Automatic differentiation

Matthew J Johnson (mattjj@google.com)

Deep Learning Summer School
Montreal 2017



Dougal Maclaurin   David Duvenaud   Ryan P Adams

HARVARD
INTELLIGENT
PROBABILISTIC
SYSTEMS

Our awesome new world

Our awesome new world

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

Our awesome new world

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

- loops? branching? recursion? closures? data structures?

Our awesome new world

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

- loops? branching? recursion? closures? data structures?
- debugger?

Our awesome new world

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

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

Our awesome new world

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

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

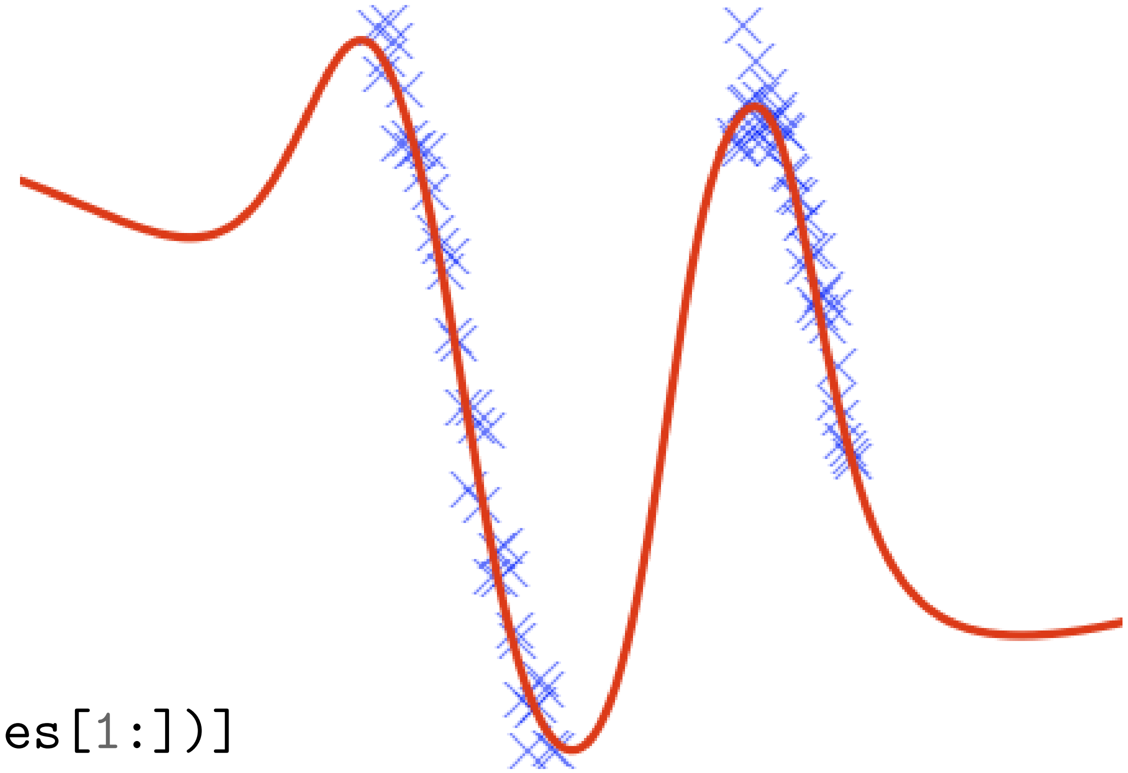
Autograd

- github.com/hips/autograd
  - differentiates native Python code
  - handles most of Numpy + Scipy
  - loops, branching, recursion, closures
  - arrays, tuples, lists, dicts, classes, …
  - derivatives of derivatives
  - a one-function API
  - small and easy to extend

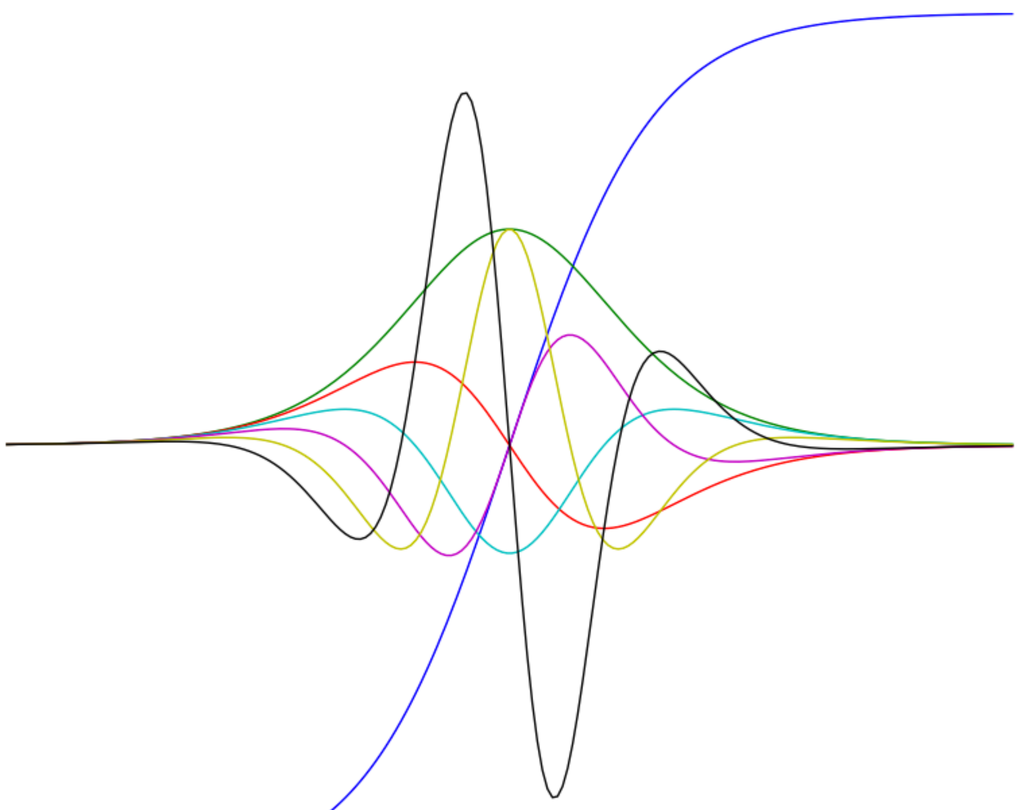
Dougal Maclaurin

Autograd examples

```
import autograd.numpy as np
import auotgrad.numpy.random as npr
from autograd import grad

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

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

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

gradient_fun = grad(logprob_fun)
```



Autograd examples



```python
import autograd.numpy as np
from autograd import grad
import matplotlib.pyplot as plt

x = np.linspace(-7, 7, 200)
plt.plot(x, np.tanh(x),
         x, grad(np.tanh)(x),
         x, grad(grad(np.tanh))(x),
         x, grad(grad(grad(np.tanh)))(x),
         x, grad(grad(grad(grad(np.tanh))))(x),
         x, grad(grad(grad(grad(grad(np.tanh)))))(x),
         x, grad(grad(grad(grad(grad(grad(np.tanh))))))(x))
```

Hessians and HVPs

```
from autograd import grad, jacobian

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

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

$$\nabla^2 f(x) \cdot v = \nabla_x (\nabla_x f(x) \cdot v)$$

Black-box inference in a tweet

Ryan Adams @ryan_p_adams · 7 Nov 2015

@DavidDuvenaud

```
def elbo(p, lp, D, N):
 v=exp(p[D:])
 s=randn(N,D)*sqrt(v)+p[:D]
 return mvn.entropy(0, diag(v))+mean(lp(s))
gf = grad(elbo)
```

1 7 22

Tutorial goals

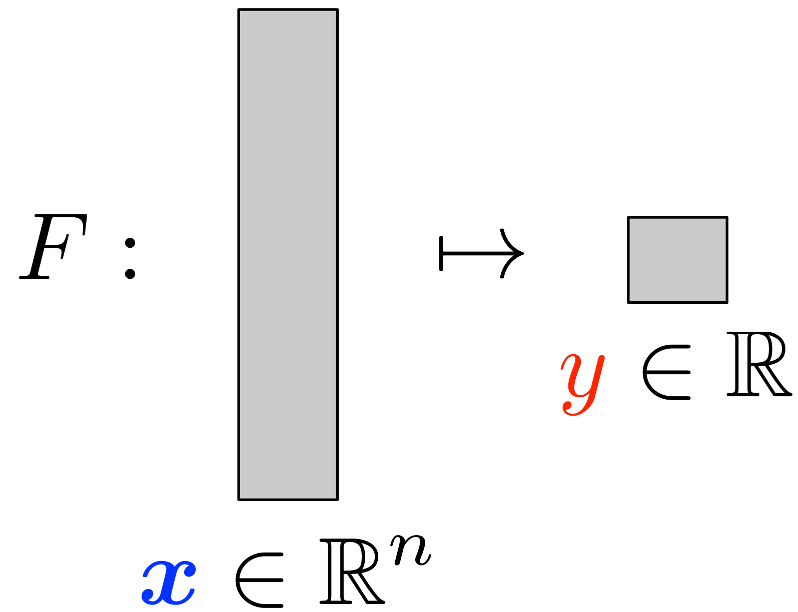
1. Jacobians and the chain rule
    - Forward and reverse accumulation
2. Autograd’s implementation
    - Fully closed tracing autodiff in Python
3. Advanced autodiff techniques
    - Checkpointing, forward from reverse, differentiating optima and fixed points

Tutorial goals

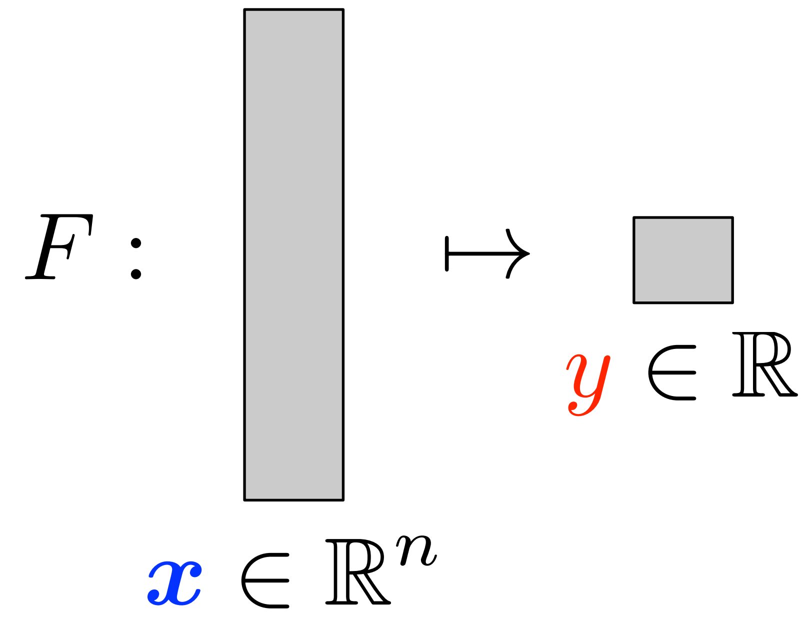
1. Jacobians and the chain rule
    - Forward and reverse accumulation
2. Autograd's implementation
    - Fully closed tracing autodiff in Python
3. Advanced autodiff techniques
    - Checkpointing, forward from reverse, differentiating optima and fixed points

$$F : \mathbb{R}^n \to \mathbb{R}$$

$$F : \mathbb{R}^n \to \mathbb{R}$$

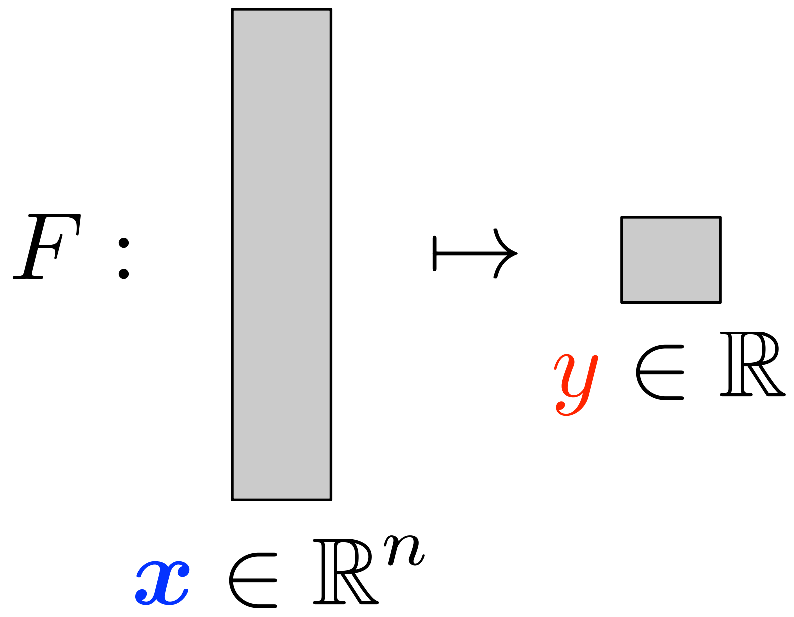


$$F : \mathbb{R}^n \to \mathbb{R}$$



$$F = D \circ C \circ B \circ A$$

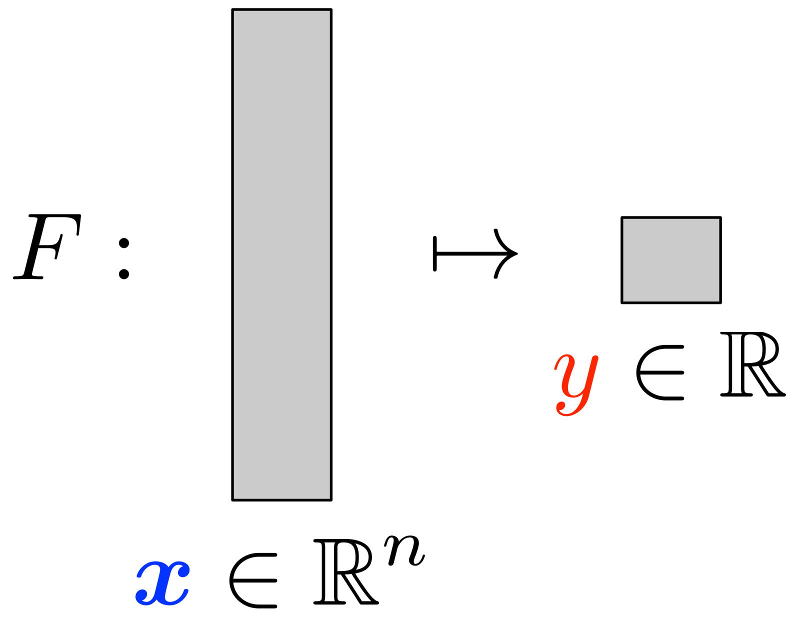
$$F : \mathbb{R}^n \to \mathbb{R}$$



$$F = D \circ C \circ B \circ A$$

$$y = F(\boldsymbol{x}) = D(C(B(A(\boldsymbol{x}))))$$

$$F : \mathbb{R}^n \to \mathbb{R}$$



$$F = D \circ C \circ B \circ A \qquad y = F(\boldsymbol{x}) = D(C(B(A(\boldsymbol{x}))))$$

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

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

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

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

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

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

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

$$F'(\boldsymbol{x}) = \quad \frac{\partial y}{\partial \boldsymbol{c}} \; \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \; \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \; \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

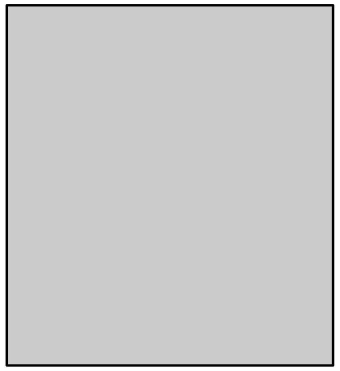
$$\frac{\partial y}{\partial \boldsymbol{c}} = D'(\boldsymbol{c})$$

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

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

$$F'(\boldsymbol{x}) = \quad \frac{\partial y}{\partial \boldsymbol{c}} \; \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \; \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \; \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\frac{\partial y}{\partial \boldsymbol{c}} = D'(\boldsymbol{c}) \qquad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} = C'(\boldsymbol{b})$$

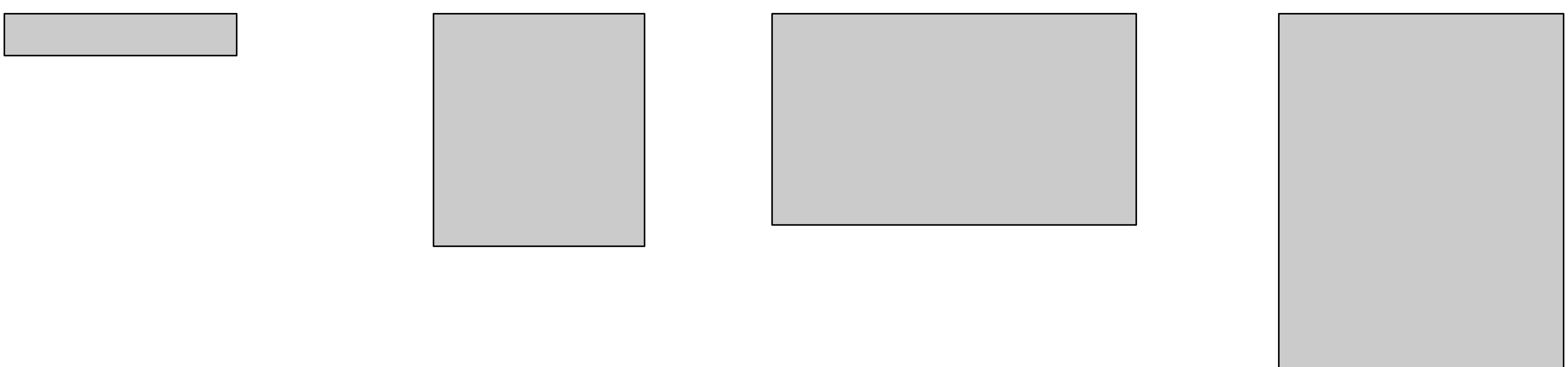


$$\color{red}{y} = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\color{blue}{\boldsymbol{x}})$$

$$F'(\color{blue}{\boldsymbol{x}}) = \frac{\partial \color{red}{y}}{\partial \color{blue}{\boldsymbol{x}}} = \left[\frac{\partial \color{red}{y}}{\partial \color{blue}{x_1}} \quad \cdots \quad \frac{\partial \color{red}{y}}{\partial \color{blue}{x_n}} \right]$$

$$F'(\color{blue}{\boldsymbol{x}}) = \quad \frac{\partial \color{red}{y}}{\partial \boldsymbol{c}} \; \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \; \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \; \frac{\partial \boldsymbol{a}}{\partial \color{blue}{\boldsymbol{x}}}$$

$$\frac{\partial \color{red}{y}}{\partial \boldsymbol{c}} = D'(\boldsymbol{c}) \qquad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} = C'(\boldsymbol{b}) \qquad \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} = B'(\boldsymbol{a}) \qquad \frac{\partial \boldsymbol{a}}{\partial \color{blue}{\boldsymbol{x}}} = A'(\color{blue}{\boldsymbol{x}})$$



$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}}\left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\left(\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \quad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\right)\right.$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}} \left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \left(\underbrace{\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \quad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}} \right) \right)$$

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

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}}\left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\left(\underbrace{\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \quad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}}\right)\right)$$

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

Forward accumulation

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}}\left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\left(\underbrace{\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \quad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}}\right)\right)$$

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

Forward accumulation

$$F'(\boldsymbol{x}) = \left(\left(\frac{\partial y}{\partial \boldsymbol{c}} \quad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\right)\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\right)\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}}\left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\left(\underbrace{\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \quad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}}\right)\right)$$

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

Forward accumulation

$$F'(\boldsymbol{x}) = \left(\left(\underbrace{\frac{\partial y}{\partial \boldsymbol{c}} \quad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}}\right)\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\right)\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\frac{\partial y}{\partial \boldsymbol{b}} = \begin{bmatrix} \frac{\partial y}{\partial b_1} & \cdots & \frac{\partial y}{\partial b_m} \end{bmatrix}$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}}\left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\underbrace{\left(\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \quad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\right)}_{}\right)$$

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

Forward accumulation

$$F'(\boldsymbol{x}) = \left(\underbrace{\left(\frac{\partial y}{\partial \boldsymbol{c}} \quad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\right)}_{}\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\right)\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\frac{\partial y}{\partial \boldsymbol{b}} = \begin{bmatrix} \frac{\partial y}{\partial b_1} & \cdots & \frac{\partial y}{\partial b_m} \end{bmatrix}$$

Reverse accumulation

$$F'(\boldsymbol{x})\,\boldsymbol{v} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \; \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \; \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \; \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\,\boldsymbol{v}$$

$$F'(\boldsymbol{x})\,\boldsymbol{v} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \quad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \quad \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \quad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\ \boldsymbol{v}$$

$$F'(\boldsymbol{x})\,\boldsymbol{v} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}}\left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\left(\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}\left(\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\ \boldsymbol{v}\right)\right)\right)$$

$$F'(\boldsymbol{x})\ \boldsymbol{v} = \quad \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \ \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \ \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \ \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\ \boldsymbol{v}$$

$$F'(\boldsymbol{x})\ \boldsymbol{v} = \quad \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \left(\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \left(\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\ \boldsymbol{v} \right) \right) \right)$$

Forward accumulation $\leftrightarrow$ Jacobian-vector products

Build Jacobian one column at a time

$$F'(\boldsymbol{x})\,\boldsymbol{v} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \; \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \; \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \; \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} \, \boldsymbol{v}$$

$$F'(\boldsymbol{x})\,\boldsymbol{v} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \left(\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \left(\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} \, \boldsymbol{v} \right) \right) \right)$$

Forward accumulation $\leftrightarrow$ Jacobian-vector products

Build Jacobian one column at a time

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}} \left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \left(\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \left(\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} \; \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{x}} \right) \right) \right)$$

$$\boldsymbol{v}^{\mathsf{T}} F'(\boldsymbol{x}) = \boldsymbol{v}^{\mathsf{T}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\boldsymbol{v}^{\mathsf{T}} F'(\boldsymbol{x}) = \boldsymbol{v}^{\mathsf{T}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\boldsymbol{v}^{\mathsf{T}} F'(\boldsymbol{x}) = \left(\left(\left(\left(\boldsymbol{v}^{\mathsf{T}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \right) \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \right) \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \right) \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} \right) \Bigg) \Bigg)$$

$$\boldsymbol{v}^{\mathsf{T}} F'(\boldsymbol{x}) = \qquad \boldsymbol{v}^{\mathsf{T}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \quad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \quad \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \quad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\boldsymbol{v}^{\mathsf{T}} F'(\boldsymbol{x}) = \left(\left(\left(\boldsymbol{v}^{\mathsf{T}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \right) \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \right) \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \right) \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} \Bigg) \Bigg) \Bigg)$$

Reverse accumulation $\quad\leftrightarrow\quad$ vector-Jacobian products

Build Jacobian one row at a time

$$\boldsymbol{v}^\mathsf{T} F'(\boldsymbol{x}) = \qquad \boldsymbol{v}^\mathsf{T} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \quad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \quad \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \quad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\boldsymbol{v}^\mathsf{T} F'(\boldsymbol{x}) = \left(\left(\left(\boldsymbol{v}^\mathsf{T} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \right) \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \right) \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \right) \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} \Bigg) \Bigg) \Bigg)$$

Reverse accumulation $\leftrightarrow$ vector-Jacobian products

Build Jacobian one row at a time

$$F'(\boldsymbol{x}) = \left(\left(\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{y}} \, \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \right) \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \right) \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \right) \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} \Bigg) \Bigg) \Bigg)$$

Forward and reverse accumulation

- Forward accumulation
  - Jacobian-vector products
  - “push-forward”
  - build Jacobian matrix one column at a time
- Reverse accumulation
  - vector-Jacobian products
  - “pull-back”
  - build Jacobian matrix one row at a time

Non-chain composition

Non-chain composition

Fan-in $\quad y = F(\boldsymbol{x}_1, \boldsymbol{x}_2)$

Non-chain composition

Fan-in $\quad y = F(\boldsymbol{x}_1, \boldsymbol{x}_2)$

$$\frac{\partial y}{\partial \boldsymbol{x}_1} = F'_1(\boldsymbol{x}_1, \boldsymbol{x}_2) \qquad \frac{\partial y}{\partial \boldsymbol{x}_2} = F'_2(\boldsymbol{x}_1, \boldsymbol{x}_2)$$

Non-chain composition

Fan-in

$$y = F(\boldsymbol{x}_1, \boldsymbol{x}_2)$$

$$\frac{\partial y}{\partial \boldsymbol{x}_1} = F_1'(\boldsymbol{x}_1, \boldsymbol{x}_2) \qquad \frac{\partial y}{\partial \boldsymbol{x}_2} = F_2'(\boldsymbol{x}_1, \boldsymbol{x}_2)$$

Fan-out

$$G(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} I \\ I \end{bmatrix} \boldsymbol{x}$$

Non-chain composition

Fan-in

$$y = F(\boldsymbol{x}_1, \boldsymbol{x}_2)$$

$$\frac{\partial y}{\partial \boldsymbol{x}_1} = F_1'(\boldsymbol{x}_1, \boldsymbol{x}_2) \qquad \frac{\partial y}{\partial \boldsymbol{x}_2} = F_2'(\boldsymbol{x}_1, \boldsymbol{x}_2)$$

Fan-out

$$G(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} I \\ I \end{bmatrix} \boldsymbol{x}$$

$$G'(\boldsymbol{x}) = \begin{bmatrix} I \\ I \end{bmatrix} \qquad \boldsymbol{v}^\top G'(\boldsymbol{x}) = \begin{bmatrix} \boldsymbol{v}_1{}^\top & \boldsymbol{v}_2{}^\top \end{bmatrix} \begin{bmatrix} I \\ I \end{bmatrix} = \boldsymbol{v}_1{}^\top + \boldsymbol{v}_2{}^\top$$

Tutorial goals

1. Jacobians and the chain rule
    - Forward and reverse accumulation
2. Autograd's implementation
    - Fully closed tracing autodiff in Python
3. Advanced autodiff techniques
    - Checkpointing, forward from reverse, differentiating optima and fixed points

Autodiff implementations

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

Autodiff implementations

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

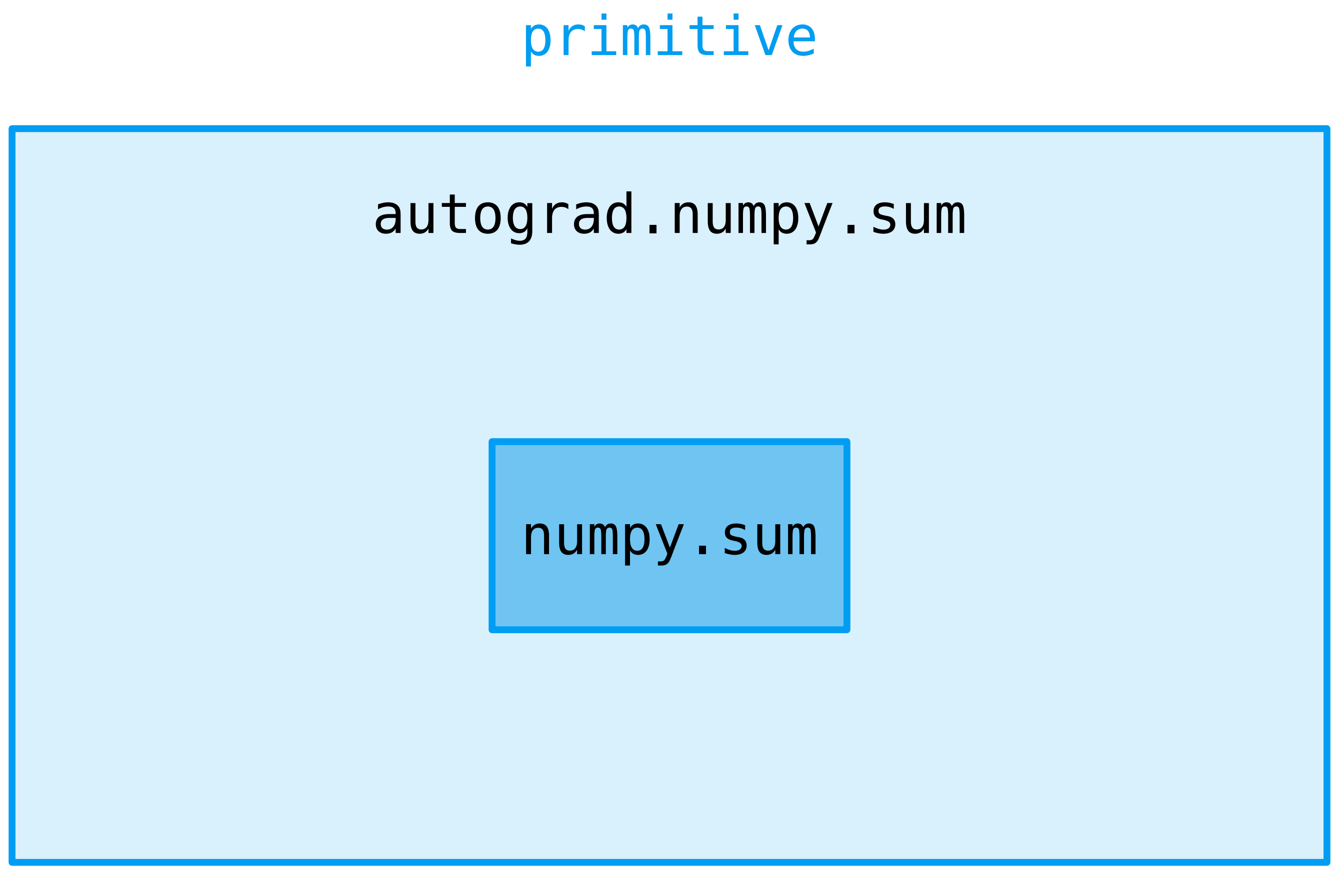
Autograd's ingredients

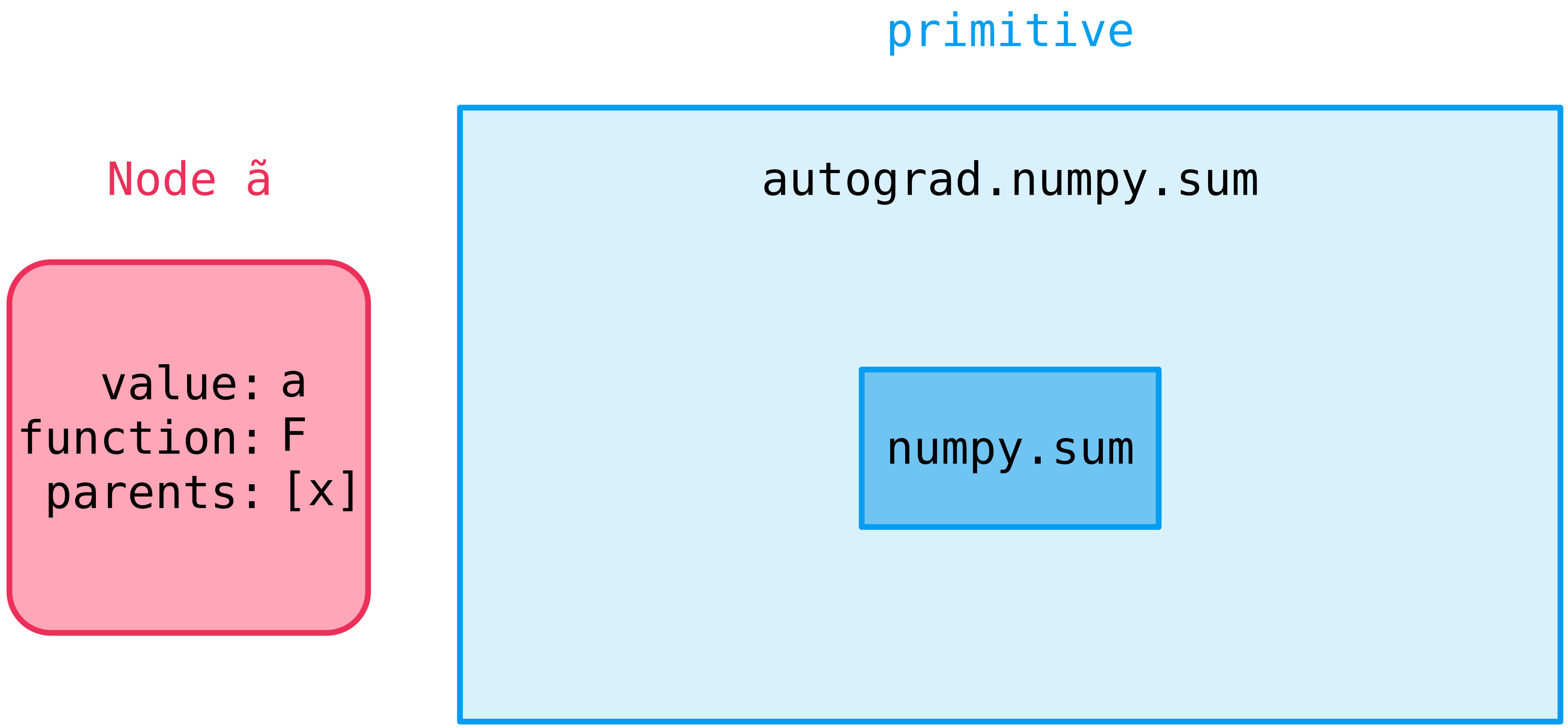
1. Tracing the composition of primitive functions
2. Defining a vector-Jacobian product (VJP) operator for each primitive
3. Composing VJPs backward

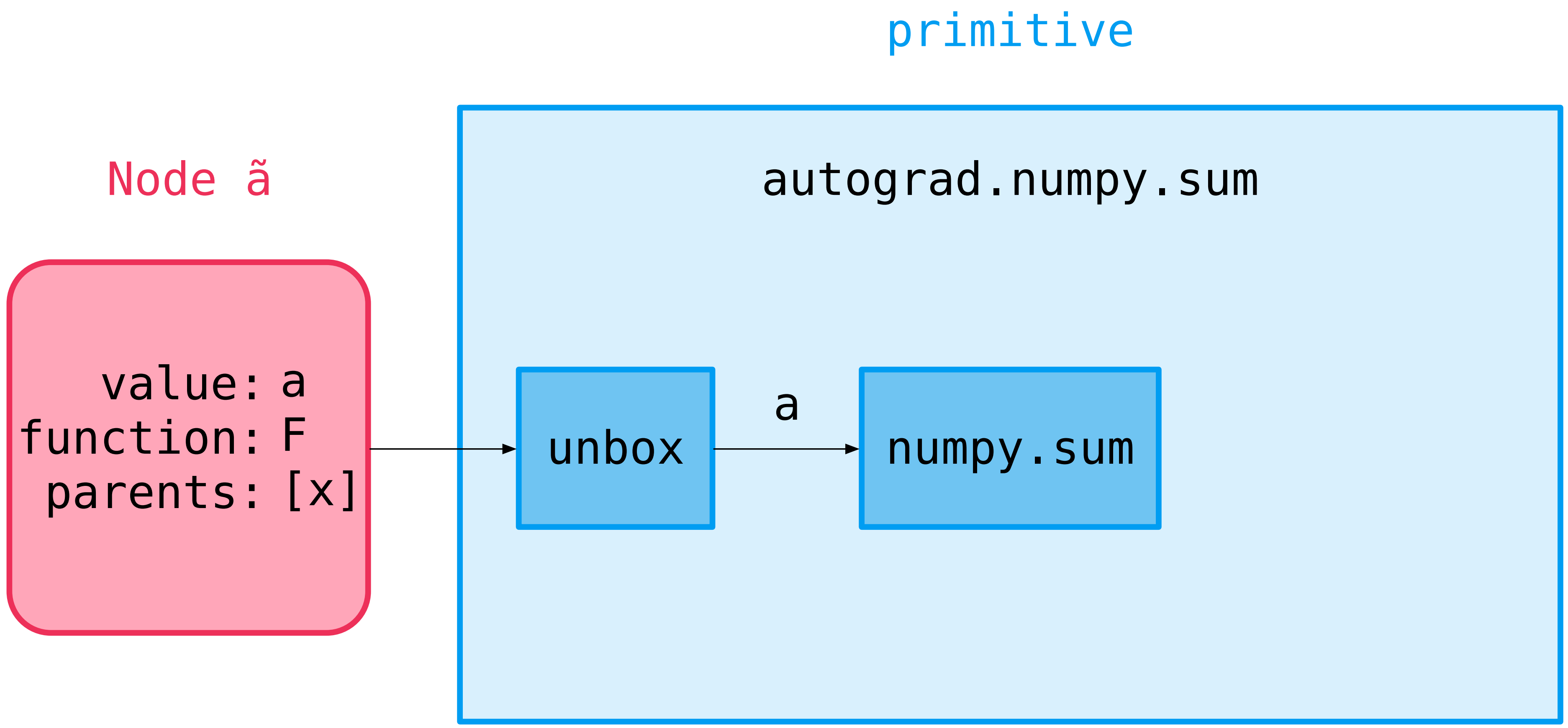
Autograd's ingredients

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

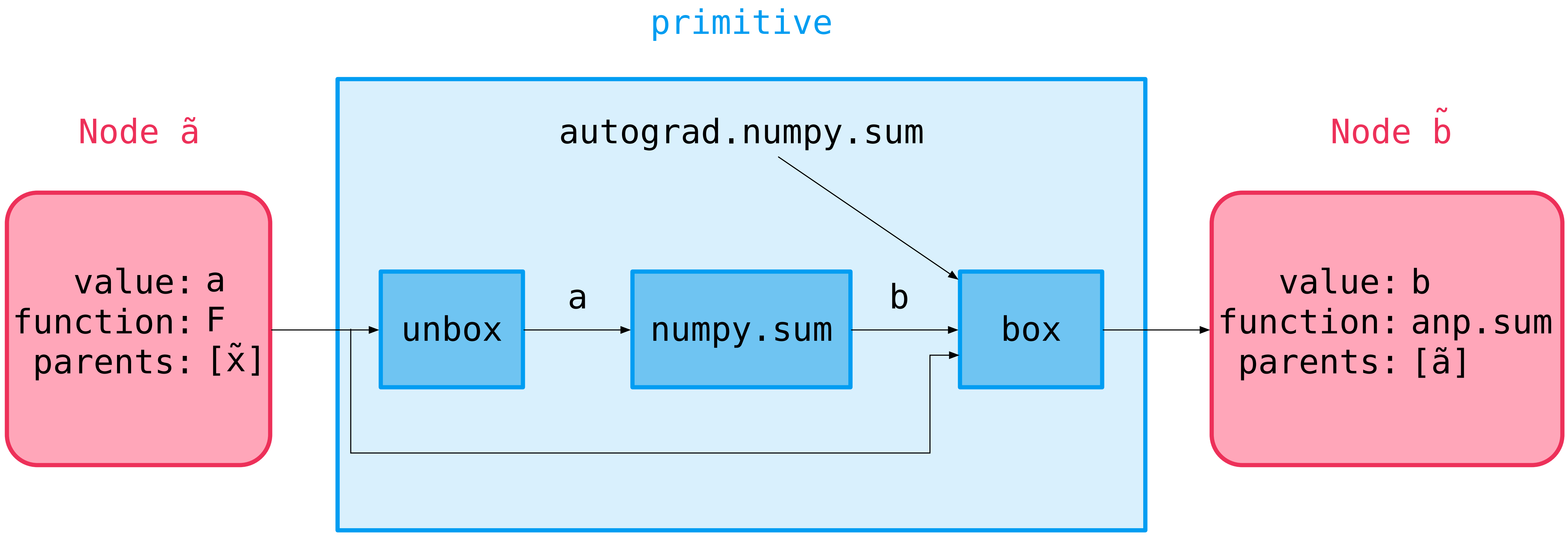
`numpy.sum`


primitive
autograd.numpy.sum
numpy.sum


primitive
Node ã
value: a
function: F
parents: [x]
autograd.numpy.sum
numpy.sum



primitive
Node ã
autograd.numpy.sum
value: a
function: F
parents: [x]
unbox
a
numpy.sum

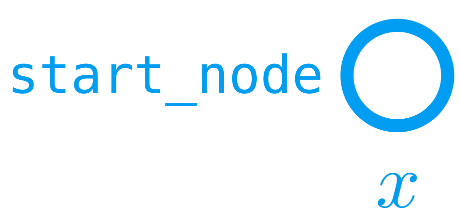

primitive
Node ã
value: a
function: F
parents: [x̃]
autograd.numpy.sum
unbox
a
numpy.sum
b
box
Node b̃
value: b
function: anp.sum
parents: [ã]

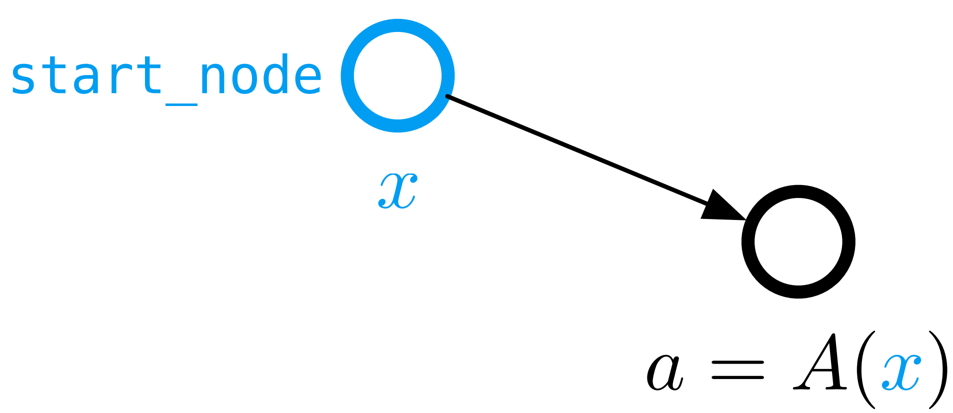
```python
class Node(object):
    __slots__ = ['value', 'recipe', 'progenitors', 'vspace']

    def __init__(self, value, recipe, progenitors):
        self.value = value
        self.recipe = recipe
        self.progenitors = progenitors
        self.vspace = vspace(value)
```

```python
class primitive(object):
    def __call__(self, *args, **kwargs):
        argvals = list(args)

        parents = []
        for argnum, arg in enumerate(args):
            if isnode(arg):
                argvals[argnum] = arg.value
                if argnum in self.zero_vjps: continue
                parents.append((argnum, arg))


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

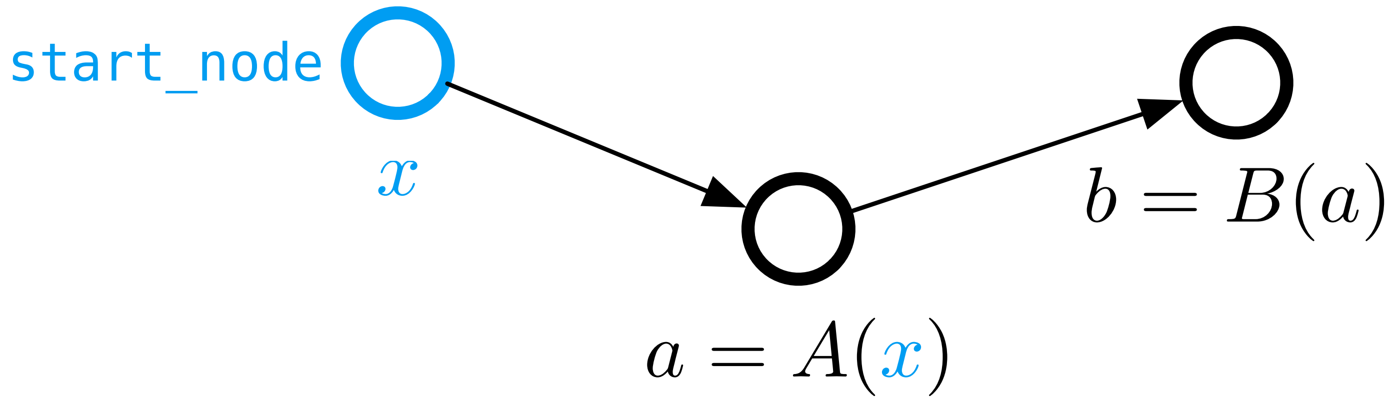
```python
class primitive(object):
    def __call__(self, *args, **kwargs):
        argvals = list(args)
        progenitors = set()
        parents = []
        for argnum, arg in enumerate(args):
            if isnode(arg):
                argvals[argnum] = arg.value
                if argnum in self.zero_vjps: continue
                parents.append((argnum, arg))
                progenitors.update(arg.progenitors & active_progenitors)

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

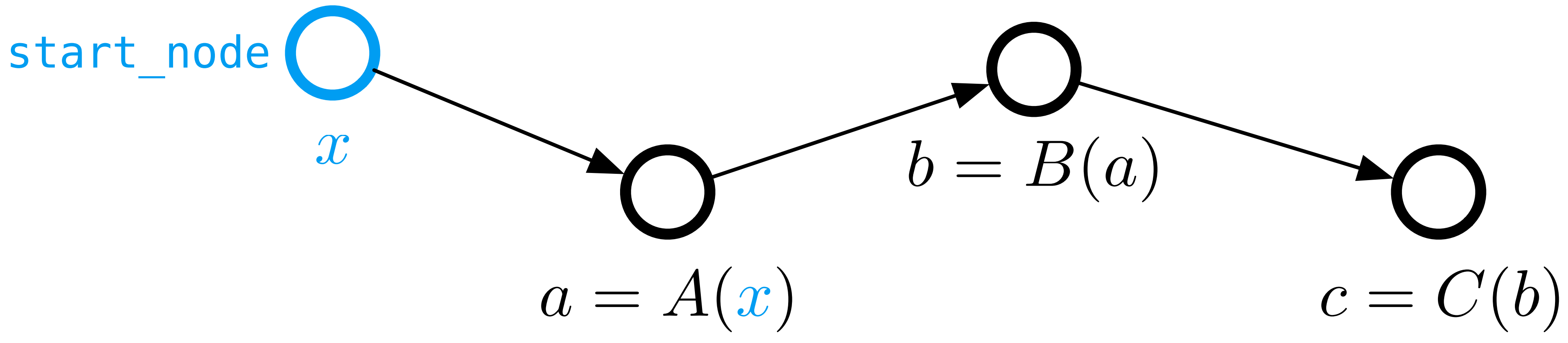
```python
def forward_pass(fun, args, kwargs, argnum=0):
    args = list(args)
    start_node = new_progenitor(args[argnum])
    args[argnum] = start_node
    active_progenitors.add(start_node)
    end_node = fun(*args, **kwargs)
    active_progenitors.remove(start_node)
    return start_node, end_node
```

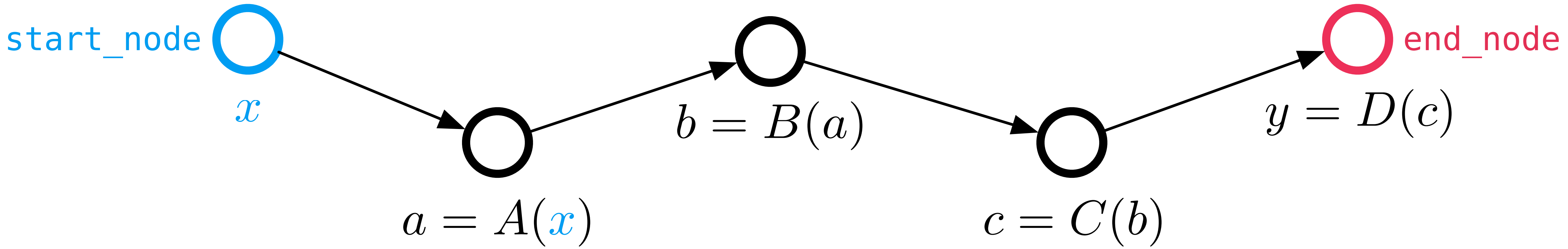

start_node
x

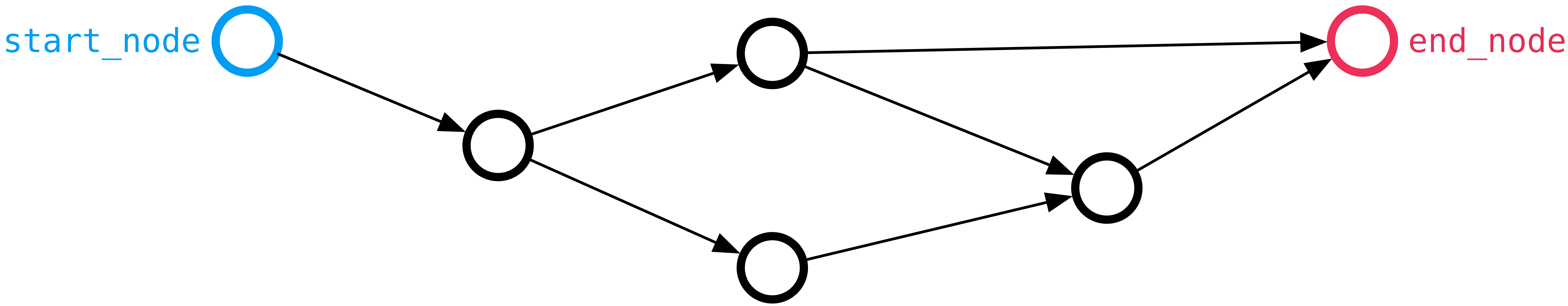


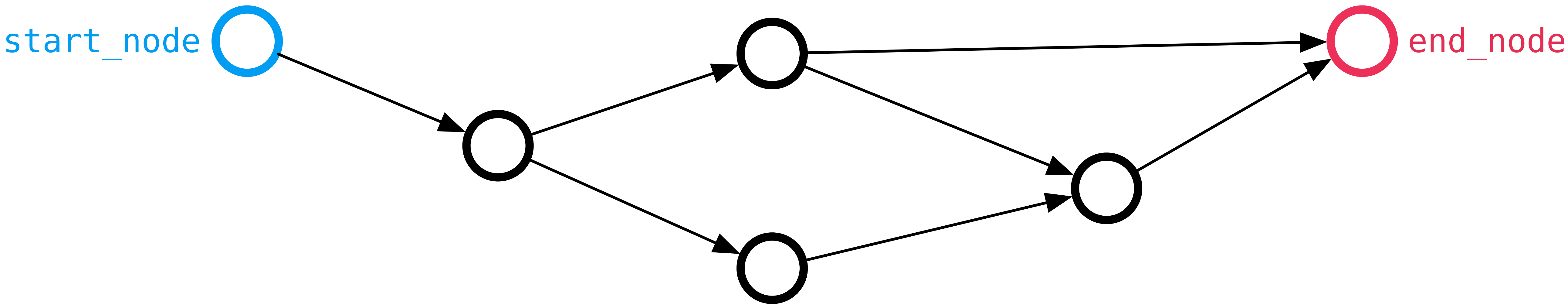
start_node
x
$a = A(x)$


start_node
x
$a = A(x)$
$b = B(a)$


start_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$


start_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
end_node

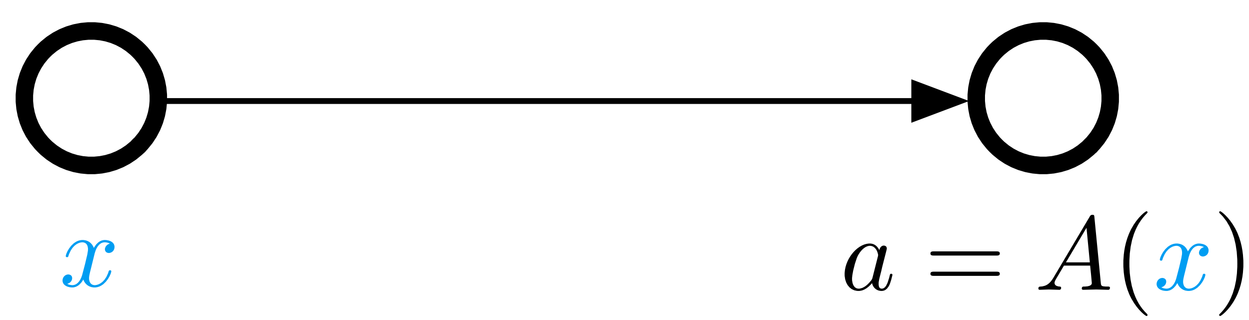

start_node
end_node

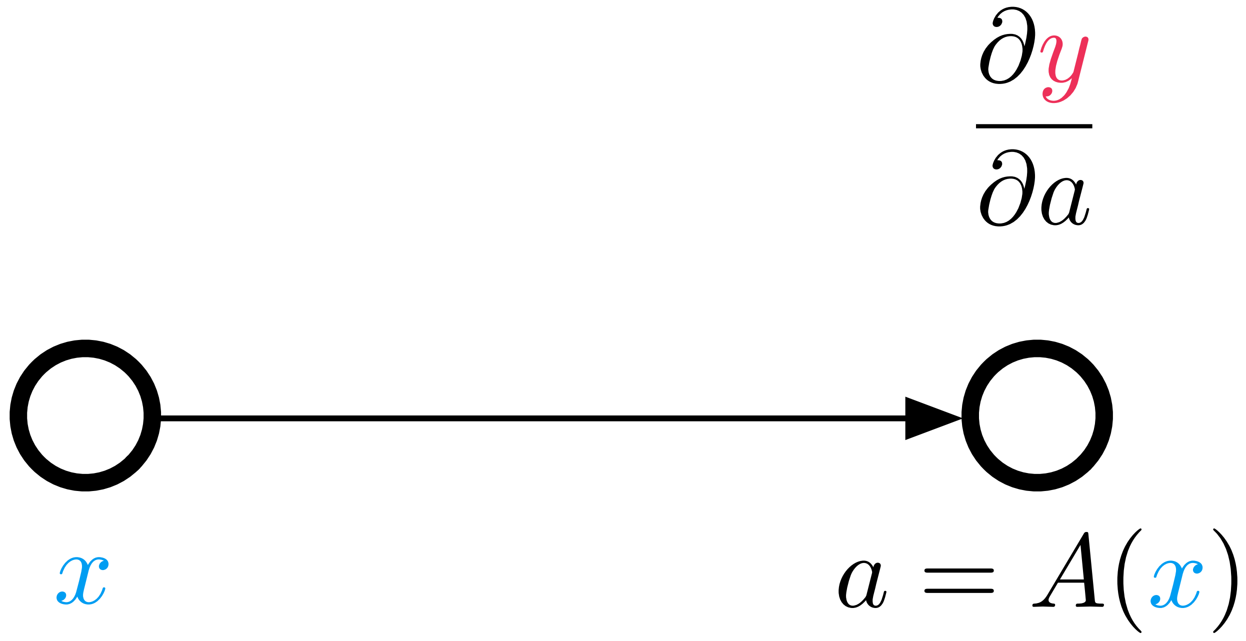


No control flow!

Autograd’s ingredients

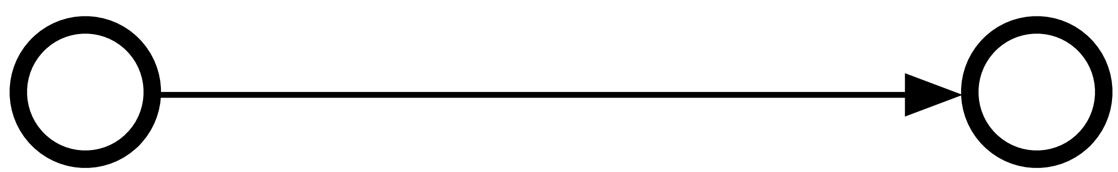
1. Tracing the composition of primitive functions
2. Defining a vector-Jacobian product (VJP) operator for each primitive
3. Composing VJPs backward


x
$a = A(x)$



$\frac{\partial y}{\partial a}$
x
$a = A(x)$

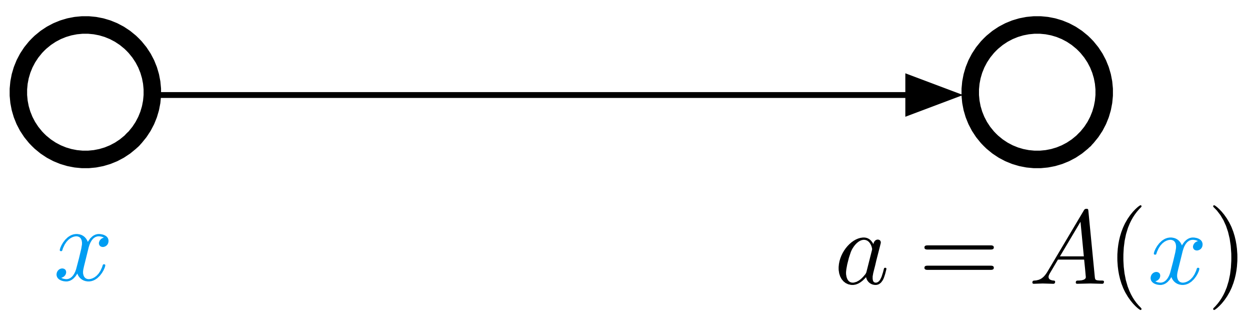
$$\frac{\partial y}{\partial x} = ? \qquad\qquad \frac{\partial y}{\partial a}$$



$$x \qquad\qquad a = A(x)$$

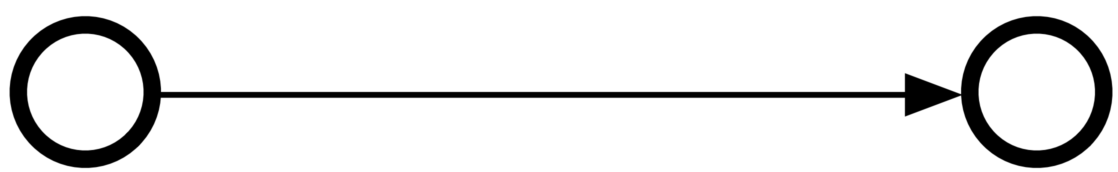
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial a} \cdot \frac{\partial a}{\partial x}$$

$$\frac{\partial y}{\partial a}$$



$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial a} \cdot A'(x)$$

$$\frac{\partial y}{\partial a}$$

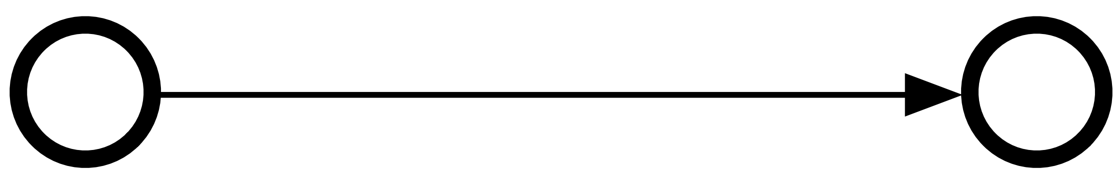


x

$a = A(x)$

vector-Jacobian product

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial a} \boxed{\cdot A'(x)} \qquad \frac{\partial y}{\partial a}$$

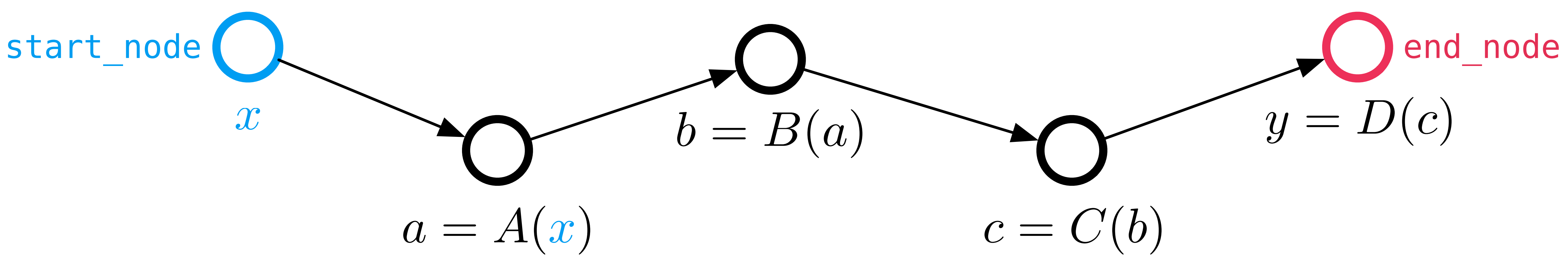


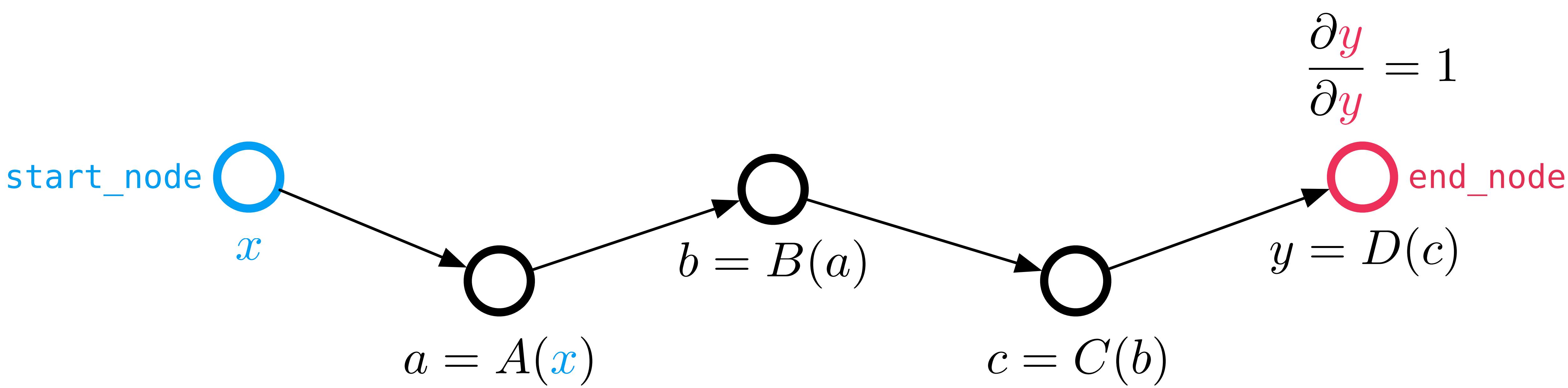
x $\qquad$ $a = A(x)$

```python
anp.sinh.defvjp(lambda g, ans, vs, gvs, x: g * anp.cosh(x))
anp.cosh.defvjp(lambda g, ans, vs, gvs, x: g * anp.sinh(x))
anp.tanh.defvjp(lambda g, ans, vs, gvs, x: g / anp.cosh(x)**2)

anp.cross.defvjp(lambda g, ans, vs, gvs, a, b, axisa=-1, axisb=-1, axisc=-1, axis=None:
                 anp.cross(b, g, axisb, axisc, axisa, axis), argnum=0)

def grad_sort(g, ans, vs, gvs, x, axis=-1, kind='quicksort', order=None):
    sort_perm = anp.argsort(x, axis, kind, order)
    return unpermuter(g, sort_perm)
anp.sort.defvjp(grad_sort)
```

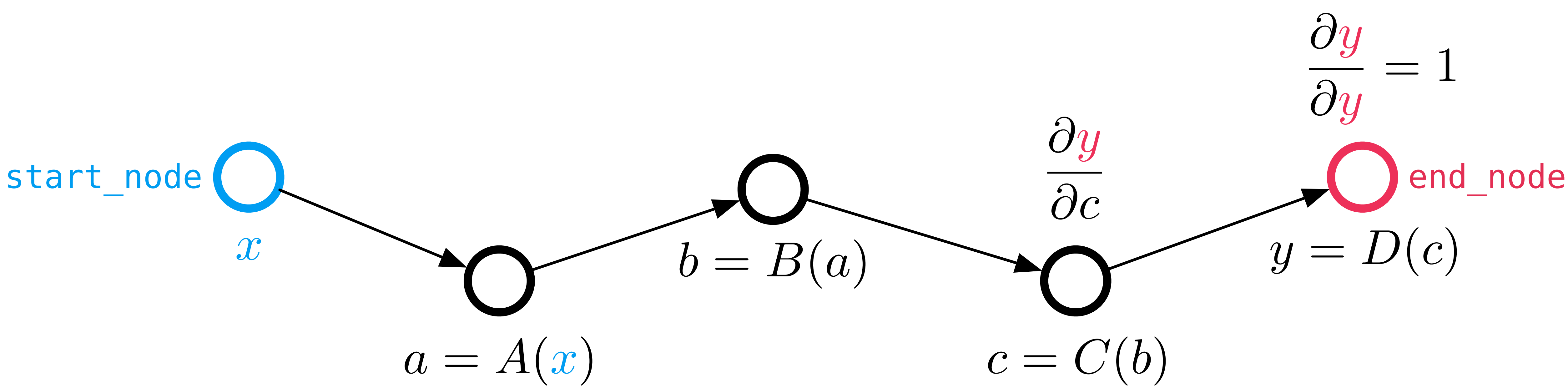
Autograd's ingredients

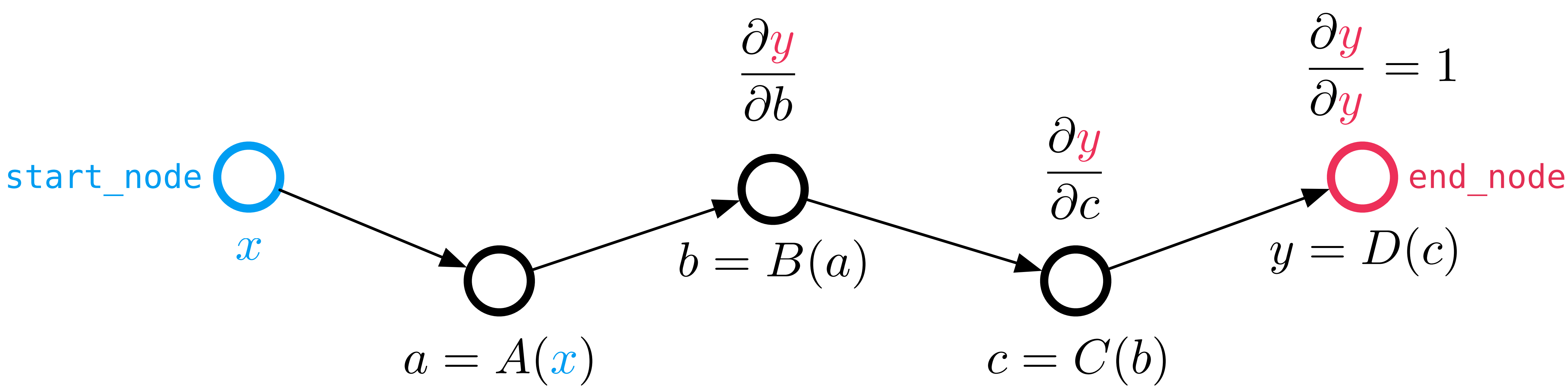
1. Tracing the composition of primitive functions
2. Defining a vector-Jacobian product (VJP) operator for each primitive
3. Composing VJPs backward


start_node
x
a = A(x)
b = B(a)
c = C(b)
y = D(c)
end_node

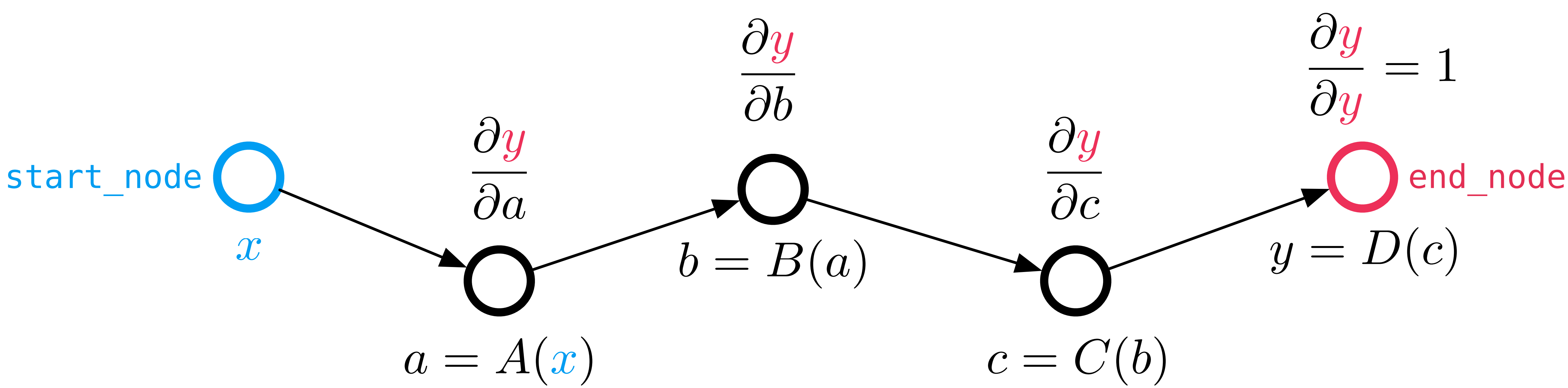


$\frac{\partial y}{\partial y} = 1$
start_node
end_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$

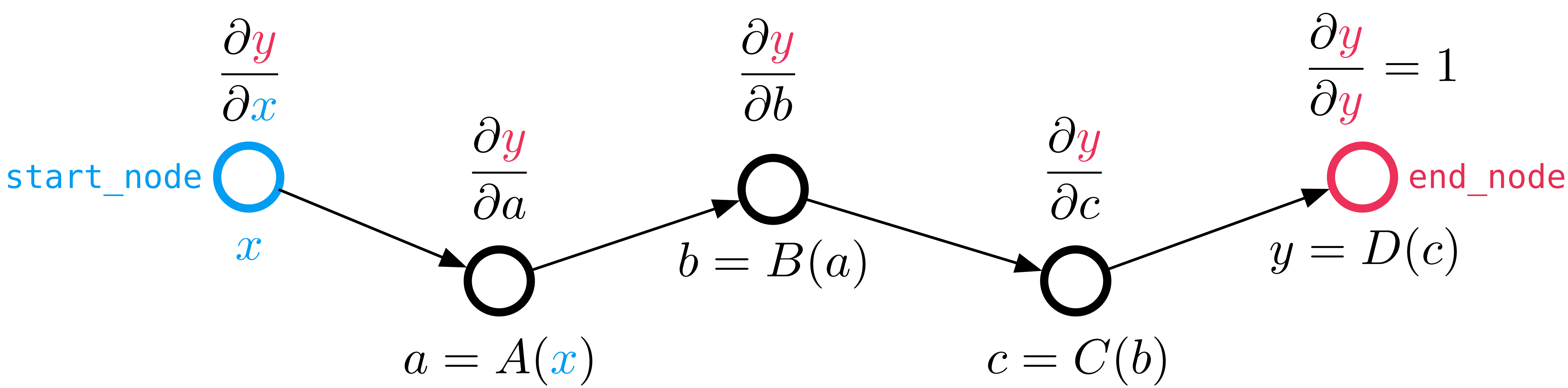

start_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$\frac{\partial y}{\partial c}$
$y = D(c)$
$\frac{\partial y}{\partial y} = 1$
end_node



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



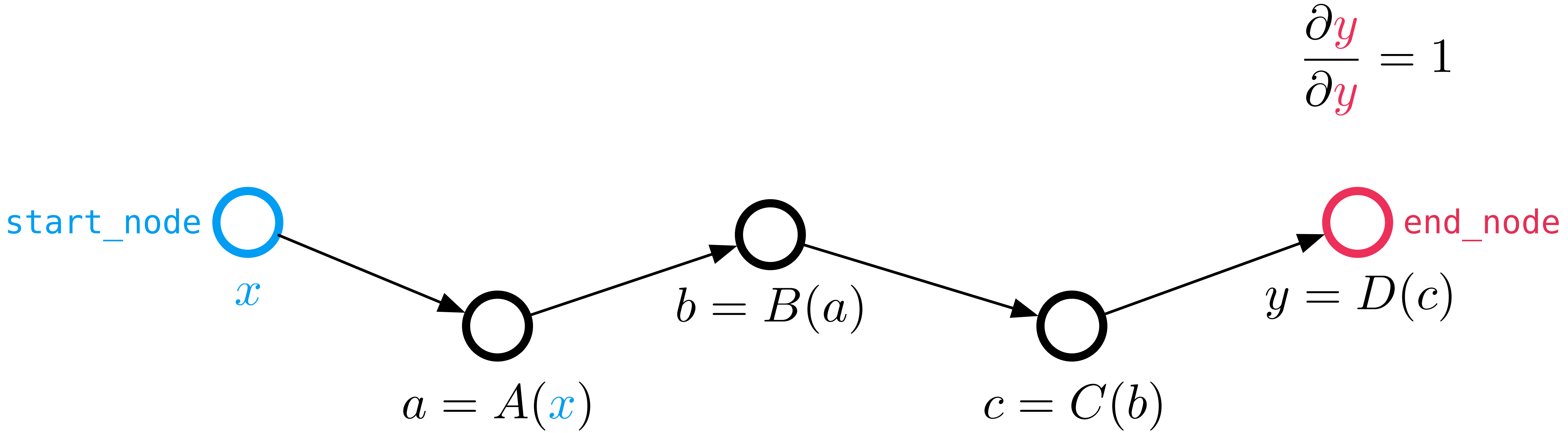
start_node
x
$\frac{\partial y}{\partial a}$
$a = A(x)$
$\frac{\partial y}{\partial b}$
$b = B(a)$
$\frac{\partial y}{\partial c}$
$c = C(b)$
$\frac{\partial y}{\partial y} = 1$
$y = D(c)$
end_node

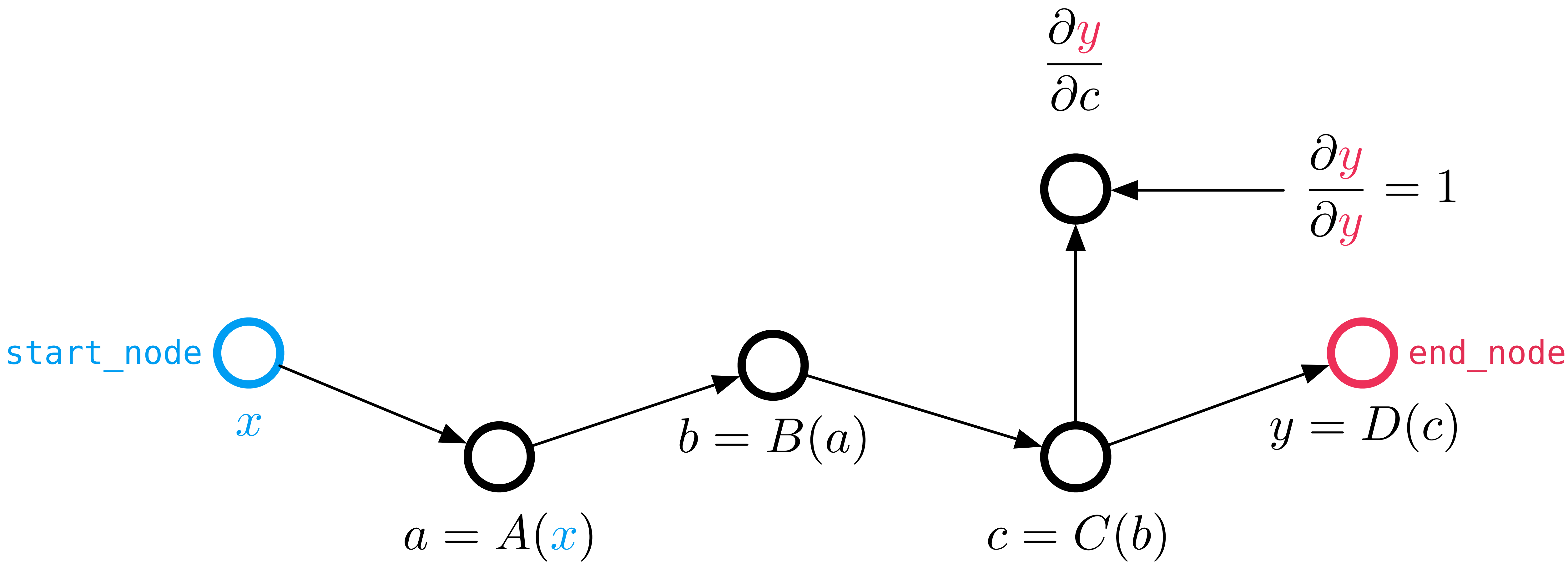


start_node
end_node
$\frac{\partial y}{\partial x}$
x
$\frac{\partial y}{\partial a}$
$a = A(x)$
$\frac{\partial y}{\partial b}$
$b = B(a)$
$\frac{\partial y}{\partial c}$
$c = C(b)$
$\frac{\partial y}{\partial y} = 1$
$y = D(c)$

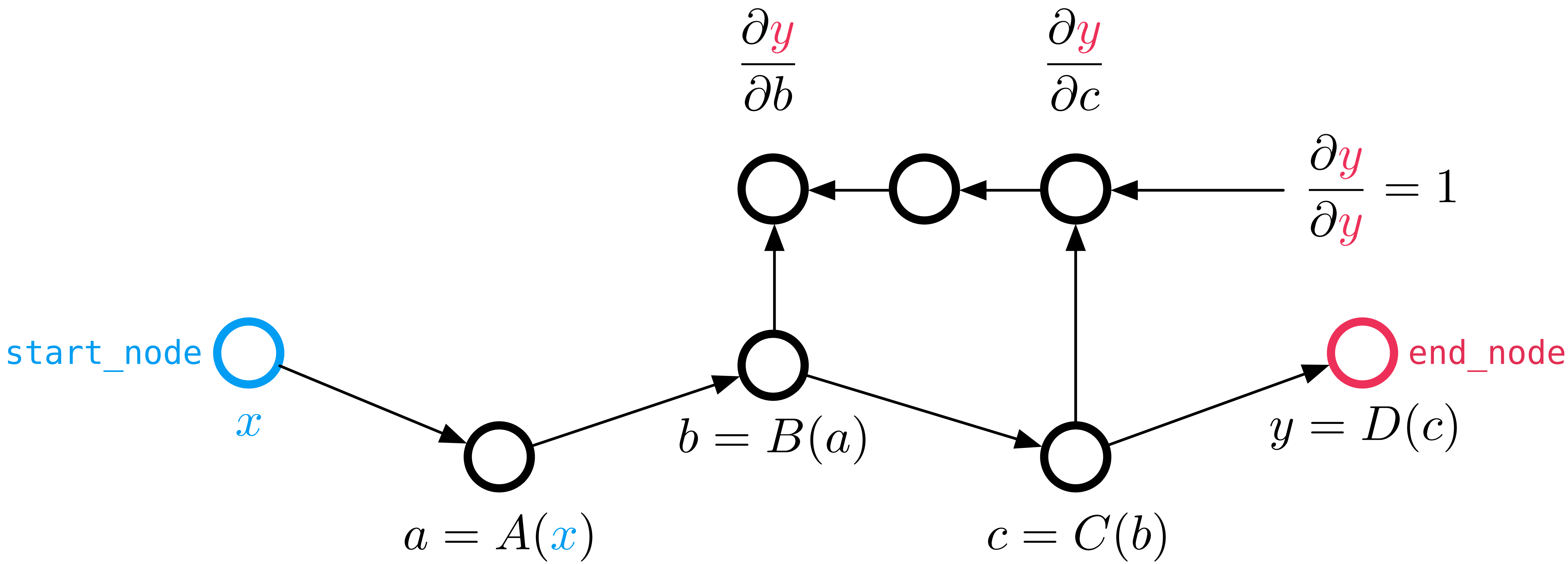
higher-order autodiff just works:

the backward pass can itself be traced

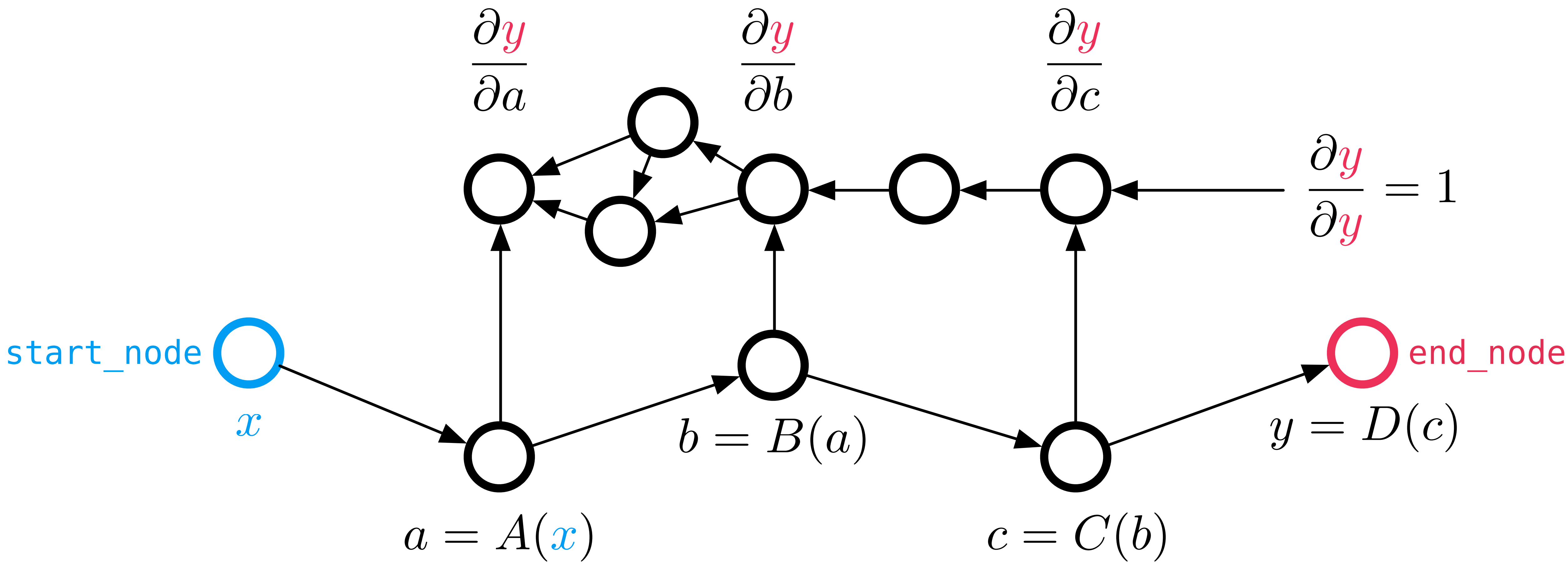

start_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
end_node
$\frac{\partial y}{\partial y} = 1$



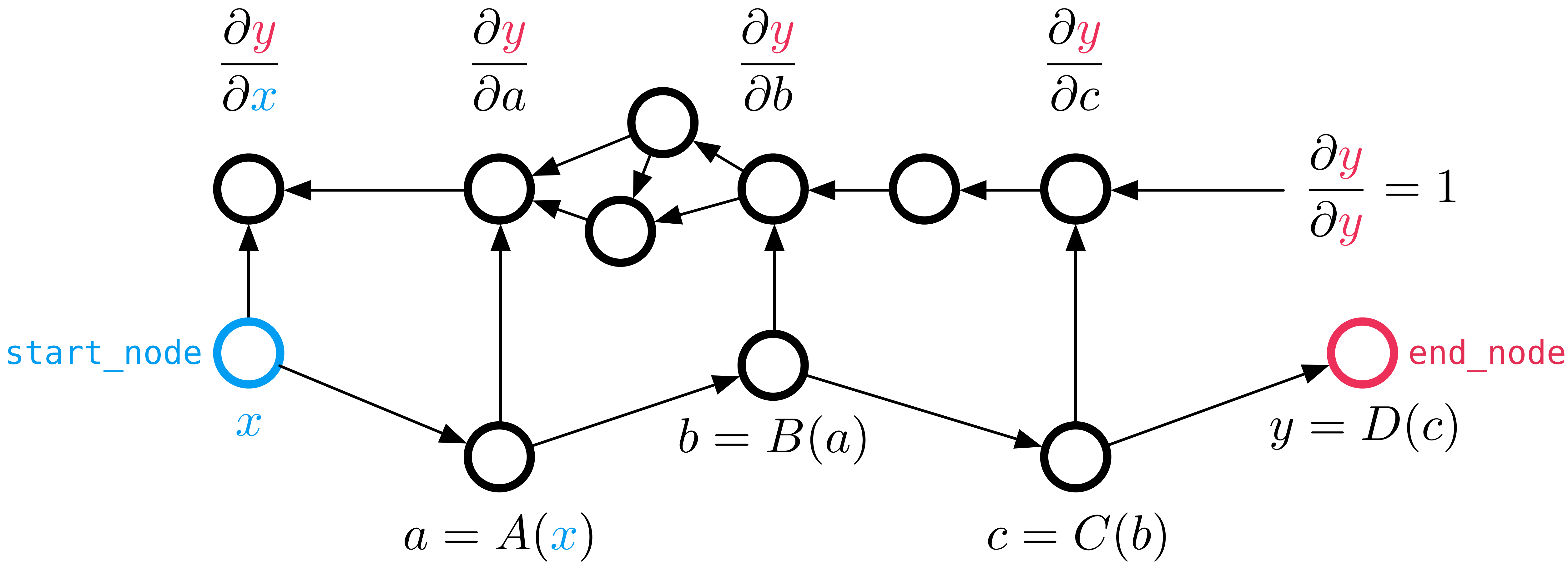
start_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
end_node
$\frac{\partial y}{\partial c}$
$\frac{\partial y}{\partial y} = 1$



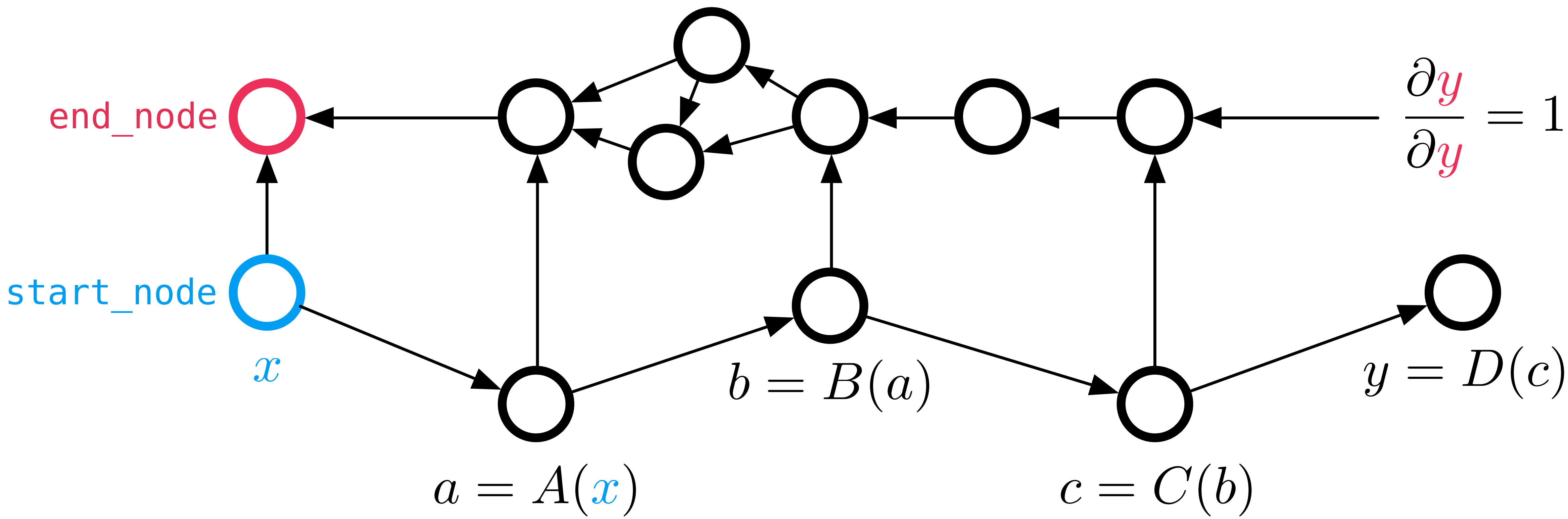
start_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
end_node
$\frac{\partial y}{\partial b}$
$\frac{\partial y}{\partial c}$
$\frac{\partial y}{\partial y} = 1$



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



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



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

```python
def backward_pass(g, end_node, start_node):
    outgrads = {end_node : (g, False)}
    assert_vspace_match(outgrads[end_node][0], end_node.vspace, None)
    for node in toposort(end_node, start_node):
        if node not in outgrads: continue
        cur_outgrad = outgrads.pop(node)
        function, args, kwargs, parents = node.recipe
        for argnum, parent in parents:
            outgrad = function.vjp(argnum, cur_outgrad[0], node,
                                   parent.vspace, node.vspace, args, kwargs)
            assert_vspace_match(outgrad, parent.vspace, function)
            outgrads[parent] = add_outgrads(parent.vspace, outgrads.get(parent),
                                            outgrad)
    return cur_outgrad[0]
```

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

```python
def make_vjp(fun, argnum=0):
    def vjp_maker(*args, **kwargs):
        start_node, end_node = forward_pass(fun, args, kwargs, argnum)
        if not isnode(end_node) or start_node not in end_node.progenitors:
            warnings.warn("Output seems independent of input.")
            def vjp(g): return start_node.vspace.zeros()
        else:
            def vjp(g): return backward_pass(g, end_node, start_node)
        return vjp, end_node
    return vjp_maker
```

Autograd’s ingredients

1. Tracing the composition of primitive functions

   `Node`, `primitive`, `forward_pass`

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

   `defvjp`

3. Composing VJPs backward

   `backward_pass`, `make_vjp`, `grad`

Tradeoffs in forward vs reverse

Tradeoffs in forward vs reverse

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

Tradeoffs in forward vs reverse

- Reverse-mode requires tracing a program's execution
  - Memory cost scales like depth of program
  - Checkpointing can trade off time and memory
- Forward-mode evaluates a JVP with constant memory overhead
  - But requires n calls to form Jacobian of $F : \mathbb{R}^n \to \mathbb{R}$
  - Autograd forward-mode by @j-towns: github.com/BB-UCL/autograd-forward

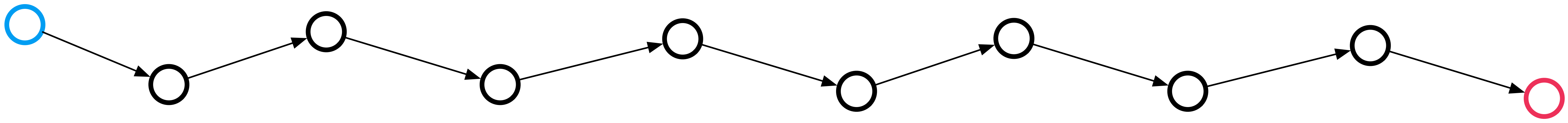
Tradeoffs in forward vs reverse

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

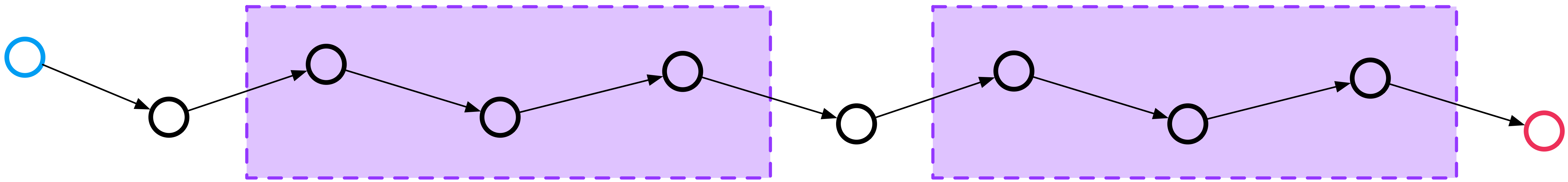
Tutorial goals

1. Jacobians and the chain rule
    - Forward and reverse accumulation
2. Autograd's implementation
    - Fully closed tracing autodiff in Python
3. Advanced autodiff techniques
    - Checkpointing, forward from reverse, differentiating optima and fixed points

Checkpointing

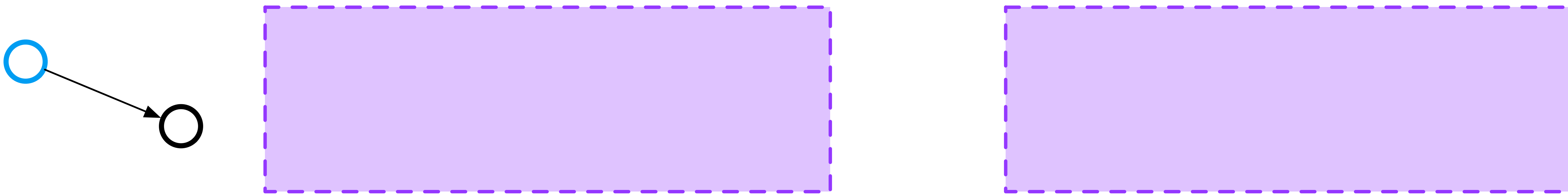


Checkpointing

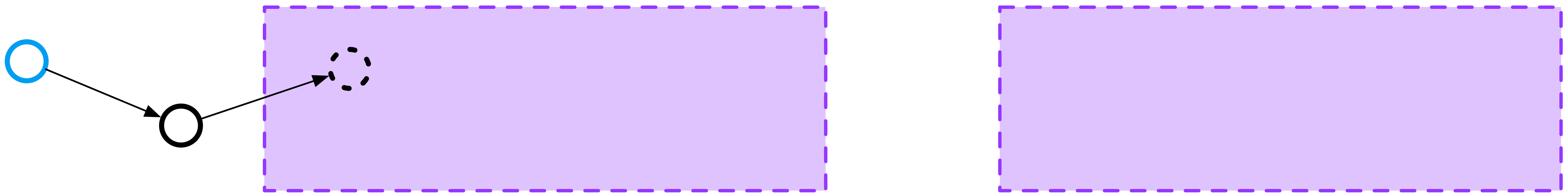


Checkpointing

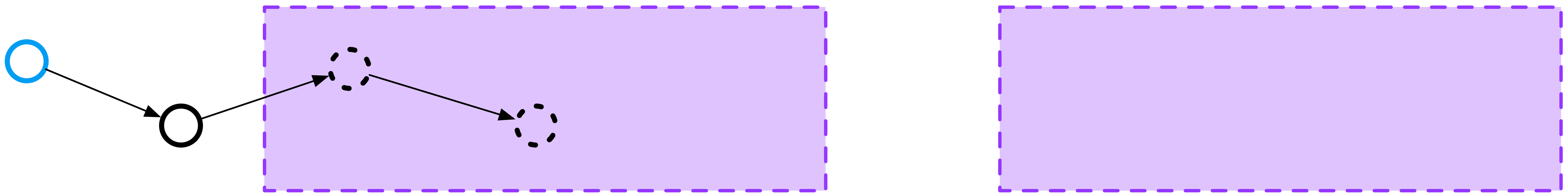
Checkpointing



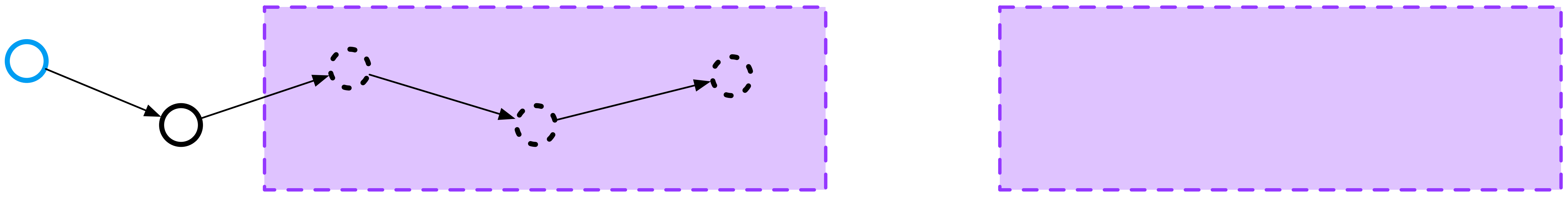
Checkpointing



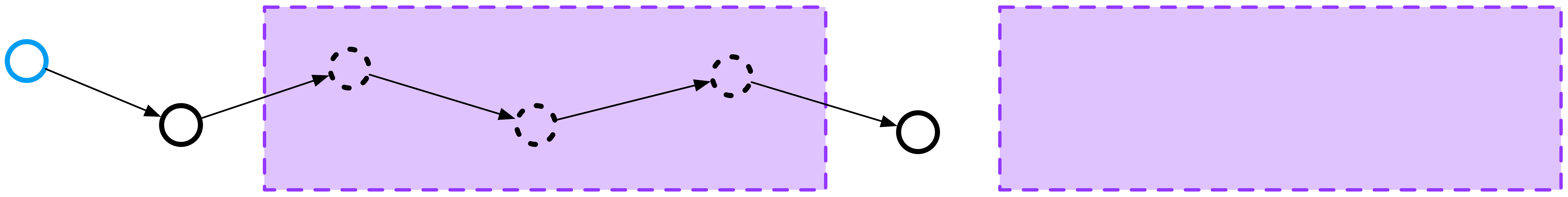
Checkpointing



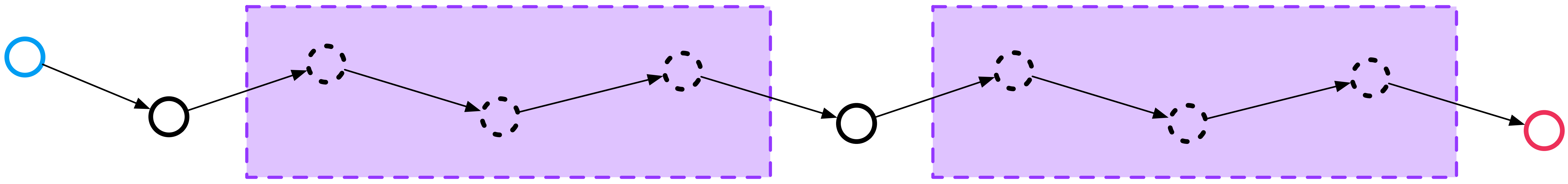
Checkpointing



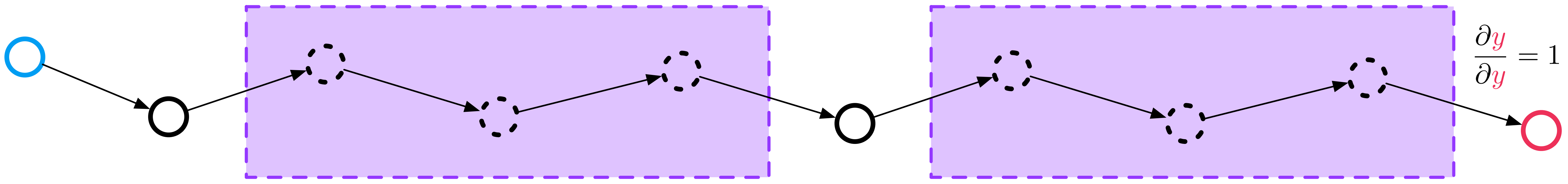
Checkpointing



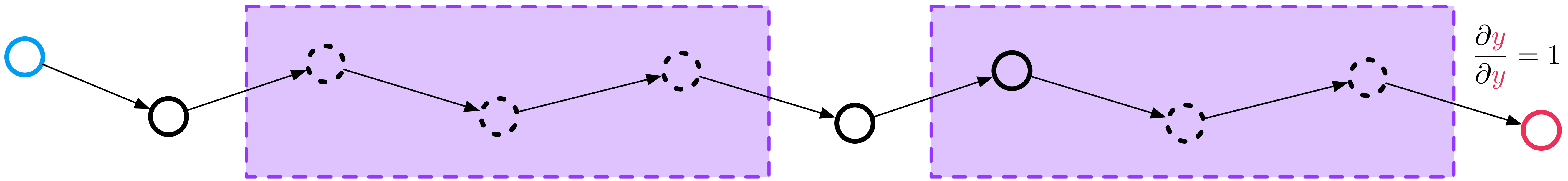
Checkpointing



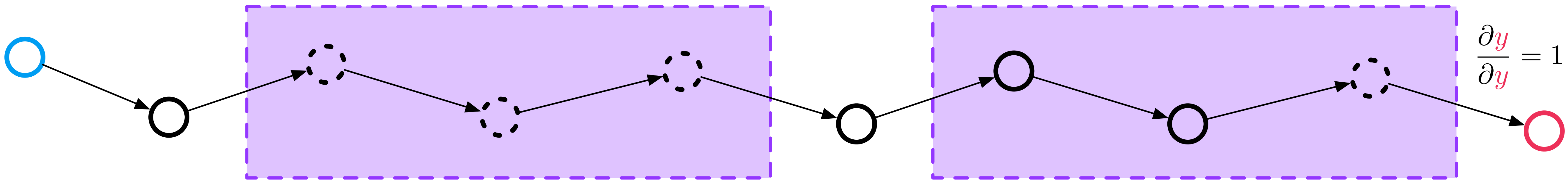
Checkpointing



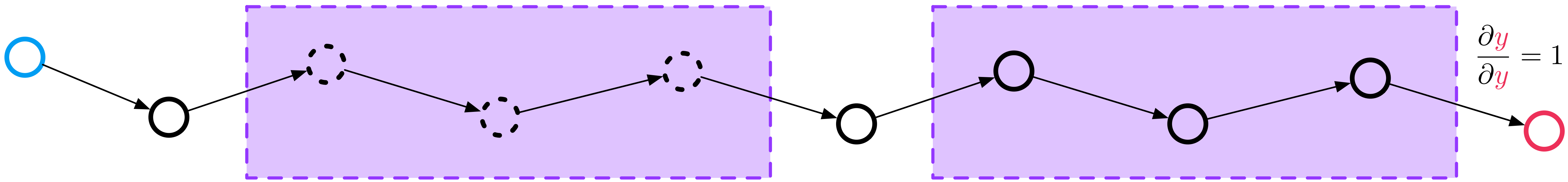
Checkpointing



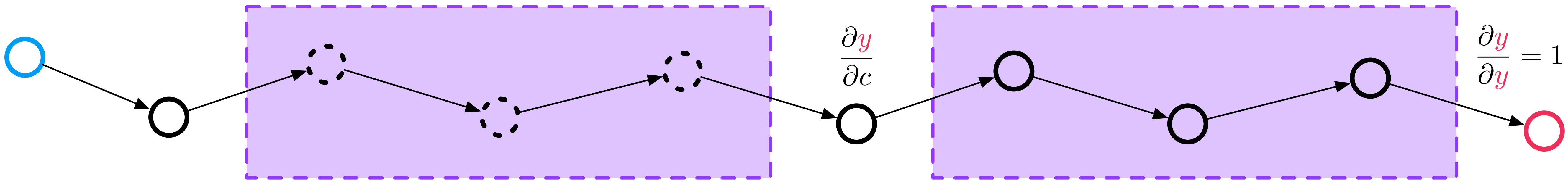
Checkpointing



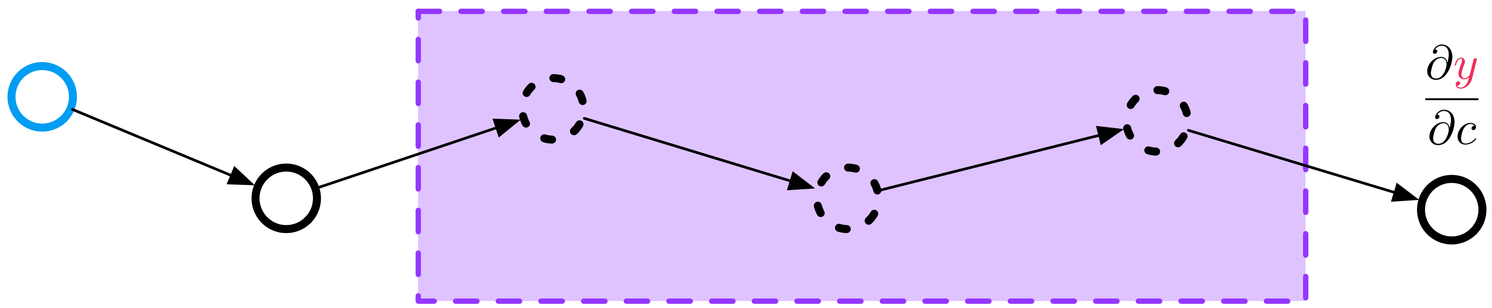
Checkpointing



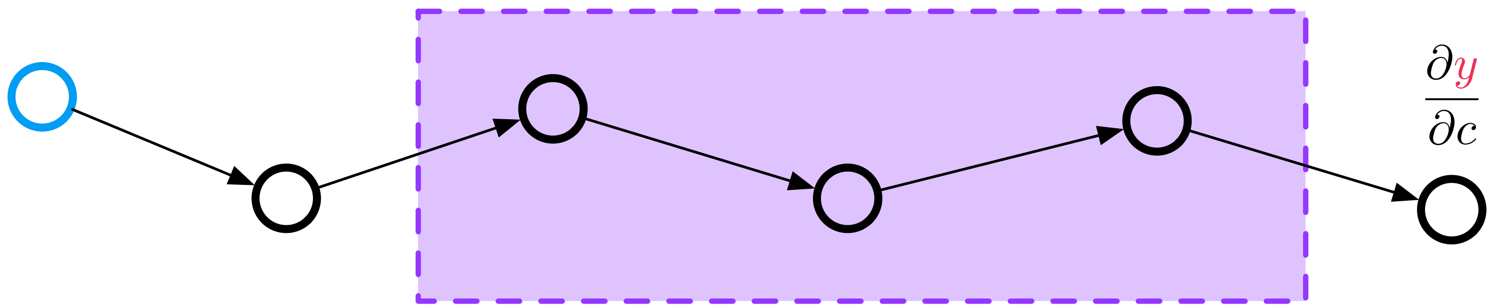
Checkpointing



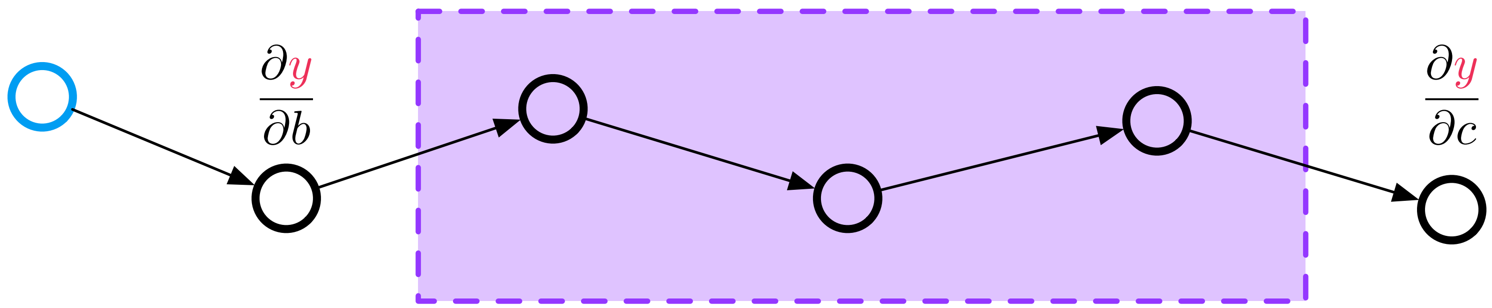
Checkpointing



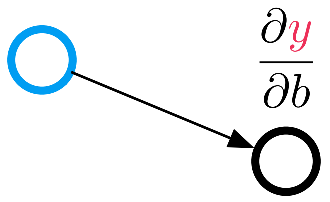
Checkpointing



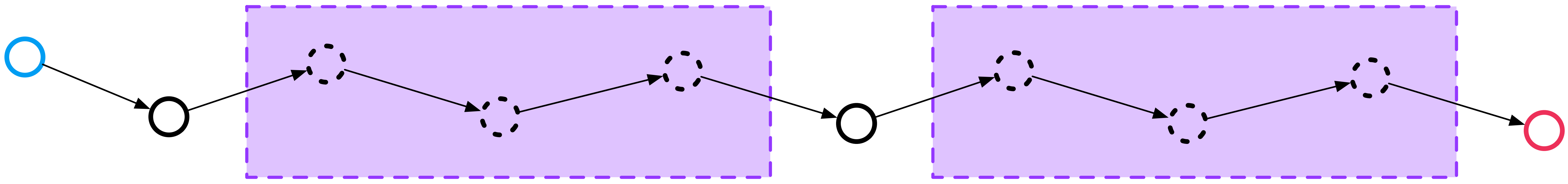
Checkpointing



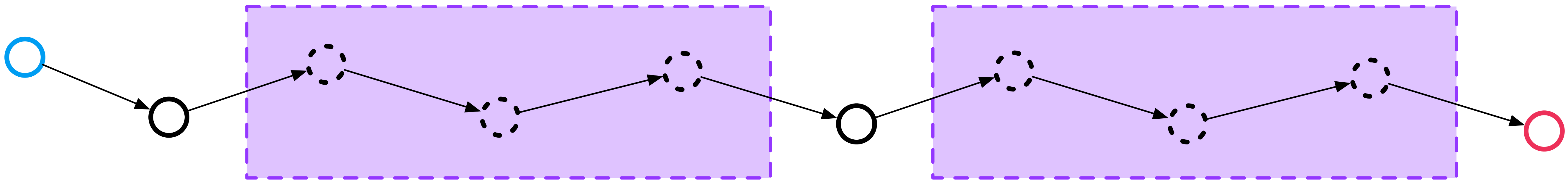
Checkpointing



Checkpointing



Checkpointing



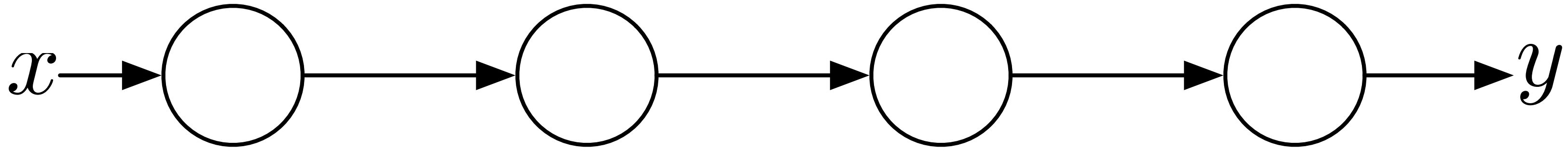
```
def checkpoint(fun):
    """Returns a checkpointed version of ‘fun‘, where intermediate values
    computed during the forward pass of ‘fun‘ are discarded and then recomputed
    for the backward pass. Useful to trade off time and memory."""
    def wrapped_grad(argnum, g, ans, vs, gvs, args, kwargs):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

Getting forward from reverse

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

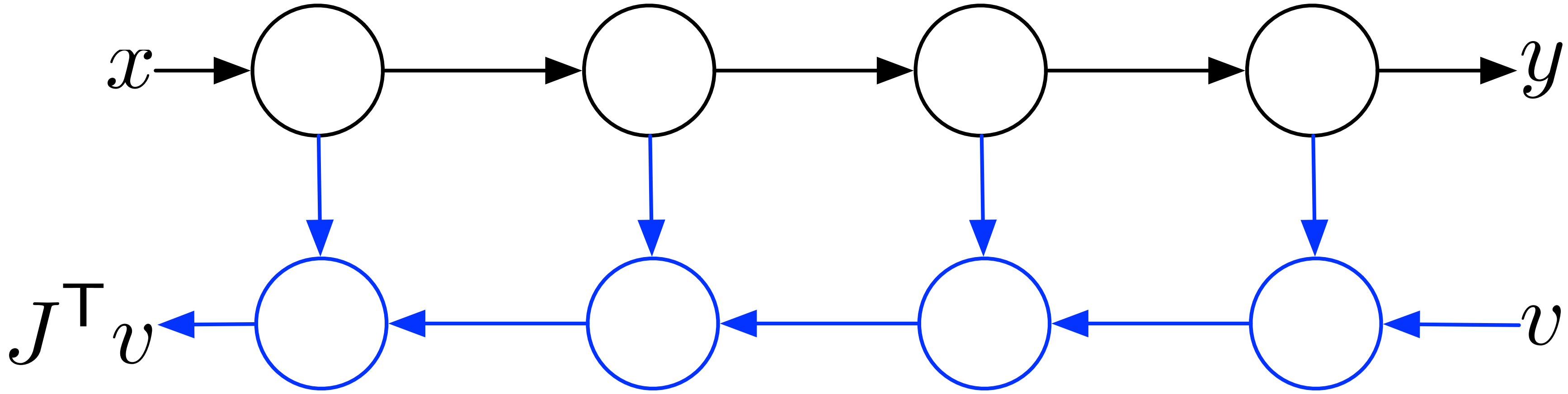
Getting forward from reverse

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



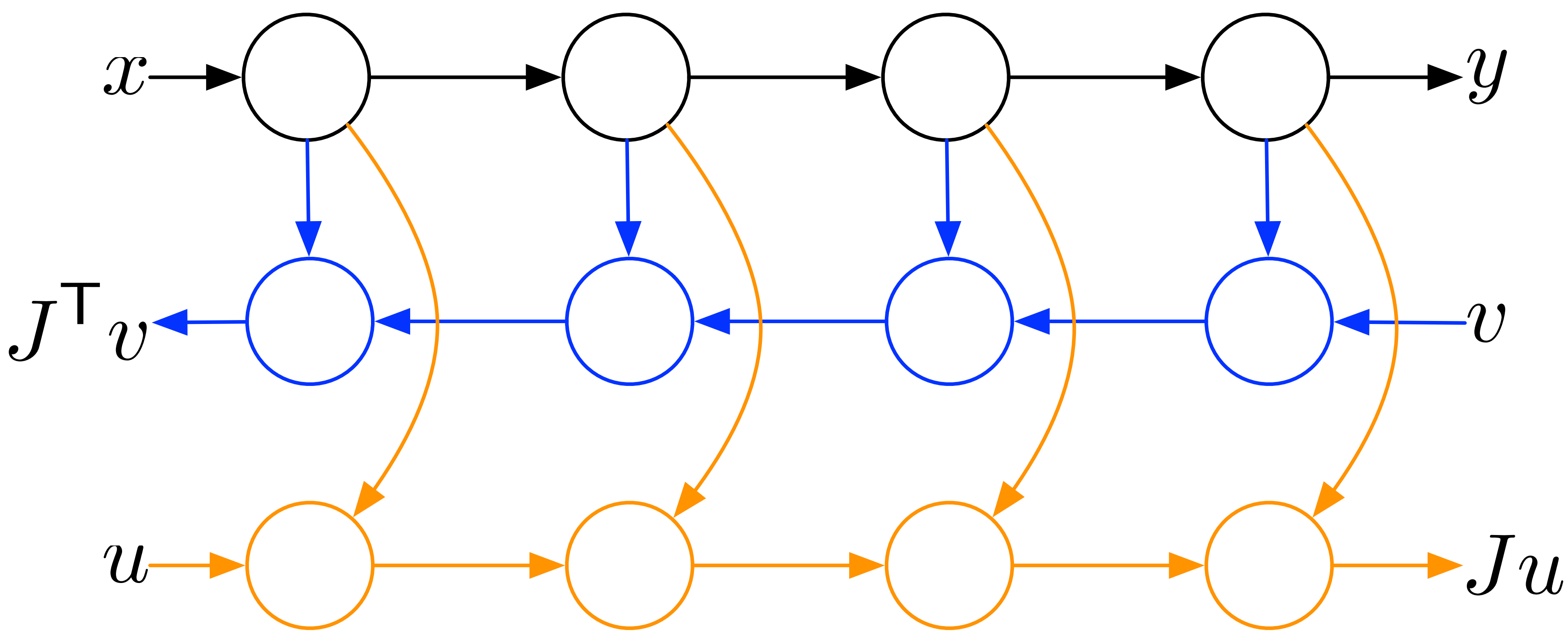
Getting forward from reverse

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



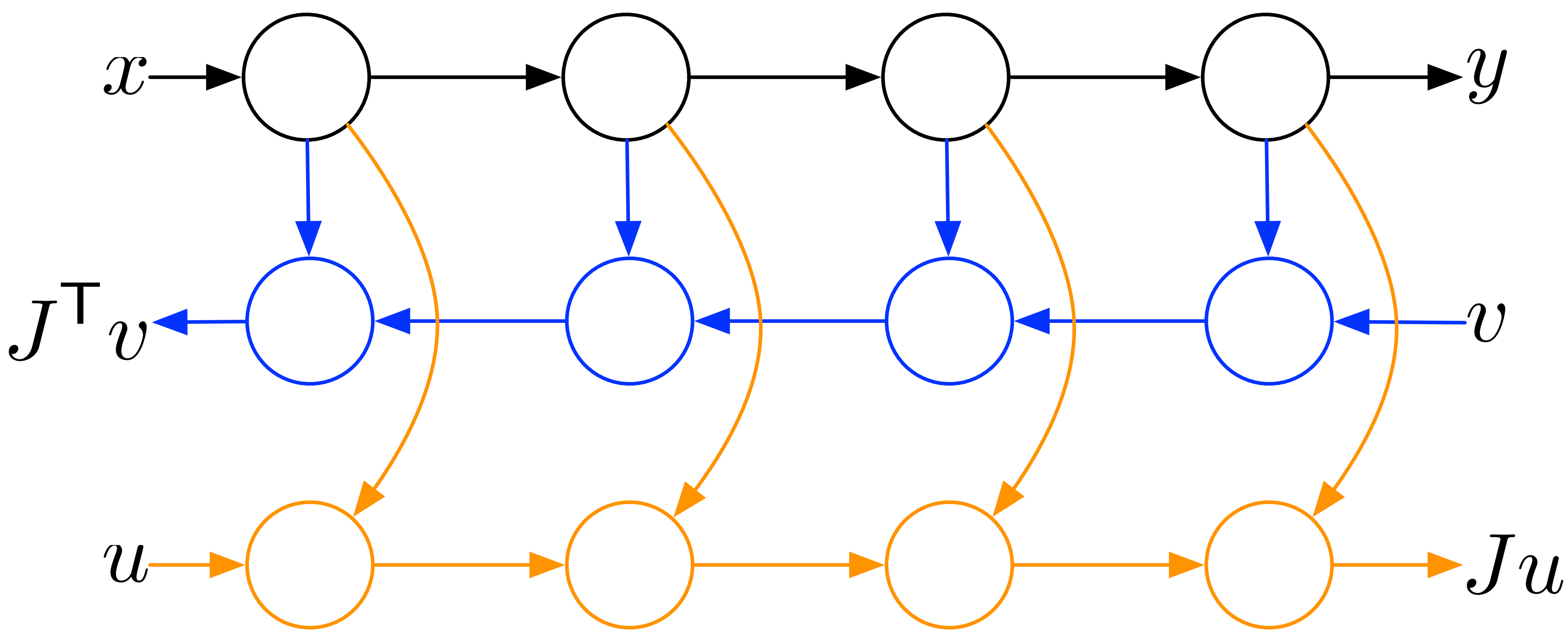
Getting forward from reverse

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



Getting forward from reverse

```
import tensorflow as tf

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



Solutions, optima, and fixed points

Solutions, optima, and fixed points

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

$$\nabla x^*(a) = ?$$

Solutions, optima, and fixed points

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

$$\nabla x^*(a) = \,?$$

$$\text{solve} \;\; g(a, x) = 0 \;\; \text{for } x$$

$$g(a, x^*(a)) = 0$$

$$\nabla x^*(a) = \,?$$

The implicit function theorem

$$g(a, x^*(a)) = 0$$

The implicit function theorem

$$g(a, x^*(a)) = 0$$

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

The implicit function theorem

$$g(a, x^*(a)) = 0$$

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

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

The implicit function theorem

$$g(a, x^*(a)) = 0$$

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

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

differentiate solutions / optima $\leftrightarrow$ solve linearized systems

The implicit function theorem

$$g(a, x^*(a)) = 0$$

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

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

differentiate solutions / optima $\leftrightarrow$ solve linearized systems

automatically generate a linear solver from the forward solver?

Differentiating fixed points

Differentiating fixed points

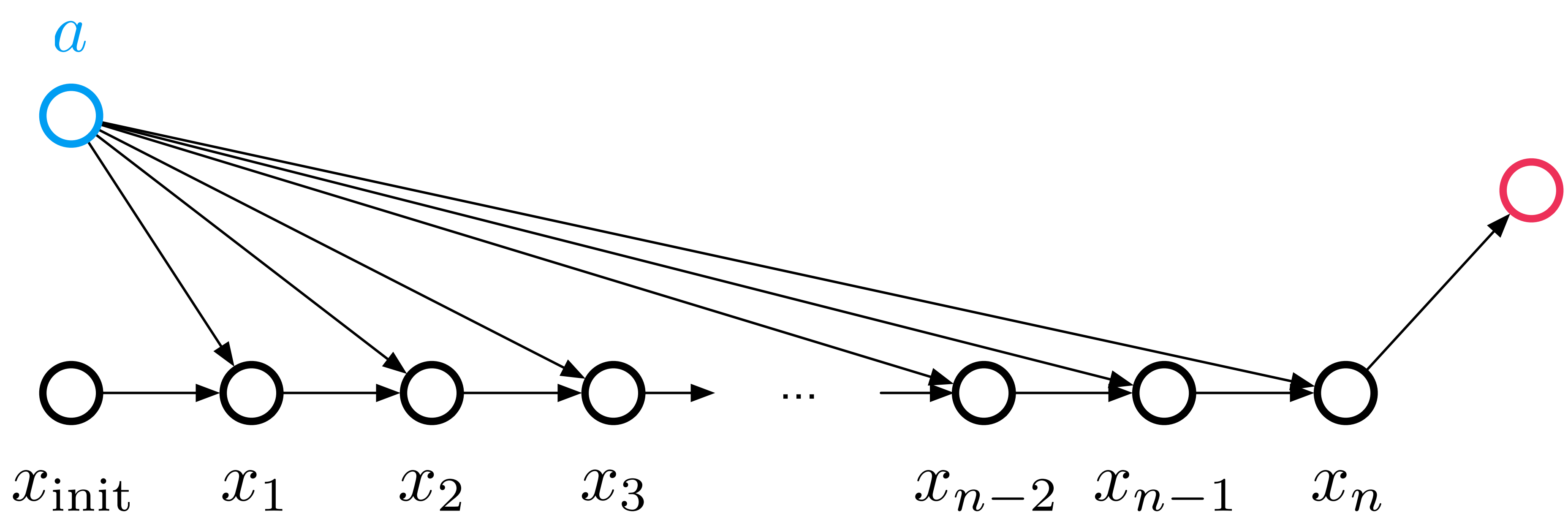
$$x^*(a) \;\; \text{solves} \;\; x = f(a, x) \;\; \text{for} \;\; x$$

Differentiating fixed points

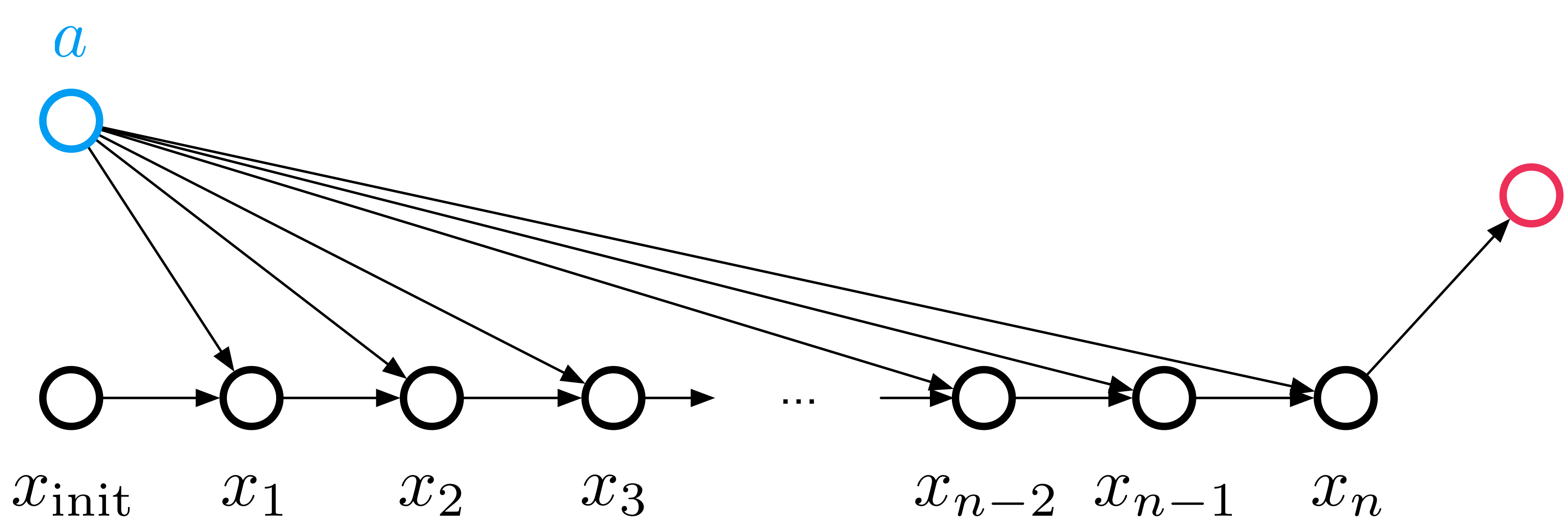
$$x^*(a) \text{ solves } x = f(a, x) \text{ for } x$$

```
from autograd import primitive
from functools import partial

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

Differentiating fixed points



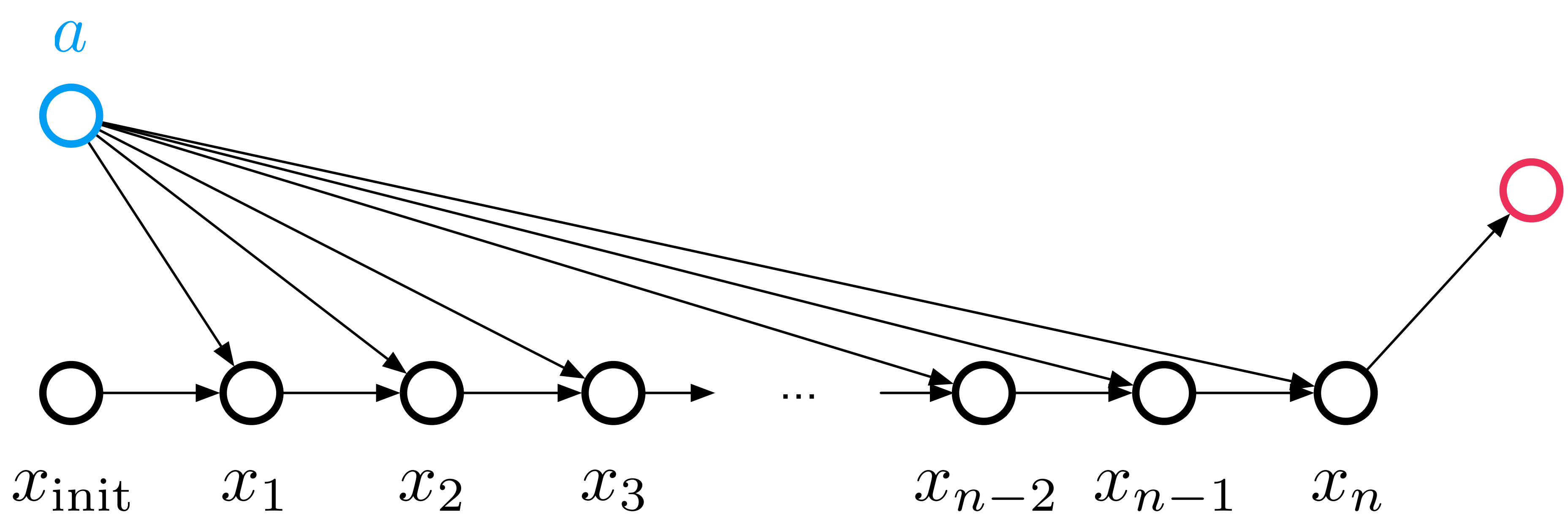
Differentiating fixed points



$$n \to \infty$$

$$x^* = x_n = x_{n-1} = x_{n-2} = \cdots$$

Differentiating fixed points




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

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

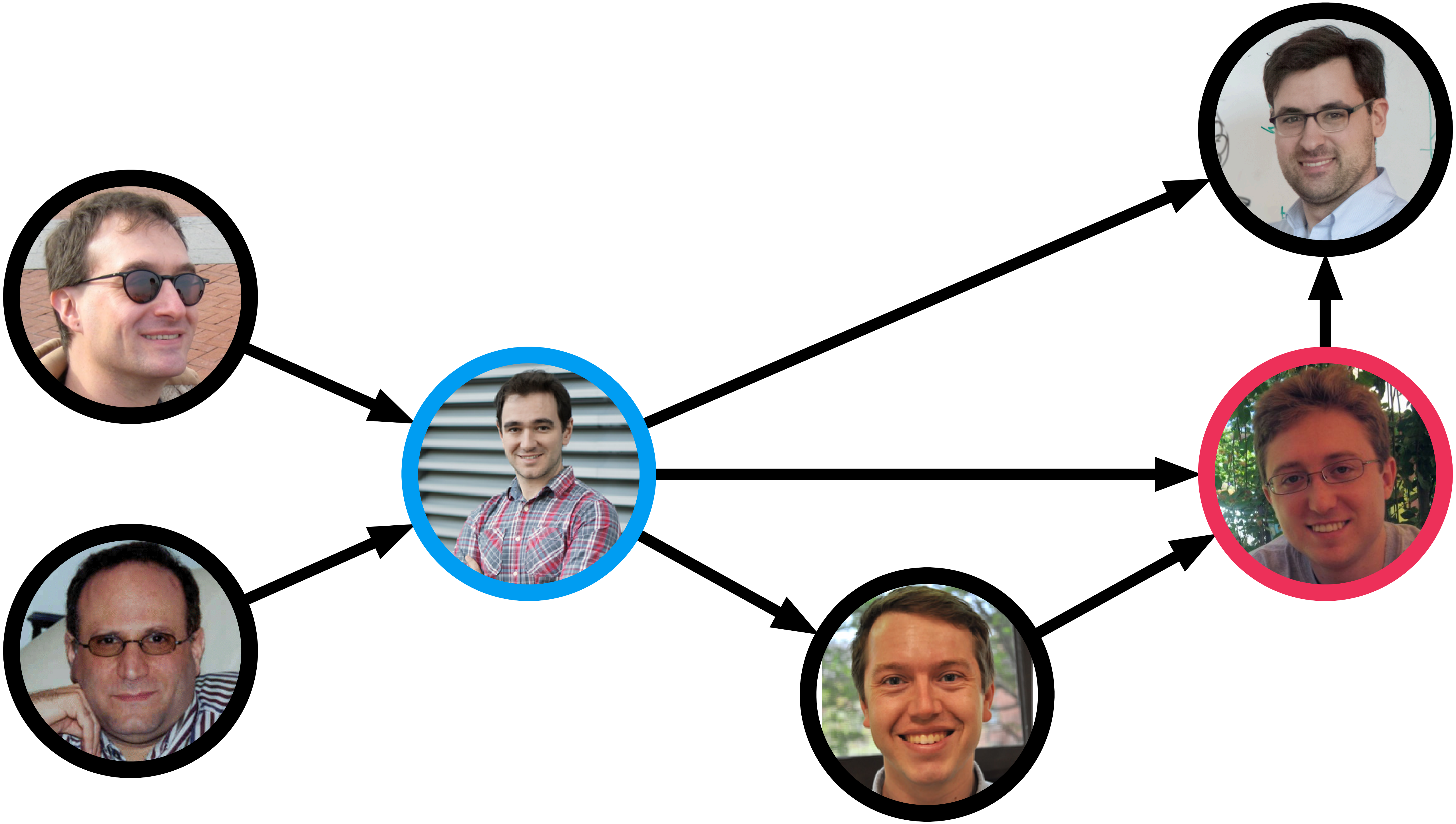
Differentiating fixed points

- Inherits structure from forward iteration
  - Forward is Newton $\Rightarrow$ reverse requires only one step
  - Forward is block coordinate descent $\Rightarrow$ reverse is block Gauss-Seidel
- May be preferable to decouple forward and reverse
  - Then choose any linear solver for implicit linearized system
  - Can reuse dual variables from forward solver

Second-order optimization

```python
def make_hvp(fun, argnum=0):
    """Builds a function for evaluating the Hessian-vector product at a point,
    which may be useful when evaluating many Hessian-vector products at the same
    point while caching the results of the forward pass."""
    def hvp_maker(*args, **kwargs):
        return make_vjp(grad(fun, argnum), argnum)(*args, **kwargs)[0]
    return hvp_maker


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

Thanks!



References

- Dougal Maclaurin. Modeling, Inference and Optimization with Composable Differentiable Procedures. Harvard Physics Ph.D. Thesis, 2016. URL: https://dougalmaclaurin.com/phd-thesis.pdf
- github.com/hips/autograd