Deep Learning (BEV033DLE)
Lecture 3. Backpropagation

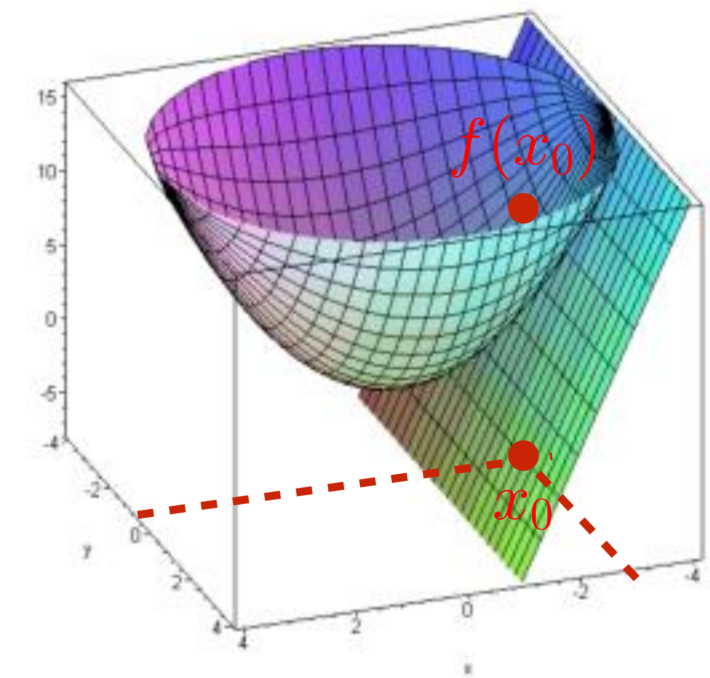
Czech Technical University in Prague

- Theory and Intuition
  - Linear approximation
  - Derivative of compositions
- Practice
  - Forward / backward propagation
  - Efficient implementation, computation graph

Derivative

- Function $f\colon \mathbb{R}^m \to \mathbb{R}^n$ (from domain $\mathbb{R}^m$ to codomain $\mathbb{R}^n$)
- Local linear approximation: $f(x_0 + \Delta x) = f(x_0) + J\Delta x + o(\|\Delta x\|)$
- When such J exists, it is unique and called **derivative**
- Expressed in coordinates J is called the **Jacobian (matrix) (at x_0):**

$\mathbb{R}^2 \to \mathbb{R}$



$$f(x+\Delta x) \approx f(x) + \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_m} \end{pmatrix} \begin{pmatrix} \Delta x_1 \\ \Delta x_2 \\ \vdots \\ \Delta x_m \end{pmatrix}$$

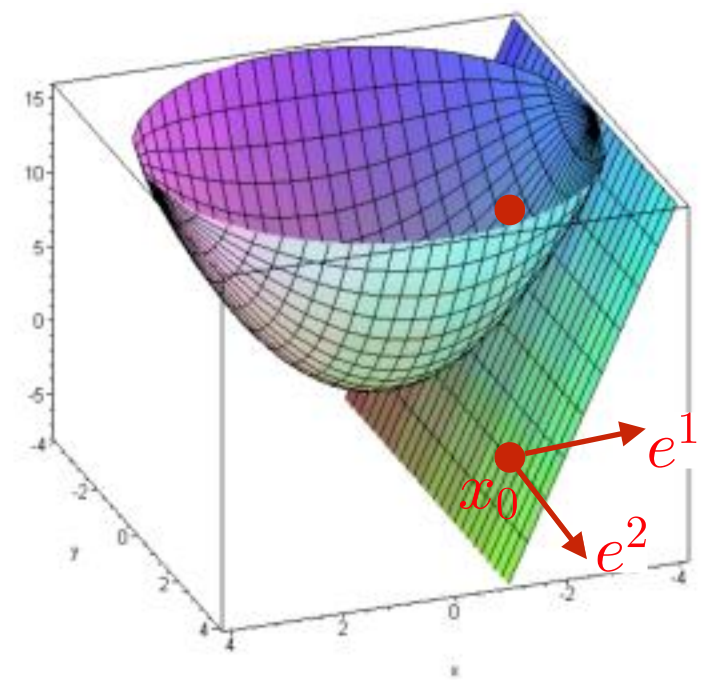
$\frac{\partial f_i}{\partial x_j}$ – speed of growth (slope) of f_i along x_j

- Linear approximations form a closed class under addition and composition:
  - sum of linear functions is linear -- can approximate sum of log-likelihoods over many data points
  - composition of linear functions is linear -- can approximate a deep feed-forward network

Finite Difference

- Given **scalar-valued** function: $\mathcal{L}\colon \mathbb{R}^n \to \mathbb{R}$

$$\mathcal{L}(x_0 + \Delta x) \approx \mathcal{L}(x_0) + \sum_i \frac{\partial L(x)}{\partial x_i} \Delta x_i$$



$$\frac{\partial L(x)}{\partial x_i} = \lim_{\varepsilon \to 0} \frac{L(x_0 + e^i \varepsilon) - L(x_0)}{\varepsilon}$$

Finite difference approximation:

$$\frac{\partial L(x)}{\partial x_i} = \frac{L(x_0 + e^i \varepsilon) - L(x_0)}{\varepsilon} + o(\varepsilon)$$

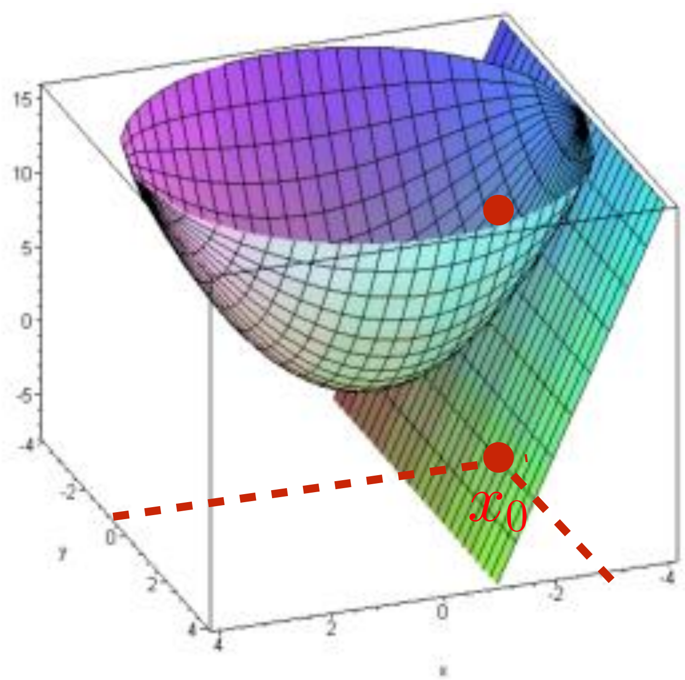
For a twice differentiable function symmetric difference is more accurate:

$$\frac{\partial L(x)}{\partial x_i} = \frac{L(x_0 + e^i \varepsilon) - L(x_0 - e^i \varepsilon)}{\varepsilon} + o(\varepsilon^2)$$

- Can be used for numerical verification
- Can be used with some complex functions that depend on few variables (e.g. solver for the camera pose from 4 correspondences)

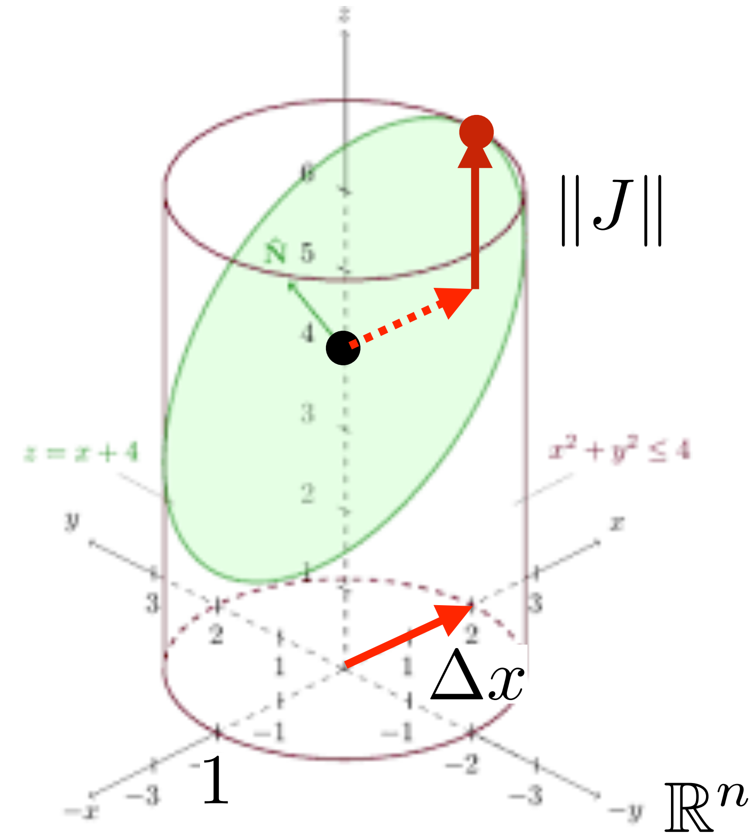
Steepest Ascent

- Given **scalar-valued** function: $\mathcal{L}\colon \mathbb{R}^n \to \mathbb{R}$



$$\mathcal{L}(x_0 + \Delta x) \approx \mathcal{L}(x_0) + J\Delta x$$

J is a *row* vector $\left(\cdots \frac{\partial f}{\partial x_i} \cdots\right)$



- What is the steepest ascent direction to maximize the linear approximation?

$$\max_{\Delta x\colon \|\Delta x\|_2 = 1} \left(f(x^0) + J\Delta x\right) \quad \Rightarrow \quad \Delta x = \frac{J^\mathsf{T}}{\|J\|}$$

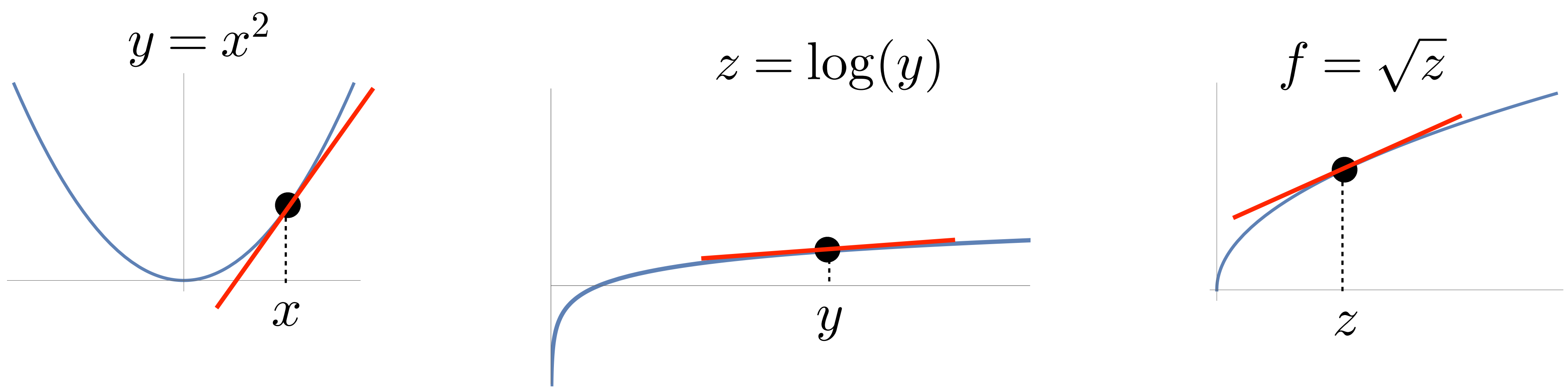
- **Gradient** $\nabla_x f$ is the *column* vector of partial derivatives $\begin{pmatrix} \vdots \\ \frac{\partial f}{\partial x_i} \\ \vdots \end{pmatrix} = J^\mathsf{T}$

- (Steepest) gradient descent: $x^{t+1} = x^t - \varepsilon \nabla_x f(x^t)$

Compositions

- **Our notation** J^f_x – derivative of f in x.
- Linear function: $f(x) = Ax$, then $J^f_x = A$
- Composition of linear functions: $f(x) = ABx$ $\qquad J^f_x = AB$
- Non-linear composition: make a linear approximation to all steps and compose

Example $f = \sqrt{\log(x^2)} = \sqrt{} \circ \log \circ x^2$



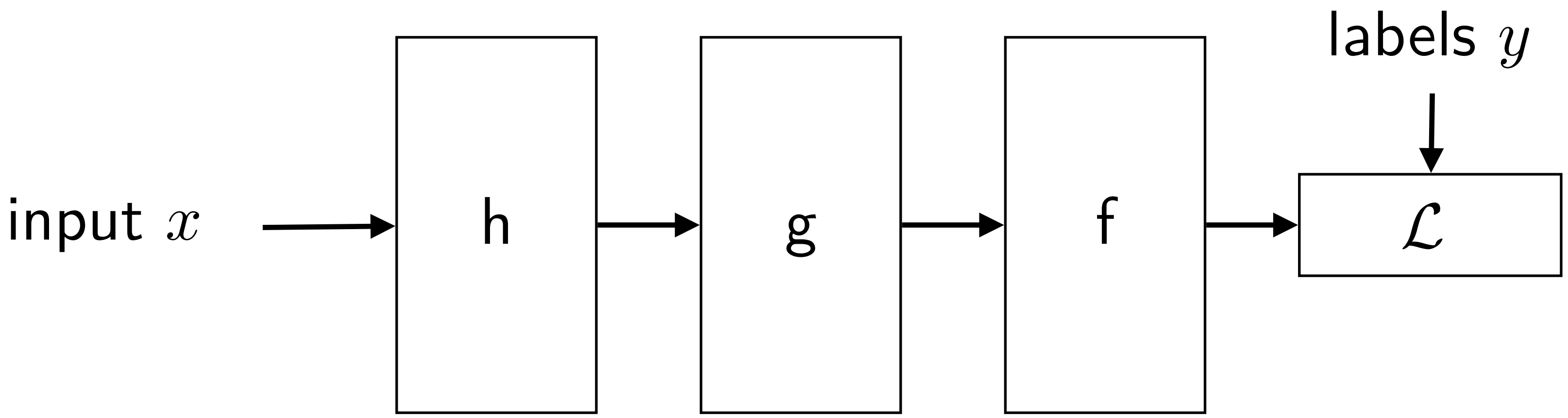
$$J^f_x = \frac{\partial \sqrt{z}}{\partial z}\bigg|_{z=\log(y)} \frac{\partial \log y}{\partial y}\bigg|_{y=x^2} \frac{\partial x^2}{\partial x}\bigg|_x = (\tfrac{1}{2}z^{-1/2})(y^{-1})(2x)$$

- General case, $f(g(x)) = (f \circ g)(x)$:

$$J^f_x = J^f_g J^g_x$$

$$\frac{\mathrm{d} f_i}{\mathrm{d} x_j} = \sum_k \frac{\partial f_i}{\partial g_k}\frac{\partial g_k}{\partial x_j} \qquad \text{(chain / total derivative rule)}$$

In Neural Networks



Loss $\mathcal{L}(f(g(h(x))))$

- We will need derivatives of the loss in different parameters. Shorthand: $J_x \equiv J_x^{\mathcal{L}} \equiv \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}x}$.
- In order to compute J_x we need to multiply all Jacobians:

Loss: $\mathcal{L} \circ f \circ g \circ h \circ x$

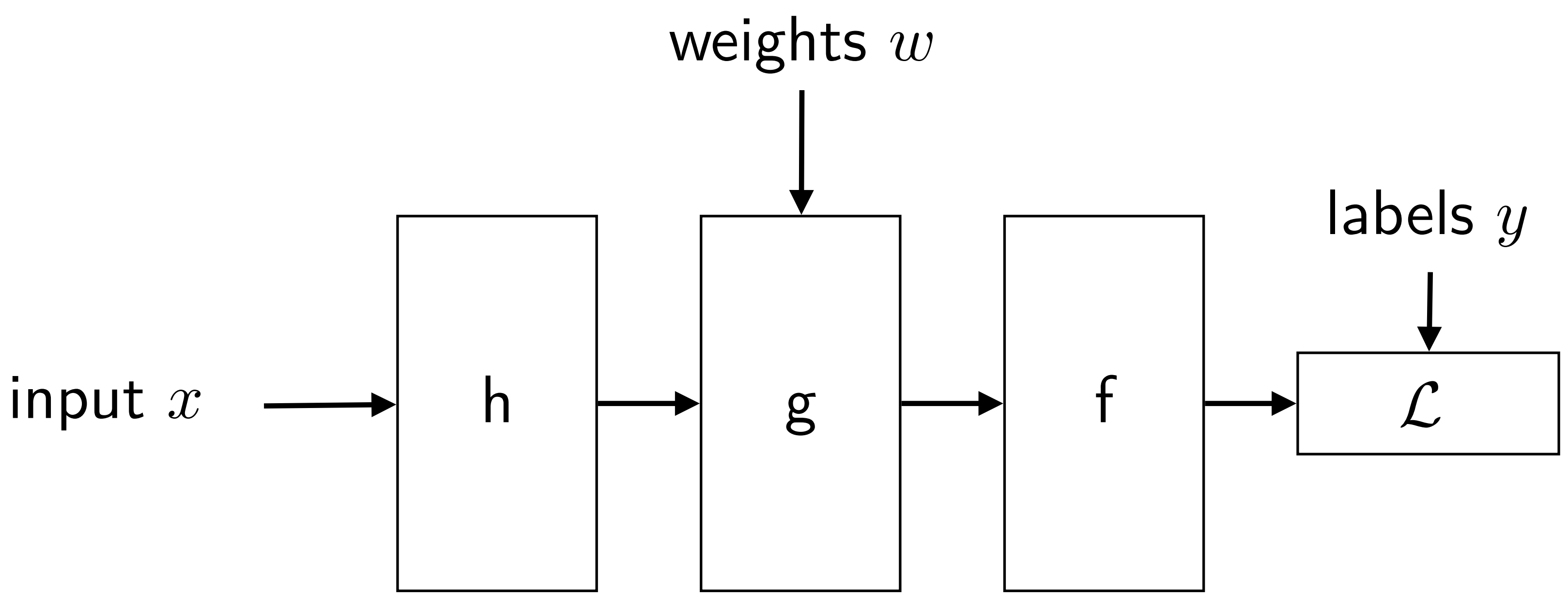
Derivative: $J_x = J_f^{\mathcal{L}} \; J_g^f \; J_h^g \; J_x^h$

Expanded: $J_x = \left(\frac{\partial \mathcal{L}}{\partial f_1} \; \frac{\partial \mathcal{L}}{\partial f_2} \; \cdots \; \frac{\partial \mathcal{L}}{\partial f_n} \right) \left(\quad J_g^f \quad \right) \left(\quad J_h^g \quad \right) \left(\quad J_x^h \quad \right)$

- Matrix product is associative
- Going left-to-right is cheaper: $O(Ln^2)$ *vs.* $O((L-1)n^3 + n^2)$, for L layers with n neurons

Backpropagation

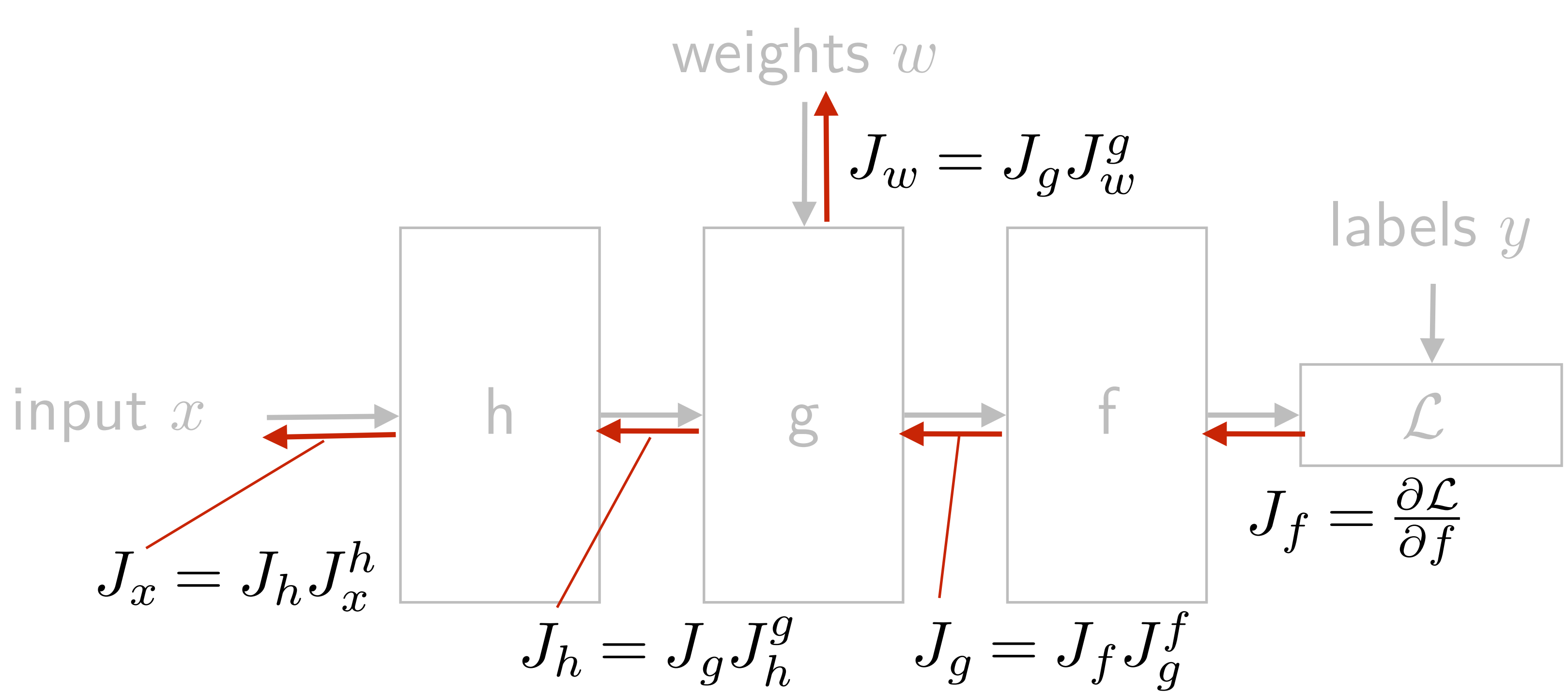
✦ **Forward** — composition of functions



Loss $\mathcal{L}(f(g(h(x),w)),y)$

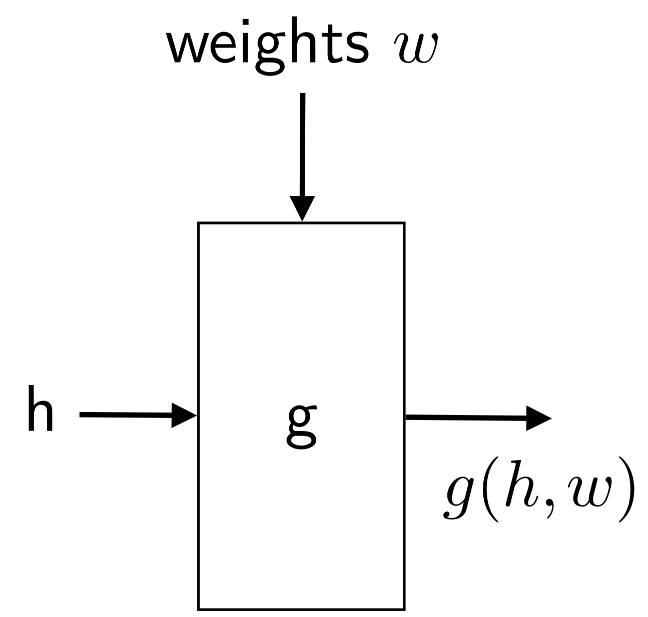
- Feed-forward network ⇒ computation graph is a DAG
- Composition (∘) notation possible when fixing all but one inputs

✦ **Backward:**

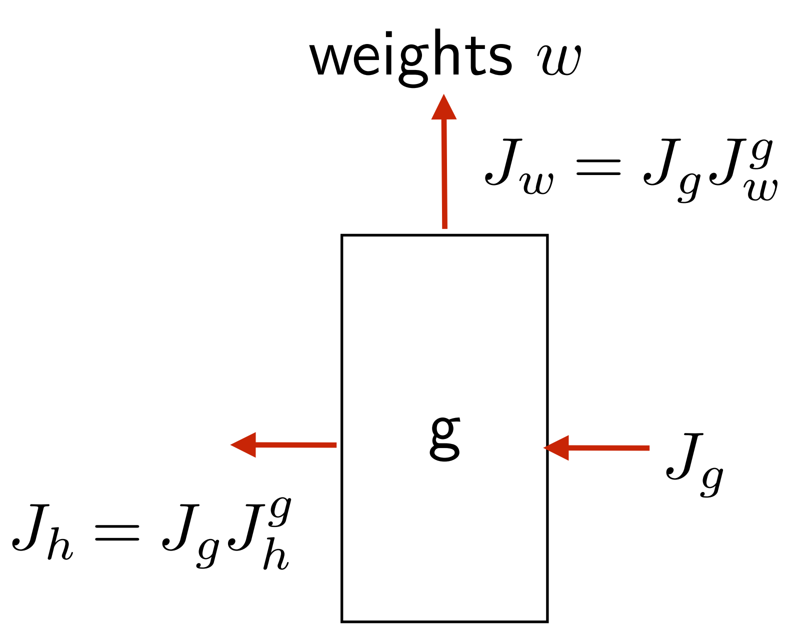


Both Forward and Backward are Modular

- **Forward:**

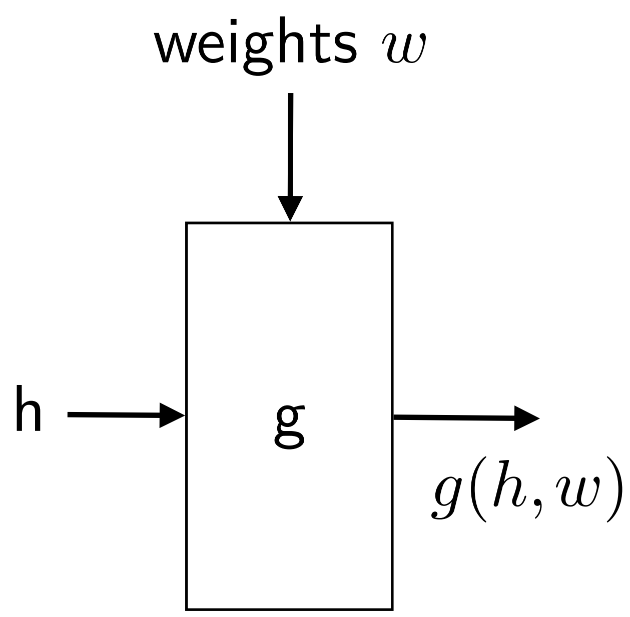


- **Backward:**

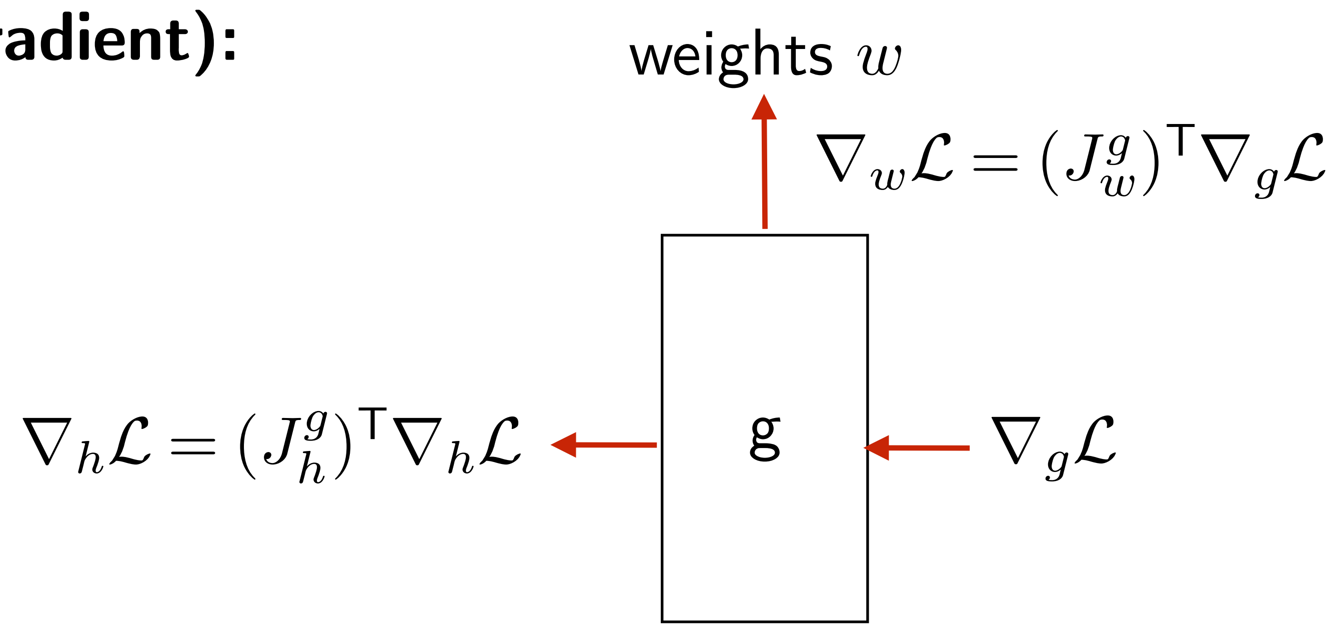


Both Forward and Backward are Modular

- **Forward:**



- **Backward (gradient):**



Example

- Correlation layer with inputs x and weights w: $y_j = \sum_k w_k x_{j+k} + b_j$
  - Backward input: $g_j = \frac{d\mathcal{L}}{dy_j}$
  - Need $\frac{d\mathcal{L}}{dx_i}, \frac{d\mathcal{L}}{dw_k}, \frac{d\mathcal{L}}{db_j}$
  - Total derivative (chain) rule:

$$
\begin{aligned}
\frac{d\mathcal{L}}{dx_i} &= \sum_j \frac{d\mathcal{L}}{dy_j}\frac{\partial y_j}{\partial x_i} \\
&= \sum_j g_j \frac{\partial}{\partial x_i}\Big(\sum_k w_k x_{j+k} + b_j\Big) \\
&= \sum_j g_j \sum_k w_k \frac{\partial}{\partial x_i} x_{j+k} \\
&= \sum_j g_j \sum_k w_k [\![j+k=i]\!] \\
&= \sum_j g_j w_{i-j}
\end{aligned}
$$

Various special cases of linear dependencies can be handled in O(n) instead of O(n^2)

Computation Graph, Forward Propagation

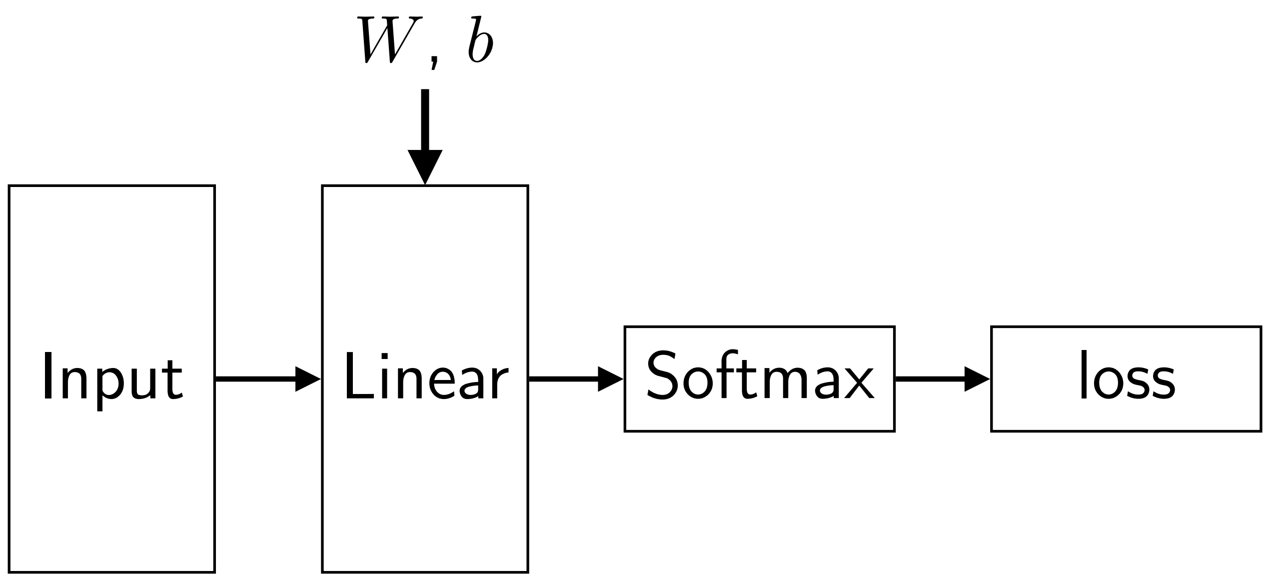
- Dynamic Graph (Eager Execution): just compute what we need

Declare and initialize variables

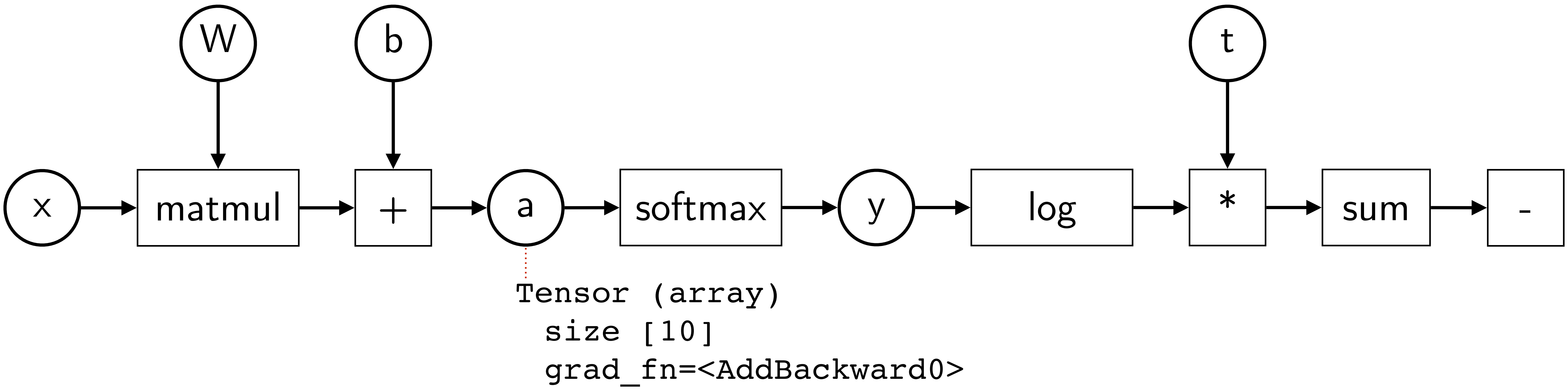
```
from torch.nn import Parameter
import torch.nn.functional as F

W = Parameter(torch.randn(10, 748))
b = Parameter(torch.randn(10))
```

Perform some operations

```
x = torch.randn(748)
t = torch.ones(10)
a = W @ x + b
y = F.softmax(a)
loss = -(t * y.log()).sum()
```

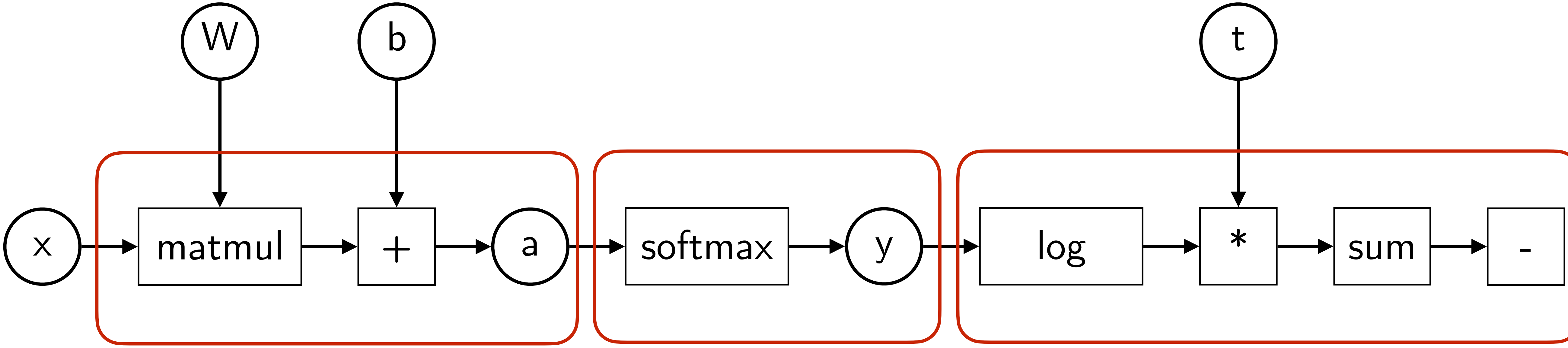


Computation graph defined by the operations performed:



- Wow! Any computation can be made a part of a neural network

Backward Propagation



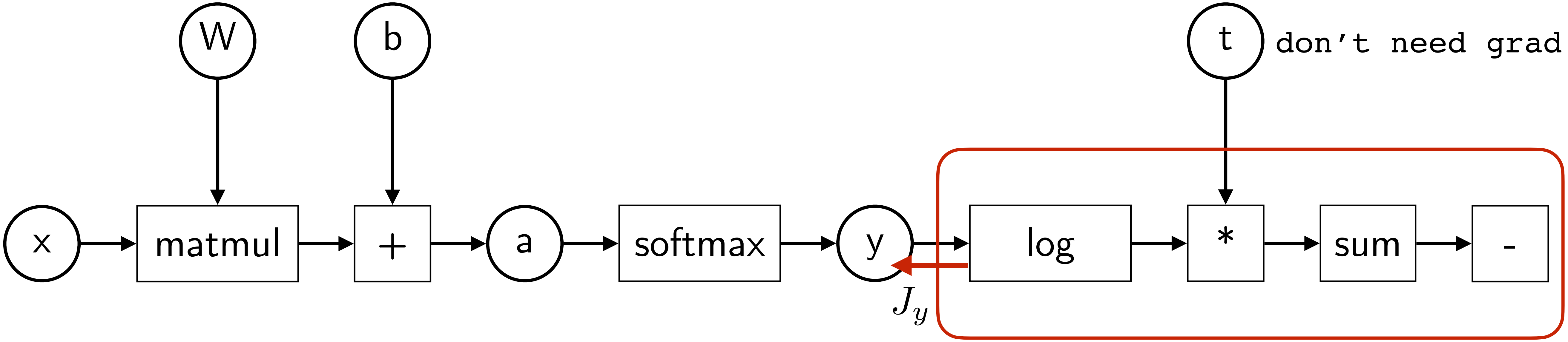
$a = \mathrm{linear}(x, W, b)$ $\quad y = \mathrm{softmax}(a)$ $\quad L = -t^{\mathsf{T}} \log(y)$

```
a = F.linear(x,W,b)
```

```
y = F.softmax(a)
```

- Computationally more efficient to compute backward for larger blocks. Also convenient for this example.

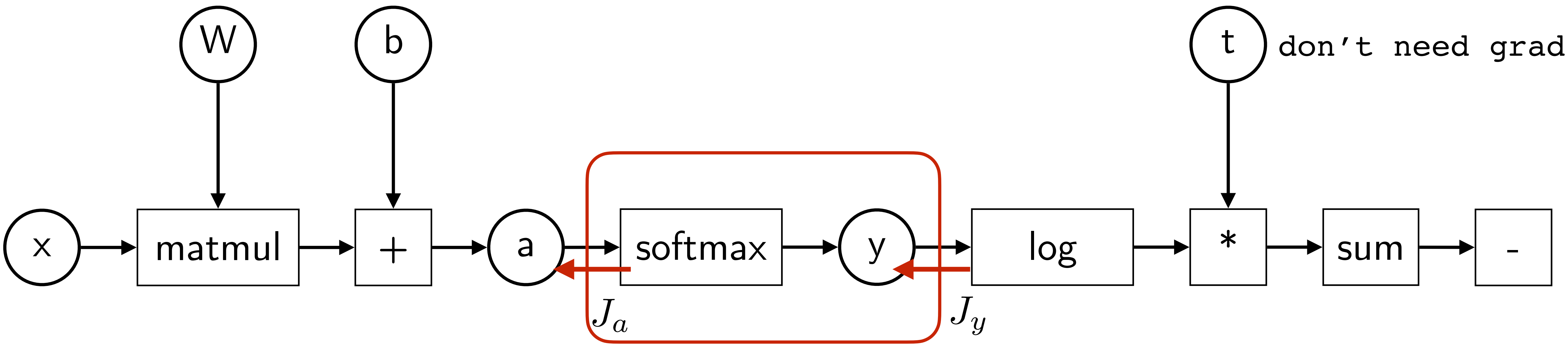
Backward Propagation



$$\mathcal{L} = -t^\top \log(y)$$

$$J_{y_i} = \frac{\partial \mathcal{L}}{\partial y_i} = -\frac{\partial}{\partial y_i} \sum_j t_j \log(y_j) = -\frac{1}{y_i} t_i$$

Backward Propagation



$$y_j = \mathrm{softmax}(a)_j = \frac{e^{a_j}}{\sum_i e^{a_i}}$$

$$J_{a_i} = J_y J_a^y = \sum_j J_{y_j} \frac{\partial y_j}{\partial a_i}$$

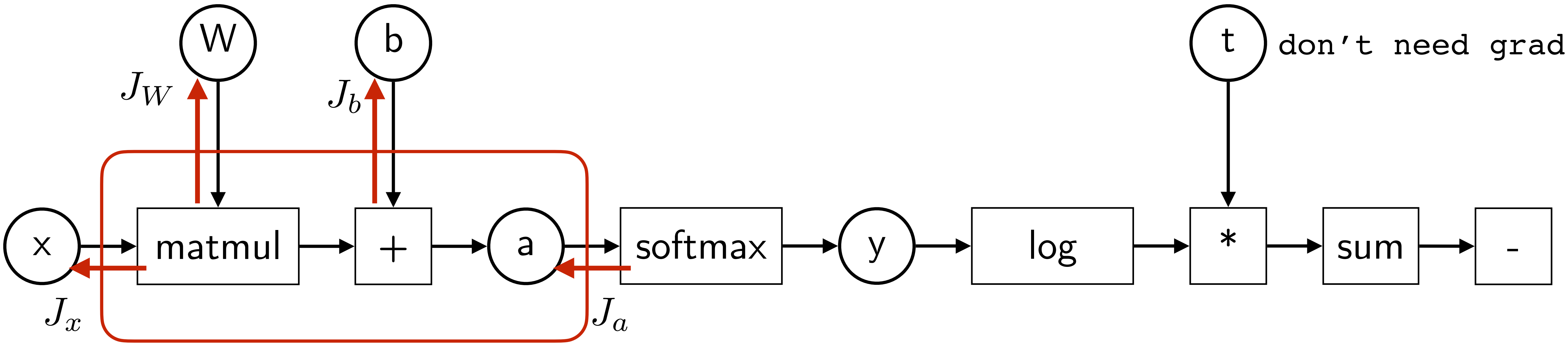
$$= \sum_j J_{y_j} (y_i [\![i{=}j]\!] - y_i y_j) = y_i \big(J_{y_i} - \sum_j y_j J_{y_j}\big)$$

$$J_a = J_y(\mathrm{Diag}(y) - yy^{\mathsf{T}}) = J_y \odot y - (J_y y) y^{\mathsf{T}}$$

(need to remember either input a or directly the output y)

Notice: forward and backward are both linear complexity

Backward Propagation



$$a_j = \sum_i W_{ji} x_i + b$$

$$J_b = J_a \qquad (\star)$$

$$J_{x_k} = \sum_j J_{a_j} \frac{\partial a_j}{\partial x_i} = \sum_j J_{a_j} W_{i,j} [\![i{=}k]\!] = \sum_j J_{a_j} W_{k,j}$$

$$J_x = J_a W$$

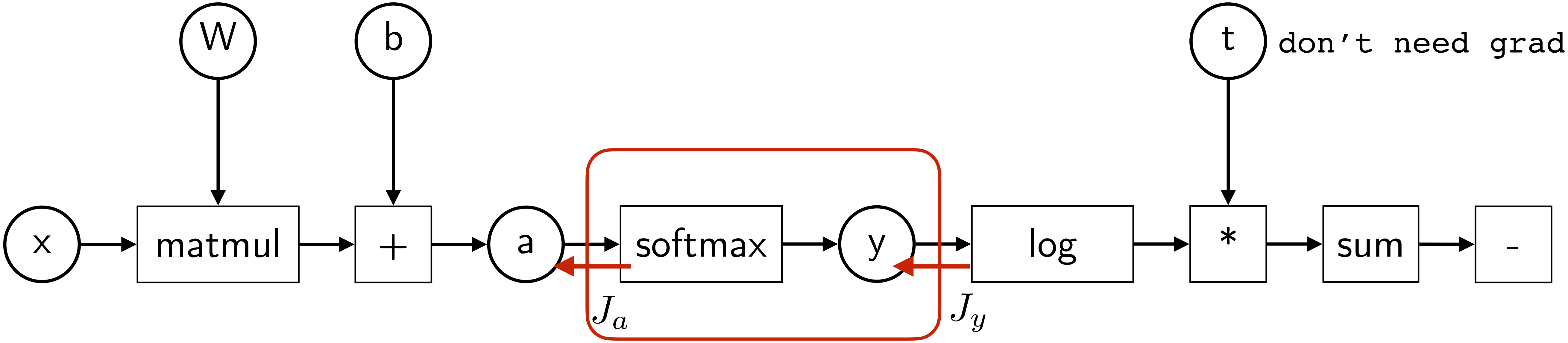
$$\nabla_x \mathcal{L} = W^{\mathsf{T}} \nabla_a \mathcal{L}$$

Note: a transposed product in comparison with Wx

$$J_{W_{ij}} = \sum_j J_{a_j} \frac{\partial a_j}{\partial W_{ij}} = x_i J_j$$

$J_W = xJ$ – outer (column-row) product

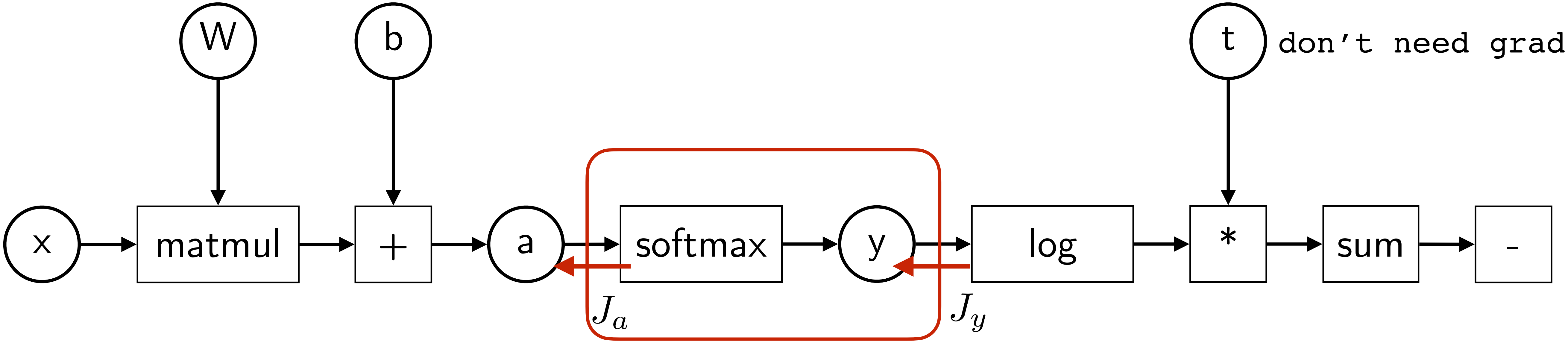
Backward Propagation



✦ What we have learned towards practical implementation:

- Do not need to explicitly compute the Jacobian of each layer, only need to "backpropagate" through the layer
- The granularity is up to the implementation: flexibility vs. efficiency
- Need to store the input (point at which the Jacobian is evaluated) or recompute it
- In real applications gradients are often shaped as higher dimensional tensors:
  E.g. convolution with weights w [in, out, k_h, k_w]
  - gradient in w is shaped as [in, out, k_h, k_w]
  - special efficient implementation for forward
  - special efficient implementation for backward (transposed convolution)

Backward Propagation



$$y_j = \mathrm{softmax}(a)_j = \frac{e^{a_j}}{\sum_i e^{a_i}}$$

$$J_a = J_y(\mathrm{Diag}(y) - yy^\mathsf{T}) = J_y \odot y - (J_y y) y^\mathsf{T}$$

1)
```
y = a.softmax()
```

2)
```
class MySoftmax(torch.nn.Module):
    def forward(self, a):
        y = a.exp()
        y = y / y.sum()
        return y
```

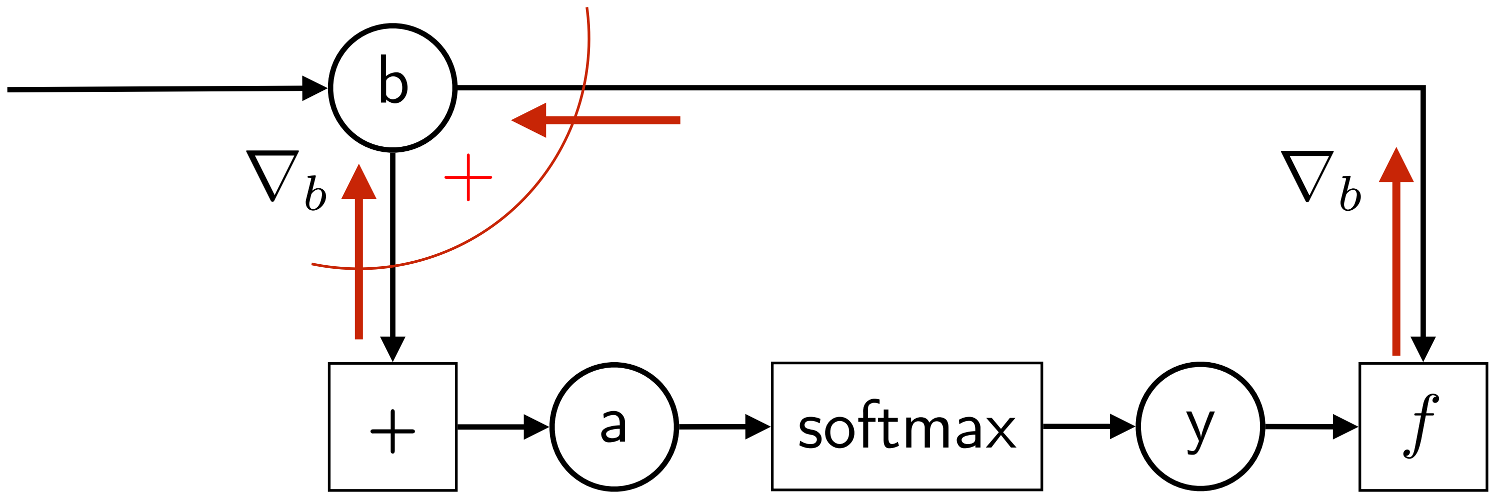
```
y = MySoftmax().forward(a)
```

3)
```
class MySoftmax(torch.autograd.Function):
    @staticmethod
    def forward(ctx, a):
        y = a.exp()
        y /= y.sum()
        ctx.save_for_backward(y)
        return y

    @staticmethod
    def backward(ctx, dy):
        y, = ctx.saved_tensors
        da = y * dy - y * (y * dy).sum()
        return da
```

```
y = MySoftmax.apply(a)
```

General DAG

- Consider the case when some of the inputs are used in several places



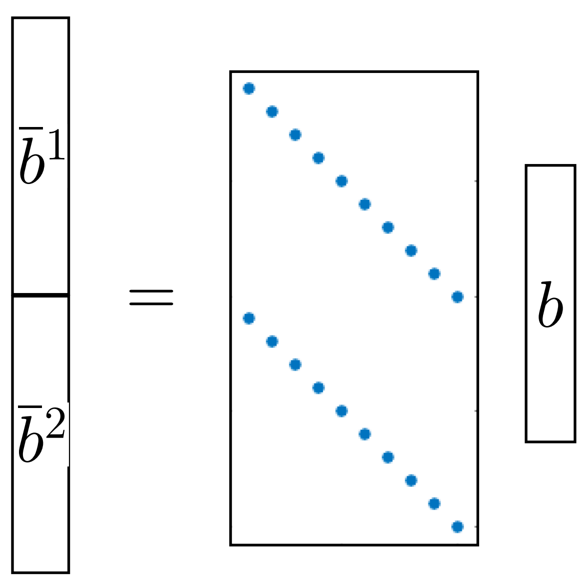
- The total derivative rule:

$$\frac{\mathrm{d}}{\mathrm{d}b} f(b, y(b)) = \frac{\partial f}{\partial b} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}b}$$

- The copy operation:

$$\bar{b}^1 = b$$

$$\bar{b}^2 = b$$

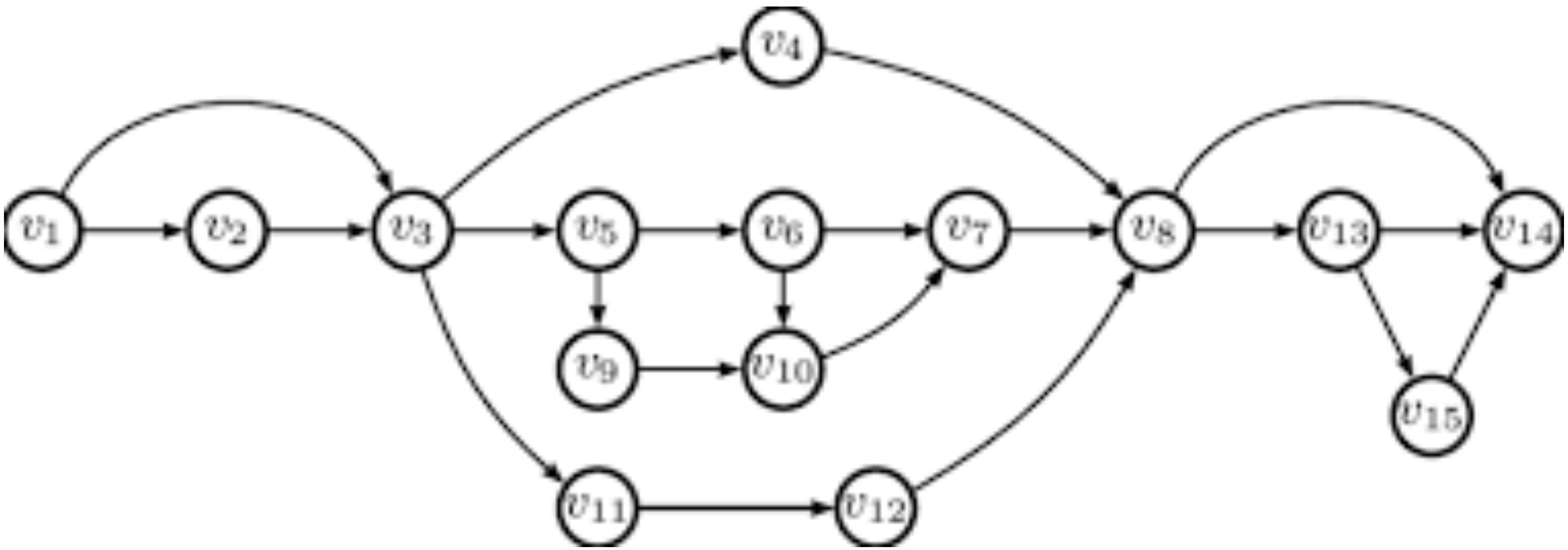


$$J_b = J_{\bar{b}^1} + J_{\bar{b}^2}$$

General DAG

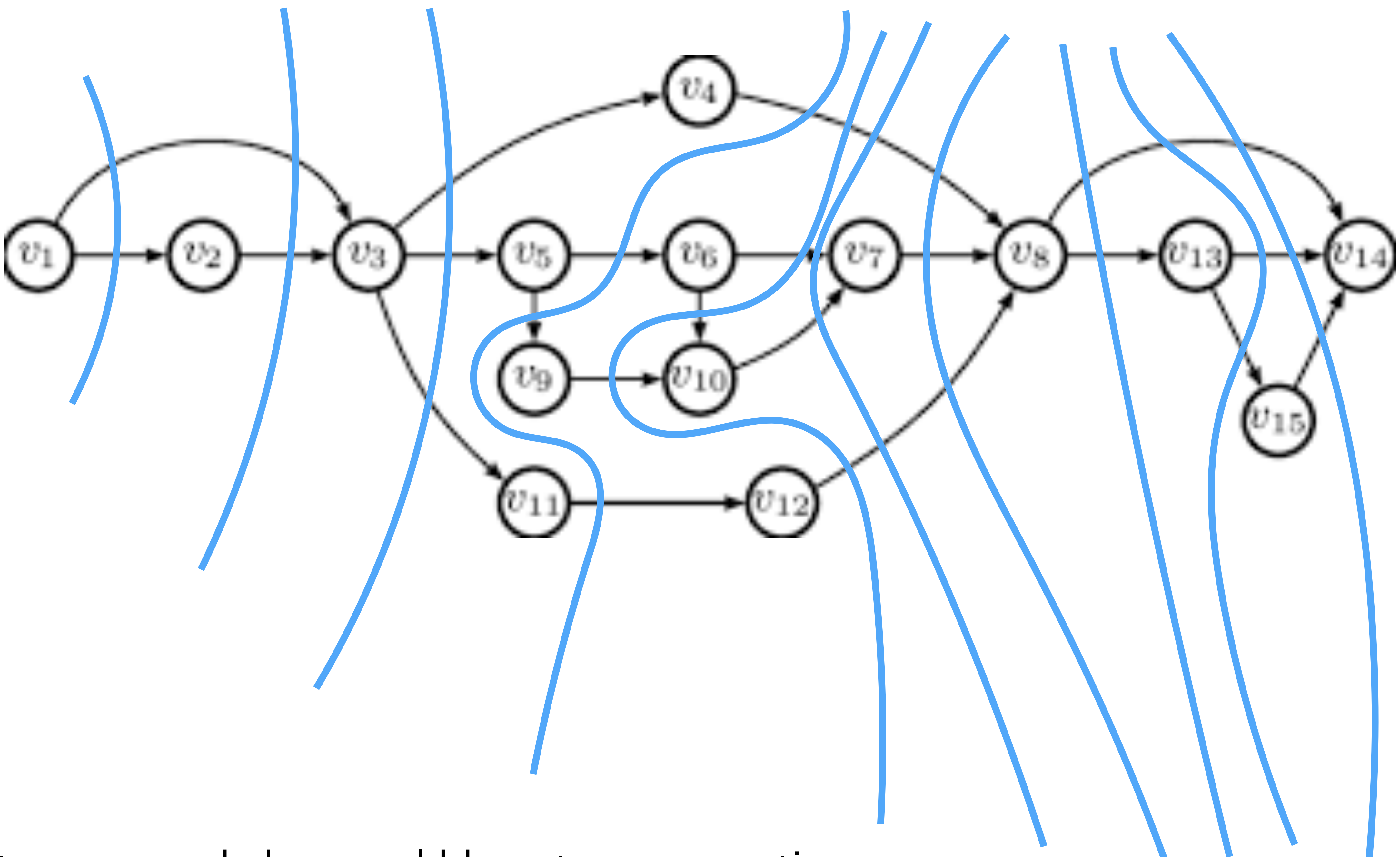
✦ Need to find the order of processing

- a node may be processed when all its parents are ready
- some operations can be executed in parallel
- reverse the edges for the backward pass



General DAG

- Any directed acyclic graph can be topologically ordered
  - Equivalent to a layered network with skip connections



Note: every node here could be a tensor operation

Implicit Layers

- Explicit layer: $z = f(x)$
- Implicit Layer: $g(x, z) = 0$

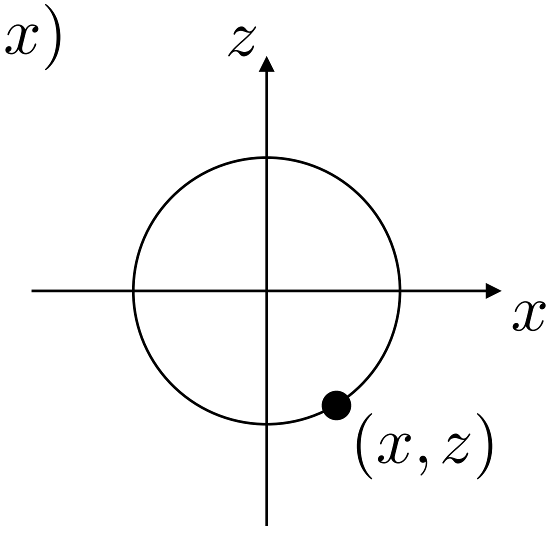
  Examples:
  - Eigenvalue decomposition: input X output (U, S) such that $X = USU^T$.
  - Finding camera pose from correspondences (system of algebraic equations)
  - Limit of fixed point iteration: $z^* = f(x, z^*)$
  - Optimization problems: $z^* = \mathrm{argmin}_z f(z, x)$
  - Differential equations

- Deep Equilibrium Model: $z^* = f(Wz^* + Ux + b)$
  - Can represent any feed-forward NN, with the same number of parameters

Implicit Layers

- Implicit Layer
  - Let $g\colon \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$. Consider vector equation $g(x,z) = \mathbf{0}$
  - Suppose we can compute a solution (x,z)
  - Under some conditions z is locally a function of x, let's call it $z(x)$

$$x^2 + z^2 = 1$$



$$g(x, z(x)) = 0$$

$$\frac{\mathrm{d}g(x, z(x))}{dx} = 0$$

$$\frac{\partial g(x, z(x))}{\partial x} + \frac{\partial g(x,z)}{\partial z}\Big|_{z=z(x)} \frac{\partial z(x)}{\partial x} = 0$$

$$\frac{\partial z(x)}{\partial x} = \Big(\frac{\partial g(x,z)}{\partial z}\Big)^{-1} \frac{\partial g(x,z)}{\partial x}.$$

- Backprop of $\mathcal{L}(z)$:
  - $J_z^{\mathcal{L}} \Big(\frac{\partial g(x,z)}{\partial z}\Big)^{-1} \Big(\frac{\partial g(x,z)}{\partial x}\Big)$
  - Need to solve a linear system on backward