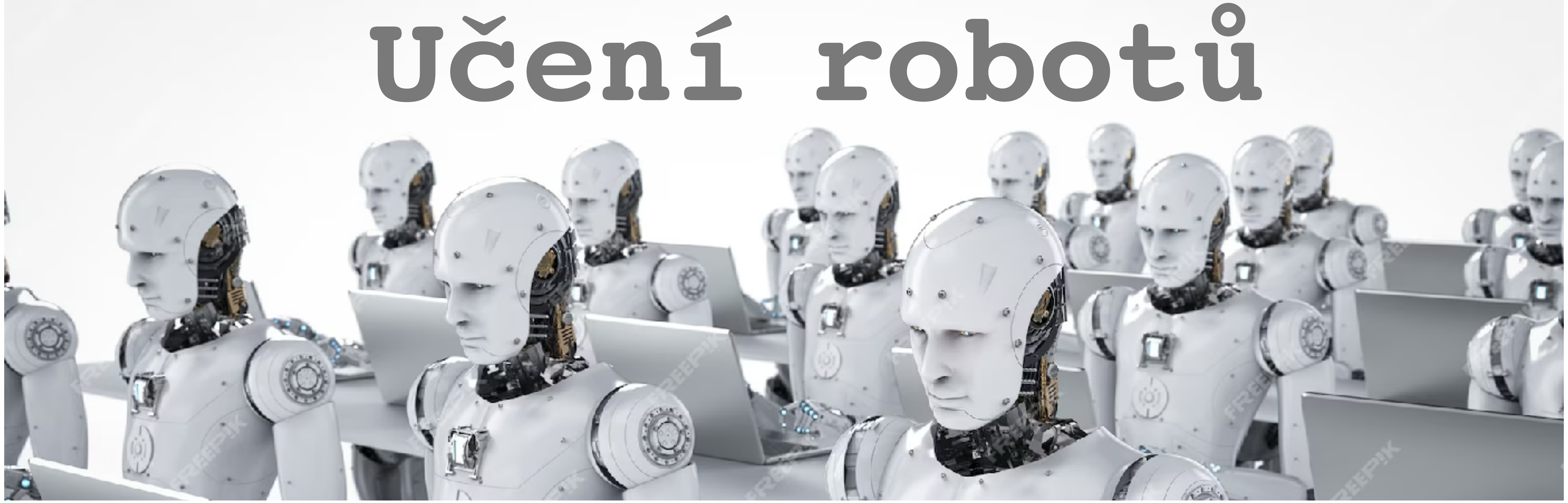


Under the hood of autograd

Neurons, fully connected networks, computational graphs, Jacobians and vector-Jacobian product

Karel Zimmermann
Czech Technical University in Prague
Faculty of Electrical Engineering, Department of Cybernetics

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy

Linear classifier and neuron

Labels

RGB images

$+1$

-1

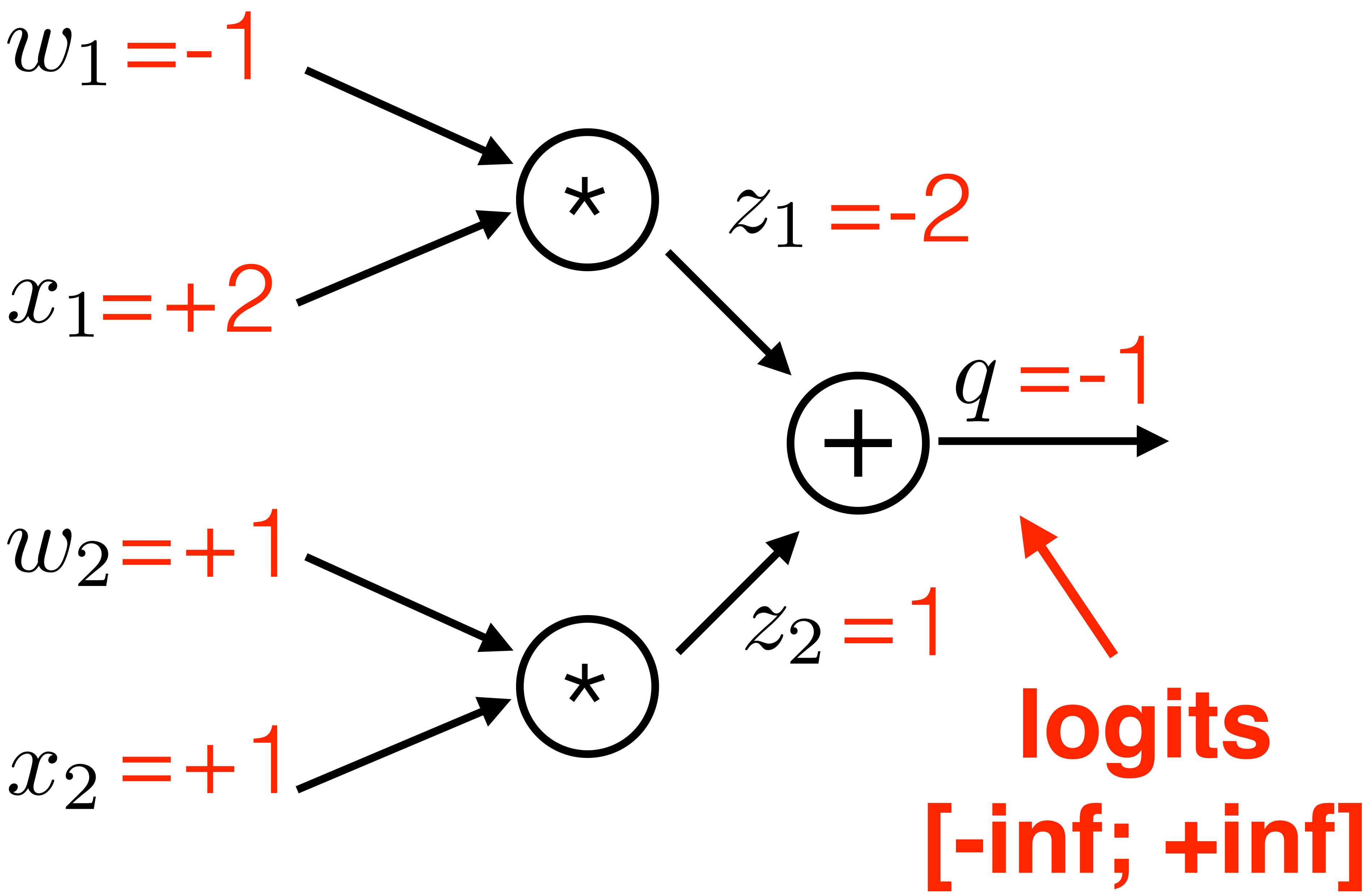
def classify():

 # Linear classifier

$\mathbf{x} = \mathrm{vec}($  $)$

return $\mathbf{w}^\top \mathbf{x}$

Computational graph of linear classifier



Linear classifier and neuron

Labels | RGB images

$+1$

-1

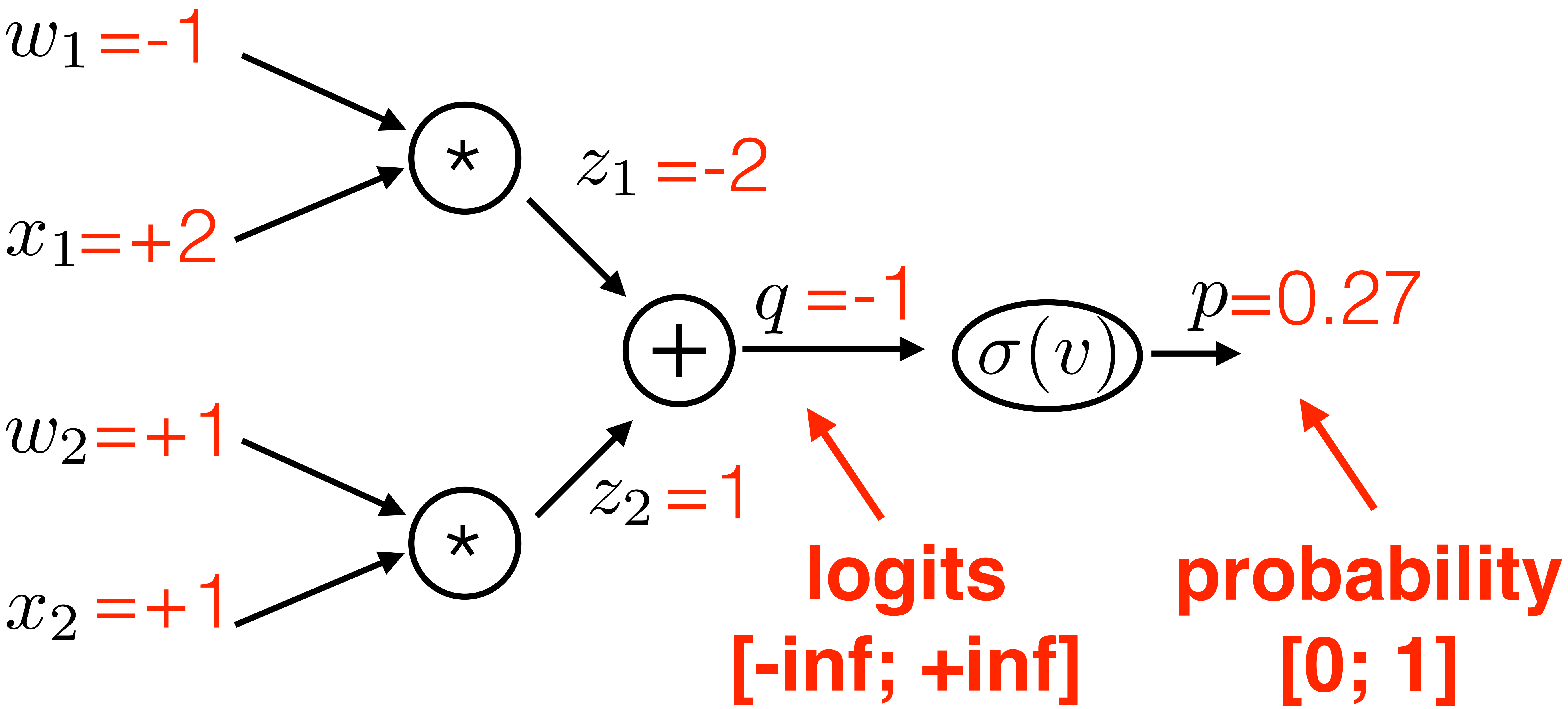
```
def classify(
```


```
):
    # Neuron
```
$\mathbf{x} = \mathrm{vec}($  $)$

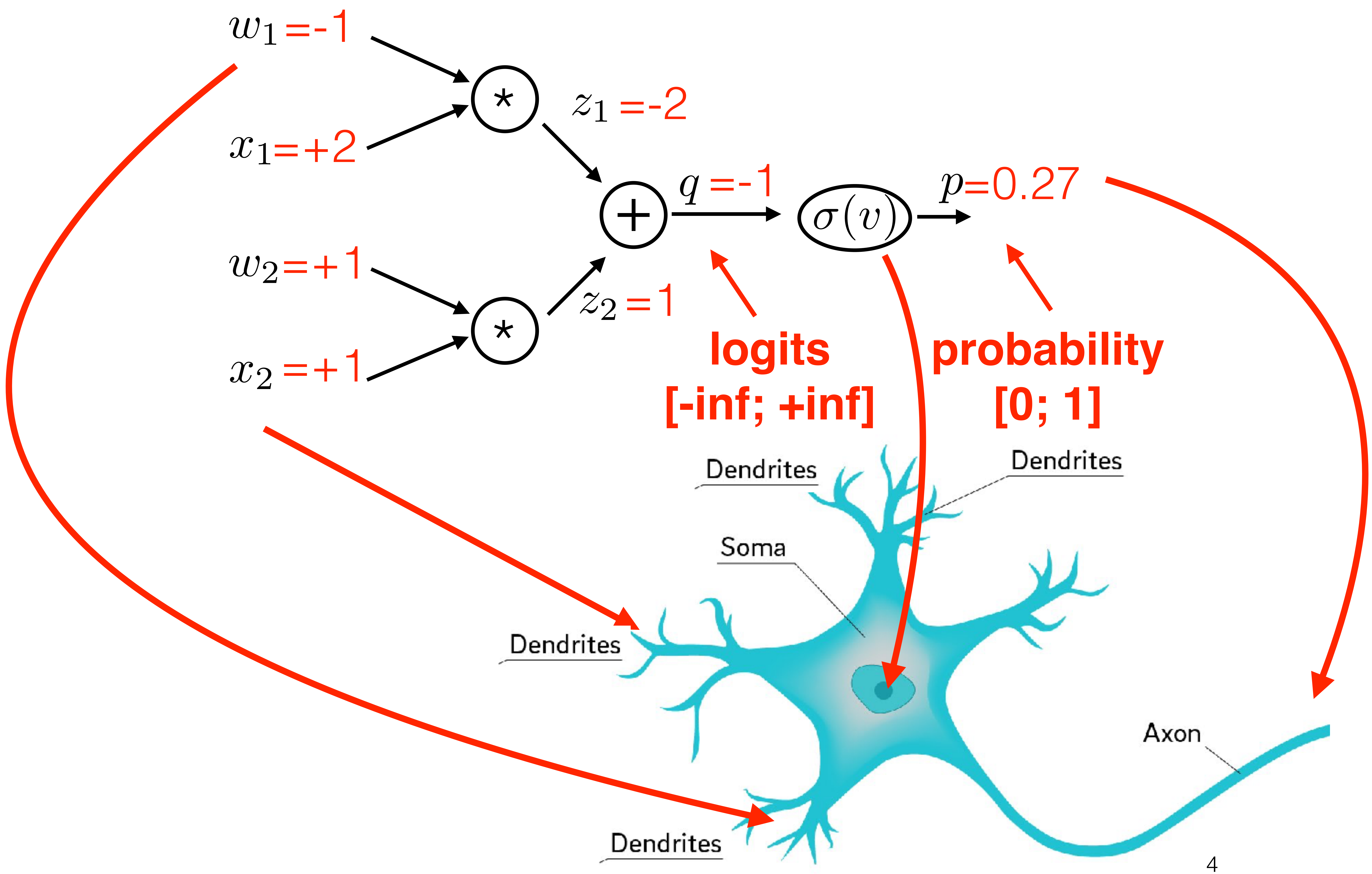
$p = \sigma\left(\mathbf{w}^\top \mathbf{x}\right)$

return p

Computational graph of neuron

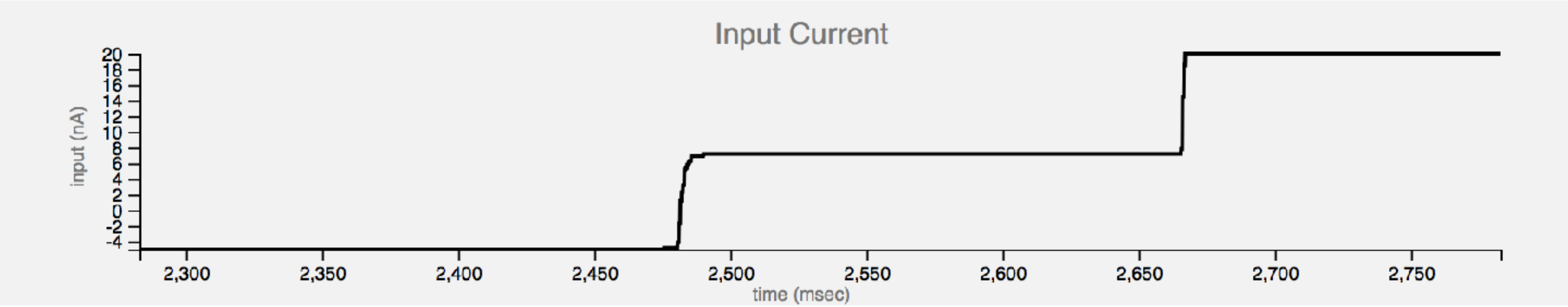


Relation to biological neuron

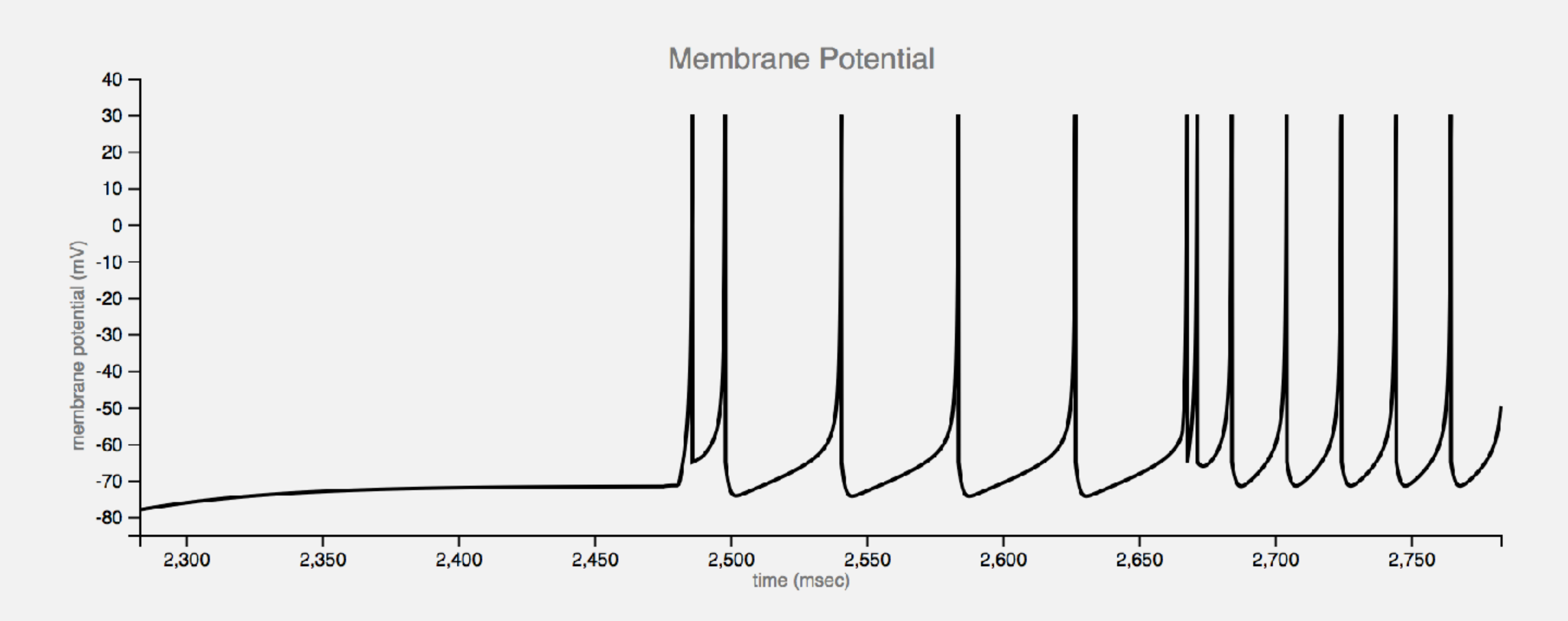


Modeling dynamic neuron behaviour

Input:

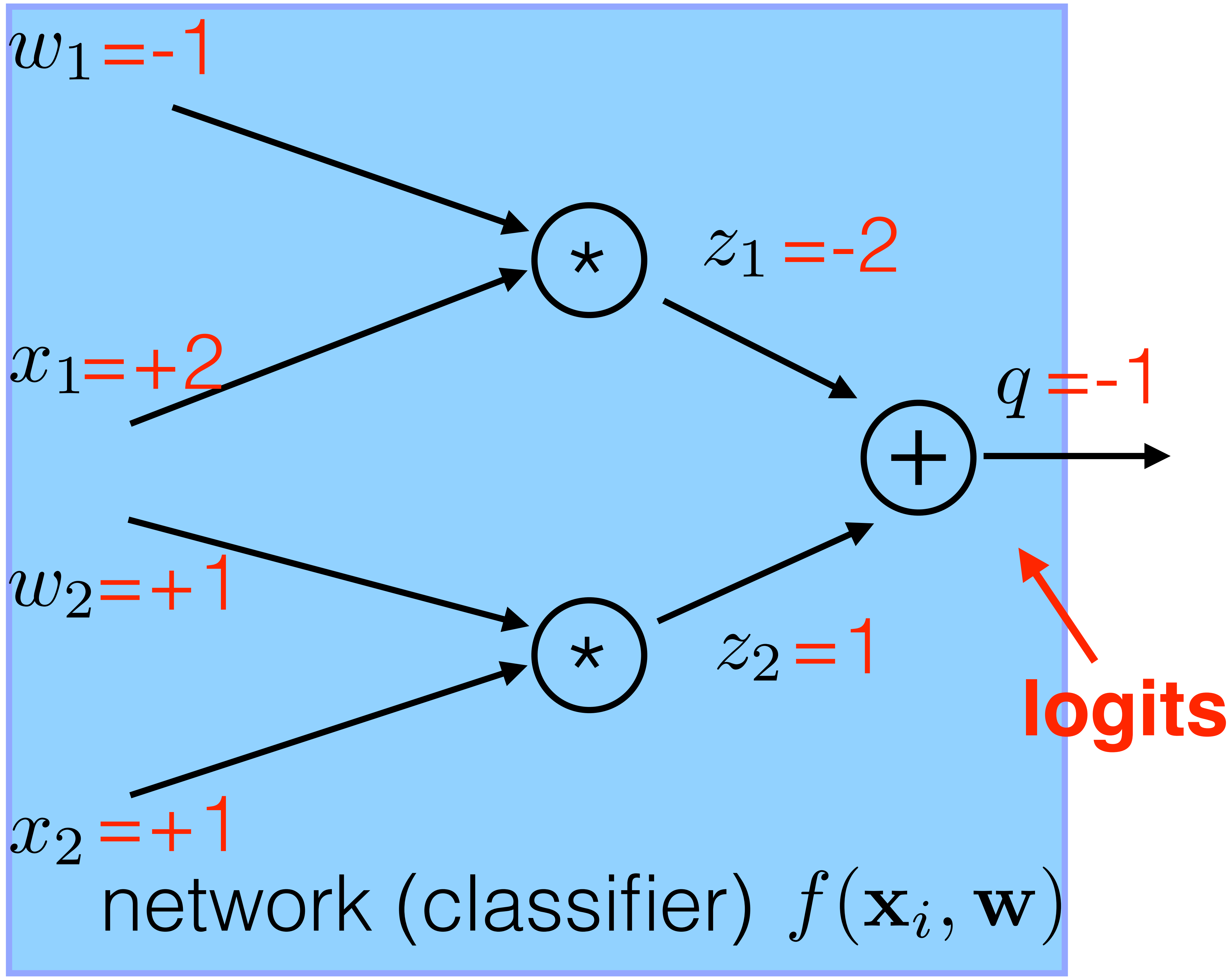


Output:

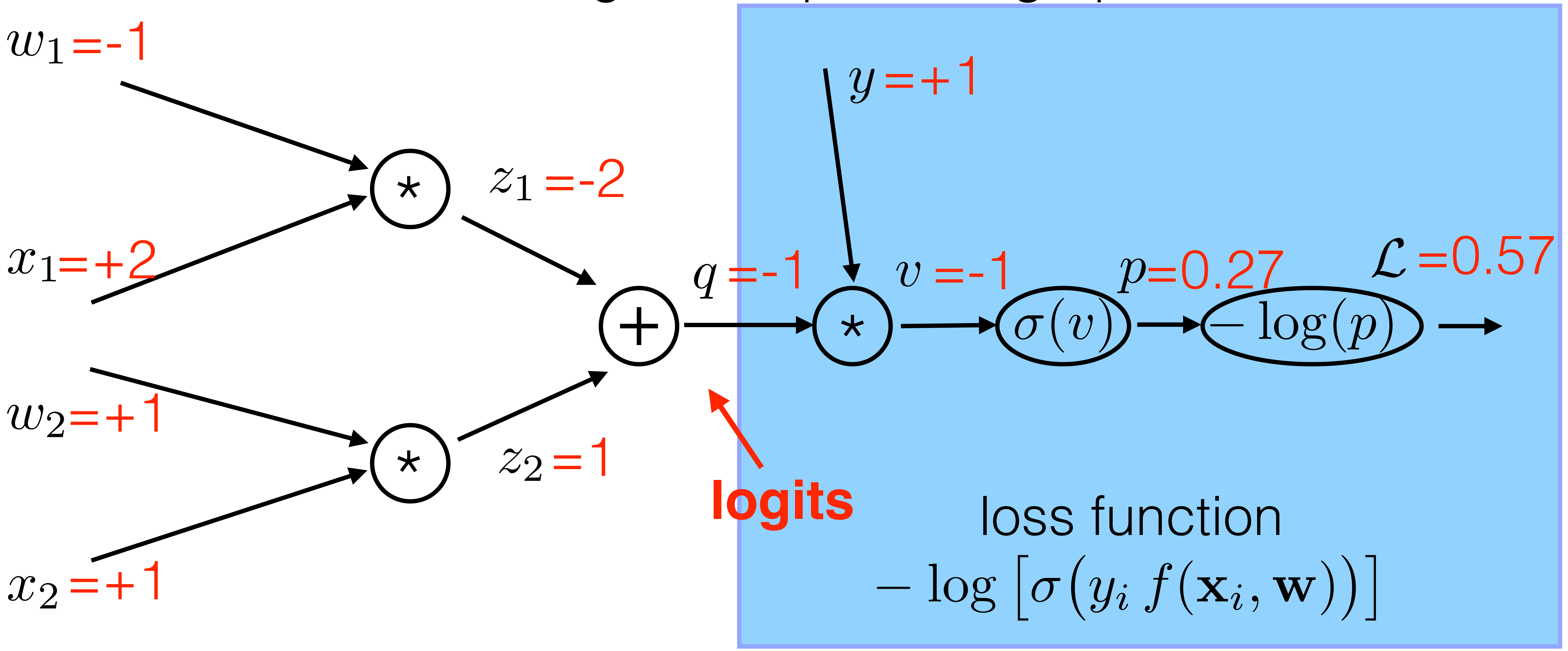


http://jackterwilliger.com/biological-neural-networks-part-i-spiking-neurons/

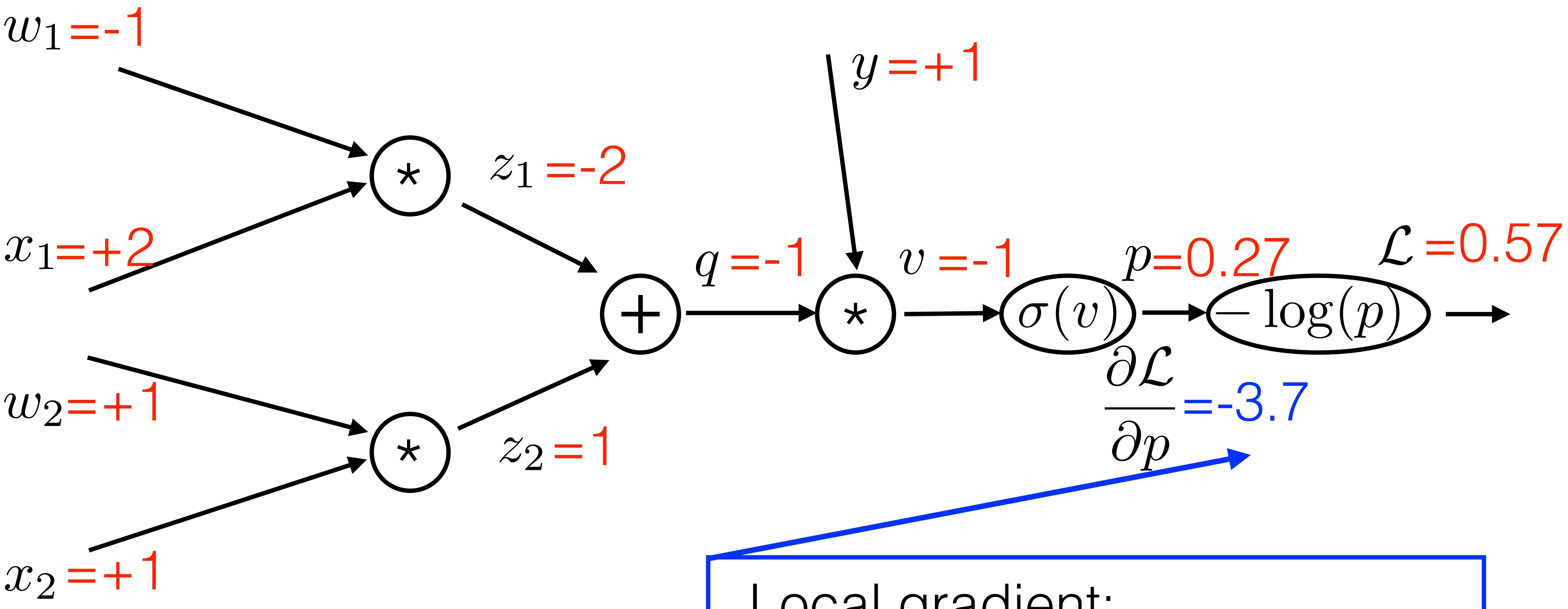
Learning in computation graph



Learning in computation graph



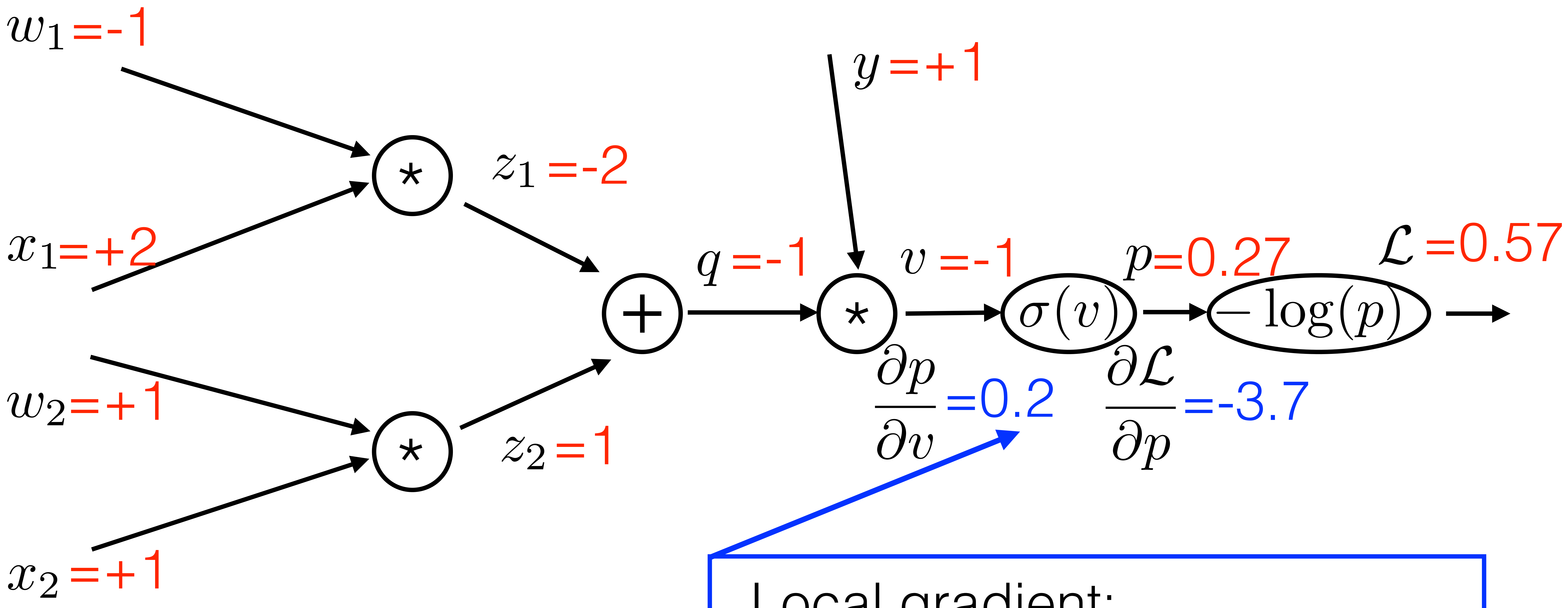
Learning in computation graph



Local gradient:

$$\frac{\partial \mathcal{L}}{\partial p} = \frac{\partial(-\log(p))}{\partial p} = -\frac{1}{p}$$

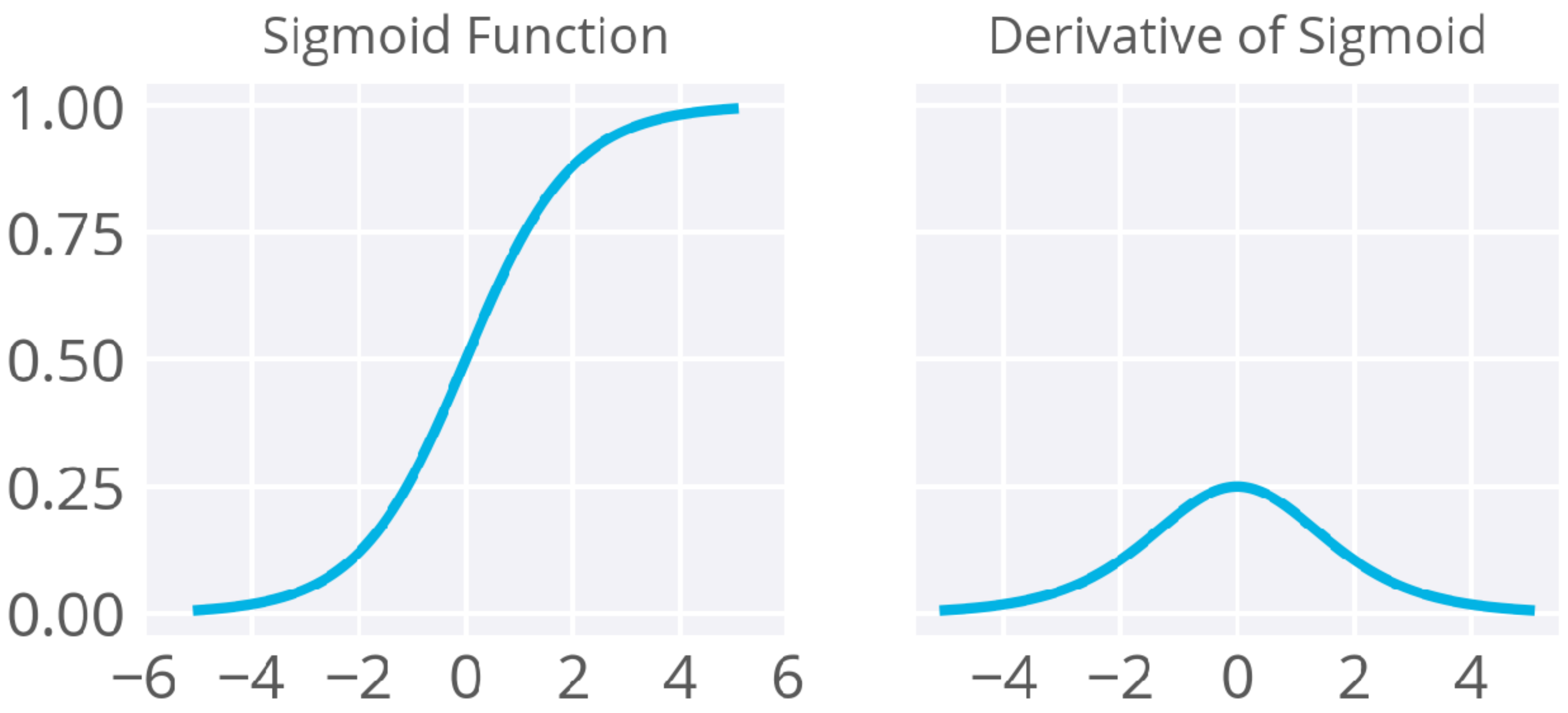
Learning in computation graph



Local gradient:

$$\frac{\partial p}{\partial v} = \frac{\partial \sigma(v)}{\partial v} = \sigma(v)(1-\sigma(v))$$

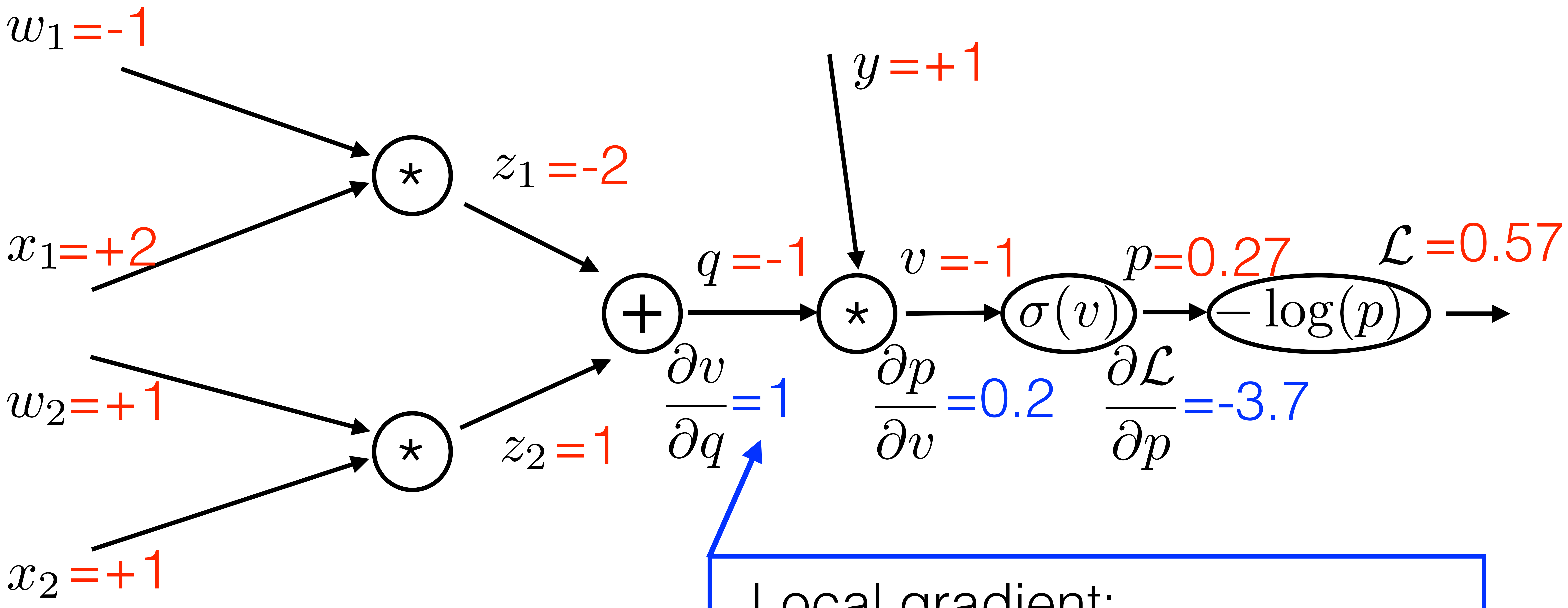
Learning in computation graph



Local gradient:

$$\frac{\partial p}{\partial v} = \frac{\partial \sigma(v)}{\partial v} = \sigma(v)(1 - \sigma(v))$$

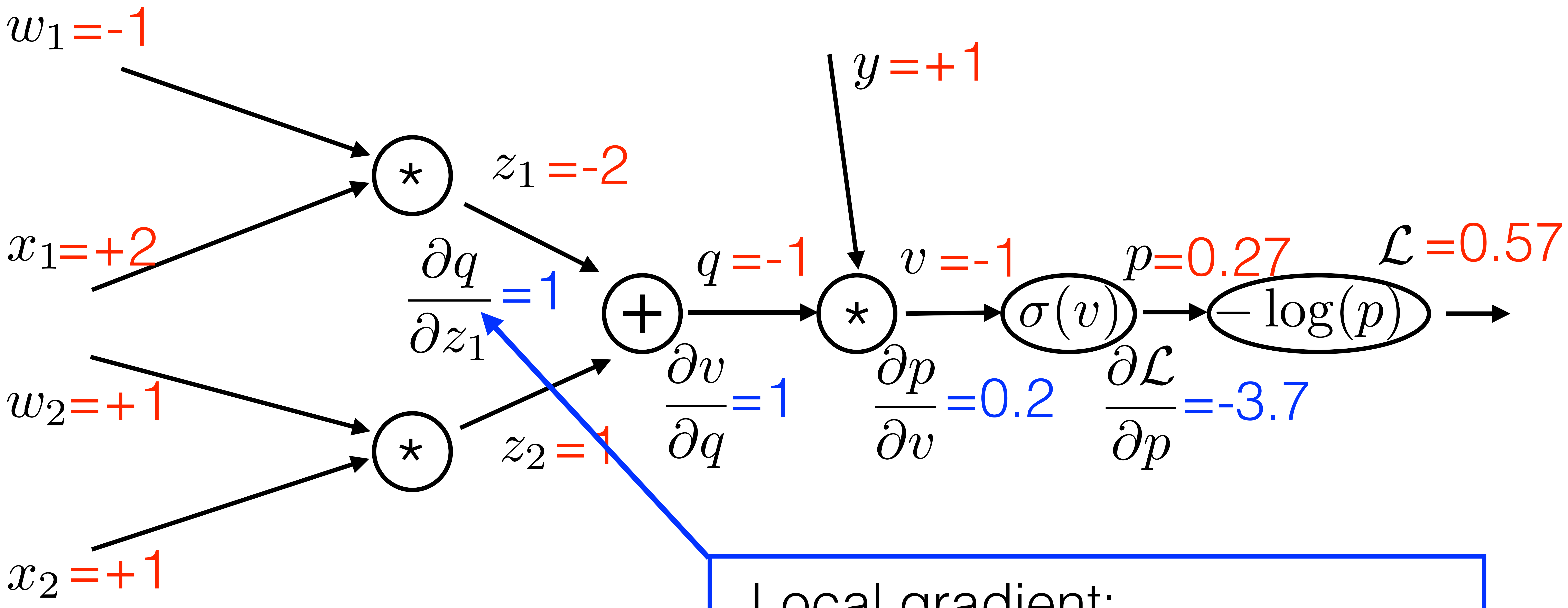
Learning in computation graph



Local gradient:

$$\frac{\partial v}{\partial q} = \frac{\partial (yq)}{\partial q} = y$$

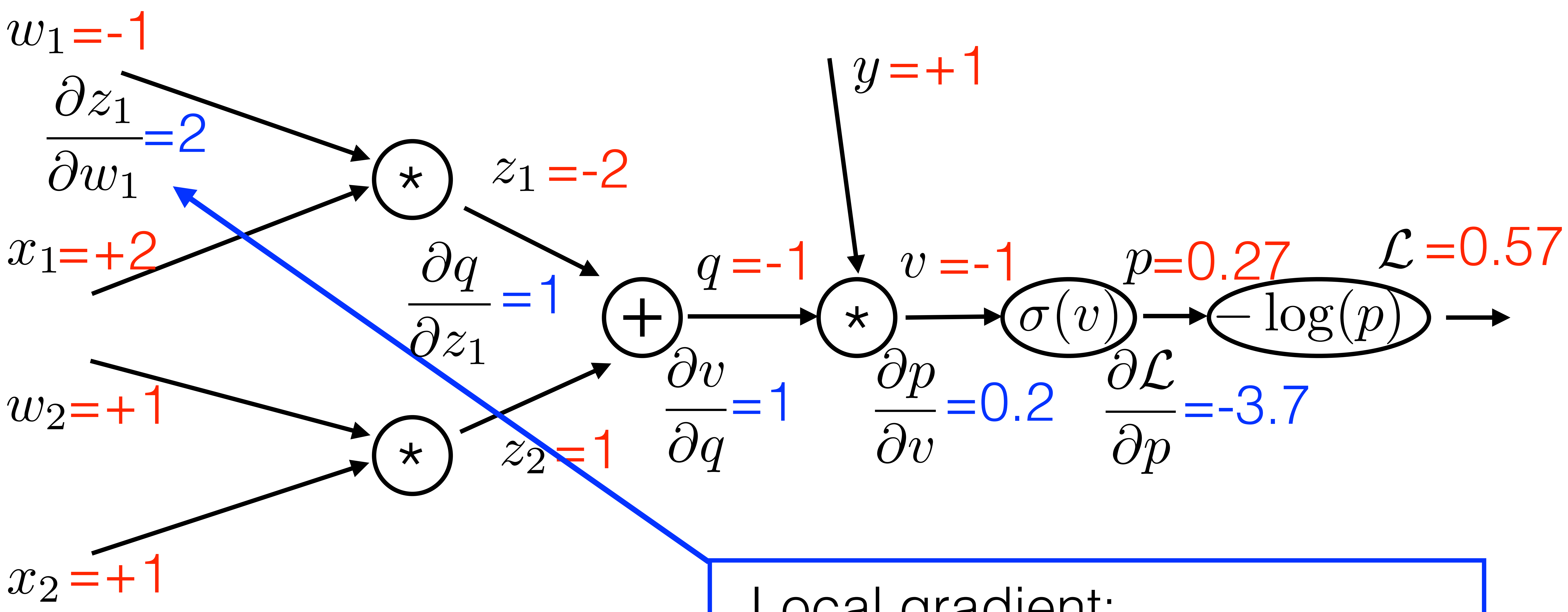
Learning in computation graph



Local gradient:

$$\frac{\partial q}{\partial z_1} = \frac{\partial (z_1 + z_2)}{\partial z_1} = 1$$

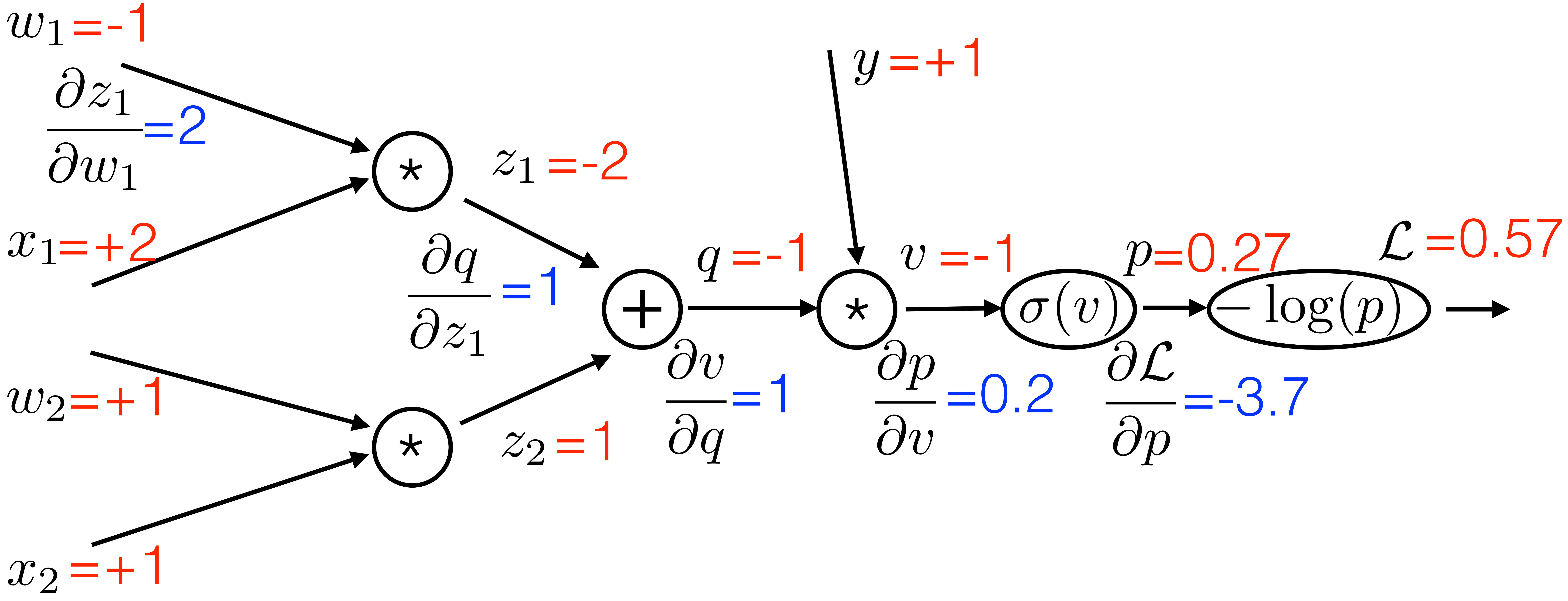
Learning in computation graph



Local gradient:

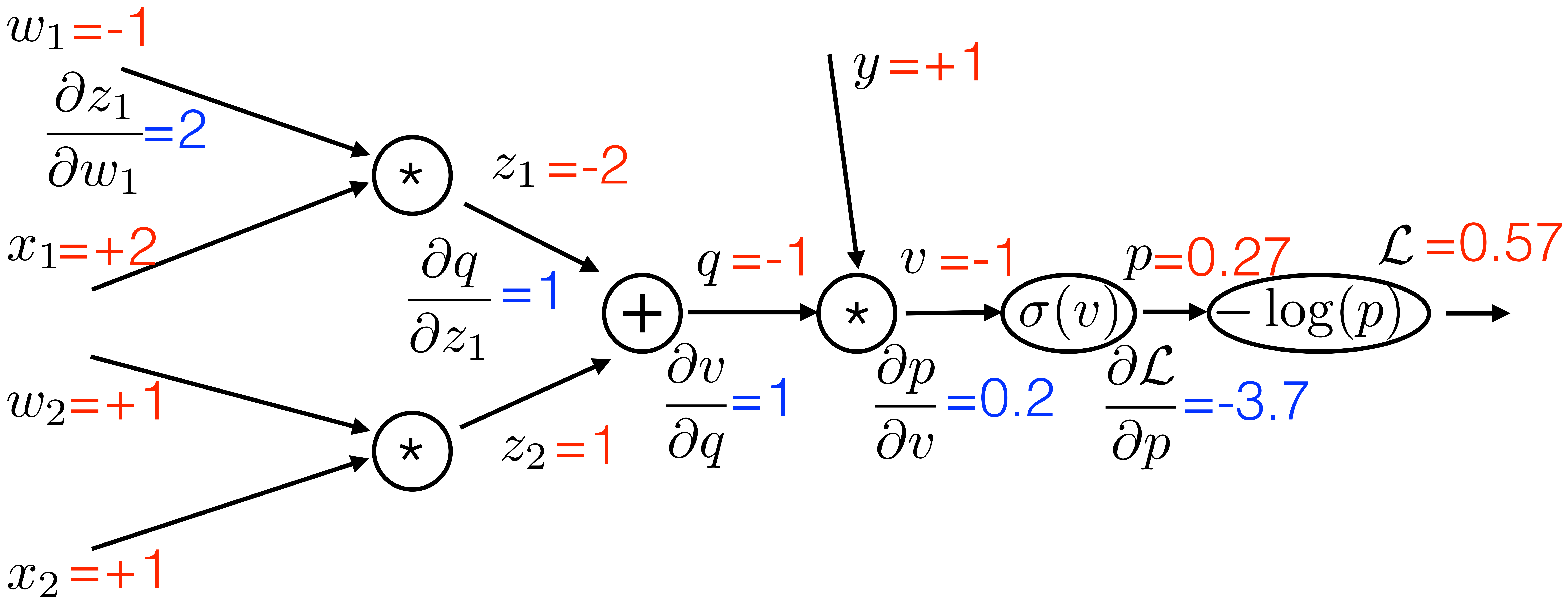
$$\frac{\partial z_1}{\partial w_1} = \frac{\partial (w_1 x_1)}{\partial w_1} = x_1$$

Learning in computation graph



$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial p}\frac{\partial p}{\partial v}\frac{\partial v}{\partial q}\frac{\partial q}{\partial z_1}\frac{\partial z_1}{\partial w_1}$$ =-3.7*0.2*1*1*2 =-1.48

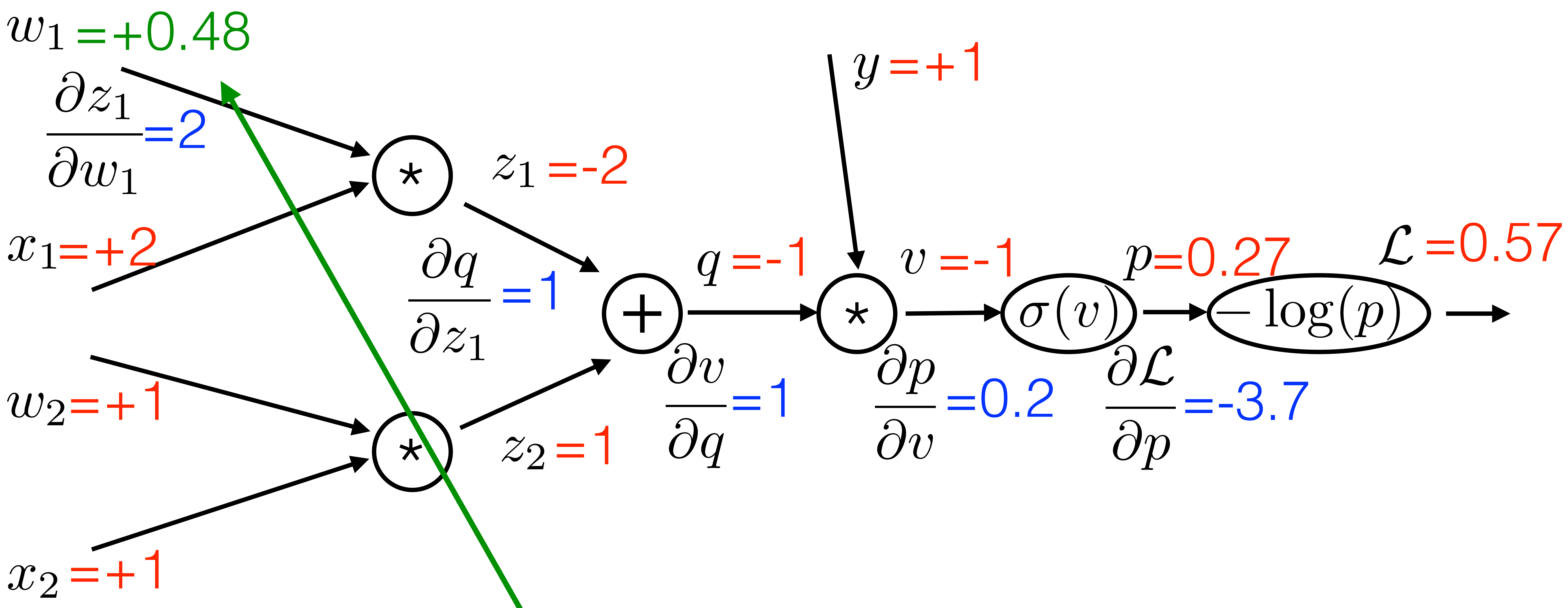
Learning in computation graph



$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial p}\frac{\partial p}{\partial v}\frac{\partial v}{\partial q}\frac{\partial q}{\partial z_1}\frac{\partial z_1}{\partial w_1} = -3.7*0.2*1*1*2 = -1.48$$

$$w_1 = w_1 - \alpha\frac{\partial \mathcal{L}}{\partial w1} \quad = +0.48$$

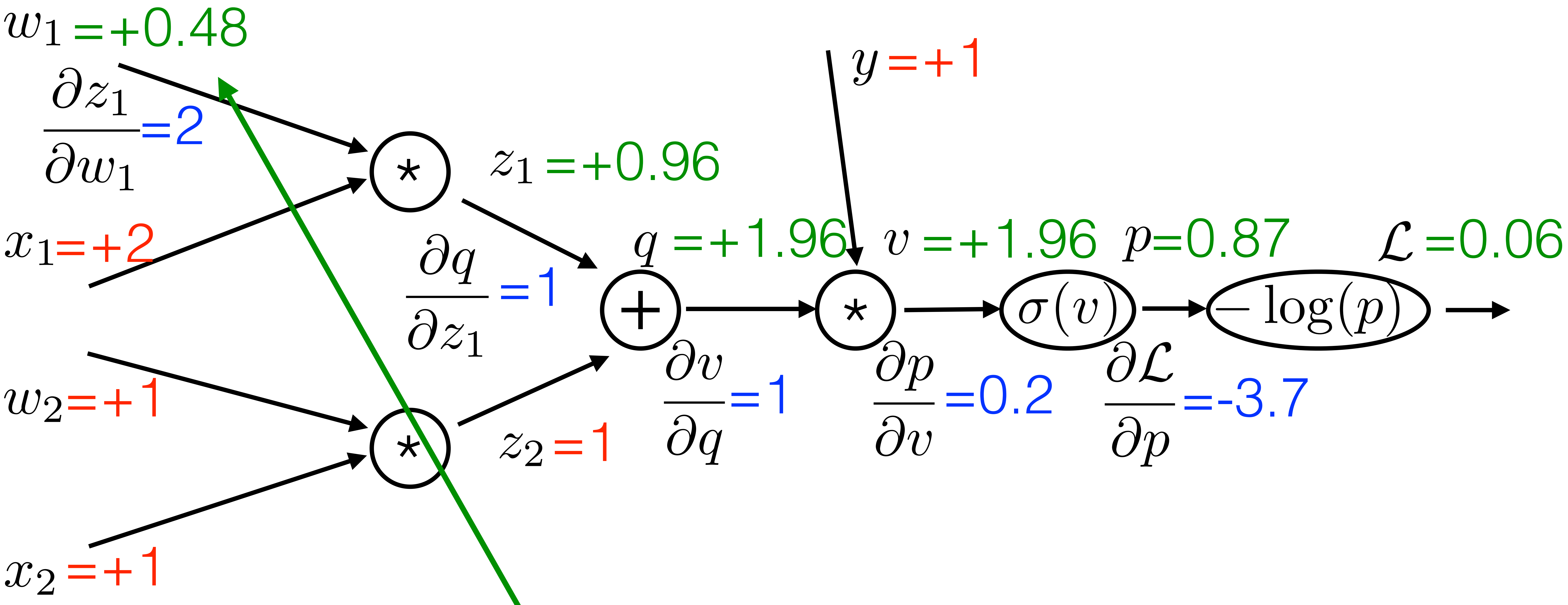
Learning in computation graph



$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial p}\frac{\partial p}{\partial v}\frac{\partial v}{\partial q}\frac{\partial q}{\partial z_1}\frac{\partial z_1}{\partial w_1} = -3.7*0.2*1*1*2 = -1.48$$

$$w_1 = w_1 - \alpha\frac{\partial \mathcal{L}}{\partial w1} = +0.48$$

Learning in computation graph



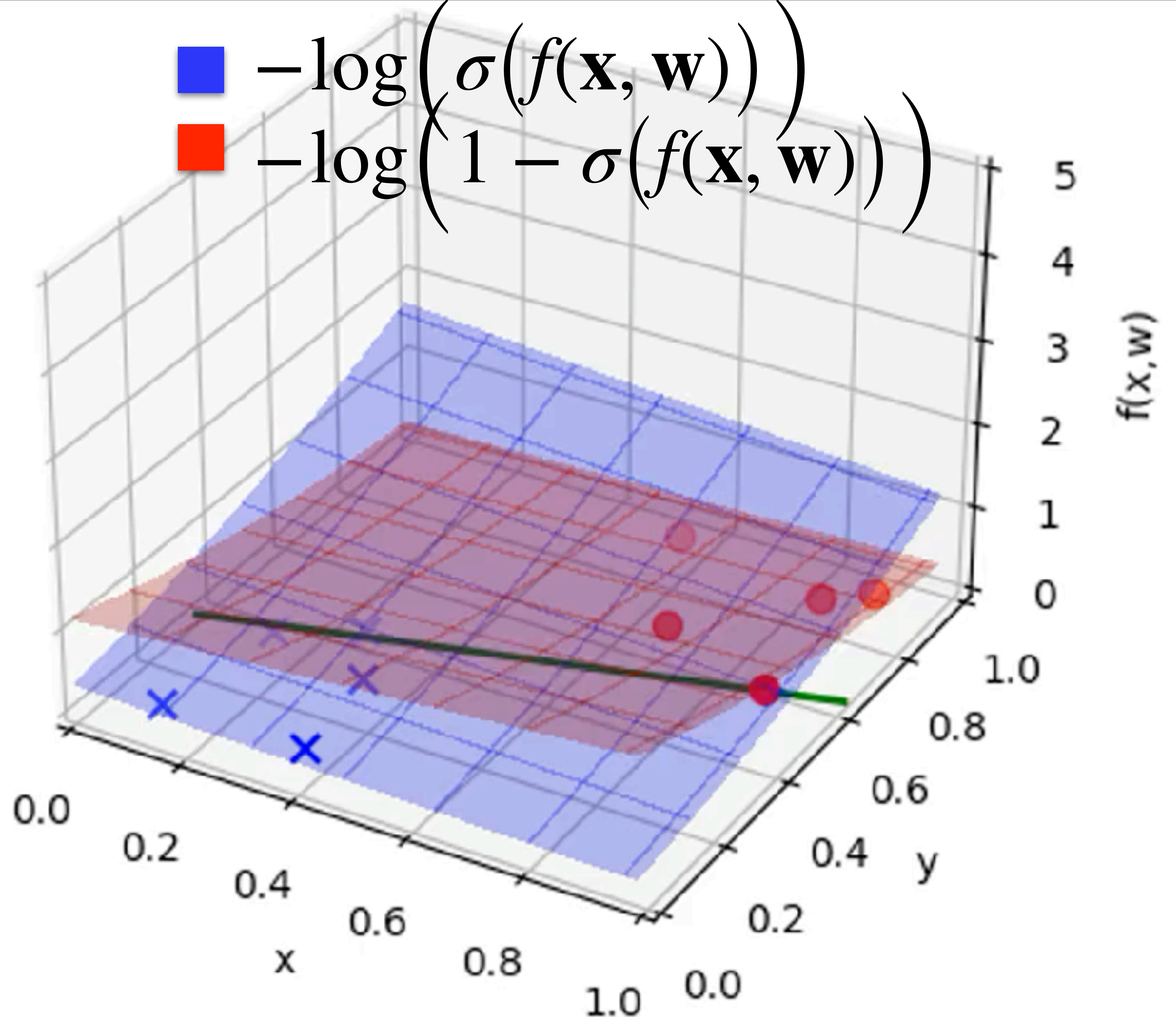
$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial p}\frac{\partial p}{\partial v}\frac{\partial v}{\partial q}\frac{\partial q}{\partial z_1}\frac{\partial z_1}{\partial w_1}$$ =-3.7*0.2*1*1*2 =-1.48

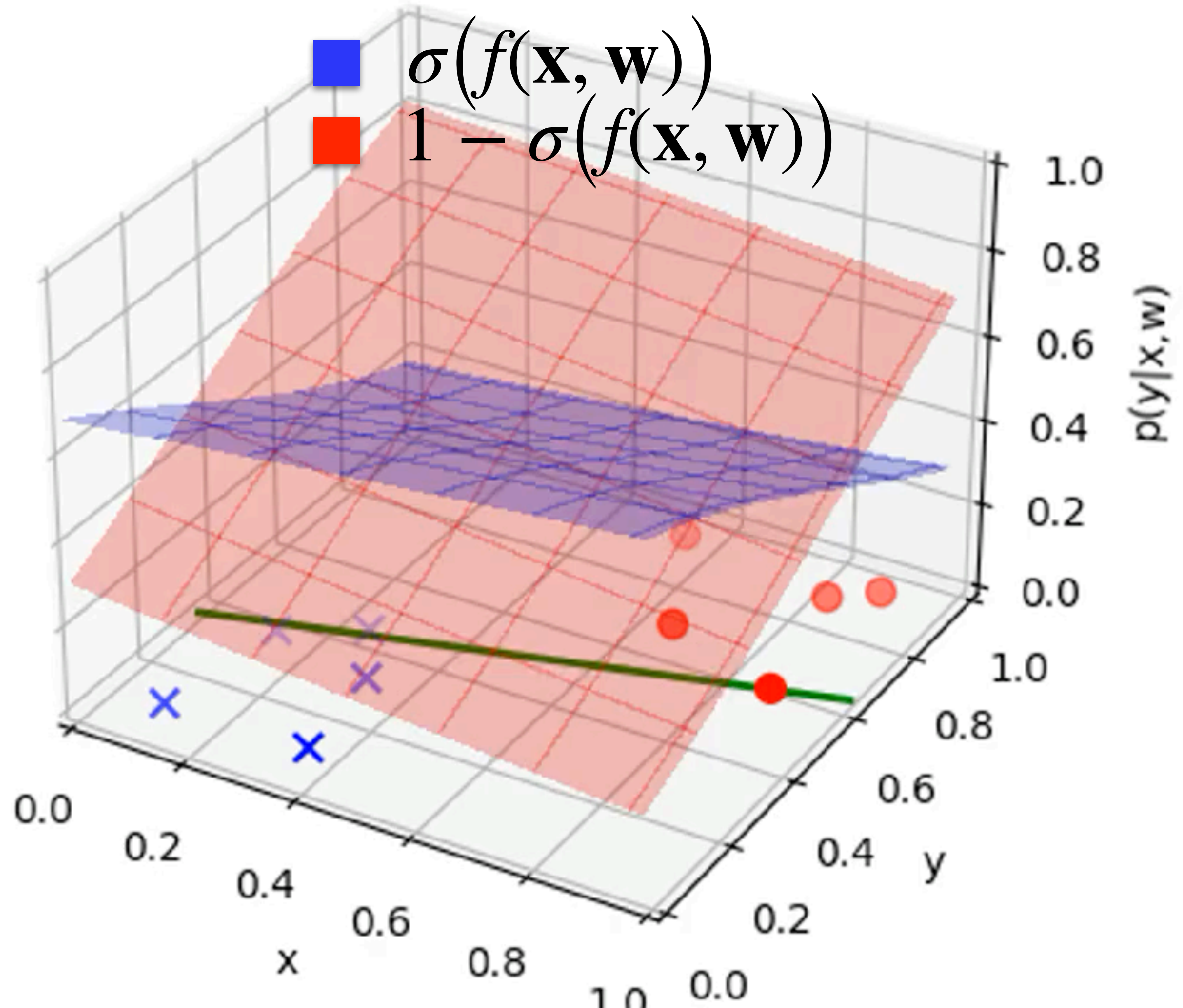
$$w_1 = w_1 - \alpha\frac{\partial \mathcal{L}}{\partial w1}$$ = +0.48

Learning from multiple training samples means summing up the partial derivative over all samples

Numpy implementation

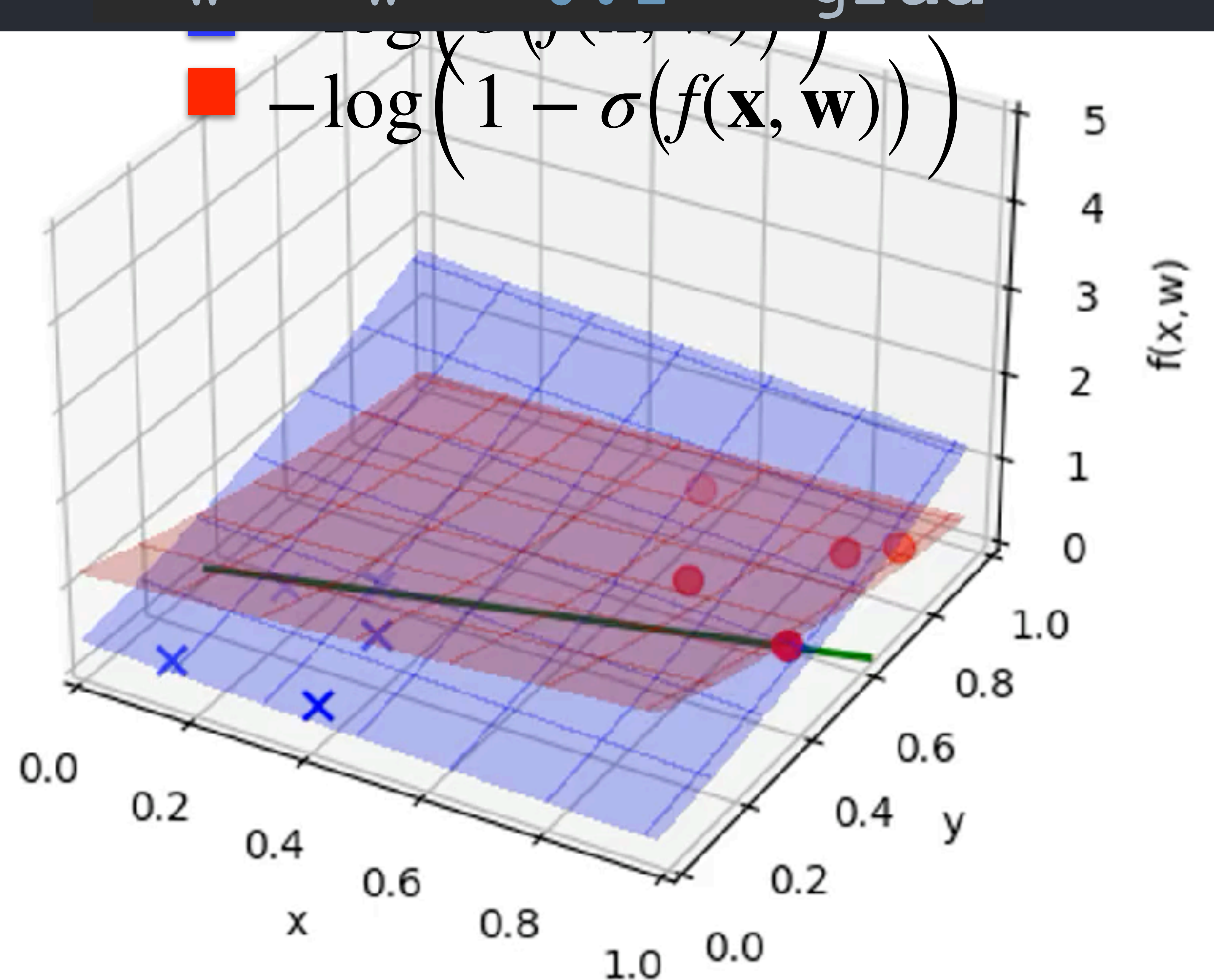
```
for i in range(30):
    u = w[0] * x[:, 0] + w[1] * x[:, 1] + w[2]
    p = sigmoid(u)
    loss = (-np.log(p)*y + -np.log(1-p)*(1-y)).sum()
    grad = -1/p * sigmoid(u) * (1-sigmoid(u)) * …
    w = w - 0.1 * grad
```

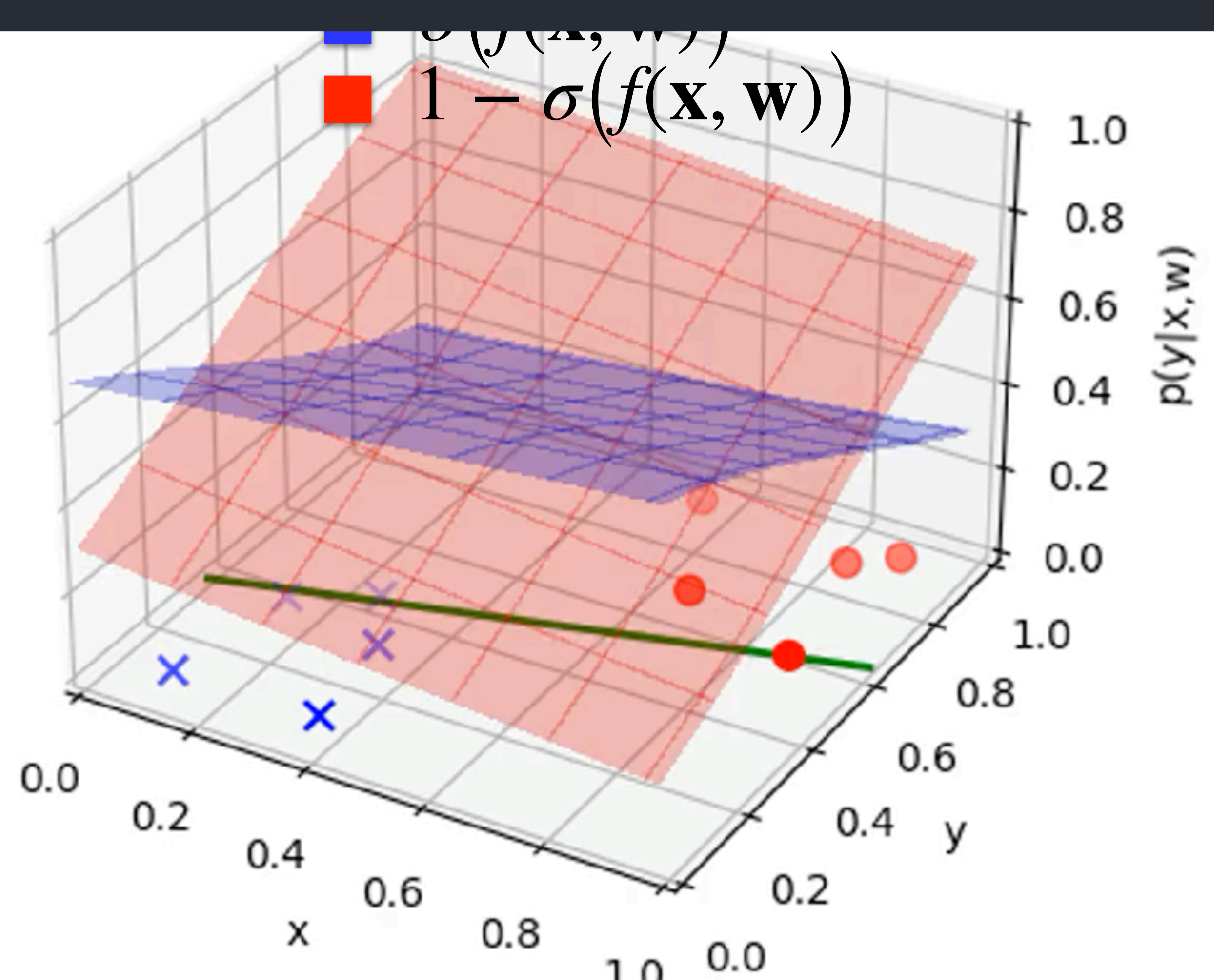




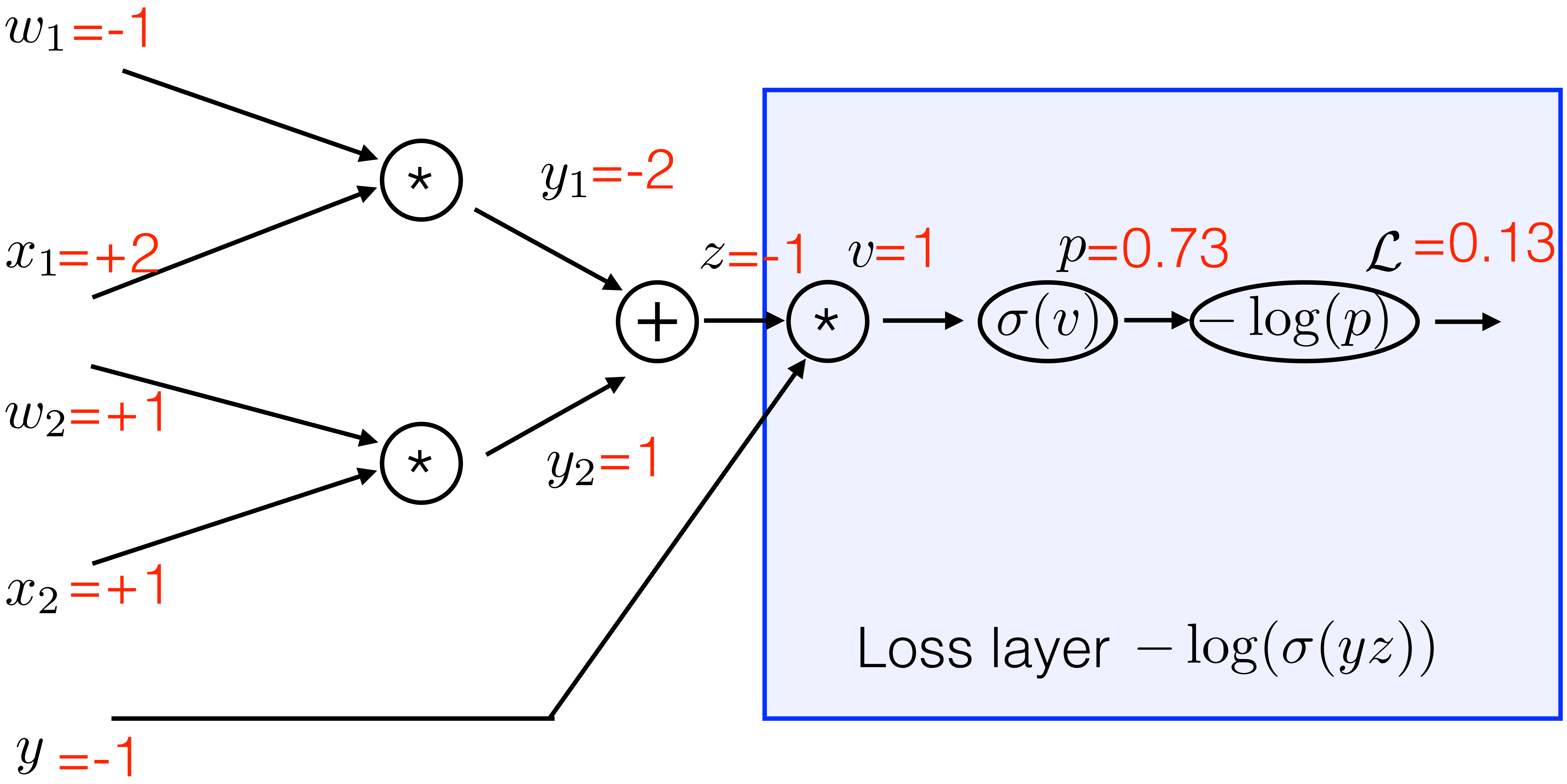
PyTorch implementation

```
for i in range(30):
    u = w[0] * x[:, 0] + w[1] * x[:, 1] + w[2]
    p = torch.sigmoid(u)
    loss = (-torch.log(p) * y - torch.log(1 - p) * (1 - y)).sum()
    grad, = torch.autograd.grad(loss, w)
    with torch.no_grad():
        w = w - 0.1 * grad
```

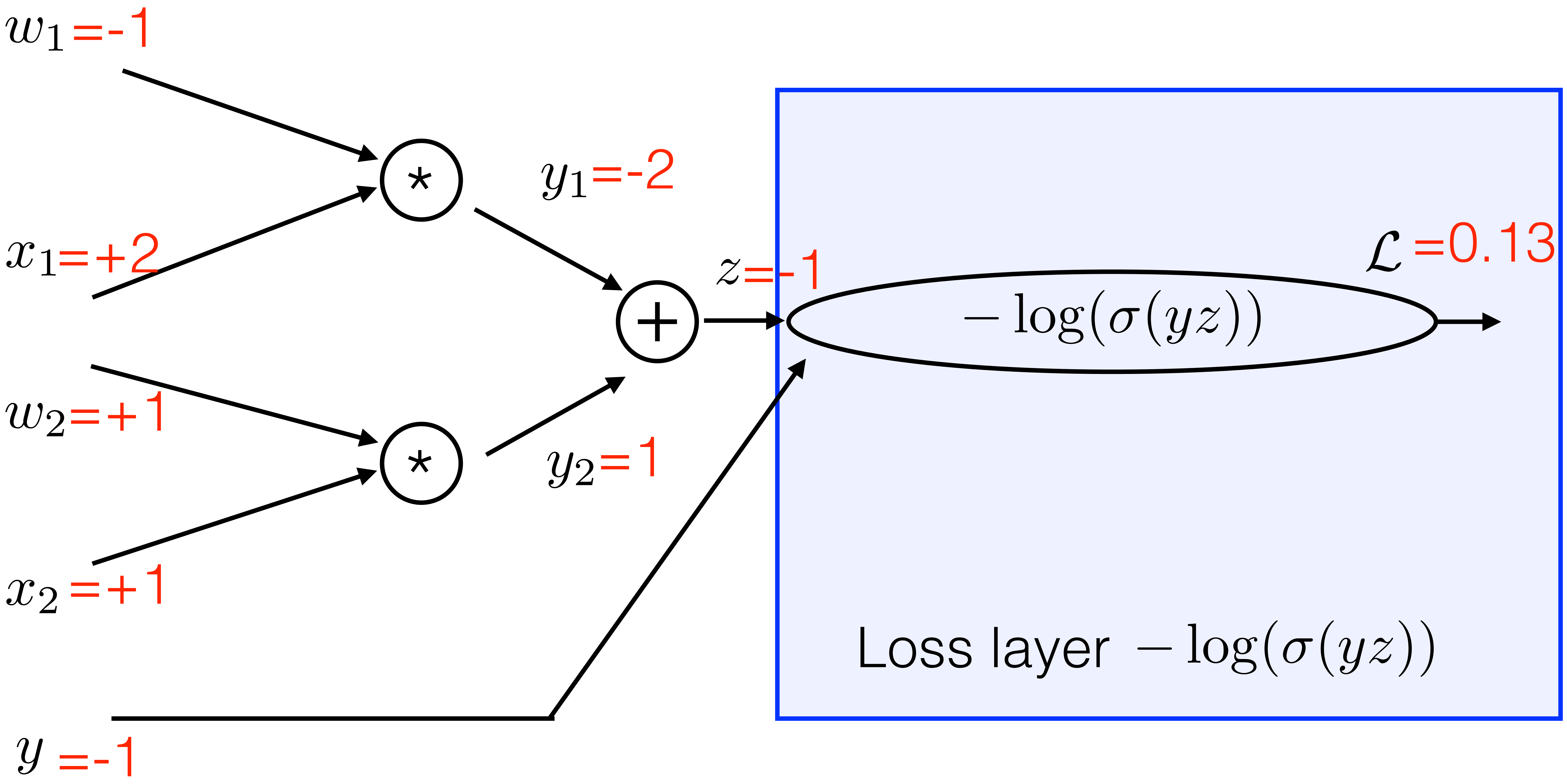




Computational graph of the learning



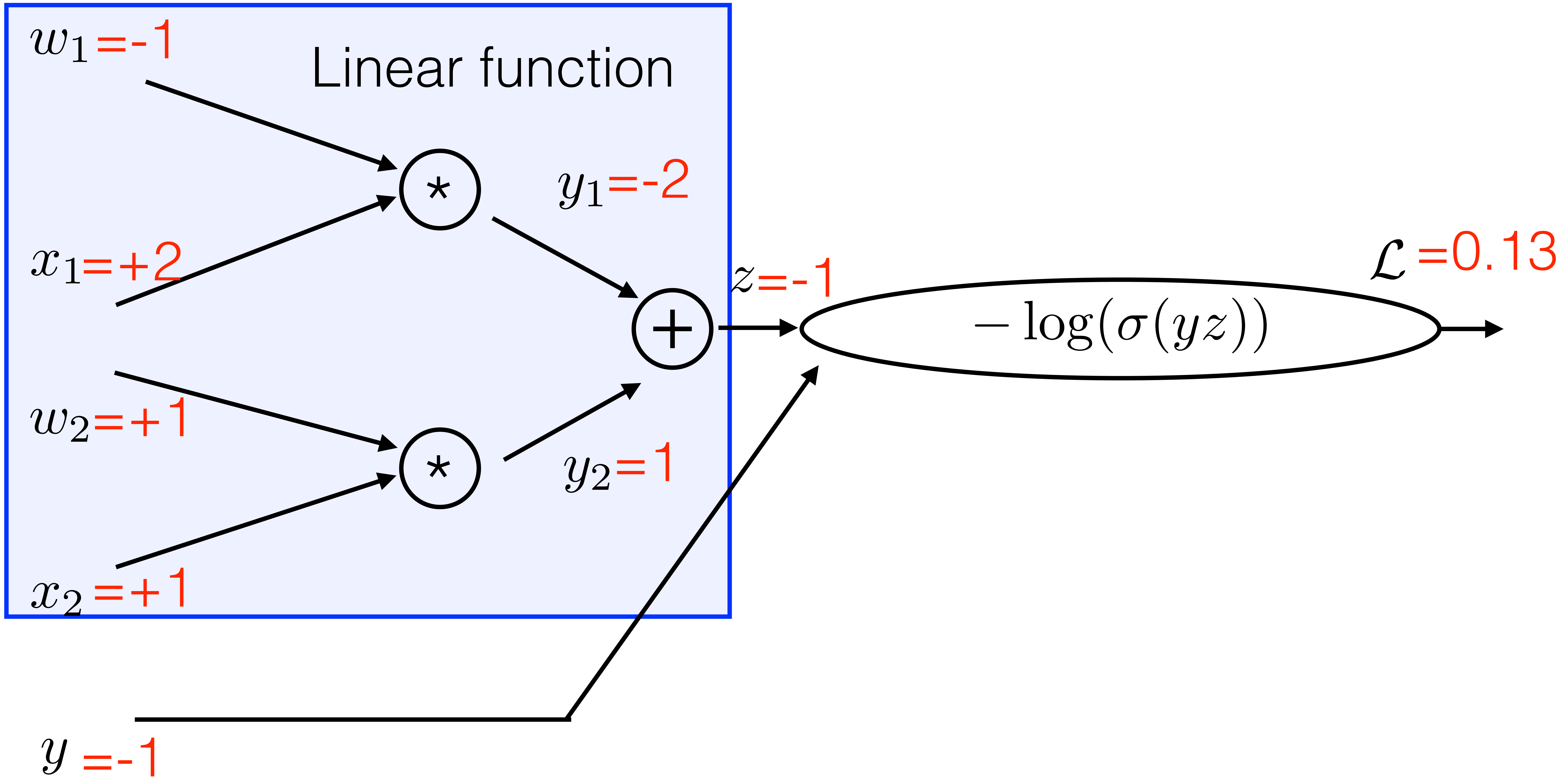
Backprop in vector representation



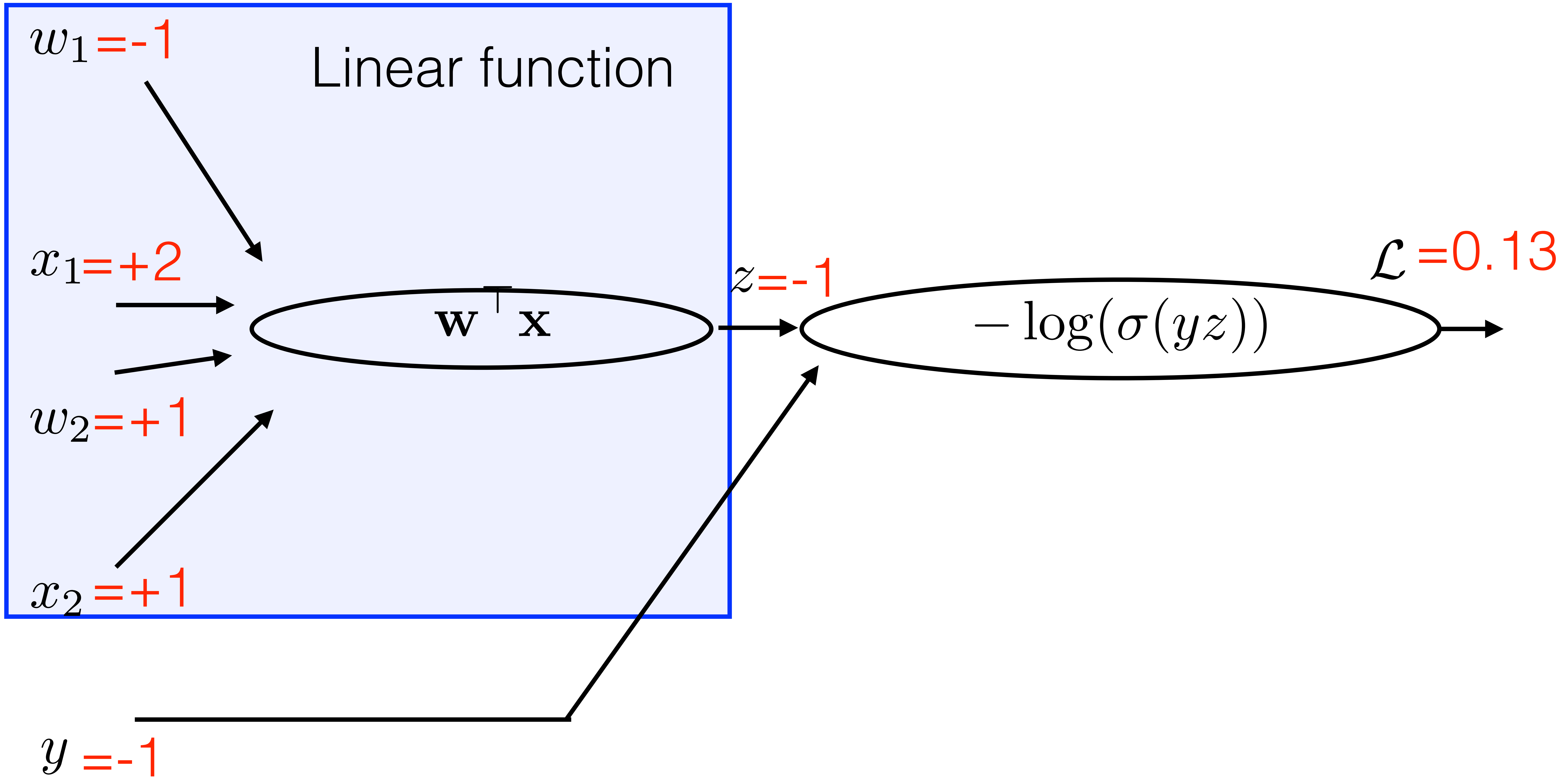
This is the logistic loss!

$$\mathcal{L}(y,z) = -\log(\sigma(yz)) = \log(1+\exp(-yz))$$

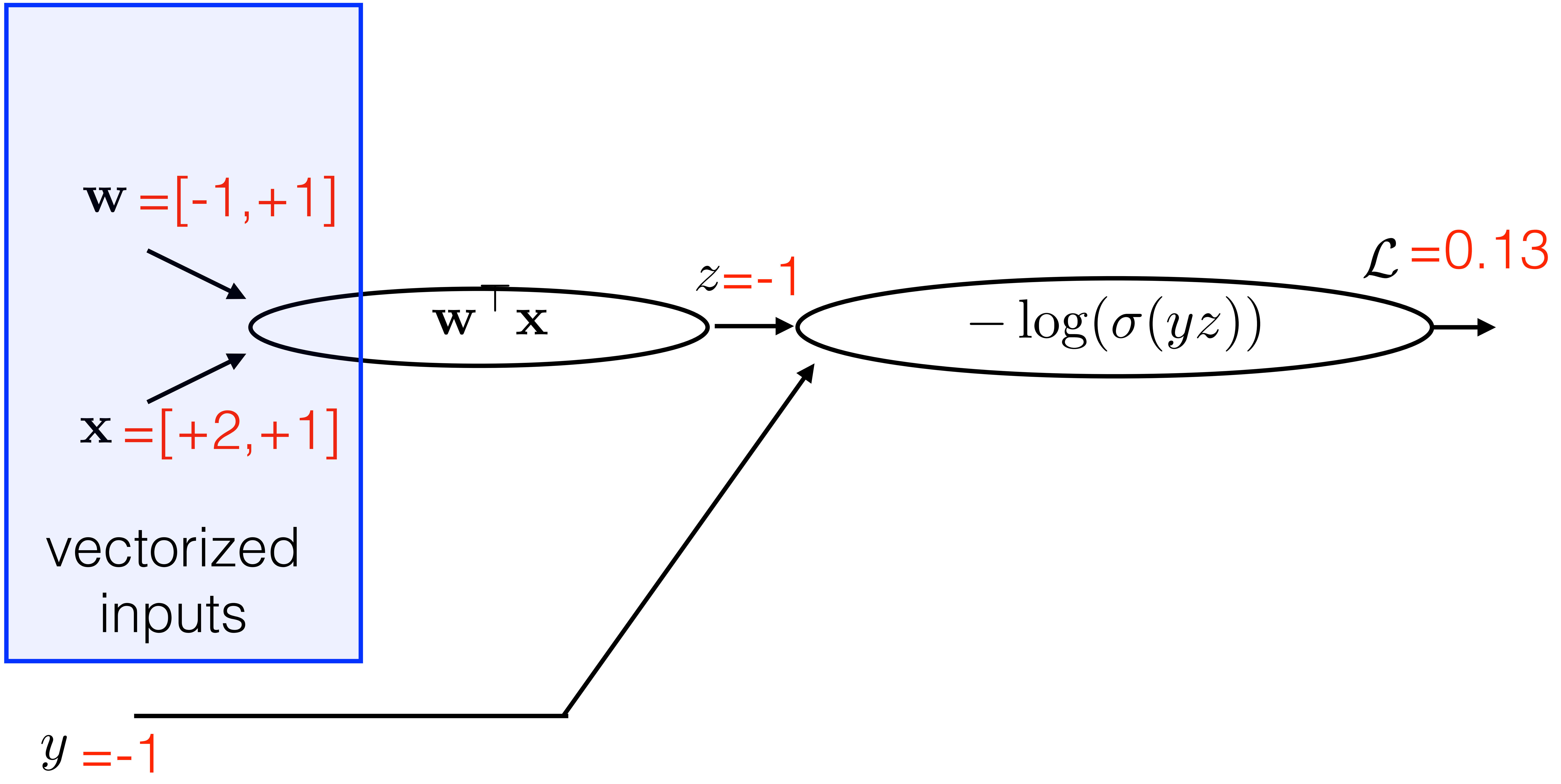
Backprop in vector representation



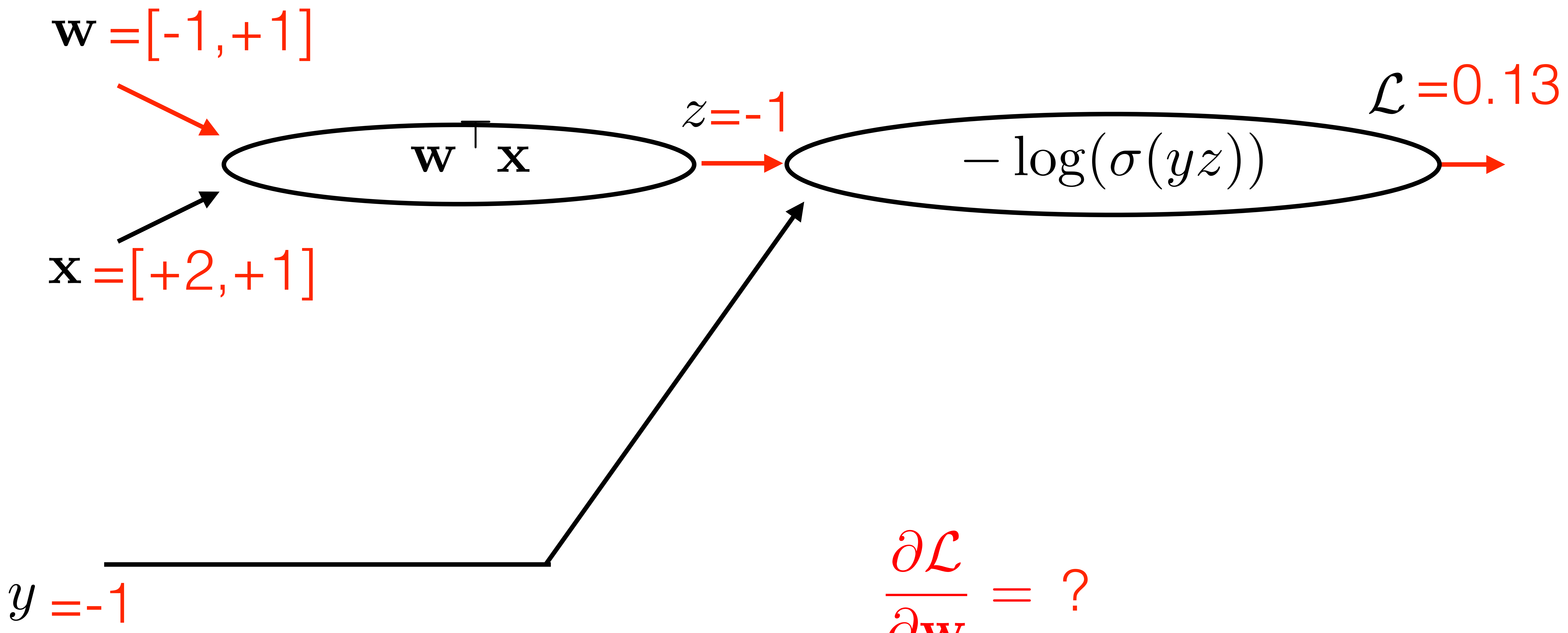
Backprop in vector representation



Backprop in vector representation

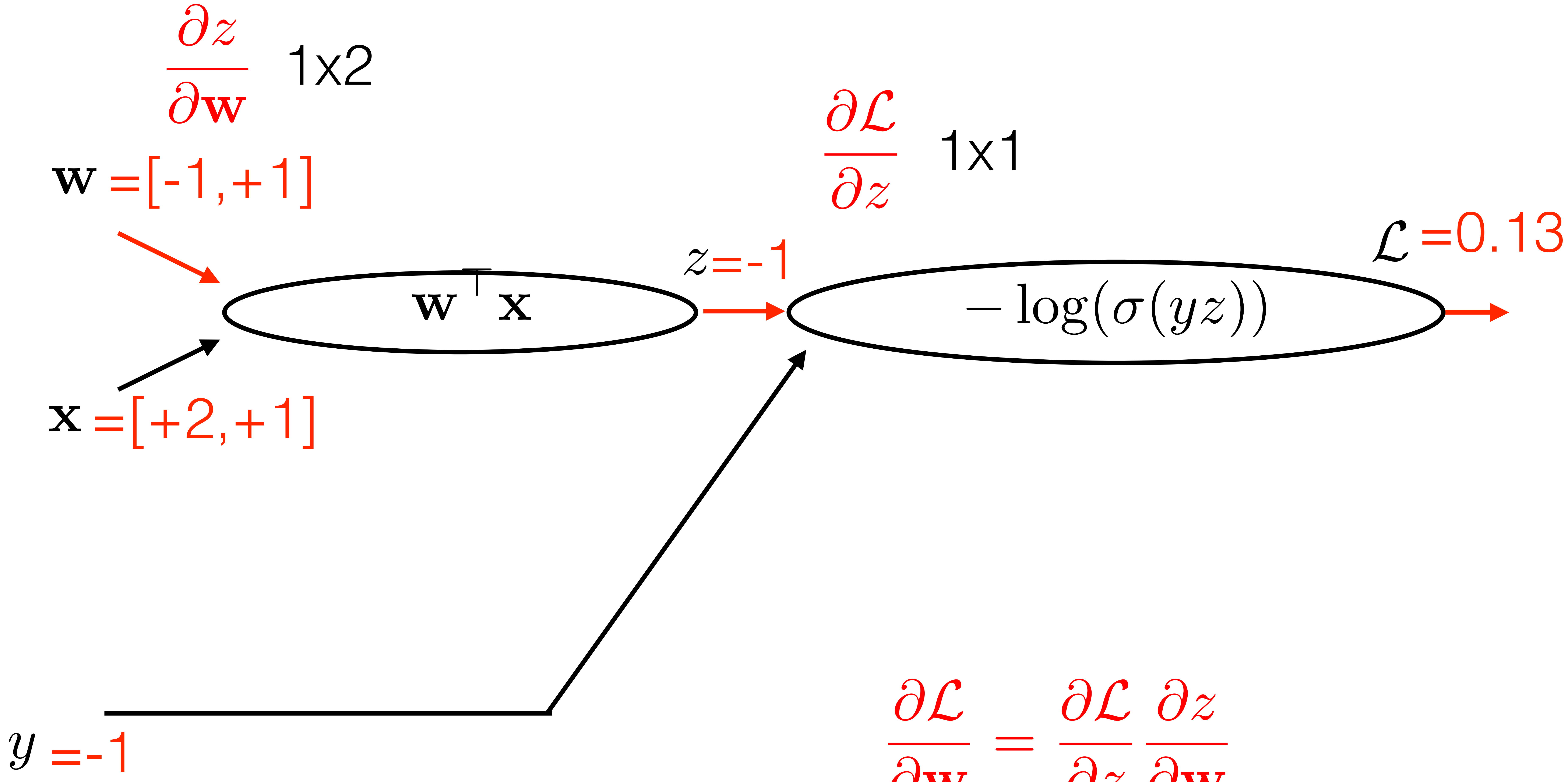


Backprop in vector representation



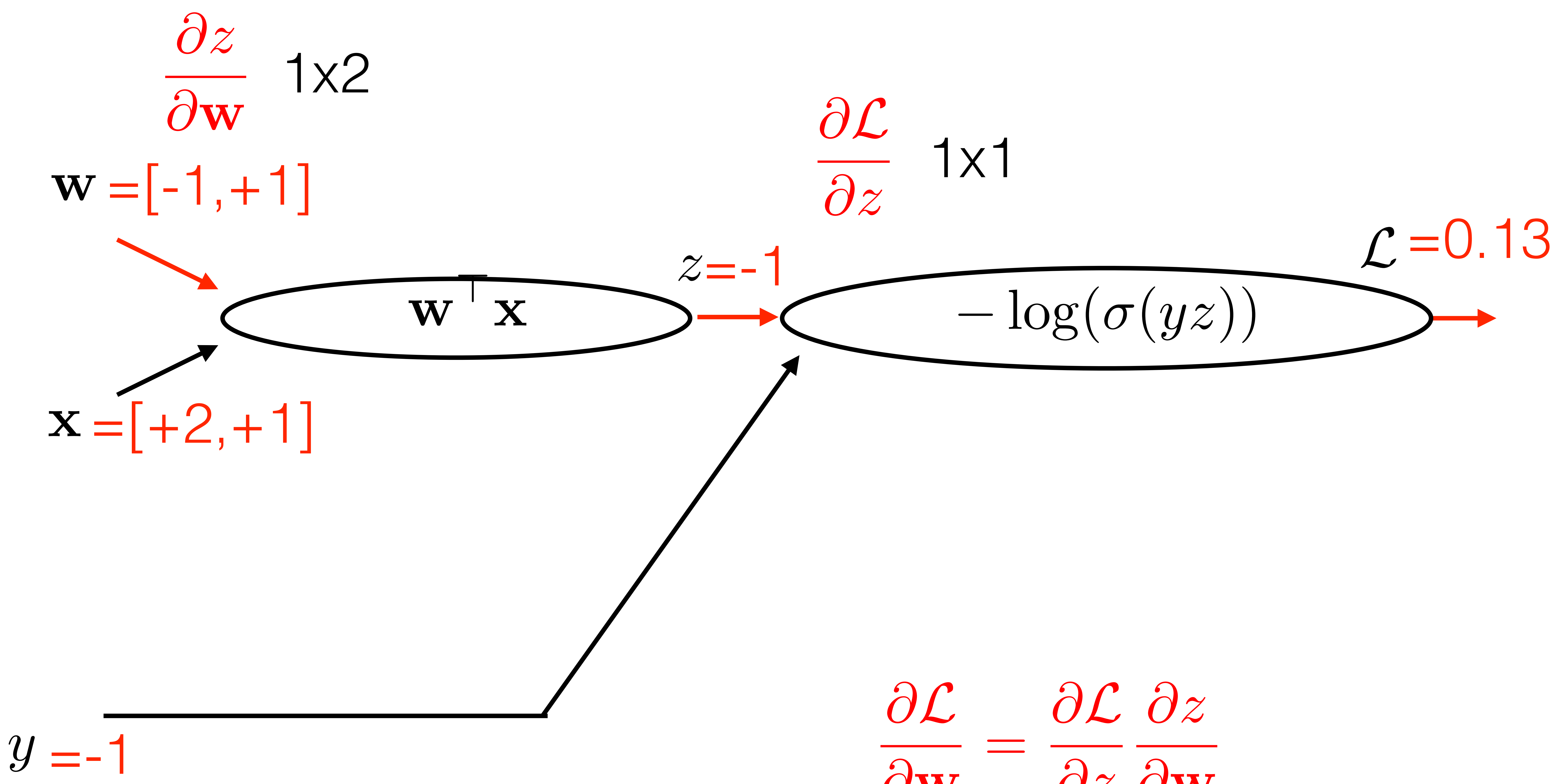
$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = ?$$

Backprop in vector representation



$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial \mathbf{w}}$$

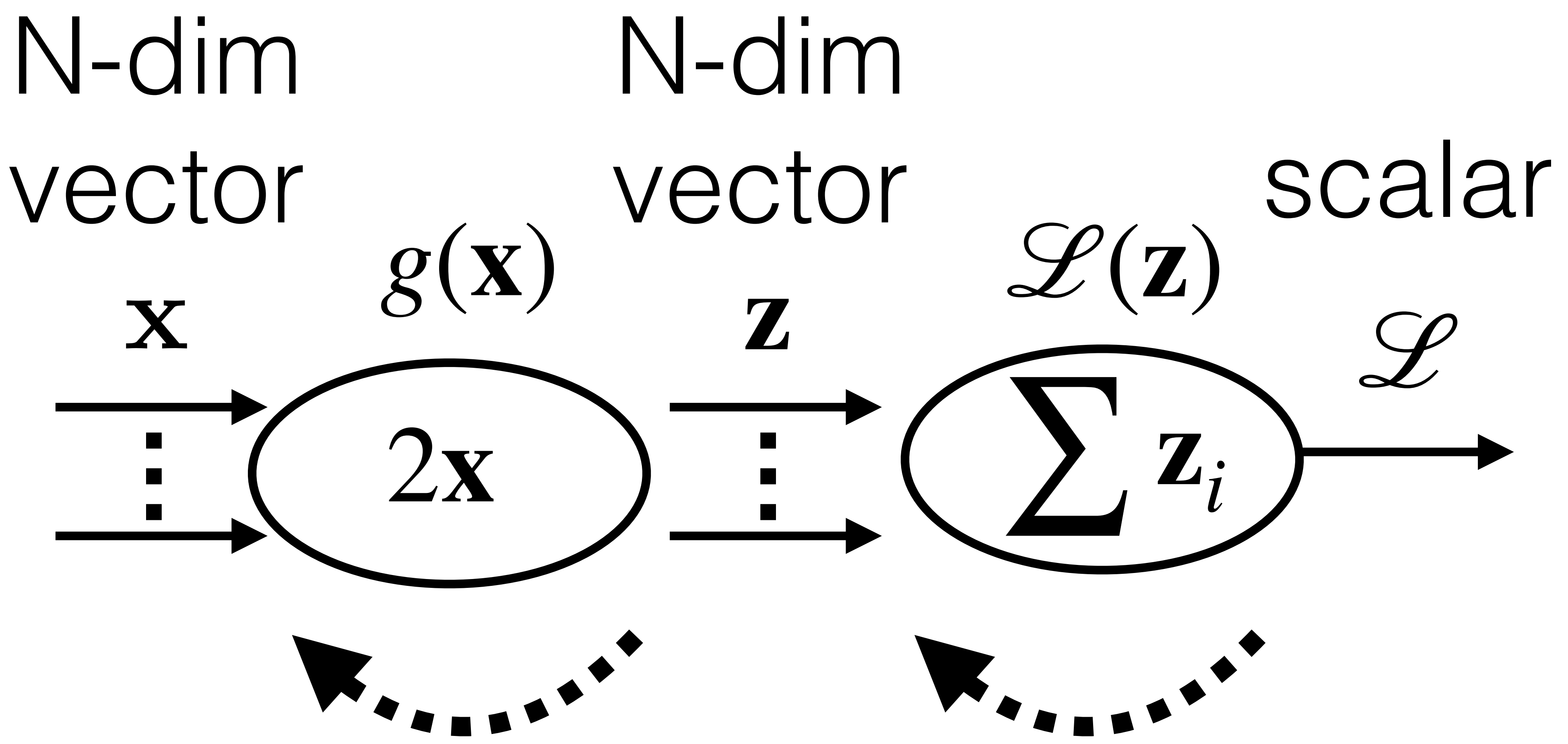
Backprop in vector representation



$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial z} \frac{\partial z}{\partial \mathbf{w}}$$

- How does it work **under the hood**?
- Can I design my special layer with **hand-design gradient**?

Example



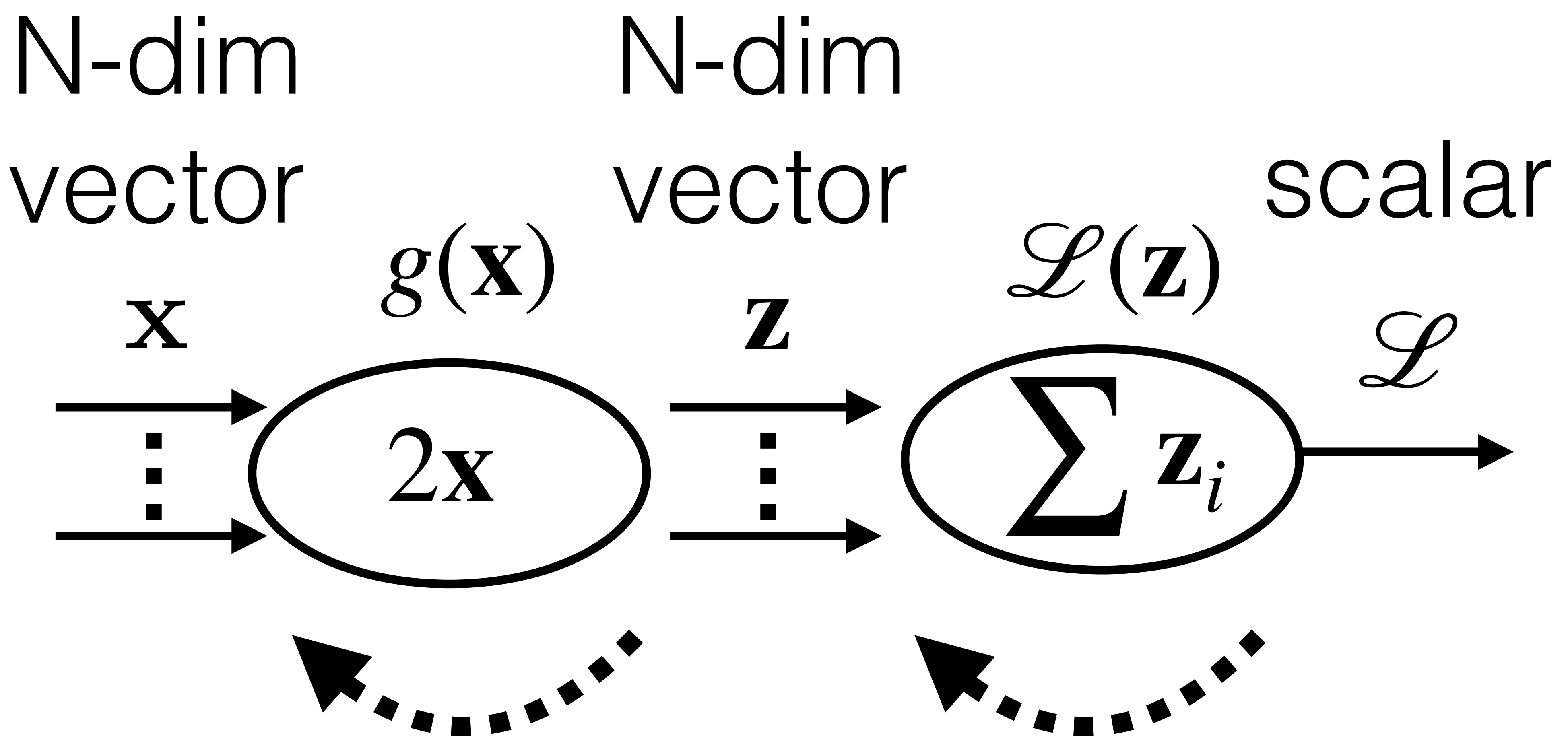
$$\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \dots & \\ & & & & 2 \end{bmatrix}$$

NXN

$$\frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} = \begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix}$$

1xN

Example



$$\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \dots & \\ & & & & 2 \end{bmatrix}$$

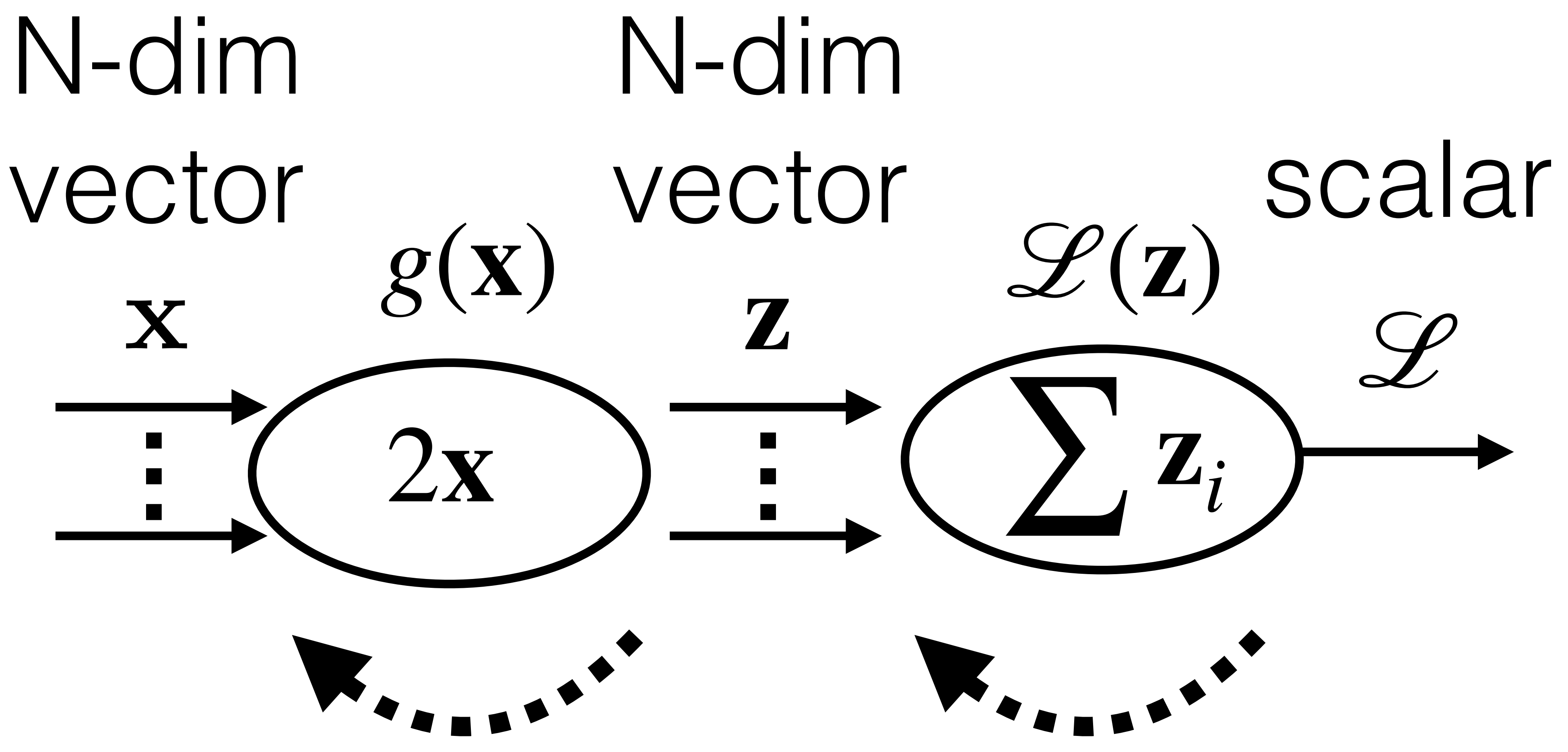
NXN

$$\frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} = \begin{bmatrix} 1, & 1, & 1, & \dots & 1 \end{bmatrix}$$

1xN

$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$$

Example



$$\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \dots & \\ & & & & 2 \end{bmatrix}$$

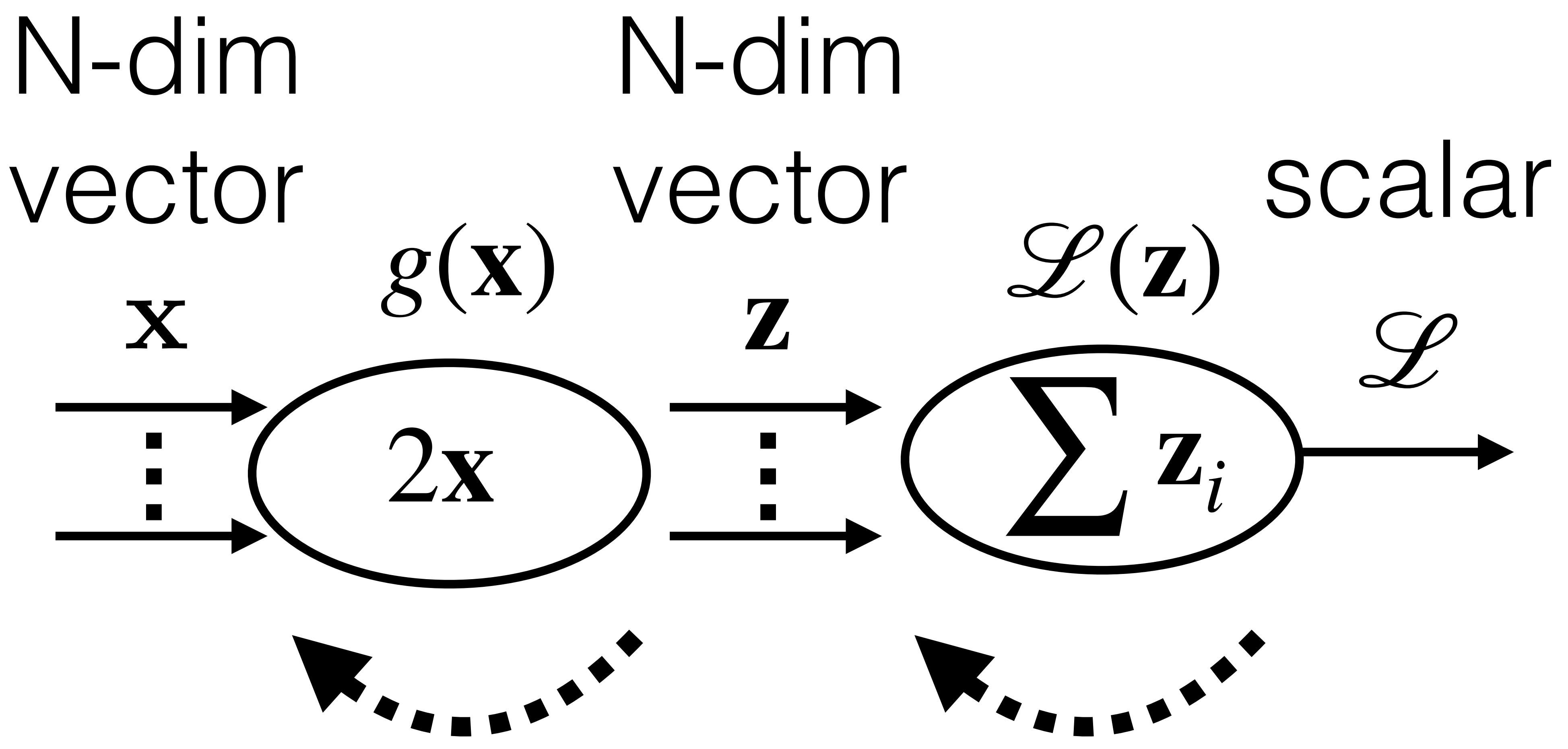
NXN

$$\frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} = \begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix}$$

1xN

$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} =$$

Example



$$\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \dots & \\ & & & & 2 \end{bmatrix}$$

NXN

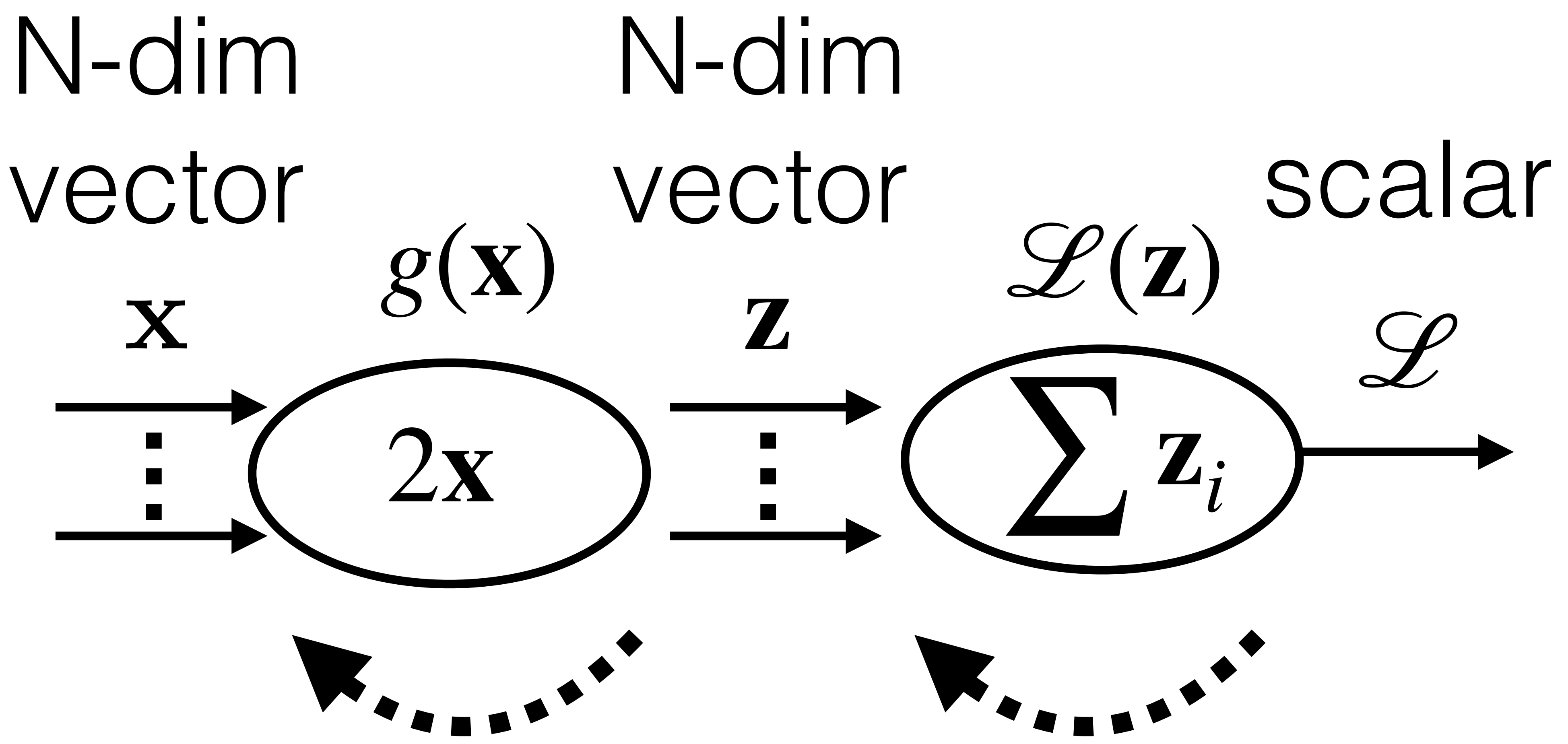
$$\frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} = \begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix}$$

1xN

$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix}$$

1xN

Example

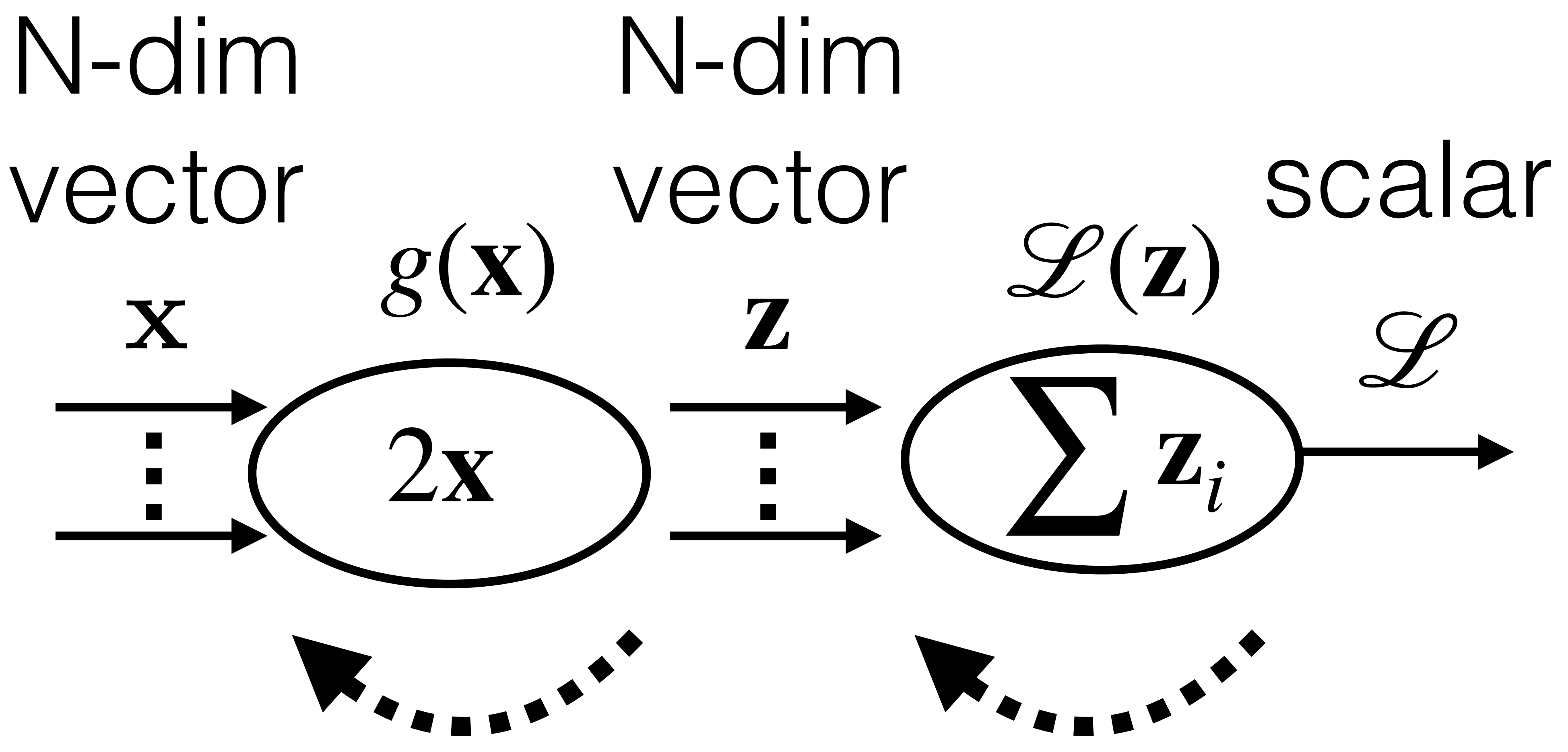



```
def vjp_𝓛(v, z):
    return v · ∂𝓛(z)/∂z
```

$$\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \underbrace{\begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \dots & \\ & & & & 2 \end{bmatrix}}_{\text{NXN}} \qquad \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} = \underbrace{\begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix}}_{\text{1xN}}$$

$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \underbrace{\begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix}}_{\text{1xN}} \underbrace{\begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \dots & \\ & & & & 2 \end{bmatrix}}_{\text{NXN}} =$$

Example




```
def vjp_𝓛(v, z):
    return v · ∂𝓛(z)/∂z
```

```
def vjp_g(v, x):


    return v · ∂g(y)/∂y
```

$$\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \cdots & \\ & & & & 2 \end{bmatrix}$$

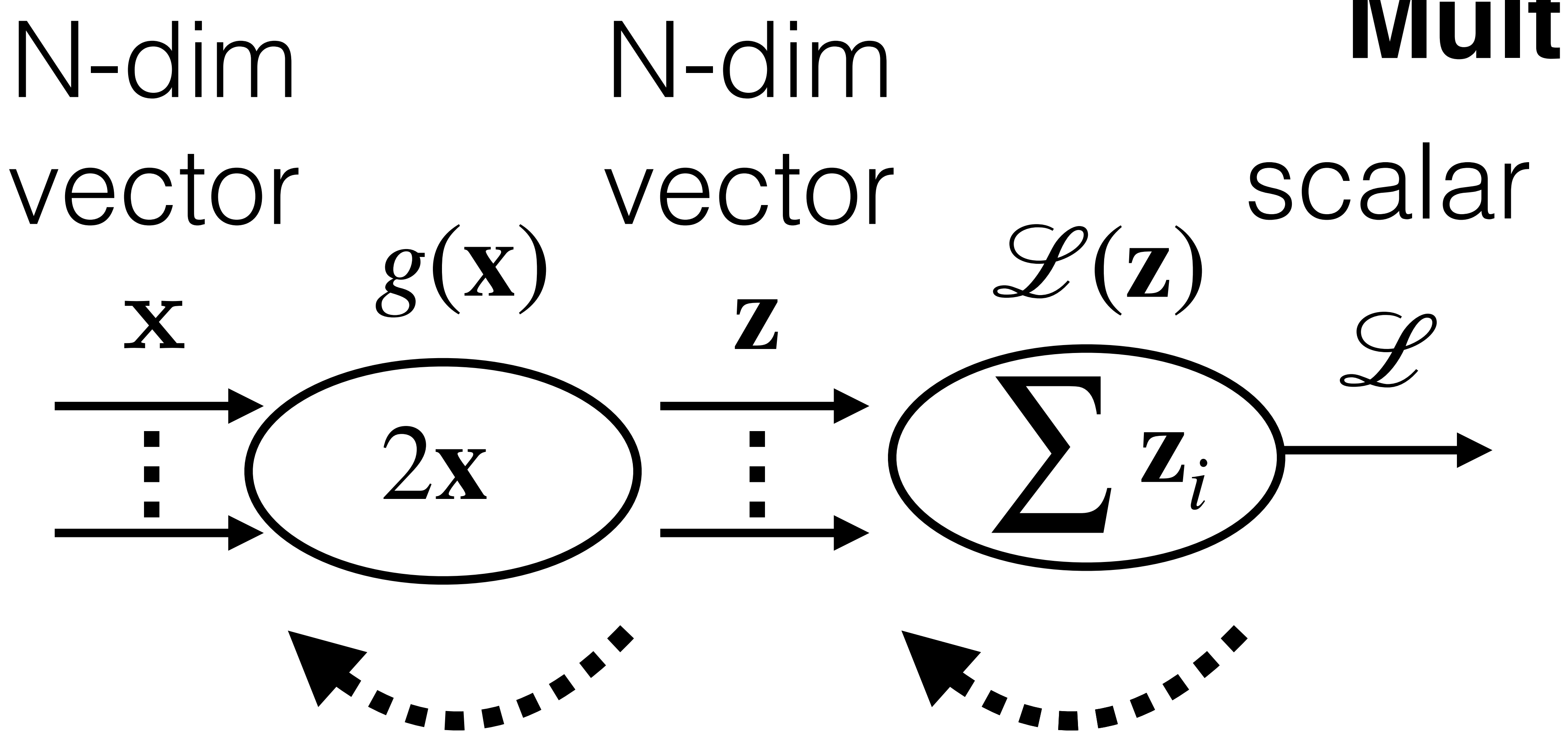
NXN

$$\frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} = \begin{bmatrix} 1, 1, 1, \ldots 1 \end{bmatrix}$$

1xN

$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \underbrace{\begin{bmatrix} 1, 1, 1, \ldots 1 \end{bmatrix}}_{1\text{xN}} \underbrace{\begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \cdots & \\ & & & & 2 \end{bmatrix}}_{\text{NXN}} =$$

Example



$$\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \underbrace{\begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \ddots & \\ & & & & 2 \end{bmatrix}}_{\text{NXN}}$$

$$\frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} = \underbrace{\begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix}}_{\text{1xN}}$$

$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \underbrace{\begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix}}_{\text{1xN}} \underbrace{\begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \ddots & \\ & & & & 2 \end{bmatrix}}_{\text{NXN}} =$$

Multiplications?

N

def $\text{vjp}_{\mathscr{L}}(\mathbf{v}, \mathbf{z})$:

return $\mathbf{v} \cdot \begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix}$

N*N

def $\text{vjp}_{g}(\mathbf{v}, \mathbf{x})$:

return $\mathbf{v} \cdot \begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \ddots & \\ & & & & 2 \end{bmatrix}$

(1) $\mathbf{v} = 1$

(2) $\mathbf{v} = \text{vjp}_{\mathscr{L}}(\mathbf{v}, \mathbf{z})$

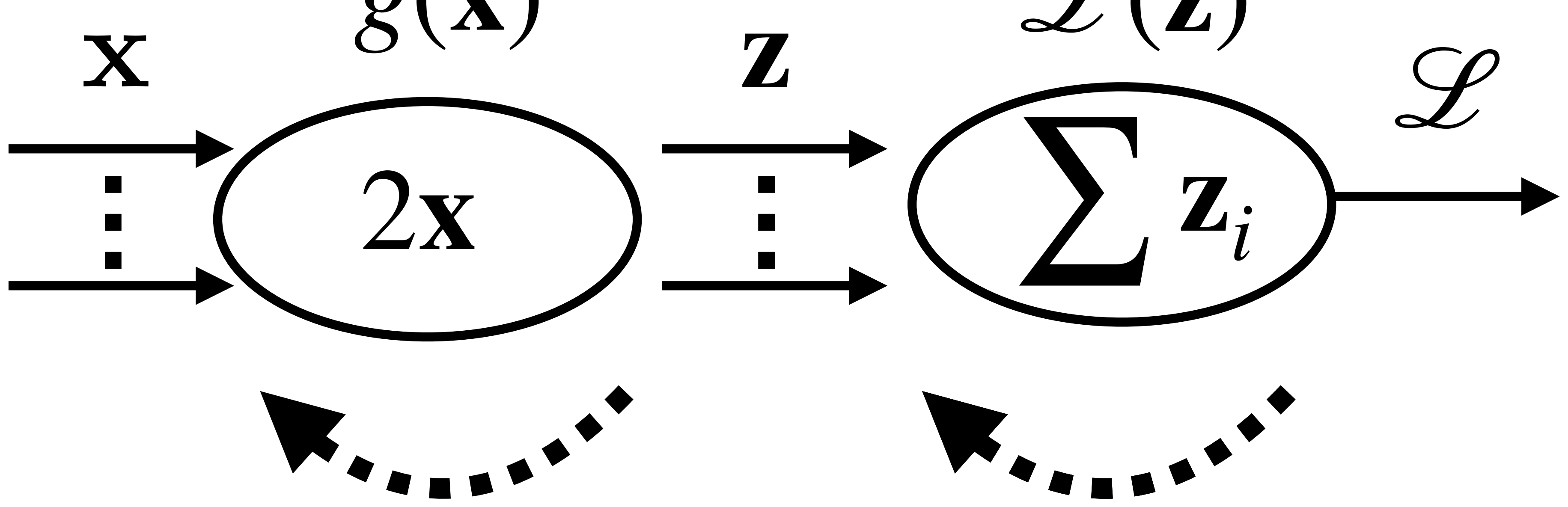
(3) $\mathbf{v} = \text{vjp}_{g}(\mathbf{v}, \mathbf{x})$

(4) $\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \mathbf{v}$

Example

Multiplications?

N-dim vector | N-dim vector | scalar



$$\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \cdots & \\ & & & & 2 \end{bmatrix}$$

NXN

$$\frac{\partial \mathcal{L}(\mathbf{z})}{\partial \mathbf{z}} = \begin{bmatrix} 1, 1, 1, \ldots 1 \end{bmatrix}$$

1xN

~~N~~

0

N*N

```
def vjp_L(v, z):
    return tile(v,1,N)
```

```
def vjp_g(v, x):
    return v ·
```

$$\begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \cdots & \\ & & & & 2 \end{bmatrix}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \underbrace{\begin{bmatrix} 1, 1, 1, \ldots 1 \end{bmatrix}}_{1\text{xN}} \underbrace{\begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \cdots & \\ & & & & 2 \end{bmatrix}}_{\text{NXN}} =$$

(1) $\mathbf{v} = 1$

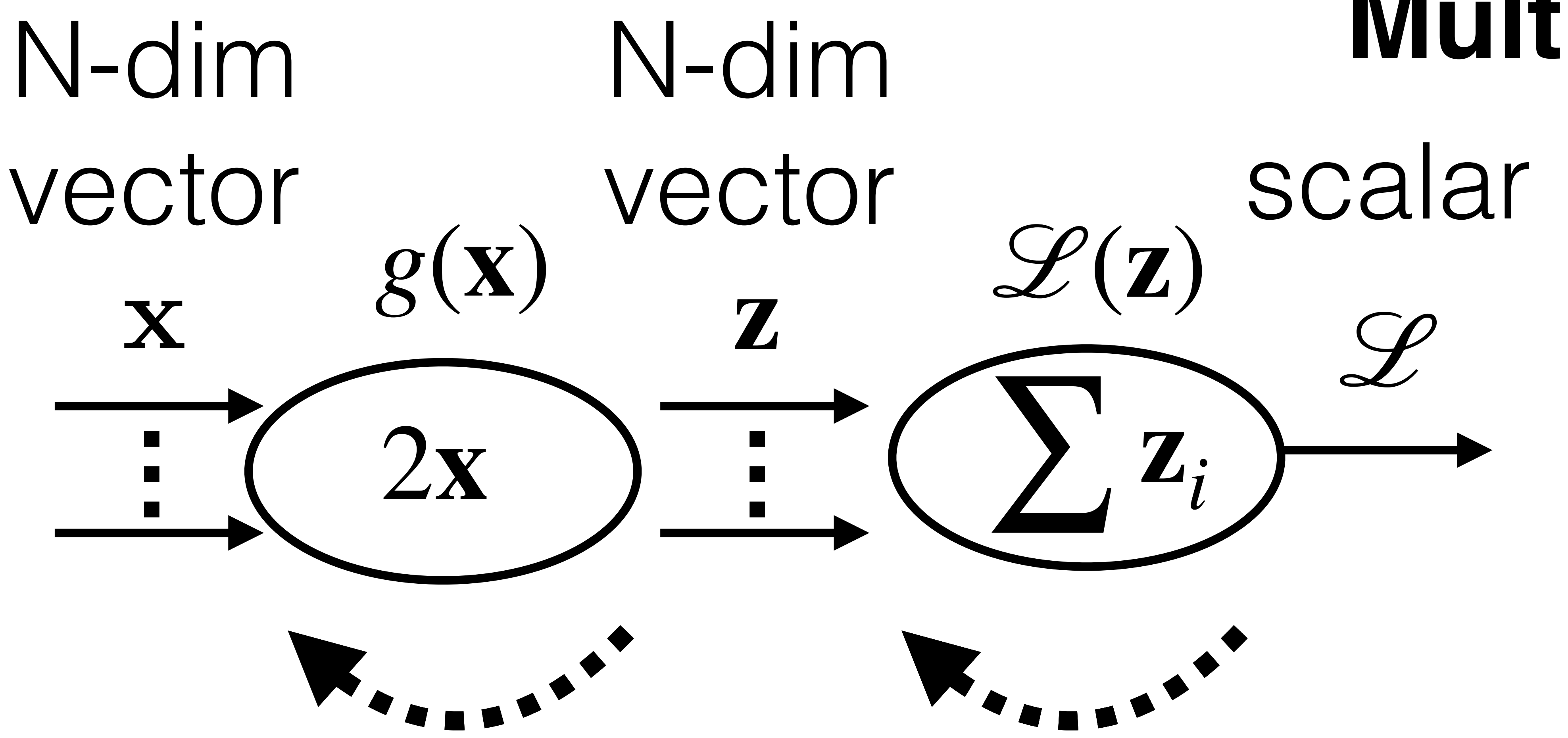
(2) $\mathbf{v} = \text{vjp}_{\mathcal{L}}(\mathbf{v}, \mathbf{z})$

(3) $\mathbf{v} = \text{vjp}_{g}(\mathbf{v}, \mathbf{x})$

(4) $\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \mathbf{v}$

Example

Multiplications?



$$\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \dots & \\ & & & & 2 \end{bmatrix}$$

NXN

$$\frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} = \begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix}$$

1xN

$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \underbrace{\begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix}}_{1\text{xN}} \underbrace{\begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \dots & \\ & & & & 2 \end{bmatrix}}_{\text{NXN}} =$$

$\cancel{\text{N}}$

0

$\cancel{\text{N*N}}$

N

def $\text{vjp}_{\mathscr{L}}(\mathbf{v}, \mathbf{z})$:

    return tile($\mathbf{v}$,1,N)

def $\text{vjp}_{g}(\mathbf{v}, \mathbf{x})$:

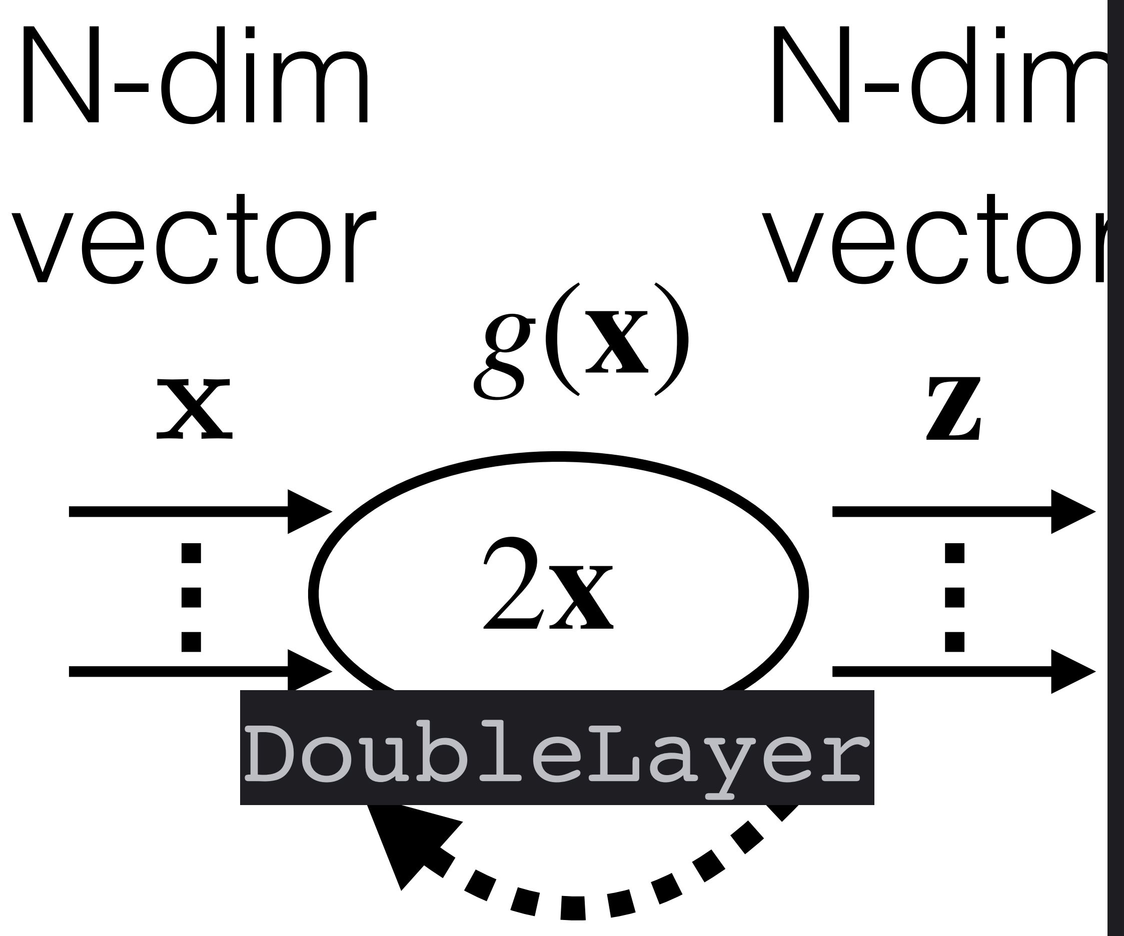
    return $\mathbf{v} \cdot 2$

(1) $\mathbf{v} = 1$

(2) $\mathbf{v} = \text{vjp}_{\mathscr{L}}(\mathbf{v}, \mathbf{z})$

(3) $\mathbf{v} = \text{vjp}_{g}(\mathbf{v}, \mathbf{x})$

(4) $\dfrac{\partial \mathscr{L}}{\partial \mathbf{x}} = \mathbf{v}$



$$\mathrm{vjp}_g(\mathbf{v}, \mathbf{x}) = \mathbf{v} \cdot 2$$

$$\frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} = \mathbf{v}$$

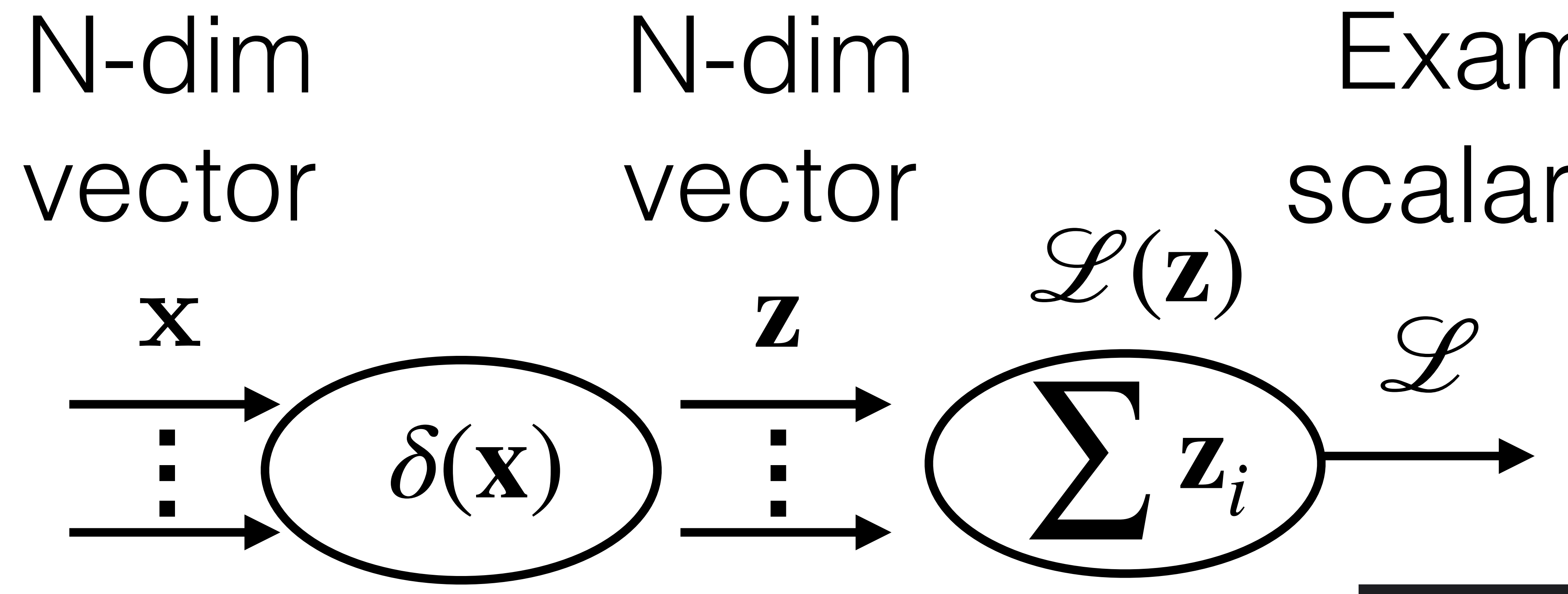
```python
class DoubleLayer(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        # forward pass
        z = 2 * x
        # save whatever is needed for backward-pas
        ctx.save_for_backward(x)
        return z

    @staticmethod
    def backward(ctx, v):
        # v is upstream gradient
        (x,) = ctx.saved_tensors
        # Local VJP: z = 2x => dL/dx = 2 * dL/dz
        return 2 * v


x = torch.tensor([1., 2., 3.], requires_grad=True)
z = DoubleLayer.apply(x)
print(z)
# tensor([2., 4., 6.], grad_fn=<DoubleLayerBackward>)
L = z.sum()
L.backward()
print(x.grad)  # => tensor([2., 2., 2.])
```

vjp is automatically added to tensors during the construction of computational graph

Example: HardStep Function



HardStepWithSigmoidGrad

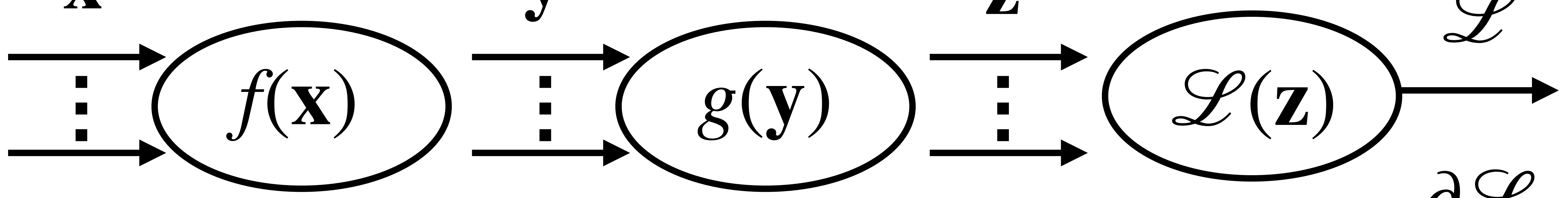
$$\mathrm{vjp}_g(\mathbf{v}, \mathbf{x}) = \sigma(\mathbf{v})\big(1 - \sigma(\mathbf{v})\big)$$

```python
class HardStepWithSigmoidGrad(torch.autograd.Function):
   @staticmethod
     def forward(ctx, x):
         ctx.save_for_backward(x)
         # hard step function
         return (x > 0).float()
     @staticmethod
     def backward(ctx, v):
         (x,) = ctx.saved_tensors
         # gradient from sigmoid
         sigma = torch.sigmoid(x)
         v = v * sigma * (1 - sigma)
         return v

x = torch.tensor([1., 2., 3.], requires_grad=True)
y = HardStepWithSigmoidGrad.apply(x)
loss = y.sum()
loss.backward()
```

Chain-rule in computational graph and Jacobians

K-dim vector $\mathbf{x}$ | M-dim vector $\mathbf{y}$ | N-dim vector $\mathbf{z}$ | scalar $\mathscr{L}$



$$\frac{\partial \mathscr{L}}{\partial \mathscr{L}} = 1$$

scalar seed

$$\frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}}$$

(1xN) matrix

Layer: $f(\mathbf{x}) : \mathbb{R}^K \rightarrow \mathbb{R}^M$

Jacobian: $\dfrac{\partial f(\mathbf{x})}{\partial \mathbf{x}} : \mathbb{R}^K \rightarrow \mathbb{R}^{M \times K}$

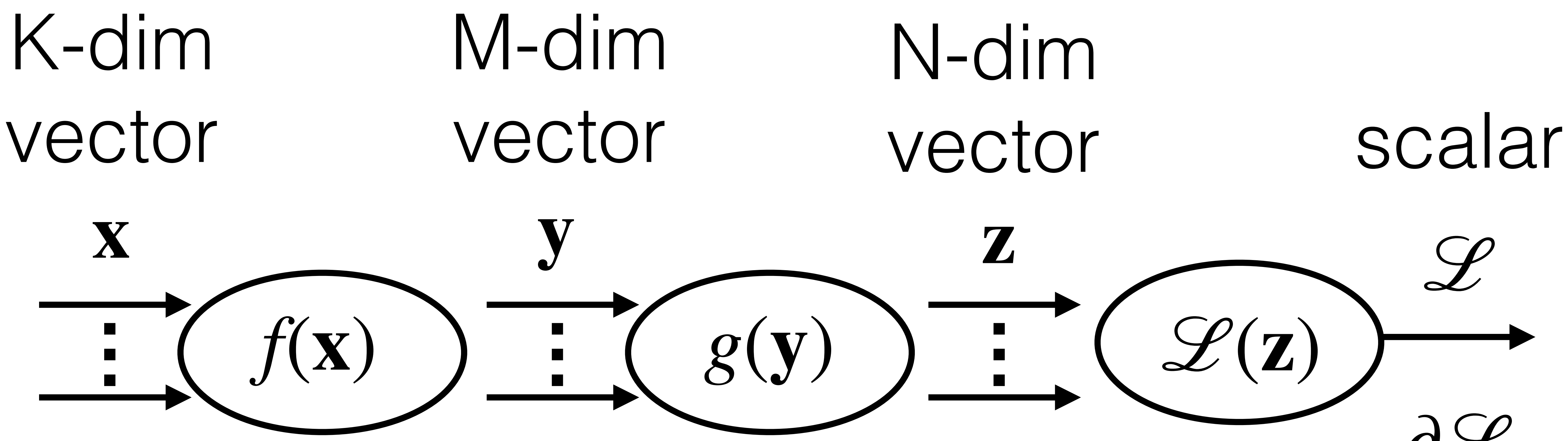
$$= \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_K} \\ \vdots & & \vdots \\ \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_K} \end{bmatrix} = \begin{bmatrix} \nabla f_1(\mathbf{x})^\top \\ \vdots \\ \nabla f_M(\mathbf{x})^\top \end{bmatrix}$$

gradient's transpose

1xK		1x1	1xN

Loss jac wrt $\mathbf{x}$ $\quad \dfrac{\partial \mathscr{L}}{\partial \mathbf{x}} = 1 \; \dfrac{\partial \mathscr{L}}{\partial \mathbf{z}}$

Chain-rule in computational graph and Jacobians



$$\frac{\partial \mathcal{L}(\mathbf{z})}{\partial \mathbf{z}} \qquad \frac{\partial \mathcal{L}}{\partial \mathcal{L}} = 1$$

(1xN) matrix

scalar seed

Layer: $f(\mathbf{x}) : \mathbb{R}^K \rightarrow \mathbb{R}^M$

Jacobian: $\dfrac{\partial f(\mathbf{x})}{\partial \mathbf{x}} : \mathbb{R}^K \rightarrow \mathbb{R}^{M \times K}$

$$= \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_K} \\ \vdots & & \vdots \\ \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_K} \end{bmatrix} = \begin{bmatrix} \nabla f_1(\mathbf{x})^\top \\ \vdots \\ \nabla f_M(\mathbf{x})^\top \end{bmatrix}$$

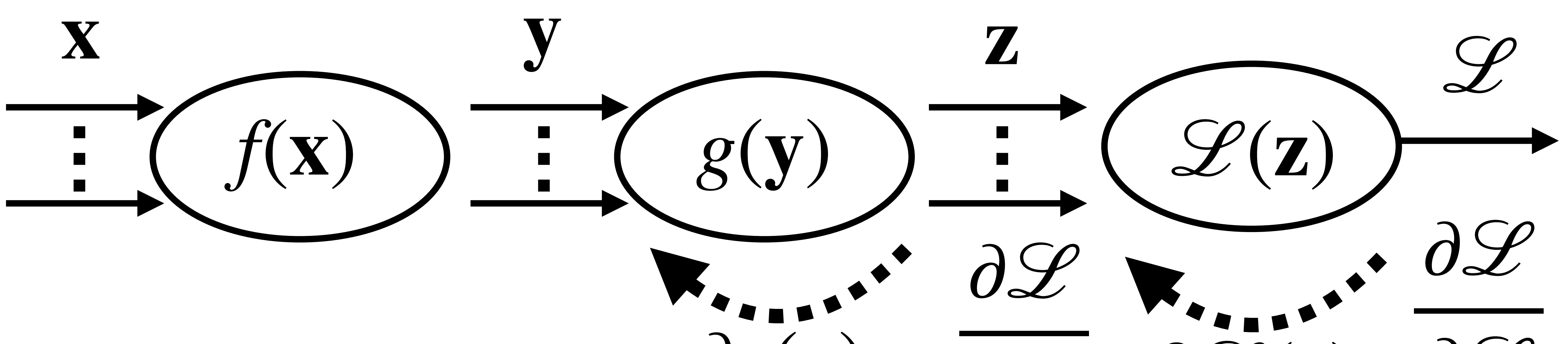
gradient's transpose

1xK 1x1 1xN

Loss jac wrt $\mathbf{x}$ $\dfrac{\partial \mathcal{L}}{\partial \mathbf{x}} = \boxed{1 \quad \dfrac{\partial \mathcal{L}}{\partial \mathbf{z}} = \dfrac{\partial \mathcal{L}}{\partial \mathbf{z}} \text{ N-dim vector}}$

Chain-rule in computational graph and Jacobians

K-dim vector | M-dim vector | N-dim vector | scalar



$\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$ (NxM) matrix

$\frac{\partial \mathscr{L}}{\partial \mathbf{z}}$ N-dim vector

$\frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}}$ (1xN) matrix

$\frac{\partial \mathscr{L}}{\partial \mathscr{L}} = 1$ scalar seed

Layer: $f(\mathbf{x}) : \mathbb{R}^K \rightarrow \mathbb{R}^M$

Jacobian: $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} : \mathbb{R}^K \rightarrow \mathbb{R}^{M \times K}$

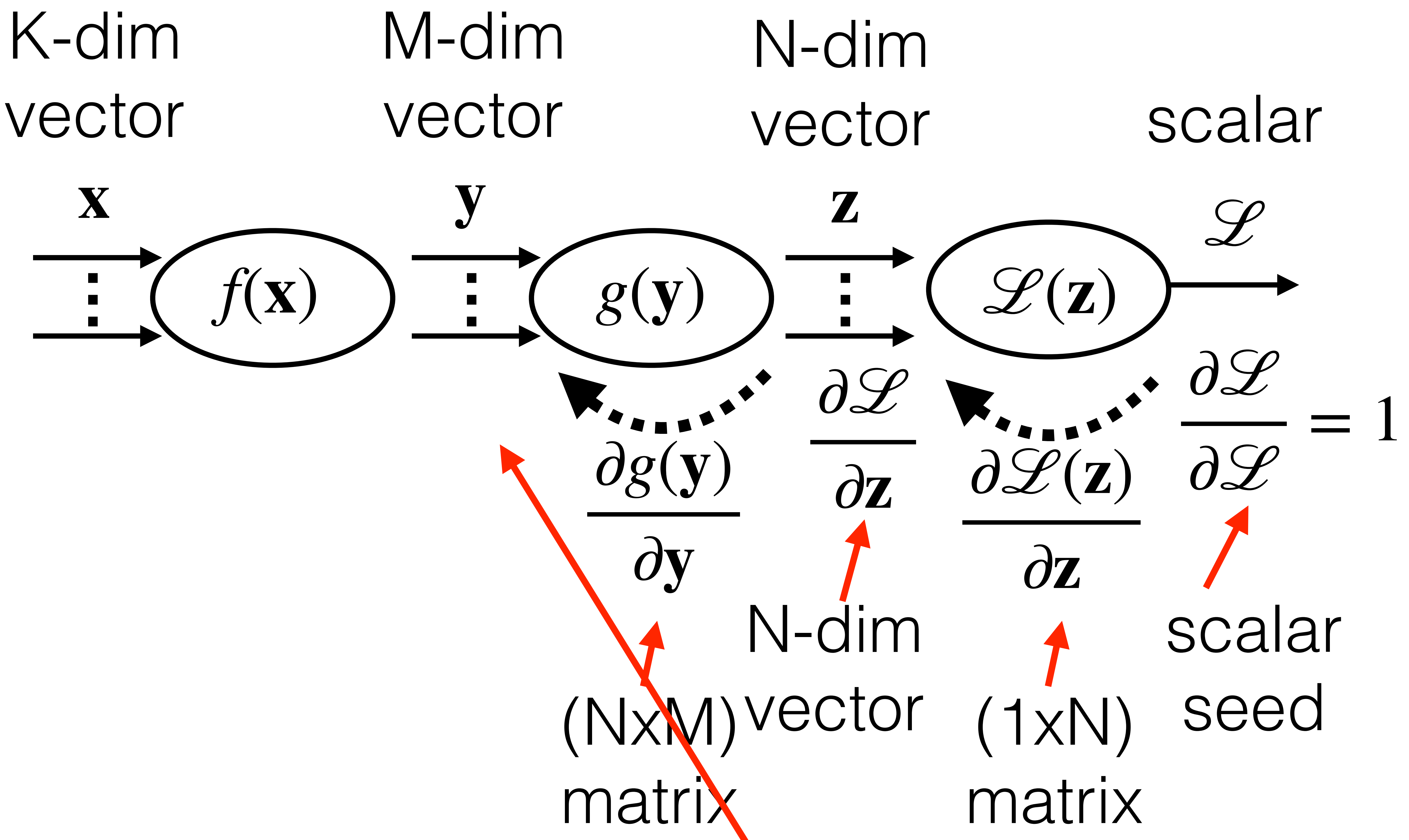
$$= \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_K} \\ \vdots & & \vdots \\ \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_K} \end{bmatrix} = \begin{bmatrix} \nabla f_1(\mathbf{x})^\top \\ \vdots \\ \nabla f_M(\mathbf{x})^\top \end{bmatrix}$$

gradient's transpose

Loss jac wrt $\mathbf{x}$

$$\underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{x}}}_{1\text{x}K} = \underbrace{1}_{1\text{x}1} \; \underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{z}}}_{1\text{x}N} \; \underbrace{\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}}_{N\text{x}M}$$

Chain-rule in computational graph and Jacobians



Layer: $f(\mathbf{x}) : \mathbb{R}^K \rightarrow \mathbb{R}^M$

Jacobian: $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} : \mathbb{R}^K \rightarrow \mathbb{R}^{M \times K}$

$$= \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_K} \\ \vdots & & \vdots \\ \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_K} \end{bmatrix} = \begin{bmatrix} \nabla f_1(\mathbf{x})^\top \\ \vdots \\ \nabla f_M(\mathbf{x})^\top \end{bmatrix}$$

gradient's transpose

Loss jac wrt $\mathbf{x}$

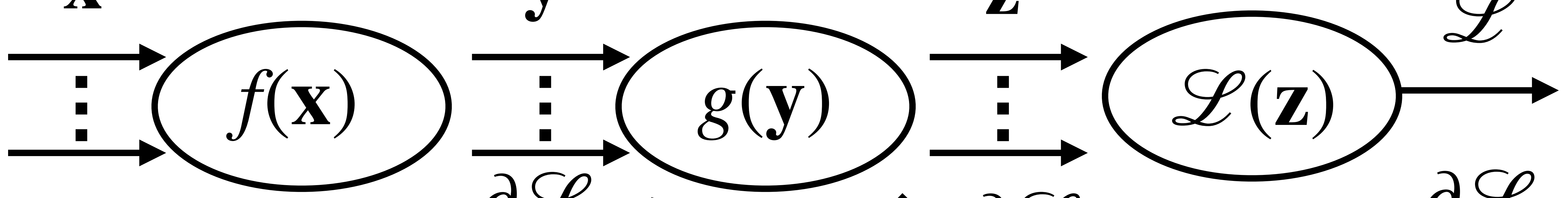
$$\underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{x}}}_{1\text{x}K} = \underbrace{1}_{1\text{x}1} \; \underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{z}}}_{1\text{x}N} \; \underbrace{\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}}_{N\text{x}M} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}}$$

M-dim vector

Chain-rule in computational graph and Jacobians

K-dim vector | M-dim vector | N-dim vector | scalar

$\mathbf{x}$ | $\mathbf{y}$ | $\mathbf{z}$ | $\mathscr{L}$



$$\frac{\partial \mathscr{L}}{\partial \mathbf{y}} \qquad \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} \qquad \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \qquad \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}} \qquad \frac{\partial \mathscr{L}}{\partial \mathscr{L}} = 1$$

M-dim vector | (NxM) matrix | N-dim vector | (1xN) matrix | scalar seed

Layer: $f(\mathbf{x}) : \mathbb{R}^K \rightarrow \mathbb{R}^M$

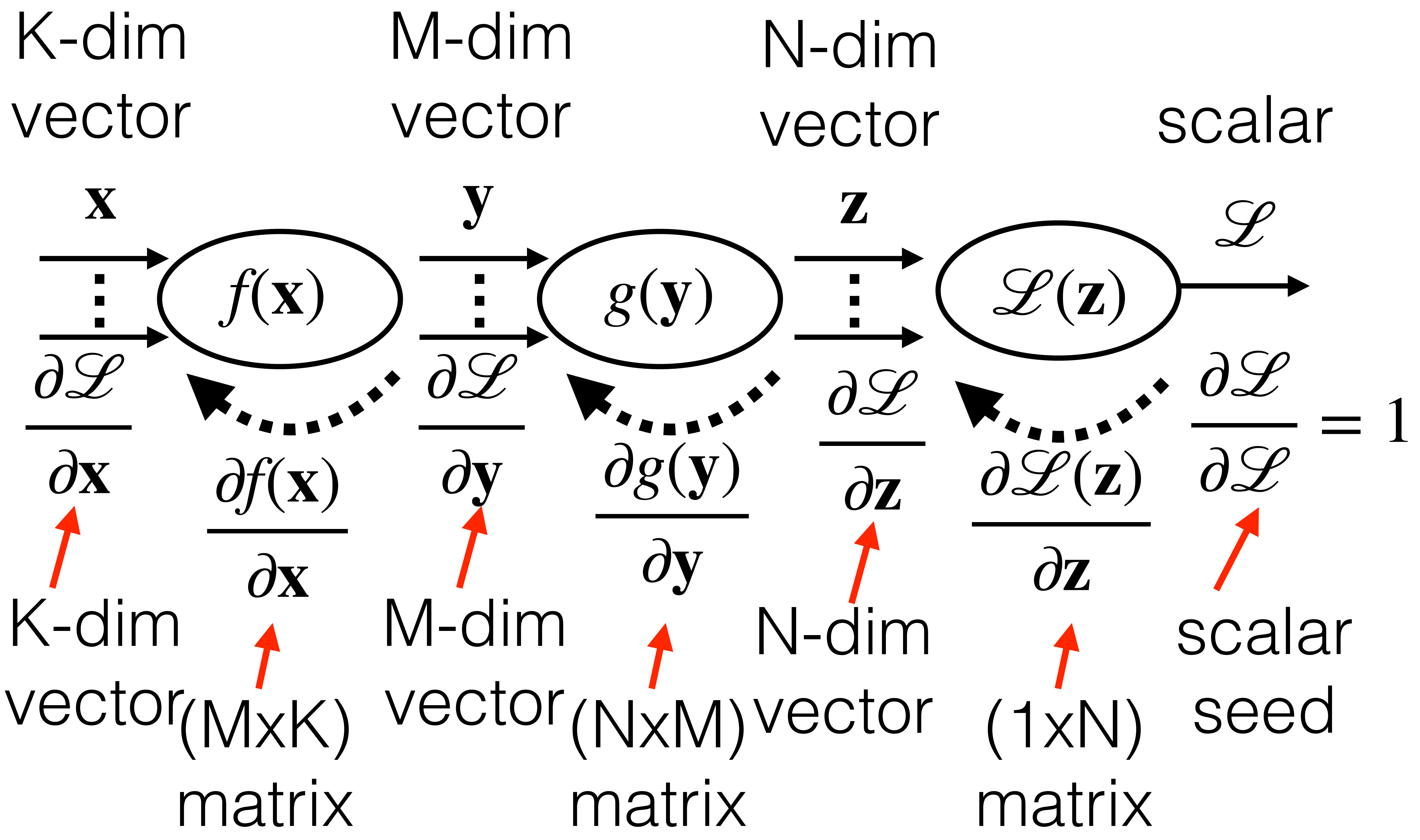
Jacobian: $\dfrac{\partial f(\mathbf{x})}{\partial \mathbf{x}} : \mathbb{R}^K \rightarrow \mathbb{R}^{M \times K}$

$$= \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_K} \\ \vdots & & \vdots \\ \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_K} \end{bmatrix} = \begin{bmatrix} \nabla f_1(\mathbf{x})^\top \\ \vdots \\ \nabla f_M(\mathbf{x})^\top \end{bmatrix}$$

gradient's transpose

	1xK		1x1	1xN	NxM
Loss jac wrt $\mathbf{x}$	$\dfrac{\partial \mathscr{L}}{\partial \mathbf{x}}$	=	1	$\dfrac{\partial \mathscr{L}}{\partial \mathbf{z}}$	$\dfrac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

Chain-rule in computational graph and Jacobians



Layer: $f(\mathbf{x}) : \mathbb{R}^K \rightarrow \mathbb{R}^M$

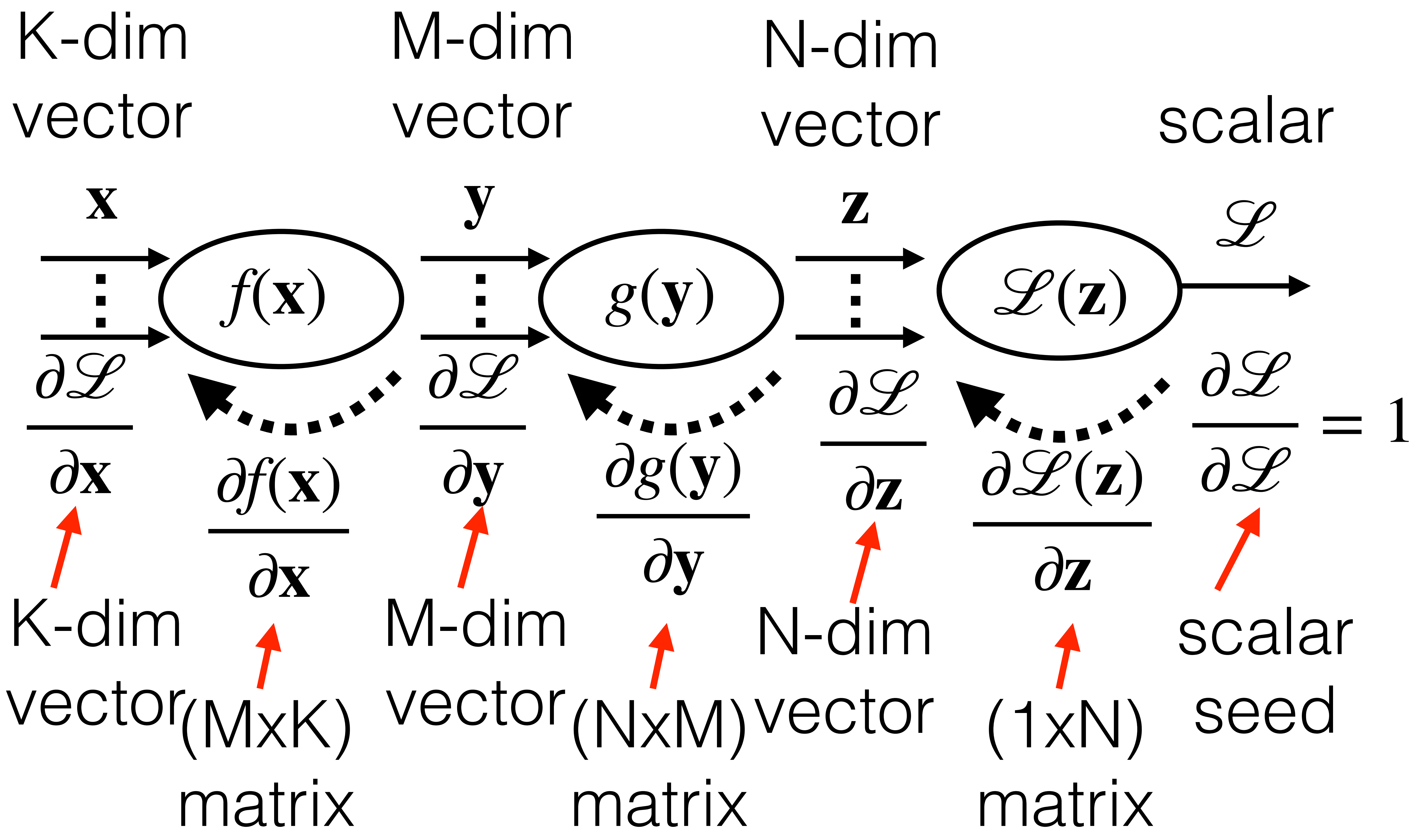
Jacobian: $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} : \mathbb{R}^K \rightarrow \mathbb{R}^{M \times K}$

$$= \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_K} \\ \vdots & & \vdots \\ \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_K} \end{bmatrix} = \begin{bmatrix} \nabla f_1(\mathbf{x})^\top \\ \vdots \\ \nabla f_M(\mathbf{x})^\top \end{bmatrix}$$

gradient's transpose

	1xK		1x1	1xN	NxM	MxK
Loss jac wrt $\mathbf{x}$	$\frac{\partial \mathscr{L}}{\partial \mathbf{x}}$	=	1	$\frac{\partial \mathscr{L}}{\partial \mathbf{z}}$	$\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$	$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$

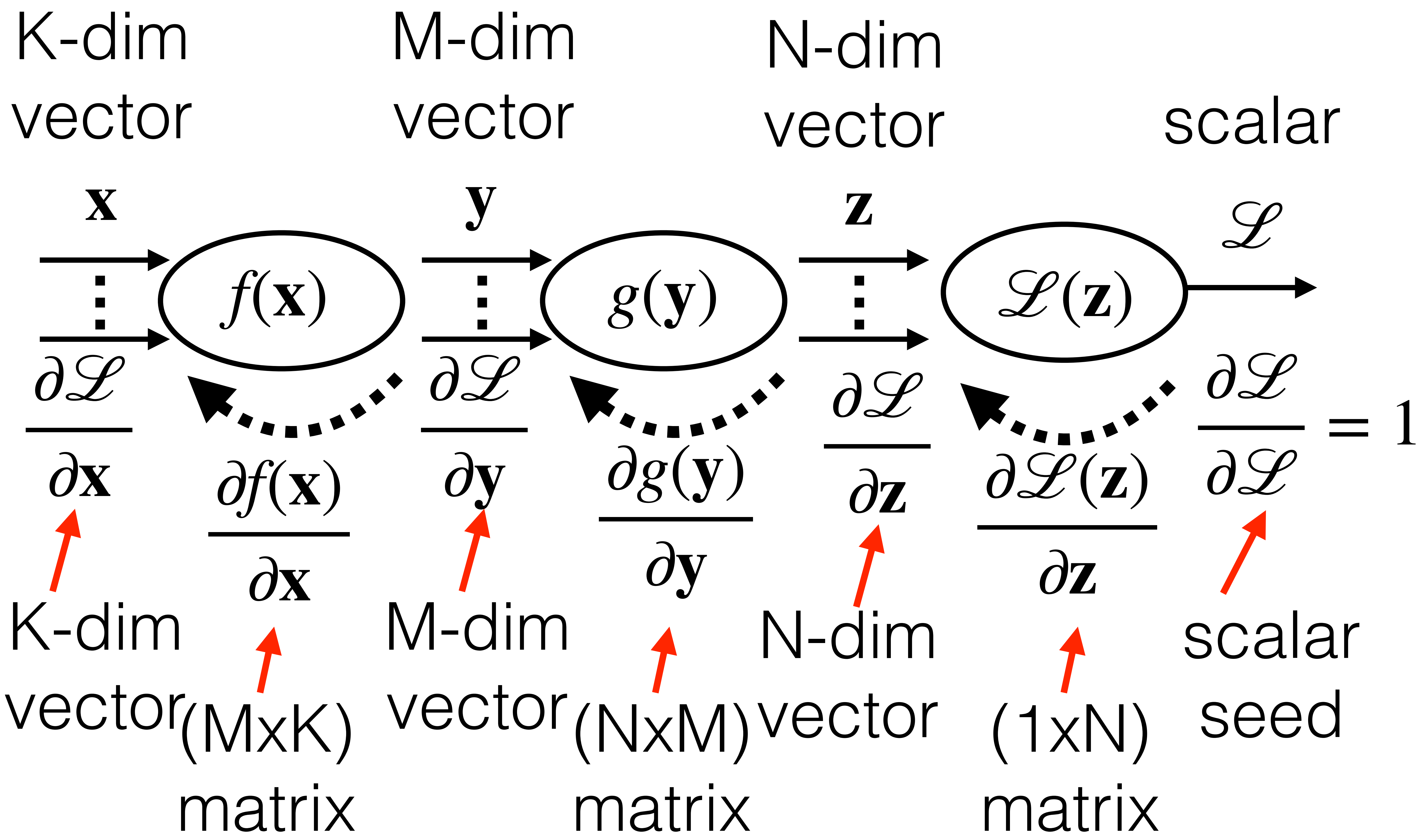
Efficient implementation of autodiff?



Loss jac wrt $\mathbf{x}$

$$\overset{1\text{x}K}{\frac{\partial \mathscr{L}}{\partial \mathbf{x}}} = \overset{1\text{x}1}{1}\ \overset{1\text{x}N}{\frac{\partial \mathscr{L}}{\partial \mathbf{z}}}\ \overset{N\text{x}M}{\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}}\ \overset{M\text{x}K}{\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}}$$

Efficient implementation of autodiff?



def $\mathrm{vjp}_{\mathscr{L}}(\mathbf{v}, \mathbf{z})$:
  return $\mathbf{v} \cdot \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}}$

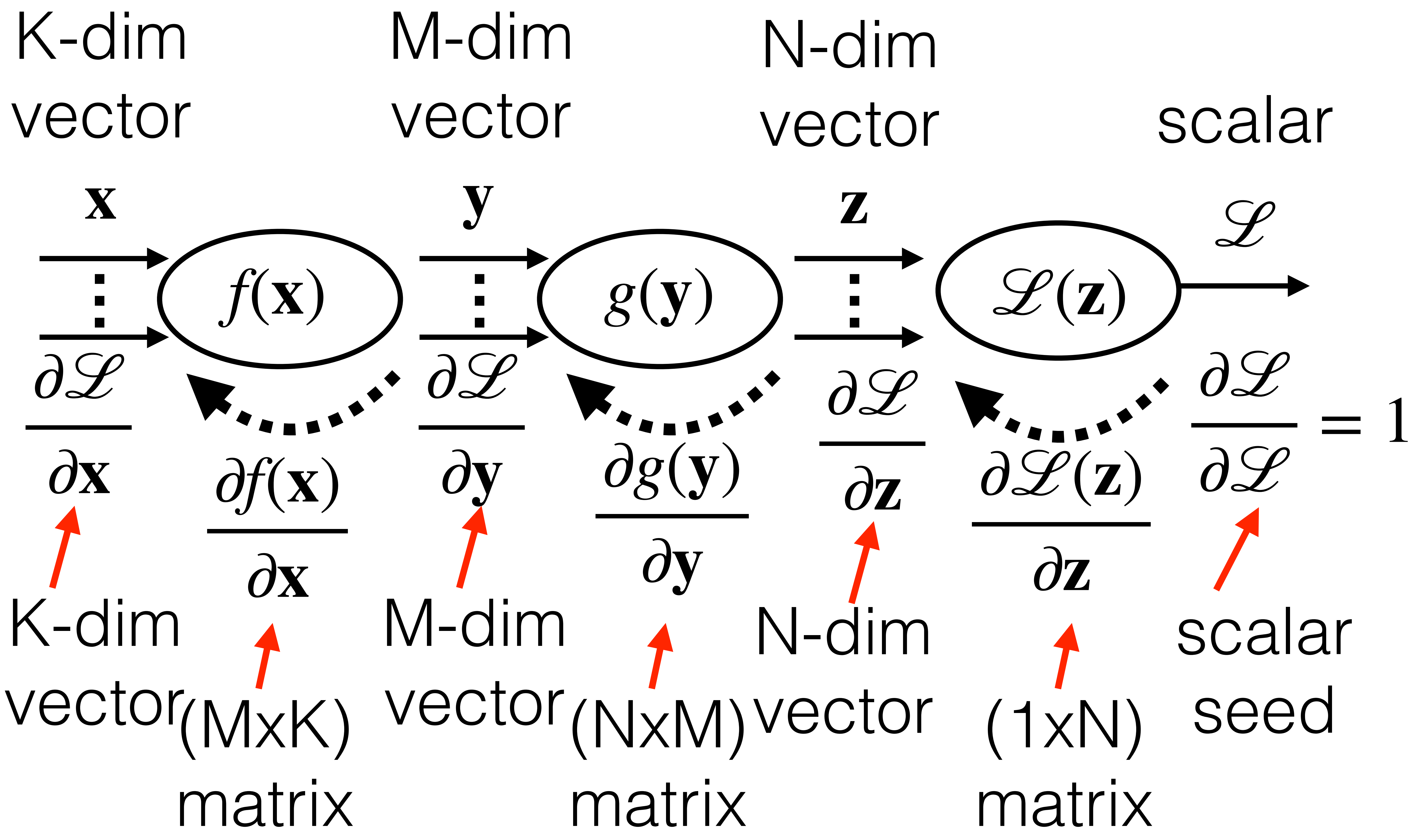
def $\mathrm{vjp}_{g}(\mathbf{v}, \mathbf{y})$:
  return $\mathbf{v} \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

def $\mathrm{vjp}_{f}(\mathbf{v}, \mathbf{x})$:
  return $\mathbf{v} \cdot \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$

Loss jac wrt $\mathbf{x}$

$$\underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{x}}}_{1\text{x}K} = \underbrace{1}_{1\text{x}1}\ \underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{z}}}_{1\text{x}N}\ \underbrace{\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}}_{N\text{x}M}\ \underbrace{\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}}_{M\text{x}K}$$

Efficient implementation of autodiff?



def $\mathrm{vjp}_{\mathscr{L}}(\mathbf{v}, \mathbf{z})$:
    return $\mathbf{v} \cdot \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}}$

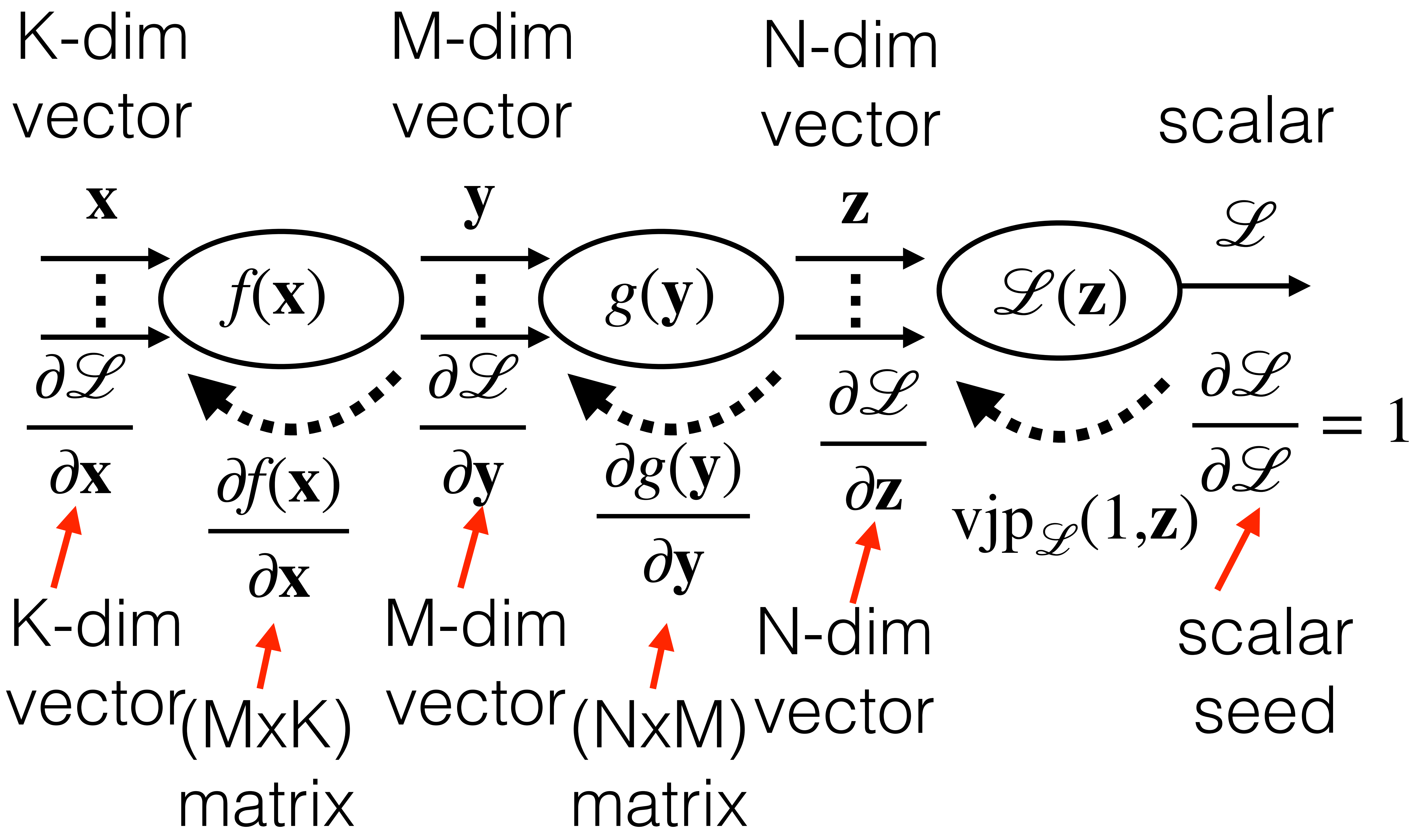
def $\mathrm{vjp}_{g}(\mathbf{v}, \mathbf{y})$:
    return $\mathbf{v} \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

def $\mathrm{vjp}_{f}(\mathbf{v}, \mathbf{x})$:
    return $\mathbf{v} \cdot \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$

Loss jac wrt $\mathbf{x}$

$$\underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{x}}}_{1\times K} = \underbrace{\underbrace{1}_{1\times 1}\ \underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{z}}}_{1\times N}}_{\mathrm{vjp}_{\mathscr{L}}(1,\mathbf{z})}\ \underbrace{\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}}_{N\times M}\ \underbrace{\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}}_{M\times K}$$

Efficient implementation of autodiff?



def $\text{vjp}_{\mathscr{L}}(\mathbf{v}, \mathbf{z})$:
    return $\mathbf{v} \cdot \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}}$

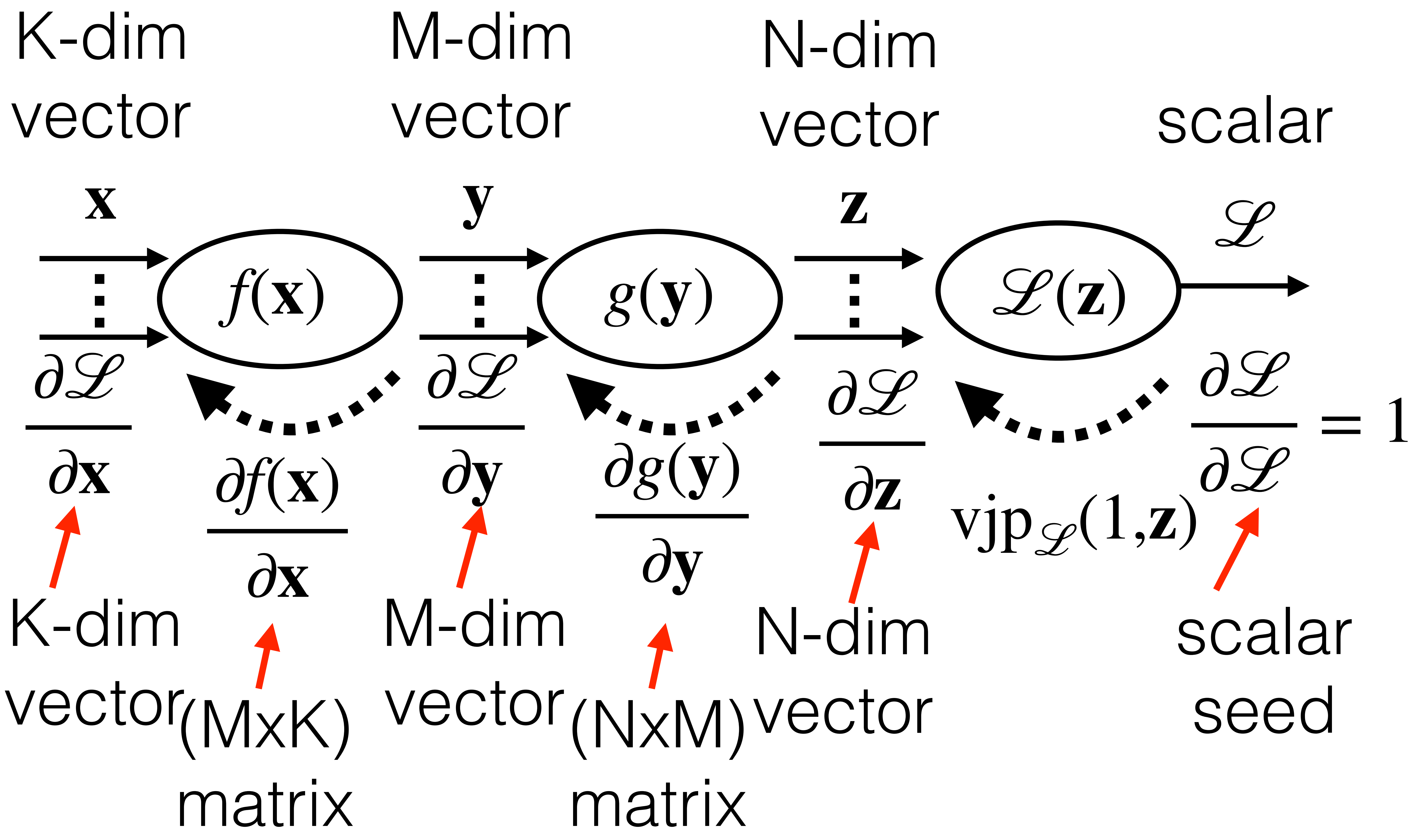
def $\text{vjp}_{g}(\mathbf{v}, \mathbf{y})$:
    return $\mathbf{v} \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

def $\text{vjp}_{f}(\mathbf{v}, \mathbf{x})$:
    return $\mathbf{v} \cdot \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$

Loss jac wrt $\mathbf{x}$

$$\underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{x}}}_{1\text{x}K} = \text{vjp}_{\mathscr{L}}(1,\mathbf{z}) \underbrace{\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}}_{\text{NxM}} \underbrace{\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}}_{\text{MxK}}$$

Efficient implementation of autodiff?



def $\text{vjp}_{\mathscr{L}}(\mathbf{v}, \mathbf{z})$:
    return $\mathbf{v} \cdot \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}}$

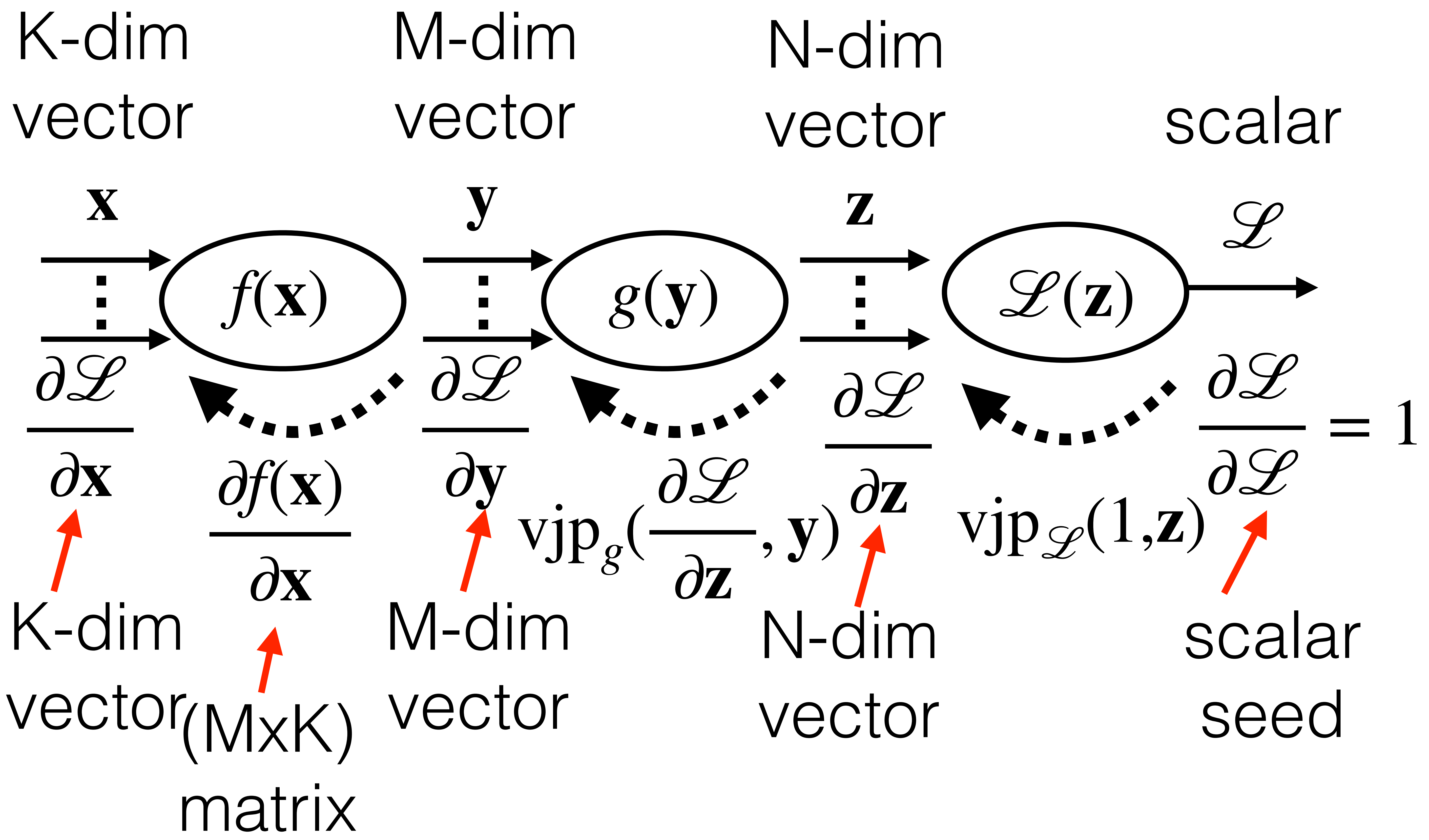
def $\text{vjp}_{g}(\mathbf{v}, \mathbf{y})$:
    return $\mathbf{v} \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

def $\text{vjp}_{f}(\mathbf{v}, \mathbf{x})$:
    return $\mathbf{v} \cdot \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$

Loss jac wrt $\mathbf{x}$

$$\underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{x}}}_{1\text{x}K} = \text{vjp}_{\mathscr{L}}(1,\mathbf{z}) \underbrace{\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}}_{N\text{x}M} \underbrace{\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}}_{M\text{x}K}$$

Efficient implementation of autodiff?




```
def vjp_L(v, z):
    return v · ∂L(z)/∂z
```

$$\texttt{def } \mathrm{vjp}_{\mathscr{L}}(\mathbf{v}, \mathbf{z}): \quad \texttt{return } \mathbf{v} \cdot \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}}$$

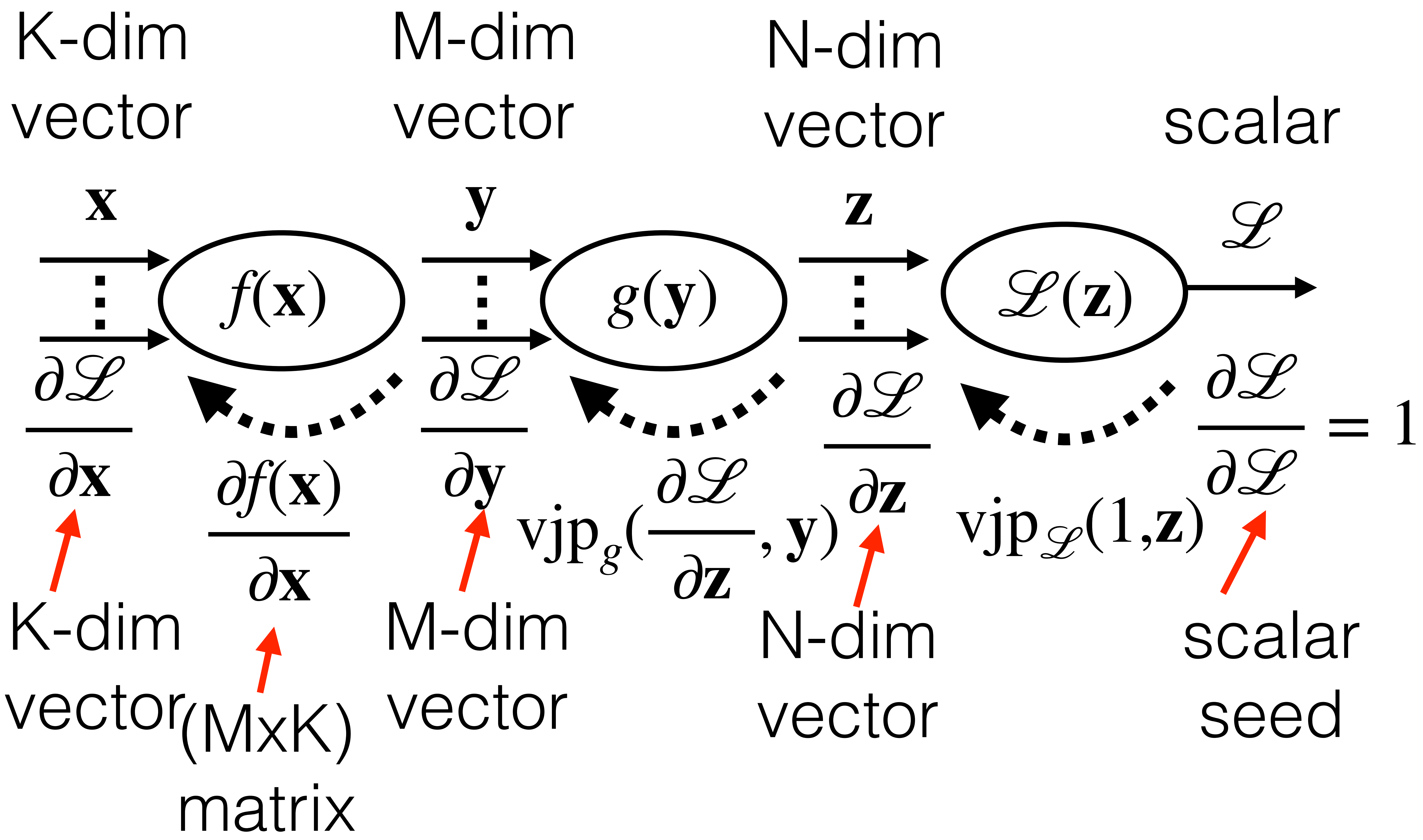
$$\texttt{def } \mathrm{vjp}_{g}(\mathbf{v}, \mathbf{y}): \quad \texttt{return } \mathbf{v} \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$$

$$\texttt{def } \mathrm{vjp}_{f}(\mathbf{v}, \mathbf{x}): \quad \texttt{return } \mathbf{v} \cdot \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$$

1xK | | MxK

Loss jac wrt $\mathbf{x}$ $\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \mathrm{vjp}_g\left(\mathrm{vjp}_{\mathscr{L}}(1, \mathbf{z}), \mathbf{y}\right) \; \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$

Efficient implementation of autodiff?



def $\mathrm{vjp}_{\mathscr{L}}(\mathbf{v}, \mathbf{z})$:
    return $\mathbf{v} \cdot \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}}$

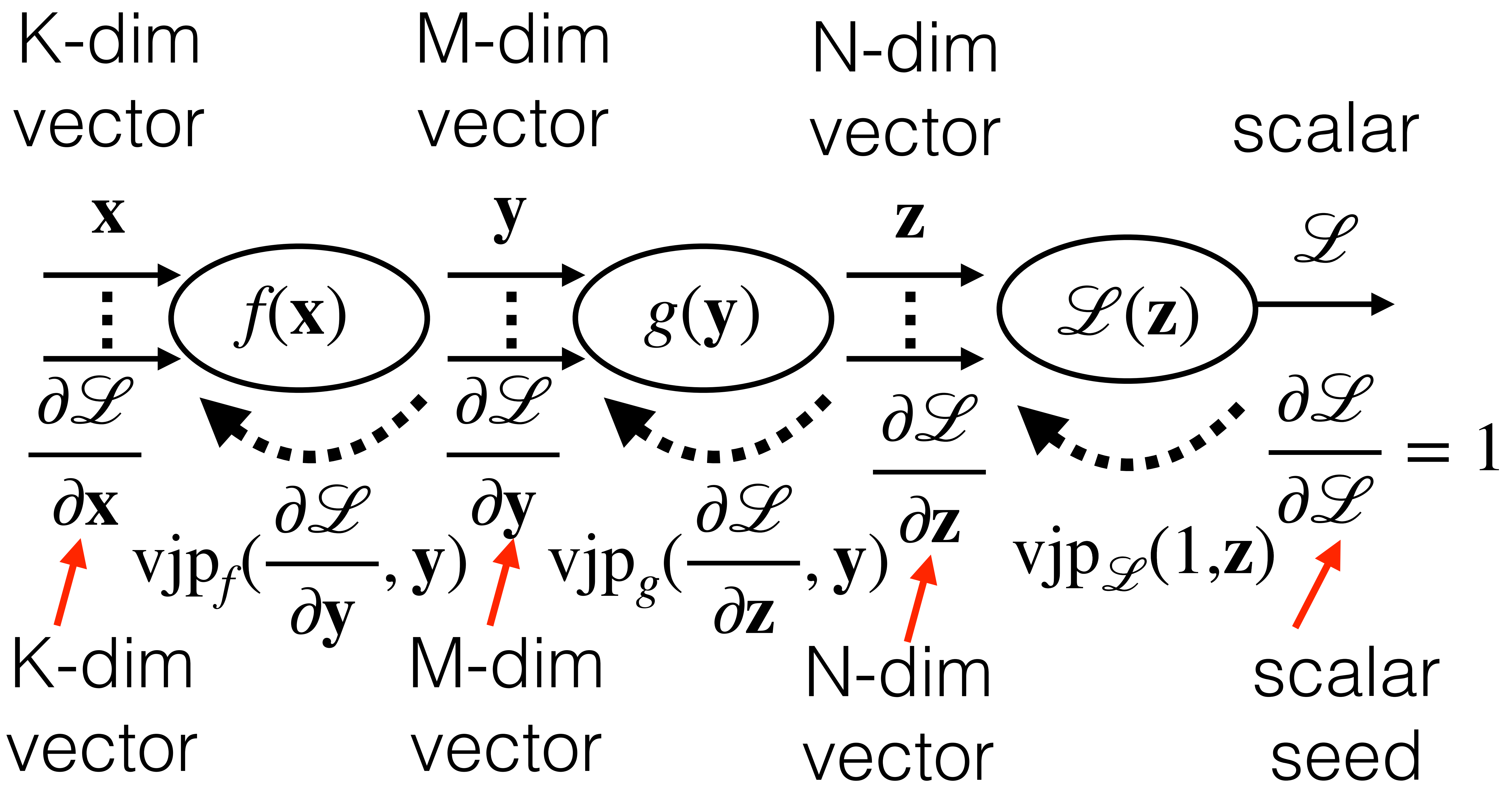
def $\mathrm{vjp}_{g}(\mathbf{v}, \mathbf{y})$:
    return $\mathbf{v} \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

def $\mathrm{vjp}_{f}(\mathbf{v}, \mathbf{x})$:
    return $\mathbf{v} \cdot \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$

1xK | MxK

Loss jac wrt $\mathbf{x}$ $\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \mathrm{vjp}_g\left(\mathrm{vjp}_{\mathscr{L}}(1,\mathbf{z}), \mathbf{y}\right) \quad \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$

Efficient implementation of autodiff?




```
def vjp_L(v, z):
    return v · ∂L(z)/∂z
```

$$\texttt{def } \mathrm{vjp}_{\mathscr{L}}(\mathbf{v},\mathbf{z}): \quad \texttt{return } \mathbf{v}\cdot\frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}}$$

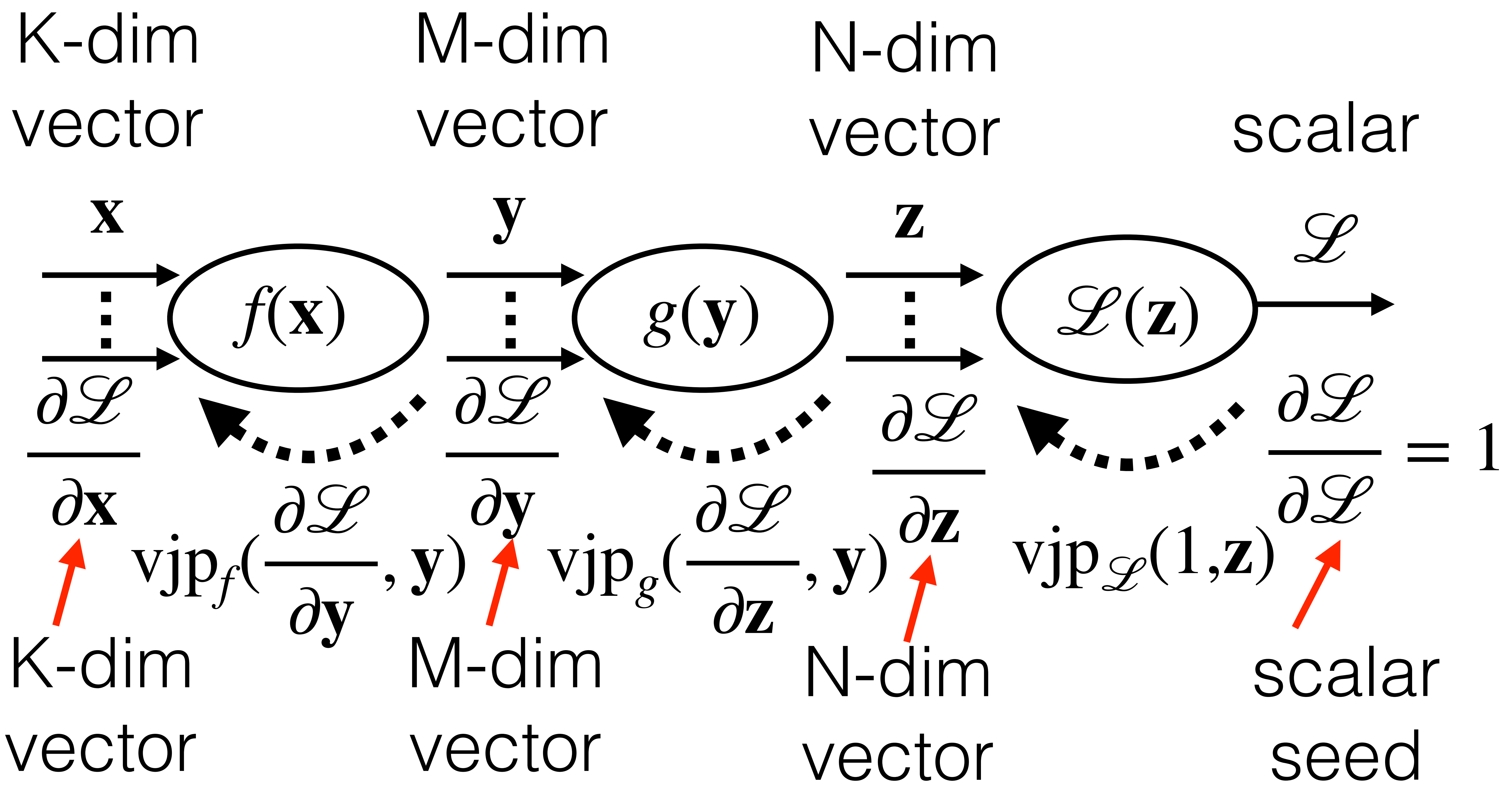
$$\texttt{def } \mathrm{vjp}_{g}(\mathbf{v},\mathbf{y}): \quad \texttt{return } \mathbf{v}\cdot\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$$

$$\texttt{def } \mathrm{vjp}_{f}(\mathbf{v},\mathbf{x}): \quad \texttt{return } \mathbf{v}\cdot\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$$

1xK

Loss jac wrt $\mathbf{x}$ $\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \mathrm{vjp}_f\Big(\mathrm{vjp}_g\big(\mathrm{vjp}_{\mathscr{L}}(1,\mathbf{z}),\mathbf{y}\big),\mathbf{x}\Big)$

Efficient implementation of autodiff?



def $\text{vjp}_{\mathscr{L}}(\mathbf{v}, \mathbf{z})$:

    return $\mathbf{v} \cdot \frac{\partial \mathscr{L}(\mathbf{z})}{\partial \mathbf{z}}$

def $\text{vjp}_g(\mathbf{v}, \mathbf{y})$:

    return $\mathbf{v} \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

def $\text{vjp}_f(\mathbf{v}, \mathbf{x})$:

    return $\mathbf{v} \cdot \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$

Reverse autodiff algorithm

input: $\mathbf{x}$

forward pass:

$$\mathbf{y} = f(\mathbf{x})$$
$$\mathbf{z} = g(\mathbf{y})$$
$$\mathscr{L} = \mathscr{L}(\mathbf{z})$$

backward pass:

$$\mathbf{v} = \text{vjp}_{\mathscr{L}}(1,\mathbf{z})$$
$$\mathbf{v} = \text{vjp}_g(\mathbf{v}, \mathbf{y})$$
$$\mathbf{v} = \text{vjp}_f(\mathbf{v}, \mathbf{x})$$

output:

$$\mathbf{v} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}}$$

Example: Implementation

When you should define your own backward pass?

- **Inefficient** autodifferentiation (e.g. forward: y = x @ x.T, backward: vjp = v + v.T)

$$\xrightarrow{\mathbf{x}} \boxed{\mathbf{y} = \mathbf{x}\mathbf{x}^\top} \xrightarrow{\mathbf{y}}$$

- **Discontinuous** forward pass (define your own backward pass)

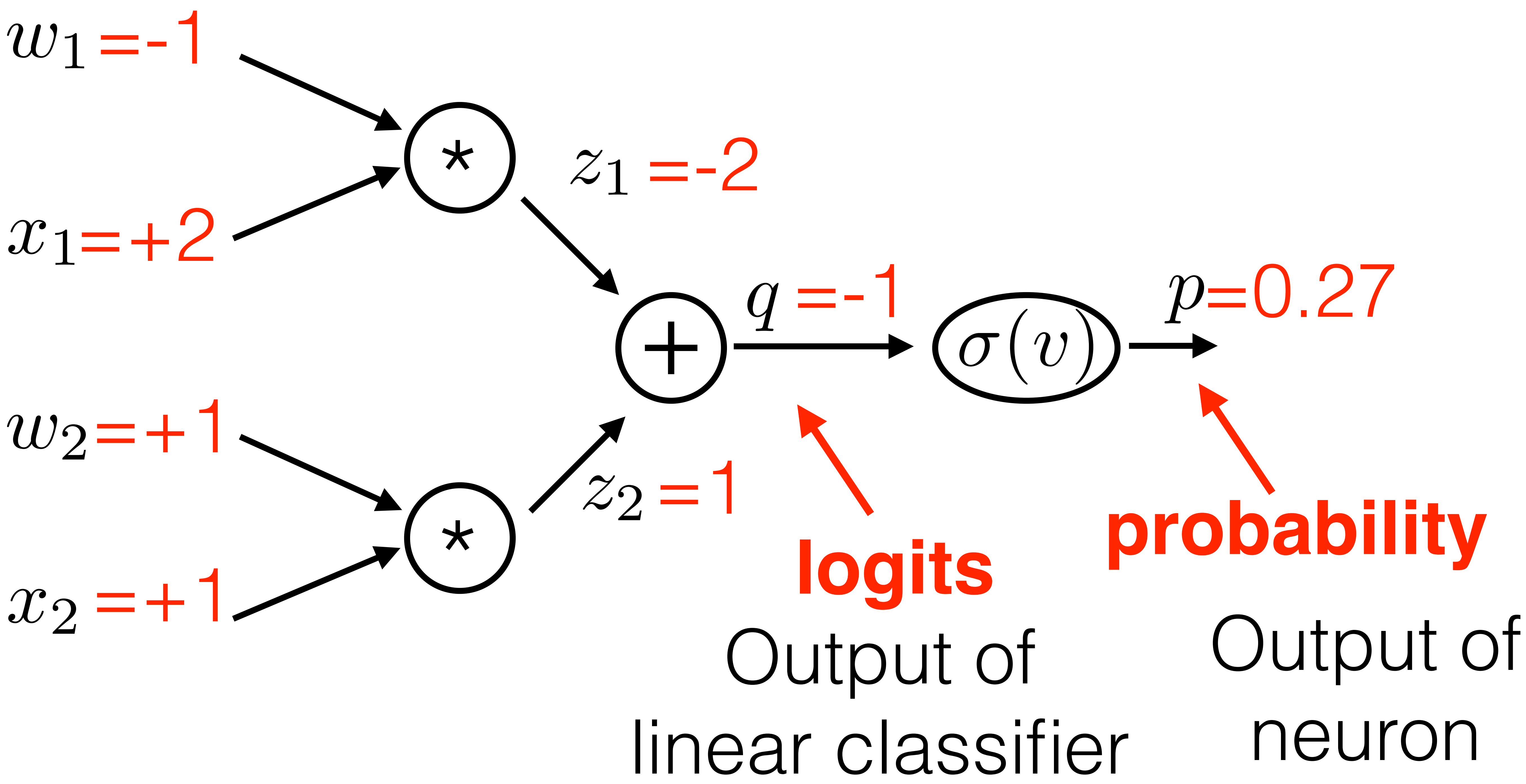
$$\xrightarrow{\mathbf{x}} \boxed{\mathbf{y} = \delta(\mathbf{x})} \xrightarrow{\mathbf{y}}$$

- **Custom** forward **operation** (e.g. forward: physics simulation, ODE/PDE solver, convex program solver, graph operations, backward: implicit gradient)

$$\xrightarrow{\omega} \boxed{\arg\min_{\mathbf{x}} f(\mathbf{x}, \omega)} \xrightarrow{\mathbf{x}^\star} \qquad \xrightarrow{\omega} \boxed{\mathrm{ODE}\big(\dot{\mathbf{x}} = f(\mathbf{x})\big)} \xrightarrow{\mathbf{x}^\star}$$

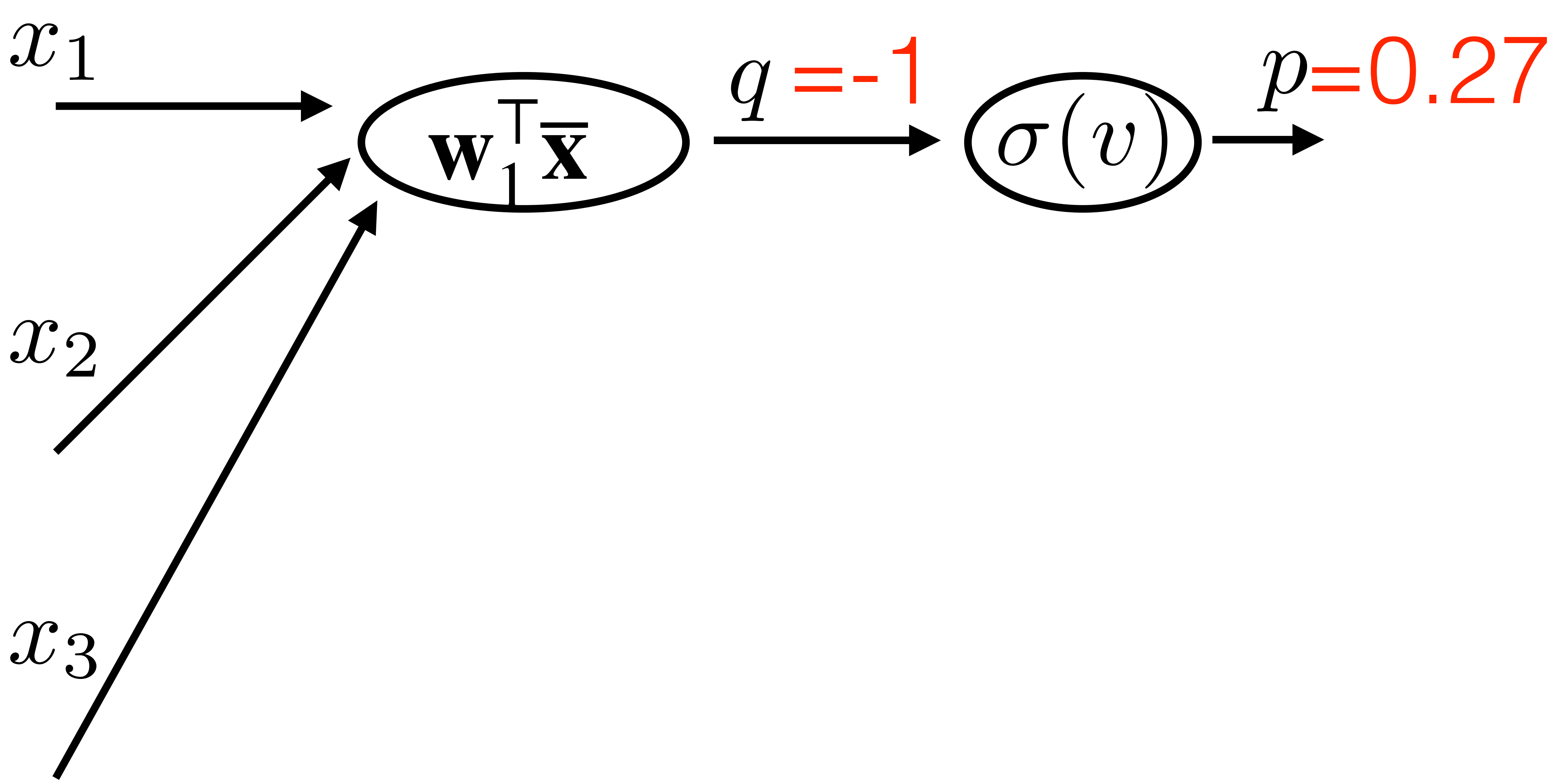
- **Unreliable** gradient (unstable or not pointing towards desired global optimum)

Neuron

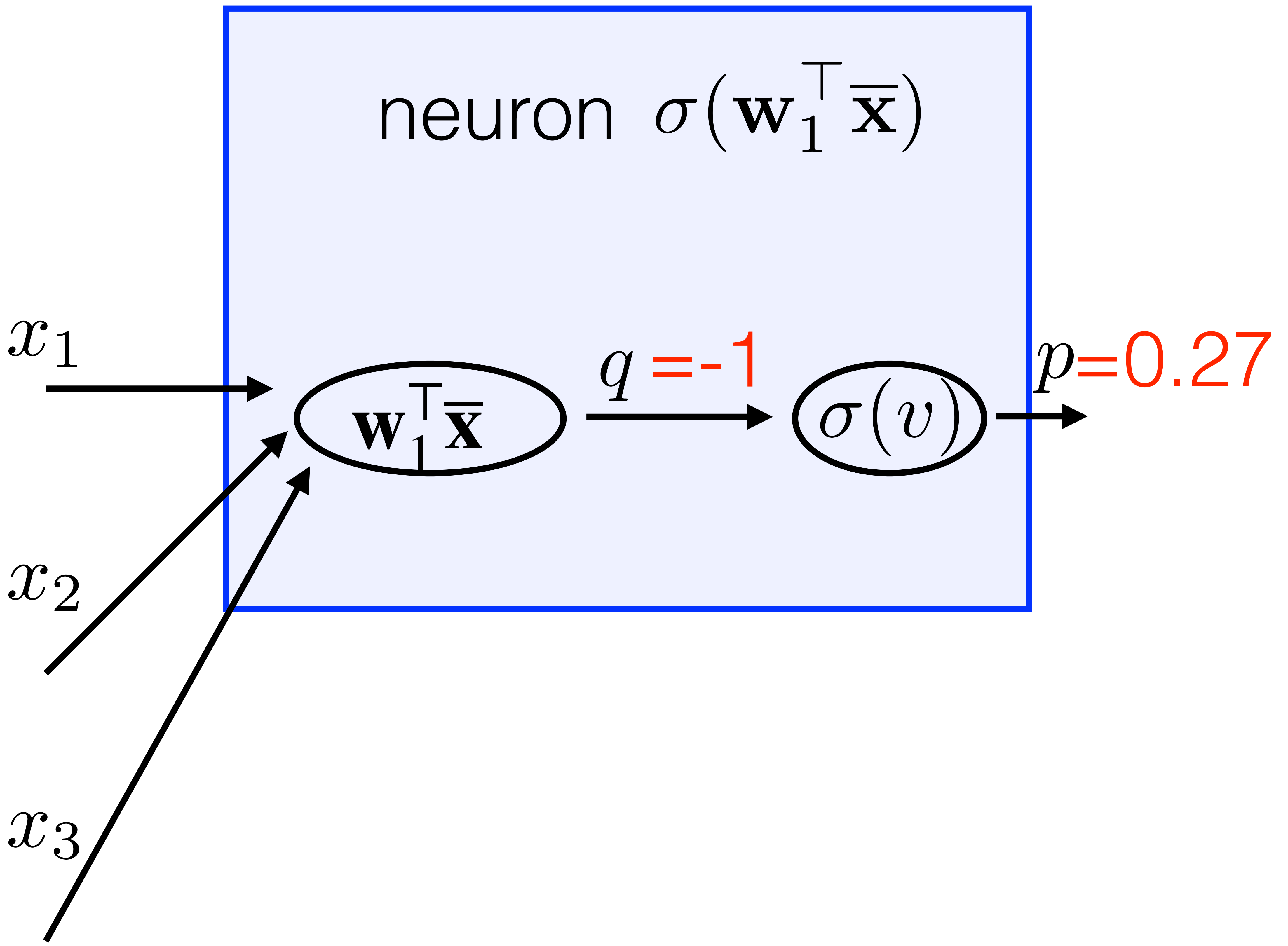


w_1=-1
x_1=+2
w_2=+1
x_2=+1
*
*
z_1 =-2
z_2 =1
+
q =-1
$\sigma(v)$
p=0.27
logits
Output of
linear classifier
probability
Output of
neuron

Neuron

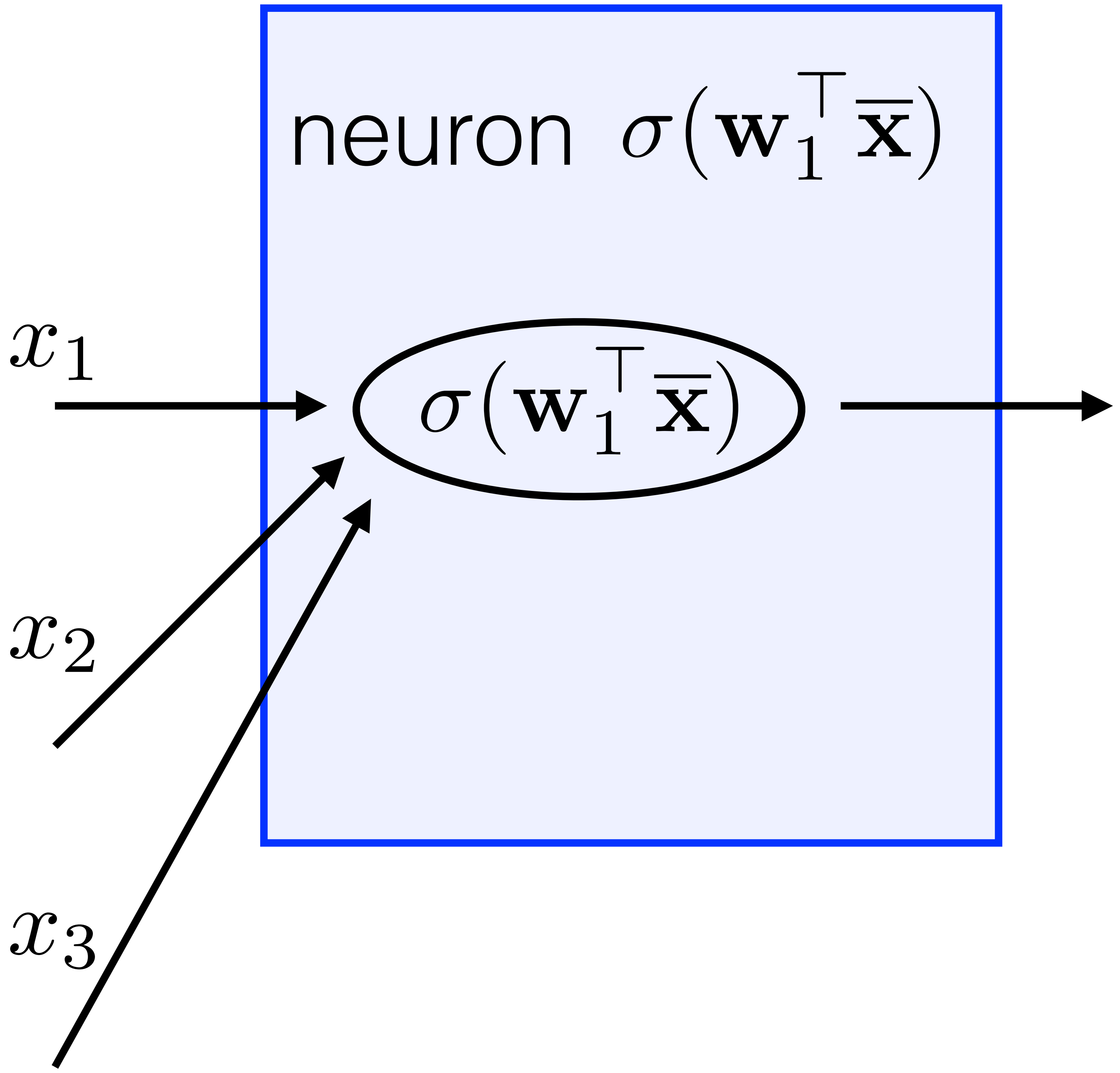


Neuron

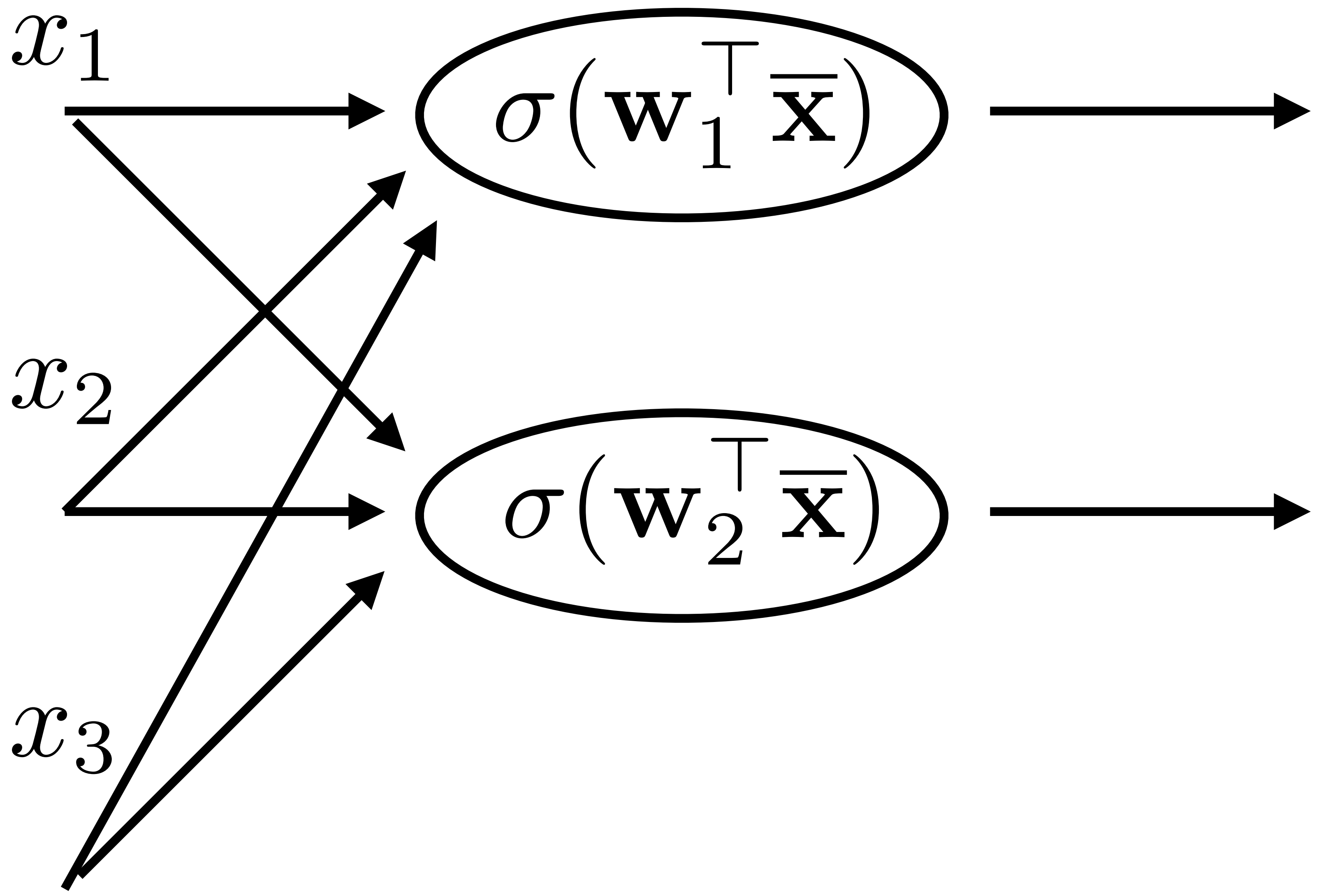


neuron $\sigma(\mathbf{w}_1^\top \overline{\mathbf{x}})$
x_1
x_2
x_3
$\mathbf{w}_1^\top \overline{\mathbf{x}}$
q =-1
$\sigma(v)$
p=0.27

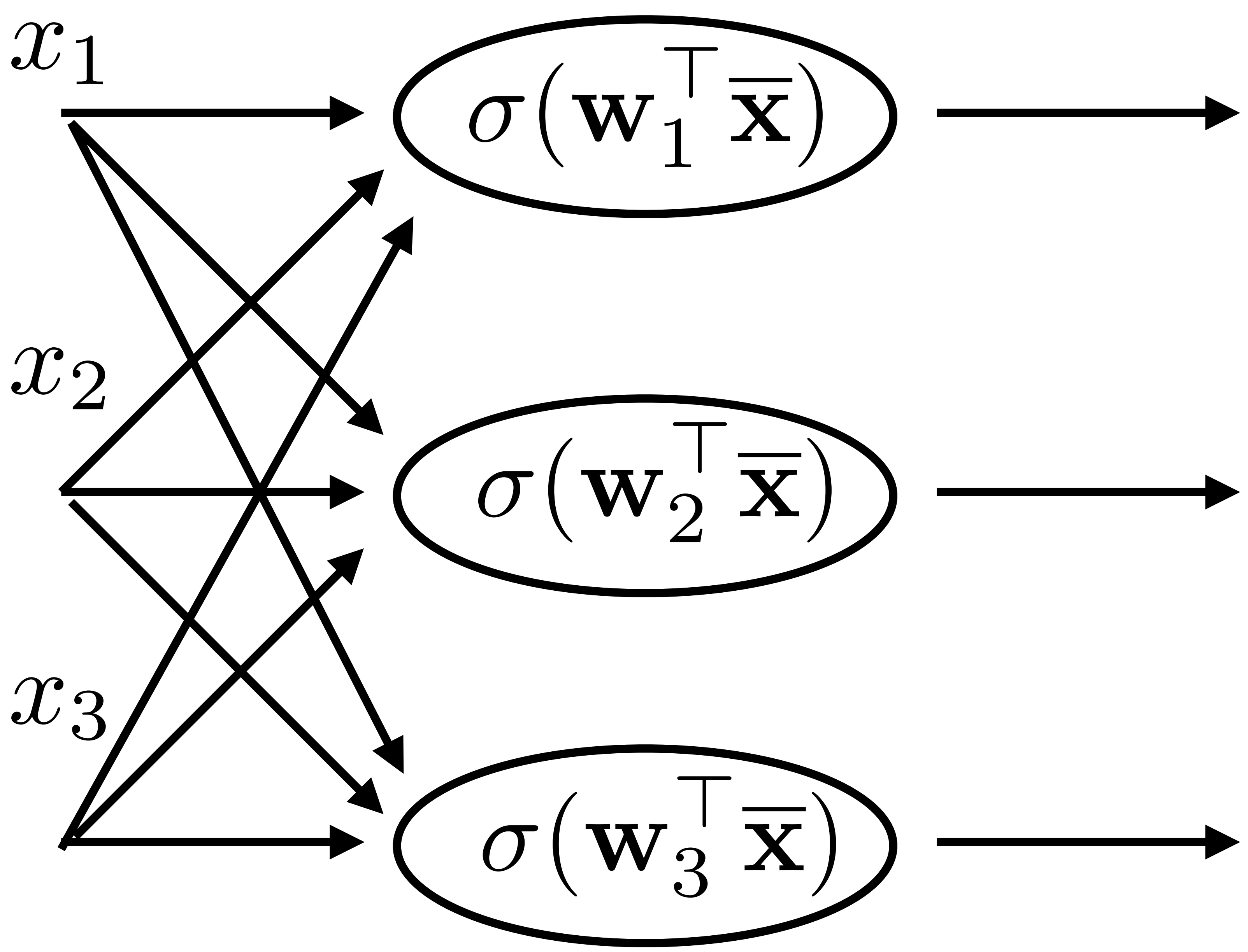
Fully-connected neural network



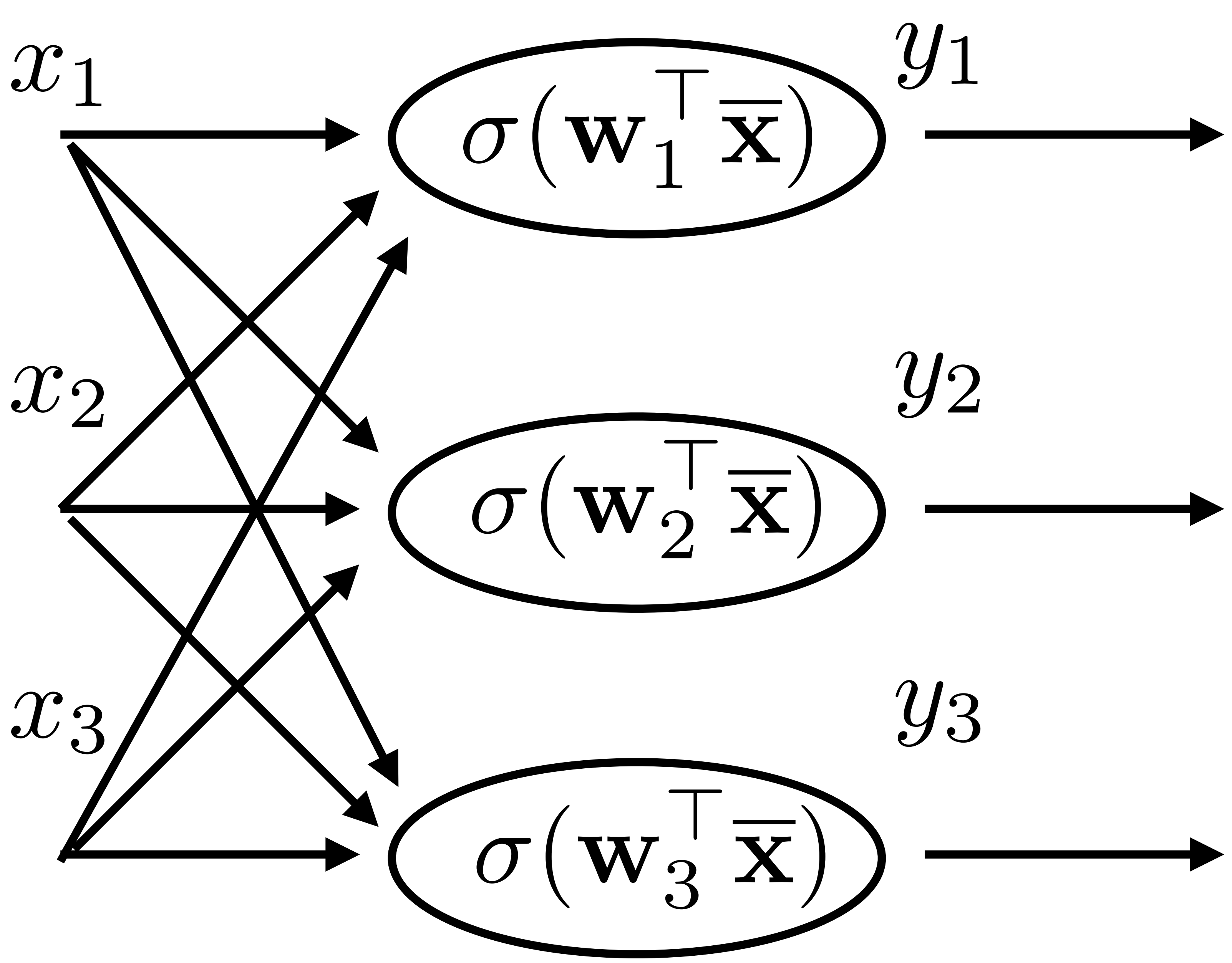
Fully-connected neural network



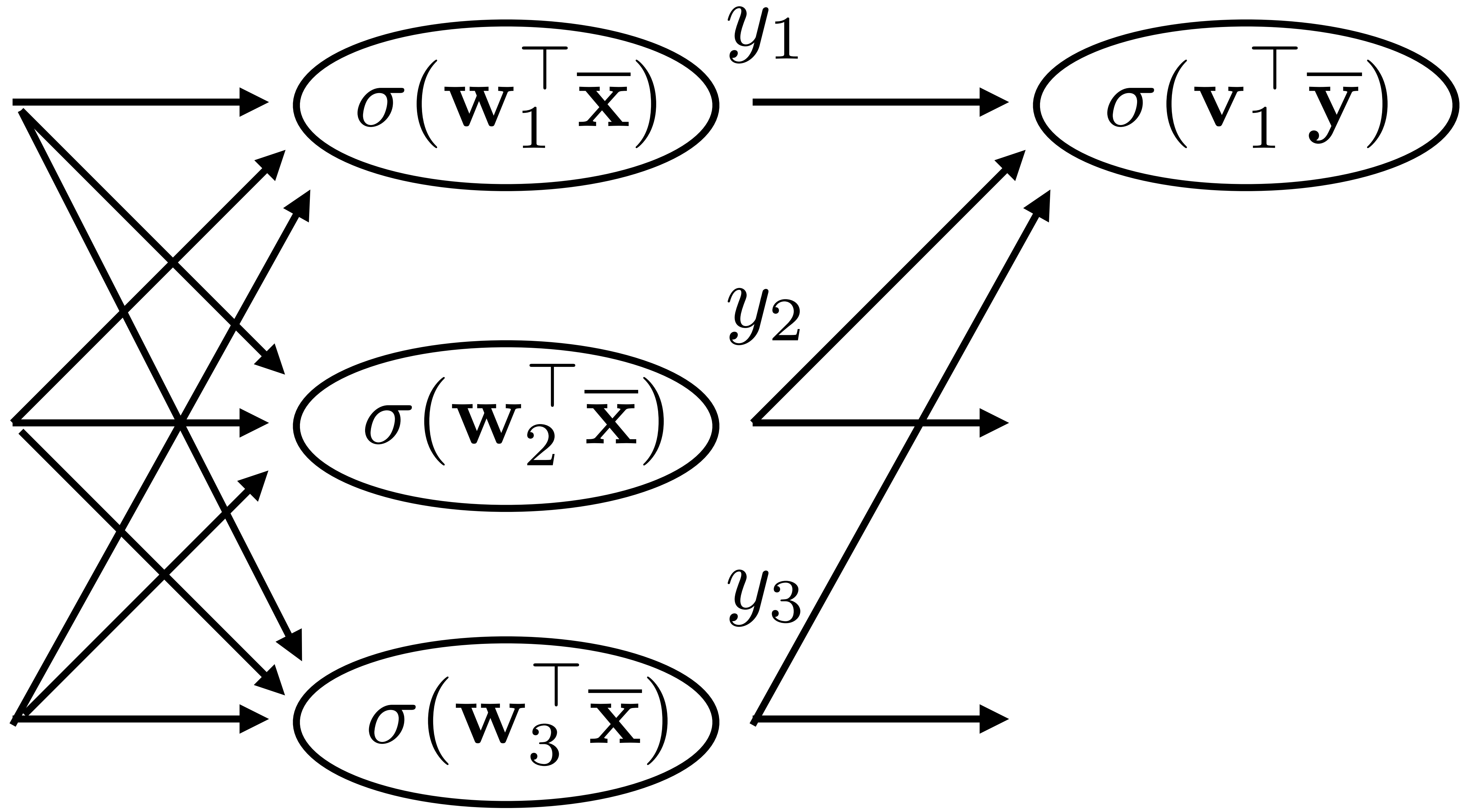
Fully-connected neural network



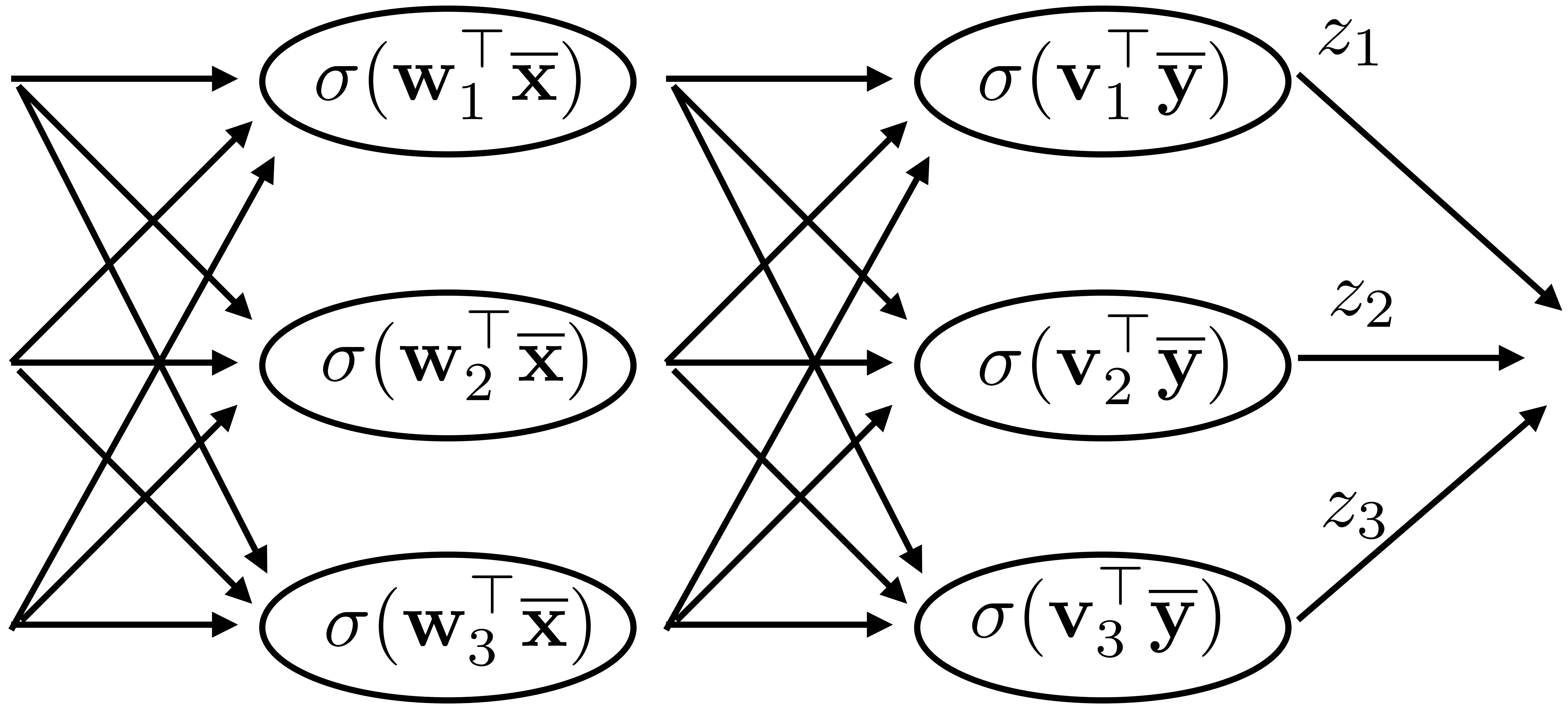
Fully-connected neural network



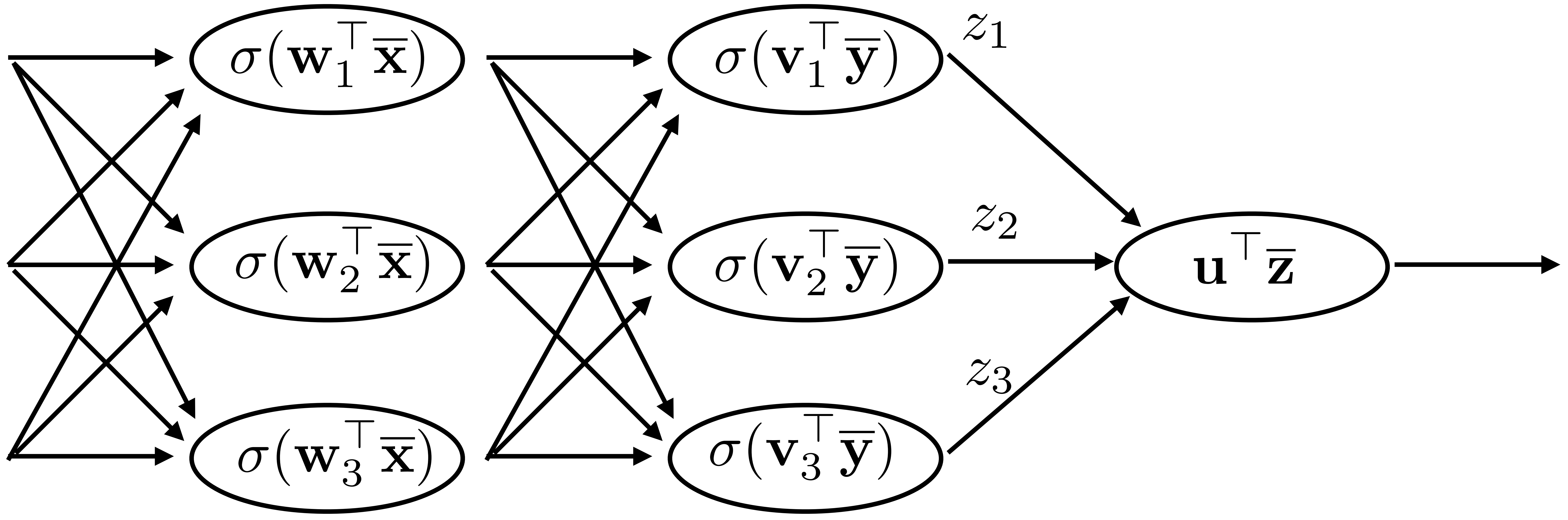
Fully-connected neural network



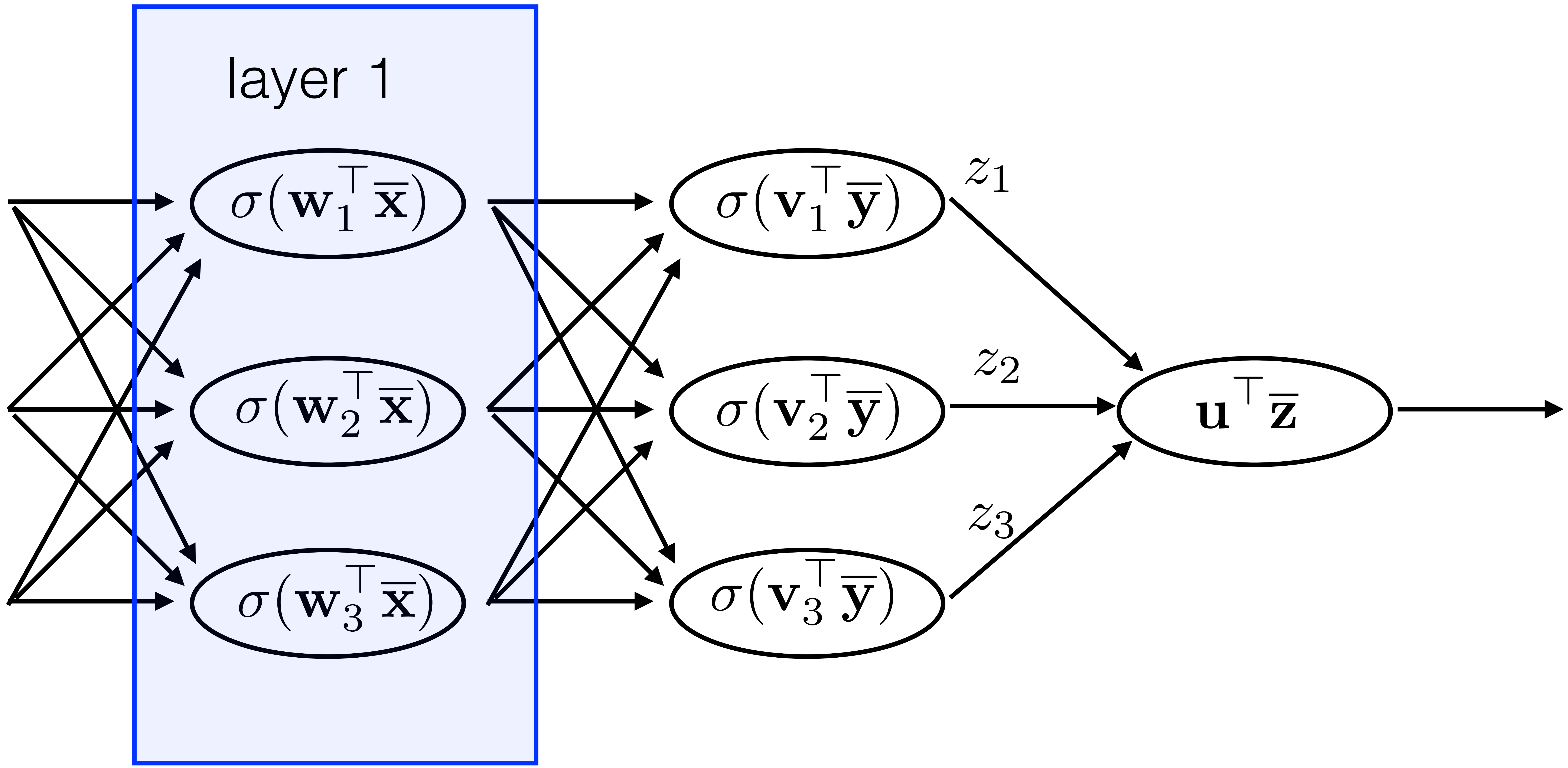
Fully-connected neural network



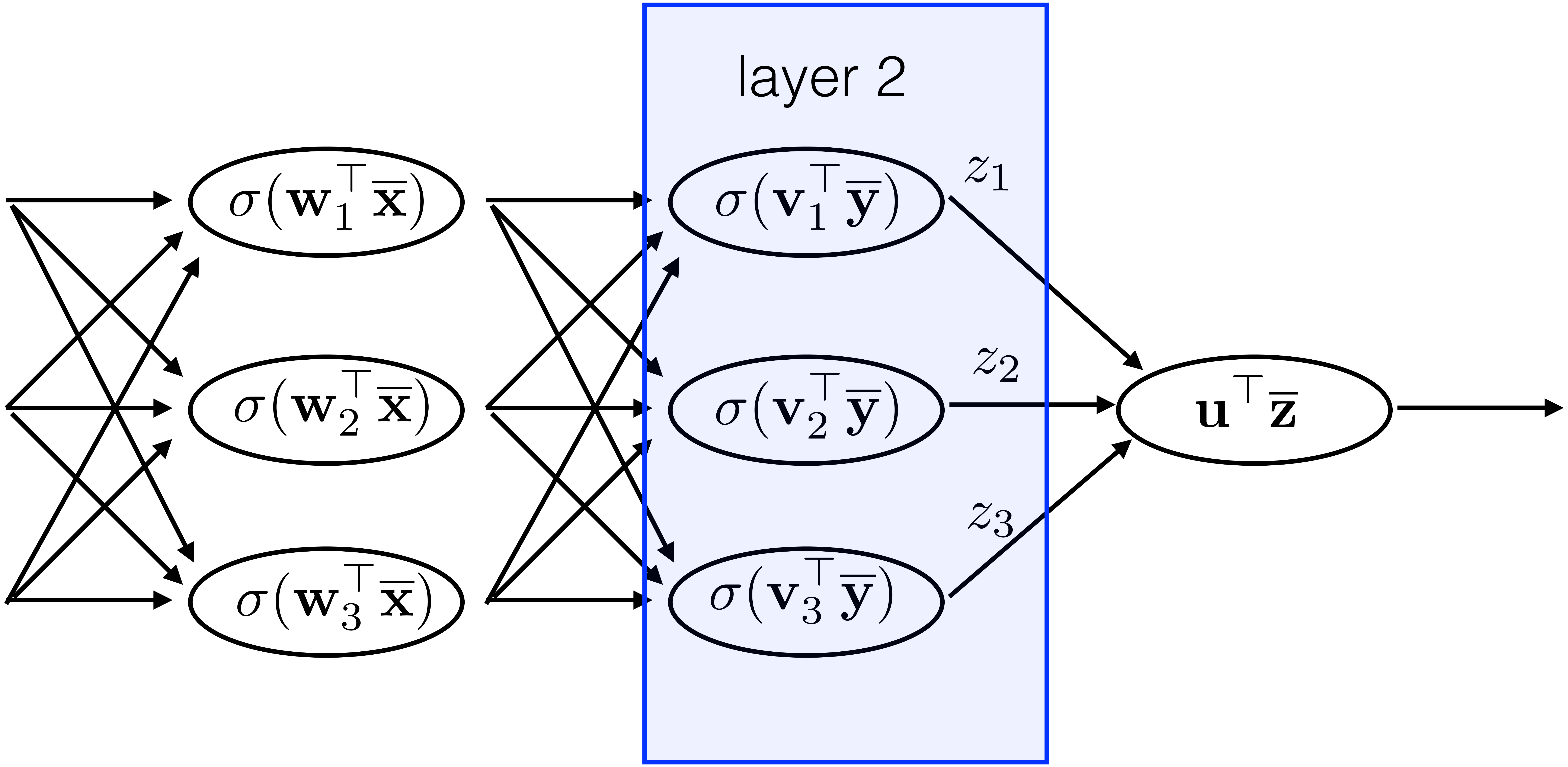
Fully-connected neural network



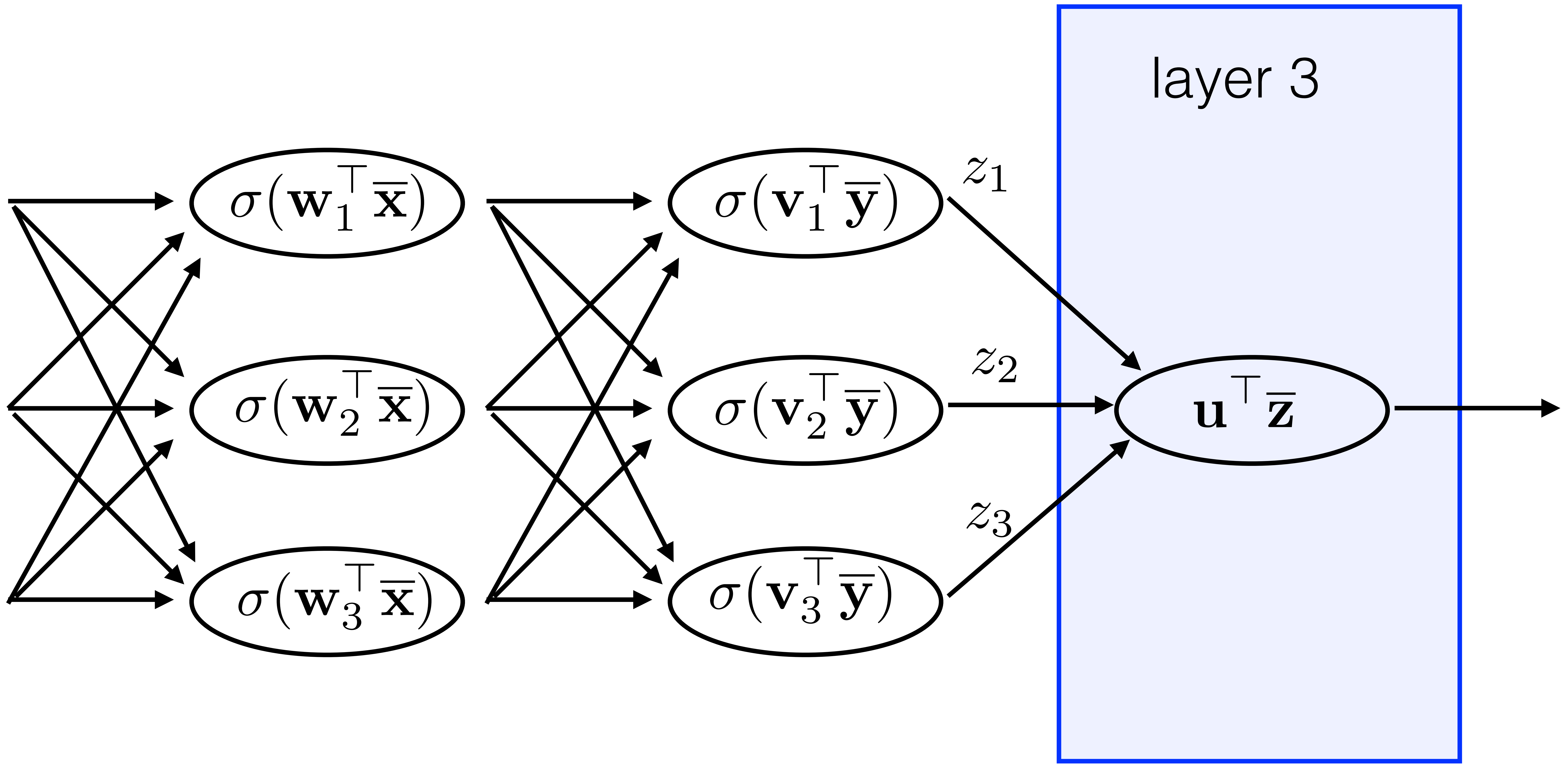
Fully-connected neural network



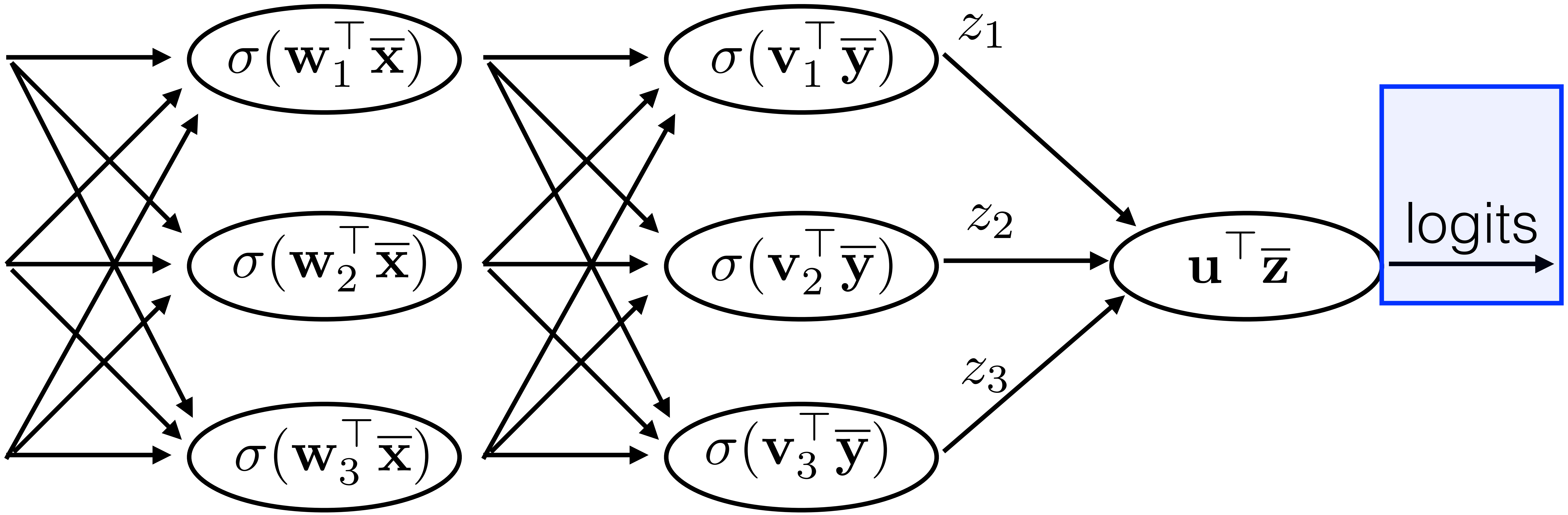
Fully-connected neural network



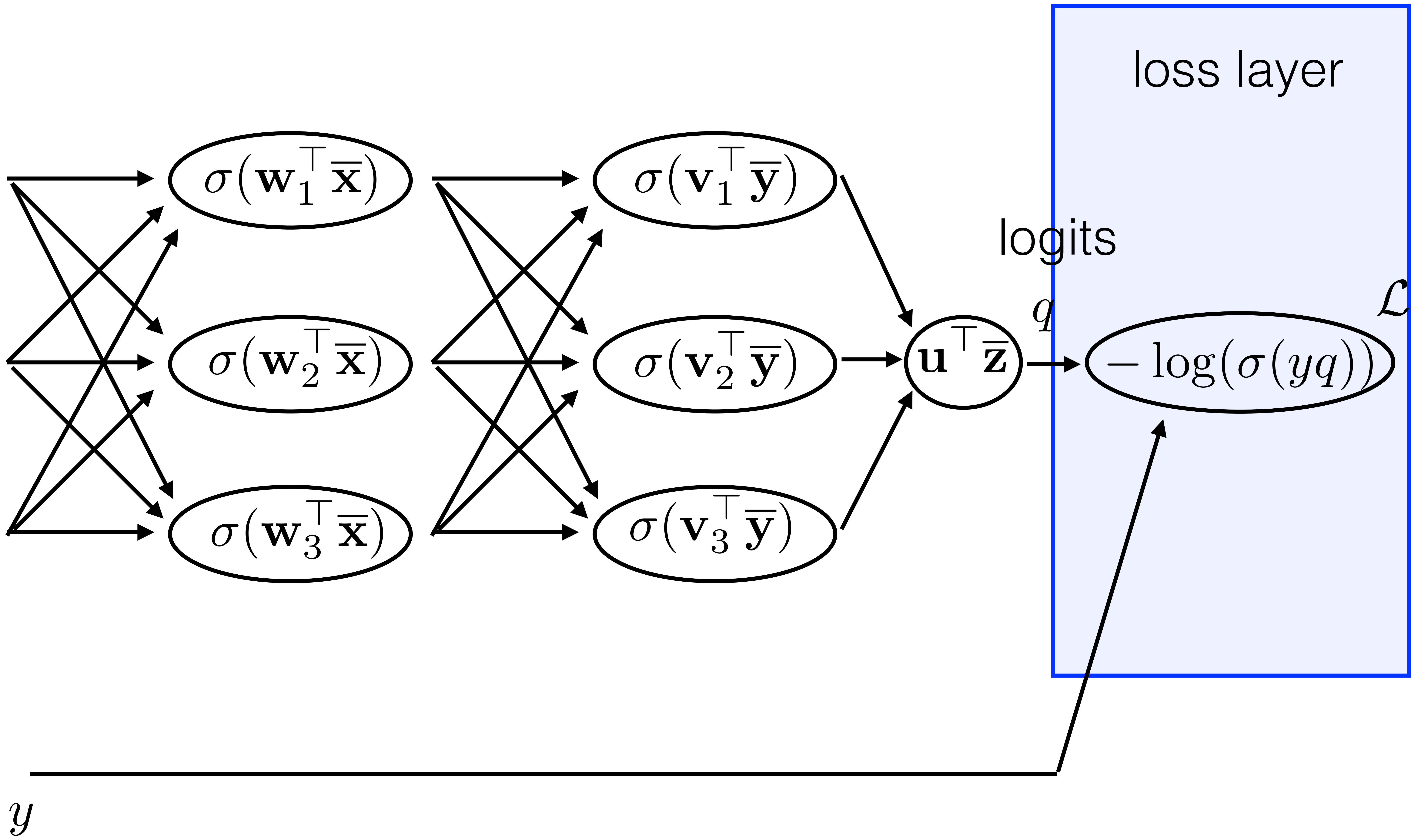
Fully-connected neural network



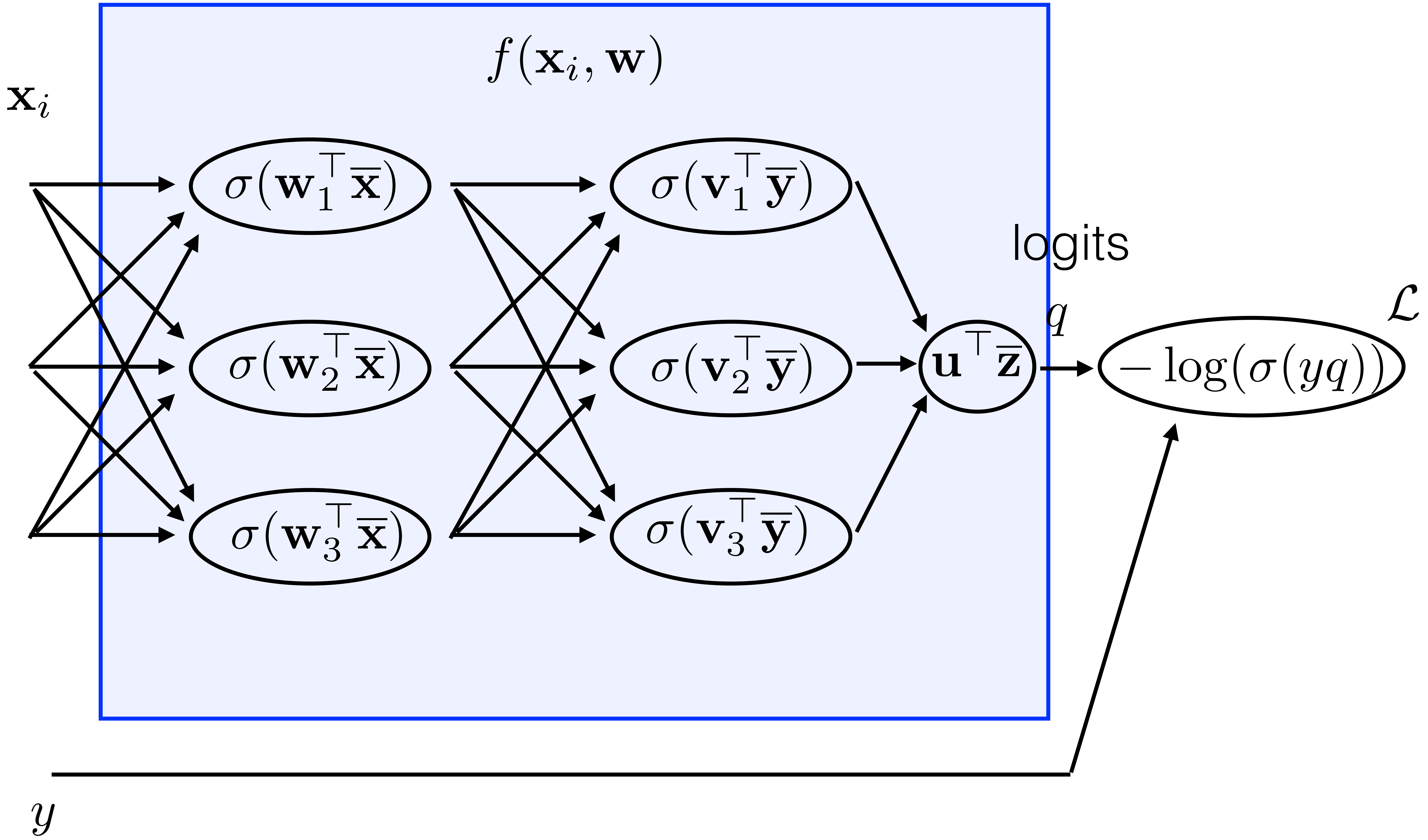
Fully-connected neural network



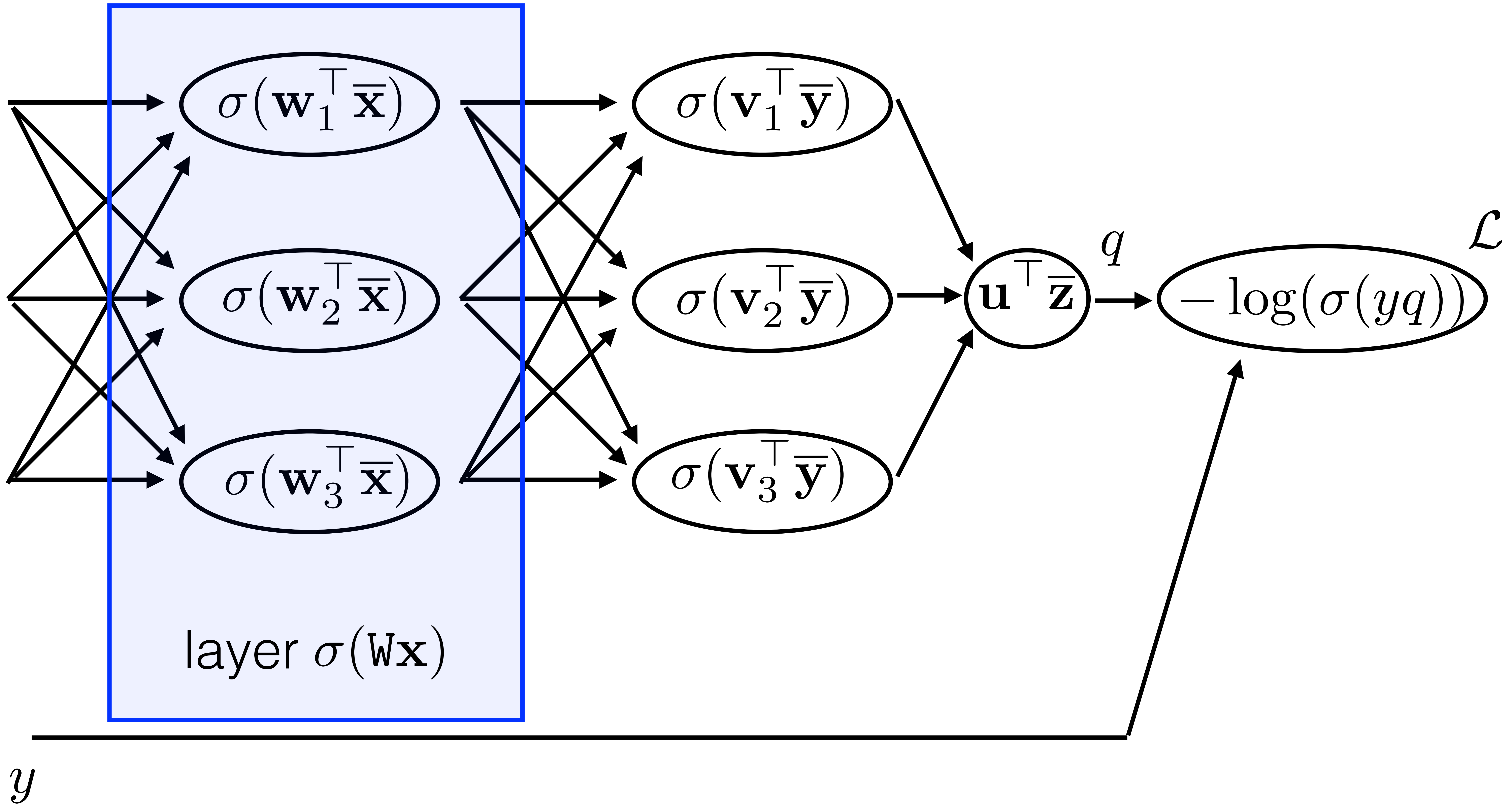
Fully-connected neural network



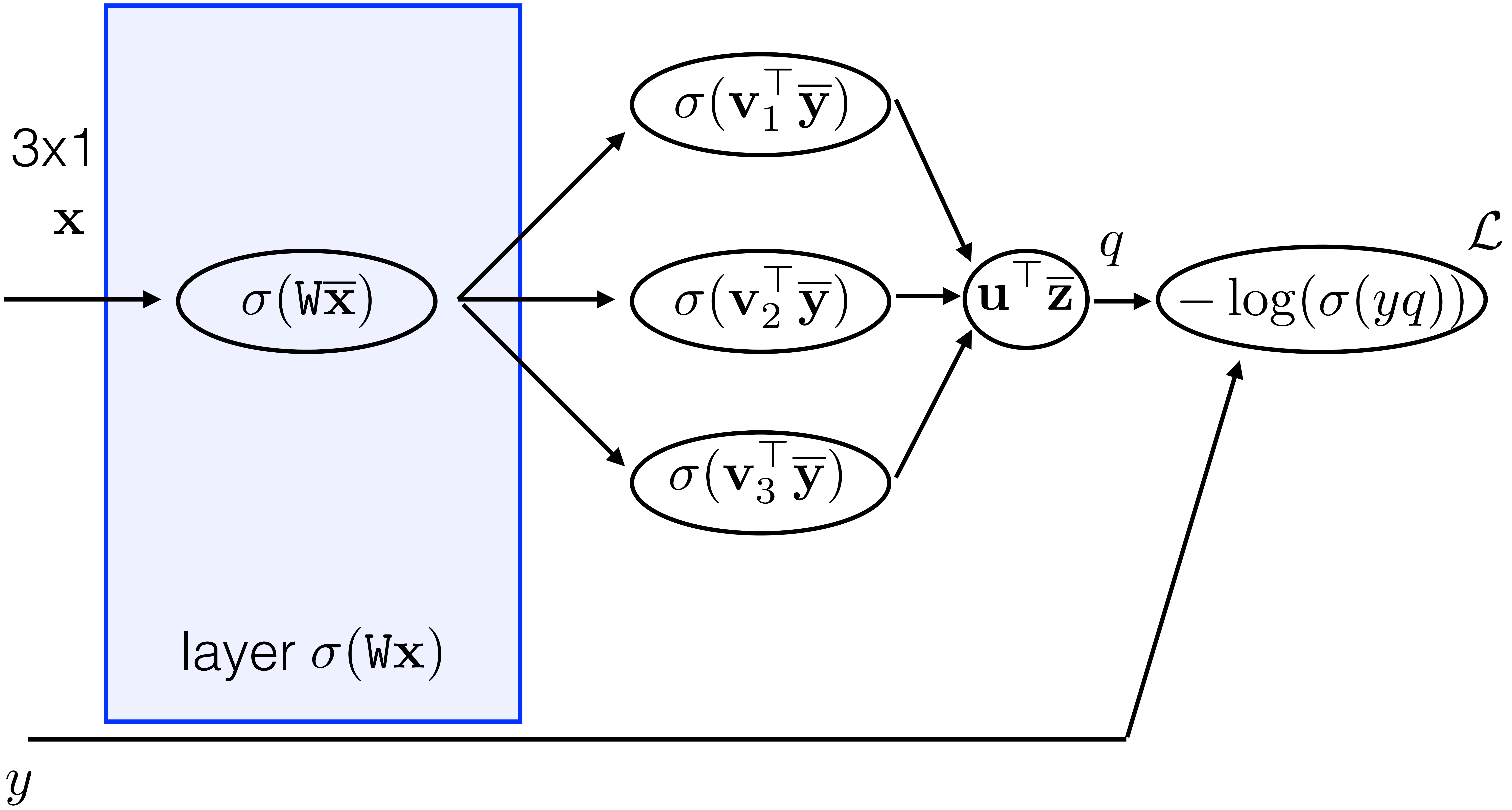
Fully-connected neural network



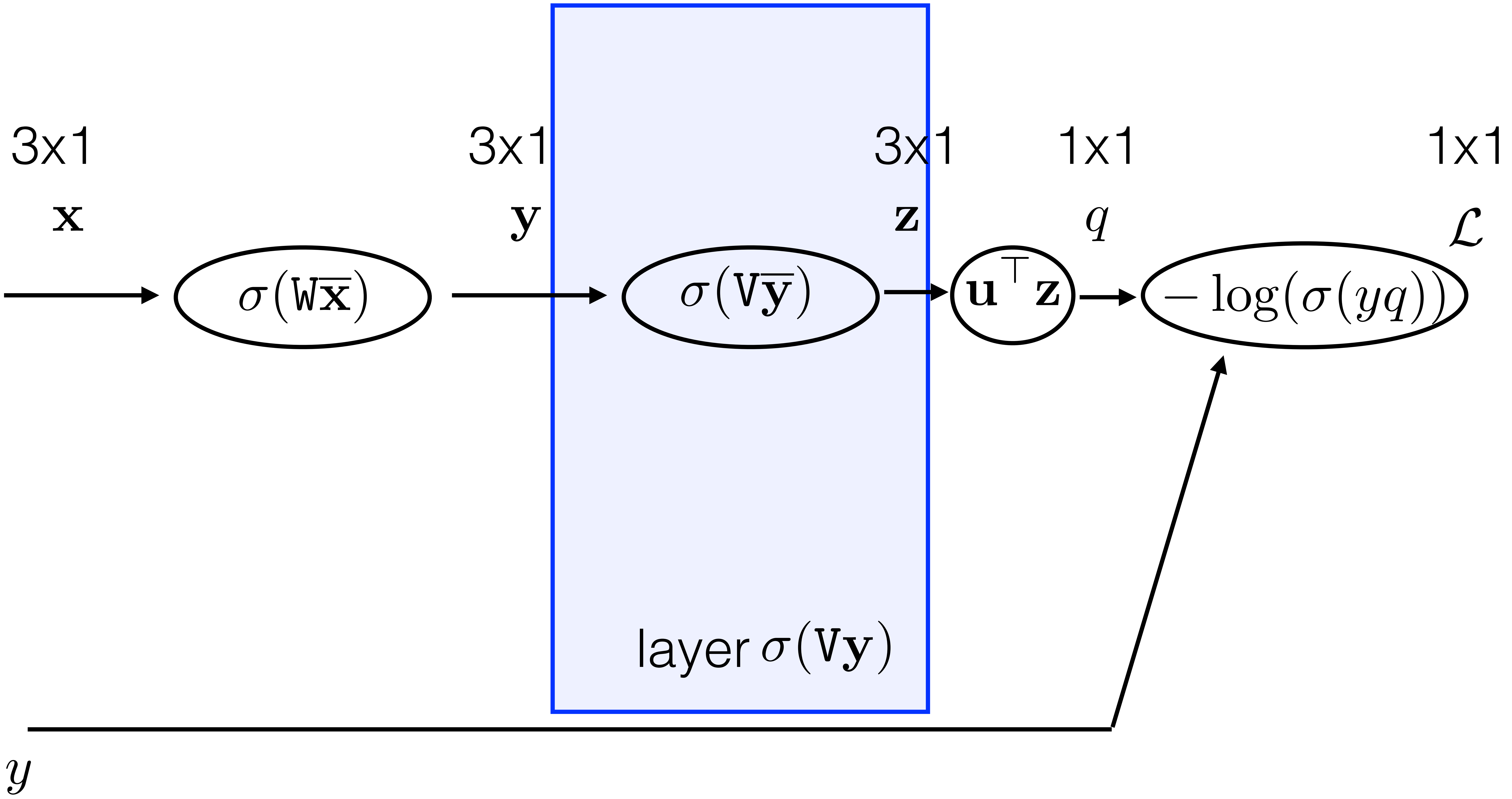
Fully-connected neural network



Fully-connected neural network



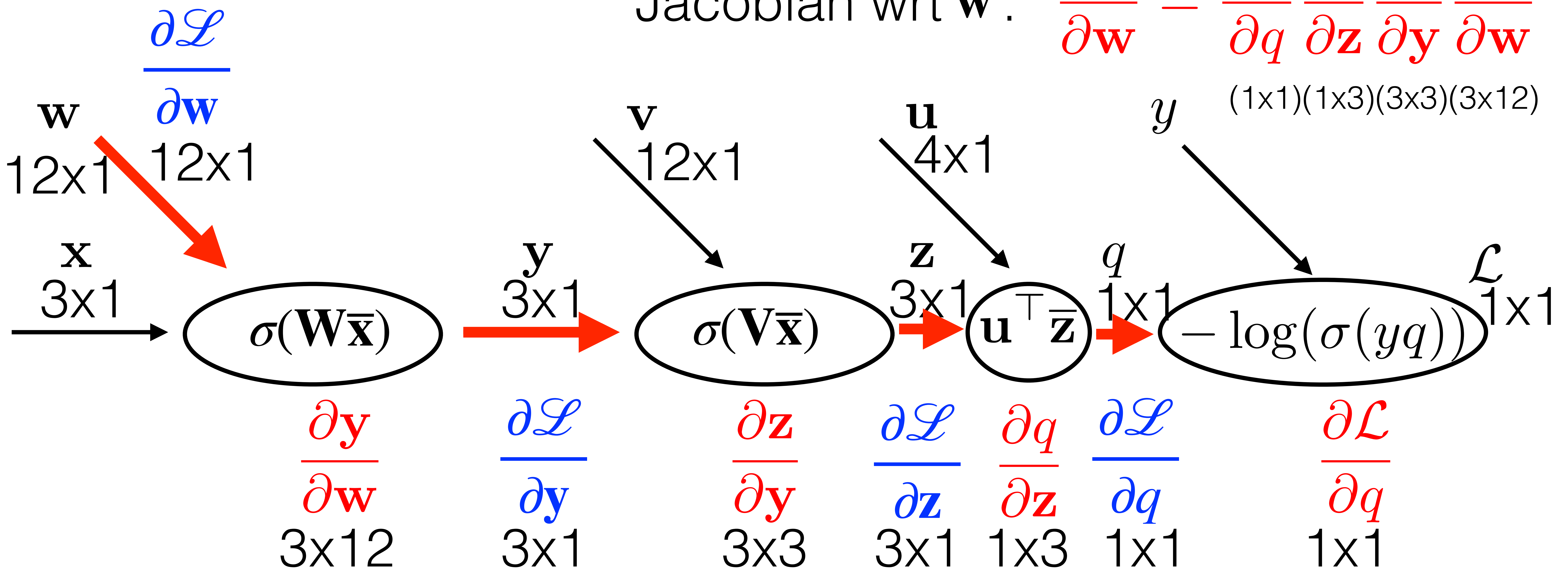
Fully-connected neural network



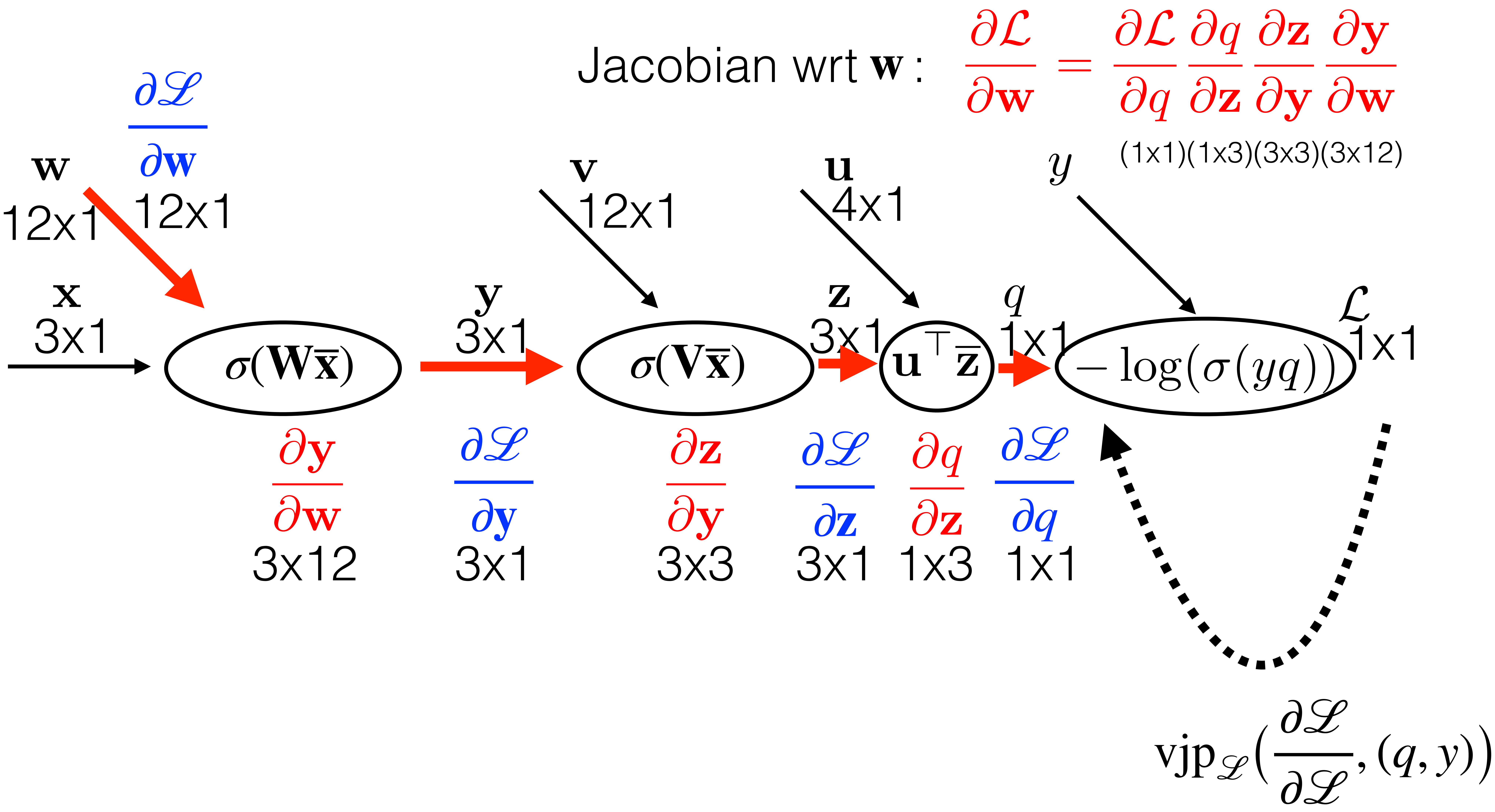
- preserves dimensionality of inputs in backward pass

Jacobian wrt $\mathbf{w}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial q}\frac{\partial q}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{w}}$

(1x1)(1x3)(3x3)(3x12)



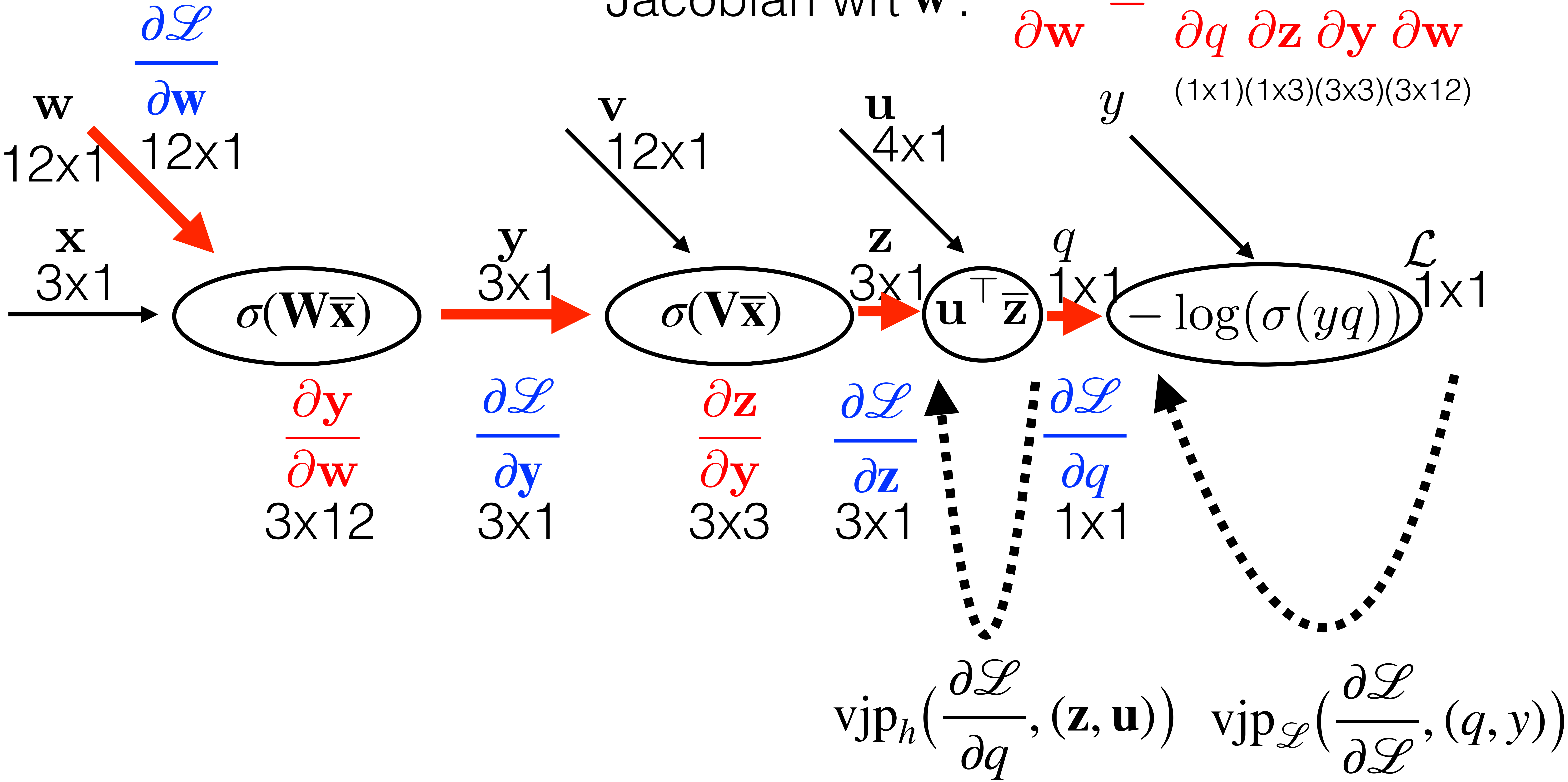
- preserves dimensionality of inputs in backward pass



- preserves dimensionality of inputs in backward pass

Jacobian wrt $\mathbf{w}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial q}\frac{\partial q}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{w}}$

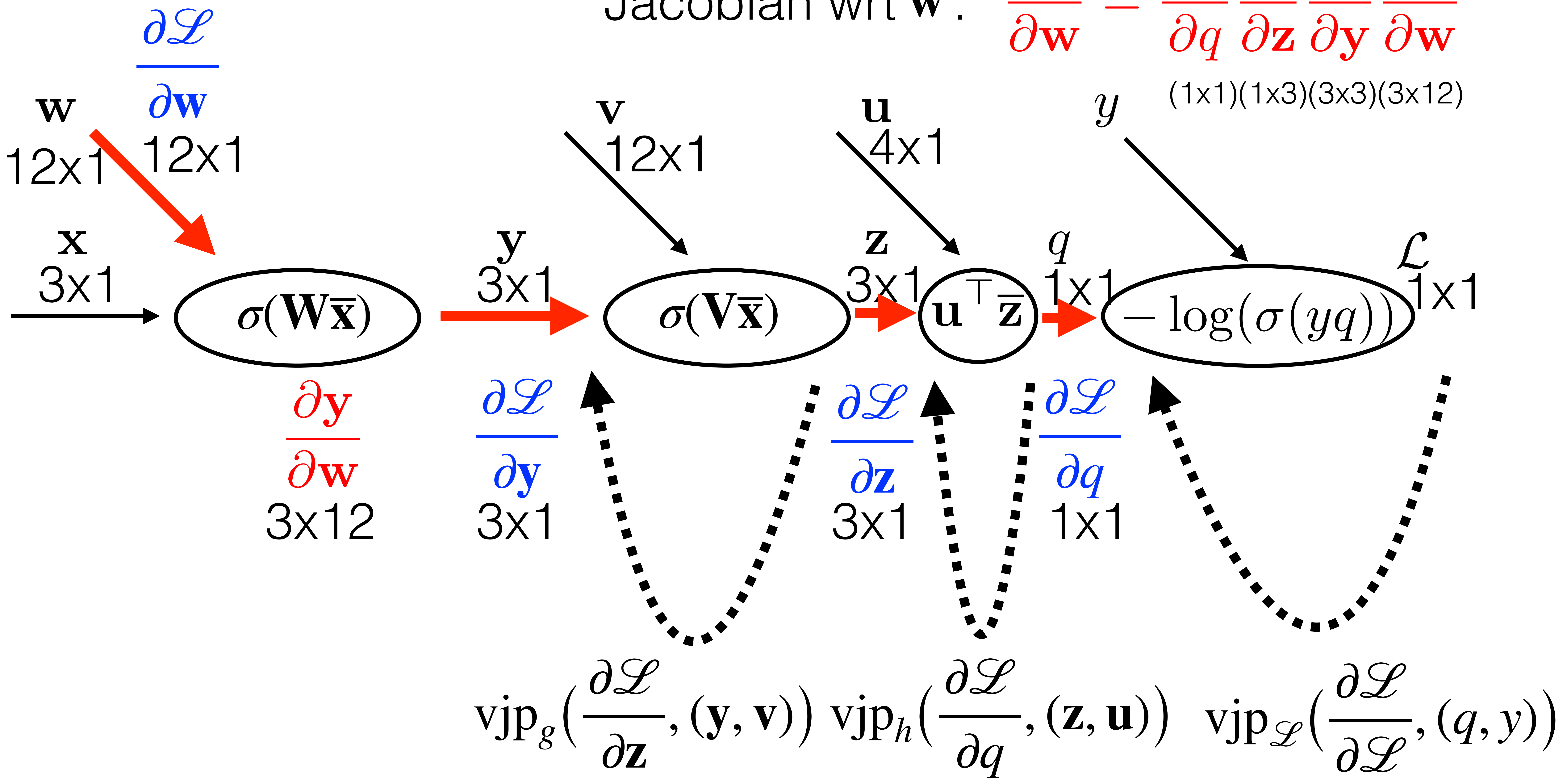
(1x1)(1x3)(3x3)(3x12)



- preserves dimensionality of inputs in backward pass

Jacobian wrt $\mathbf{w}$: $\dfrac{\partial \mathcal{L}}{\partial \mathbf{w}} = \dfrac{\partial \mathcal{L}}{\partial q}\dfrac{\partial q}{\partial \mathbf{z}}\dfrac{\partial \mathbf{z}}{\partial \mathbf{y}}\dfrac{\partial \mathbf{y}}{\partial \mathbf{w}}$

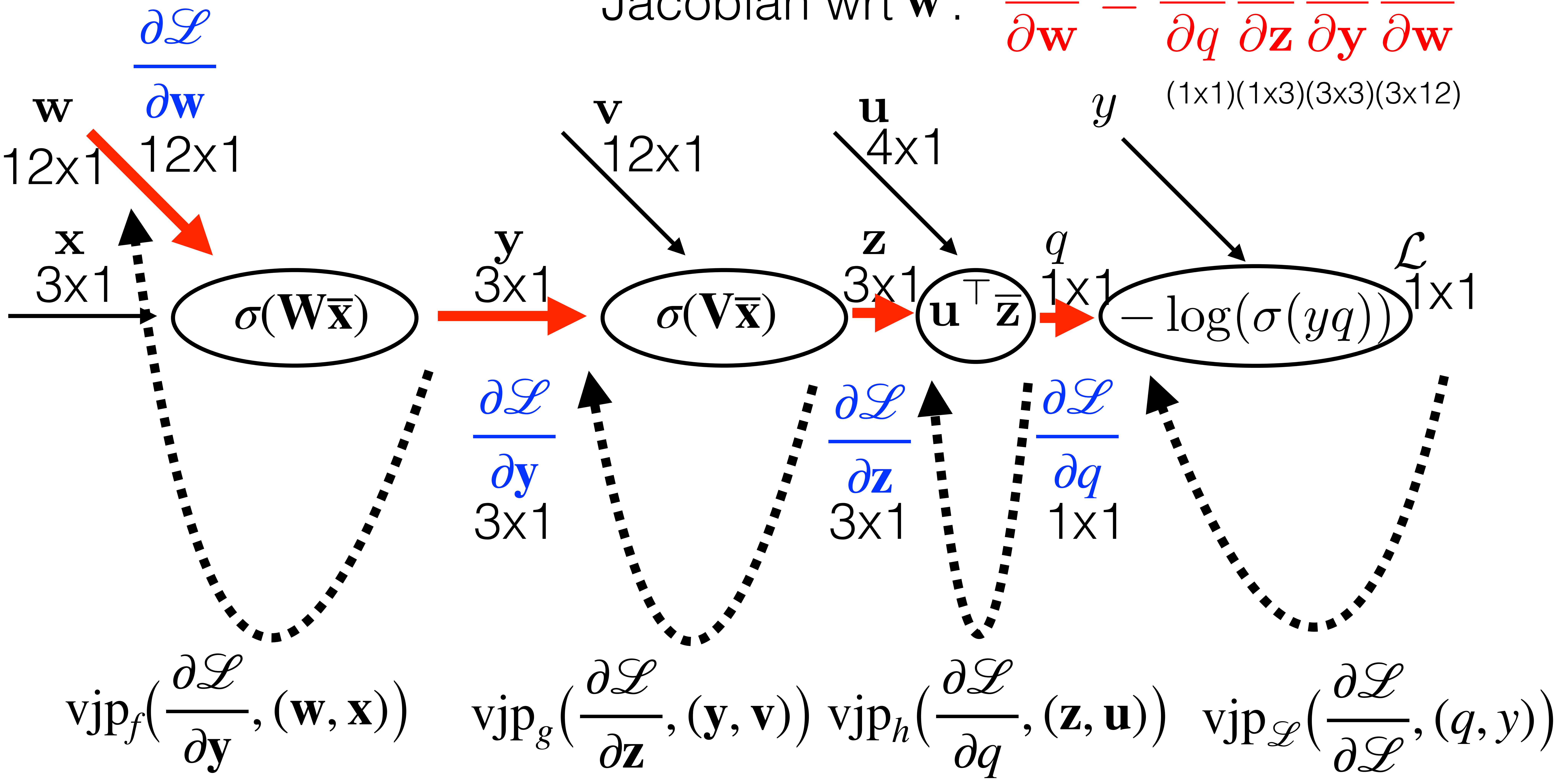
(1x1)(1x3)(3x3)(3x12)



- preserves dimensionality of inputs in backward pass

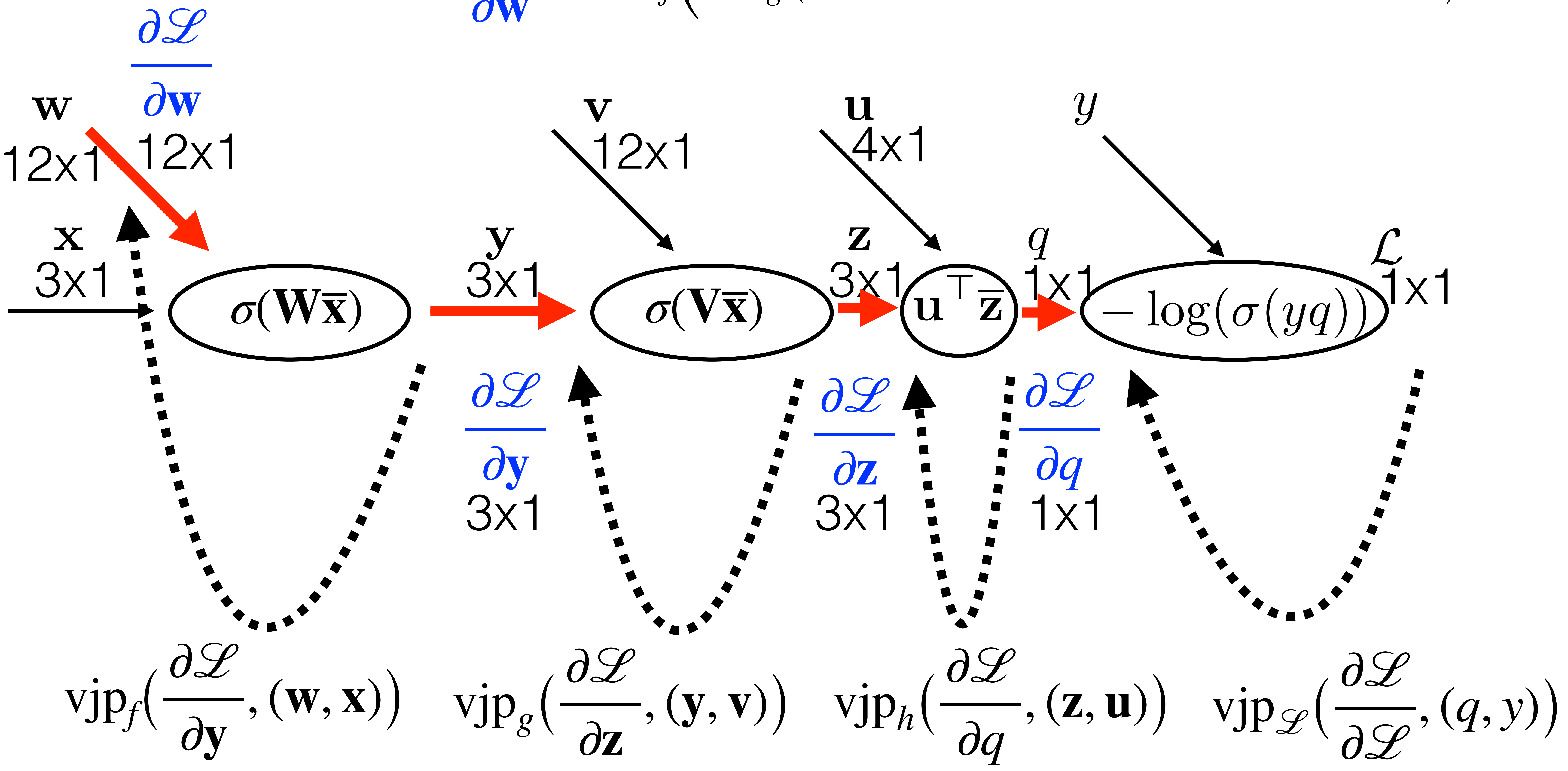
Jacobian wrt $\mathbf{w}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial q}\frac{\partial q}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{w}}$

(1x1)(1x3)(3x3)(3x12)



- preserves dimensionality of inputs in backward pass

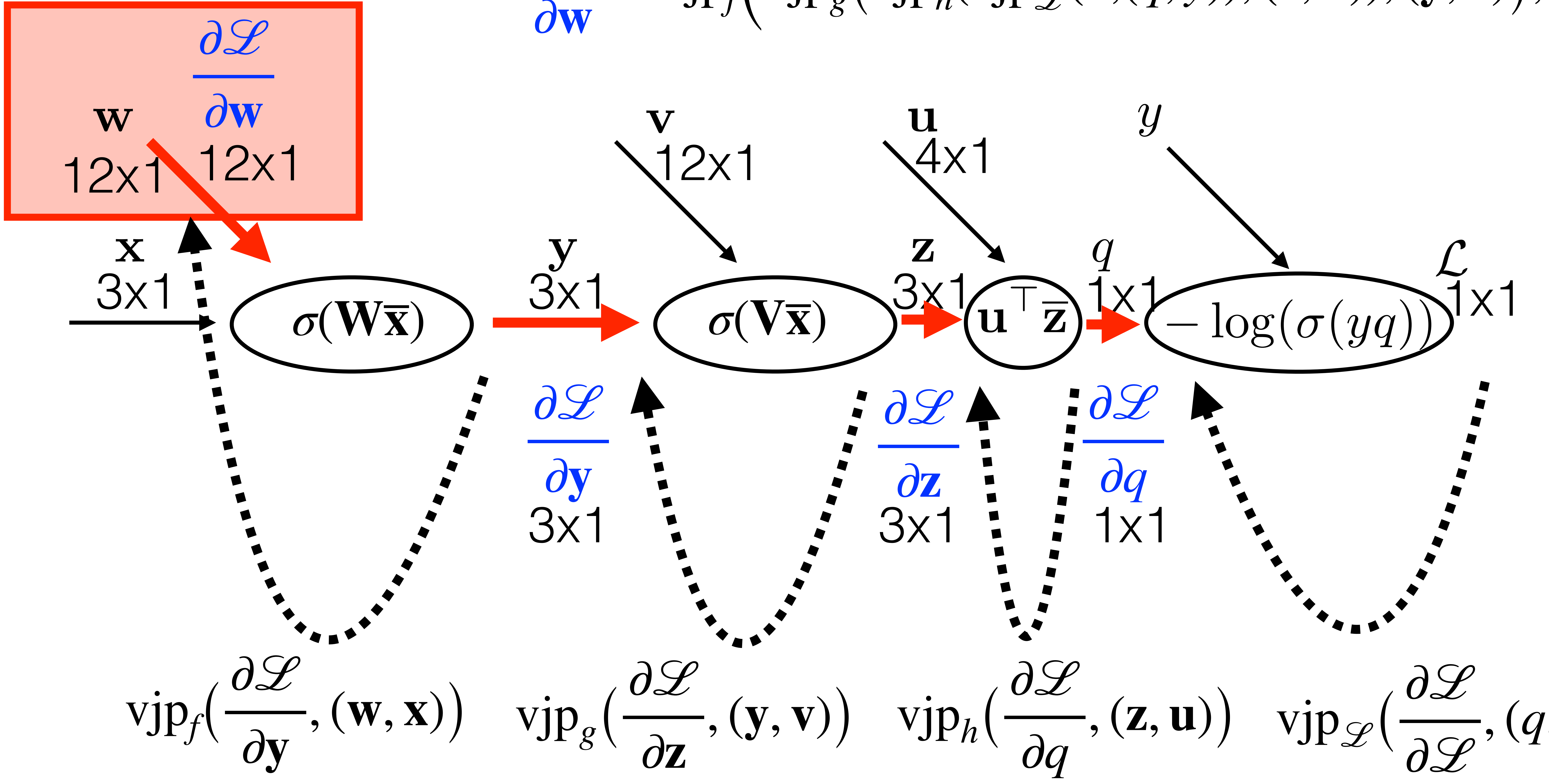
Jacobian wrt $\mathbf{w}$: $\frac{\partial \mathscr{L}}{\partial \mathbf{w}} = \mathrm{vjp}_f\Big(\mathrm{vjp}_g\big(\mathrm{vjp}_h(\mathrm{vjp}_{\mathscr{L}}(1,(q,y)),(\mathbf{z},\mathbf{u})),(\mathbf{y},\mathbf{v})\big),(\mathbf{w},\mathbf{x})\Big)$



There are no more jacobians

- preserves dimensionality of inputs in backward pass

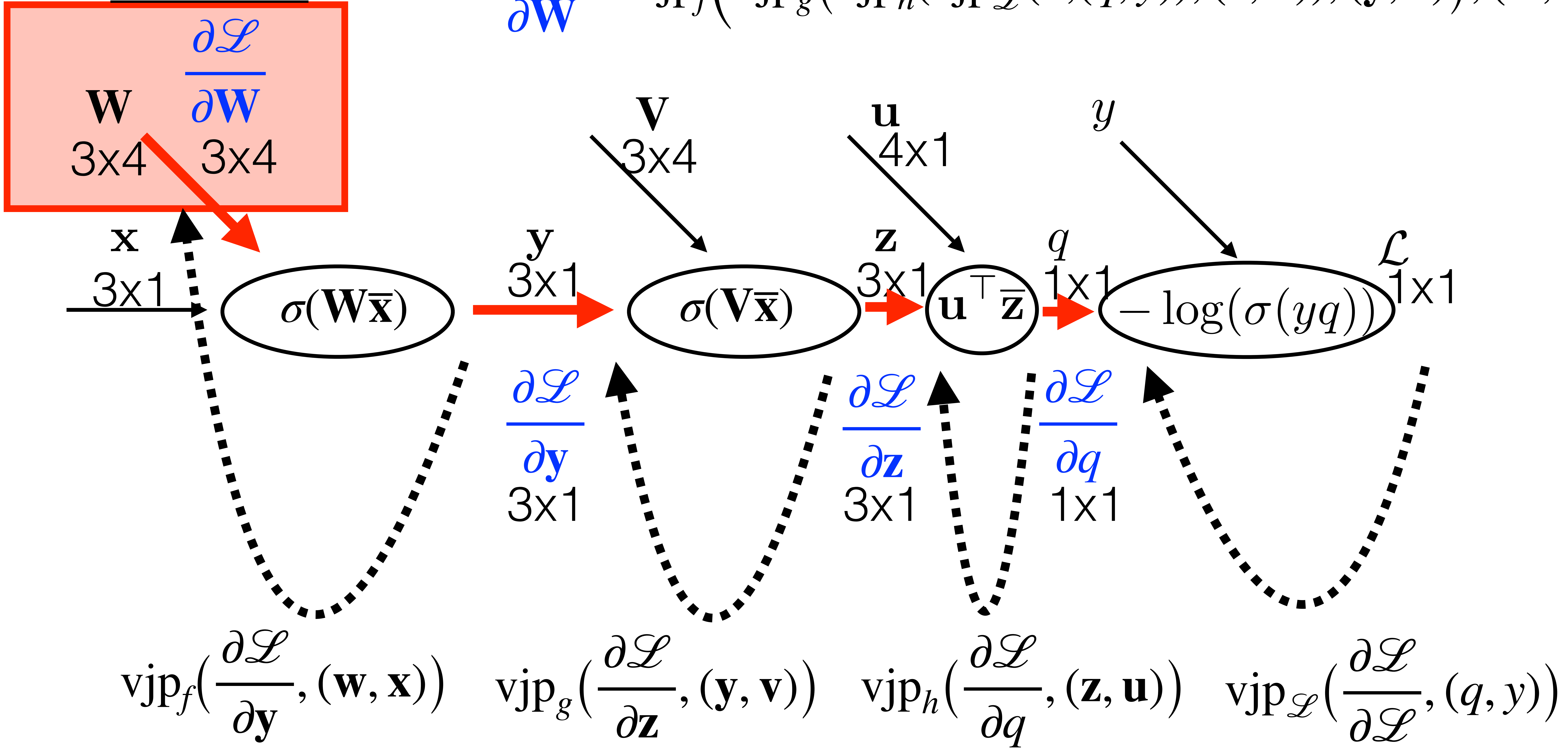
Jacobian wrt $\mathbf{w}$: $\frac{\partial \mathscr{L}}{\partial \mathbf{w}} = \mathrm{vjp}_f\Big(\mathrm{vjp}_g\big(\mathrm{vjp}_h(\mathrm{vjp}_{\mathscr{L}}(1,(q,y)),(\mathbf{z},\mathbf{u})),(\mathbf{y},\mathbf{v})\big),(\mathbf{w},\mathbf{x})\Big)$



We can (re-)implement vjp in order to preserve inputs resolution

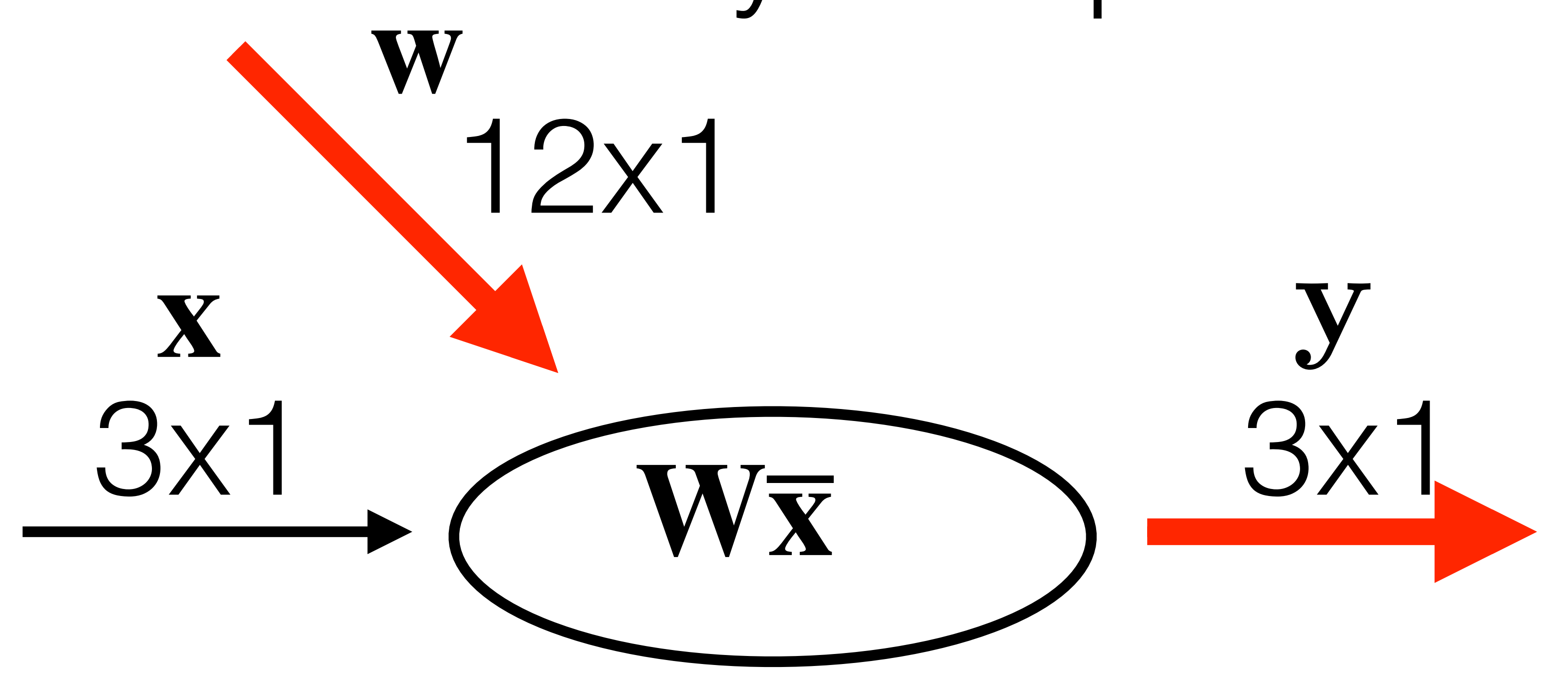
- preserves dimensionality of inputs in backward pass

Gradient wrt $\mathbf{W}$: $\frac{\partial \mathscr{L}}{\partial \mathbf{W}} = \mathrm{vjp}_f\Big(\mathrm{vjp}_g\big(\mathrm{vjp}_h(\mathrm{vjp}_{\mathscr{L}}(1,(q,y)),(\mathbf{z},\mathbf{u})),(\mathbf{y},\mathbf{v})\big),(\mathbf{w},\mathbf{x})\Big)$



Resulting matrix is reshaped jacobian/gradient of $\mathscr{L}$ wrt $\mathbf{W}$

- preserves dimensionality of inputs in backward pass



def $\mathrm{vjp}_{\mathbf{w}}\big(\mathbf{v}, (\mathbf{x}, \mathbf{w})\big)$:

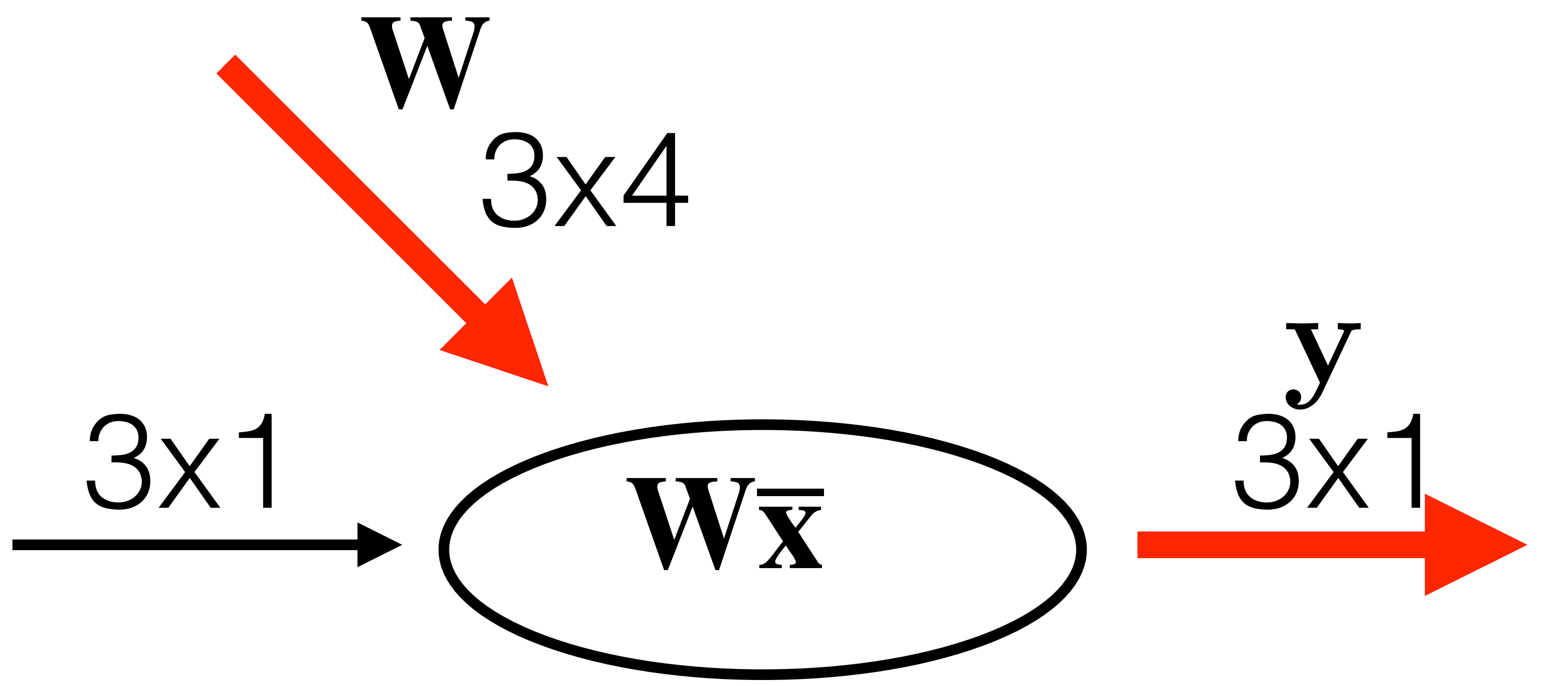
return $\mathbf{v} \cdot \dfrac{\partial \mathbf{y}}{\partial \mathbf{w}} = \mathbf{v} \cdot \begin{bmatrix} -\bar{\mathbf{x}}^\top - & & \\ & -\bar{\mathbf{x}}^\top - & \\ & & -\bar{\mathbf{x}}^\top - \end{bmatrix} = \begin{bmatrix} v_1 \cdot \bar{\mathbf{x}}^\top, & v_2 \cdot \bar{\mathbf{x}}^\top, & v_3 \cdot \bar{\mathbf{x}}^\top \end{bmatrix}$

1x3 3x12 1x12

def $\mathrm{vjp}_{\mathbf{W}}\big(\mathbf{v}, (\mathbf{x}, \mathbf{W})\big)$:

return $\begin{bmatrix} -v_1 \cdot \bar{\mathbf{x}}^\top - \\ -v_2 \cdot \bar{\mathbf{x}}^\top - \\ -v_3 \cdot \bar{\mathbf{x}}^\top - \end{bmatrix}$

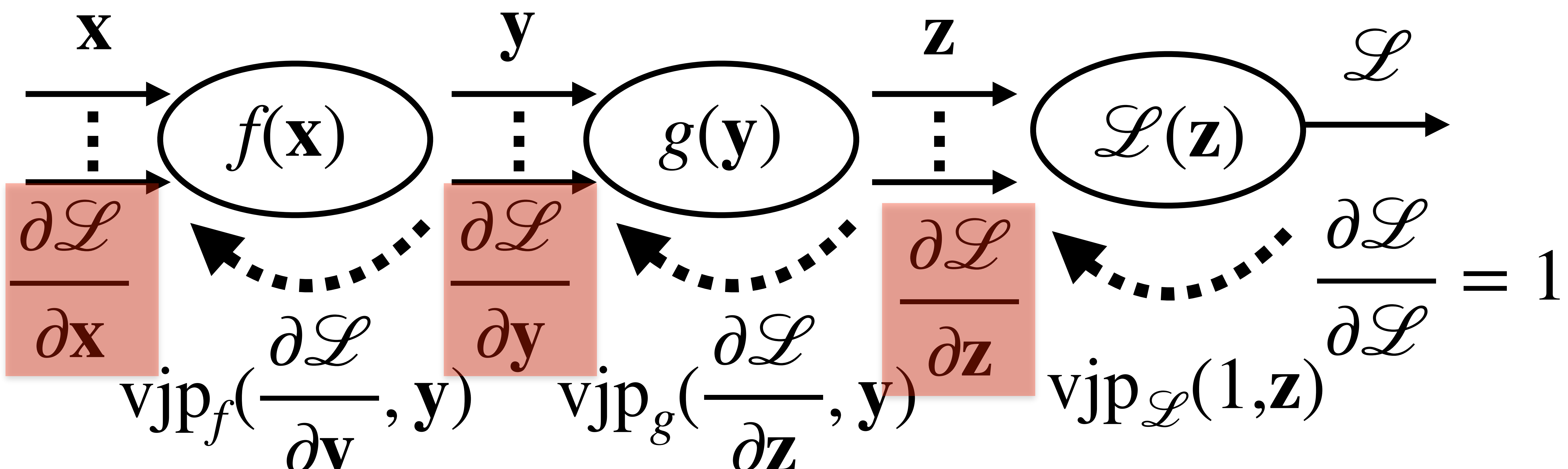
3x4



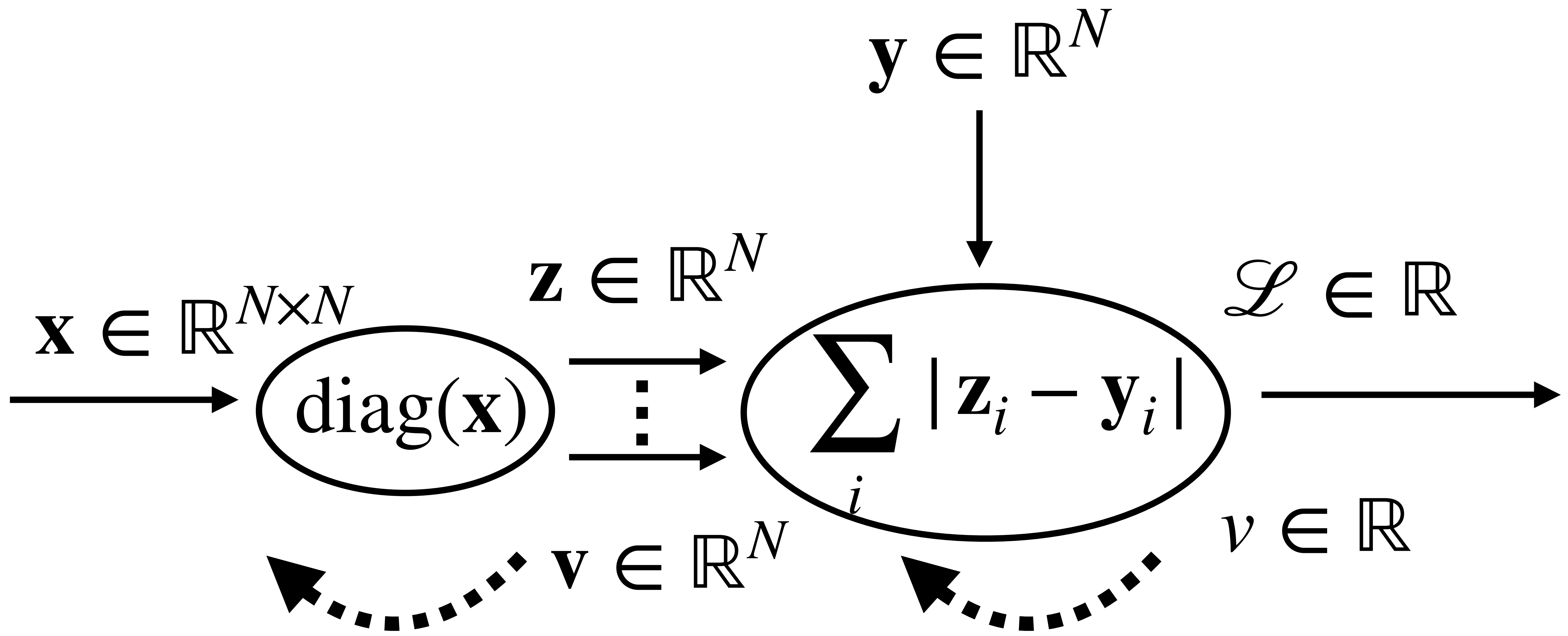
- preserves dimensionality of inputs in backward pass
- if $g : \mathbb{R}^{m \times m} \rightarrow \mathbb{R}^{n \times n}$, what is its jacobian? $\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} : \mathbb{R}^{m \times m} \rightarrow \mathbb{R}^{n \times n \times m \times m}$
- vjp **avoids** multiplications of **high-dimensional jacobian** tensors
- vjp **avoids explicit vectorization** of higher-dimensional data structures (images)

```
u = torch.tensor([[1,2], [3, 4]], dtype=torch.float32, requires_grad=True)
loss = torch.sigmoid(u).sum()
grad = torch.autograd.grad(loss, u)[0]
print(grad)
tensor([[0.1966, 0.1050],
        [0.0452, 0.0177]])
```

- edges of comp. graph are populated by gradients of loss wrt edge-variables



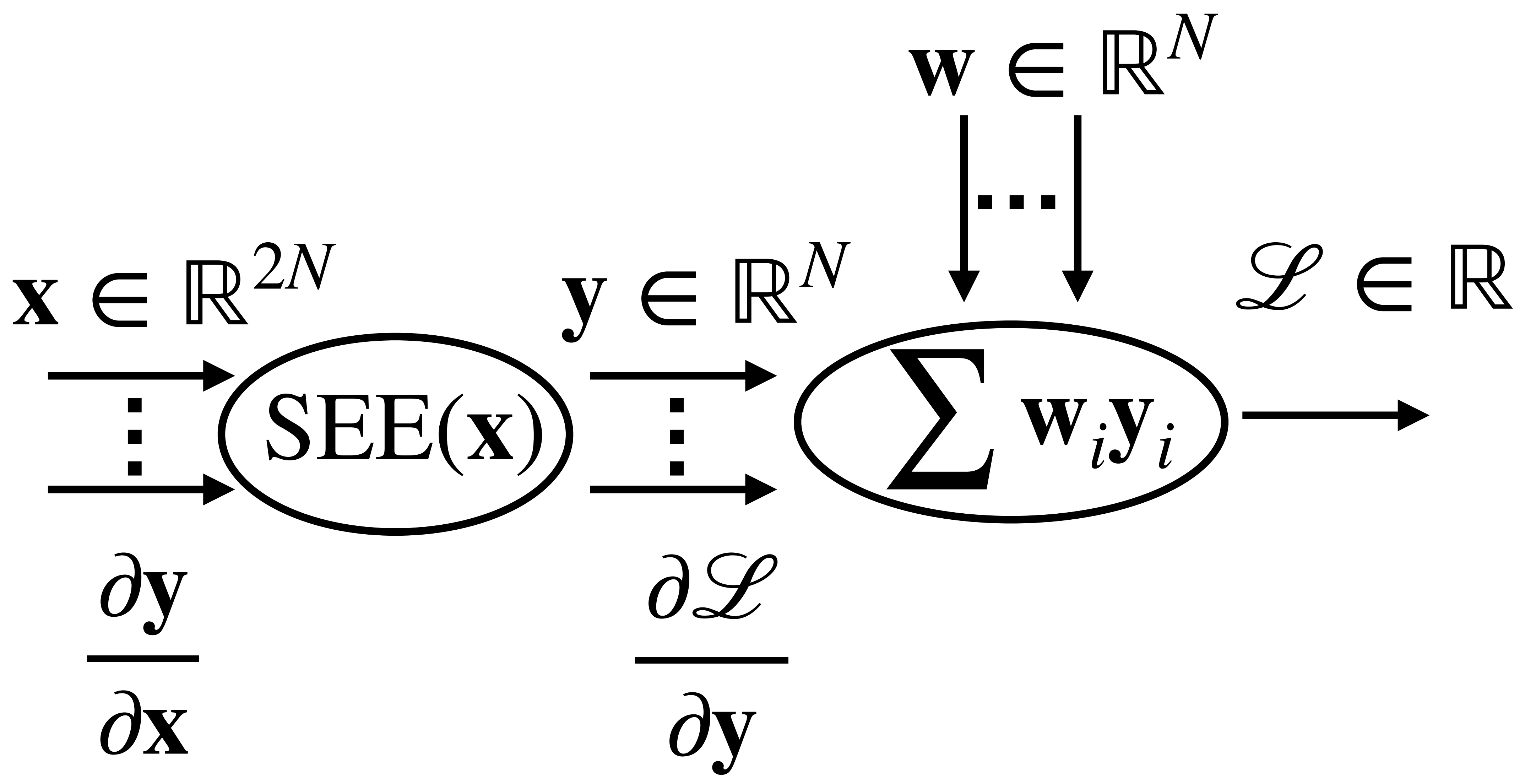
Example: Jacobian vs vector-jacobian-product function



$$\text{vjp}_{\text{diag}}\left(\mathbf{x}, \mathbf{v}\right) = ? \qquad \text{vjp}_{\mathscr{L}}\left(\mathbf{z}, \mathbf{y}, v\right) = ?$$

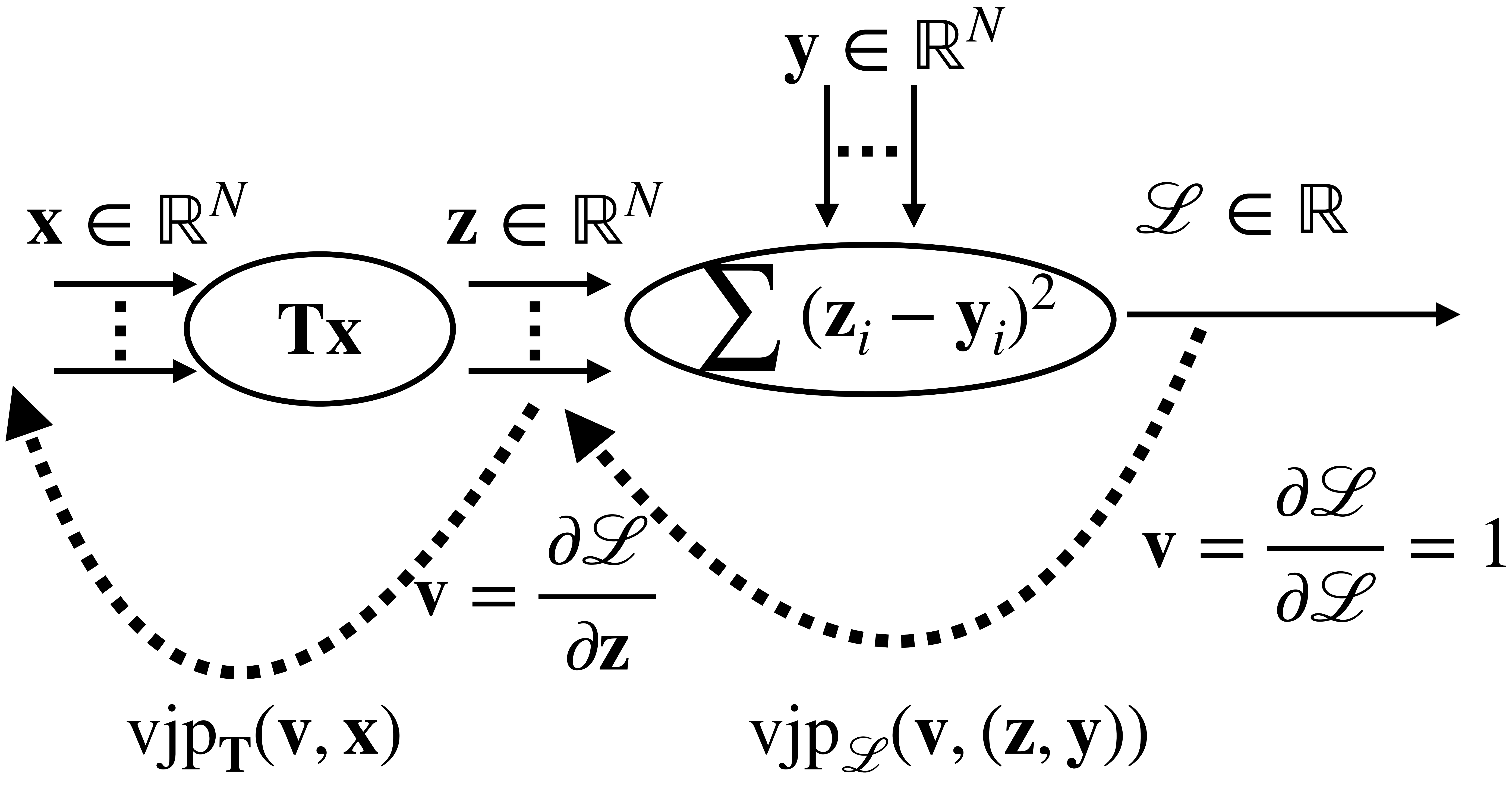
Example: Jacobian vs vector-jacobian-product function

$\mathbf{y} = \mathrm{SEE}(\mathbf{x})$ = Select Even Element from input vector $\mathbf{x}$



$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = ???$$

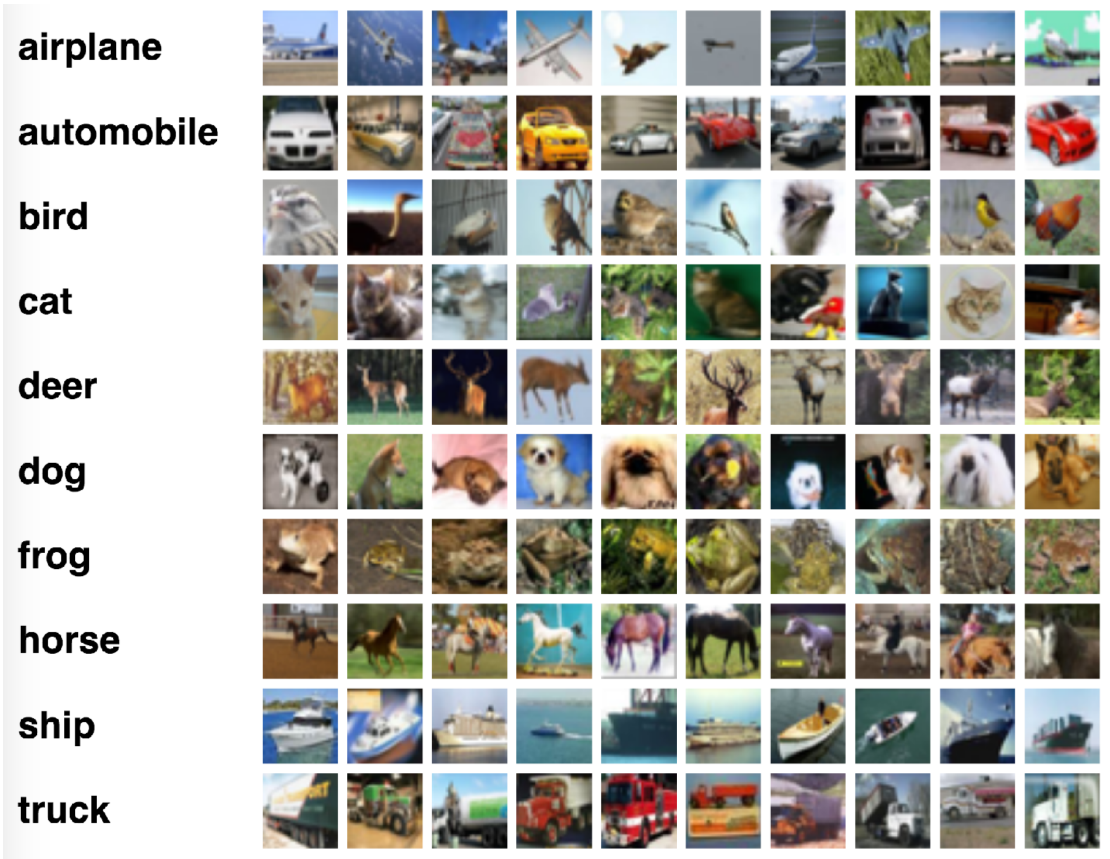
Example: Jacobian vs vector-jacobian-product function



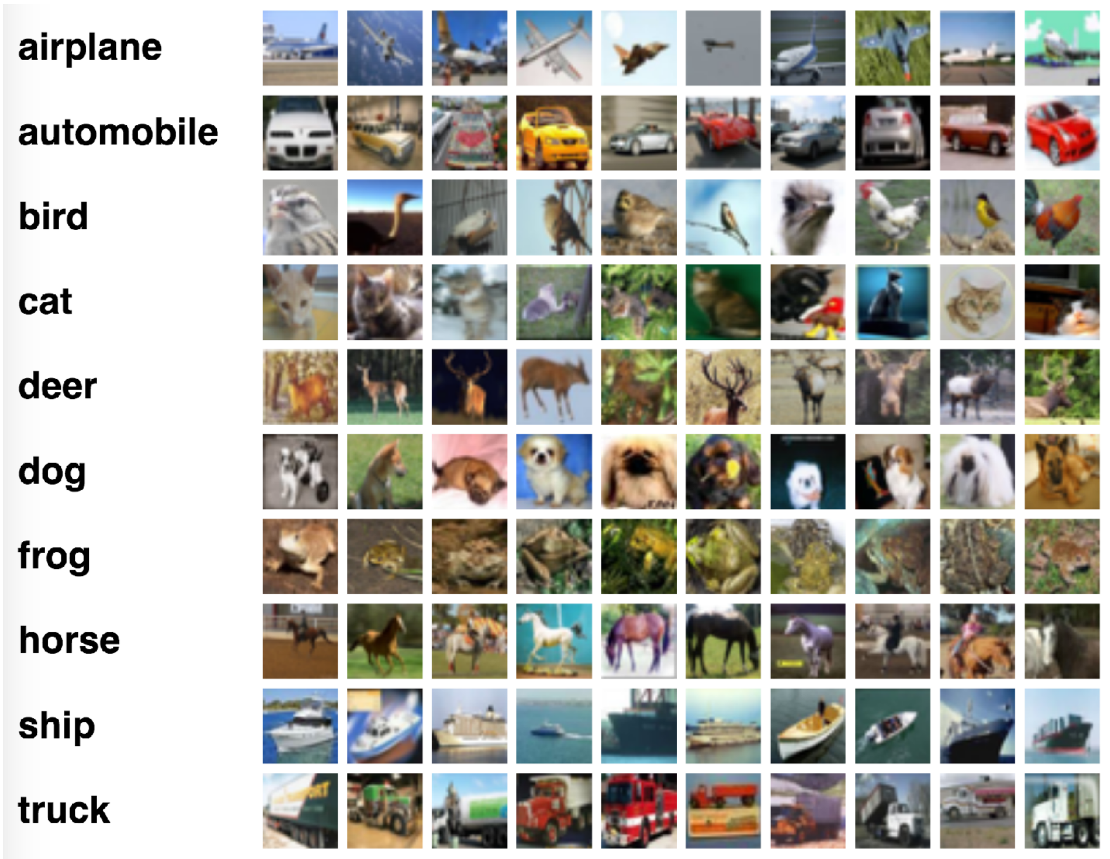
$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = ???$$

Neural nets summary

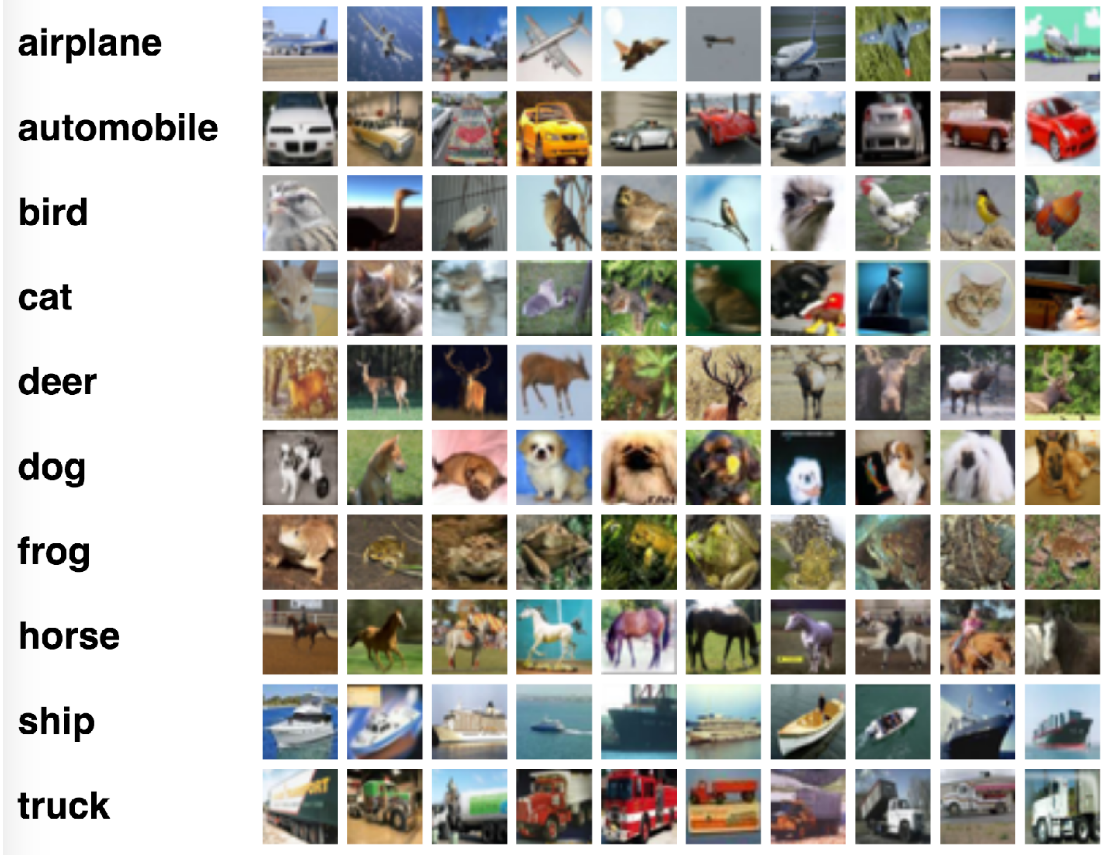
- **Neural net** is a function created as concatenation of simplier functions (e.g. neurons or layers of neurons)
- **Fully connected neural net** is neural network created from neurons, where **all** outputs from previous layer are connected to **all** inputs of the following layer
- **Learning** is gradient optimization of a neural net concatenated with a loss-layer on a training set.
- **Gradient** evaluation is implemented as backward pass through vector-jacobian functions (neither symbolic differentiation nor numeric differentiation)
- **VJP** allows to evaluate gradient of **scalar output** in one backward pass, however the full jacobian of M-dim output requires M passes of vjp! => inefficient LM
- **Reverse differentiation** (VJP): computes Jacobian of the loss one **row** at a time
- **Forward differentiation** (JVP): computes Jacobian one **column** at a time
- **Deep learning frameworks** (Pytorch, Tensorflow, Caffe) has many predefined layers and takes care about the efficient implementation of autodiff on GPU.
- **Deep learning = GPU + data + autodiff**
- **Spoiler alert:** Fully connected NN does not work on structural data (images, sound) well.

Dataset		Error	
		Linear	FCNN
MNIST		8%	**??**
CIFAR-10		63%	**??**

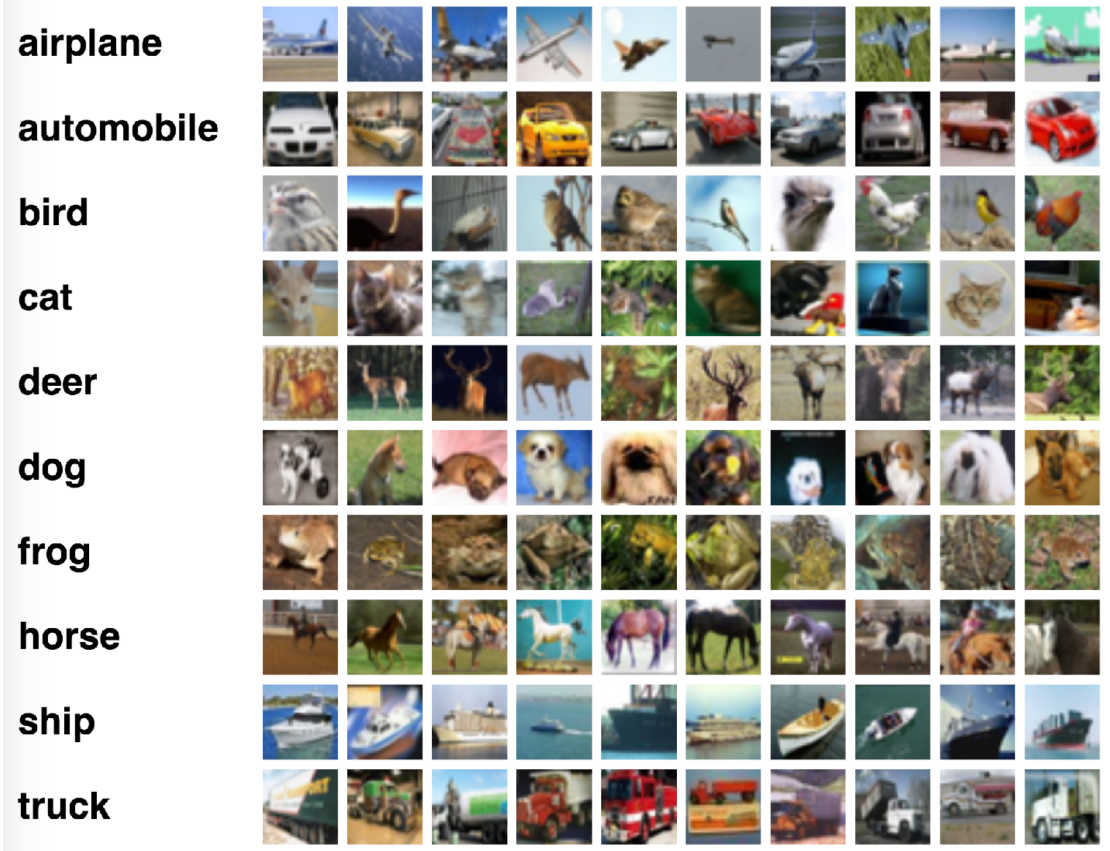
https://benchmarks.ai

Dataset		Error	
		Linear	FCNN
MNIST		8%	2%
CIFAR-10		63%	55%

https://benchmarks.ai

Dataset		Error Linear	Error FCNN	Error ConvNet
MNIST		8%	2%	**??**
CIFAR-10		63%	55%	**??**

https://benchmarks.ai

Dataset		Error		
		Linear	FCNN	ConvNet
MNIST		8%	2%	0.2% [CVPR 2013]
CIFAR-10		63%	55%	1% [EfficientNet, 2018]

https://benchmarks.ai

Dataset

Error

Linear | FCNN | ConvNet

MNIST

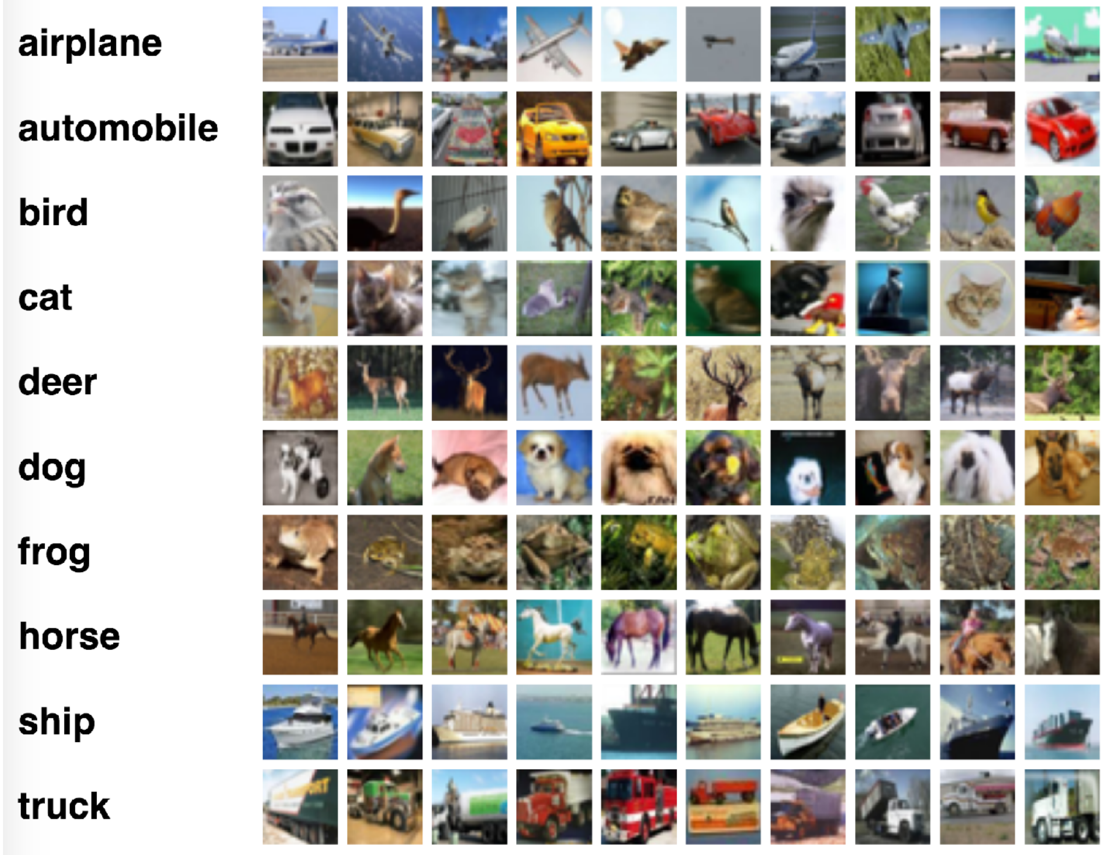
8% | 2% | 0.2% [CVPR 2013]

underfit: too simple arch.

overfit: too complex arch.

good fit ;-) cortex inspired architecture

CIFAR-10



63% | 55% | 1% [EfficientNet, 2018]

https://benchmarks.ai

Competencies required for the test T1

- Ability to draw a computational graph.
- Compute vector-jacobian-product of a given mapping
- Compute backpropagation in computational graph