Deep Learning (BEV033DLE)
Lecture 3. Backpropagation

Czech Technical University in Prague

✦ Theory and Intuition

- Linear approximation
- Derivative of compositions

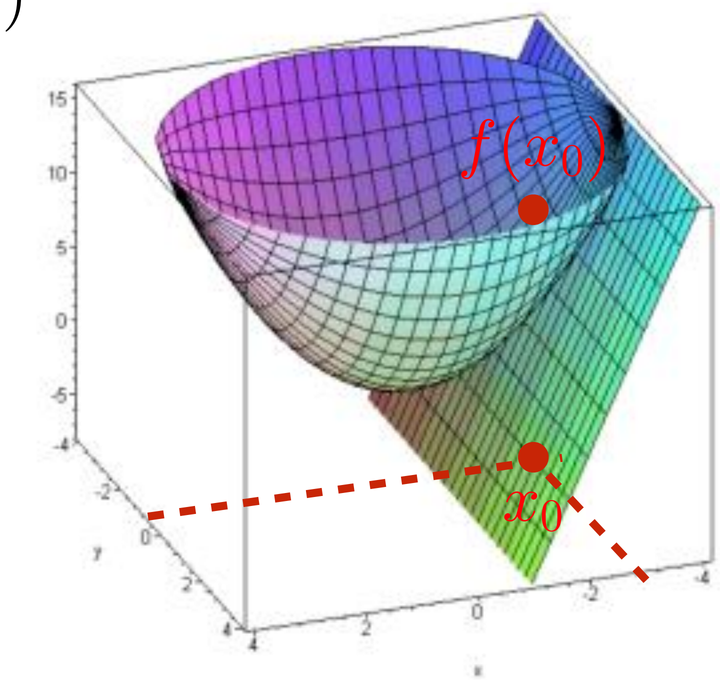
✦ Practice

- Forward / backward propagation
- Efficient implementation, computation graph

Derivative

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

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



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

$\frac{\partial f_i}{\partial x_j}$ – speed of growth of f_i when increasing x_j

- Linear approximations form a closed class under:
  - sum of functions (e.g. sum of log-likelihoods over many data points)
  - composition of functions (e.g. deep feed-forward network)

Compositions

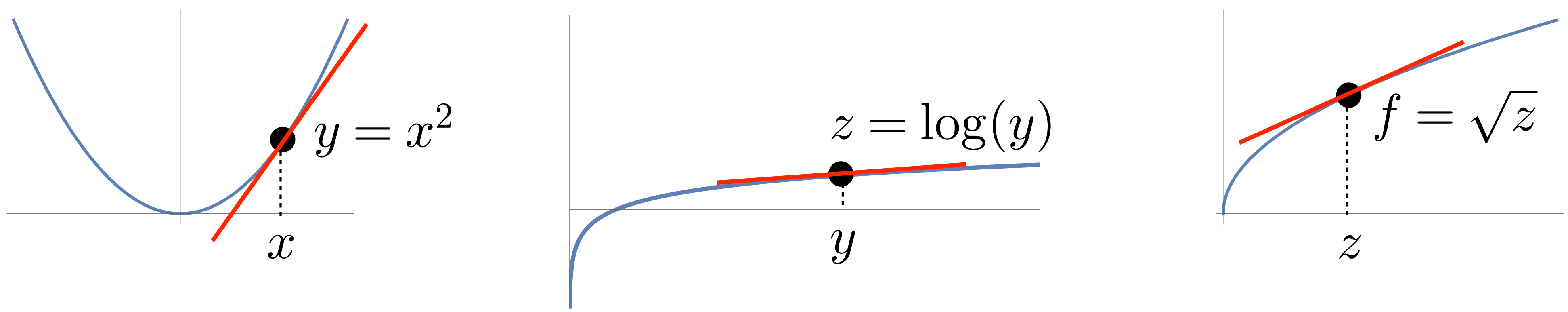
- Linear function: $f(x) = Ax$

  Derivative of f in x: **Our notation**: $J^f_x = A$

- Composition of linear functions: $f(x) = ABx$ $\quad J^f_x = AB$

- Non-linear composition: make a linear approximation to all steps

Example $f = \sqrt{\log(x^2)}$: Composition: $\sqrt{\ } \circ \log \circ \mathrm{pow}_2$



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

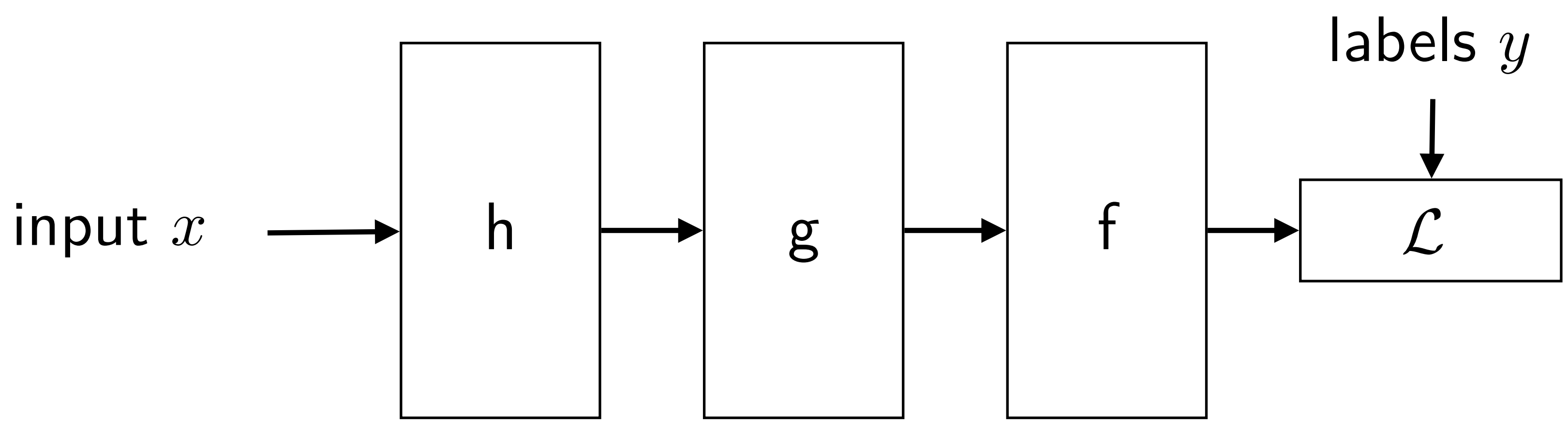
- General case, $(f \circ g)(x) = f(g(x))$:

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

$$J^{f_i}_{x_j} = \sum_k J^{f_i}_{g_k} J^{g_k}_{x_j}$$

$$\frac{\mathrm{d}f_i}{\mathrm{d}x_j} = \sum_k \frac{\partial f_i}{\partial g_k}\frac{\partial g_k}{\partial x_j}$$ (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: $\mathcal{L} \circ f \circ g \circ h \circ x$

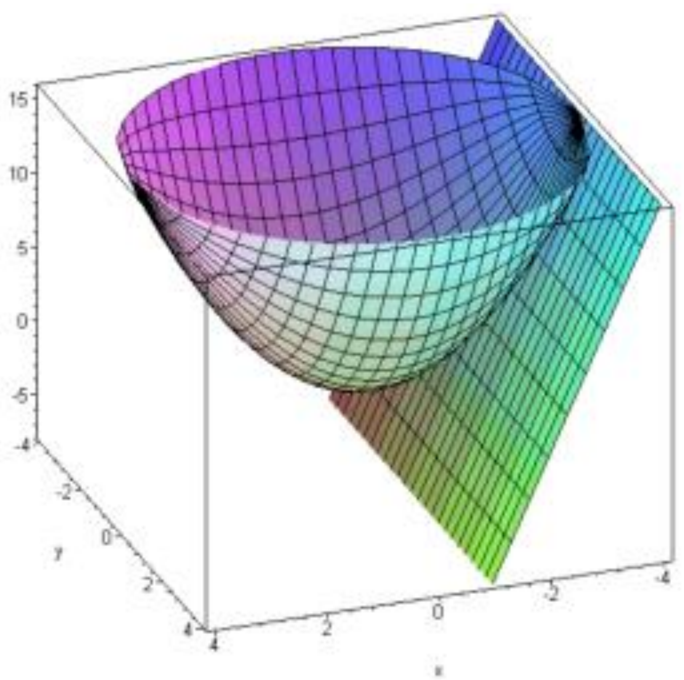
Derivative: $J_x = J_f^{\mathcal{L}} \; J_g^f \; J_h^g \; J_x^h$

Expanded:
$$J_x = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial f_1} & \frac{\partial \mathcal{L}}{\partial f_2} & \cdots & \frac{\partial \mathcal{L}}{\partial f_n} \end{pmatrix} \begin{pmatrix} & & \\ & J_g^f & \\ & & \end{pmatrix} \begin{pmatrix} & & \\ & J_h^g & \\ & & \end{pmatrix} \begin{pmatrix} & & \\ & J_x^h & \\ & & \end{pmatrix}$$

- 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

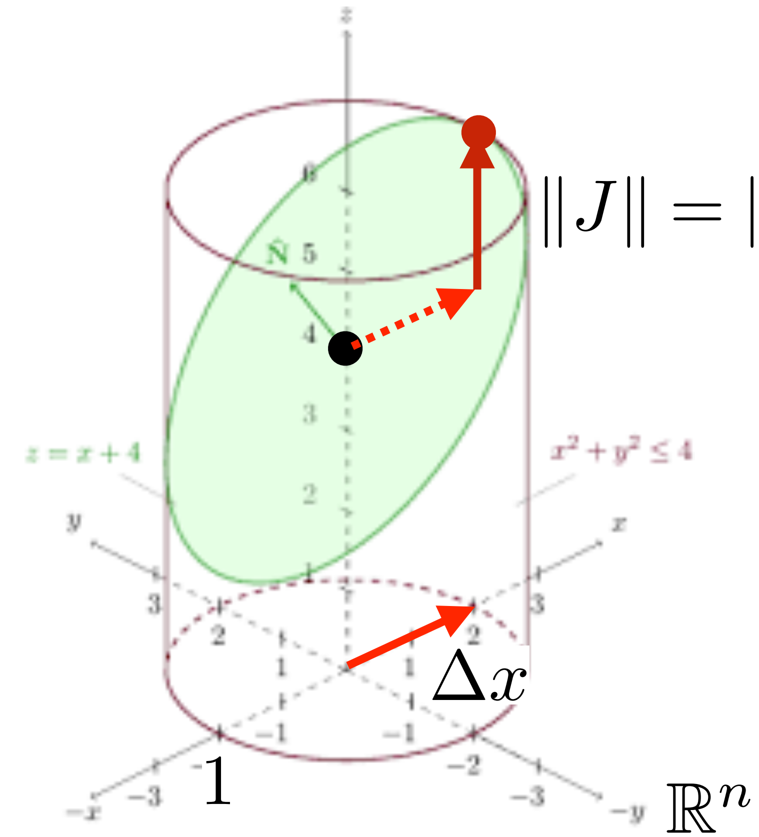
Gradient

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



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

$$(J \equiv J^{\mathcal{L}}_x \;)$$



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

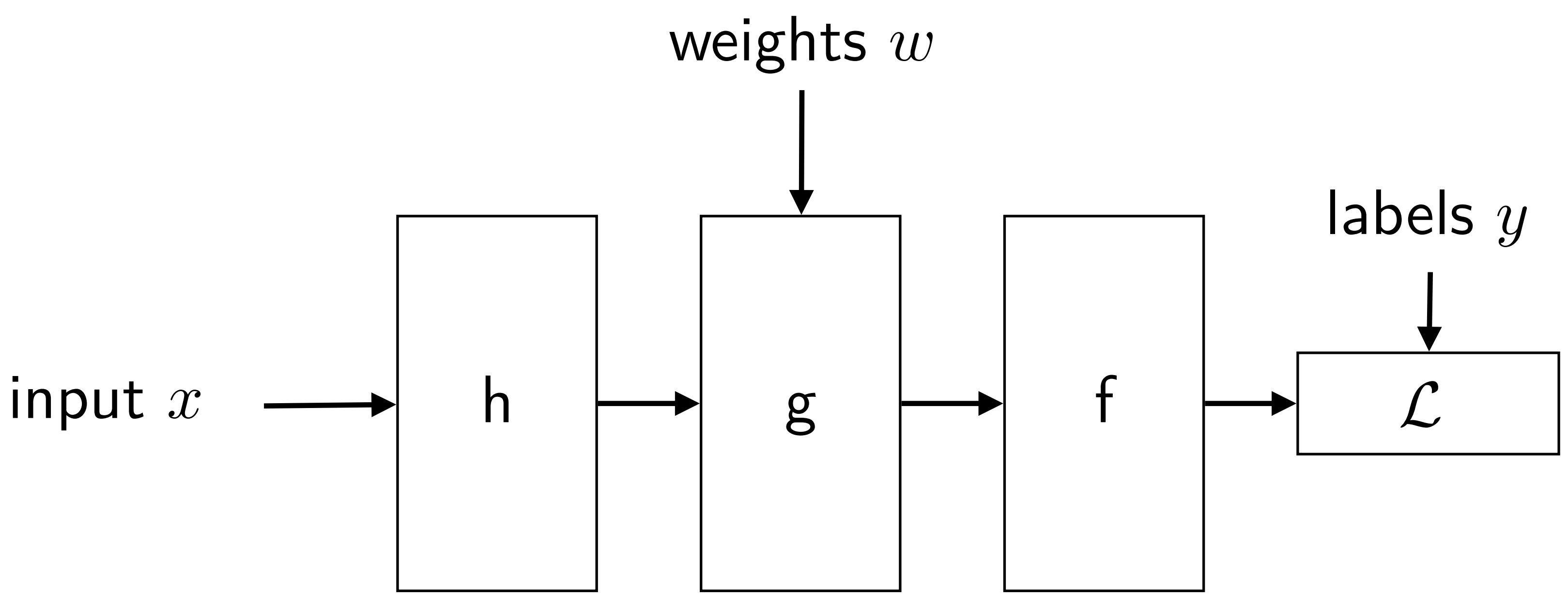
- What is the steepest ascent direction to maximize the linear approximation?

$$\max_{\Delta x\colon\, \|\Delta x\|_2 = 1} \big(f(x_0) + J\Delta x\big) \quad \Rightarrow \quad \Delta x = \frac{J^{\mathsf{T}}}{\|J\|}$$

- **Gradient** $\nabla_x f$ is the *column* vector of partial derivatives $\begin{pmatrix} \vdots \\ \frac{\partial f}{\partial x_i} \\ \vdots \end{pmatrix} = J^{\mathsf{T}}$
  - Not necessarily the best direction to make an optimization step.

Backpropagation

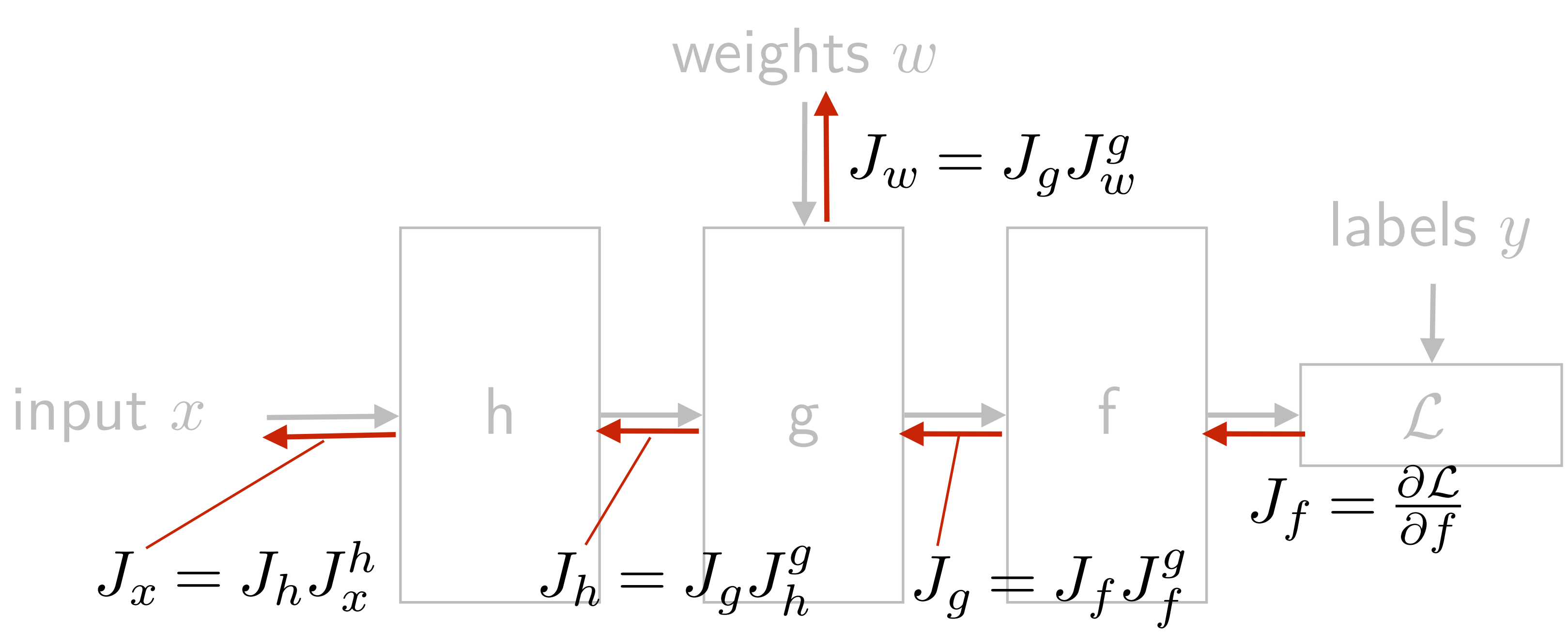
- **Forward** — composition of functions



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

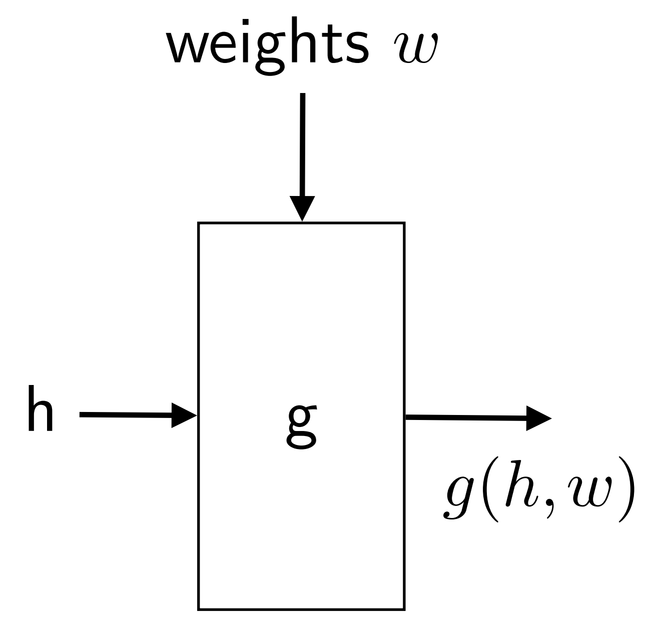
- No "∘" notation possible
- Feed-forward network ⇒ computation graph is a DAG

- **Backward:**

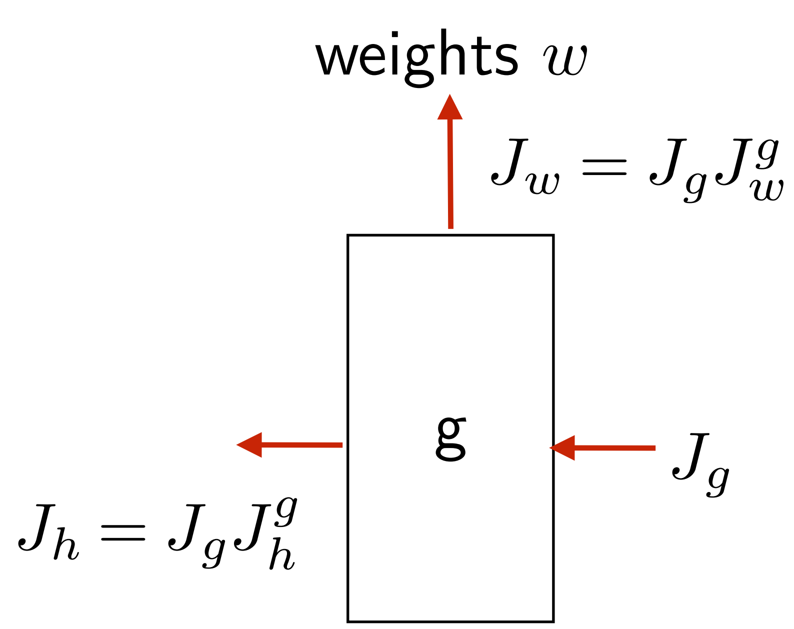


Backpropagation is "Modular"

- **Forward:**

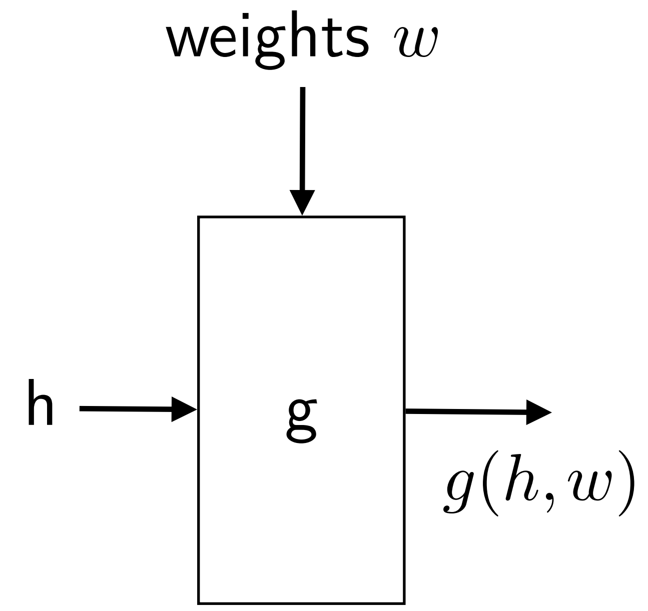


- **Backward:**

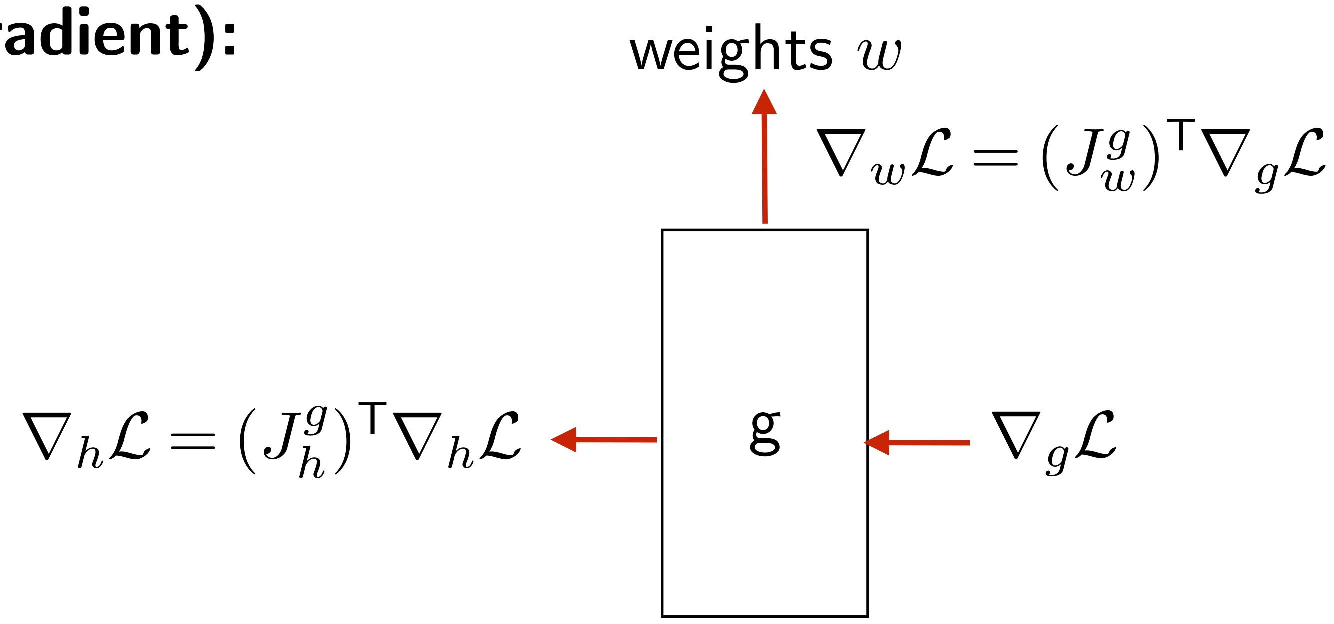


Backpropagation is "Modular"

- **Forward:**



- **Backward (gradient):**



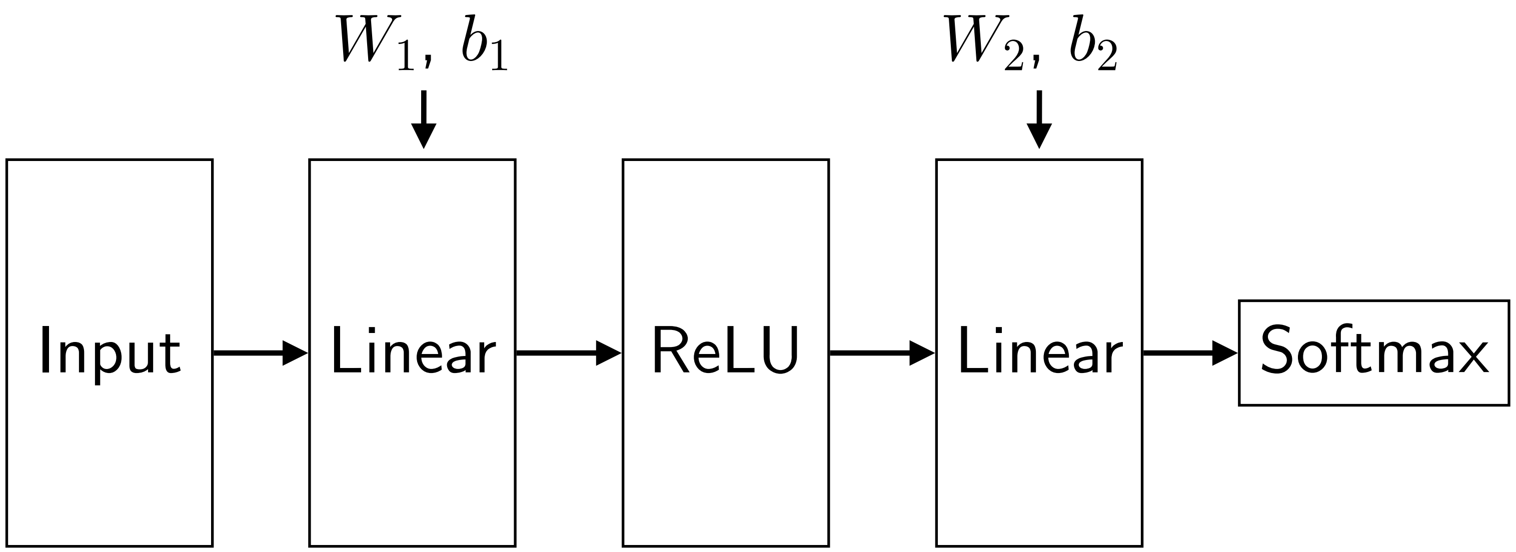
Computation Graph, Forward Propagation

✦ Approach 1:

- Declare

```
import torch
import torch.nn as nn

net = nn.Sequential(
    nn.Linear(748, 200),
    nn.ReLU(),
    nn.Linear(200, 10),
    nn.Softmax(),
)
```



- used to defining the graph in earlier approaches (in TF)

- in Pytorch no graph yet, just a list of "Modules"

- Execute it with some input (forward propagation)

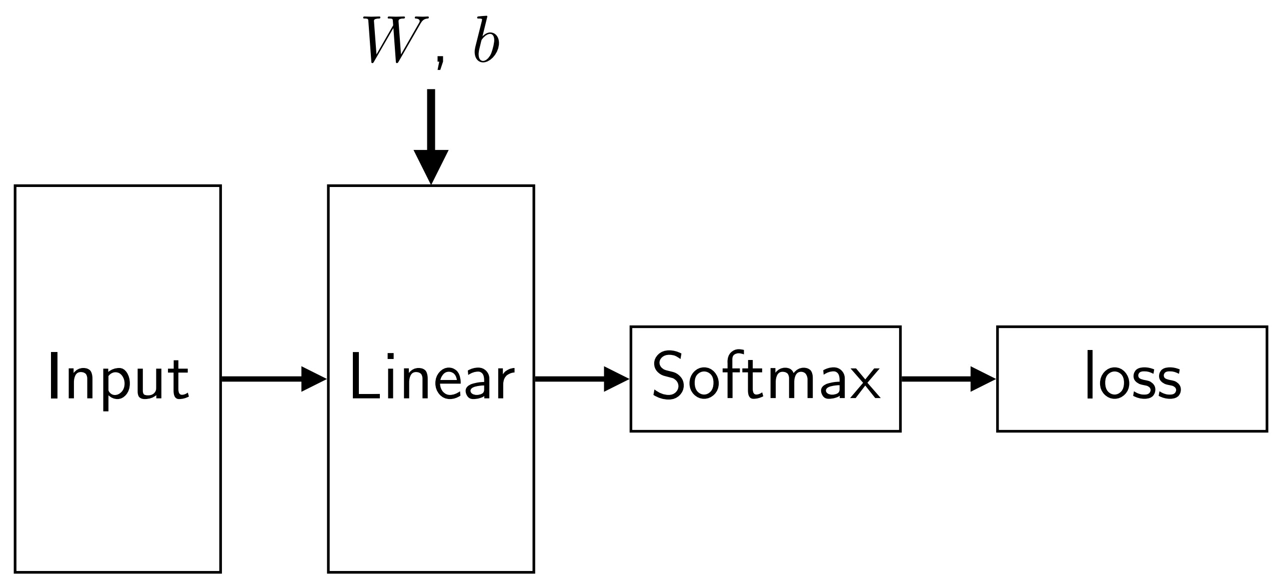
```
x = torch.randn(748)
y = net.forward(x)
```

Pytorch creates the graph dynamically when the forward computation takes place

Computation Graph, Forward Propagation

✦ Approach 2:

- Compute what we need



Declare and initialize variables

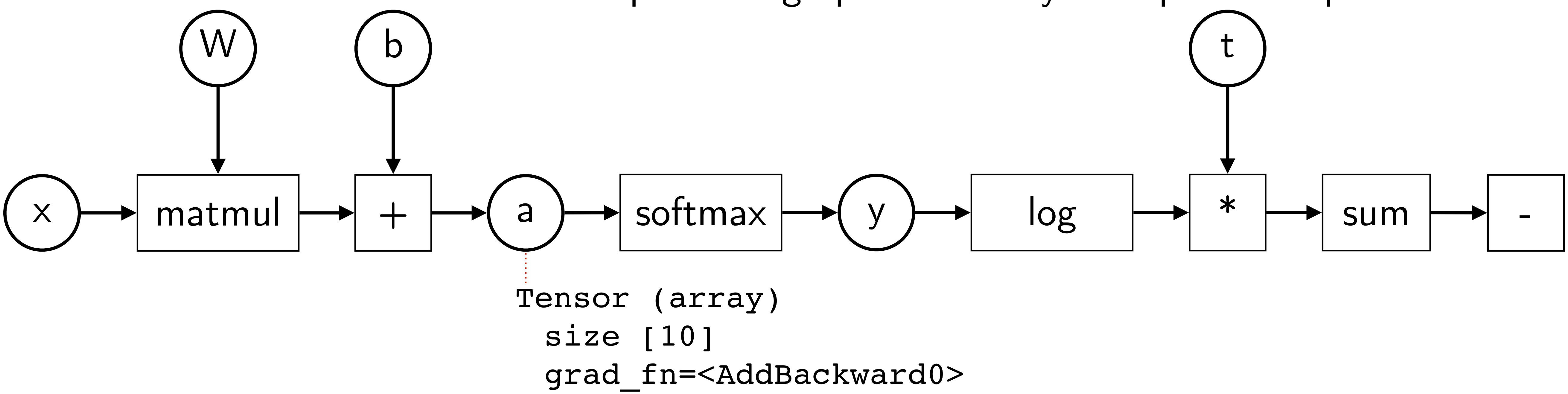
```
from torch.nn import Parameter
import torch.nn.functional as F

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

Perform some operations

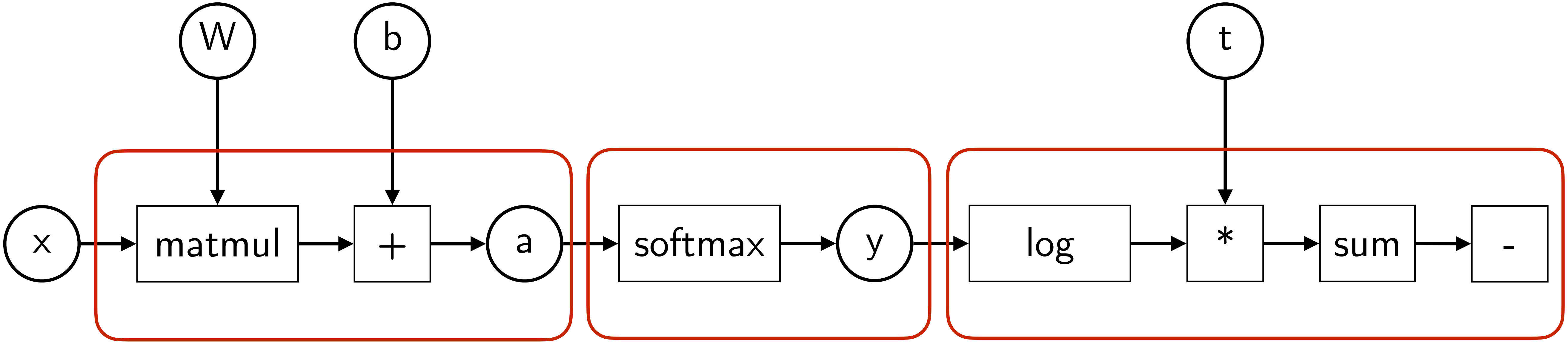
```
a = W.matmul(x) + b
y = F.softmax(a)
loss = -(t * y.log()).sum()
```

Computation graph defined by the operations performed:



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

Backward Propagation



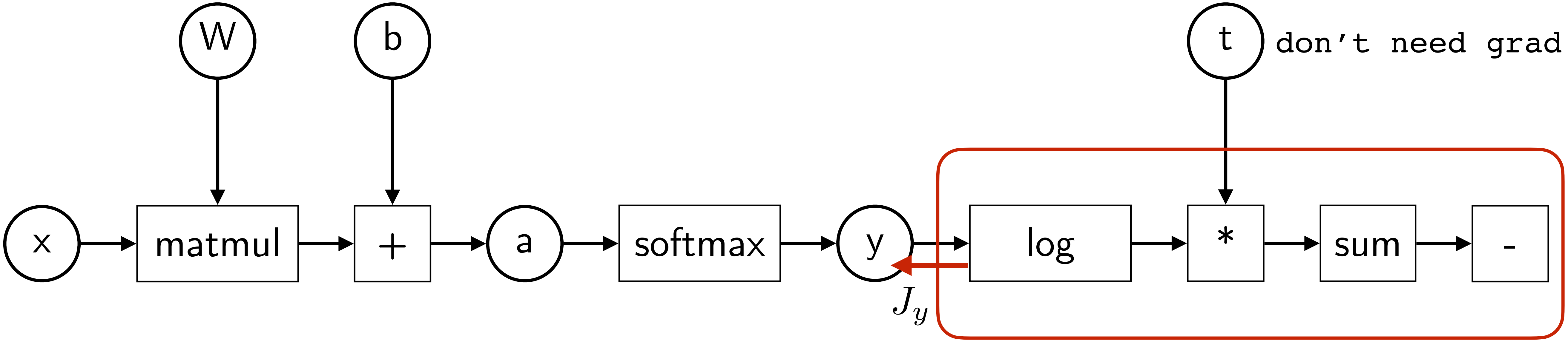
$a = \mathrm{linear}(x, W, b)$ $\quad y = \mathrm{softmax}(a)$ $\quad L = -t^{\mathsf{T}} \log(y)$

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

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

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

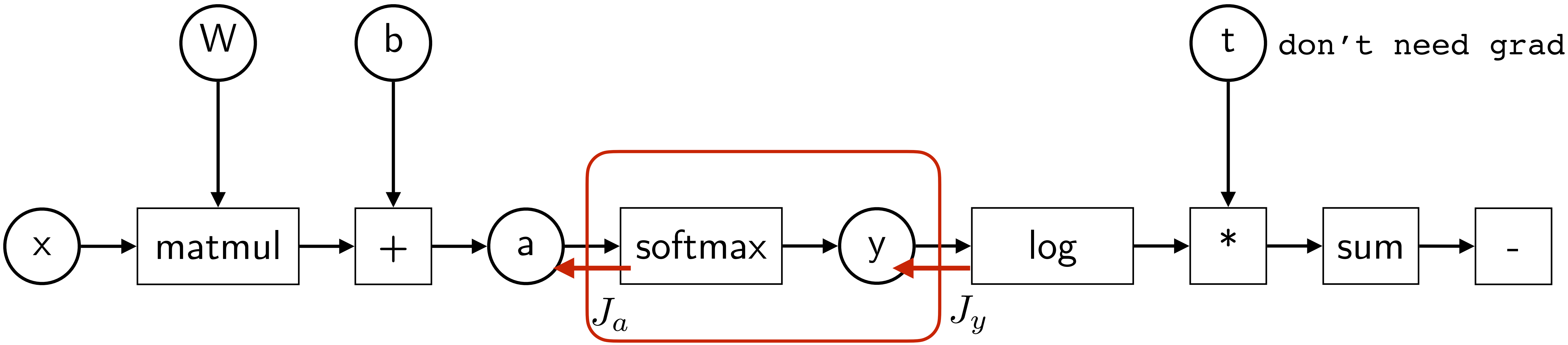
Backward Propagation



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

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

Backward Propagation



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

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

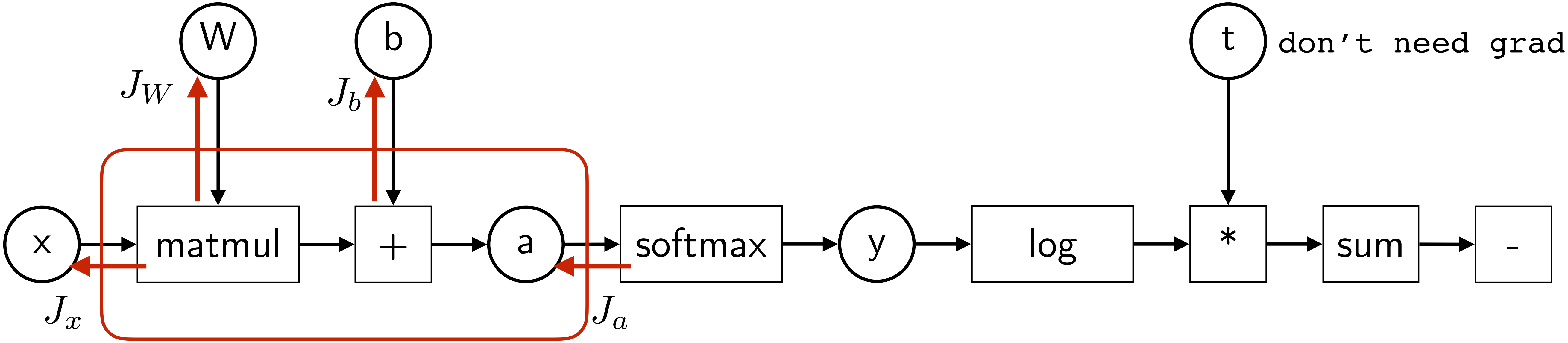
$$= \sum_j J_{y_j} (y_i [\![i{=}j]\!] - y_i y_j) = y_i \Big(J_{y_i} - \sum_j y_j J_{y_j} \Big)$$

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

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

Notice: forward and backward are both linear complexity

Backward Propagation



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

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

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

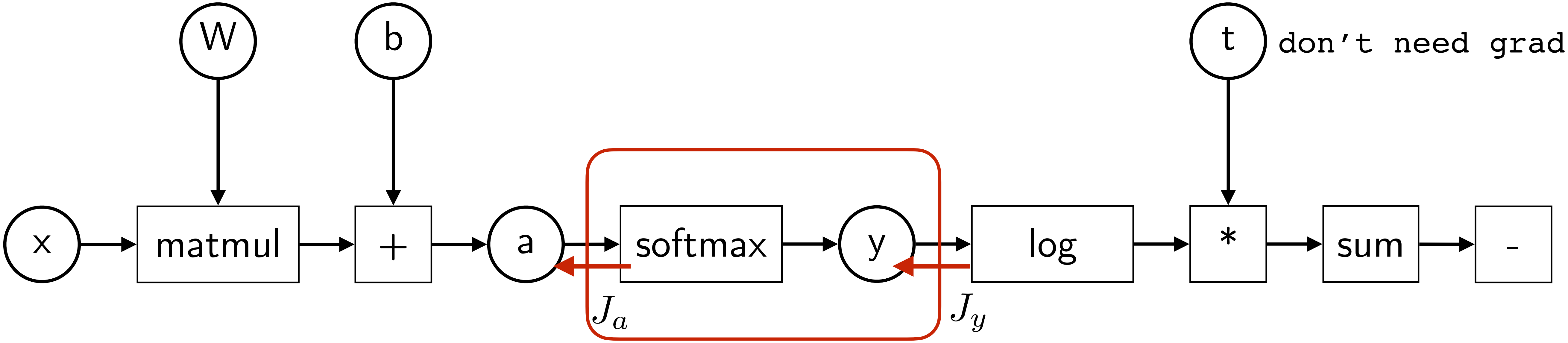
$$J_x = J_a W$$

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

Note: a transposed product in comparison with Wx

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

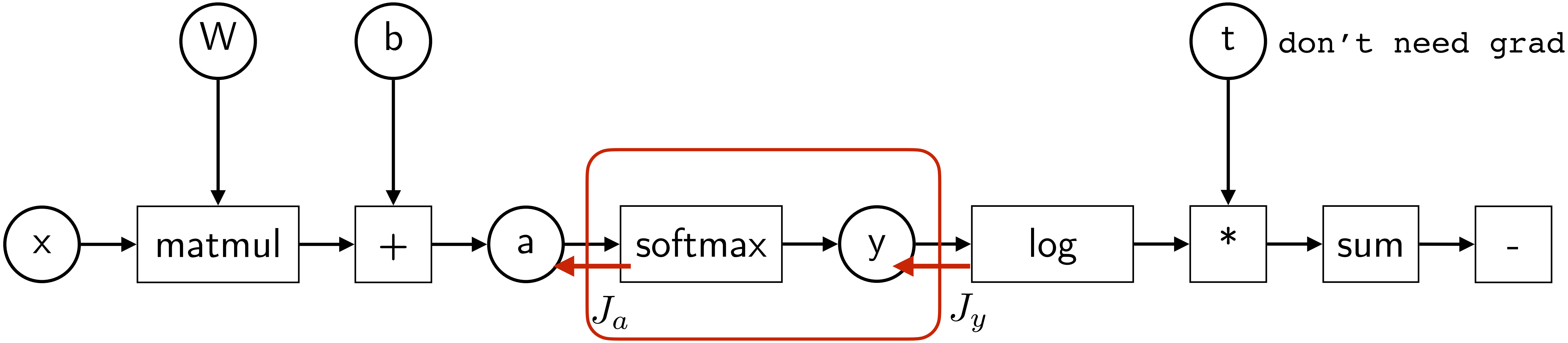
Backward Propagation



✦ 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]
  - special efficient implementation for forward
  - special efficient implementation for backward (transposed convolution)

Backward Propagation



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

$$J_a = J_y(\text{Diag}(y) - yy^\top) = J_y \odot y - (J_y y)y^\top$$

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)
```

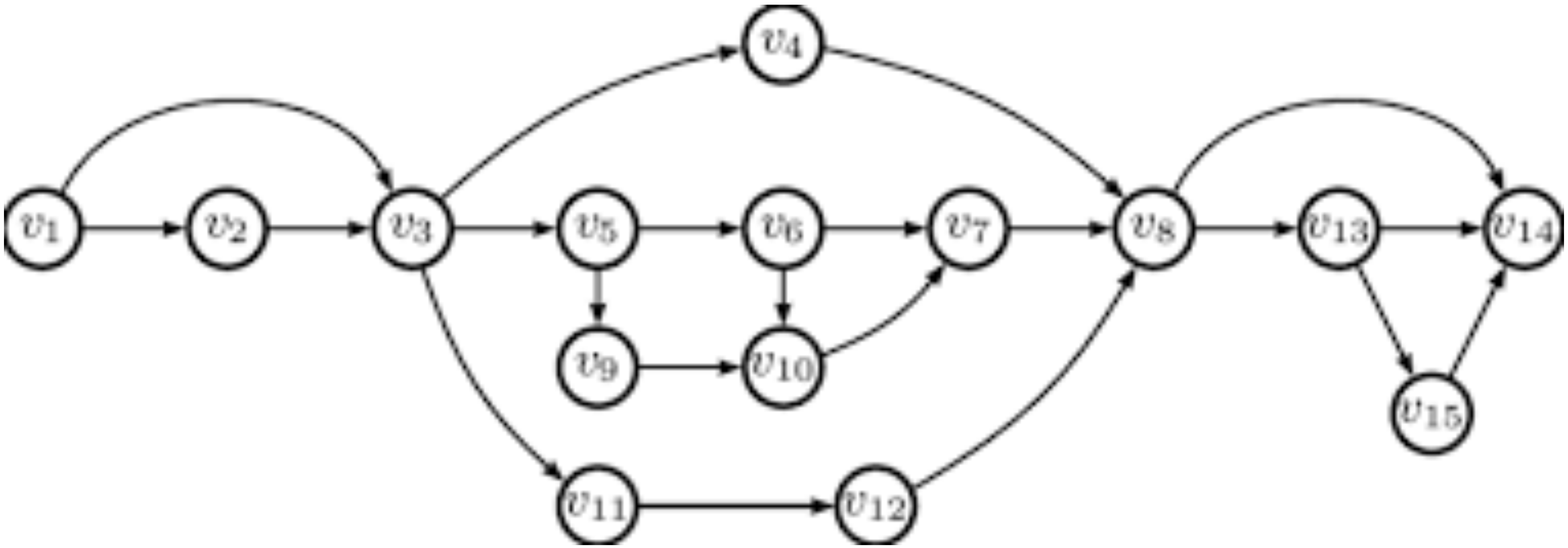
3)
```
class MySoftmax(torch.autograd.Function):
    @staticmethod
    def forward(ctx, a):
        y = a.exp()
        y /= y.sum()
        ctx.save_for_backward(y)
        return y

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

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

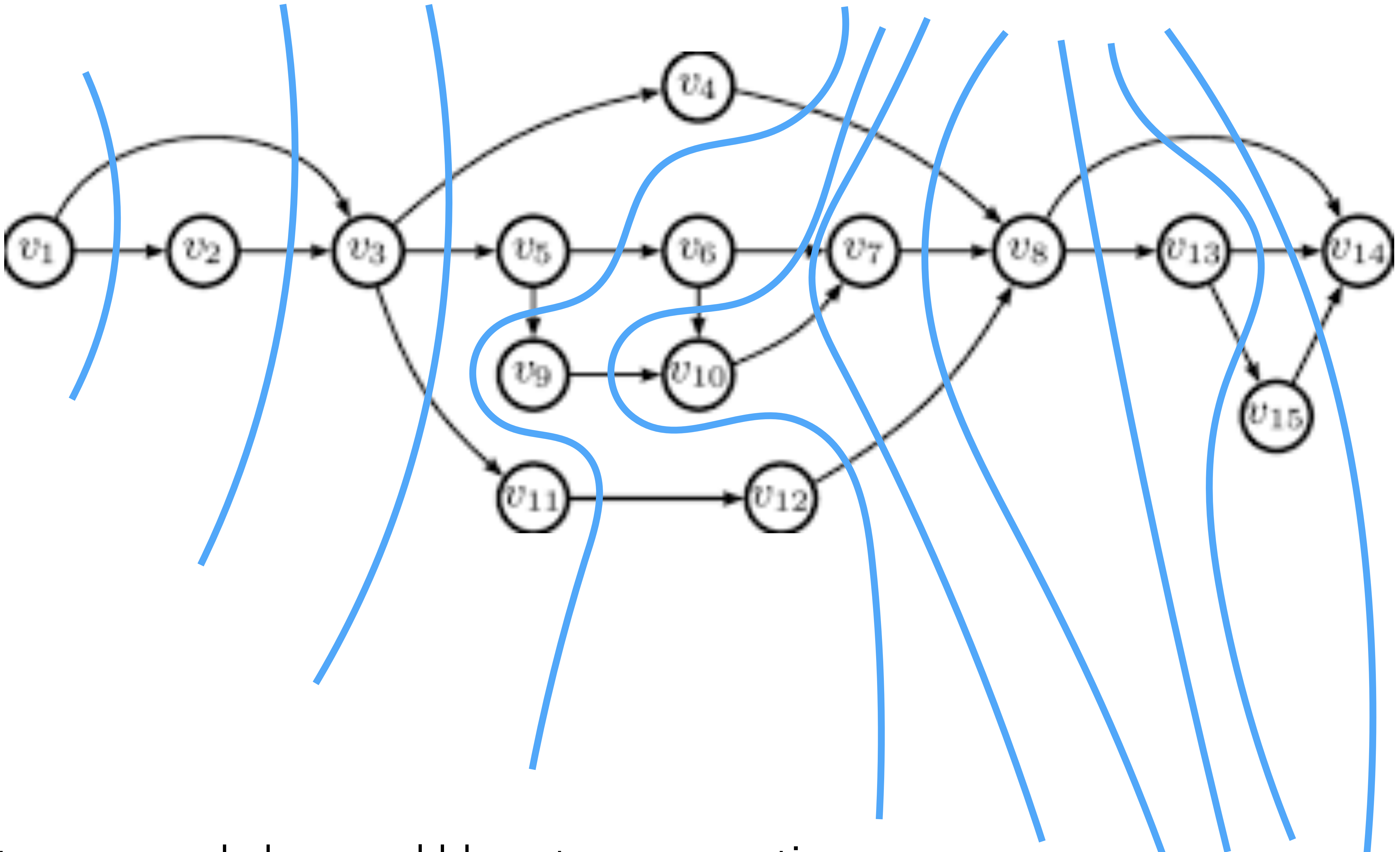
General DAG

- ✦ 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