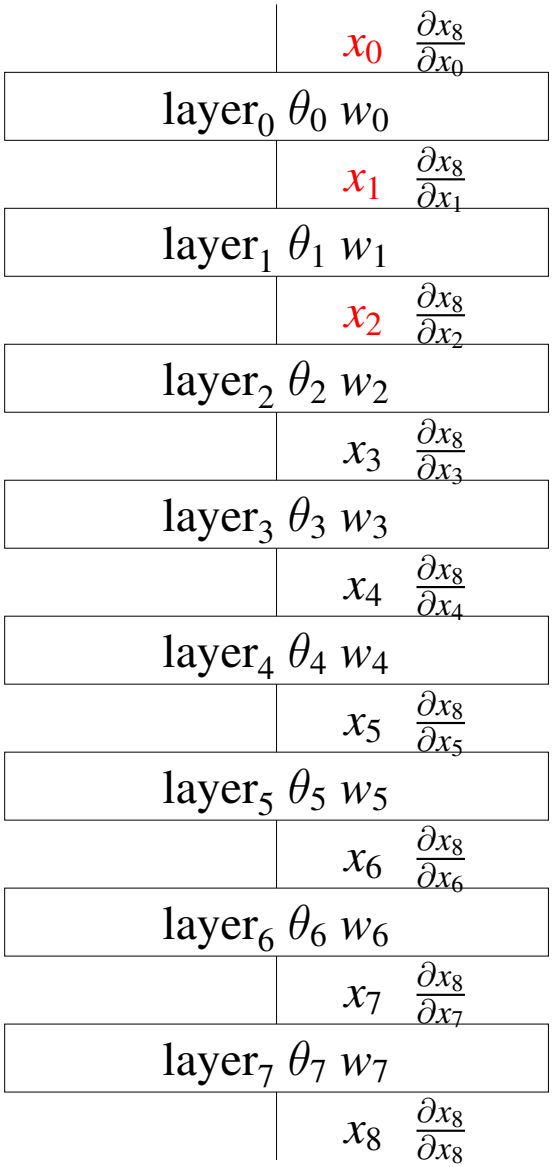
# Backpropagation in a Neural Network with Checkpointing



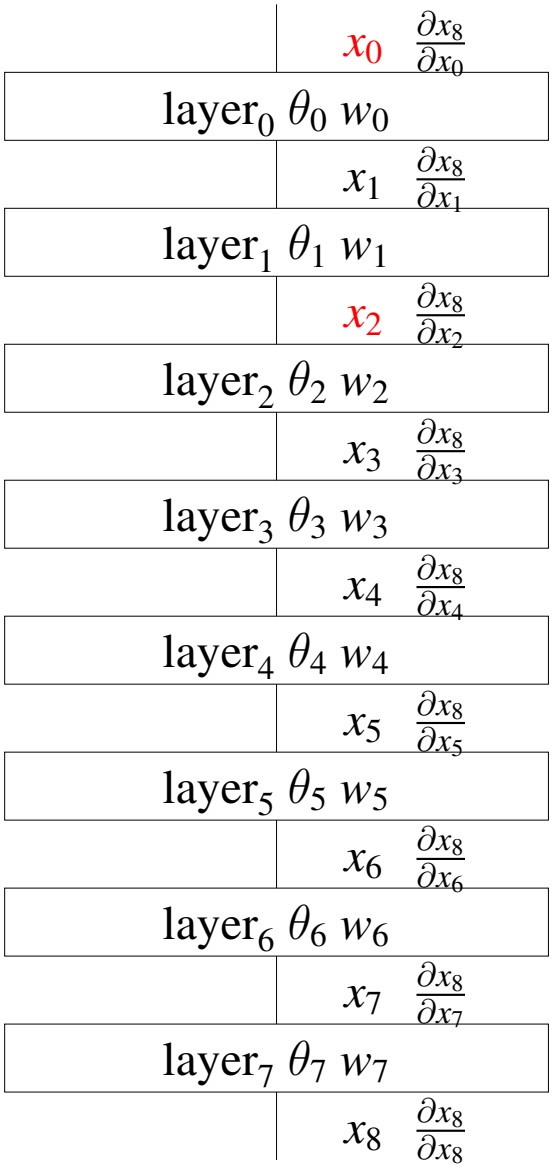
# Backpropagation in a Neural Network with Checkpointing



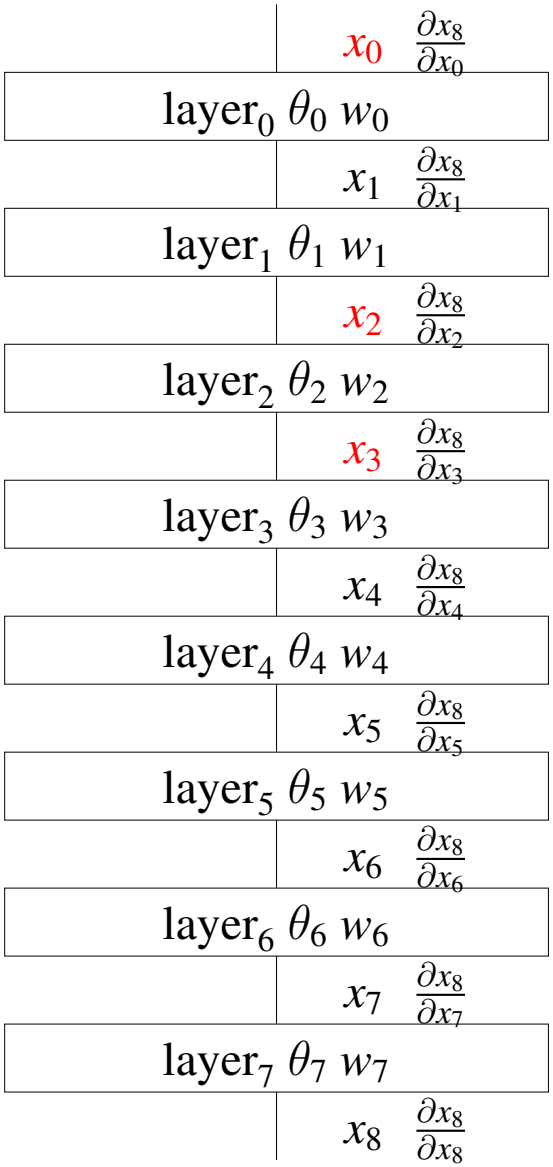
# Backpropagation in a Neural Network with Checkpointing



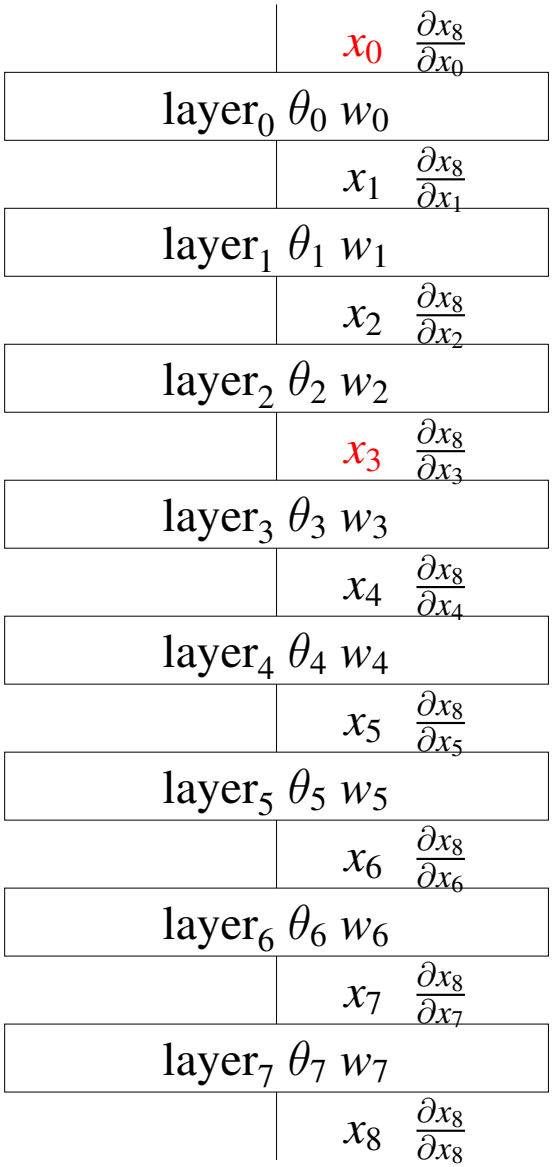
# Backpropagation in a Neural Network with Checkpointing



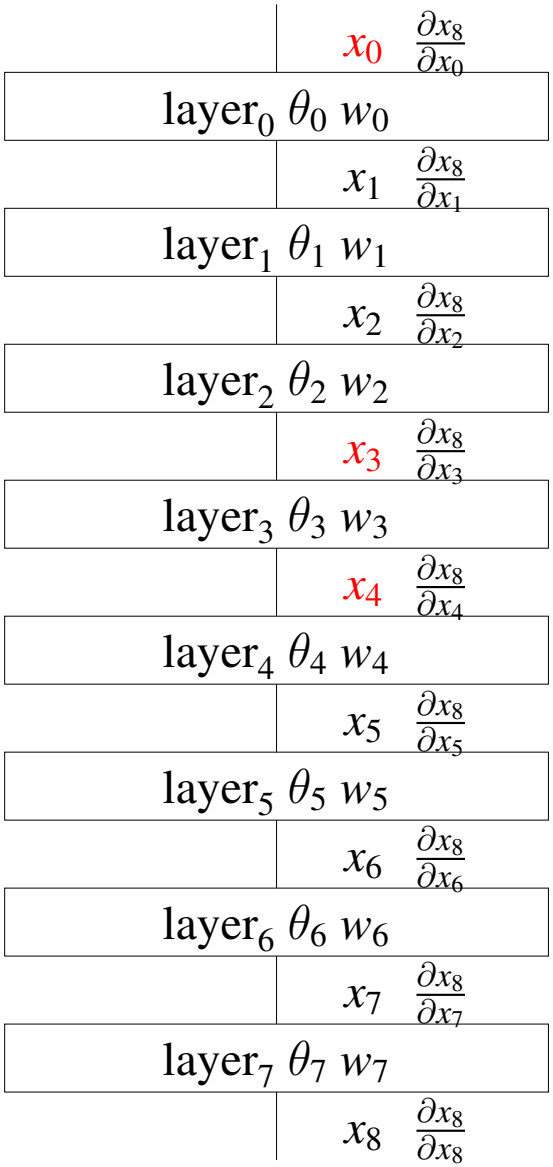
# Backpropagation in a Neural Network with Checkpointing



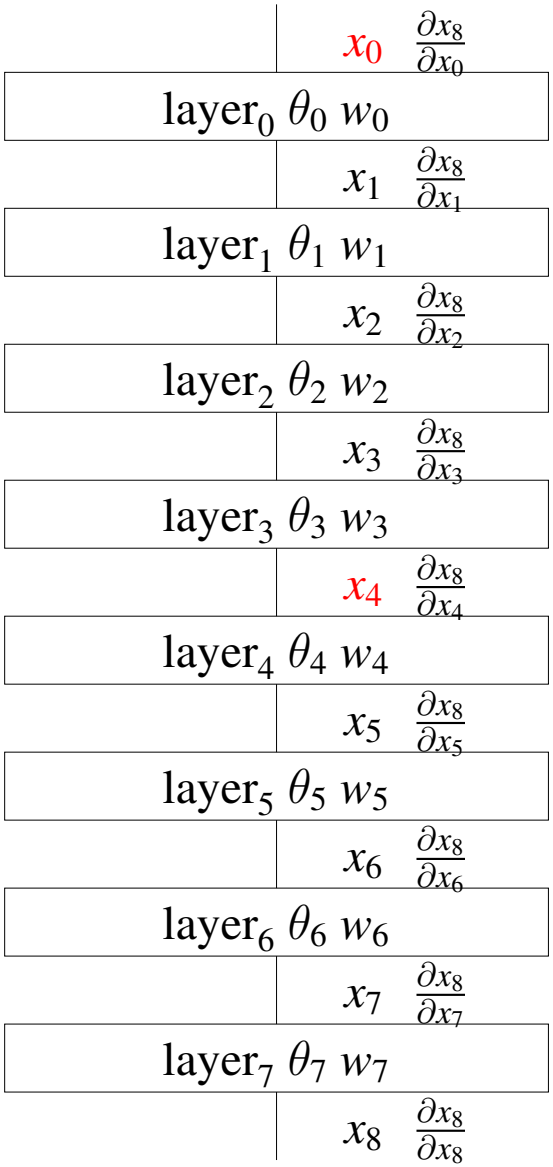
# Backpropagation in a Neural Network with Checkpointing



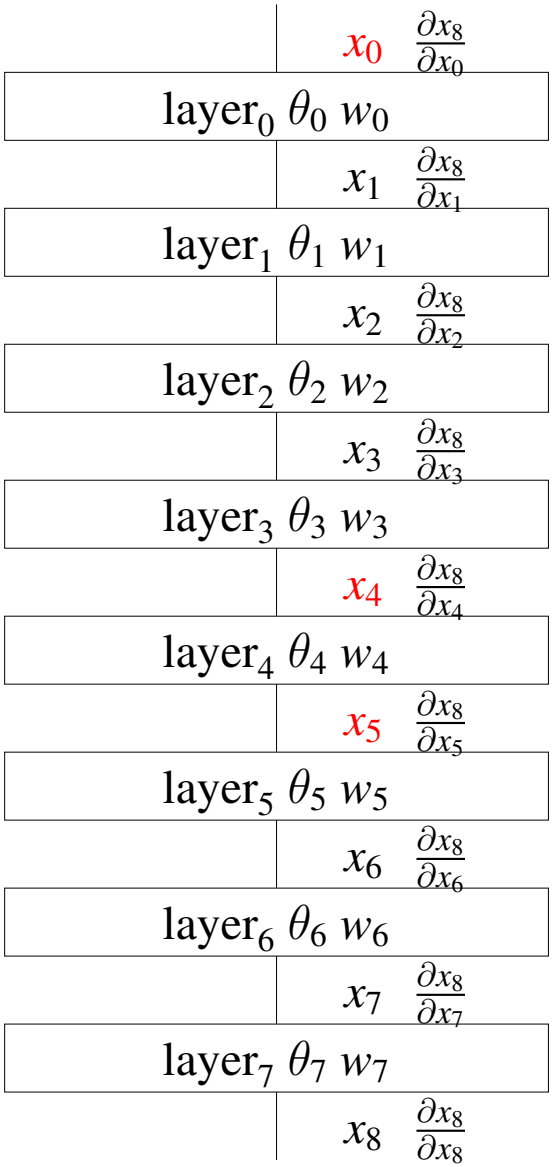
# Backpropagation in a Neural Network with Checkpointing



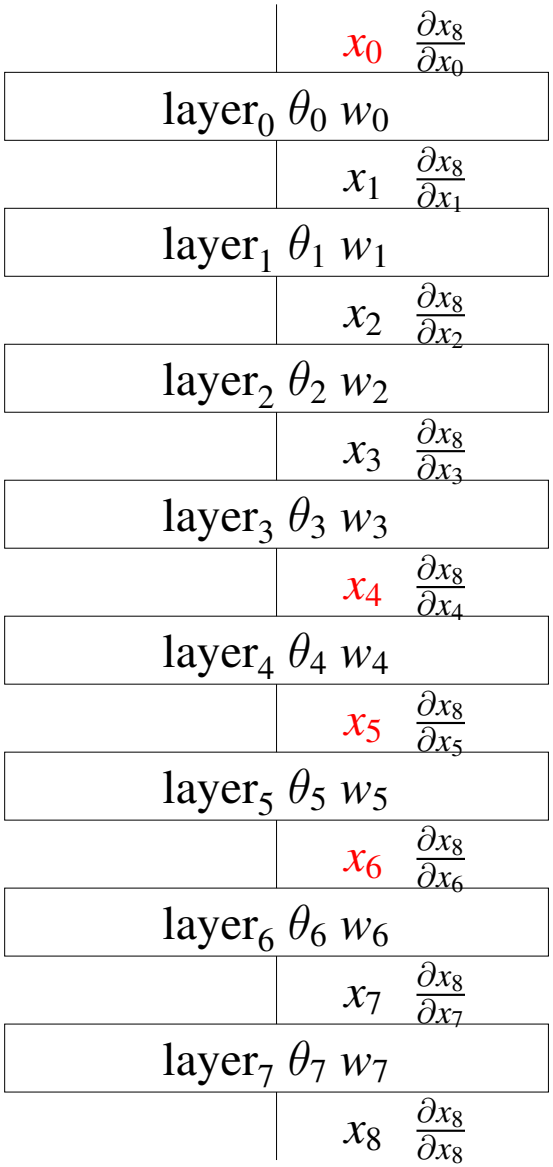
# Backpropagation in a Neural Network with Checkpointing



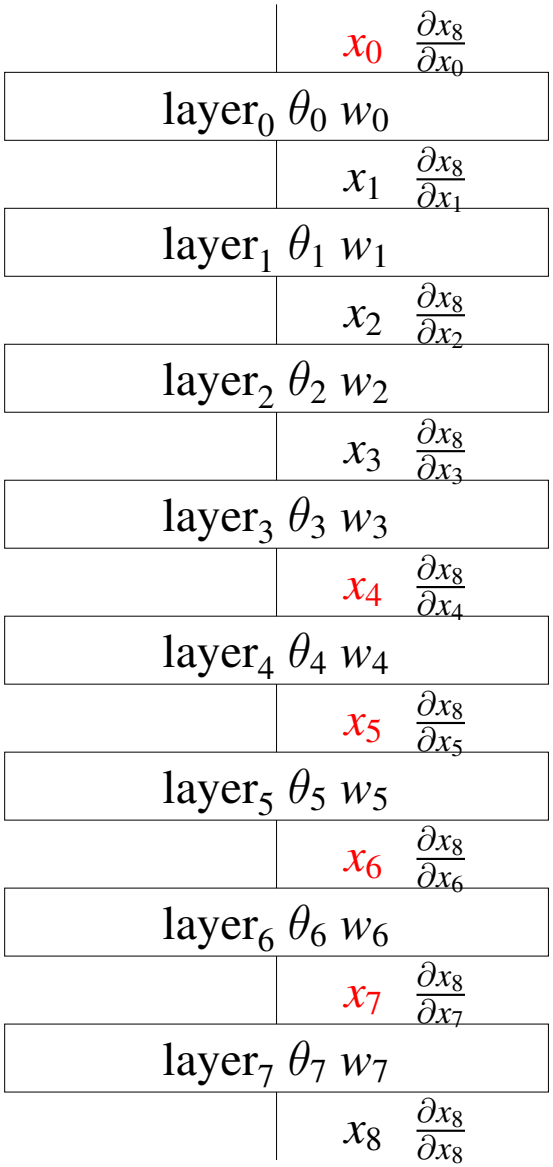
# Backpropagation in a Neural Network with Checkpointing



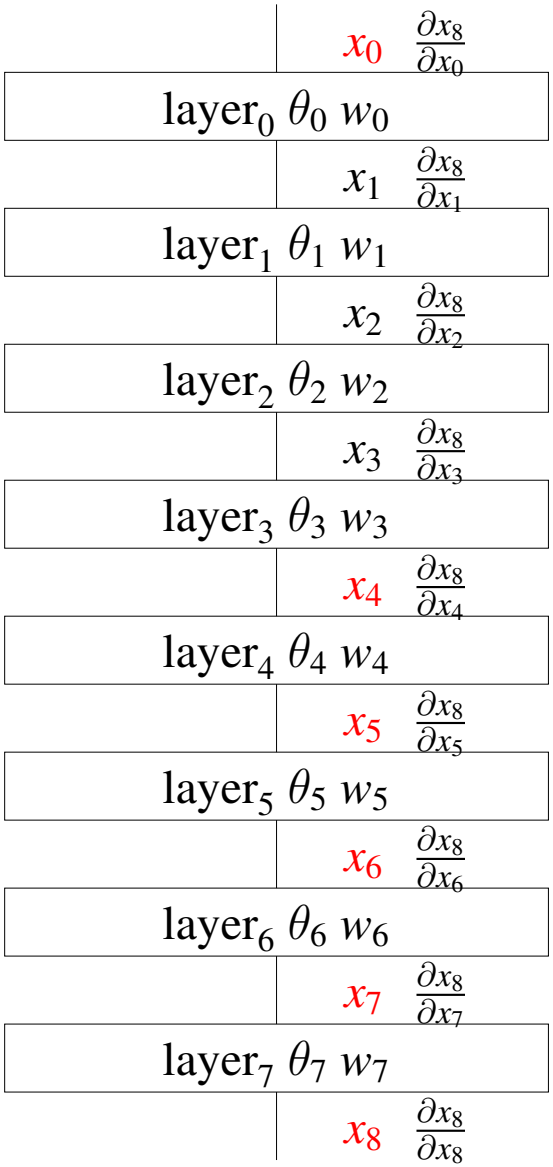
# Backpropagation in a Neural Network with Checkpointing



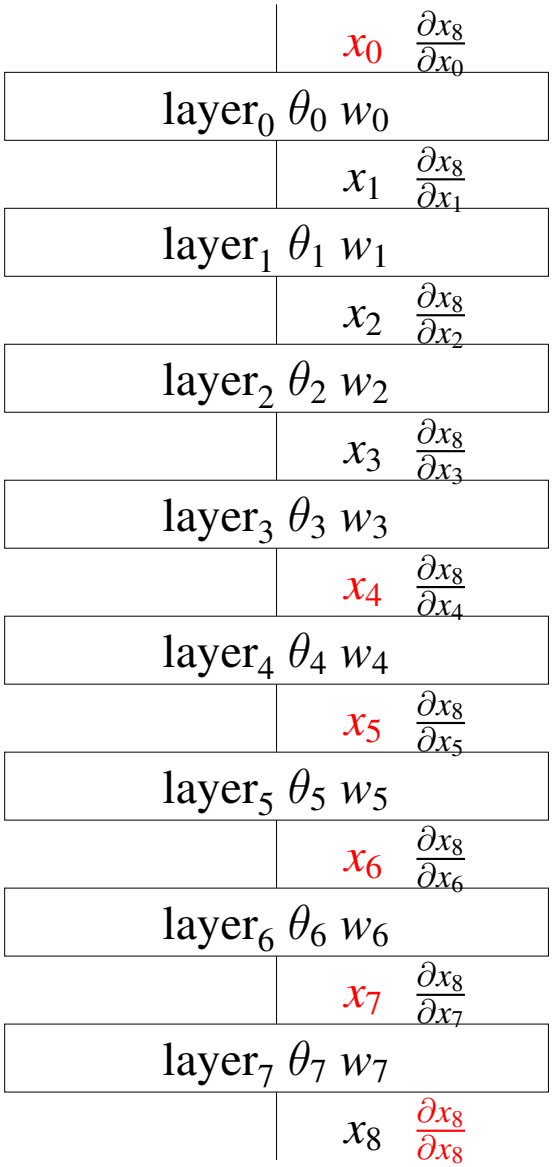
# Backpropagation in a Neural Network with Checkpointing



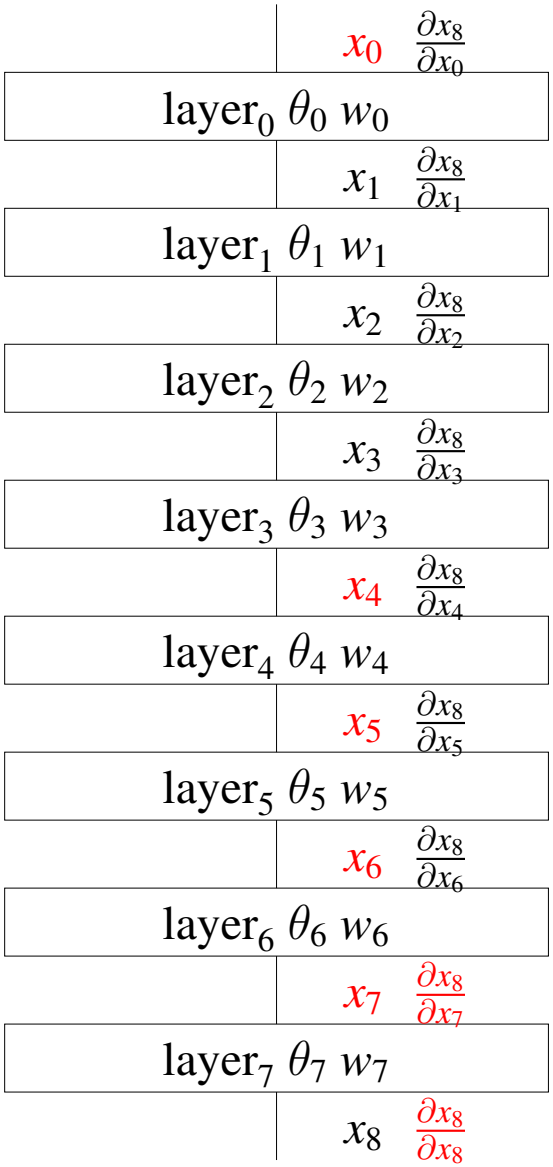
# Backpropagation in a Neural Network with Checkpointing



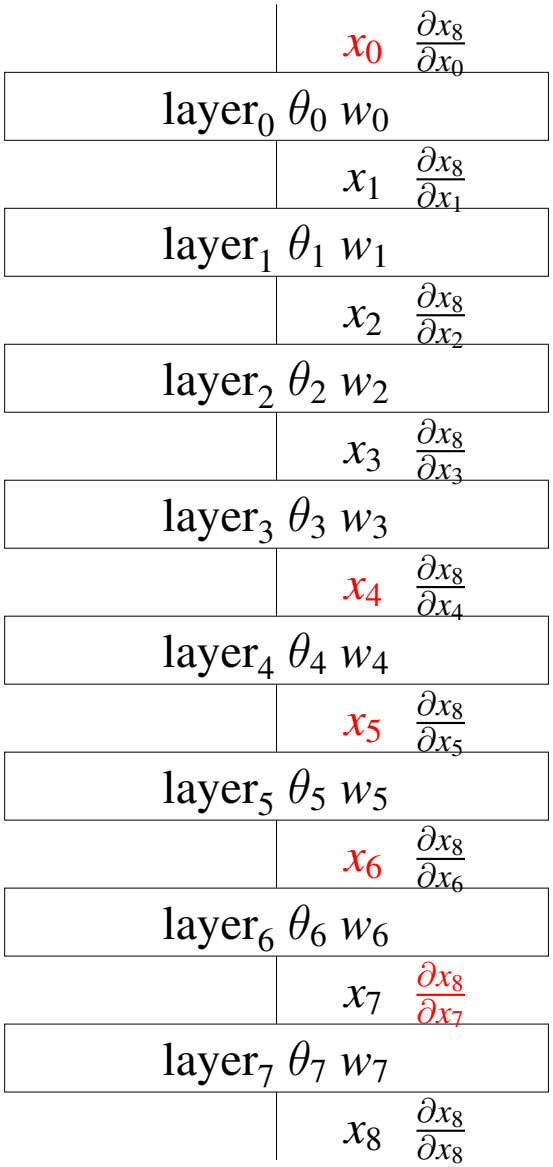
# Backpropagation in a Neural Network with Checkpointing



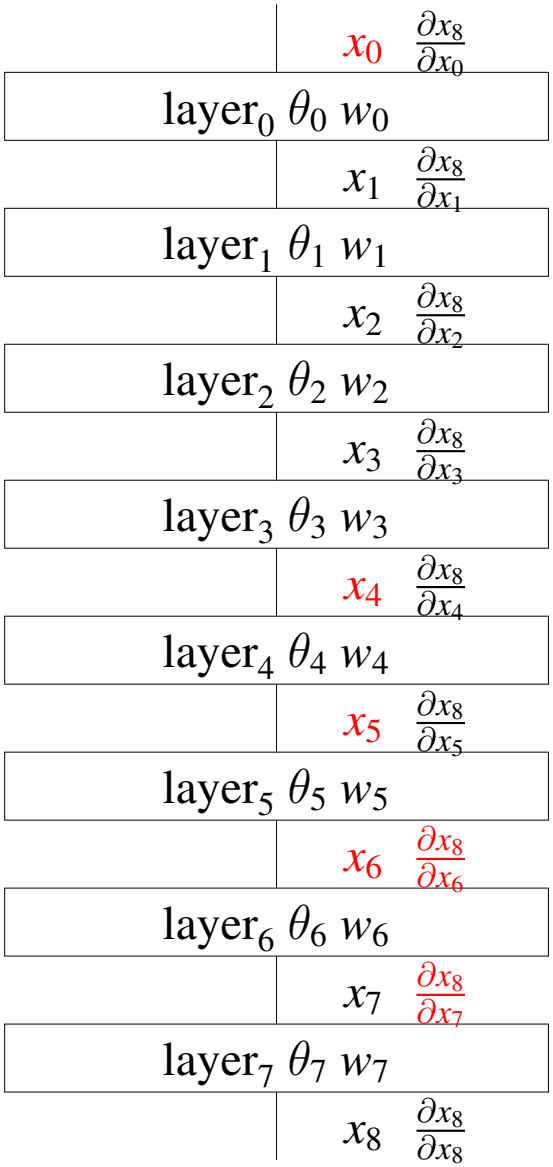
# Backpropagation in a Neural Network with Checkpointing



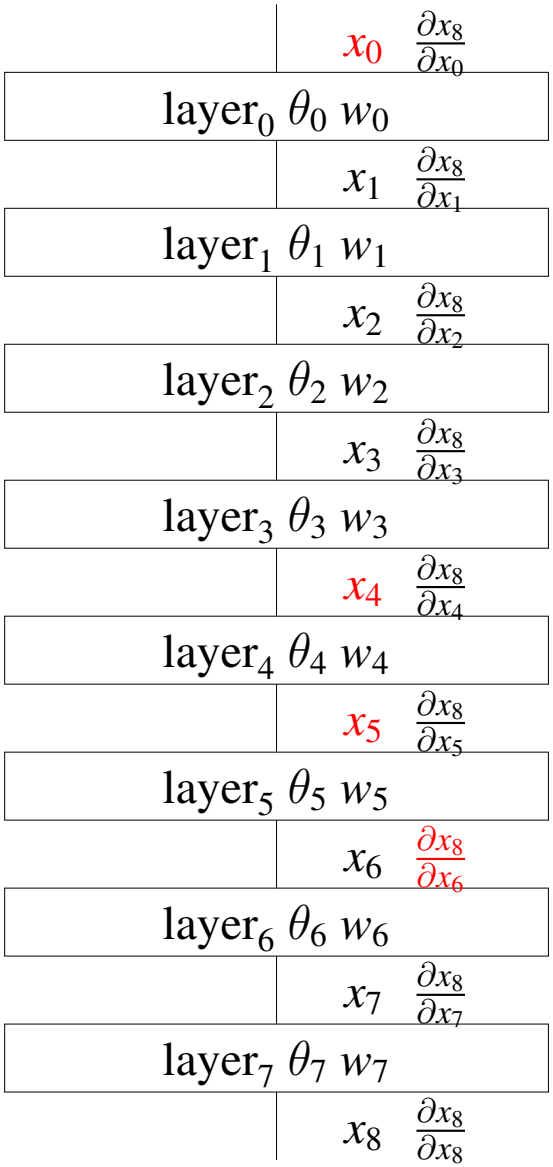
# Backpropagation in a Neural Network with Checkpointing



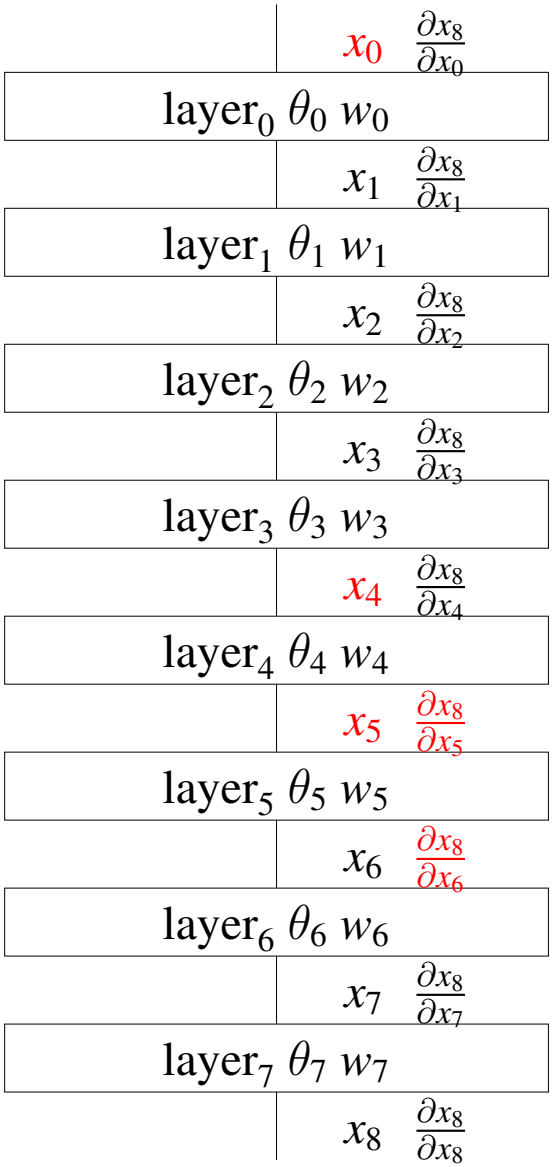
# Backpropagation in a Neural Network with Checkpointing



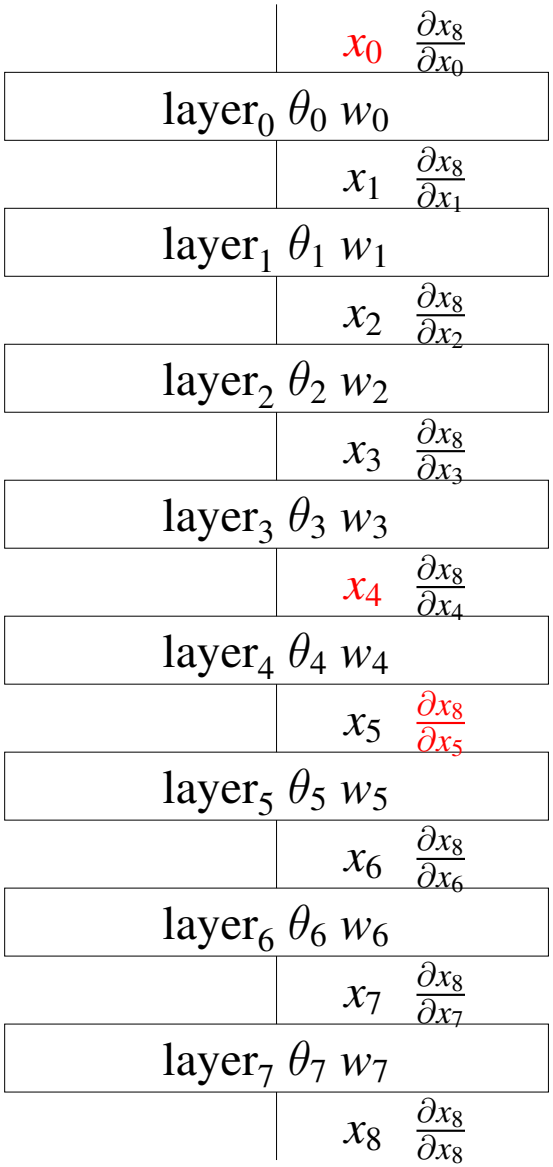
# Backpropagation in a Neural Network with Checkpointing



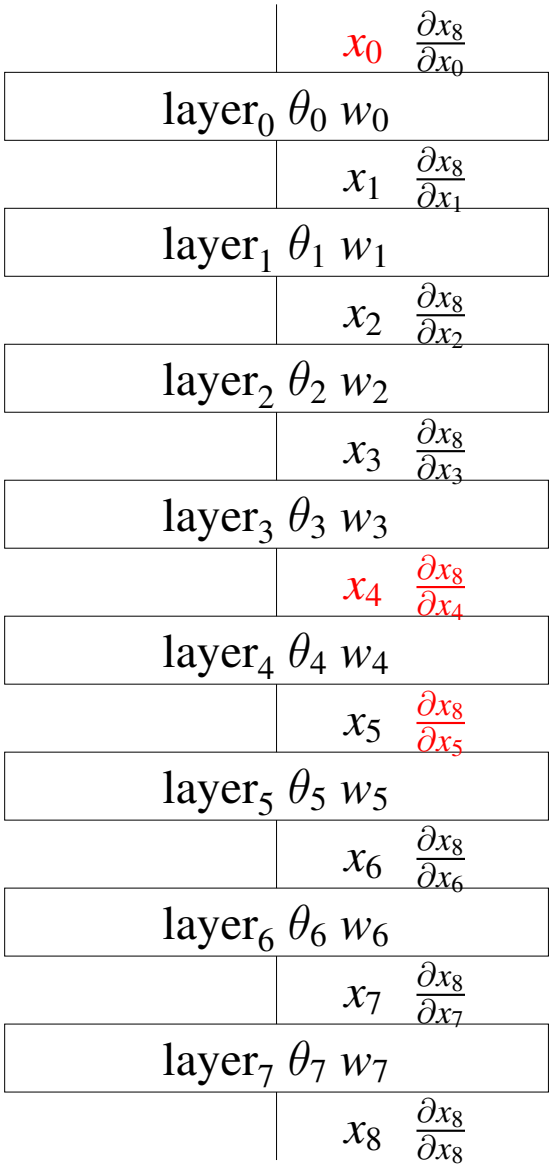
# Backpropagation in a Neural Network with Checkpointing



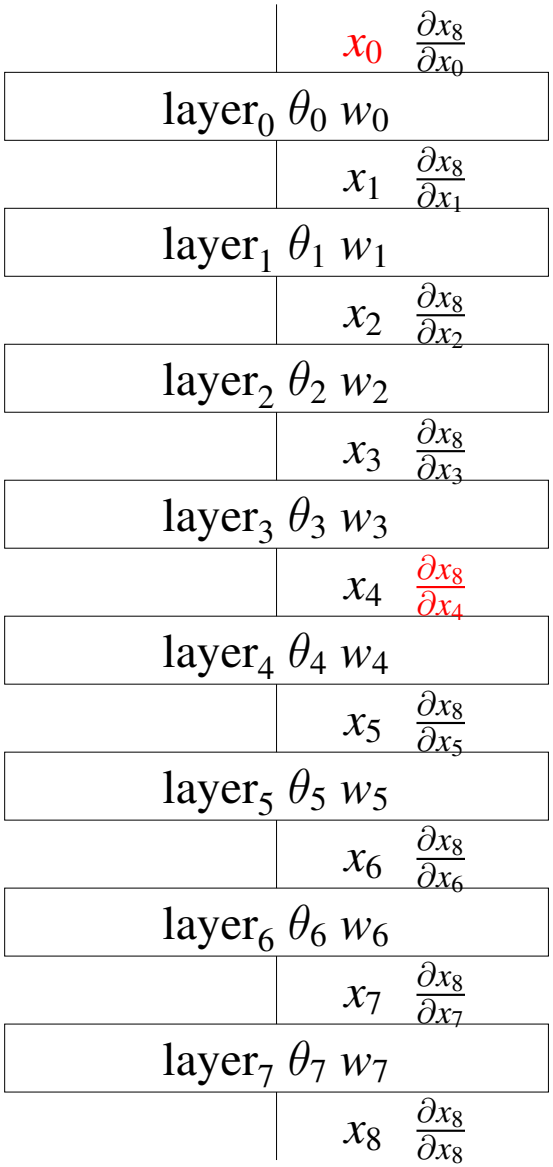
# Backpropagation in a Neural Network with Checkpointing



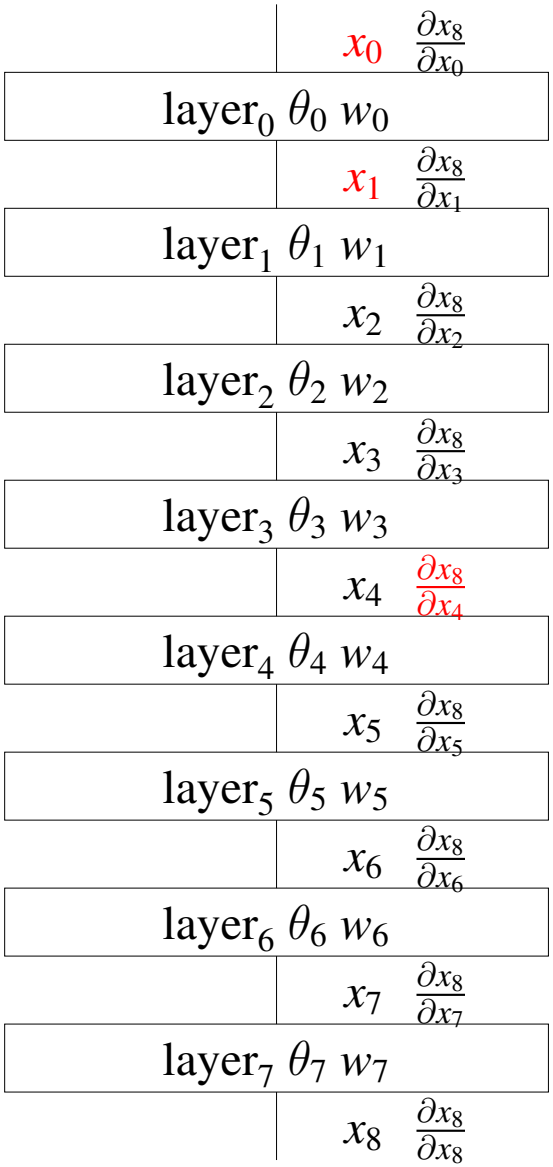
# Backpropagation in a Neural Network with Checkpointing



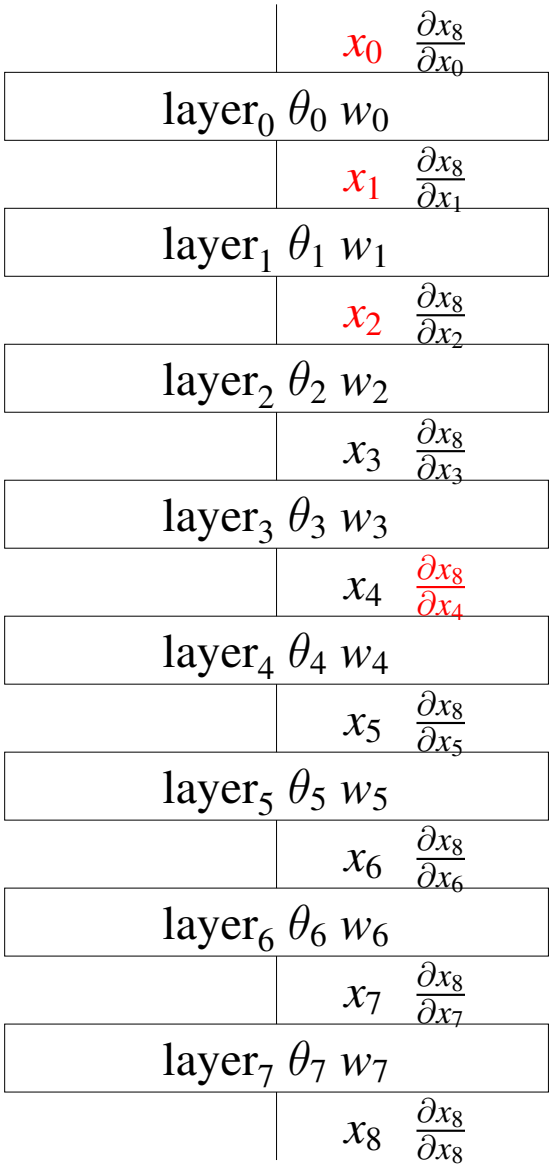
# Backpropagation in a Neural Network with Checkpointing



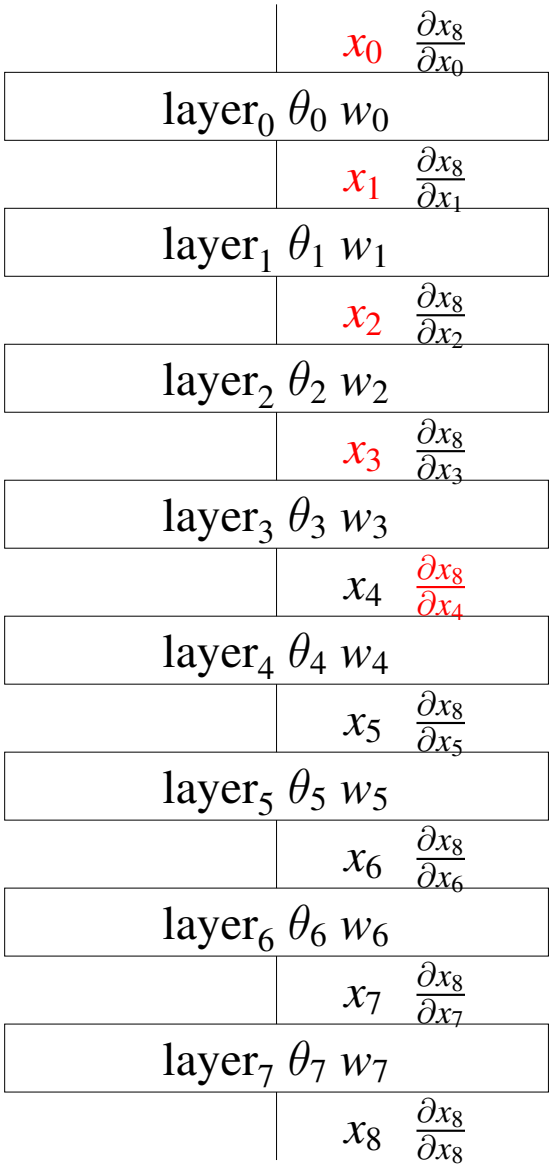
# Backpropagation in a Neural Network with Checkpointing



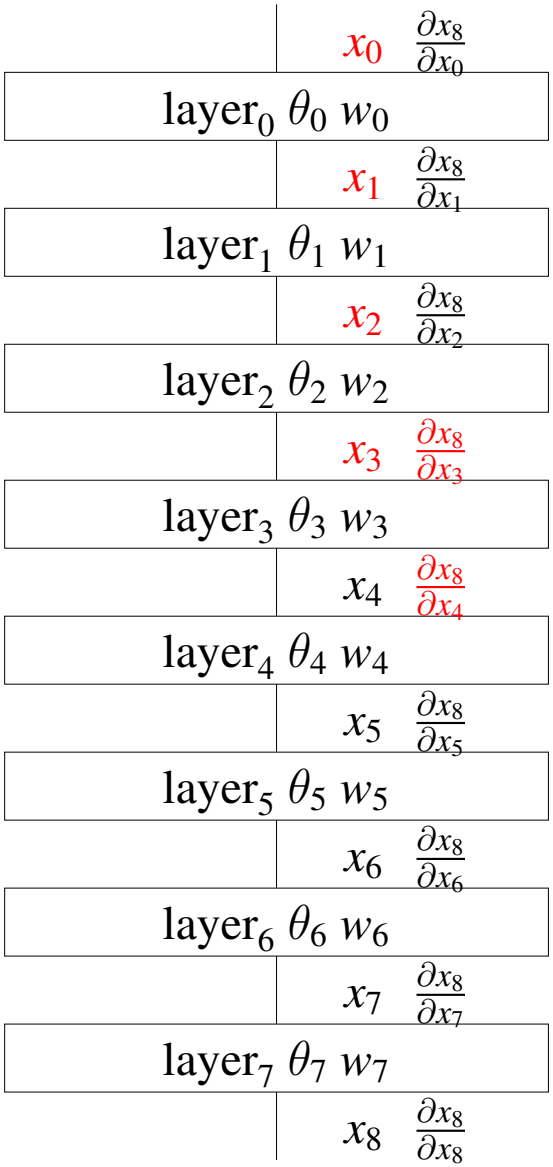
# Backpropagation in a Neural Network with Checkpointing



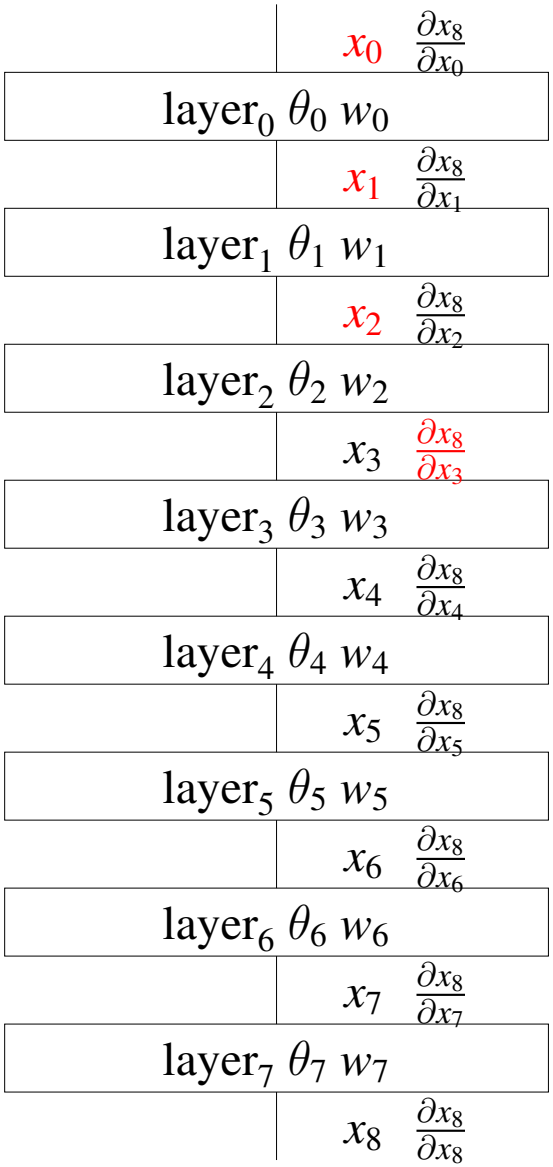
# Backpropagation in a Neural Network with Checkpointing



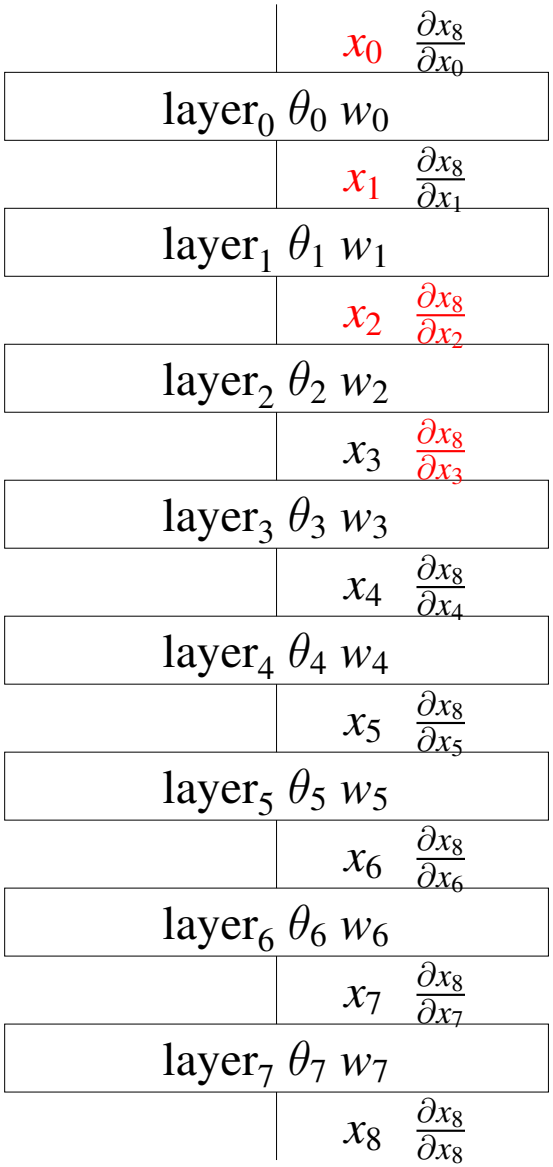
# Backpropagation in a Neural Network with Checkpointing



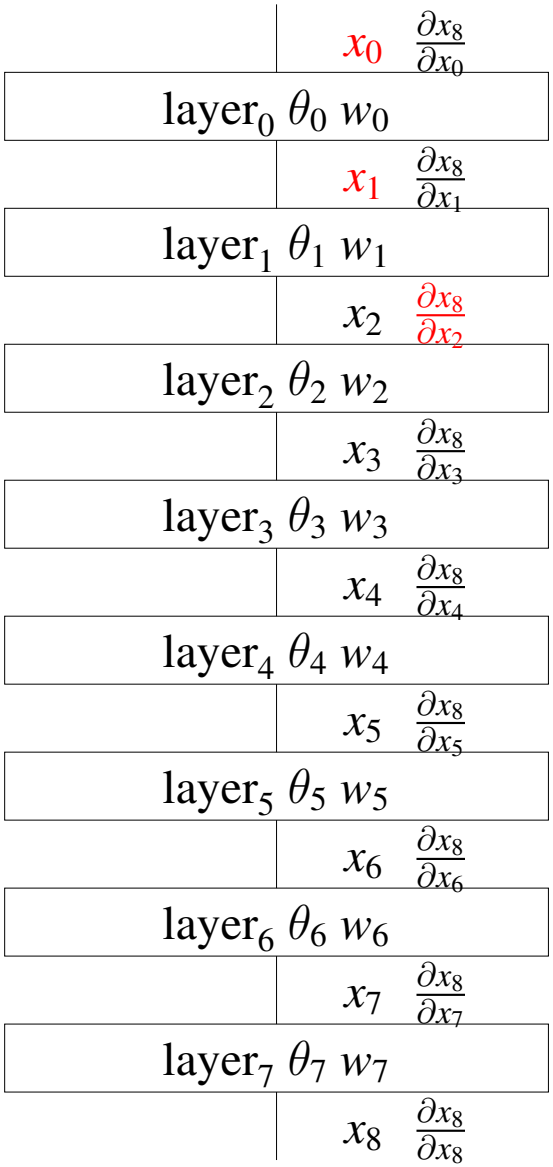
# Backpropagation in a Neural Network with Checkpointing



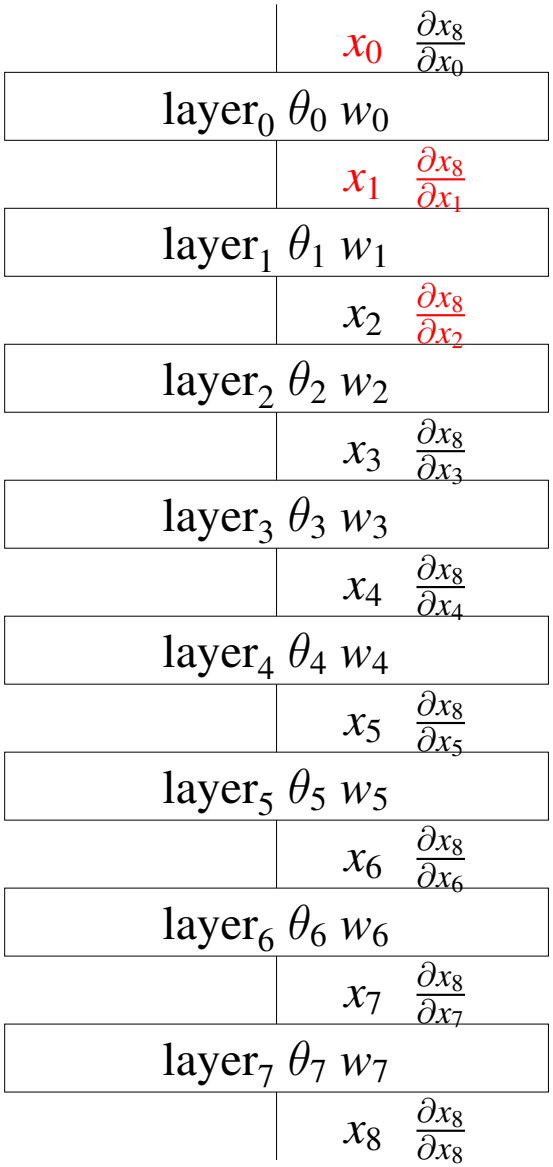
# Backpropagation in a Neural Network with Checkpointing



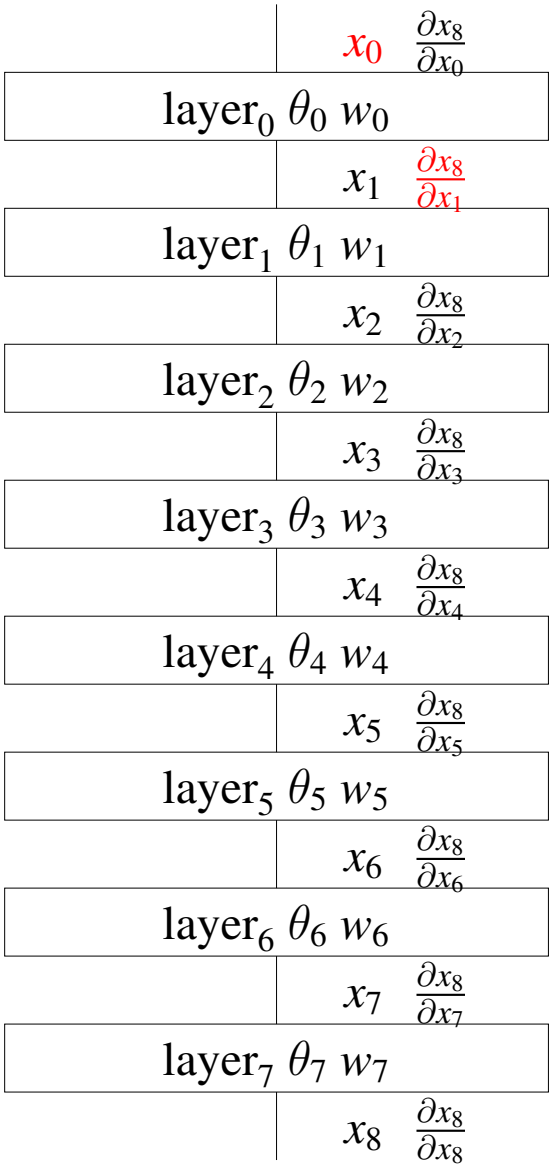
# Backpropagation in a Neural Network with Checkpointing



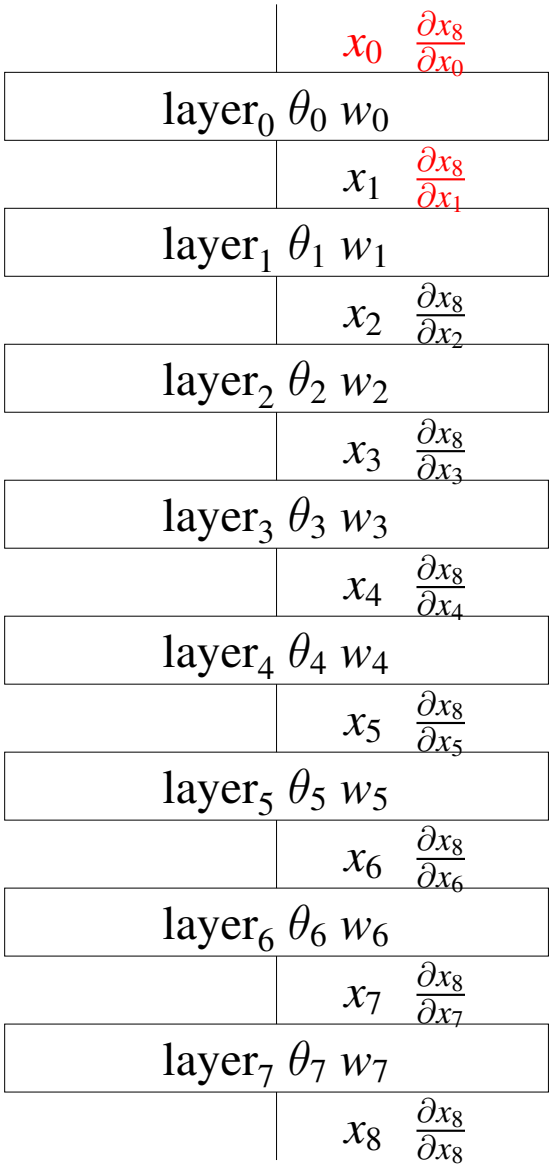
# Backpropagation in a Neural Network with Checkpointing



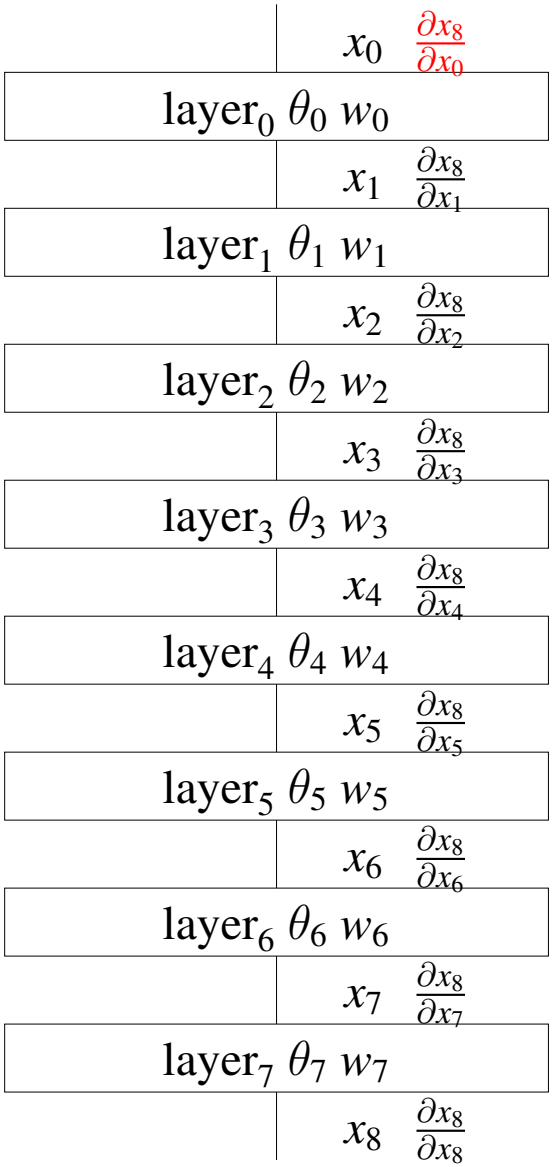
# Backpropagation in a Neural Network with Checkpointing



# Backpropagation in a Neural Network with Checkpointing



# Backpropagation in a Neural Network with Checkpointing



# Checkpointing

# Checkpointing

- Trades off extra running time for reduction in space.

# Checkpointing

- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.

# Checkpointing

- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.
  Once without saving intermediate variables.

# Checkpointing

- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.
  Once without saving intermediate variables.
  Once with saving intermediate variables.

# Checkpointing

- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.
  Once without saving intermediate variables.
  Once with saving intermediate variables.
- Backpropagation done in stages.

# Checkpointing

- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.
  Once without saving intermediate variables.
  Once with saving intermediate variables.
- Backpropagation done in stages.
  Interleaved with (re)running forward pass.

# Checkpointing

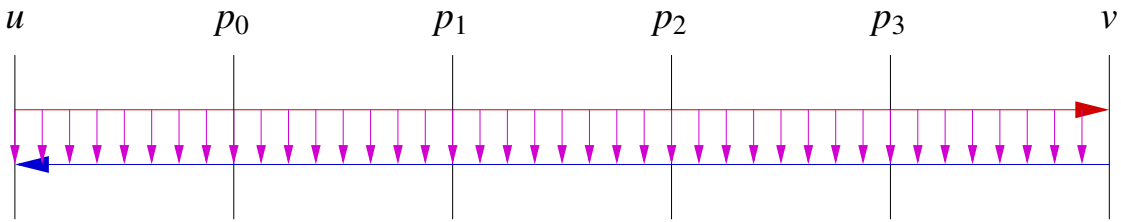
- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.
  Once without saving intermediate variables.
  Once with saving intermediate variables.
- Backpropagation done in stages.
  Interleaved with (re)running forward pass.
  Only need saved intermediate variables from forward pass for current stage.

# Checkpointing

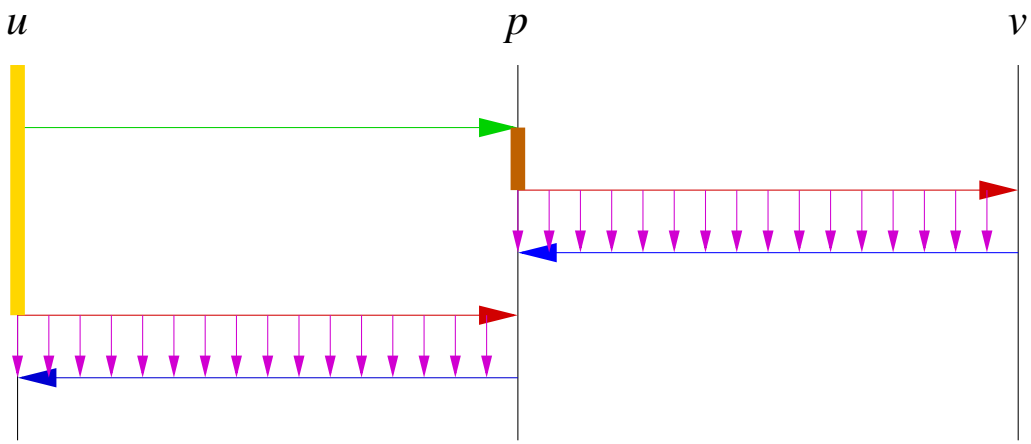
- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.
  Once without saving intermediate variables.
  Once with saving intermediate variables.
- Backpropagation done in stages.
  Interleaved with (re)running forward pass.
  Only need saved intermediate variables from forward pass for current stage.
- Can perform divide-and-conquer.

# Divide-and-Conquer Checkpointing

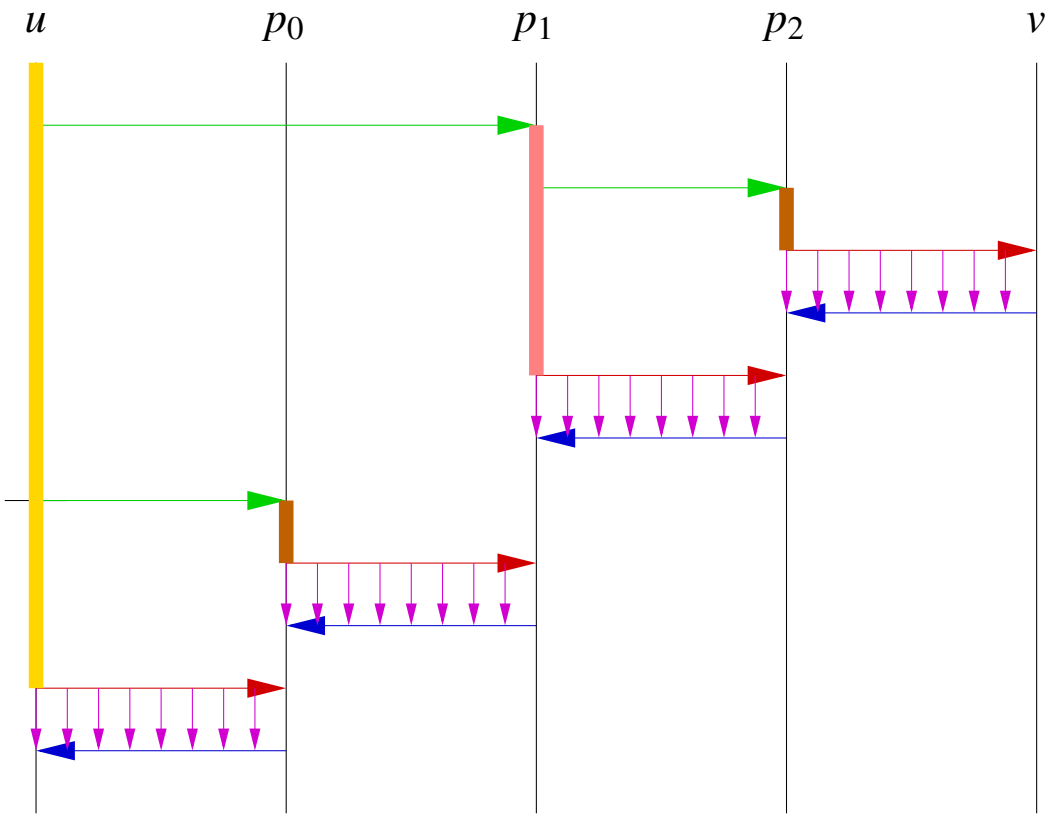
# Divide-and-Conquer Checkpointing



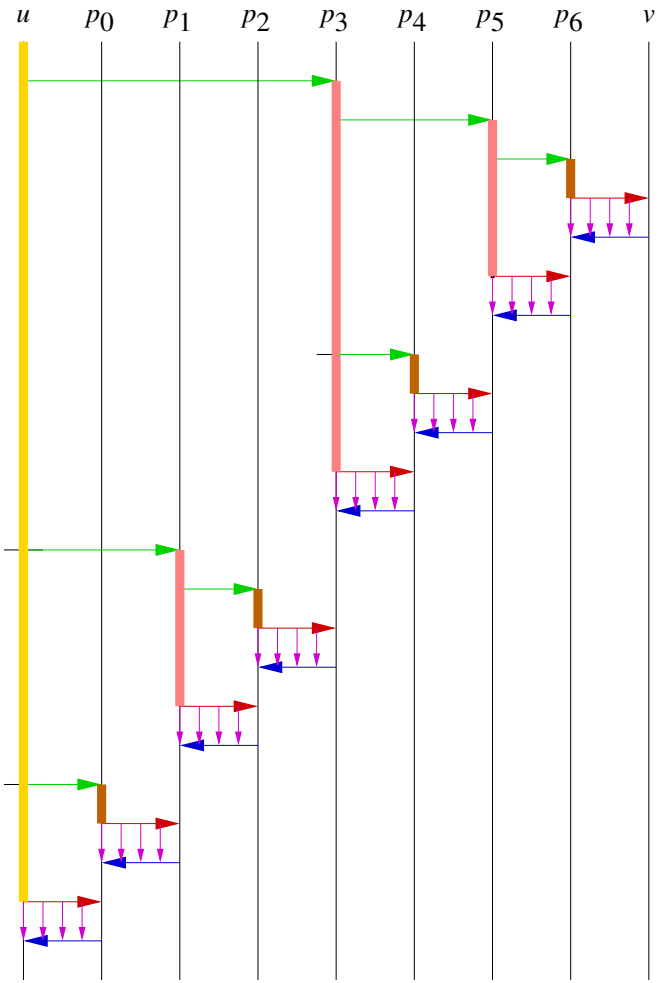
# Divide-and-Conquer Checkpointing



# Divide-and-Conquer Checkpointing



# Divide-and-Conquer Checkpointing



# Complexity of Divide-and-Conquer Checkpointing

# Complexity of Divide-and-Conquer Checkpointing

- If running time of primal is $O(t)$

# Complexity of Divide-and-Conquer Checkpointing

- If running time of primal is $O(t)$
  and primal has maximal live storage $O(w)$

# Complexity of Divide-and-Conquer Checkpointing

- If running time of primal is $O(t)$
  and primal has maximal live storage $O(w)$
- then reverse mode takes $O(w \log t)$ space

# Complexity of Divide-and-Conquer Checkpointing

- If running time of primal is $O(t)$
  and primal has maximal live storage $O(w)$
- then reverse mode takes $O(w \log t)$ space
  and $O(t \log t)$ time.

# A (Brief) History of Divide-and-Conquer Checkpointing

# A (Brief) History of Divide-and-Conquer Checkpointing

T. Chen, B. Xu, Z. Zhang, and C. Guestrin, *Training deep nets with sublinear memory cost*, arXiv 1604.06174, 2016.

A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, *Memory-Efficient Backpropagation Through Time*, NIPS, 2016.

# A (Brief) History of Divide-and-Conquer Checkpointing

T. Chen, B. Xu, Z. Zhang, and C. Guestrin, *Training deep nets with sublinear memory cost*, arXiv 1604.06174, 2016.

A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, *Memory-Efficient Backpropagation Through Time*, NIPS, 2016.

A. Griewank, *Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation*, Optimization Methods and Software, 1:35-54, 1992.

# Implemented for DO Loops

L. Hascoët and V. Pascual, *TAPENADE 2.1 User's Guide*, Rapport technique 300, INRIA, 2004.

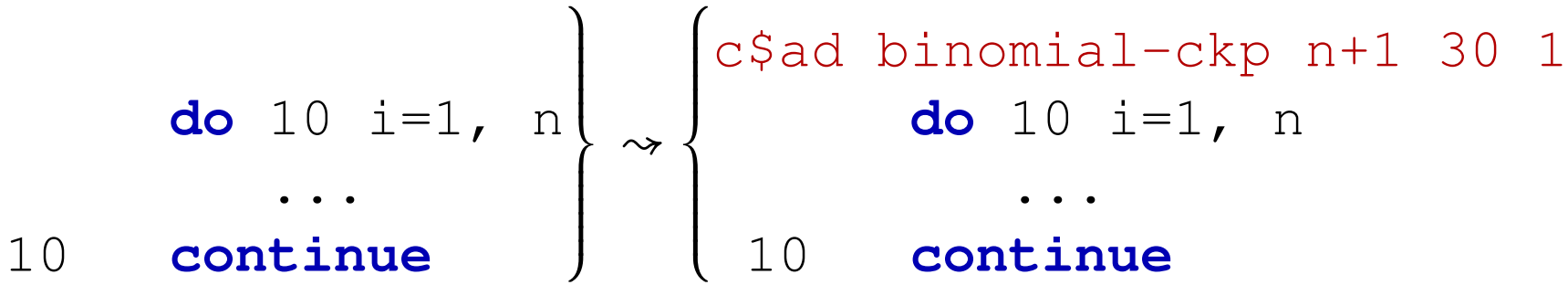
# Implemented for DO Loops

L. Hascoët and V. Pascual, *TAPENADE 2.1 User's Guide*, Rapport technique 300, INRIA, 2004.

```
      do 10 i=1, n
         ...
10    continue
```

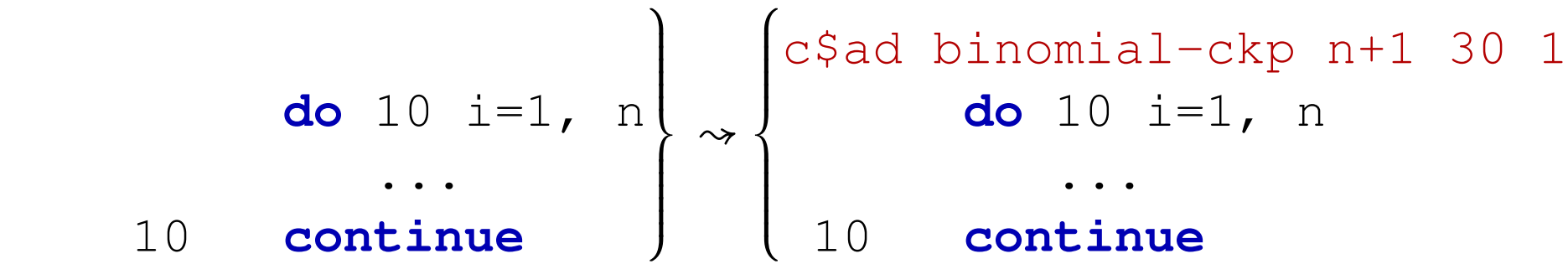
# Implemented for DO Loops

L. Hascoët and V. Pascual, *TAPENADE 2.1 User's Guide*, Rapport technique 300, INRIA, 2004.



# Implemented for DO Loops

L. Hascoët and V. Pascual, *TAPENADE 2.1 User's Guide*, Rapport technique 300, INRIA, 2004.




https://www-sop.inria.fr/tropics/tapenade/faq.html

*Assuming that the final number of iterations N is known, and assuming that each iteration has the same runtime cost,*

# Desiderata

# Desiderata

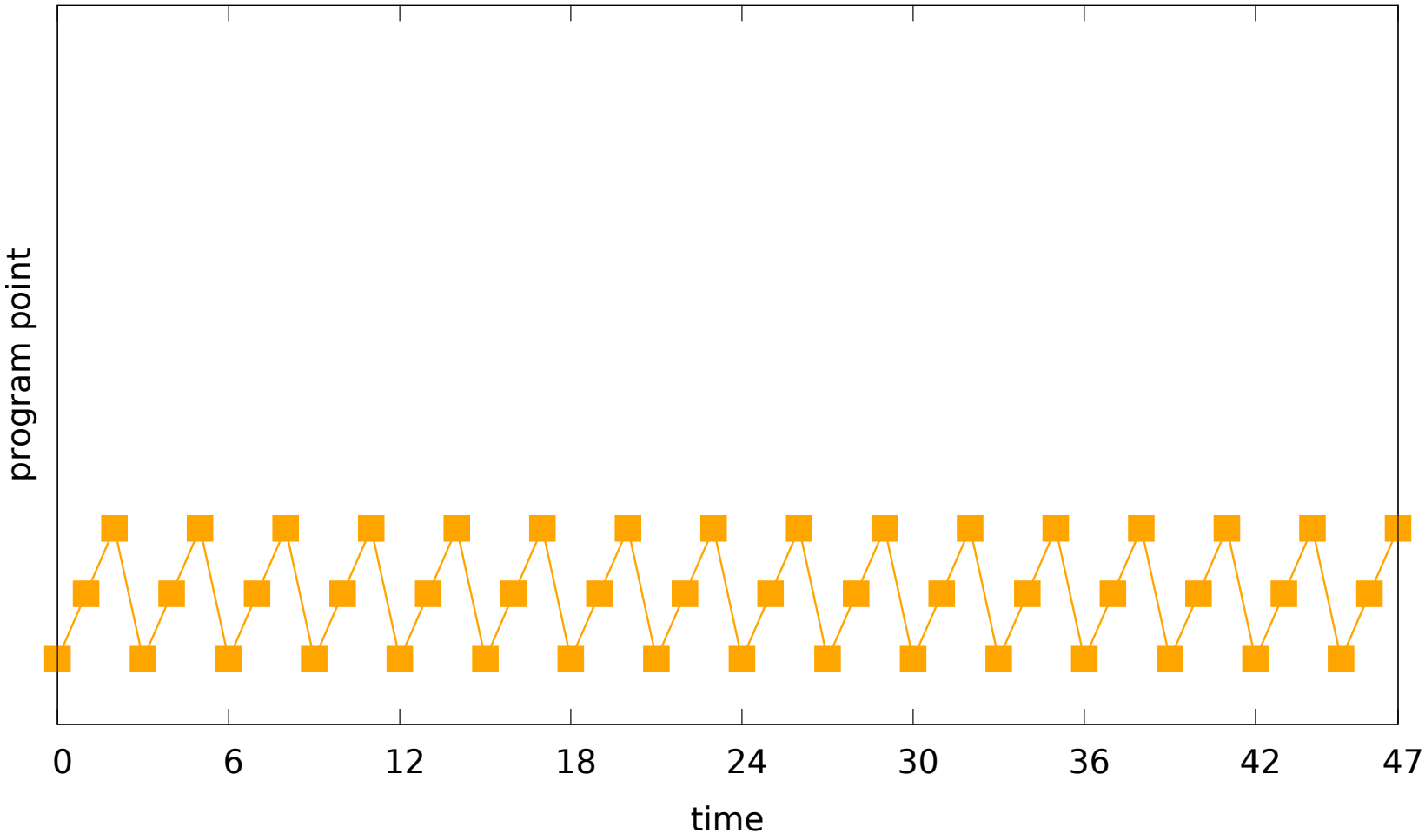
A (deep) neural network has no loops (except inside primitives).

# Desiderata

A (deep) neural network has no loops (except inside primitives).

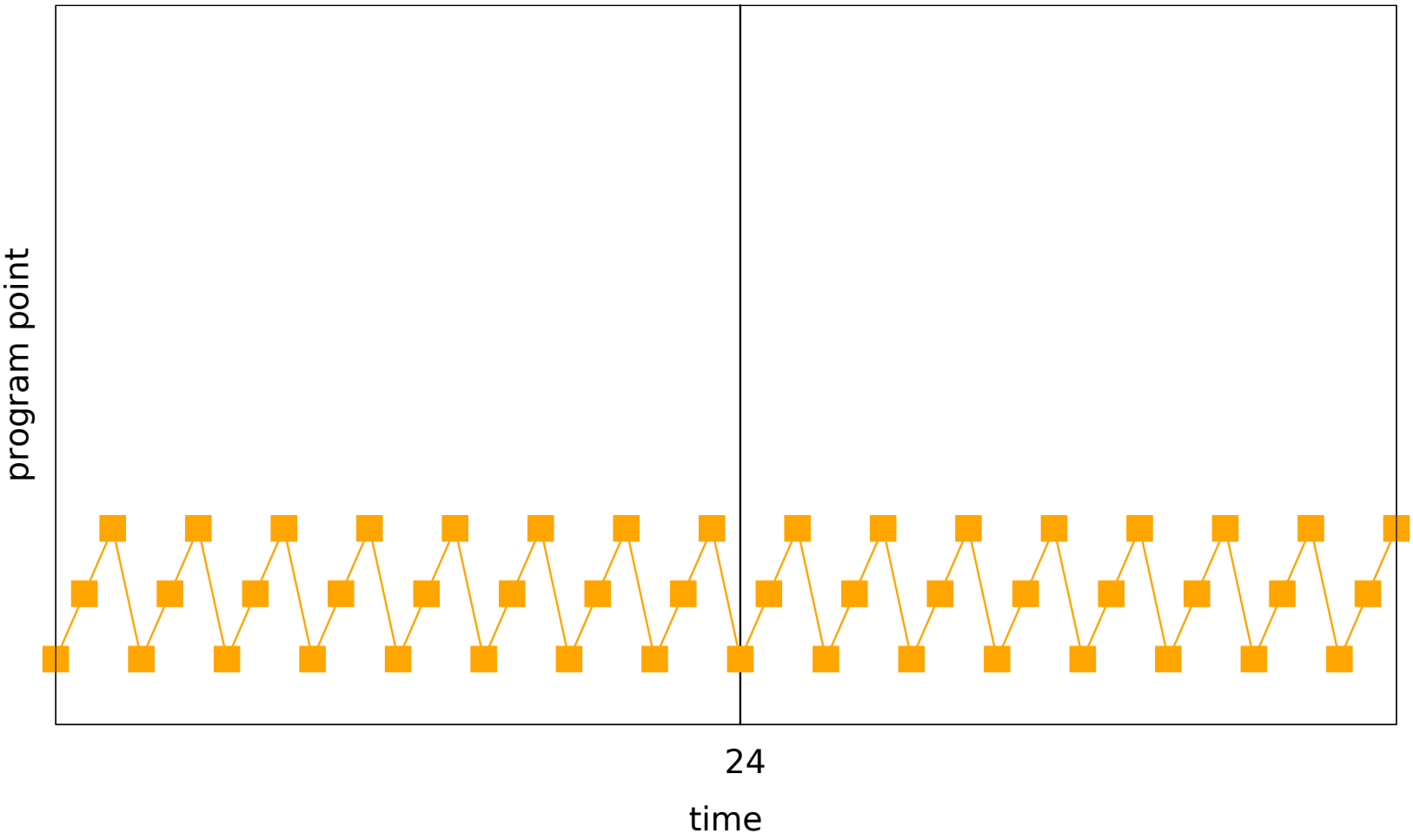
Want to implement for arbitrary code (not just a single DO loop).

# Execution Trace of Loop



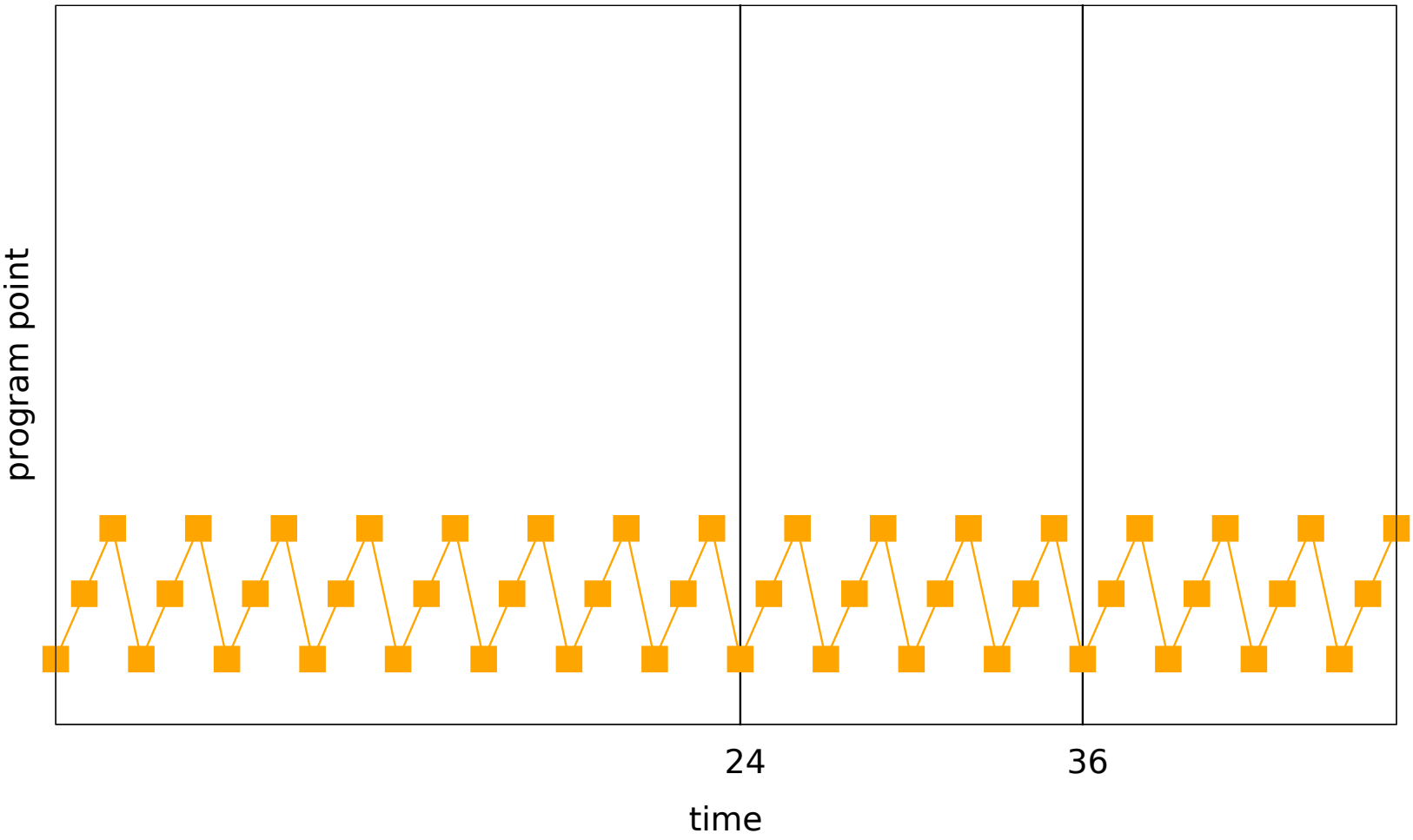
# Execution Trace of Loop

## Easy to make regular and uniform checkpoints



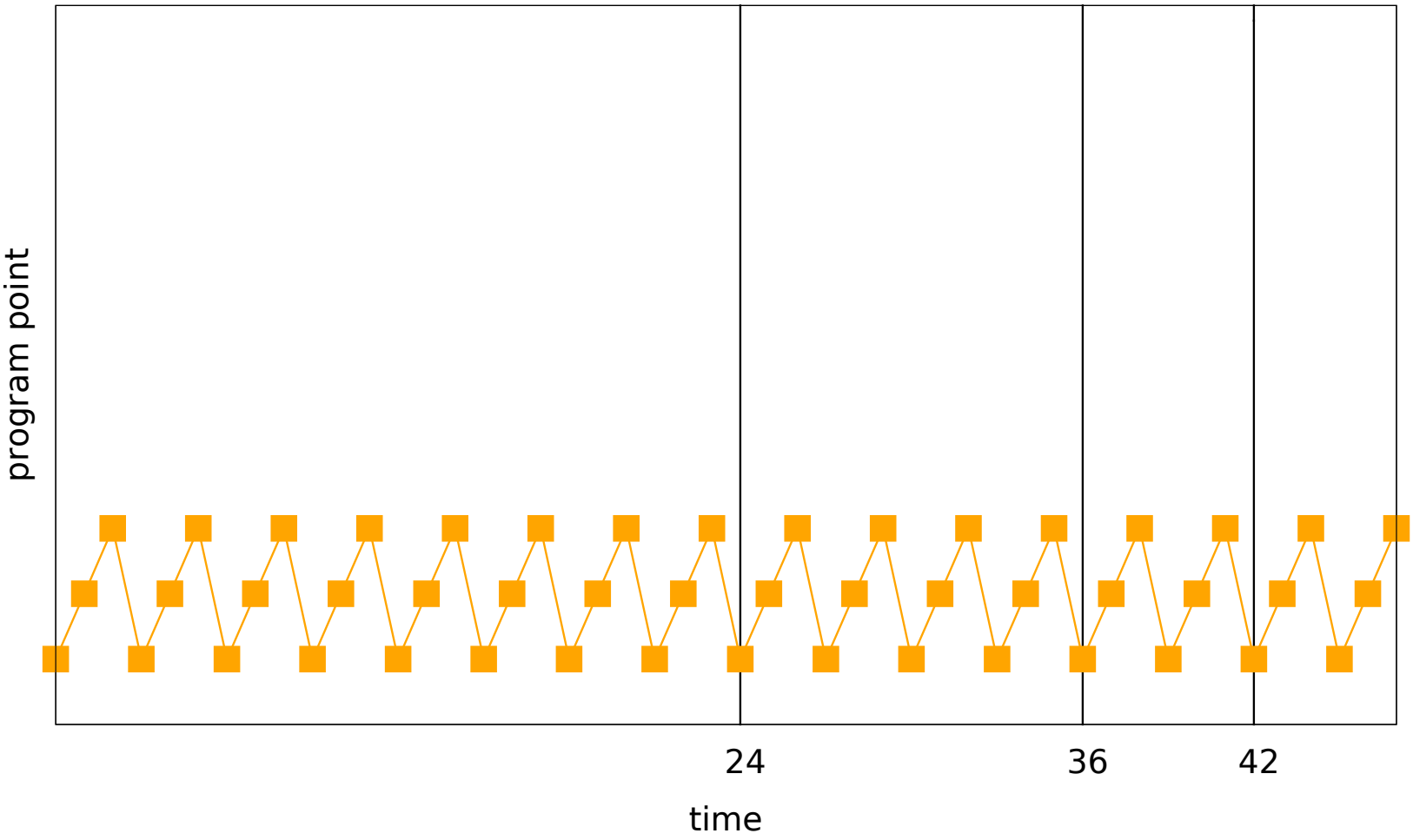
# Execution Trace of Loop

## Easy to make regular and uniform checkpoints



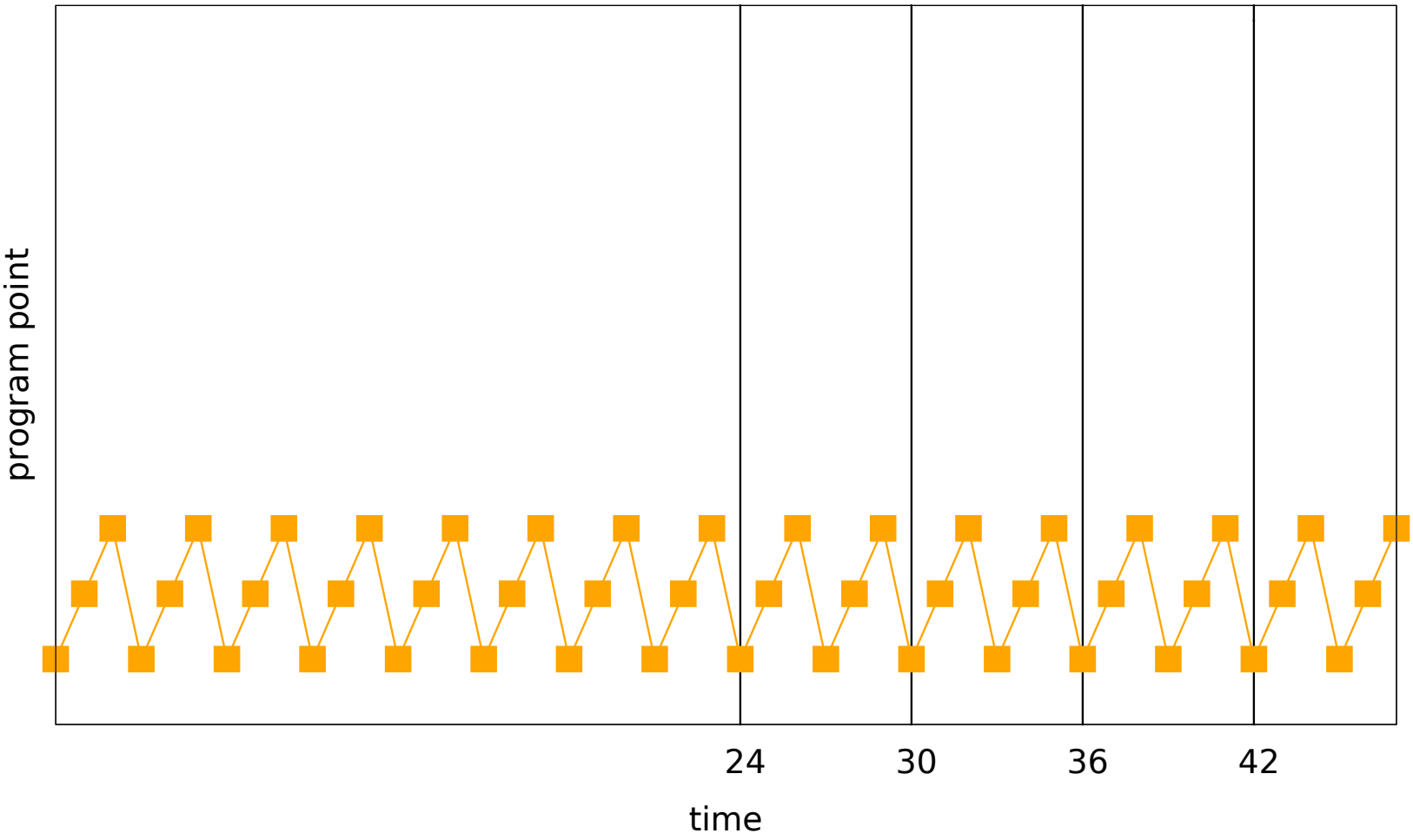
# Execution Trace of Loop

## Easy to make regular and uniform checkpoints



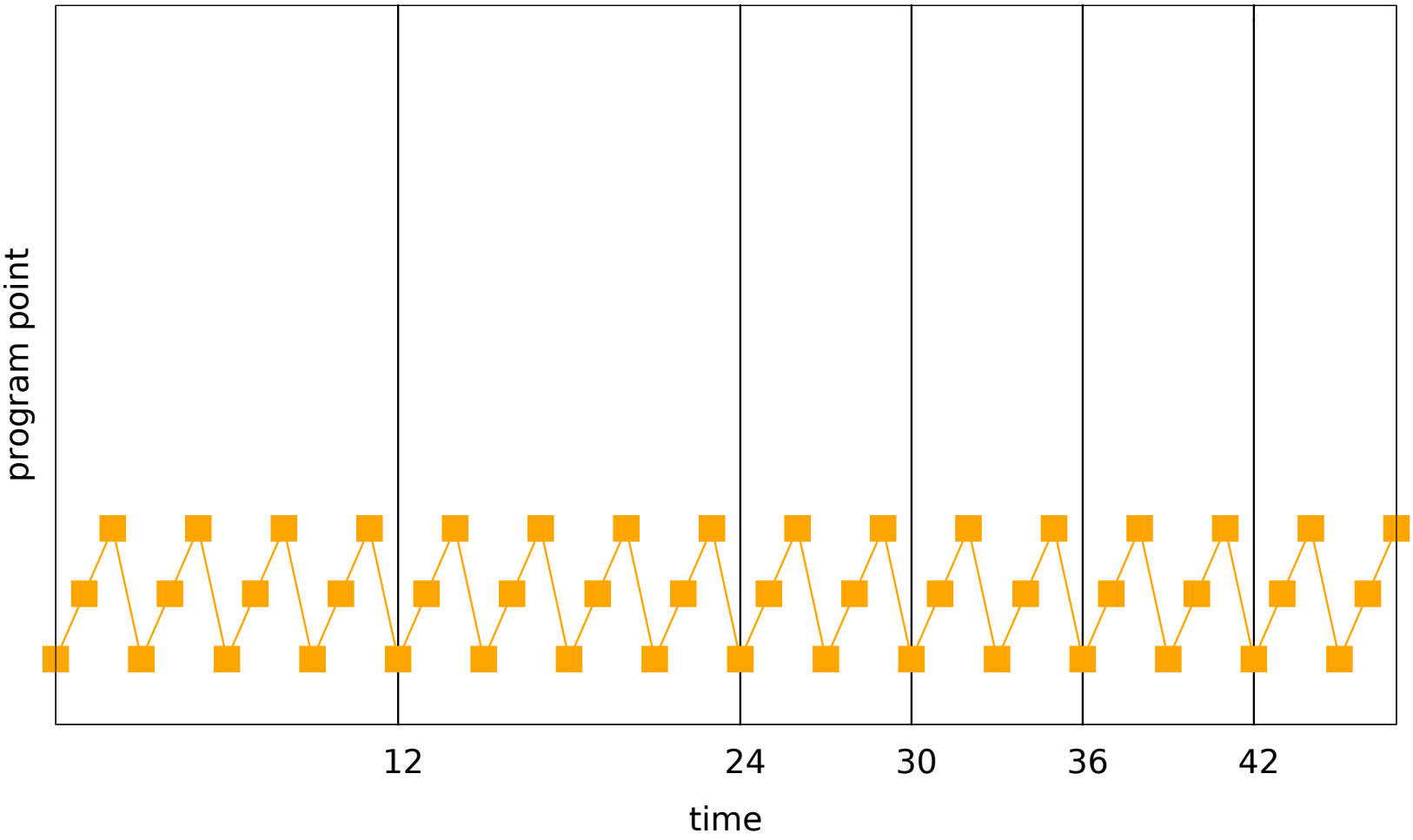
# Execution Trace of Loop

## Easy to make regular and uniform checkpoints



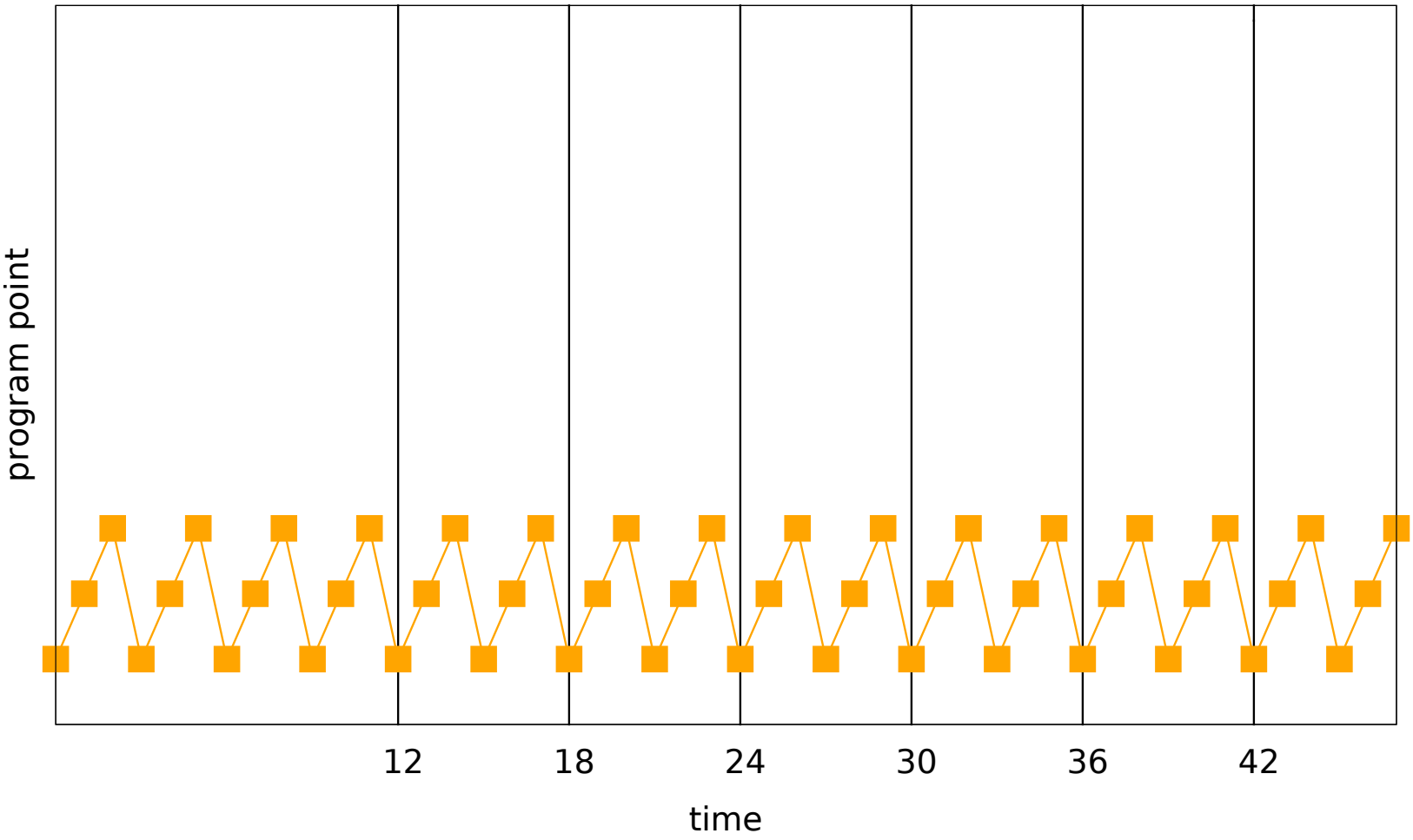
# Execution Trace of Loop

## Easy to make regular and uniform checkpoints



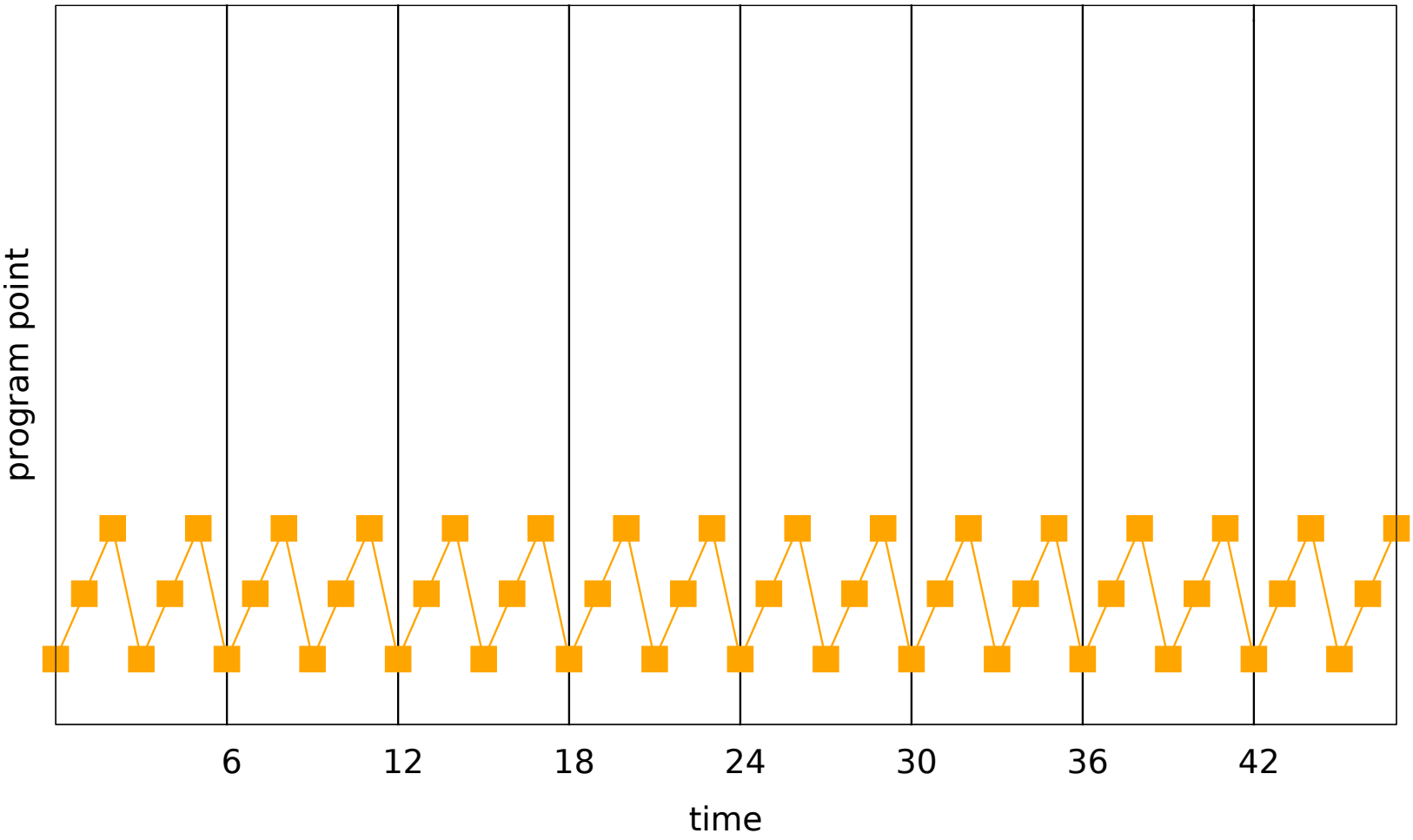
# Execution Trace of Loop

## Easy to make regular and uniform checkpoints

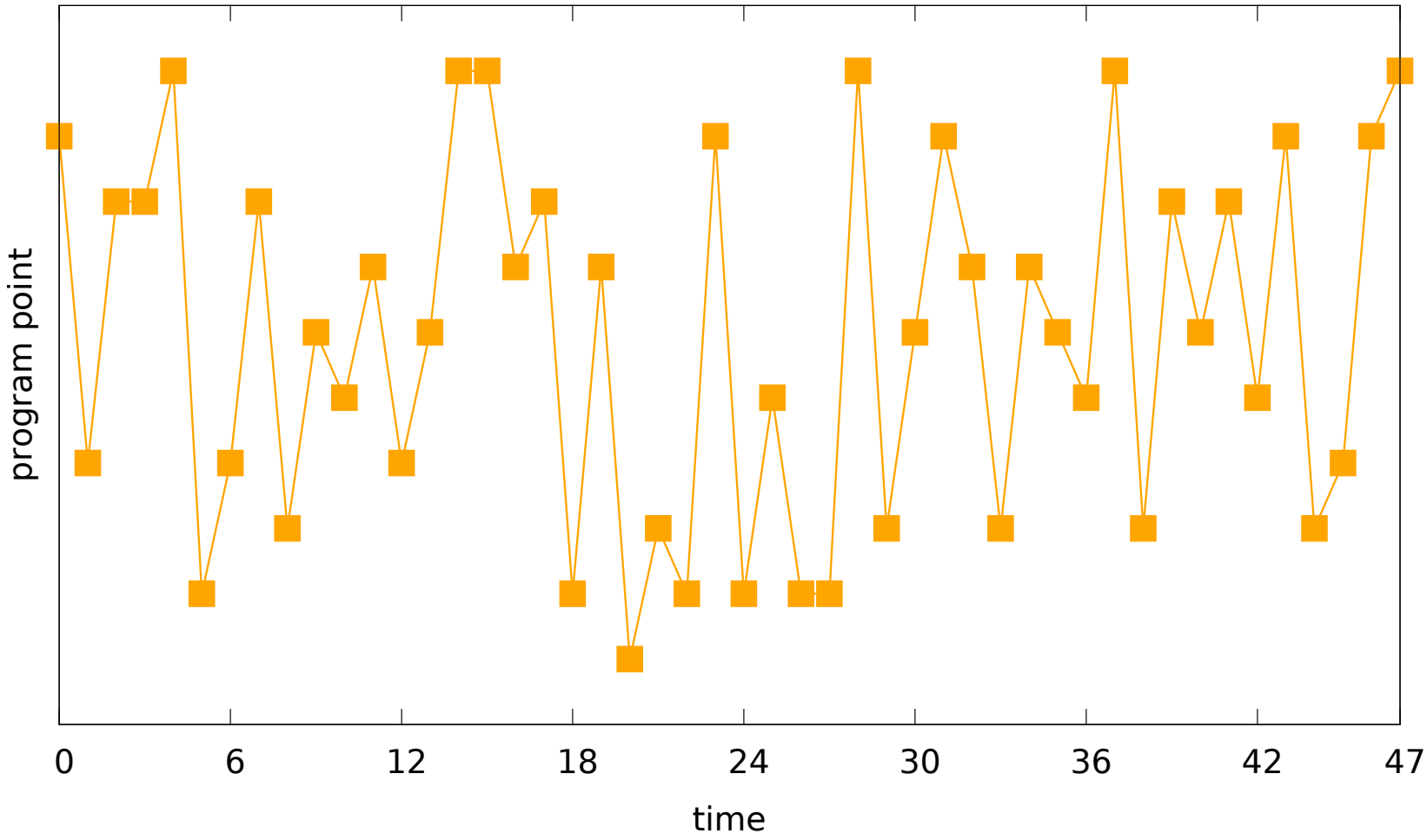


# Execution Trace of Loop

## Easy to make regular and uniform checkpoints

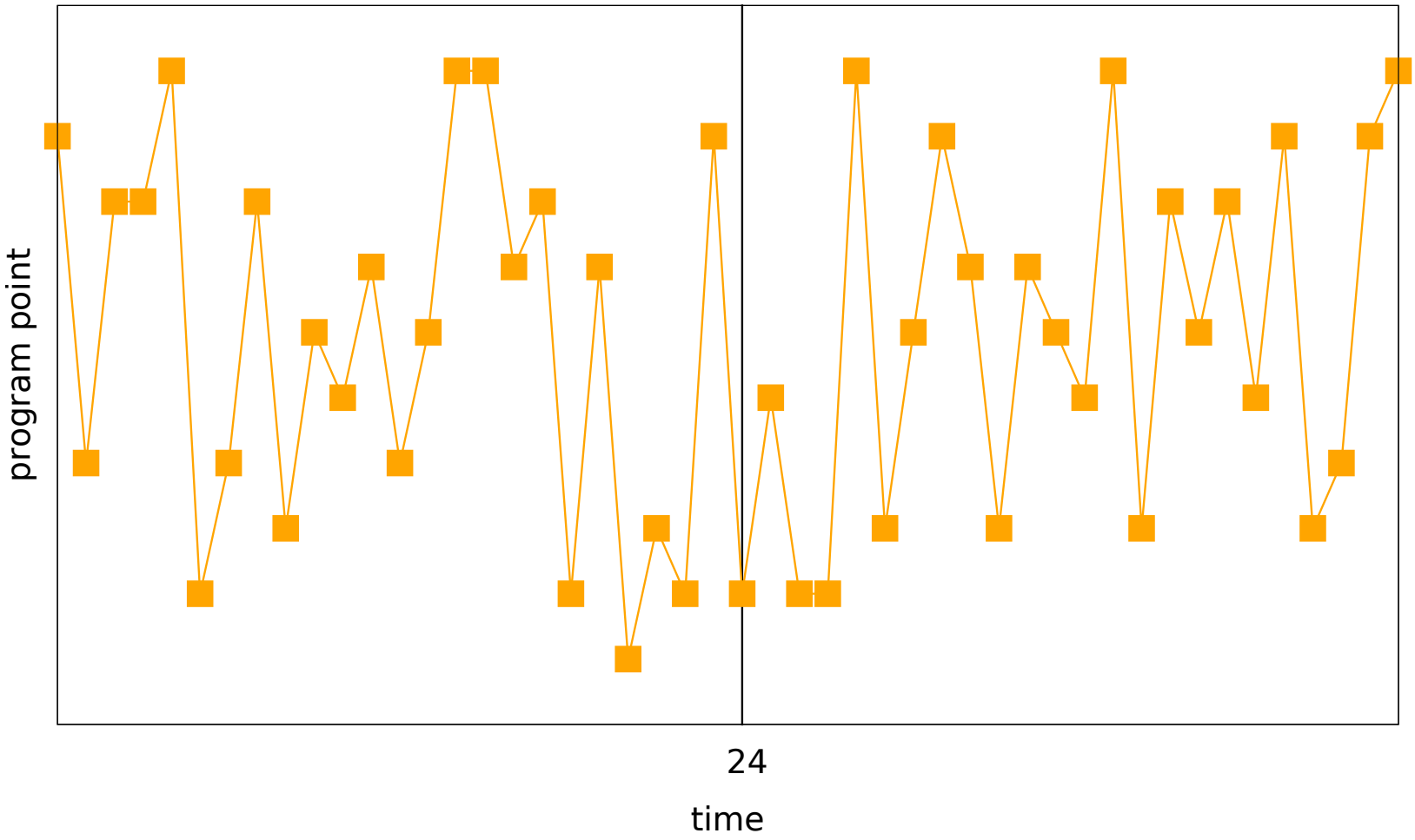


# Execution Trace of Arbitrary Code



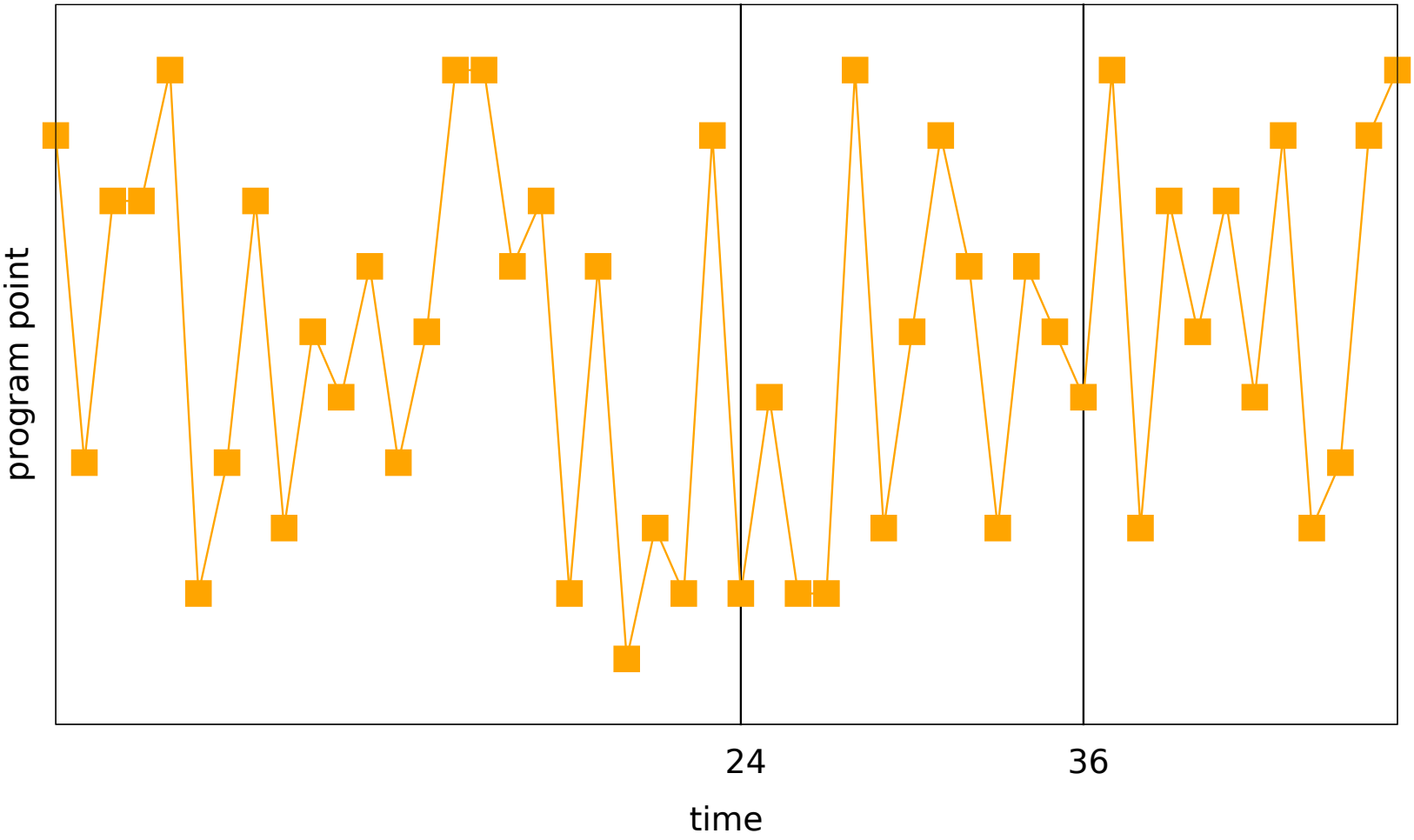
# Execution Trace of Arbitrary Code

## Difficult to make regular and uniform checkpoints



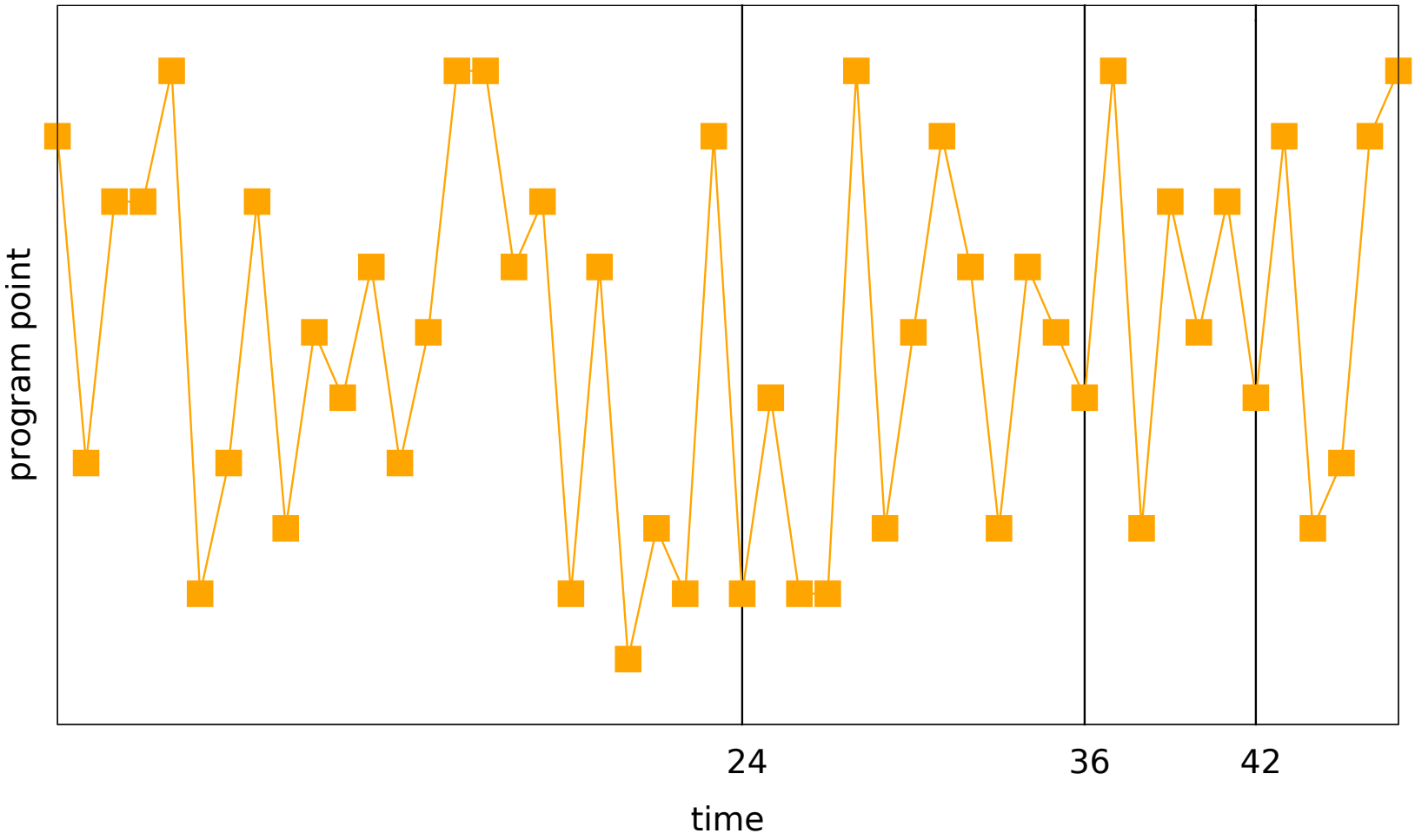
# Execution Trace of Arbitrary Code

## Difficult to make regular and uniform checkpoints



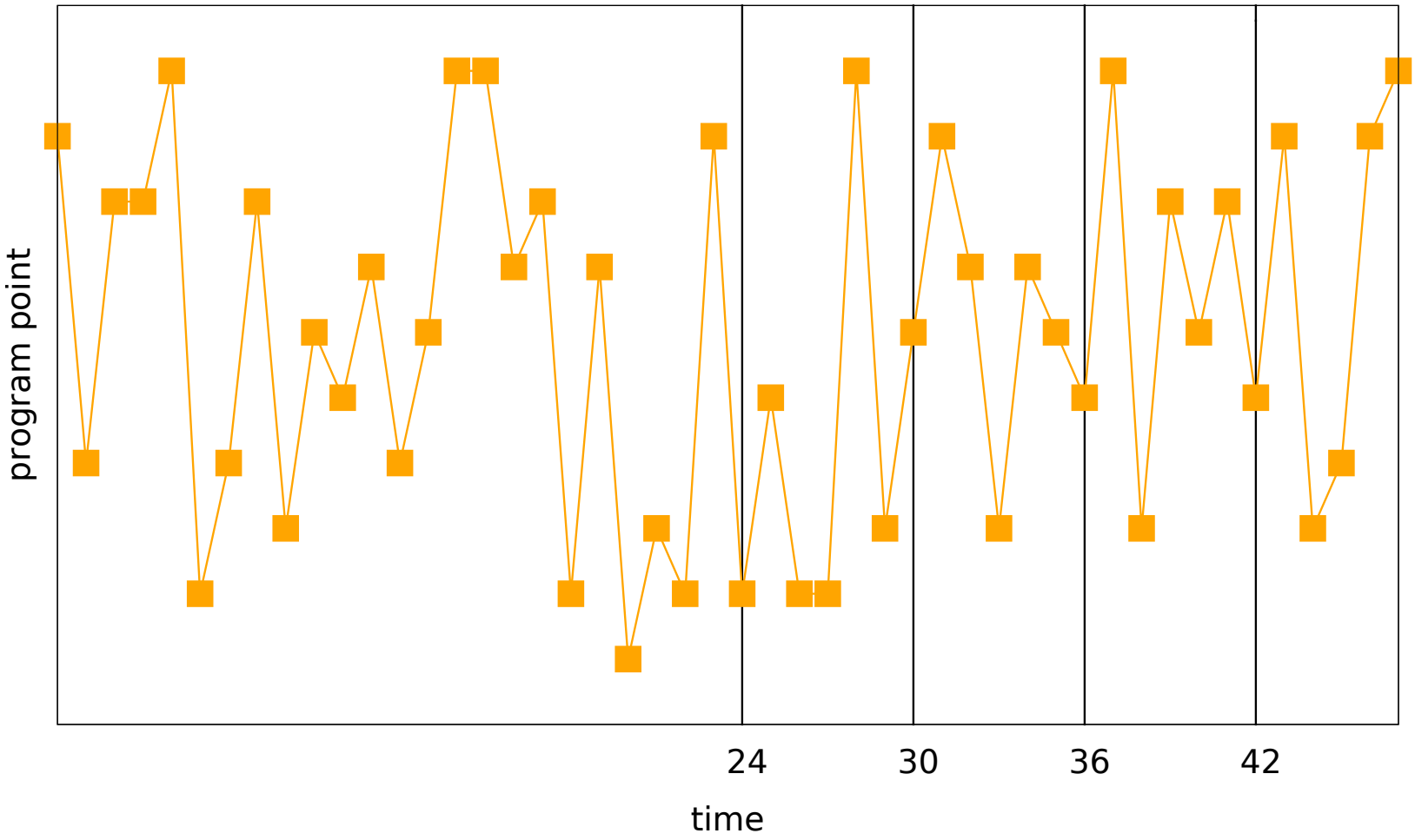
# Execution Trace of Arbitrary Code

## Difficult to make regular and uniform checkpoints



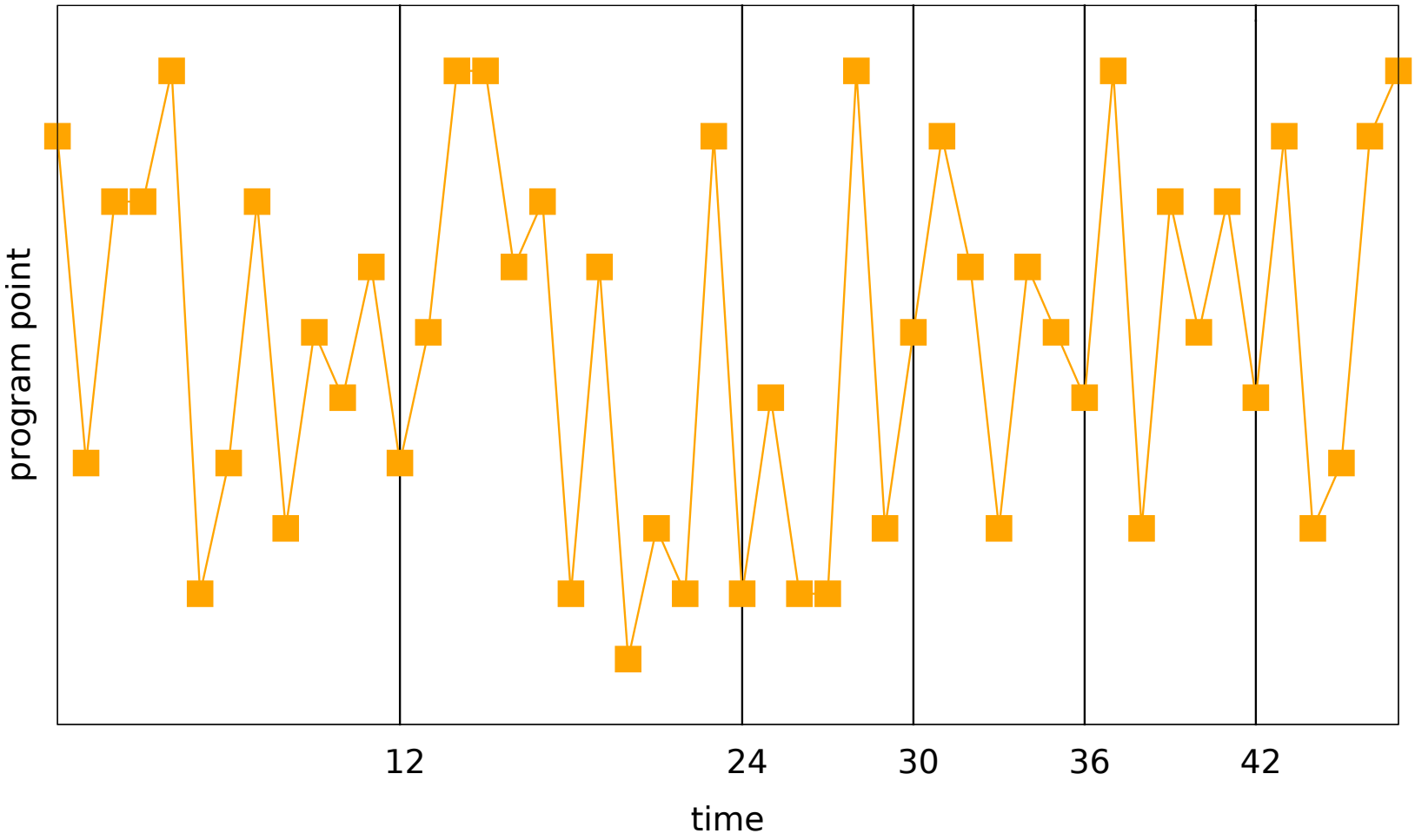
# Execution Trace of Arbitrary Code

## Difficult to make regular and uniform checkpoints



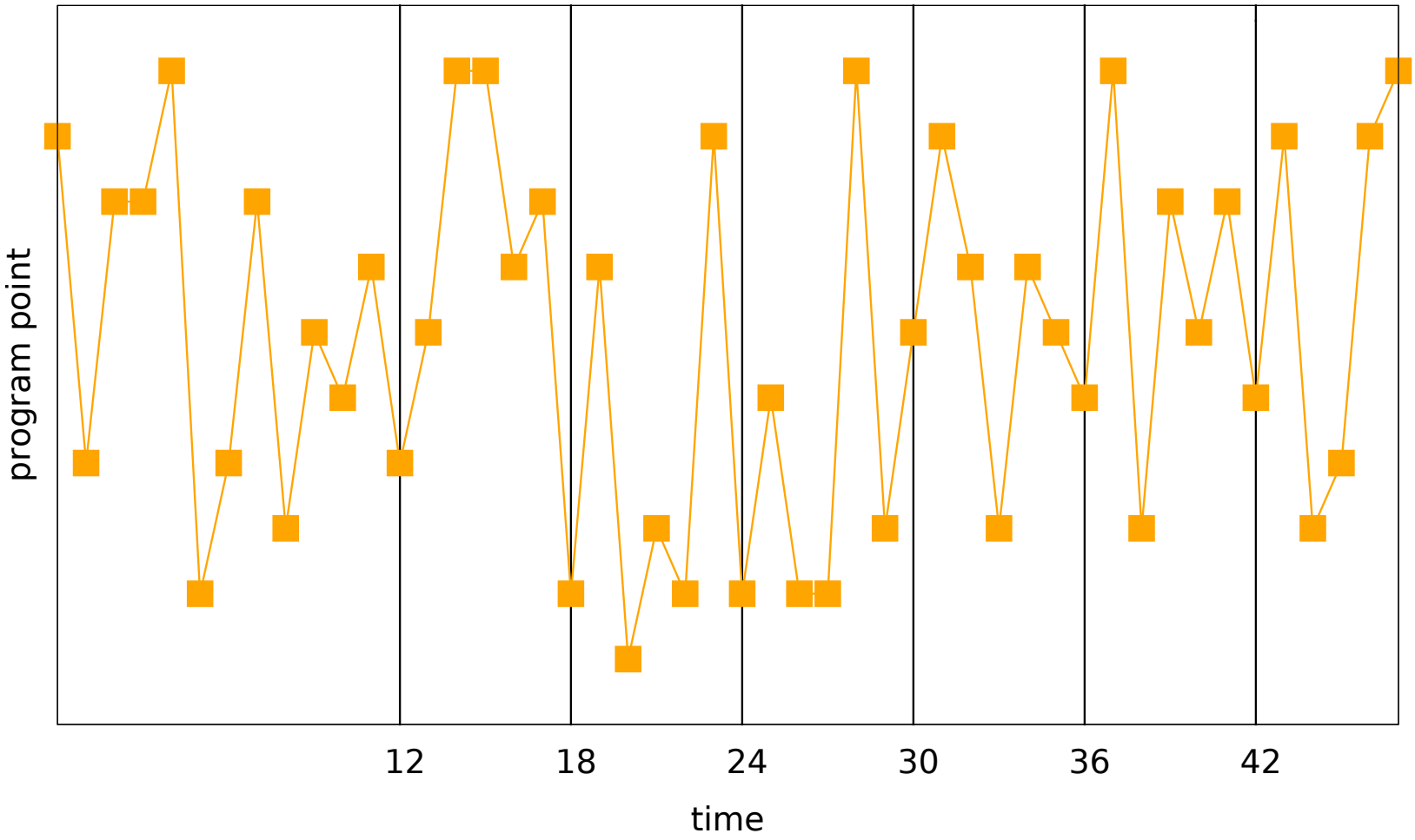
# Execution Trace of Arbitrary Code

## Difficult to make regular and uniform checkpoints



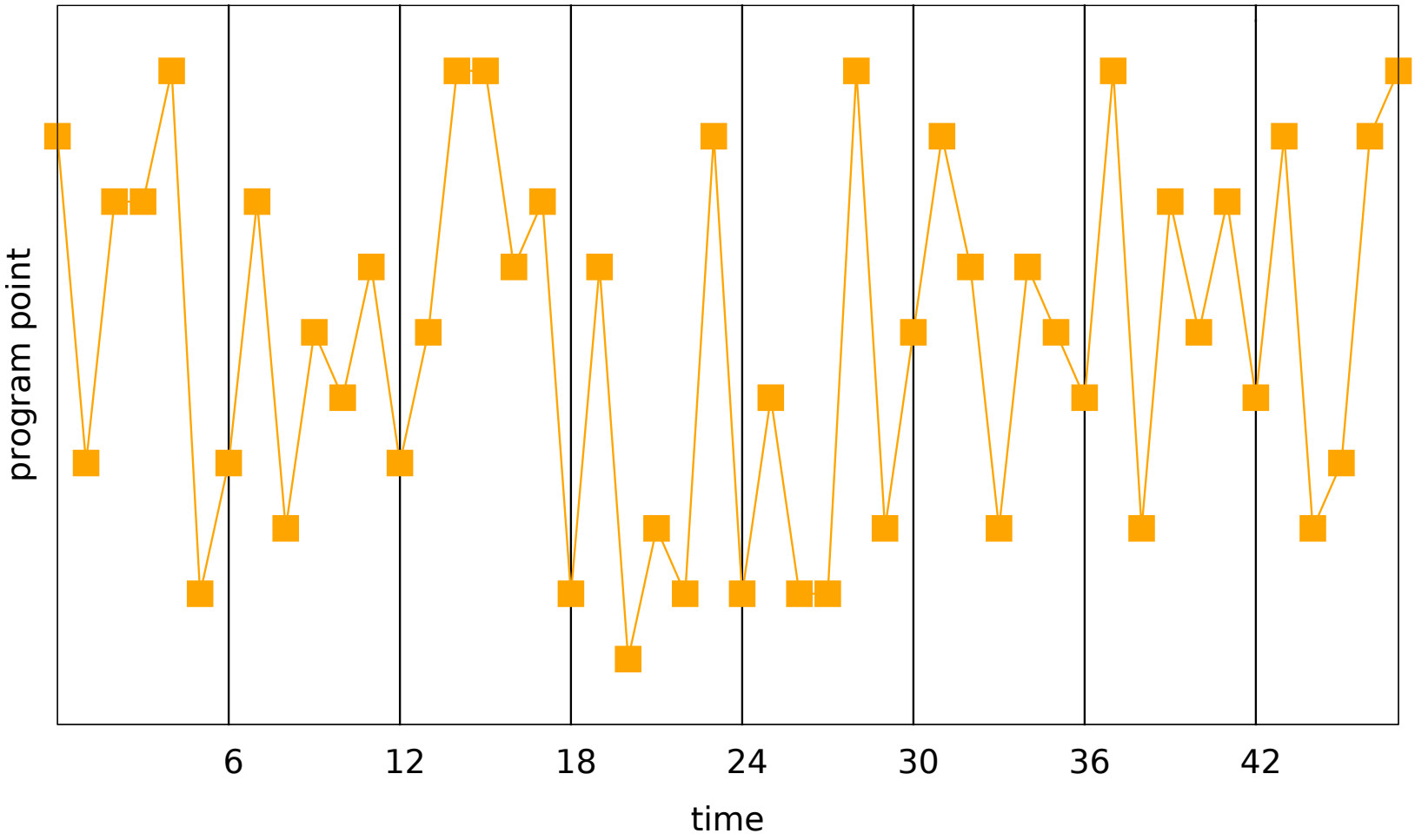
# Execution Trace of Arbitrary Code

## Difficult to make regular and uniform checkpoints



# Execution Trace of Arbitrary Code

## Difficult to make regular and uniform checkpoints



# Key Challenges

Need to interleave generation of the network with forward and backward passes through the network.

# Key Challenges

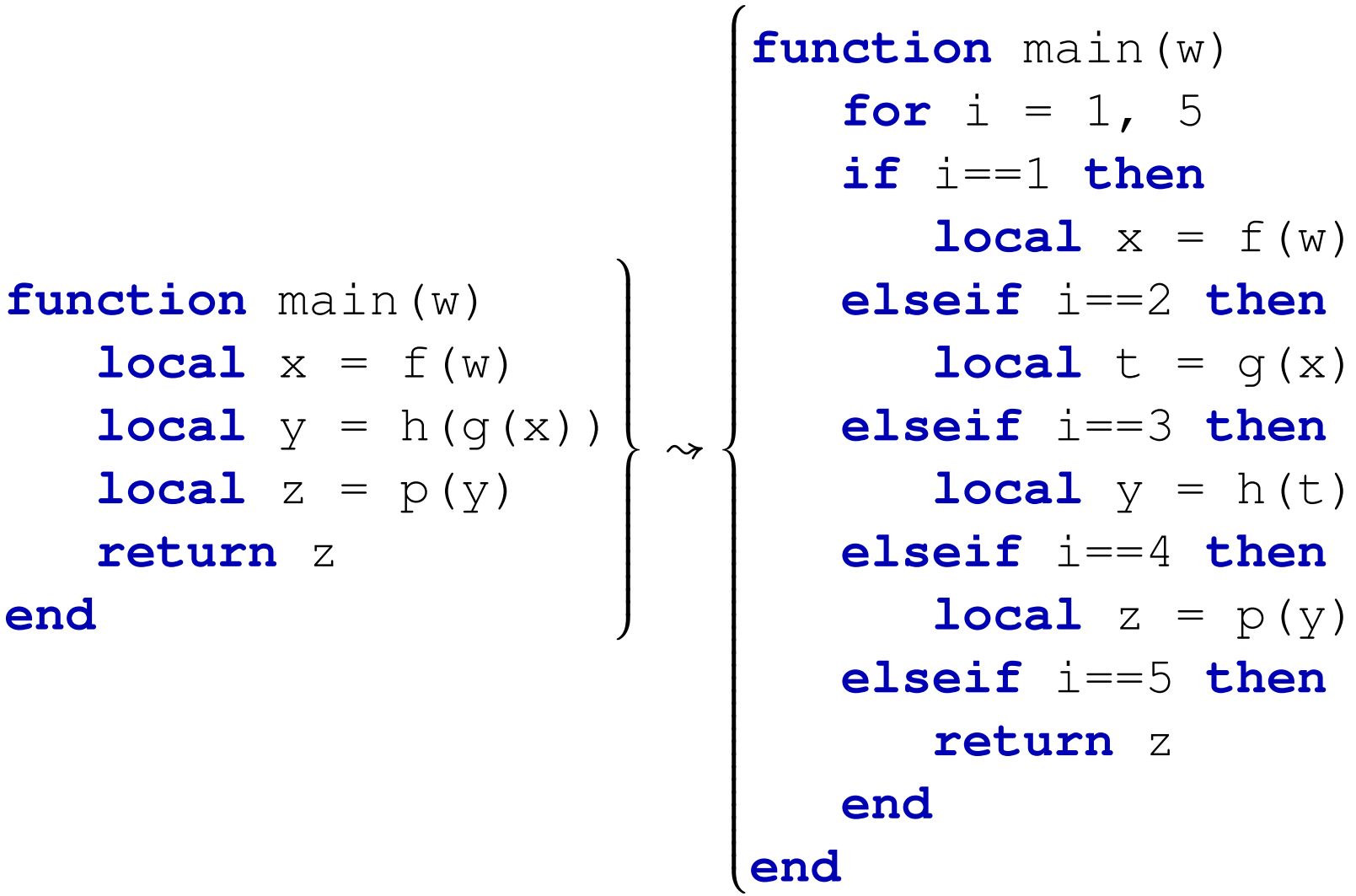
Need to interleave generation of the network with forward and backward passes through the network.

Portions of the network need to be (re)generated, and (re)evaluated with forward and backward passes, multiple times and out of order.

# Key Idea

```
function main(w)
    local x = f(w)
    local y = h(g(x))
    local z = p(y)
    return z
end
```

# Key Idea


function main(w)
local x = f(w)
local y = h(g(x))
local z = p(y)
return z
end
function main(w)
for i = 1, 5
if i==1 then
local x = f(w)
elseif i==2 then
local t = g(x)
elseif i==3 then
local y = h(t)
elseif i==4 then
local z = p(y)
elseif i==5 then
return z
end
end

# Core Language

$$e ::= c \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \diamond e \mid e_1 \bullet e_2$$

# Adding AD Operators to the Core Language

$$\overleftarrow{\mathcal{J}} : f\ x\ \grave{y} \mapsto (y, \grave{x})$$

$$\overset{\checkmark}{\mathcal{J}} : f\ x\ \grave{y} \mapsto (y, \grave{x})$$

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\swarrow}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f$ $x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\swarrow}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f$ $x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\swarrow}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\swarrow}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\swarrow}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f$ $x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\swarrow}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f$ $x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\swarrow}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f$ $x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\swarrow}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\swarrow}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\swarrow}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# What is Needed to Implement the Algorithm?

# What is Needed to Implement the Algorithm?

1. measure the length of the primal computation

# What is Needed to Implement the Algorithm?

1. measure the length of the primal computation
2. interrupt the primal computation at a portion of the measured length

# What is Needed to Implement the Algorithm?

1. measure the length of the primal computation
2. interrupt the primal computation at a portion of the measured length
3. save the state of the interrupted computation as a capsule

# What is Needed to Implement the Algorithm?

1. measure the length of the primal computation
2. interrupt the primal computation at a portion of the measured length
3. save the state of the interrupted computation as a capsule
4. resume an interrupted computation from a capsule

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# General-Purpose Interruption and Resumption Interface

| | |
|---|---|
| PRIMOPS $f\ x \mapsto l$ | Return the number $l$ of evaluation steps needed to compute $y = f(x)$. |
| INTERRUPT $f\ x\ l \mapsto z$ | Run the first $l$ steps of the computation of $f(x)$ and return a capsule $z$. |
| RESUME $z \mapsto y$ | If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$. |

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**: $l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**: $l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda z.\text{RESUME}\ z)\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda z.\text{RESUME}\ z)\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda z.\text{RESUME}\ z)\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

## via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda z.\text{RESUME}\ z)\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda x.\text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor)\ x\ \grave{z}$ (step 4)

# Example of CPS Conversion

```
function f(x)
   return q(p(g(x), h(x)))
end
```

$\rightsquigarrow$

```
function f(c, x)
   return g(function(t1)
                return h(function(t2)
                             return p(function(t3)
                                          return q(c, t3)
                                      end, t1, t2)
                         end, x)
            end, x)
end
```

# Implementation

# Implementation

1. Convert source program to CPS.

# Implementation

1. Convert source program to CPS.
2. Thread step count and limit.

# Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.

# Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\check{\mathcal{J}}$ written in C.

# Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\check{\mathcal{J}}$ written in C.
5. Compile to machine code.

# Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\check{\mathcal{J}}$ written in C.
5. Compile to machine code.

# CPS Conversion as a Program Transformation

$$\begin{aligned}
\lceil x|k \rceil &\rightsquigarrow k\ x \\
\lceil (\lambda x.e)|k \rceil &\rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil) \\
\lceil (e_1\ e_2)|k \rceil &\rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil \\
e_0 &\rightsquigarrow \lceil e_0|(\lambda x.x) \rceil
\end{aligned}$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2))\rceil)\rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x)\rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x) \rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x) \rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2))\rceil)\rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x)\rceil$$

# CPS Conversion as a Program Transformation

$$\begin{aligned}
\lceil x|k \rceil &\rightsquigarrow k\ x \\
\lceil (\lambda x.e)|k \rceil &\rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil) \\
\lceil (e_1\ e_2)|k \rceil &\rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2))\rceil)\rceil \\
e_0 &\rightsquigarrow \lceil e_0|(\lambda x.x) \rceil
\end{aligned}$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x) \rceil$$

# CPS Conversion as a Program Transformation

$$\begin{aligned}
\lceil x|k \rceil &\rightsquigarrow k\ x \\
\lceil (\lambda x.e)|k \rceil &\rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil) \\
\lceil (e_1\ e_2)|k \rceil &\rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2))\rceil)\rceil \\
e_0 &\rightsquigarrow \lceil e_0|(\lambda x.x) \rceil
\end{aligned}$$

# CPS Conversion as a Program Transformation

$$\begin{aligned}
\lceil x|k \rceil &\rightsquigarrow k\ x \\
\lceil (\lambda x.e)|k \rceil &\rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil) \\
\lceil (e_1\ e_2)|k \rceil &\rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil \\
e_0 &\rightsquigarrow \lceil e_0|(\lambda x.x) \rceil
\end{aligned}$$

# CPS Conversion as a Program Transformation

$$\begin{aligned}
\lceil x|k \rceil &\rightsquigarrow k\ x \\
\lceil (\lambda x.e)|k \rceil &\rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil) \\
\lceil (e_1\ e_2)|k \rceil &\rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil \\
e_0 &\rightsquigarrow \lceil e_0|(\lambda x.x) \rceil
\end{aligned}$$

# CPS Conversion as a Program Transformation

$$\begin{aligned}
\lceil x|k \rceil &\rightsquigarrow k\ x \\
\lceil (\lambda x.e)|k \rceil &\rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil) \\
\lceil (e_1\ e_2)|k \rceil &\rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil \\
e_0 &\rightsquigarrow \lceil e_0|(\lambda x.x) \rceil
\end{aligned}$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x) \rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x) \rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2))\rceil)\rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x)\rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x) \rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x) \rceil$$

# CPS Conversion as a Program Transformation

$$\begin{aligned}
\lceil x|k \rceil &\rightsquigarrow k\ x \\
\lceil (\lambda x.e)|k \rceil &\rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil) \\
\lceil (e_1\ e_2)|k \rceil &\rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil \\
e_0 &\rightsquigarrow \lceil e_0|(\lambda x.x) \rceil
\end{aligned}$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x) \rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x) \rceil$$

# CPS Conversion as a Program Transformation

$$\begin{aligned}
\lceil x|k \rceil &\rightsquigarrow k\ x \\
\lceil (\lambda x.e)|k \rceil &\rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil) \\
\lceil (e_1\ e_2)|k \rceil &\rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil \\
e_0 &\rightsquigarrow \lceil e_0|(\lambda x.x) \rceil
\end{aligned}$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x) \rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x) \rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2)) \rceil) \rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x) \rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2))\rceil)\rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x)\rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2))\rceil)\rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x)\rceil$$

# CPS Conversion as a Program Transformation

$$\lceil x|k \rceil \rightsquigarrow k\ x$$

$$\lceil (\lambda x.e)|k \rceil \rightsquigarrow k\ (\lambda k'\ x.\lceil e|k' \rceil)$$

$$\lceil (e_1\ e_2)|k \rceil \rightsquigarrow \lceil e_1|(\lambda x_1.\lceil e_2|(\lambda x_2.(x_1\ k\ x_2))\rceil)\rceil$$

$$e_0 \rightsquigarrow \lceil e_0|(\lambda x.x)\rceil$$

# Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\check{\mathcal{J}}$ written in C.
5. Compile to machine code.

# Treading Step Counts and Limits in CPS Conversion

$$
\begin{aligned}
\ulcorner x|k \urcorner &\rightsquigarrow k\ x \\
\ulcorner (\lambda x.e)|k \urcorner &\rightsquigarrow k\ (\lambda k\ x.\ulcorner e|k \urcorner) \\
\ulcorner (e_1\ e_2)|k \urcorner &\rightsquigarrow \ulcorner e_1|(\lambda\ x_1. \\
&\qquad \ulcorner e_2|(\lambda\ x_2. \\
&\qquad\quad (x_1\ k\ x_2)), \\
&\qquad \urcorner), \\
&\urcorner
\end{aligned}
$$

# Treading Step Counts and Limits in CPS Conversion

$$
\begin{aligned}
\lceil x|k \rceil &\rightsquigarrow k\ x \\
\lceil (\lambda x.e)|k \rceil &\rightsquigarrow k\ (\lambda k\ x.\lceil e|k \rceil) \\
\lceil (e_1\ e_2)|k \rceil &\rightsquigarrow \lceil e_1|(\lambda\ x_1. \\
&\qquad \lceil e_2|(\lambda\ x_2. \\
&\qquad\quad (x_1\ k\ x_2)), \\
&\qquad \rceil), \\
&\rceil
\end{aligned}
$$

# Treading Step Counts and Limits in CPS Conversion

$$
\begin{aligned}
\lceil x|k,n \rceil &\rightsquigarrow k\ (n+1)\ x \\
\lceil (\lambda x.e)|k,n \rceil &\rightsquigarrow k\ (n+1)\ (\lambda k\ n\ x.\lceil e|k,n \rceil) \\
\lceil (e_1\ e_2)|k,n \rceil &\rightsquigarrow \lceil e_1|(\lambda n\ x_1. \\
&\qquad \lceil e_2|(\lambda n\ x_2. \\
&\qquad\quad (x_1\ k\ n\ x_2)), \\
&\qquad\quad n \rceil), \\
&\qquad (n+1) \rceil
\end{aligned}
$$

# Treading Step Counts and Limits in CPS Conversion

$$
\begin{aligned}
\ulcorner x|k,n,l\urcorner &\rightsquigarrow k\ (n+1)\ l\ x \\
\ulcorner(\lambda x.e)|k,n,l\urcorner &\rightsquigarrow k\ (n+1)\ l\ (\lambda k\ n\ l\ x.\ulcorner e|k,n,l\urcorner) \\
\ulcorner(e_1\ e_2)|k,n,l\urcorner &\rightsquigarrow \ulcorner e_1|(\lambda n\ l\ x_1. \\
&\qquad \ulcorner e_2|(\lambda n\ l\ x_2. \\
&\qquad\quad (x_1\ k\ n\ l\ x_2)), \\
&\qquad n,l\urcorner), \\
&\quad (n+1),l\urcorner
\end{aligned}
$$

# Treading Step Counts and Limits in CPS Conversion

$$
\begin{aligned}
\ulcorner x|k,n,l\urcorner &\rightsquigarrow\ k\ (n+1)\ l\ x \\
\ulcorner(\lambda x.e)|k,n,l\urcorner &\rightsquigarrow\ k\ (n+1)\ l\ (\lambda k\ n\ l\ x.\ulcorner e|k,n,l\urcorner) \\
\ulcorner(e_1\ e_2)|k,n,l\urcorner &\rightsquigarrow\ \ulcorner e_1|(\lambda n\ l\ x_1. \\
&\qquad \ulcorner e_2|(\lambda n\ l\ x_2. \\
&\qquad\qquad (x_1\ k\ n\ l\ x_2)), \\
&\qquad\quad n,l\urcorner), \\
&\qquad (n+1),l\urcorner
\end{aligned}
$$

$$\langle\!\langle e \rangle\!\rangle_{k,n,l} \rightsquigarrow \textbf{if } n = l \textbf{ then } [\![k, \lambda k\ n\ l\ \_.e]\!] \textbf{ else } e$$

# Treading Step Counts and Limits in CPS Conversion

$$
\begin{aligned}
\ulcorner x|k,n,l\urcorner &\rightsquigarrow \langle\!\langle k\ (n+1)\ l\ x\rangle\!\rangle_{k,n,l} \\
\ulcorner(\lambda x.e)|k,n,l\urcorner &\rightsquigarrow \langle\!\langle k\ (n+1)\ l\ (\lambda k\ n\ l\ x.\ulcorner e|k,n,l\urcorner)\rangle\!\rangle_{k,n,l} \\
\ulcorner(e_1\ e_2)|k,n,l\urcorner &\rightsquigarrow \langle\!\langle \ulcorner e_1|(\lambda n\ l\ x_1.\\
&\qquad \ulcorner e_2|(\lambda n\ l\ x_2.\\
&\qquad\quad (x_1\ k\ n\ l\ x_2)),\\
&\qquad n,l\urcorner),\\
&\quad (n+1),l\urcorner\rangle\!\rangle_{k,n,l}
\end{aligned}
$$

$$
\langle\!\langle e\rangle\!\rangle_{k,n,l} \rightsquigarrow \textbf{if } n = l \textbf{ then } [\![k, \lambda k\ n\ l\ \_.e]\!] \textbf{ else } e
$$

# Treading Step Counts and Limits in CPS Conversion

$$
\begin{aligned}
\ulcorner\llcorner x|k,n,l\urcorner\lrcorner &\rightsquigarrow \langle\!\langle k\ (n+1)\ l\ x\rangle\!\rangle_{k,n,l} \\
\ulcorner\llcorner(\lambda x.e)|k,n,l\urcorner\lrcorner &\rightsquigarrow \langle\!\langle k\ (n+1)\ l\ (\lambda k\ n\ l\ x.\ulcorner\llcorner e|k,n,l\urcorner\lrcorner)\rangle\!\rangle_{k,n,l} \\
\ulcorner\llcorner(e_1\ e_2)|k,n,l\urcorner\lrcorner &\rightsquigarrow \langle\!\langle\ulcorner\llcorner e_1|(\lambda n\ l\ x_1.\\
&\qquad\ulcorner\llcorner e_2|(\lambda n\ l\ x_2.\\
&\qquad\qquad(x_1\ k\ n\ l\ x_2)),\\
&\qquad\quad n,l\urcorner\lrcorner),\\
&\quad(n+1),l\urcorner\lrcorner\rangle\!\rangle_{k,n,l}
\end{aligned}
$$

$$\vdots$$

$$\langle\!\langle e\rangle\!\rangle_{k,n,l} \rightsquigarrow \textbf{if } n = l \textbf{ then } [\![k, \lambda k\ n\ l\ \_.e]\!] \textbf{ else } e$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k, f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \perp$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

# Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\checkmark\mathcal{J}$ written in C.
5. Compile to machine code.

# Code Generation

$$\mathcal{S}\,\pi\,() = \texttt{null\_constant}$$
$$\mathcal{S}\,\pi\,\textbf{true} = \texttt{true\_constant}$$
$$\mathcal{S}\,\pi\,\textbf{false} = \texttt{false\_constant}$$
$$\mathcal{S}\,\pi\,(c_1, c_2) = \texttt{cons}((\mathcal{S}\,\pi\,c_1),\ (\mathcal{S}\,\pi\,c_1))$$
$$\mathcal{S}\,\pi\,n$$
$$\mathcal{S}\,\pi\,\texttt{`k'} = \texttt{continuation}$$
$$\mathcal{S}\,\pi\,\texttt{`n'} = \texttt{count}$$
$$\mathcal{S}\,\pi\,\texttt{`l'} = \texttt{limit}$$
$$\mathcal{S}\,\pi\,\texttt{`x'} = \texttt{argument}$$
$$\mathcal{S}\,\pi\,x = \texttt{as\_closure(target)->environment}[\pi\,x]$$

# Code Generation

$\mathcal{S}\ \pi\ (\lambda_3 n\ l\ x.e)$ = ({

```
  thing function(thing target,
                 thing count,
                 thing limit,
                 thing argument) {
  return (S (φ e) e);
  }
  thing lambda = (thing)GC_malloc(sizeof(struct {
    enum tag tag;
    struct {
      thing (*function)();
      unsigned n;
      thing environment[|φ e|];
    }))
  }
  set_closure(lambda);
  as_closure(lambda)->function = &function;
  as_closure(lambda)->n = |φ e|;
  as_closure(lambda)->environment[0] = S π (φ e)0
  ⋮
  as_closure(lambda)->environment[|φ e| - 1] = S π (φ e)|φ e|-1
  lambda;
})
```

# Code Generation

$\mathcal{S}\ \pi\ (\lambda_4 k\ n\ l\ x.e)$ = ({

```
thing function(thing target,
               thing continuation,
               thing count,
               thing limit,
               thing argument) {
return (S (φ e) e);
}
thing lambda = (thing)GC_malloc(sizeof(struct {
  enum tag tag;
  struct {
    thing (*function)();
    unsigned n;
    thing environment[|φ e|];
  }))
}
set_closure(lambda);
as_closure(lambda)->function = &function;
as_closure(lambda)->n = |φ e|;
as_closure(lambda)->environment[0] = S π (φ e)0
⋮
as_closure(lambda)->environment[|φ e| - 1] = S π (φ e)|φ e|-1
lambda;
})
```

# Code Generation

$$\mathcal{S}\,\pi\,(e_1\ e_2\ e_3\ e_4) = \texttt{continuation\_apply}\,((\mathcal{S}\,\pi\,e_1),\ (\mathcal{S}\,\pi\,e_2),\ (\mathcal{S}\,\pi\,e_3),\ (\mathcal{S}\,\pi\,e_4))$$

$$\mathcal{S}\,\pi\,(e_1\ e_2\ e_3\ e_4\ e_5) = \texttt{converted\_apply}\,((\mathcal{S}\,\pi\,e_1),\ (\mathcal{S}\,\pi\,e_2),\ (\mathcal{S}\,\pi\,e_3),\ (\mathcal{S}\,\pi\,e_4),\ (\mathcal{S}\,\pi\,e_5))$$

$$\mathcal{S}\,\pi\,(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) = \texttt{(!is\_false}\,((\mathcal{S}\,\pi\,e_1))\,\texttt{?}(\mathcal{S}\,\pi\,e_2)\texttt{:}(\mathcal{S}\,\pi\,e_3))$$

$$\mathcal{S}\,\pi\,(\diamond e) = (\mathcal{N}\,\diamond)\,((\mathcal{S}\,\pi\,e))$$

$$\mathcal{S}\,\pi\,(e_1 \bullet e_2) = (\mathcal{N}\,\bullet)\,((\mathcal{S}\,\pi\,e_1),\ (\mathcal{S}\,\pi\,e_2))$$

$$\mathcal{S}\,\pi\,(\overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = (\mathcal{N}\,\overrightarrow{\mathcal{J}})\,((\mathcal{S}\,\pi\,e_1),\ (\mathcal{S}\,\pi\,e_2),\ (\mathcal{S}\,\pi\,e_3))$$

$$\mathcal{S}\,\pi\,(\overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = (\mathcal{N}\,\overleftarrow{\mathcal{J}})\,((\mathcal{S}\,\pi\,e_1),\ (\mathcal{S}\,\pi\,e_2),\ (\mathcal{S}\,\pi\,e_3))$$

$$\mathcal{S}\,\pi\,(\overset{\checkmark}{\mathcal{J}}\ e_1\ e_2\ e_3) = (\mathcal{N}\,\overset{\checkmark}{\mathcal{J}})\,((\mathcal{S}\,\pi\,e_1),\ (\mathcal{S}\,\pi\,e_2),\ (\mathcal{S}\,\pi\,e_3))$$

# Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\checkmark\mathcal{I}$ written in C.
5. Compile to machine code.

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \perp$$

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

written in C

```
static thing lambda_expression_that_returns_x
(thing f, thing n, thing l, thing x) {
  return x;
}

static thing lambda_expression_that_returns_n
(thing f, thing n, thing l, thing x) {
  return n;
}

static thing lambda_expression_that_resumes
(thing f, thing continuation, thing n, thing l, thing x) {
  if (!is_interrupt(x)) internal_error();
  return converted_apply(as_interrupt(x)->closure,
                         as_interrupt(x)->continuation,
                         make_real(0.0),
                         l,
                         null_constant);
}
```

# Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

```
static unsigned long primops(thing f, thing x) {
  thing result = converted_apply(f,
                                 continuation_that_returns_n,
                                 make_real(0.0),
                                 make_real(HUGE_VAL),
                                 x);
  else if (is_real(result)) return (unsigned long)as_real(result);
}

static thing interrupt(thing f, thing x, thing l) {
  thing result = converted_apply(f,
                                 continuation_that_returns_x,
                                 make_real(0.0),
                                 l,
                                 x);
  if (!is_interrupt(result)) internal_error();
  return result;
}
```

# Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

**base case** ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

**inductive case**:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

# Algorithm for Divide-and-Conquer Checkpointing

written in C

```
static thing checkpoint_starj(thing f, thing x, thing y_cotangent)
{
  thing loop(thing f, thing x, thing y_cotangent, unsigned long l) {
    if (l<=base_case_duration) return ternary_starj(f, x, y_cotangent);
    else {
      thing u = interrupt(f, x, make_real(l/2));
      thing y_u_cotangent = loop(closure_that_resumes, u, y_cotangent, l-l/2);
      if (!is_pair(y_u_cotangent)) internal_error();
      thing u_x_cotangent =
        loop(make_closure_for_interrupt(f, l/2),
             x,
             as_pair(y_u_cotangent)->cdr,
             l/2);
      if (!is_pair(u_x_cotangent)) internal_error();
      return cons(as_pair(y_u_cotangent)->car,
                  as_pair(u_x_cotangent)->cdr);
    }
  }
  return loop(f, x, y_cotangent, primops(f, x));
}
```

# Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\checkmark\mathcal{J}$ written in C.
5. Compile to machine code.

# Determinant Example

```
(define (car (cons car cdr)) car)

(define (cdr (cons car cdr)) cdr)

(define (matrix-rows a)
 (if (null? a) 0 (+ (matrix-rows (cdr a)) 1)))

(define (list-ref l i)
 (if (zero? i) (car l) (list-ref (cdr l) (- i 1))))

(define (matrix-ref a i j) (list-ref (list-ref a i) j))

(define (list-set l i x)
 (if (zero? i)
     (cons x (cdr l))
     (cons (car l) (list-set (cdr l) (- i 1) x))))

(define (matrix-set a i j x)
 (list-set a i (list-set (list-ref a i) j x)))

(define (map-n f n)
 (if (zero? n) '() (cons (f (- n 1)) (map-n f (- n 1)))))

(define (identity-matrix n)
 (map-n (lambda (i) (map-n (lambda (j) (if (= i j) 1 0)) n)) n))
```

# Determinant Example

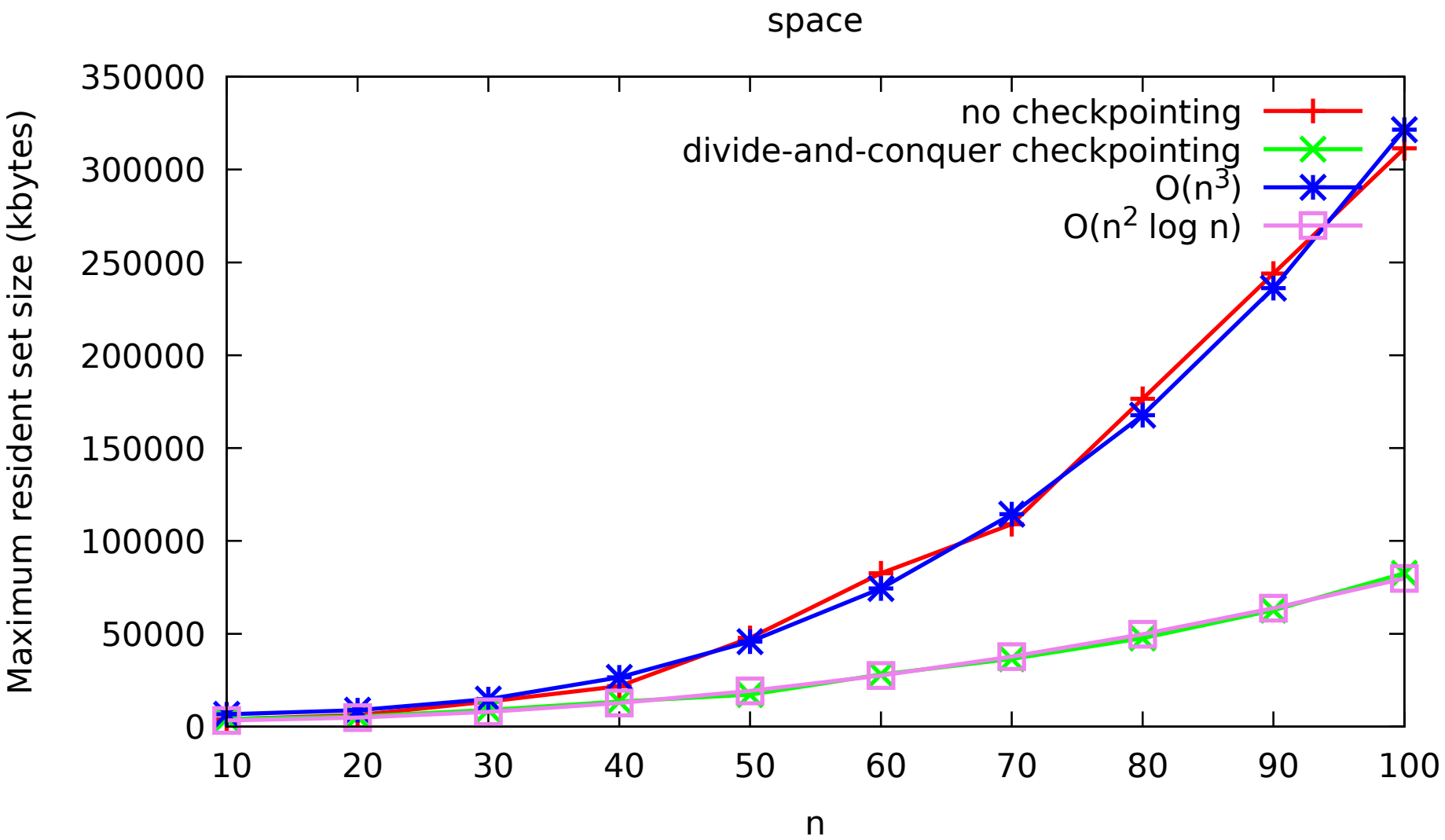
```
(define (determinant a)
 (let ((n (matrix-rows a)))
  (let loop ((i 0) (b a) (d 1))
   (if (= i n)
       d
       (let* ((c (matrix-ref b i i))
              (b (let loop ((j i) (b b))
                   (if (= j n)
                       b
                       (loop (+ j 1) (matrix-set b i j (/ (matrix-ref b i j) c)))))))
        (loop (+ i 1)
              (let loop ((j (+ i 1)) (b b))
               (if (= j n)
                   b
                   (loop (+ j 1)
                         (let ((e (matrix-ref b j i)))
                          (let loop ((k (+ i 1)) (b b))
                           (if (= k n)
                               b
                               (loop (+ k 1)
                                     (matrix-set b j k
                                                 (- (matrix-ref b j k)
                                                    (* e (matrix-ref b i k))))))))))))
              (* d c)))))))

(write-real (determinant (cdr (checkpoint-*j determinant (identity-matrix (read-real)) 1))))
```

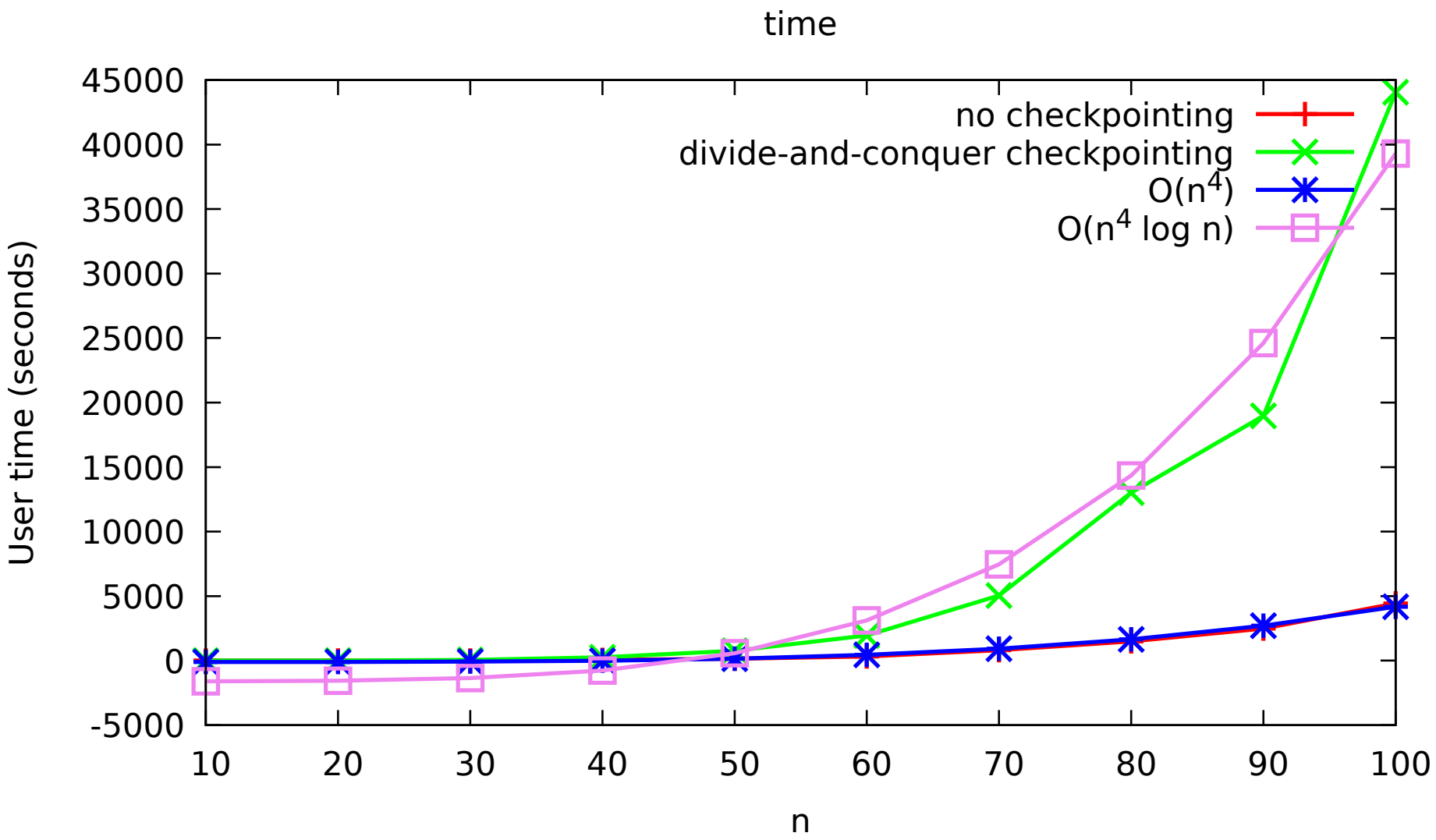
# Complexity Analysis

| | space | time |
|---|---|---|
| primal | $O(n^2)$ | $O(n^4)$ |
| no checkpointing | $O(n^3)$ | $O(n^4)$ |
| divide-and-conquer checkpointing | $O(n^2 \log n)$ | $O(n^4 \log n)$ |

# Space Usage of the Determinant Example



# Time Usage of the Determinant Example



# Three Reference Implementations

# Three Reference Implementations

1. Interpreter using CPS evaluator

# Three Reference Implementations

1. Interpreter using CPS evaluator
2. Hybrid compiler/interpreter using CPS conversion followed by direct-style evaluator

# Three Reference Implementations

1. Interpreter using CPS evaluator
2. Hybrid compiler/interpreter using CPS conversion followed by direct-style evaluator
3. Compiler using CPS conversion followed by translation to C

# Three Reference Implementations

1. Interpreter using CPS evaluator
2. Hybrid compiler/interpreter using CPS conversion followed by direct-style evaluator
3. Compiler using CPS conversion followed by translation to C

All three support exact same source language. No exceptions.
Same space and time complexity. Differ only in constant factors.

# Three Reference Implementations

1. Interpreter using CPS evaluator
2. Hybrid compiler/interpreter using CPS conversion followed by direct-style evaluator
3. Compiler using CPS conversion followed by translation to C

All three support exact same source language. No exceptions.
Same space and time complexity. Differ only in constant factors.

Could add FFI bindings to GPU Tensor library.
Could serve as reference for rewrite to PYTHON,. . . .

# Three Reference Implementations

1. Interpreter using CPS evaluator
2. Hybrid compiler/interpreter using CPS conversion followed by direct-style evaluator
3. Compiler using CPS conversion followed by translation to C

All three support exact same source language. No exceptions.
Same space and time complexity. Differ only in constant factors.

Could add FFI bindings to GPU Tensor library.
Could serve as reference for rewrite to PYTHON,. . . .

All three documented in detail in manuscript.
`http://arxiv.org/abs/1708.06799`

# Three Reference Implementations

1. Interpreter using CPS evaluator
2. Hybrid compiler/interpreter using CPS conversion followed by direct-style evaluator
3. Compiler using CPS conversion followed by translation to C

All three support exact same source language. No exceptions.
Same space and time complexity. Differ only in constant factors.

Could add FFI bindings to GPU Tensor library.
Could serve as reference for rewrite to PYTHON,. . . .

All three documented in detail in manuscript.
`http://arxiv.org/abs/1708.06799`

Will release when manuscript is accepted.

# Take-Home Message

Divide-and-Conquer checkpointing

# Take-Home Message

Divide-and-Conquer checkpointing

- is traditionally formulated around loop iterations

# Take-Home Message

Divide-and-Conquer checkpointing

- is traditionally formulated around loop iterations
- but can be extended to arbitrary code

# Take-Home Message

Divide-and-Conquer checkpointing

- is traditionally formulated around loop iterations
- but can be extended to arbitrary code
- that doesn't have same-size iterations of a single loop

# Take-Home Message

Divide-and-Conquer checkpointing

- is traditionally formulated around loop iterations
- but can be extended to arbitrary code
- that doesn't have same-size iterations of a single loop
- using CPS to make arbitrary code look like it does.

# Take-Home Message

Divide-and-Conquer checkpointing

- is traditionally formulated around loop iterations
- but can be extended to arbitrary code
- that doesn't have same-size iterations of a single loop
- using CPS to make arbitrary code look like it does.

metaphor: a CPU is an instruction-execution loop

# Thank You