Under the hood of autodiff

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

Linear classifier and neuron

Labels

RGB images

$+1$

-1

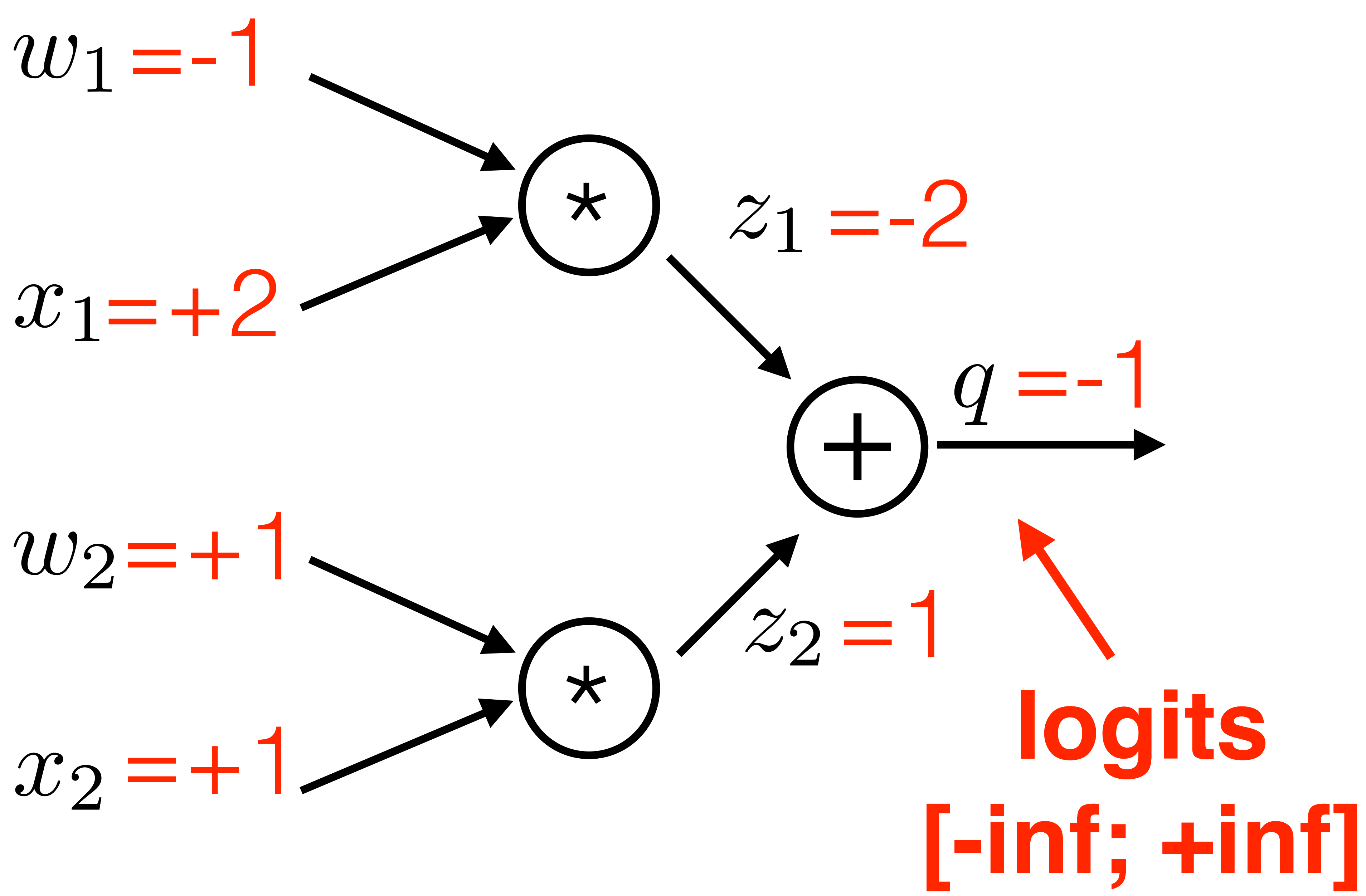
def classify():

 # Linear classifier

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

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

Computational graph of linear classifier



Linear classifier and neuron

Labels | RGB images

$+1$

-1

def classify():

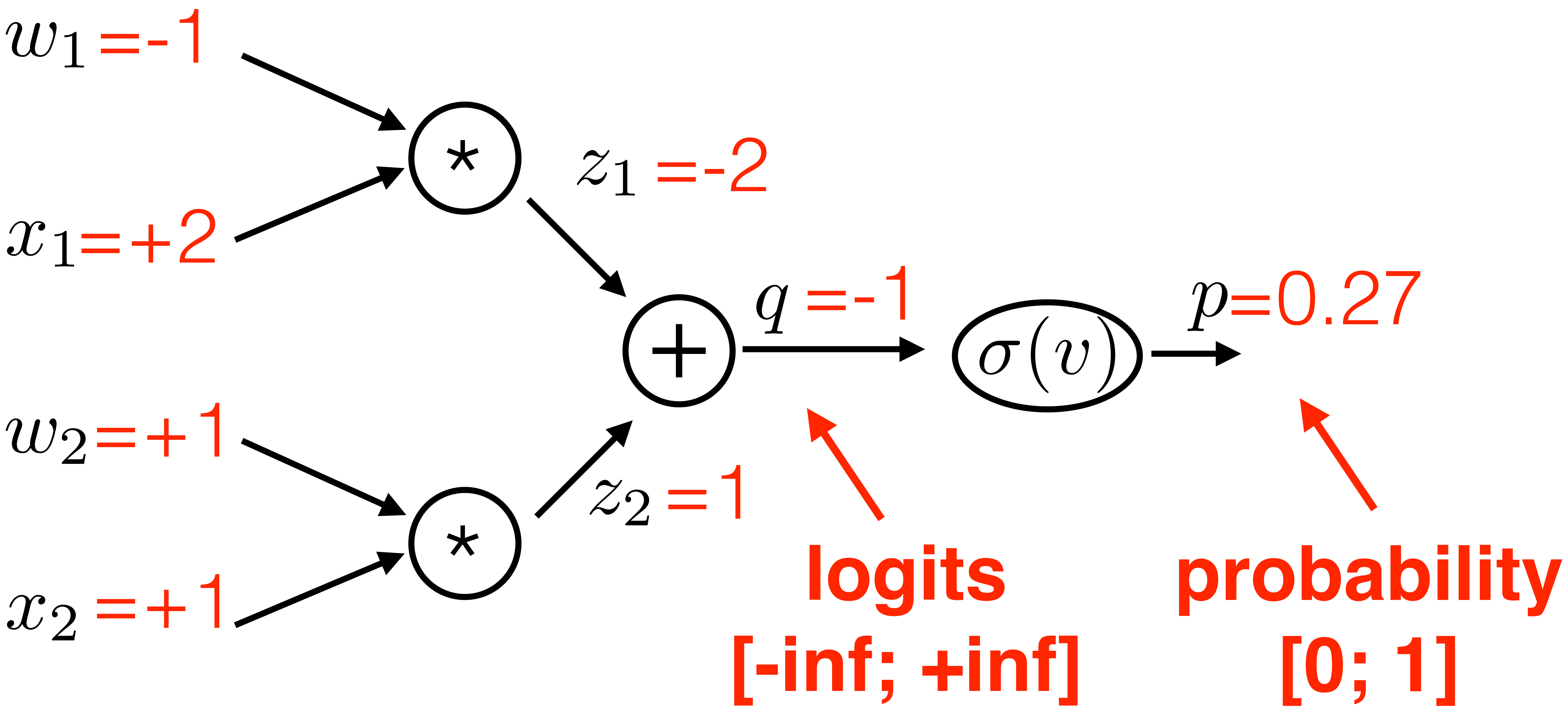
 # Neuron

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

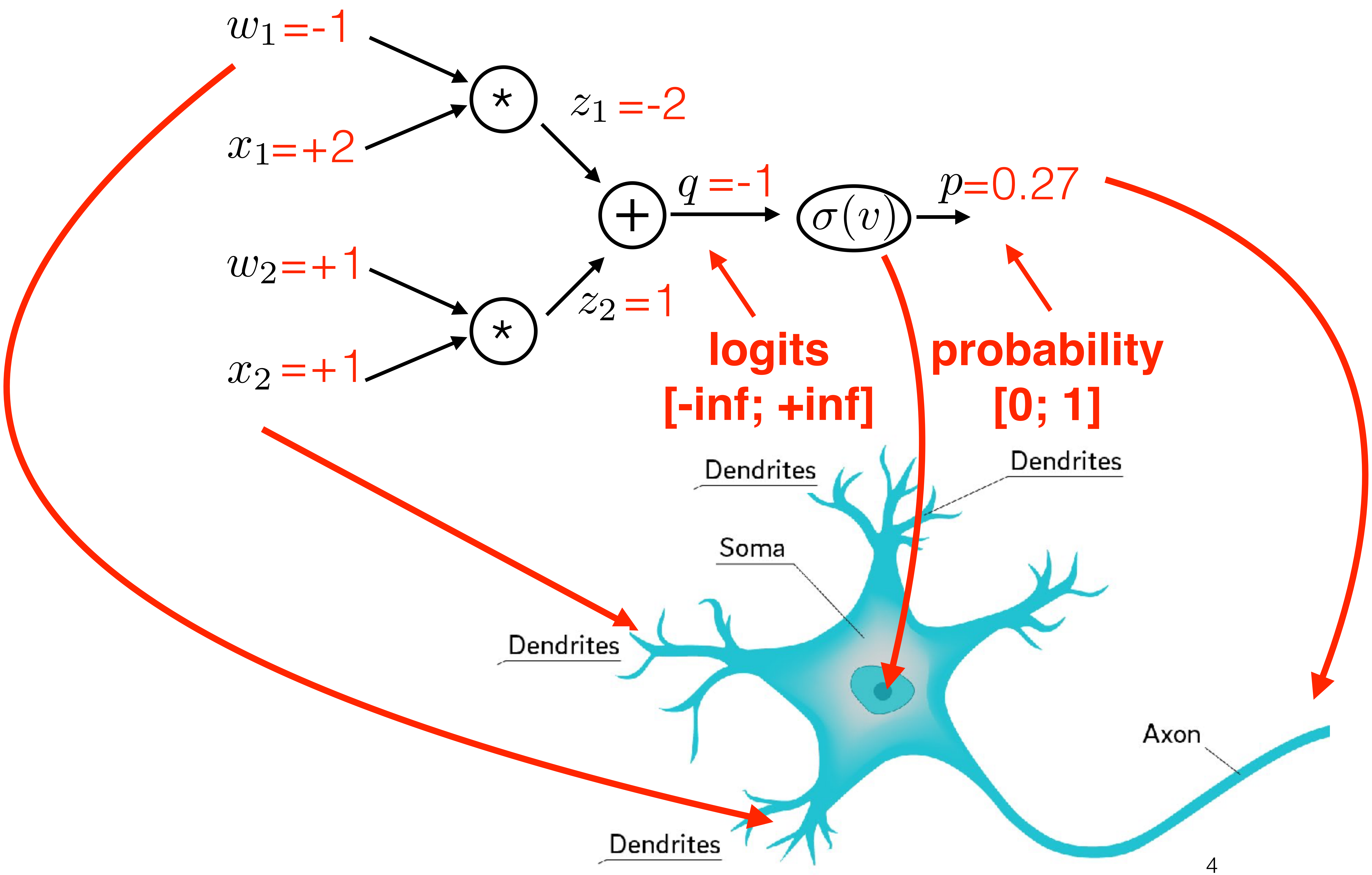
$p = \sigma\left(\mathbf{w}^\top \mathbf{x}\right)$

return p

Computational graph of neuron

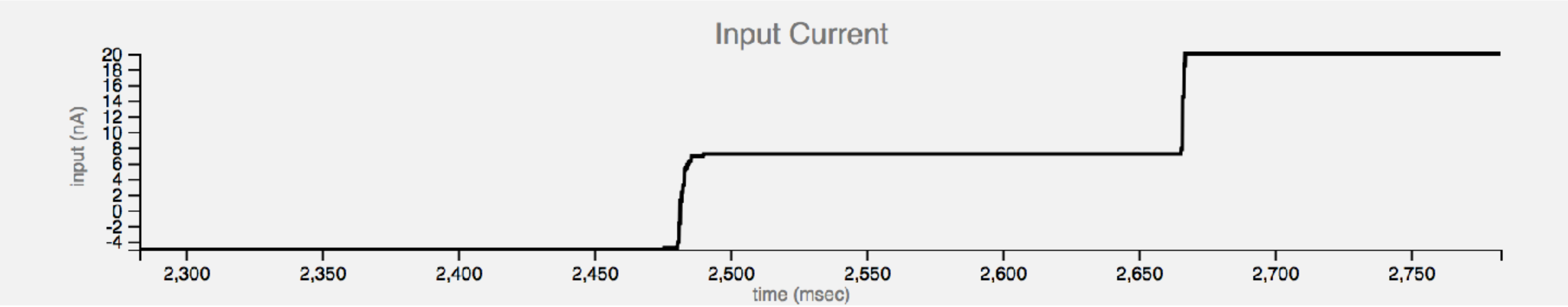


Relation to biological neuron

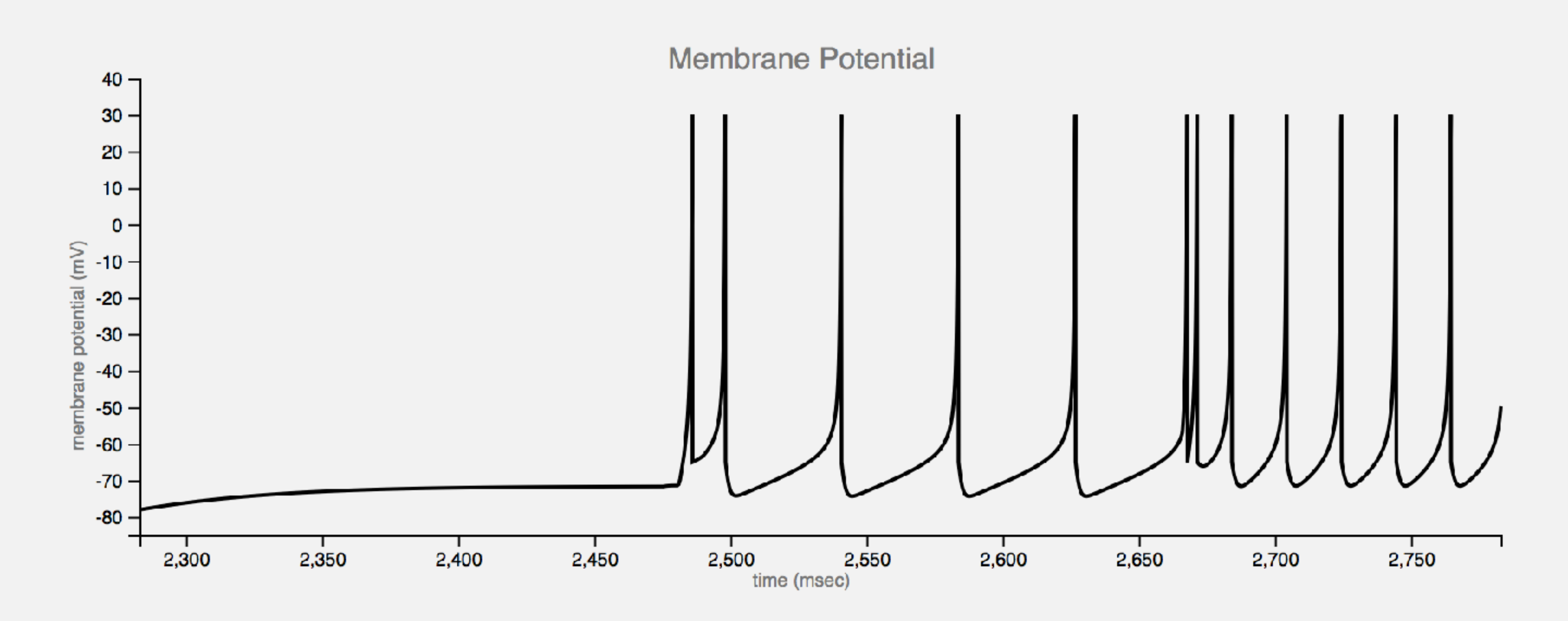


Modeling dynamic neuron behaviour

Input:

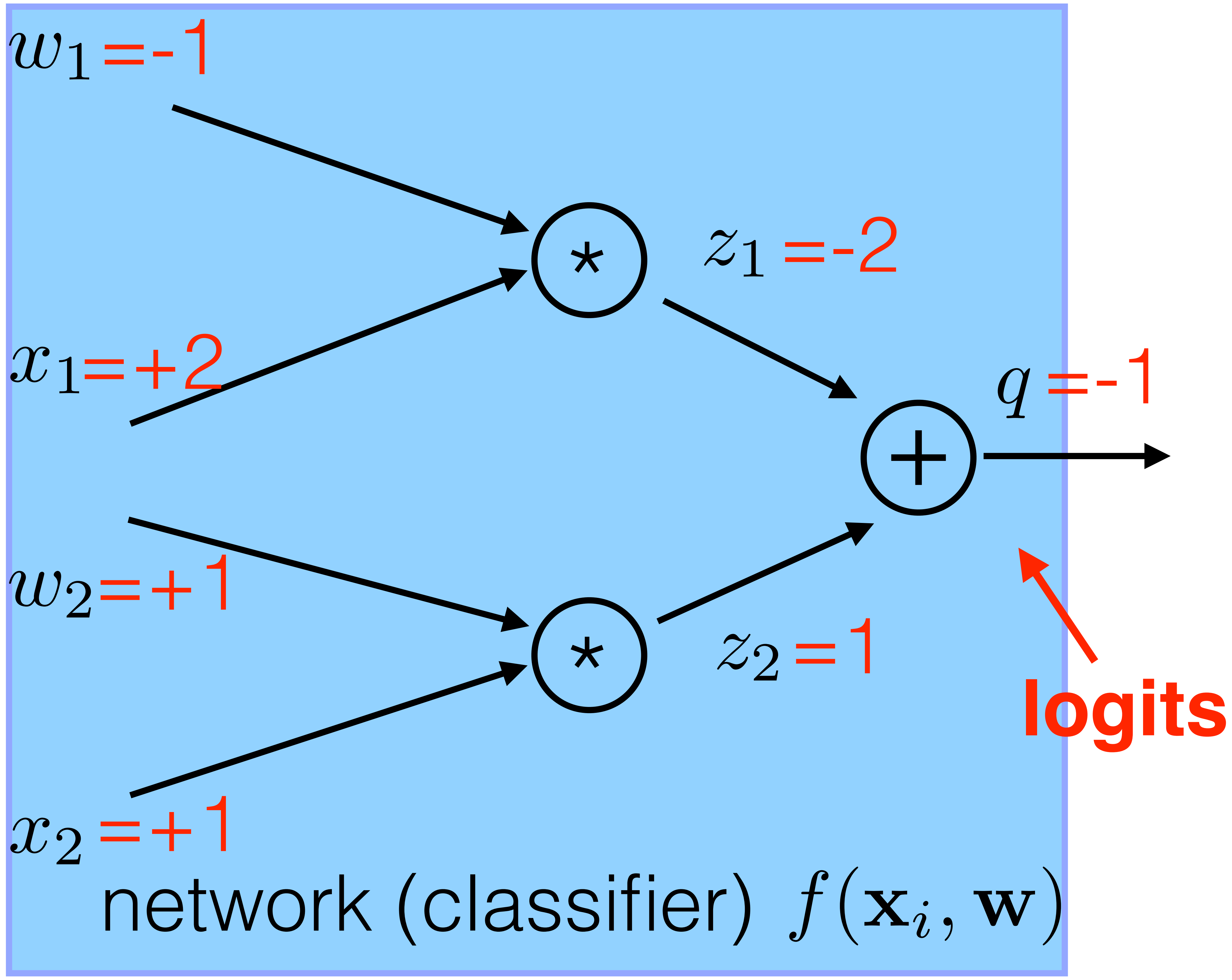


Output:

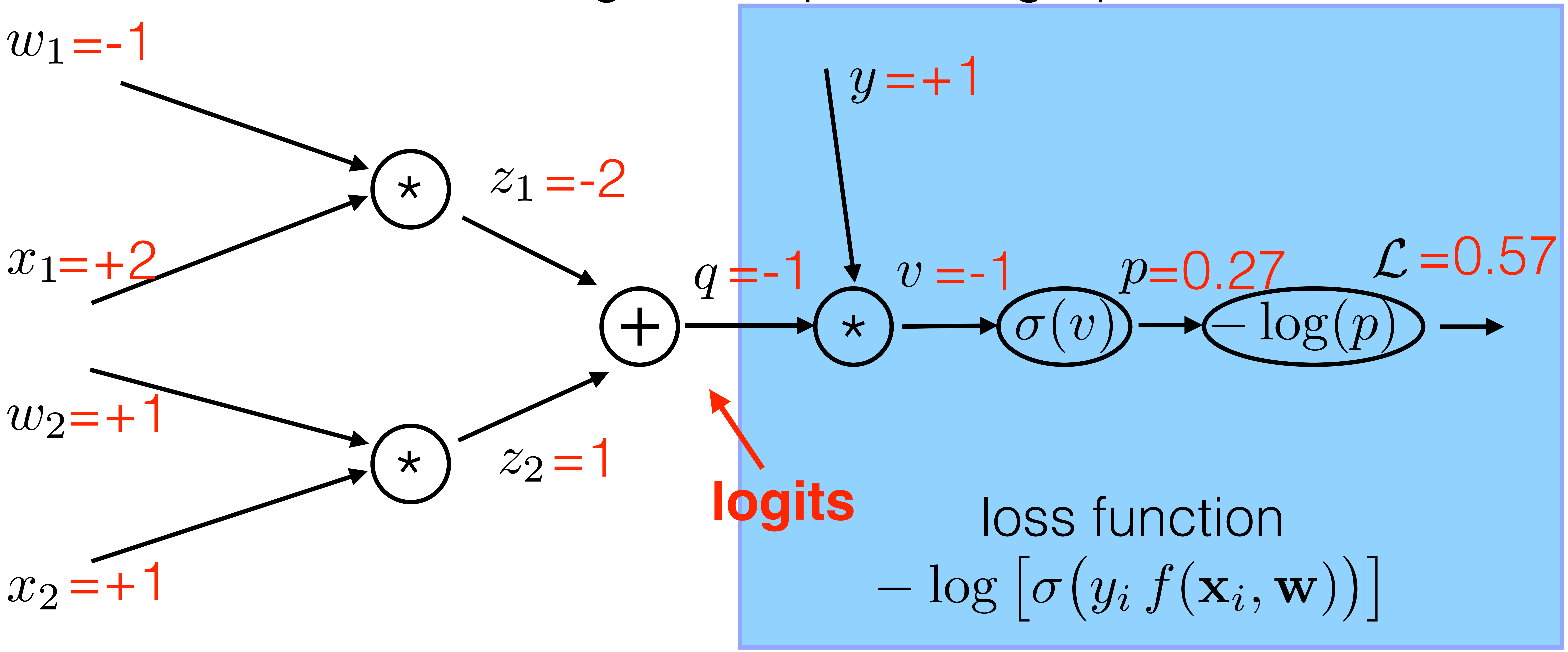


http://jackterwilliger.com/biological-neural-networks-part-i-spiking-neurons/

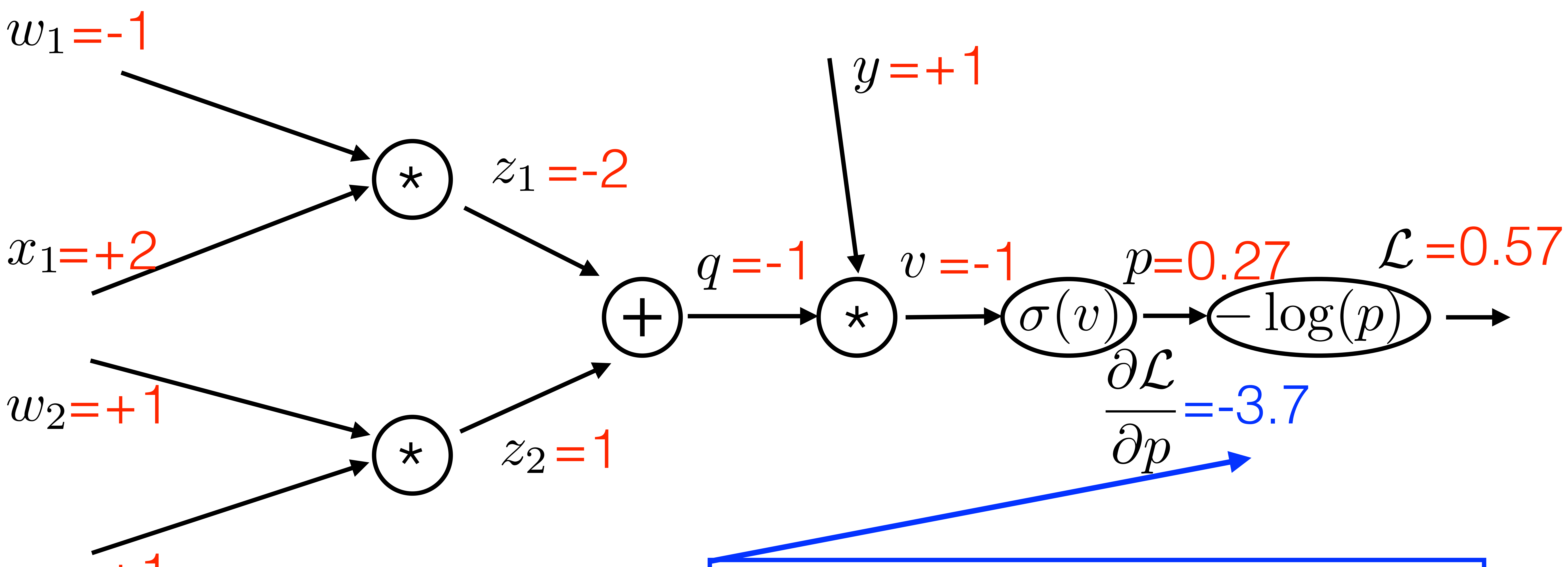
Learning in computation graph



Learning in computation graph



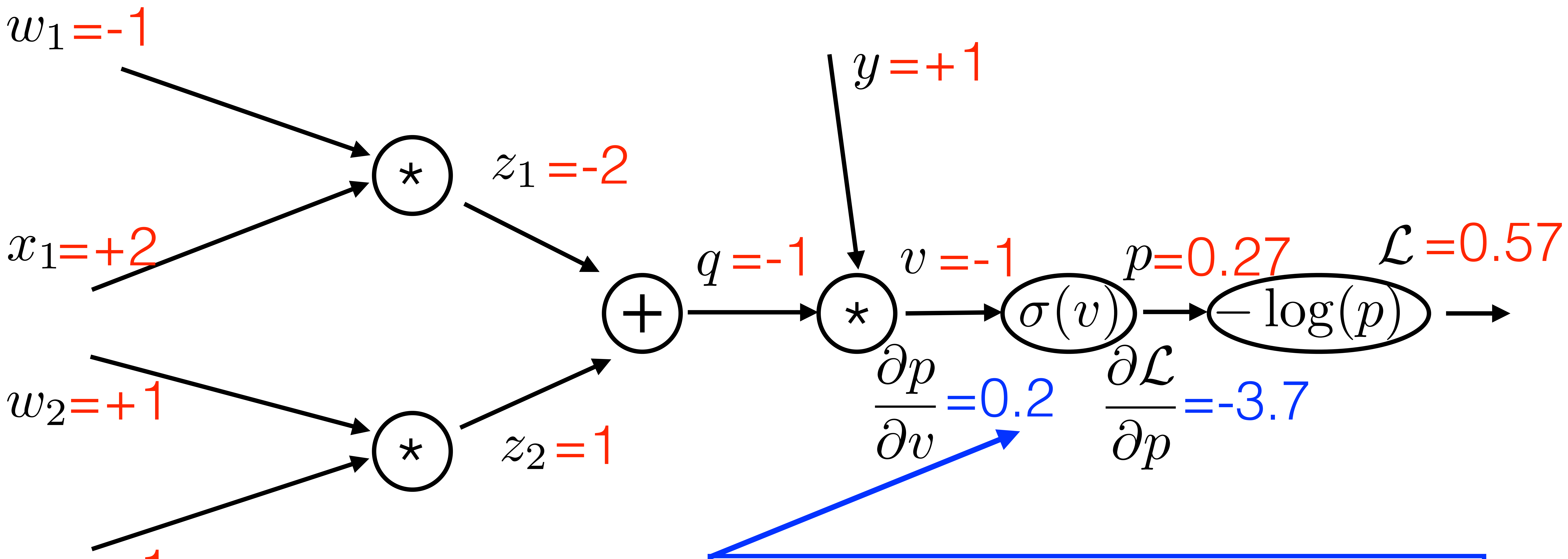
Learning in computation graph



Local gradient:

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

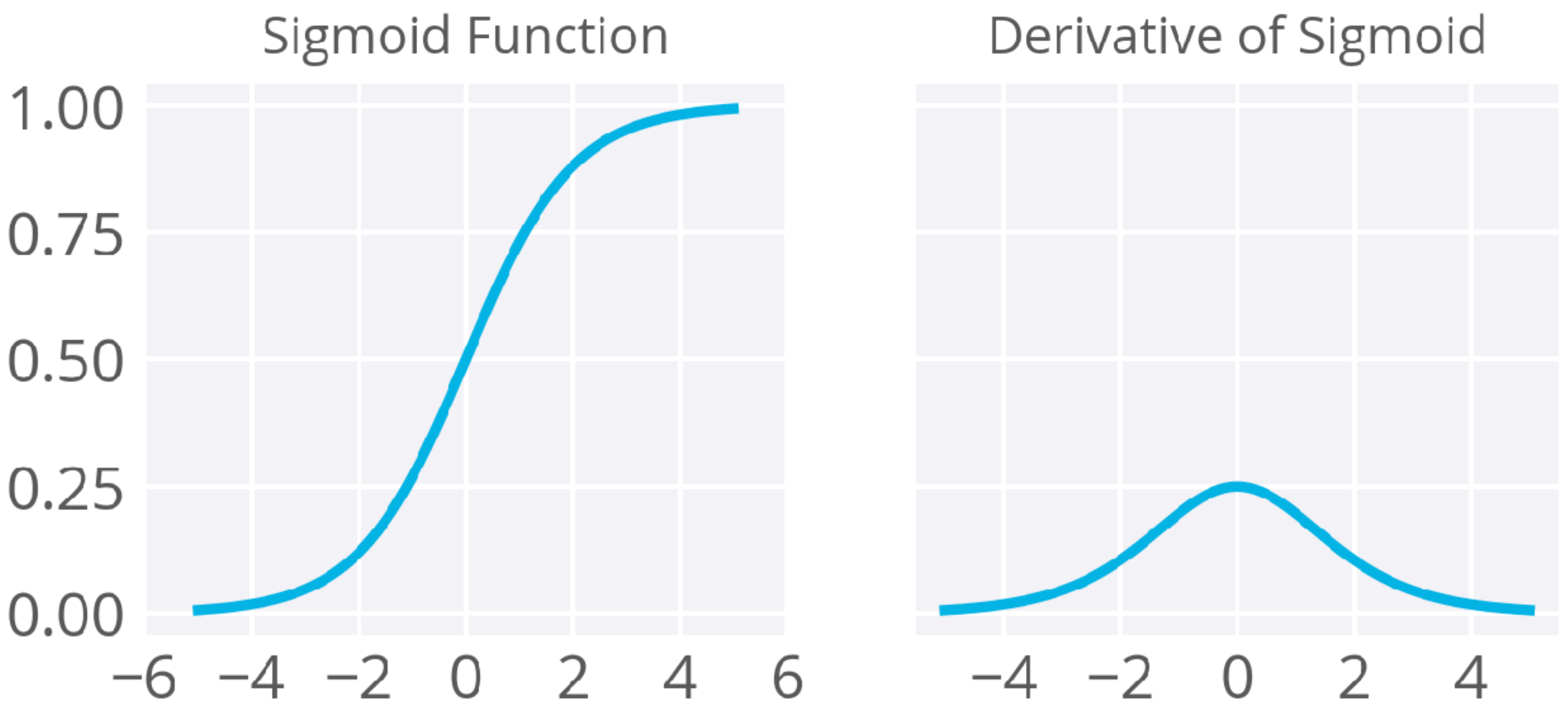
Learning in computation graph



Local gradient:

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

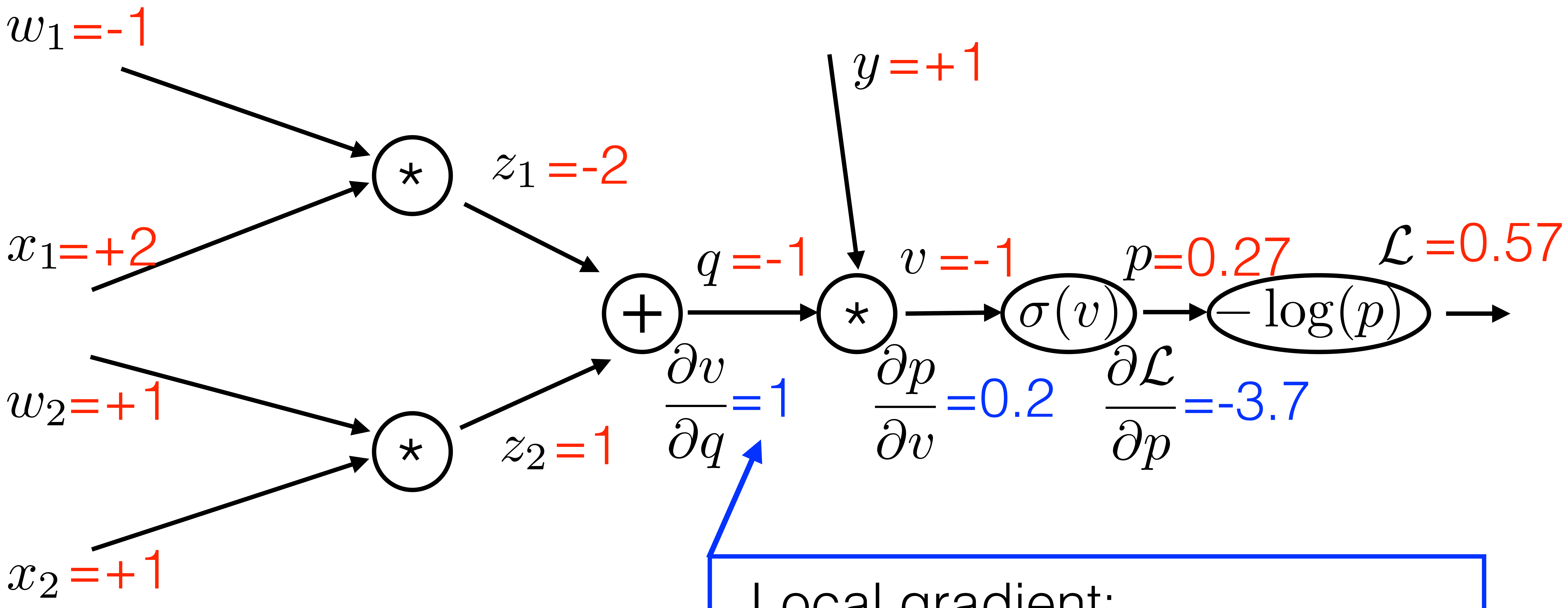
Learning in computation graph



Local gradient:

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

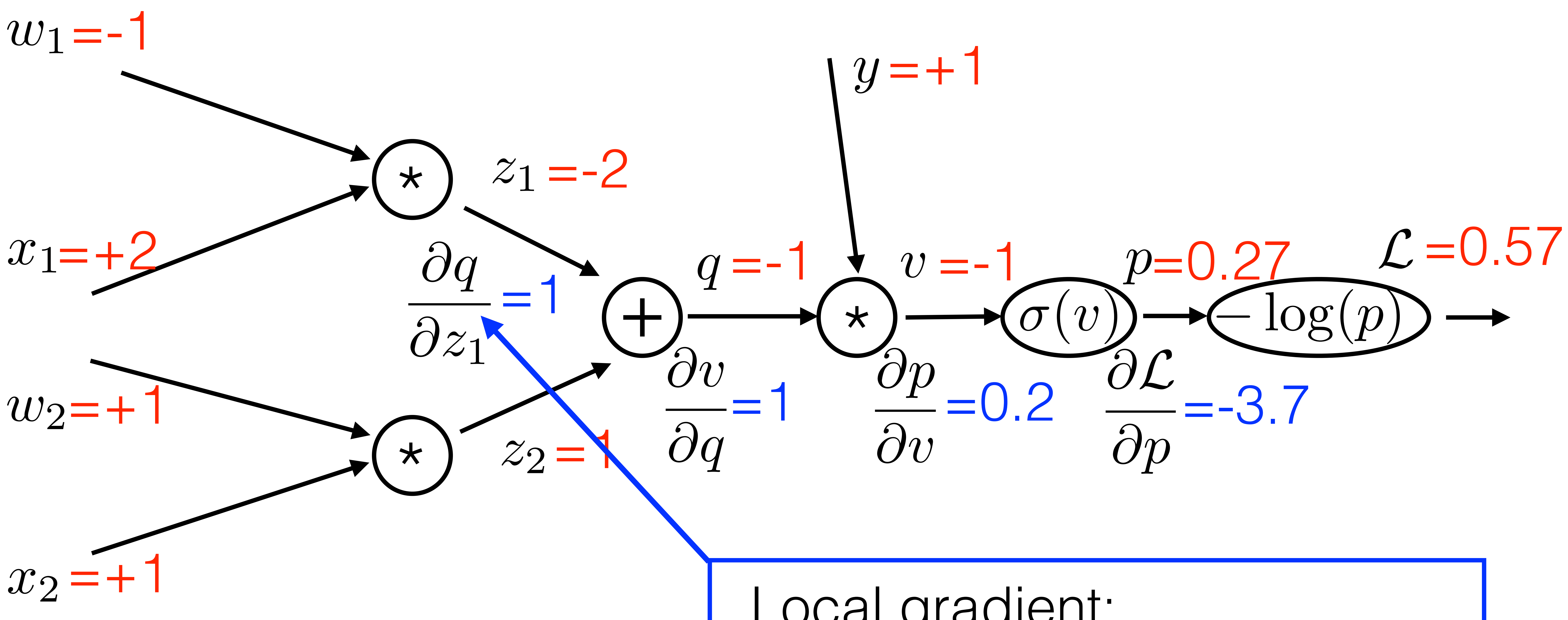
Learning in computation graph



Local gradient:

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

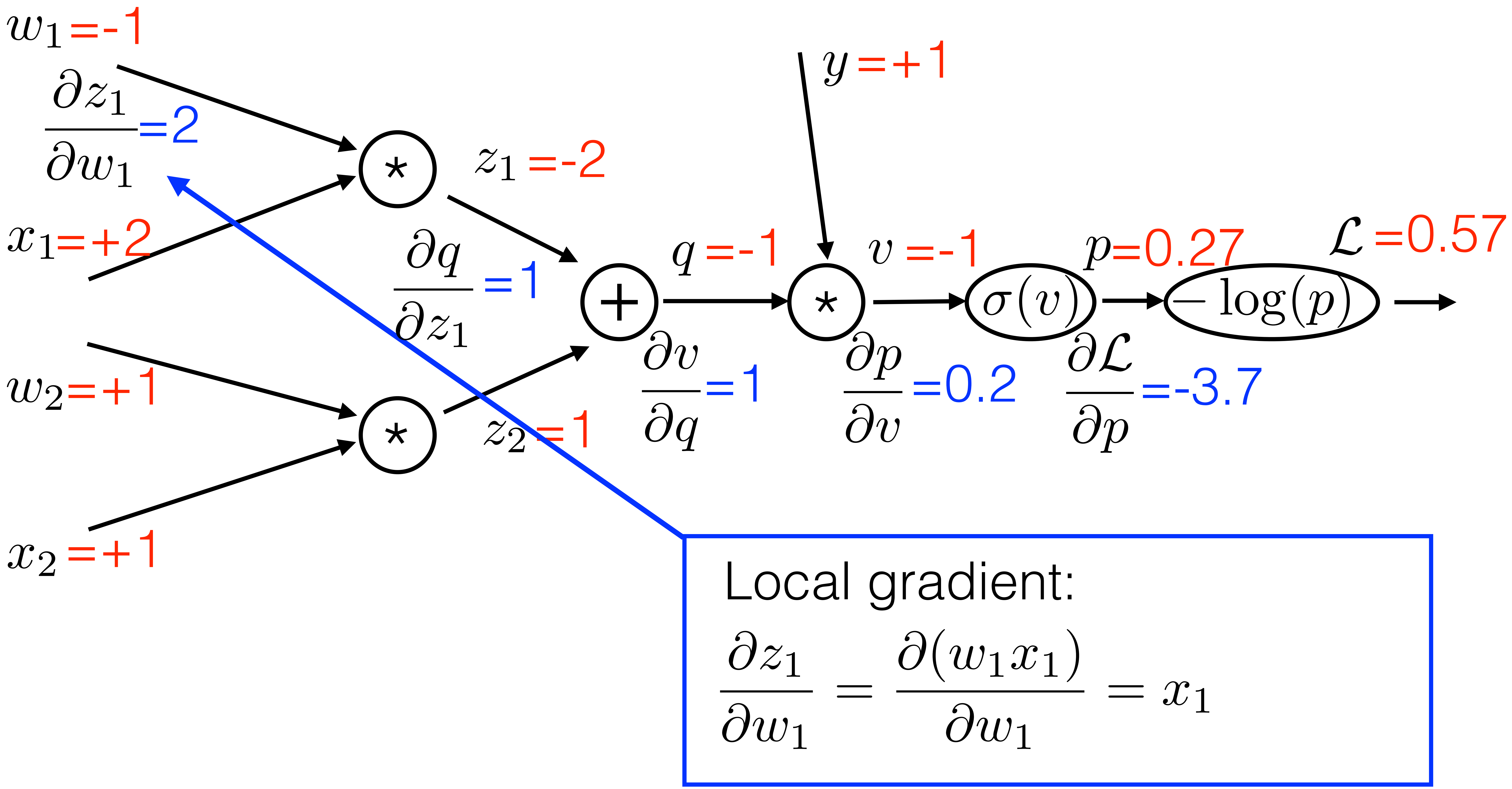
Learning in computation graph



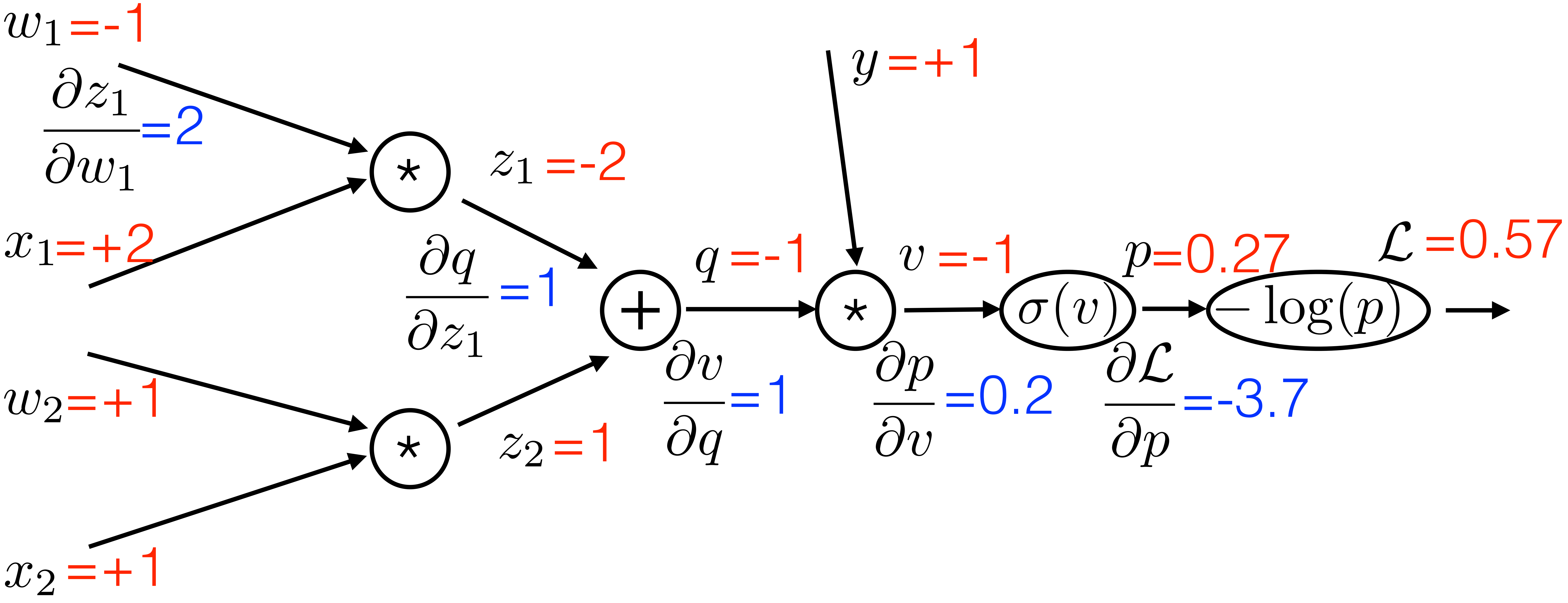
Local gradient:

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

Learning in computation graph

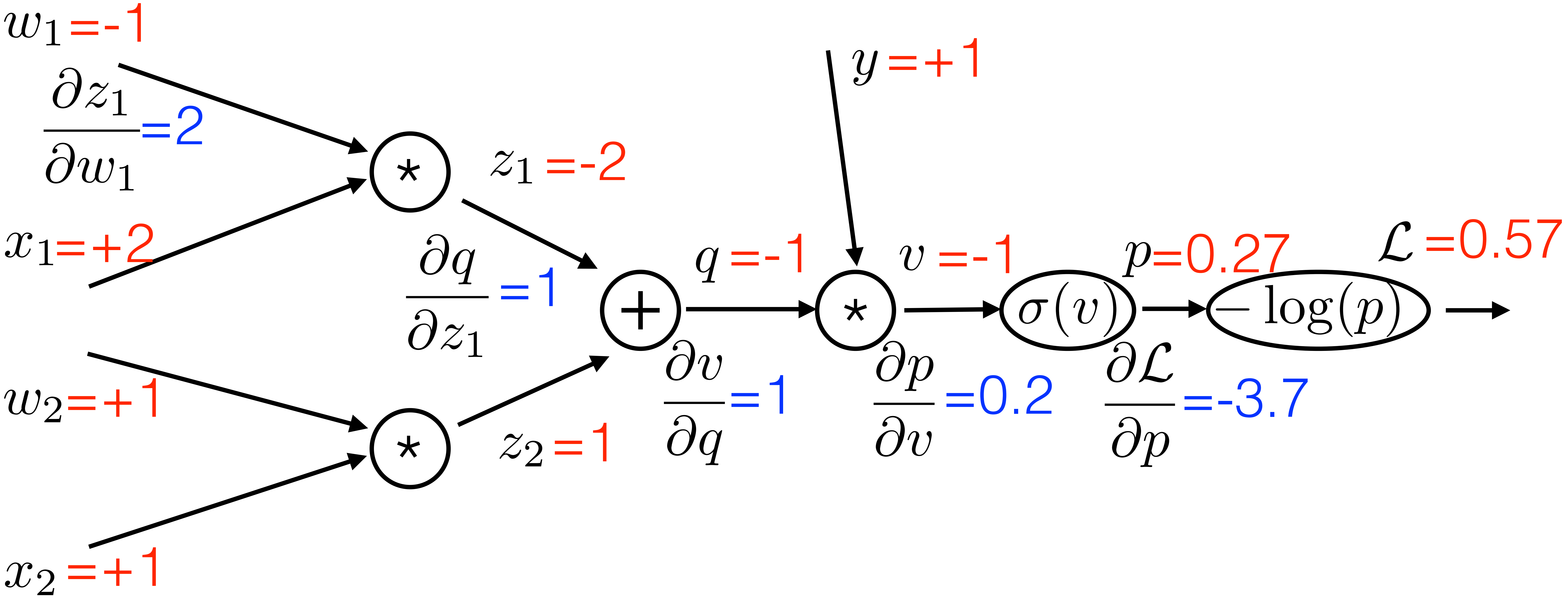


Learning in computation graph



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

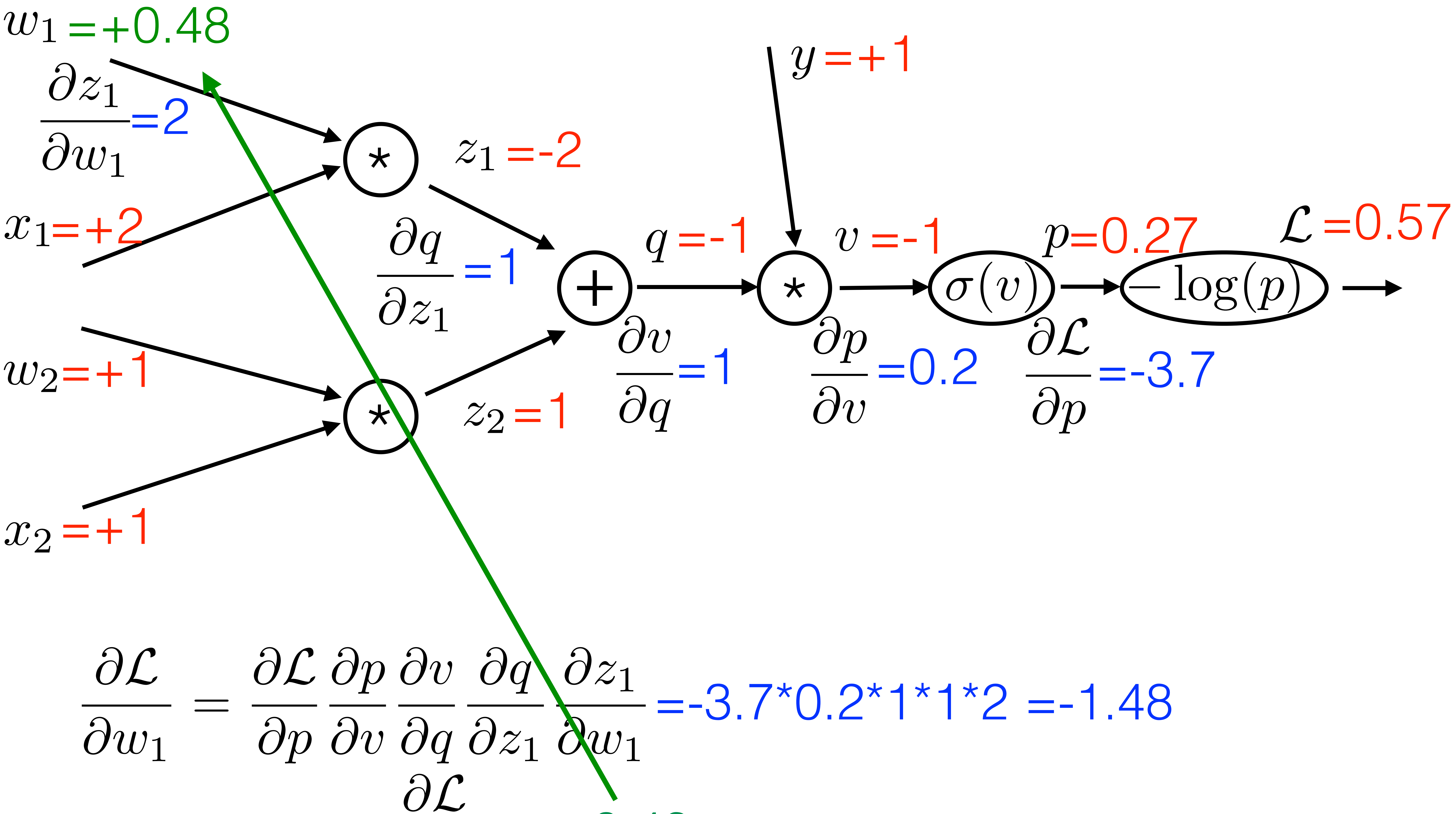
Learning in computation graph



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

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

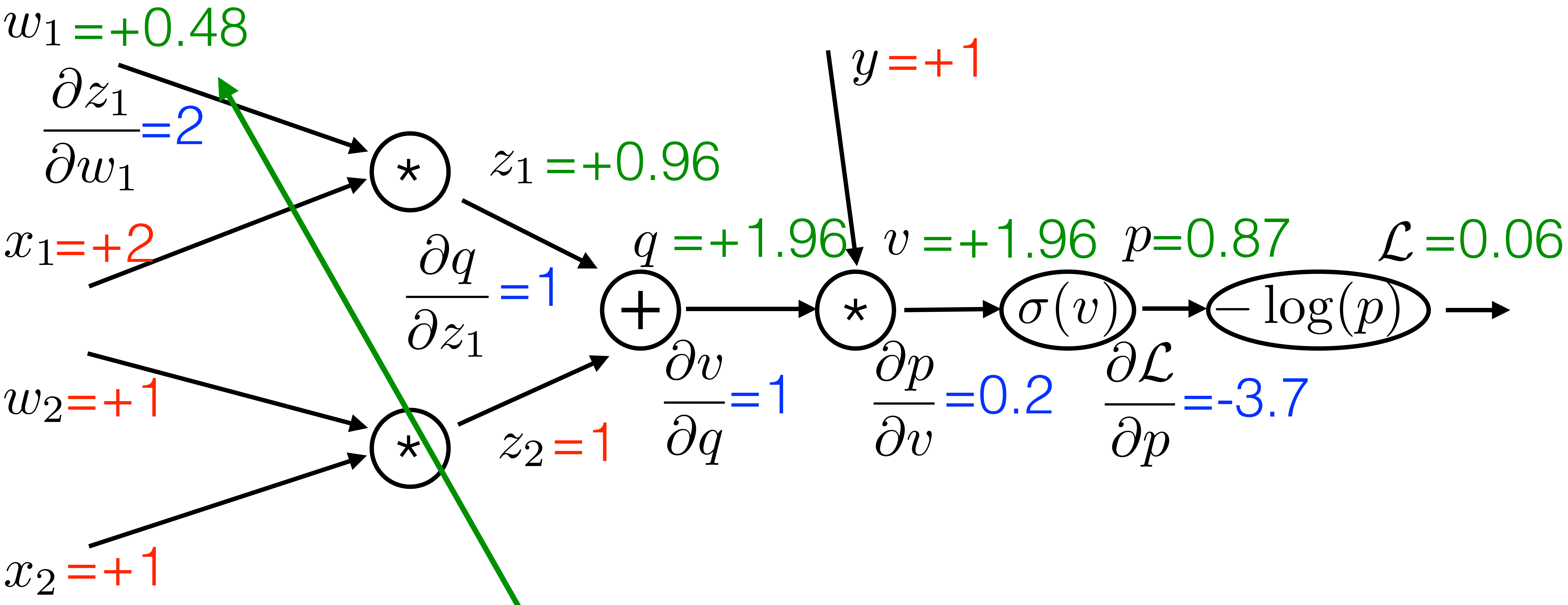
Learning in computation graph



$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial p}\frac{\partial p}{\partial v}\frac{\partial v}{\partial q}\frac{\partial q}{\partial z_1}\frac{\partial z_1}{\partial w_1}$$ =-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

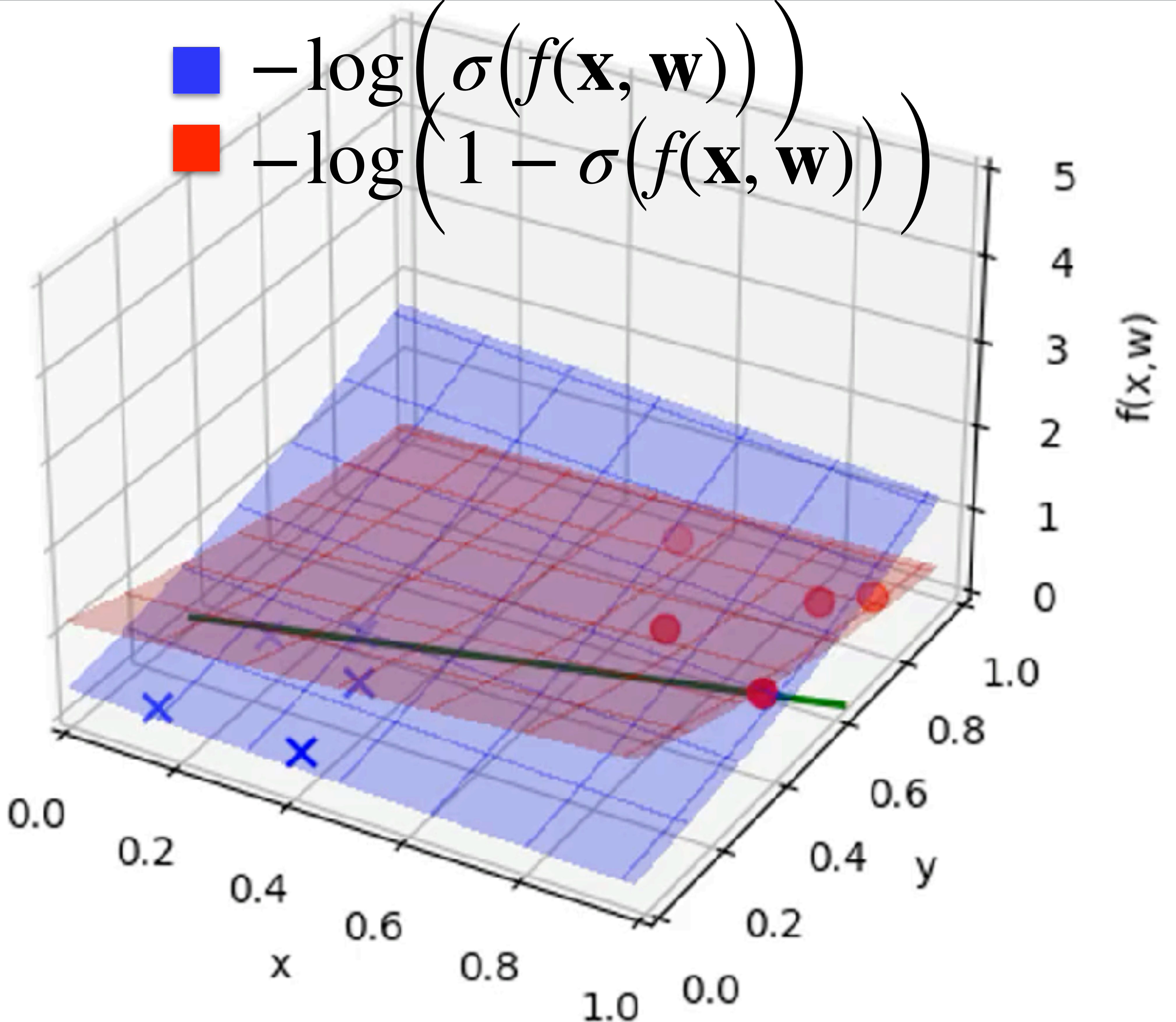


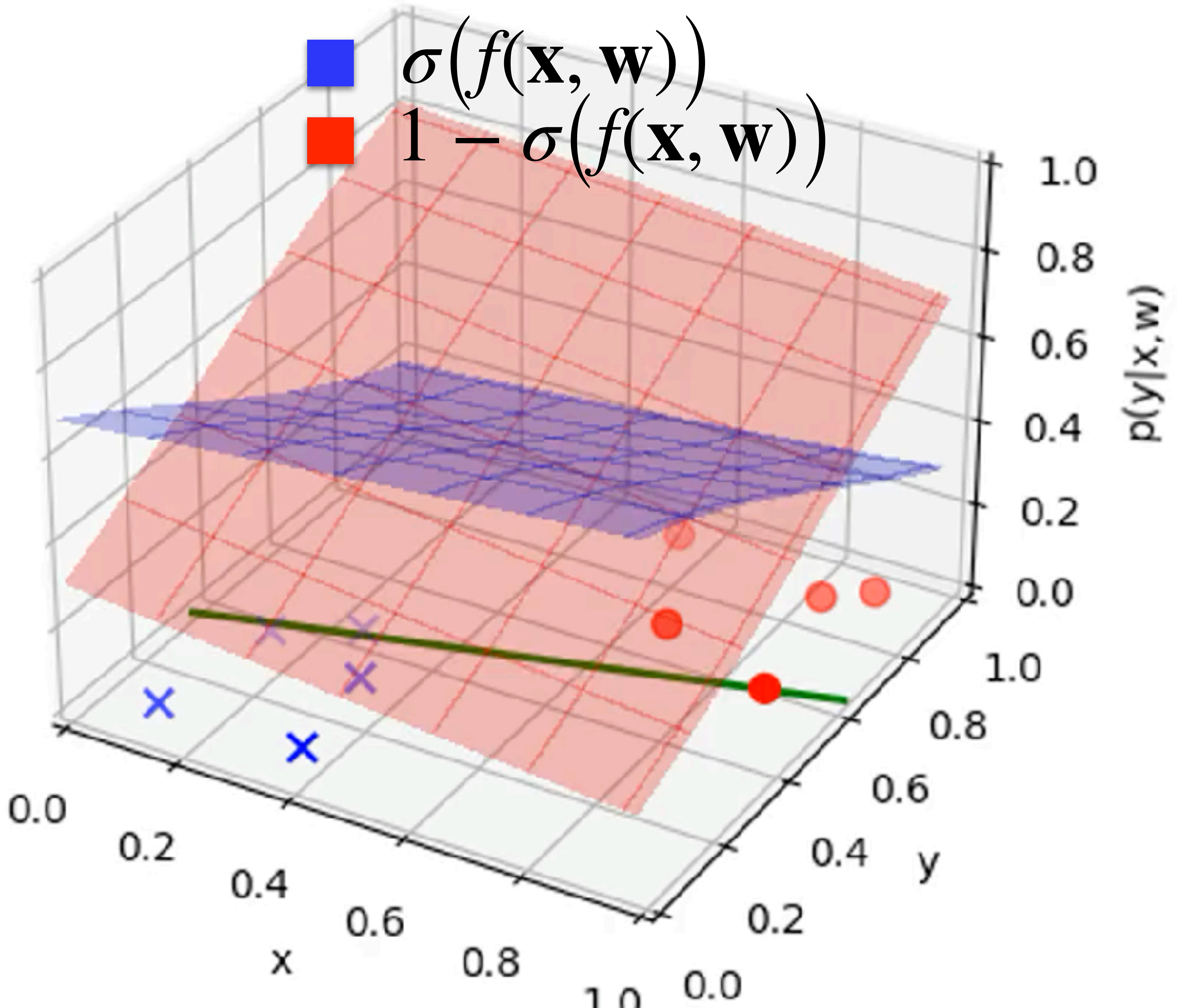
$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial p}\frac{\partial p}{\partial v}\frac{\partial v}{\partial q}\frac{\partial q}{\partial z_1}\frac{\partial z_1}{\partial w_1}$$ =-3.7*0.2*1*1*2 =-1.48

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

Numpy implementation

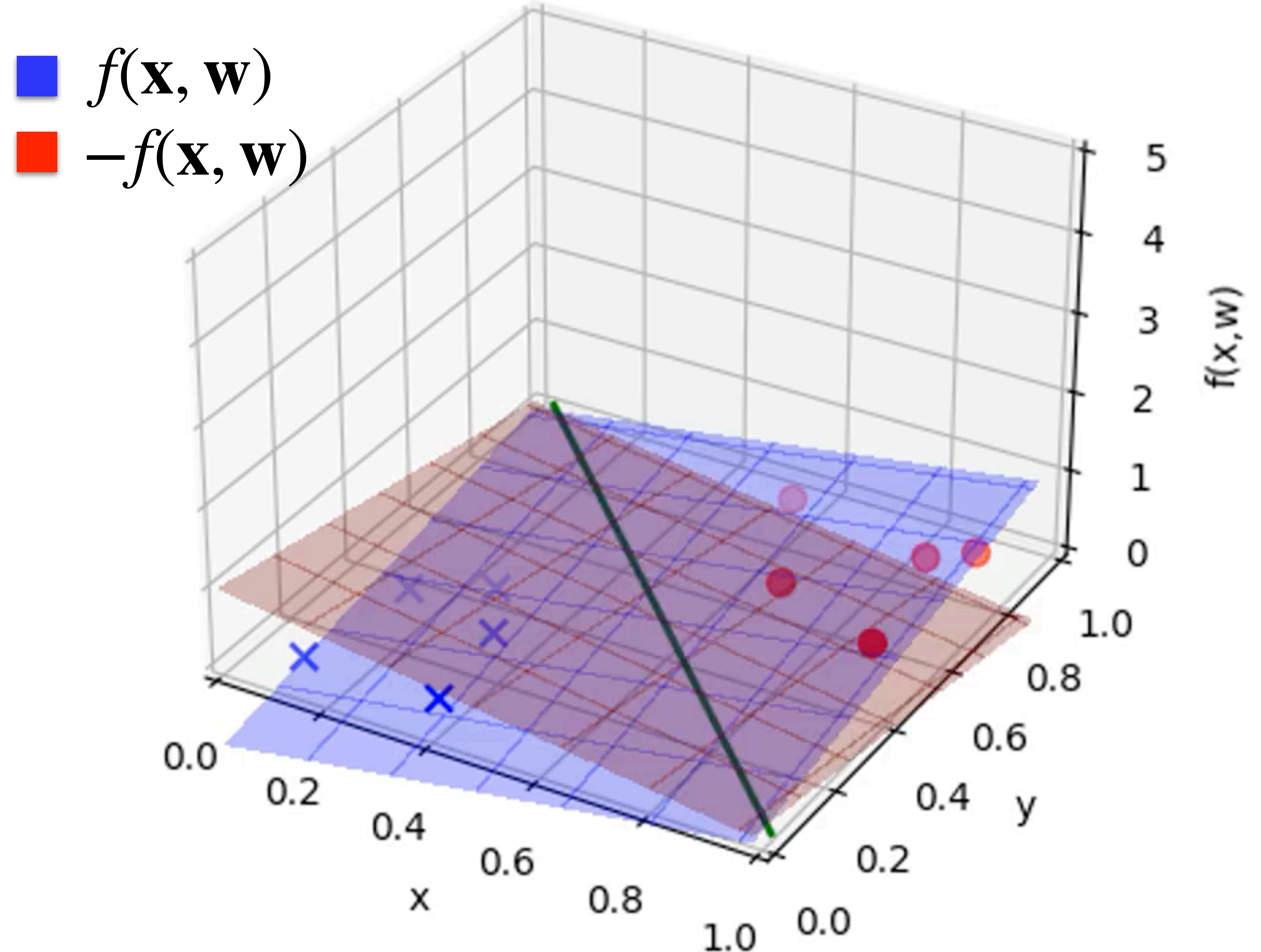
```
for i in range(30):
    u = w[0] * x[:, 0] + w[1] * x[:, 1] + w[2]
    p = sigmoid(u)
    loss = (-np.log(p)*y + -np.log(1-p)*(1-y)).sum()
    grad = -1/p * sigmoid(u) * (1-sigmoid(u)) * …
    w = w - 0.1 * grad
```



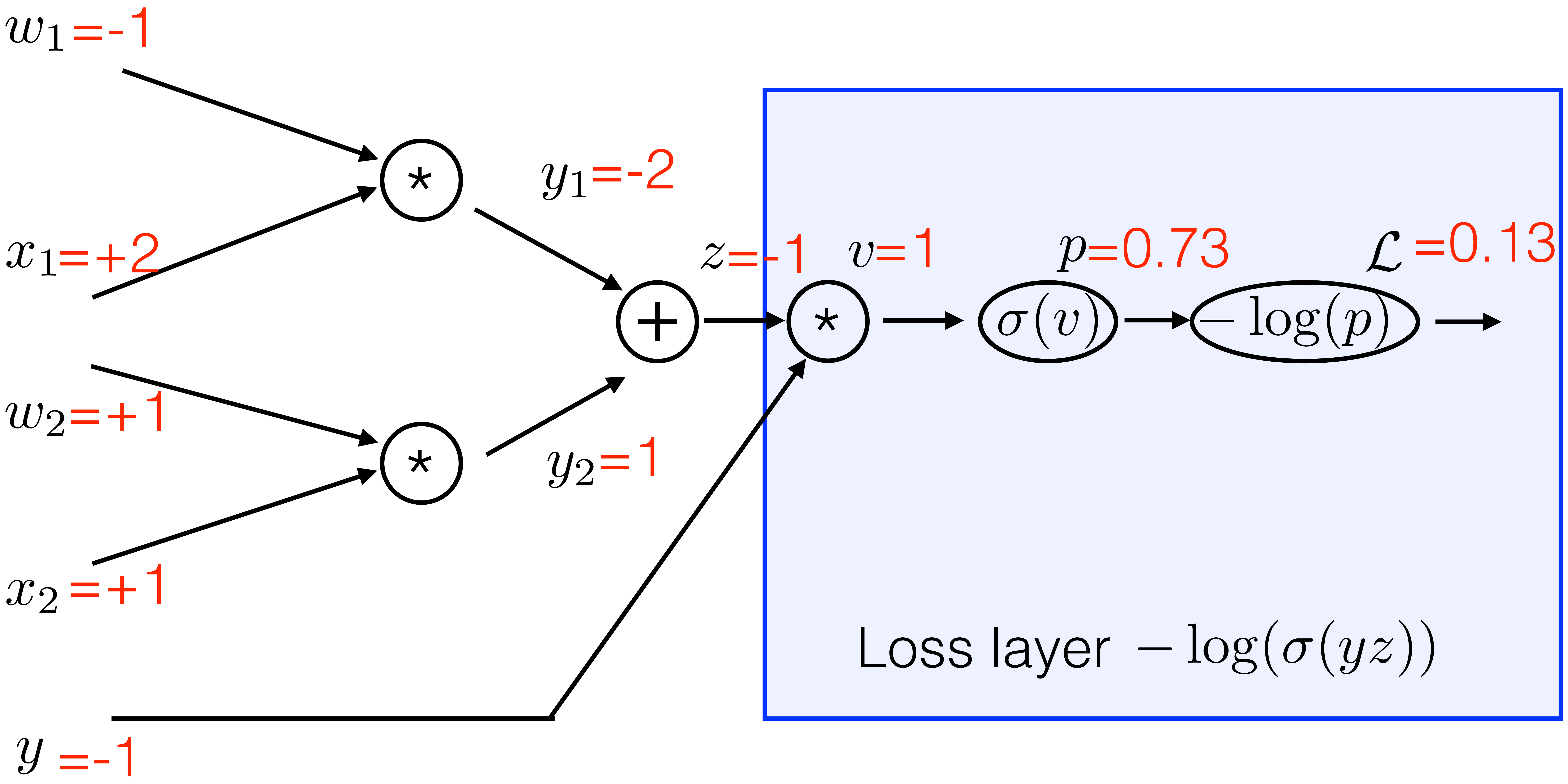


Numpy implementation

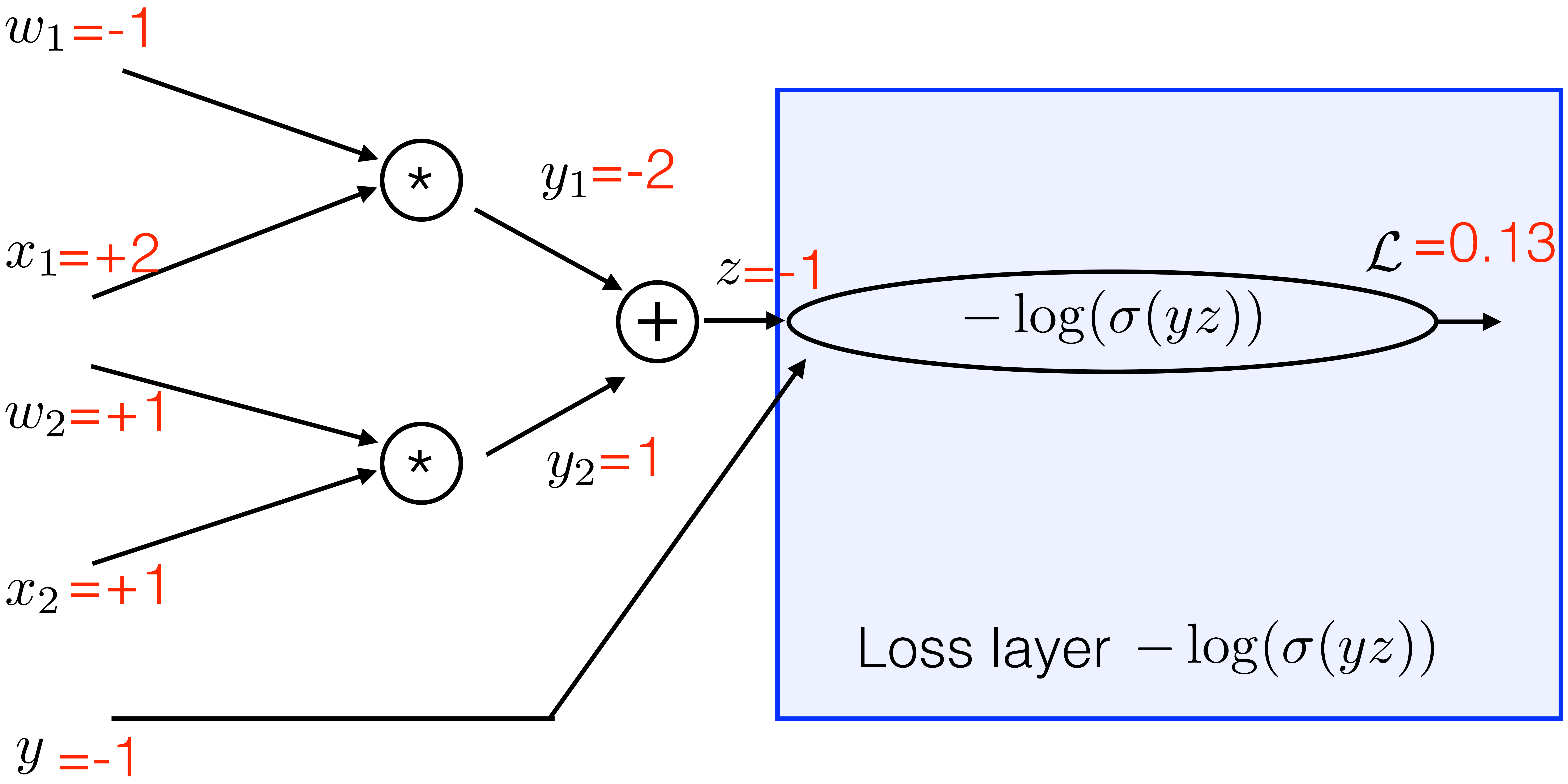
```
for i in range(30):
    u = w[0] * x[:, 0] + w[1] * x[:, 1] + w[2]
    loss = u*y + (1-u)*(1-y)).sum()
    grad = …
    w = w + 0.1 * grad
```



Computational graph of the learning



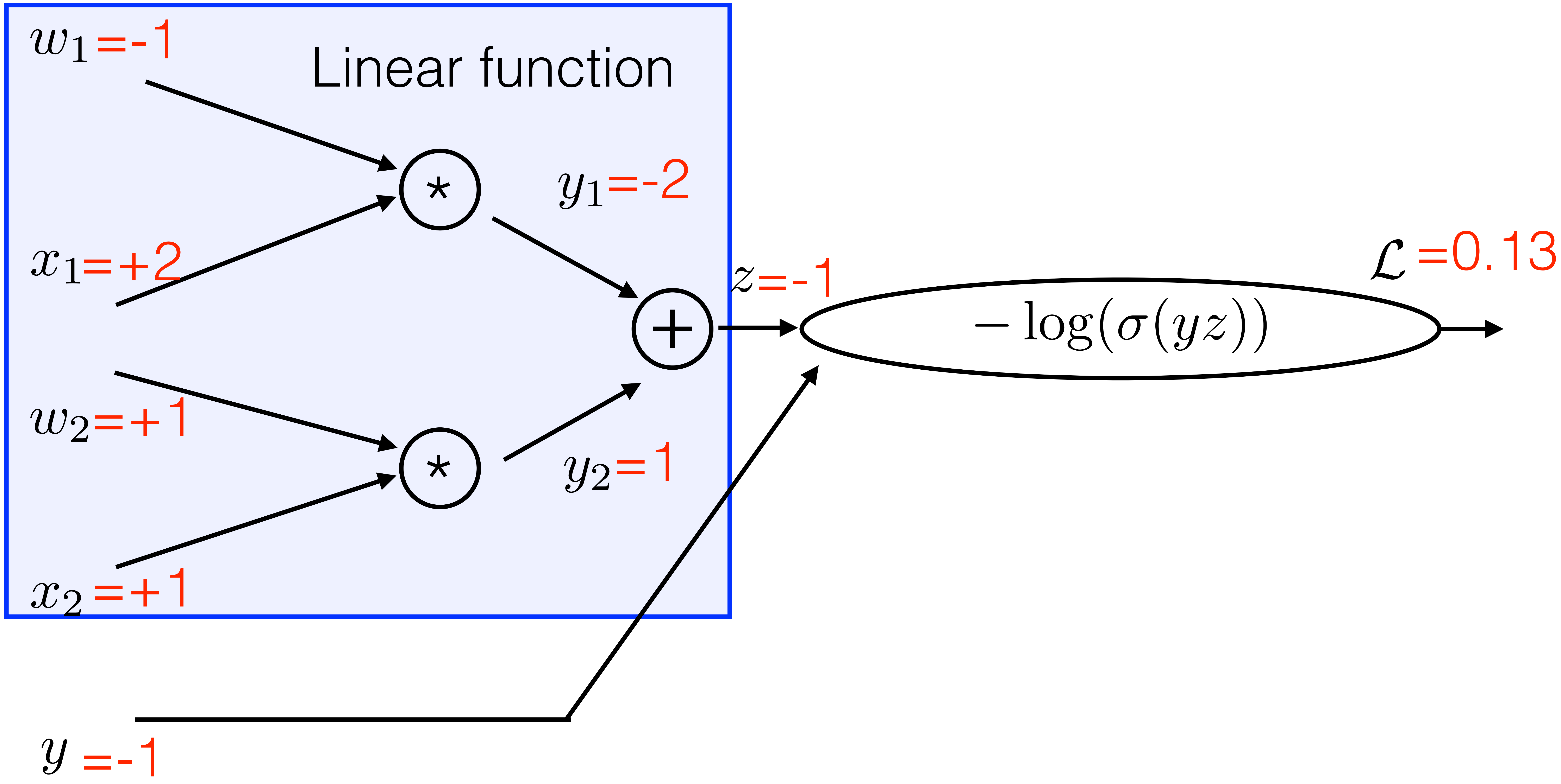
Backprop in vector representation



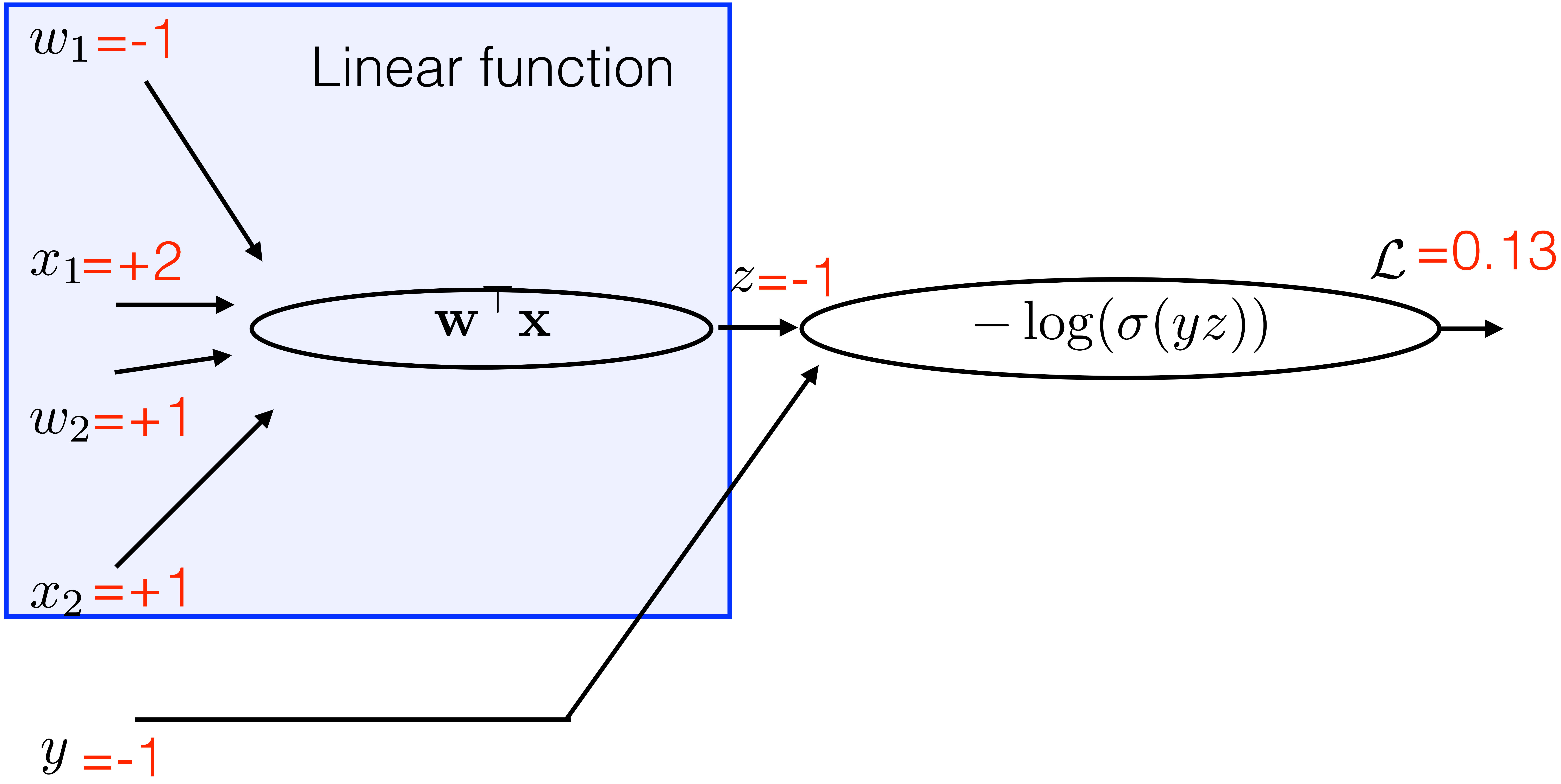
This is the logistic loss!

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

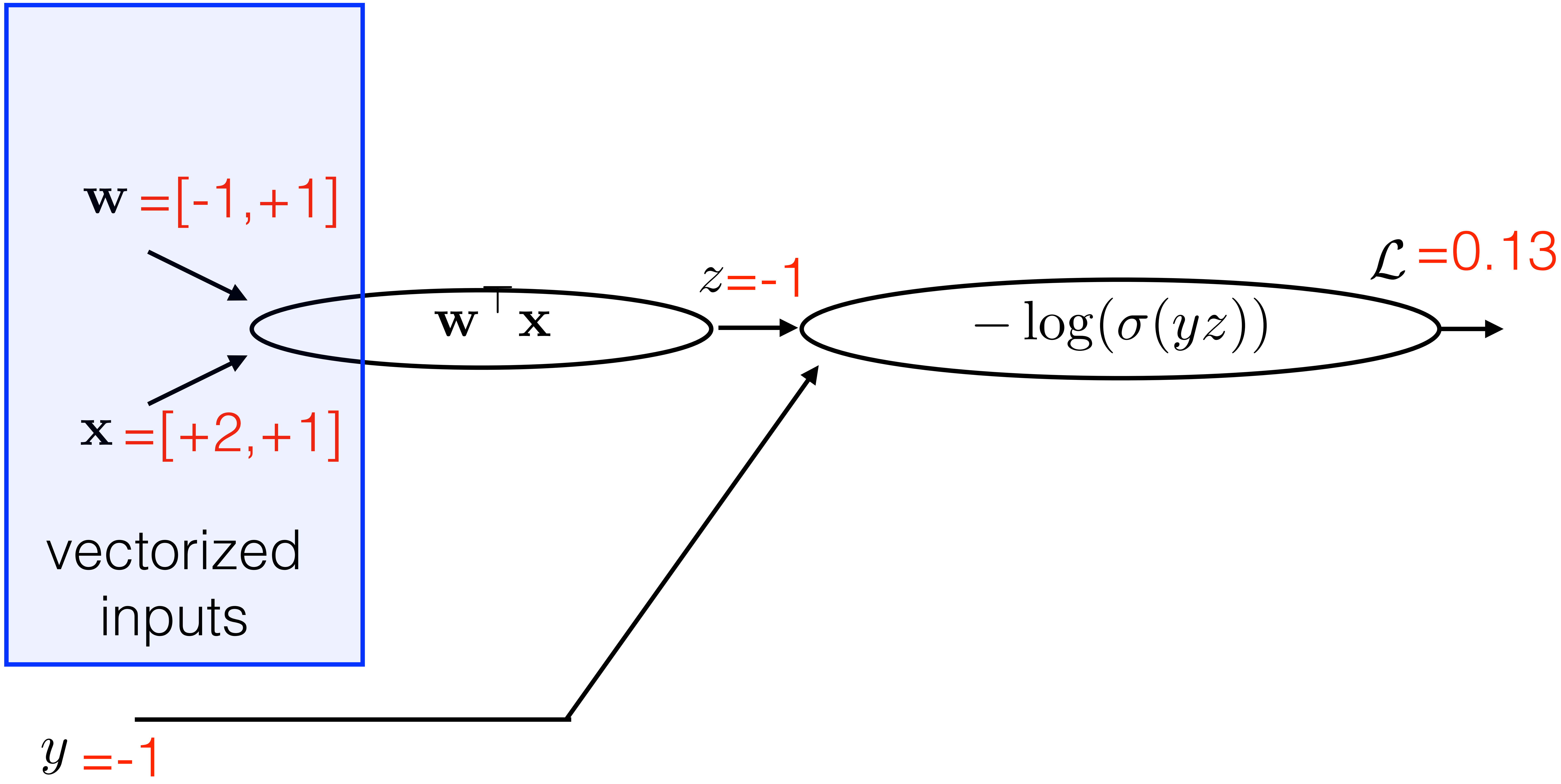
Backprop in vector representation



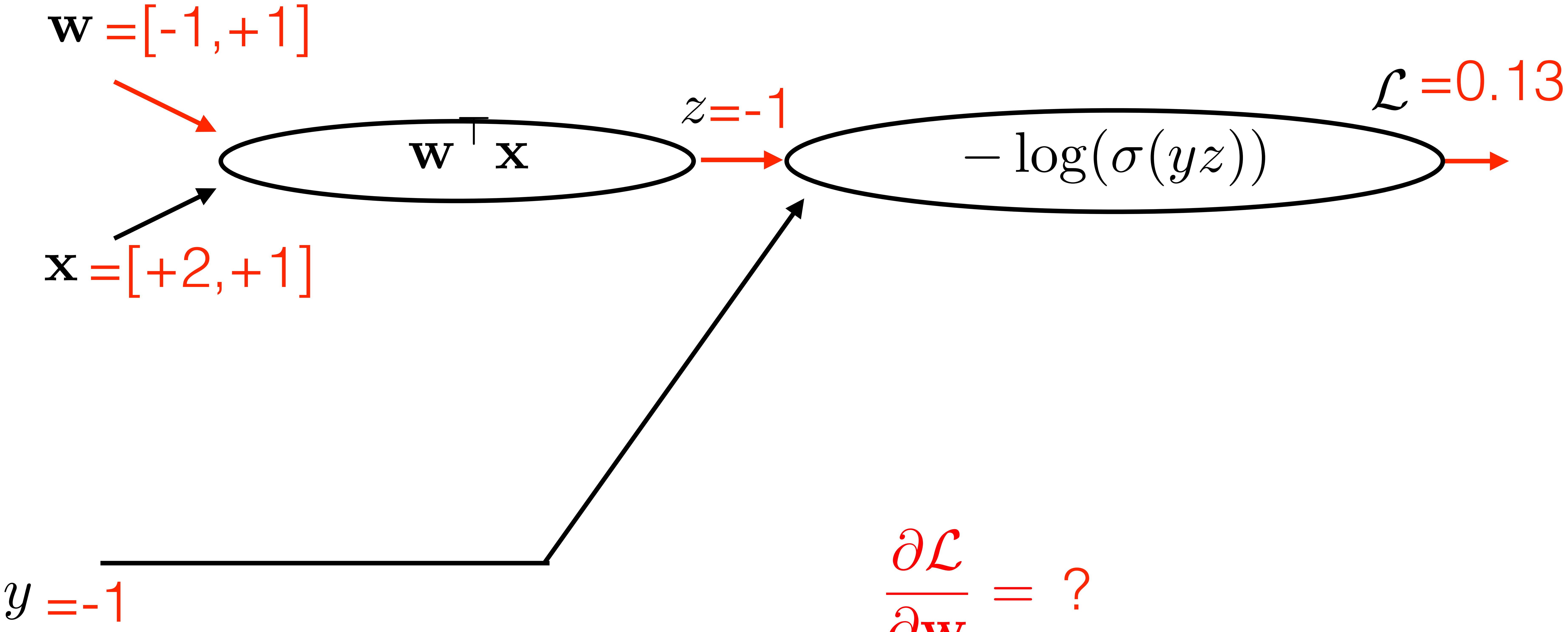
Backprop in vector representation



Backprop in vector representation

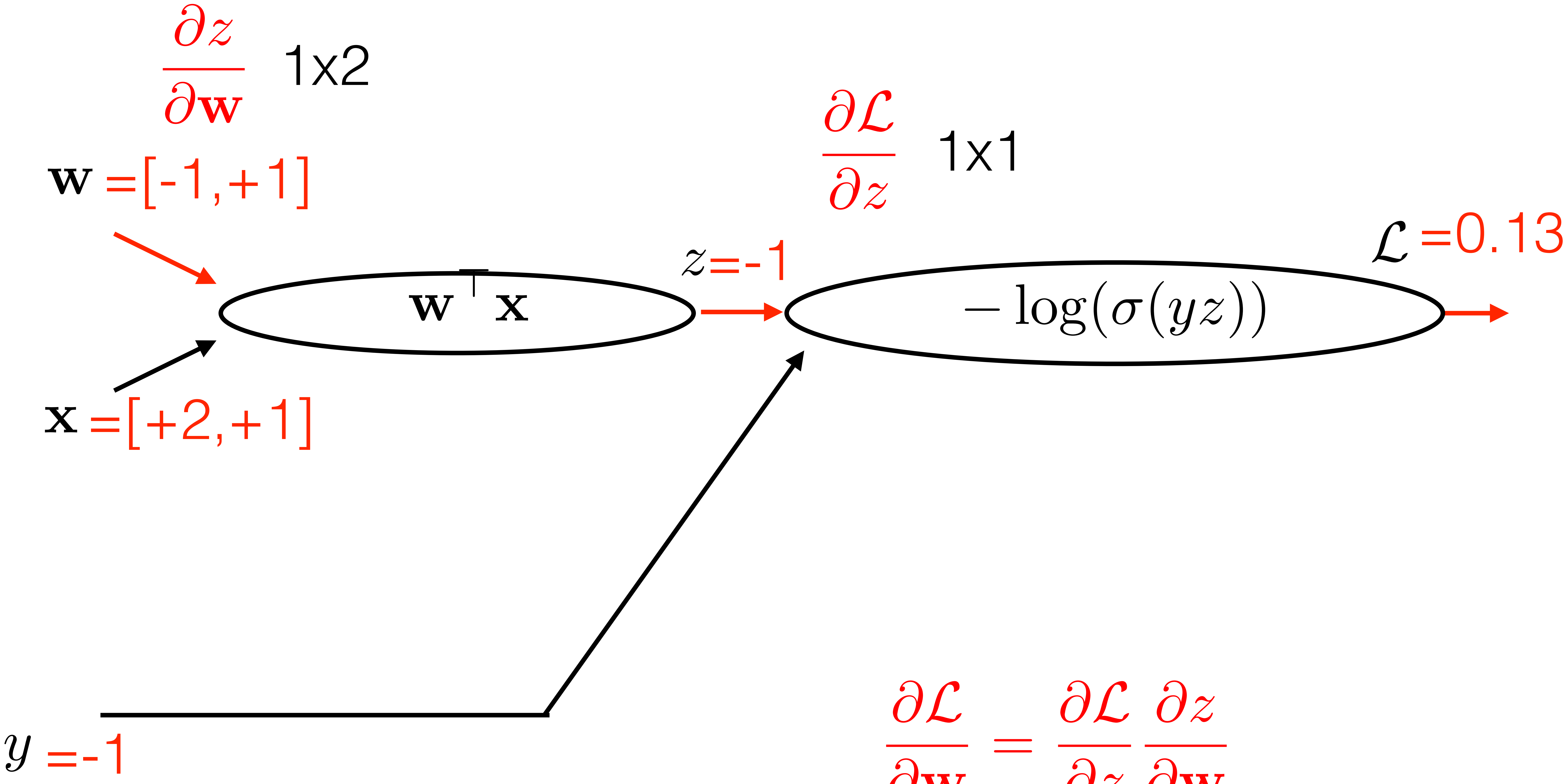


Backprop in vector representation



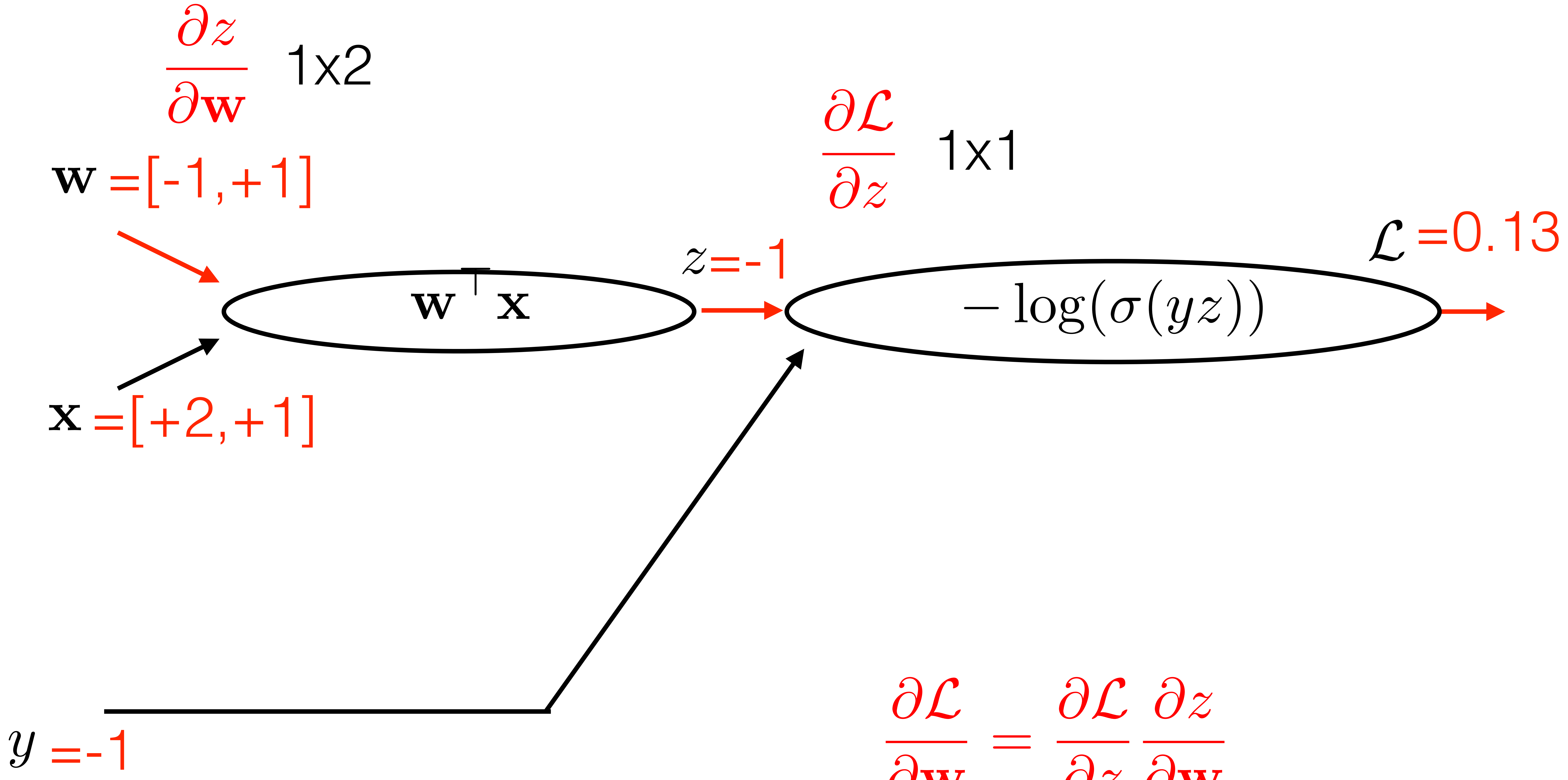
$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = ?$$

Backprop in vector representation



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

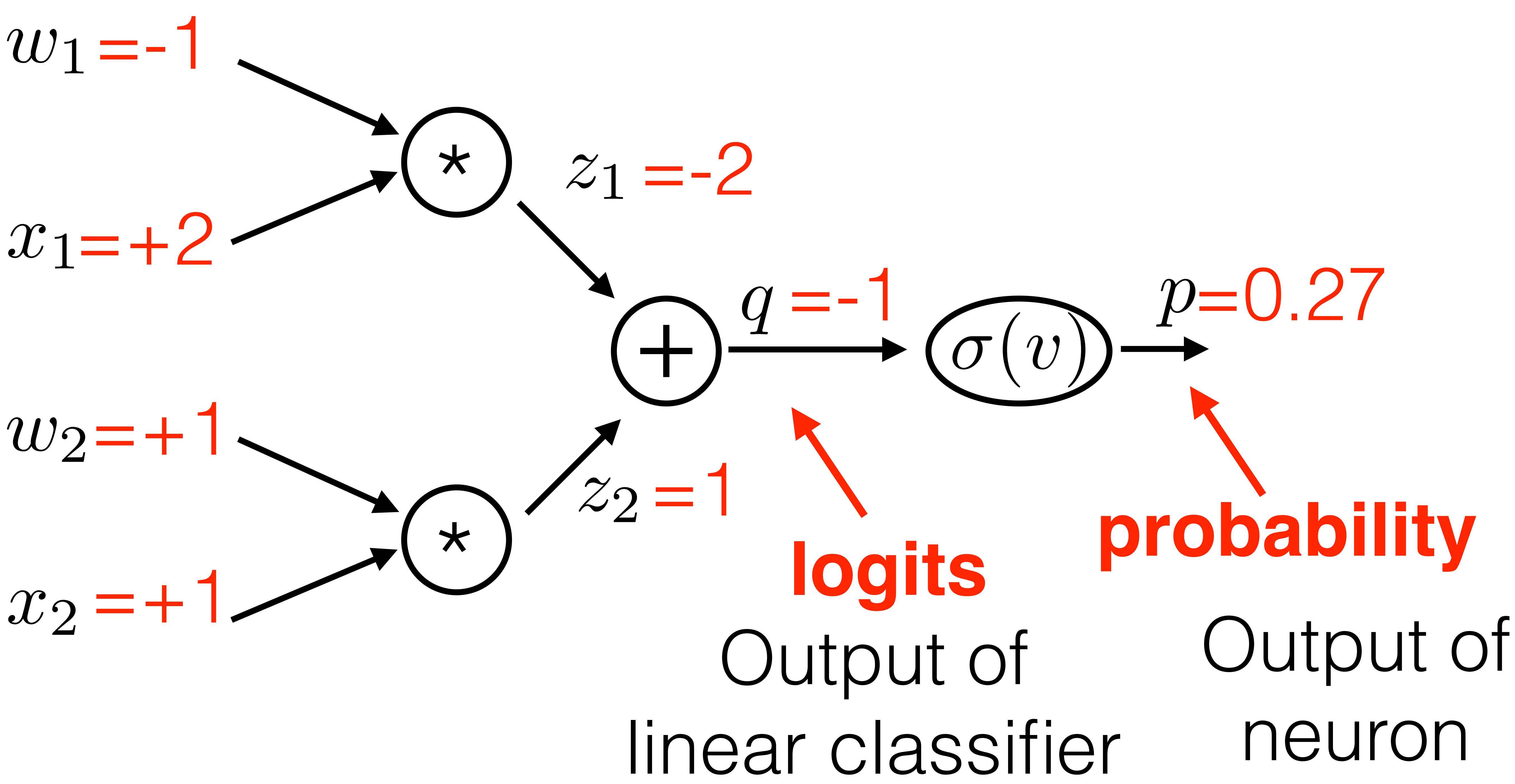
Backprop in vector representation



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

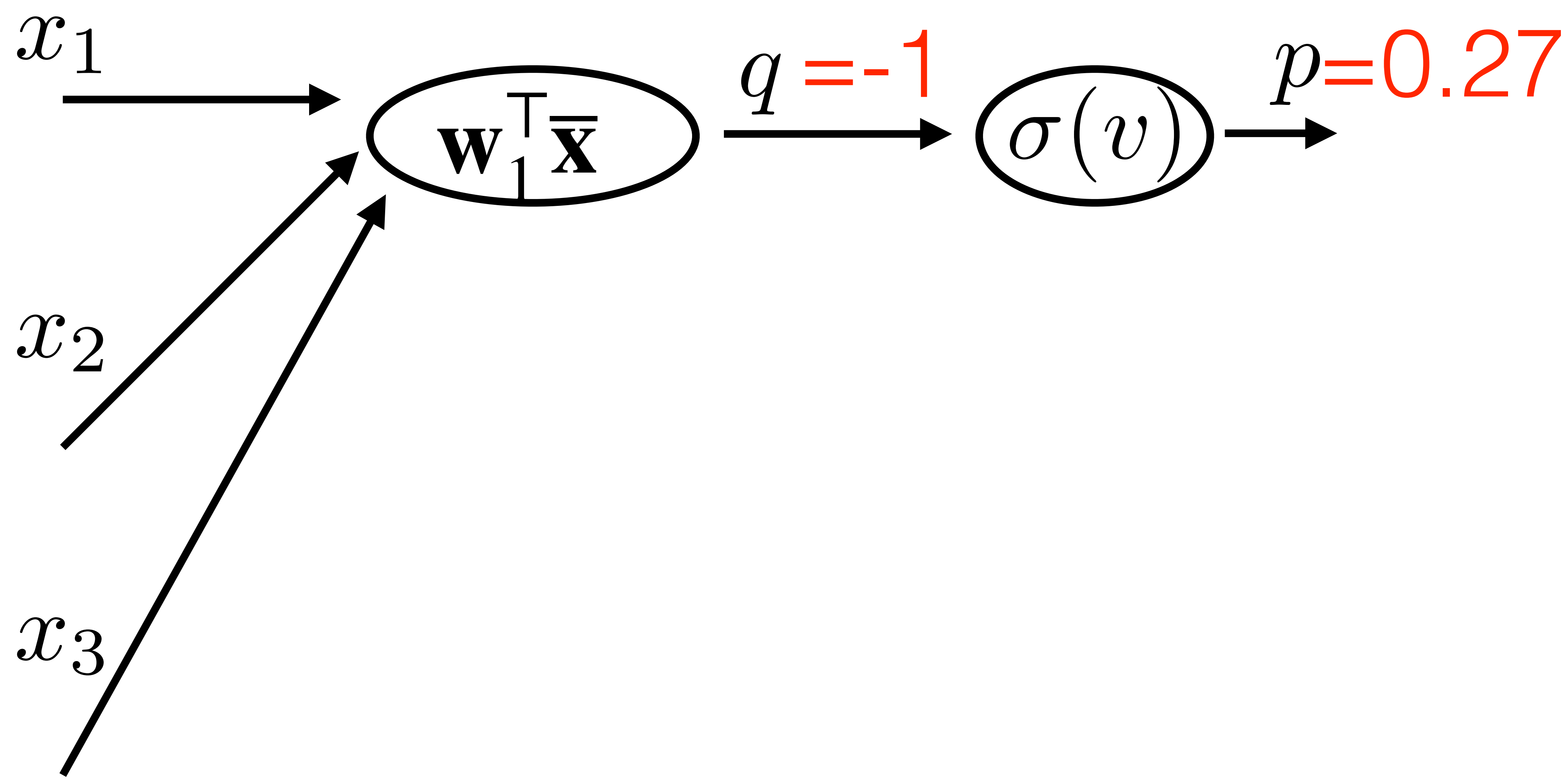
Learning from multiple training samples means summing up the partial derivative over all samples

Neuron

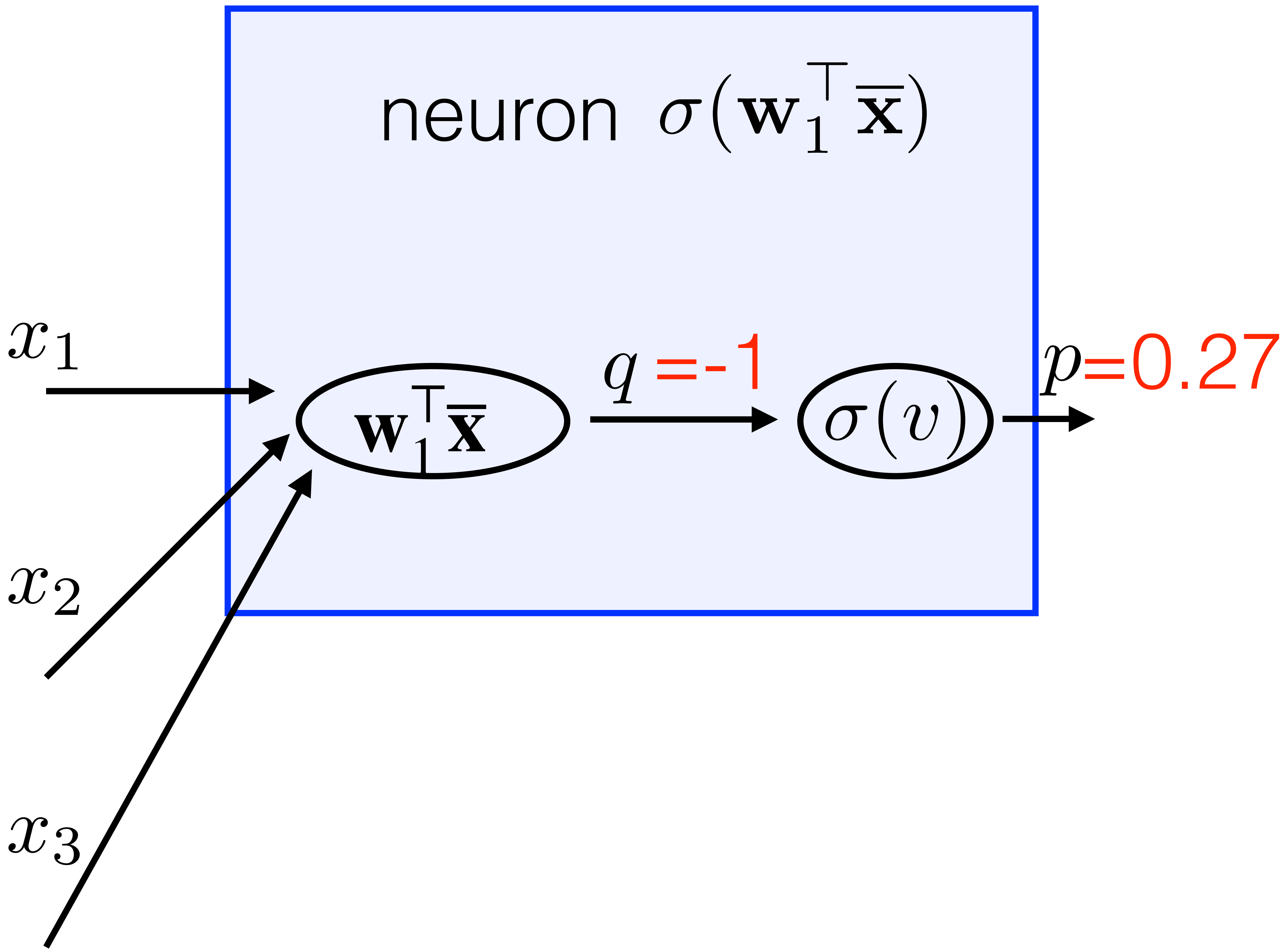


w_1=-1
x_1=+2
w_2=+1
x_2=+1
*
*
z_1 =-2
z_2 =1
+
q =-1
$\sigma(v)$
p=0.27
logits
Output of linear classifier
probability
Output of neuron

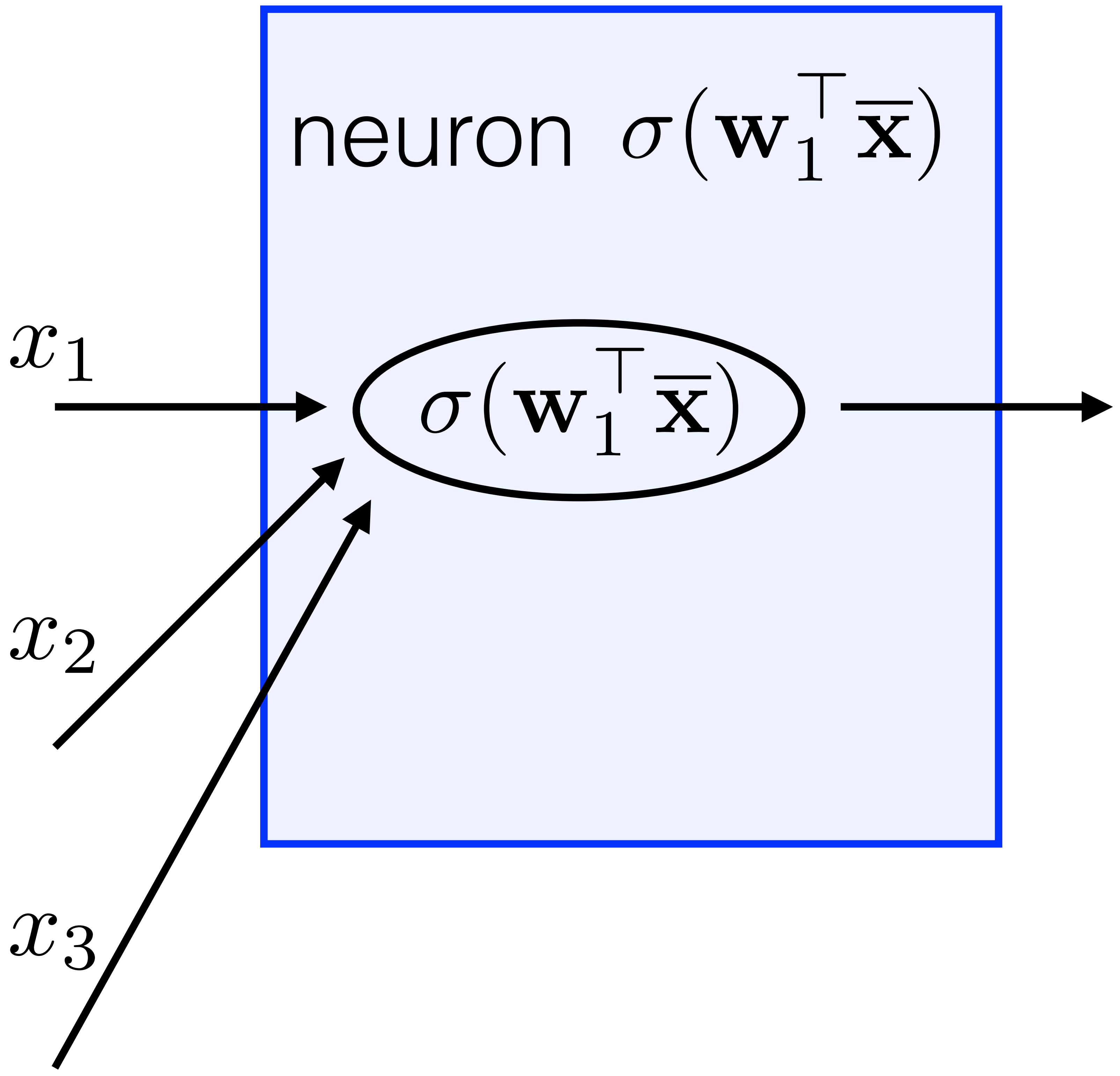
Neuron



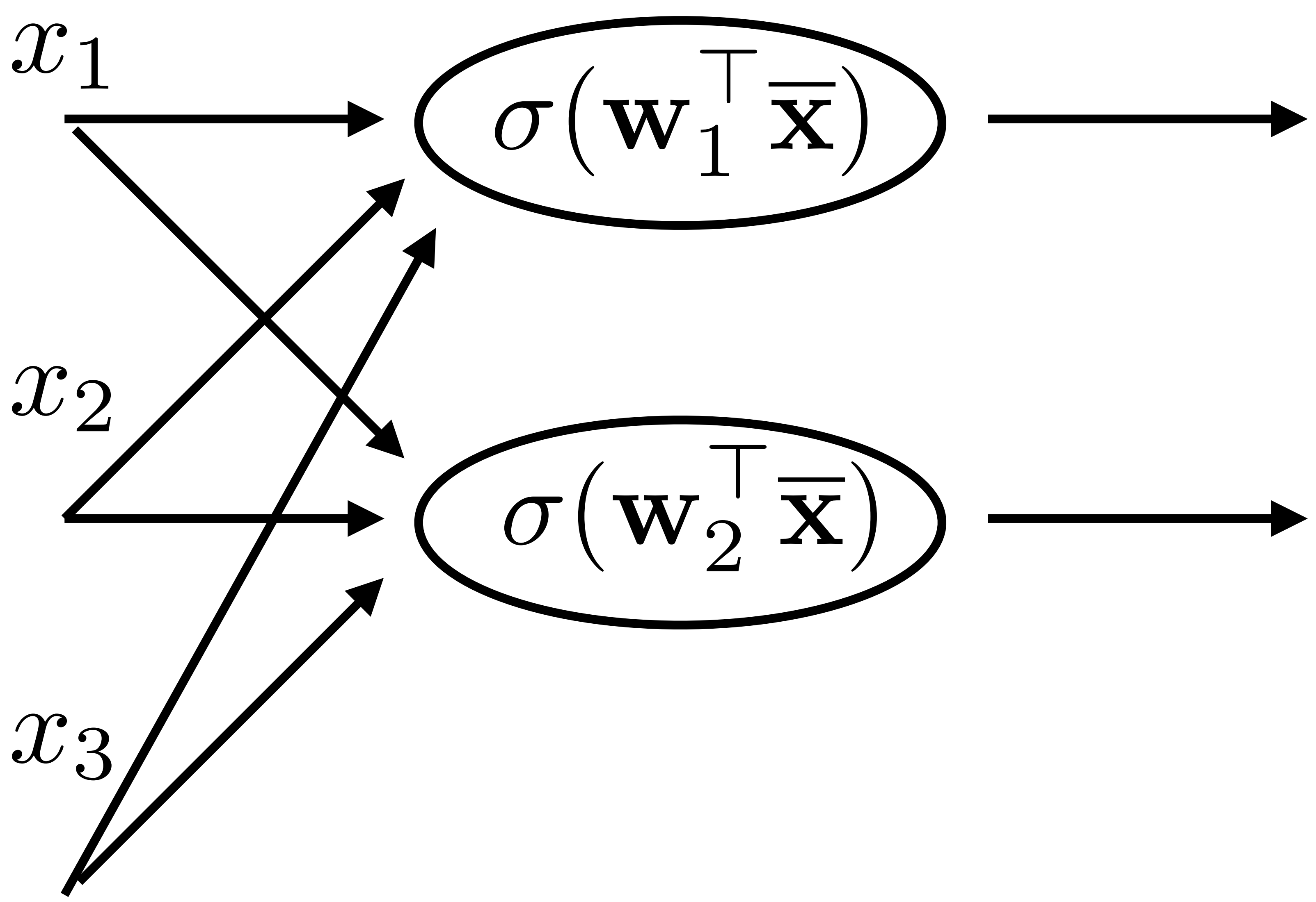
Neuron


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

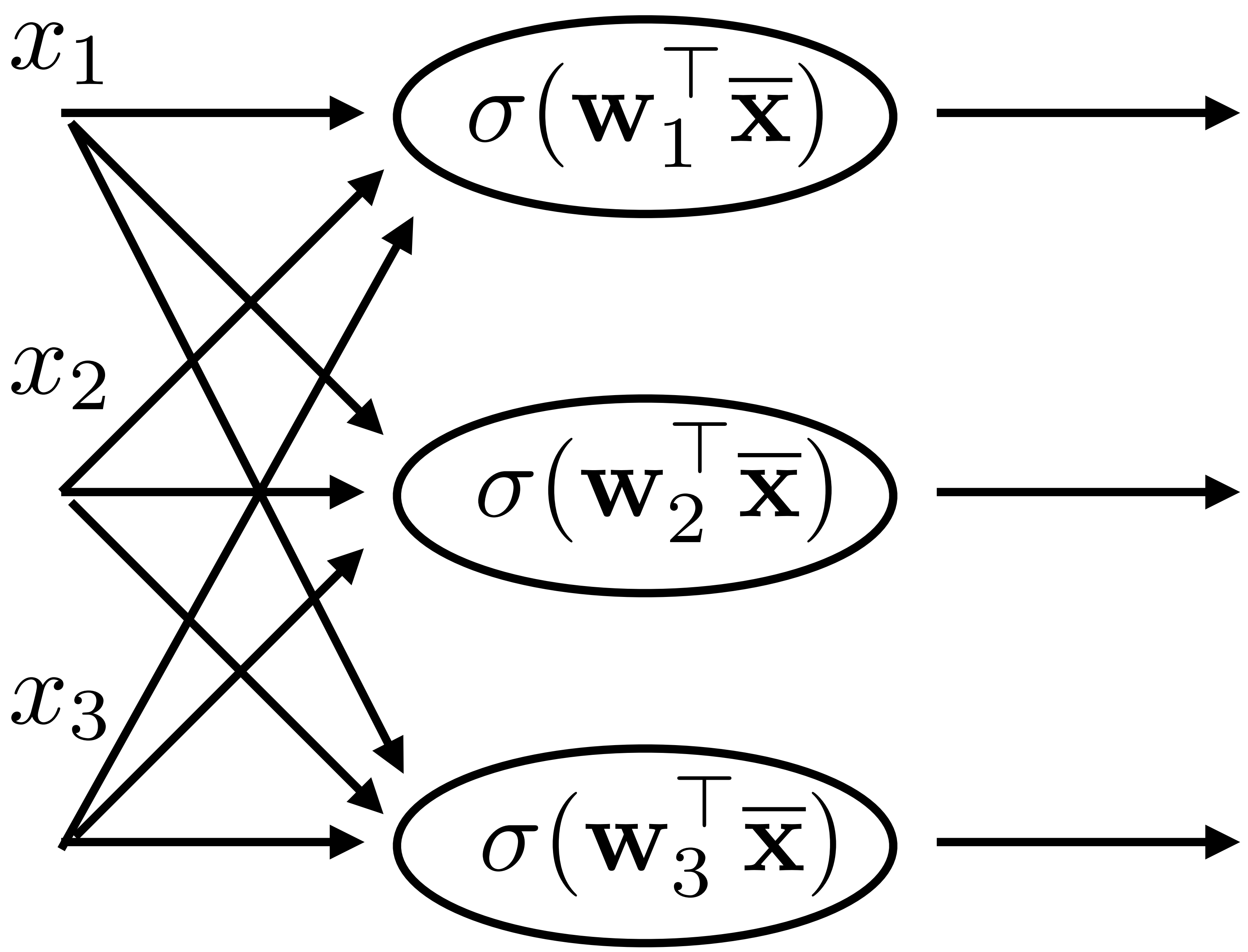
Fully-connected neural network



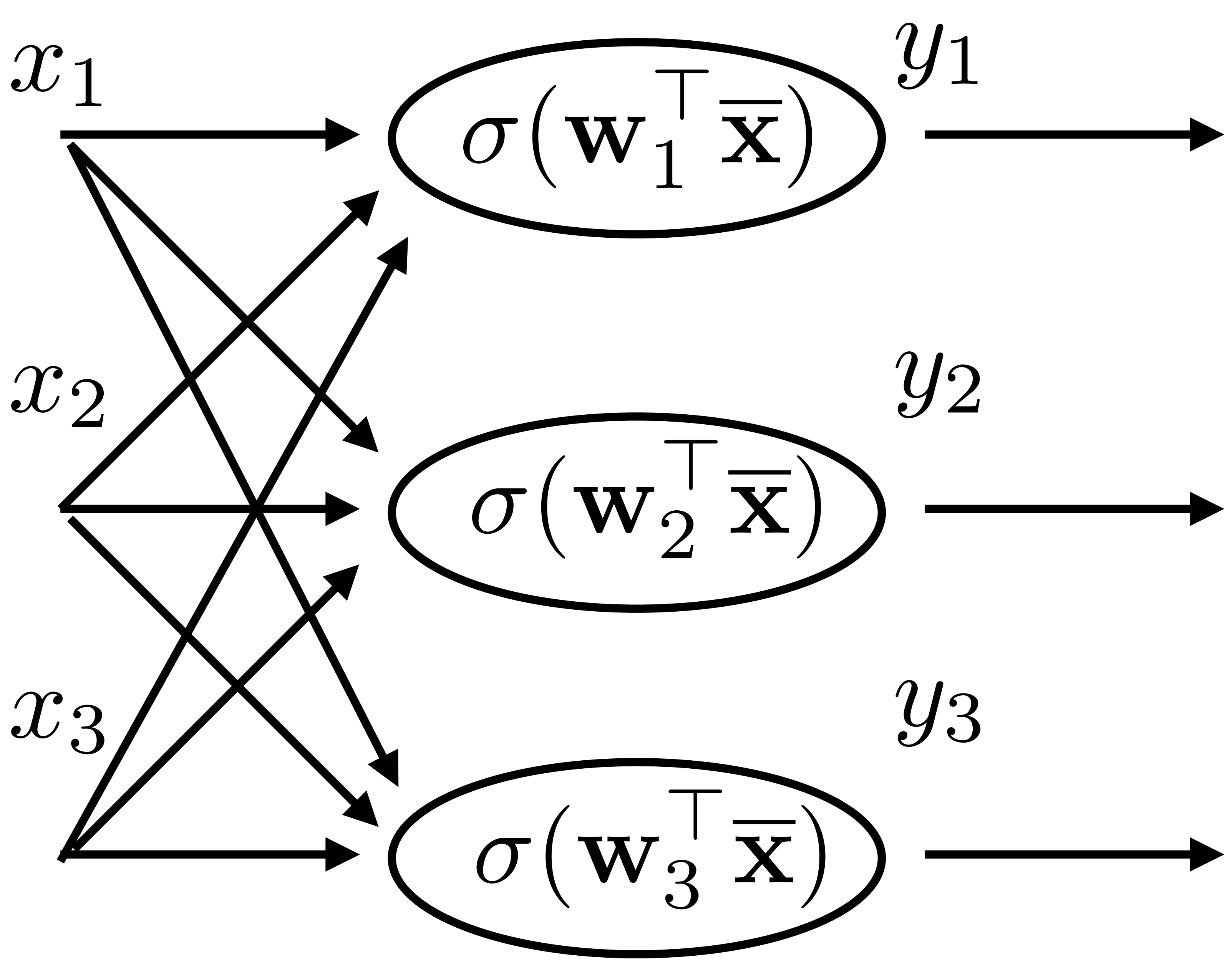
Fully-connected neural network



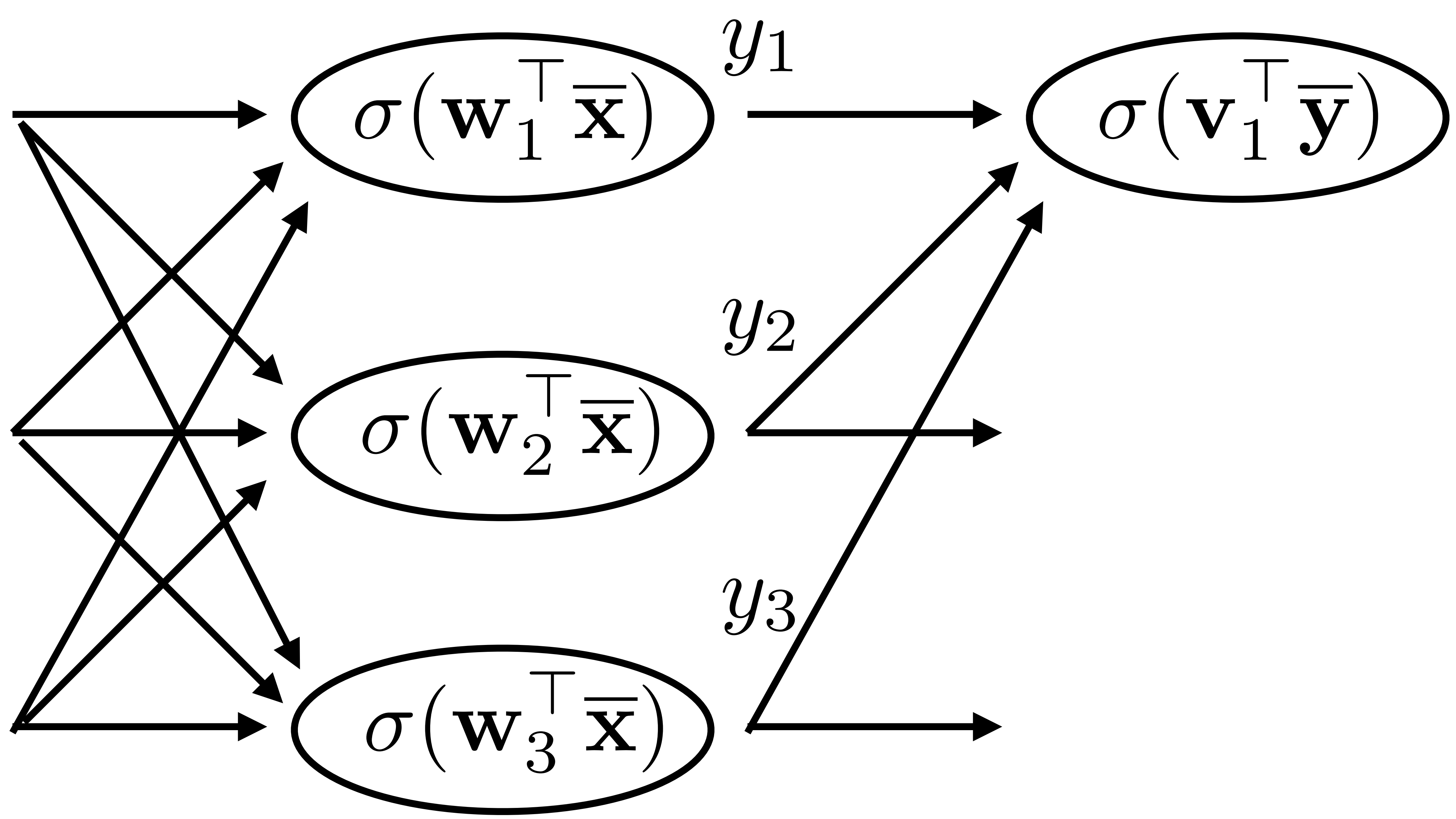
Fully-connected neural network



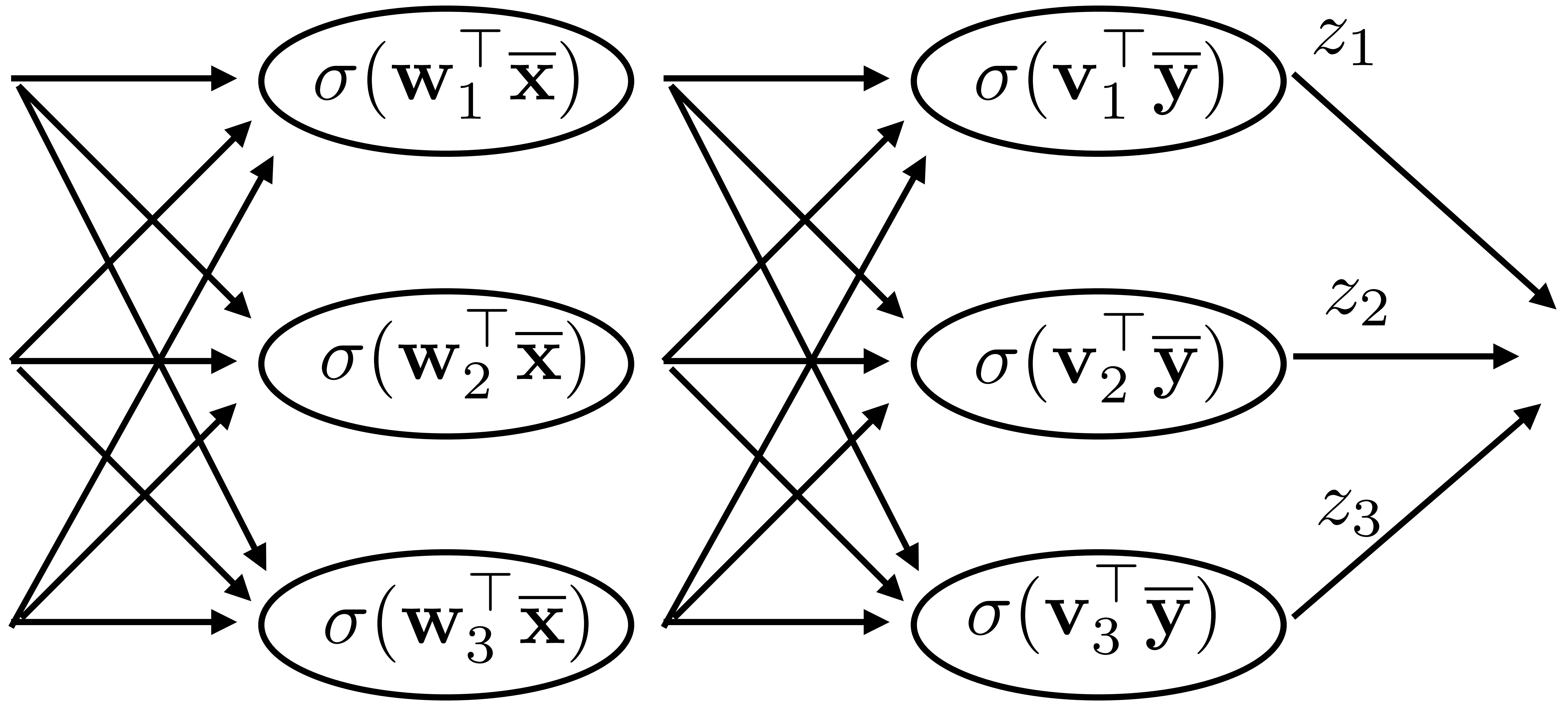
Fully-connected neural network



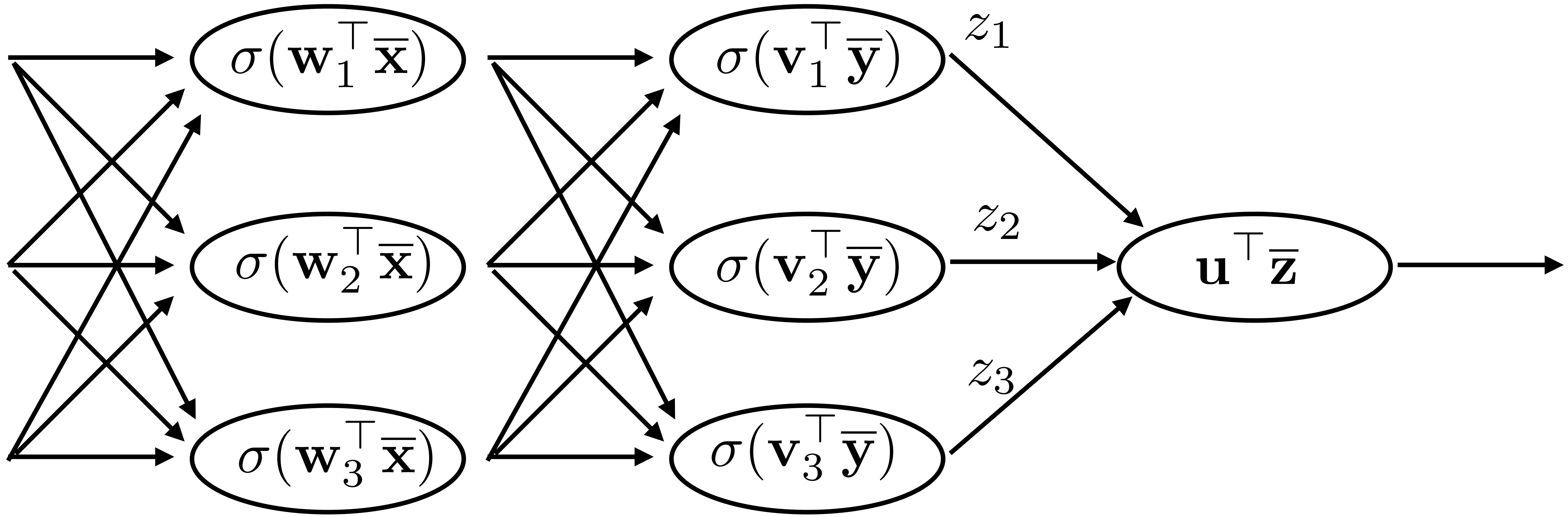
Fully-connected neural network



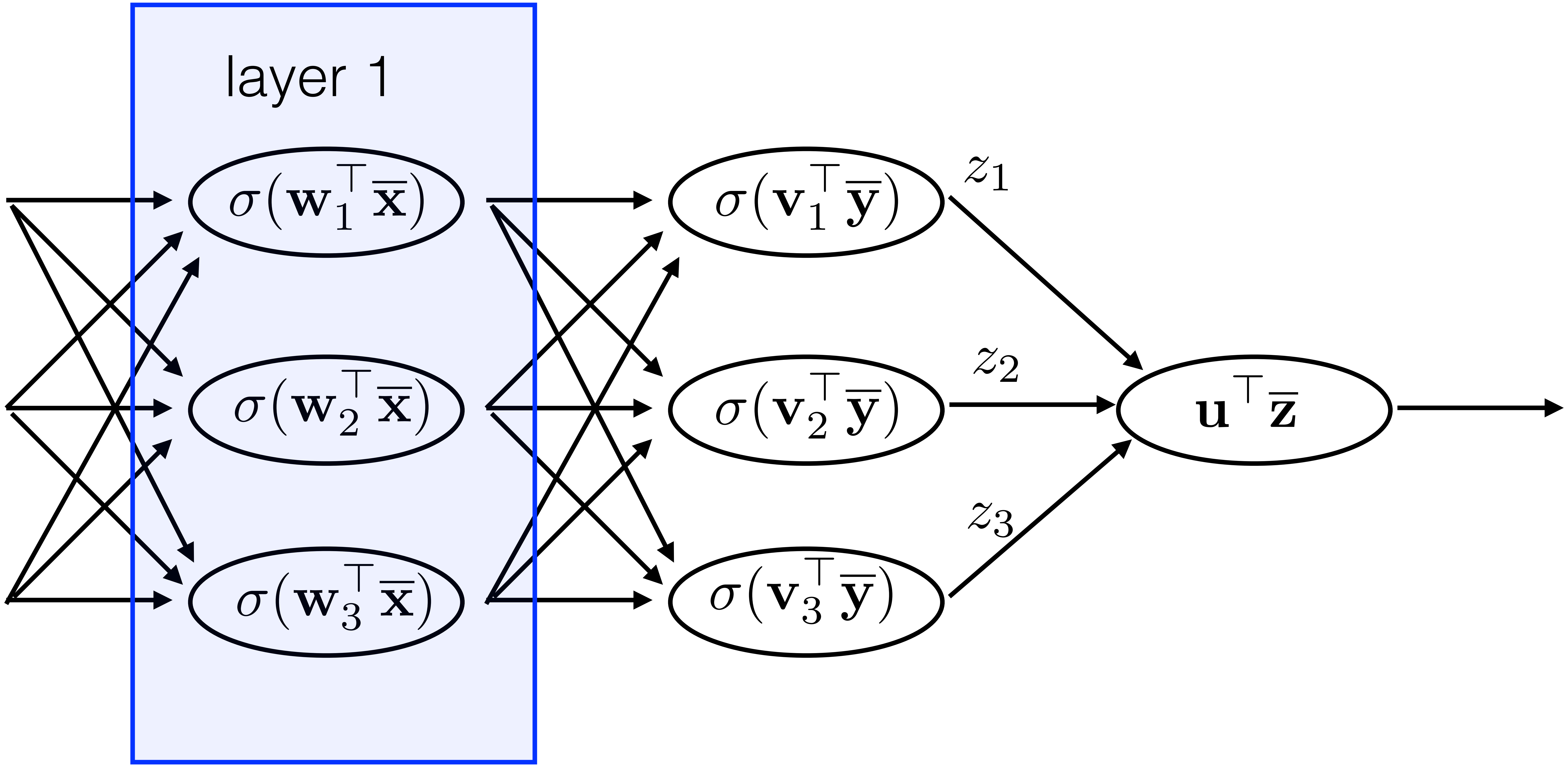
Fully-connected neural network



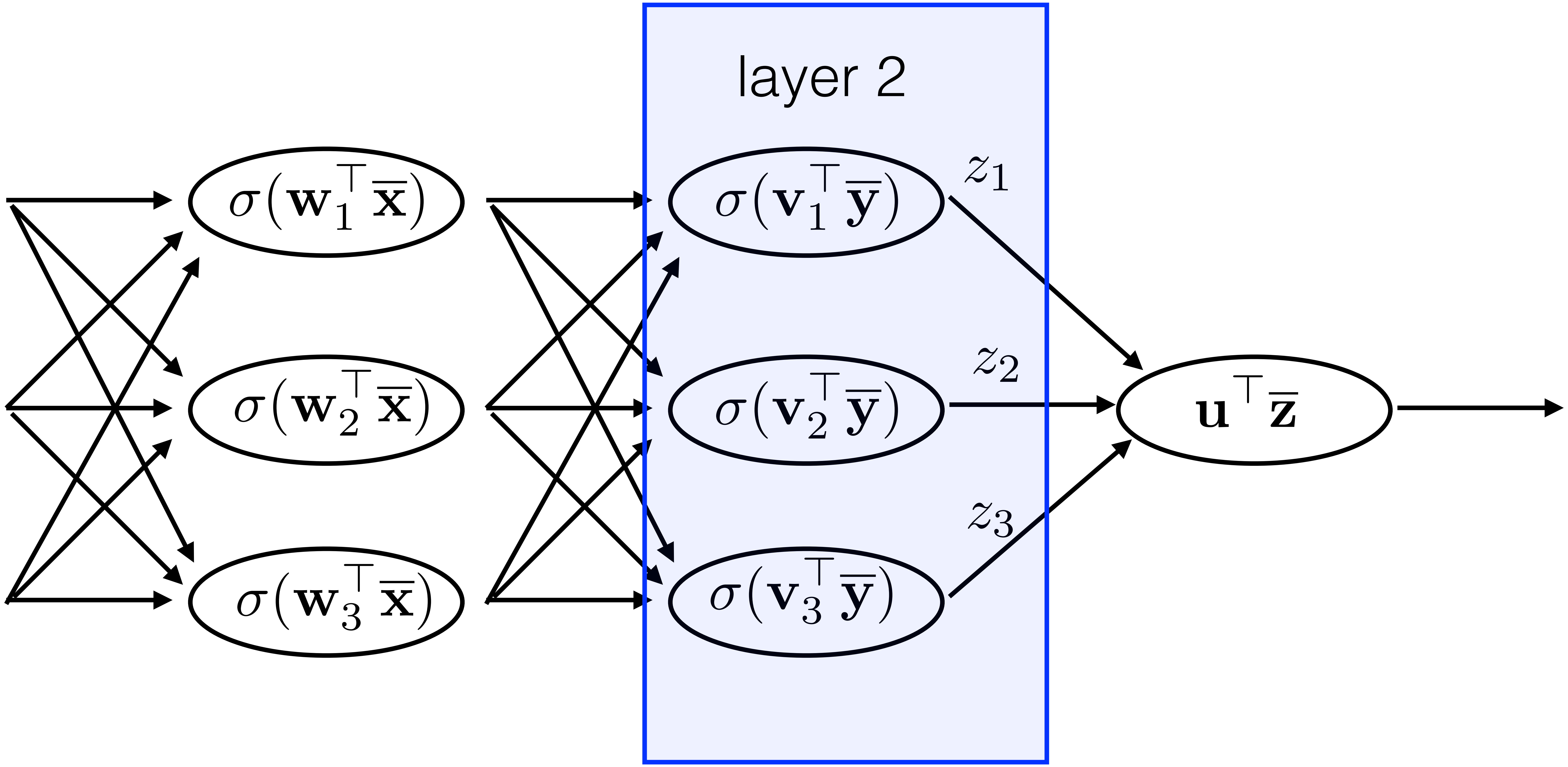
Fully-connected neural network



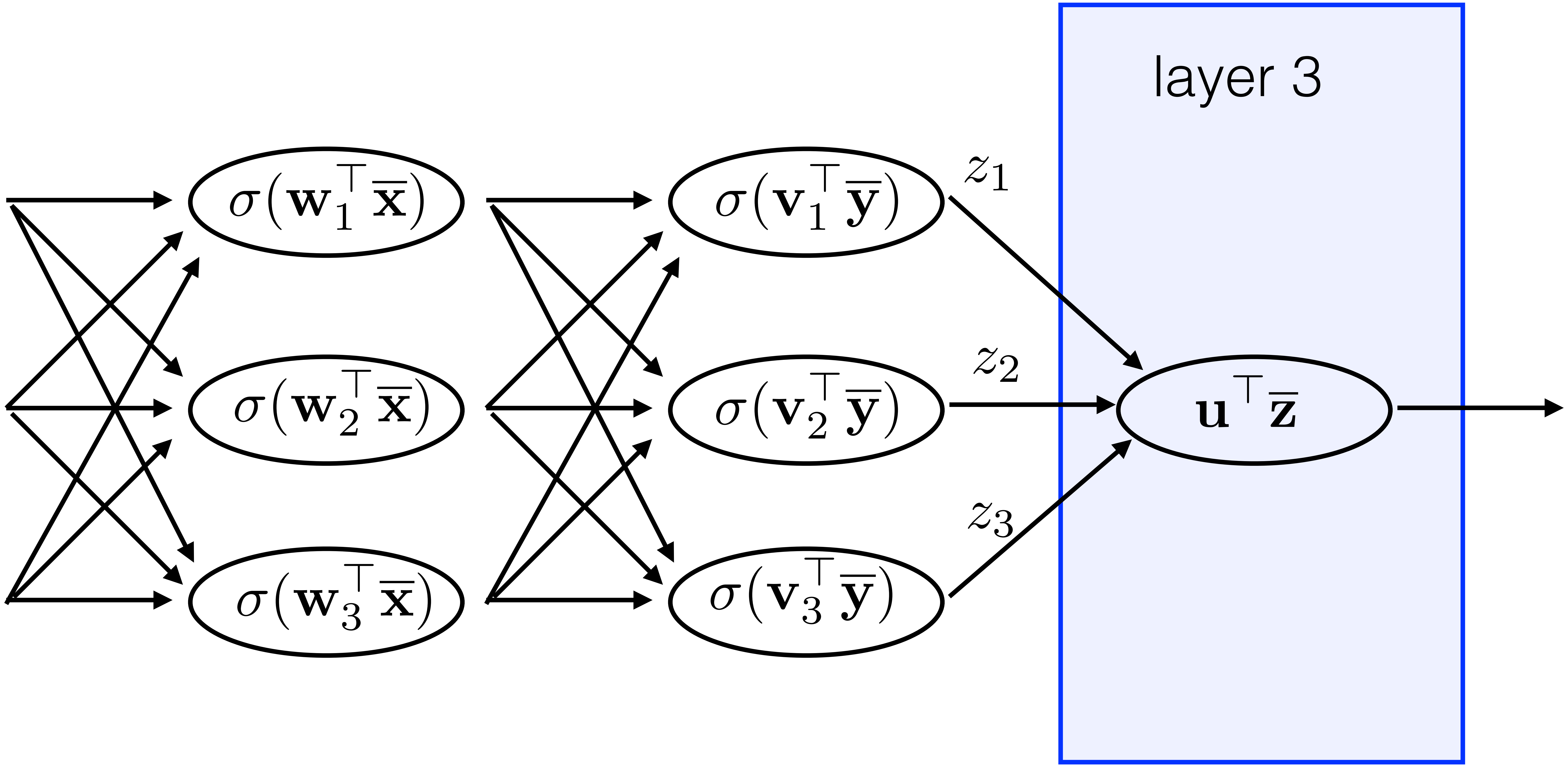
Fully-connected neural network



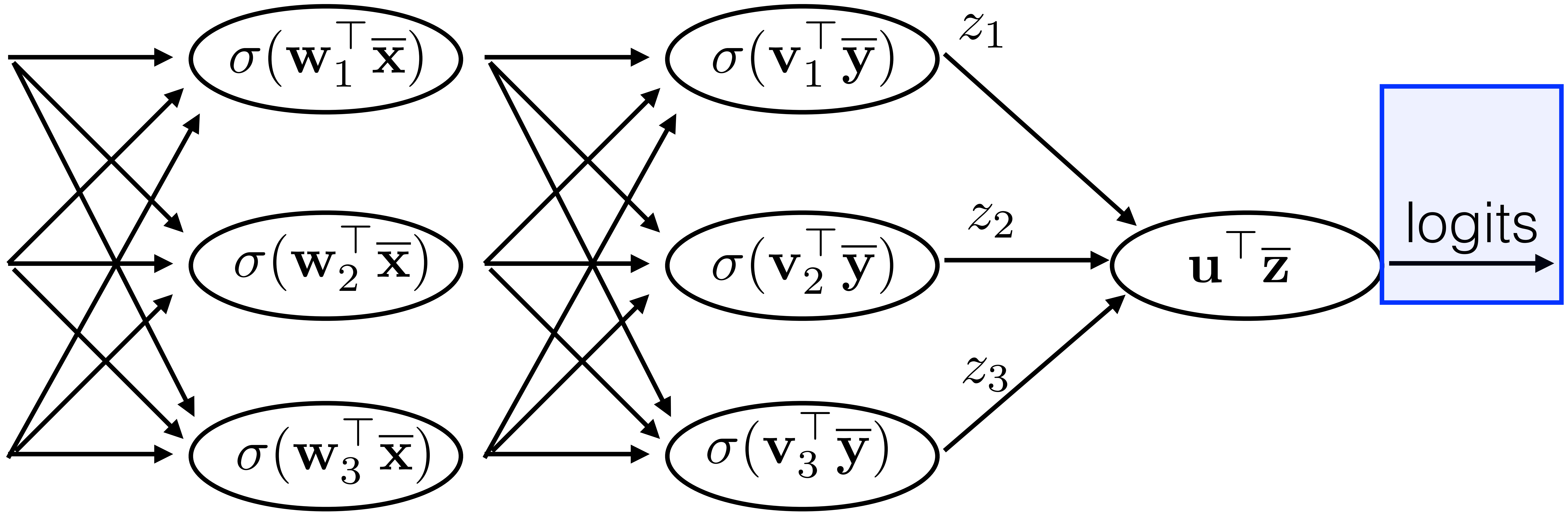
Fully-connected neural network



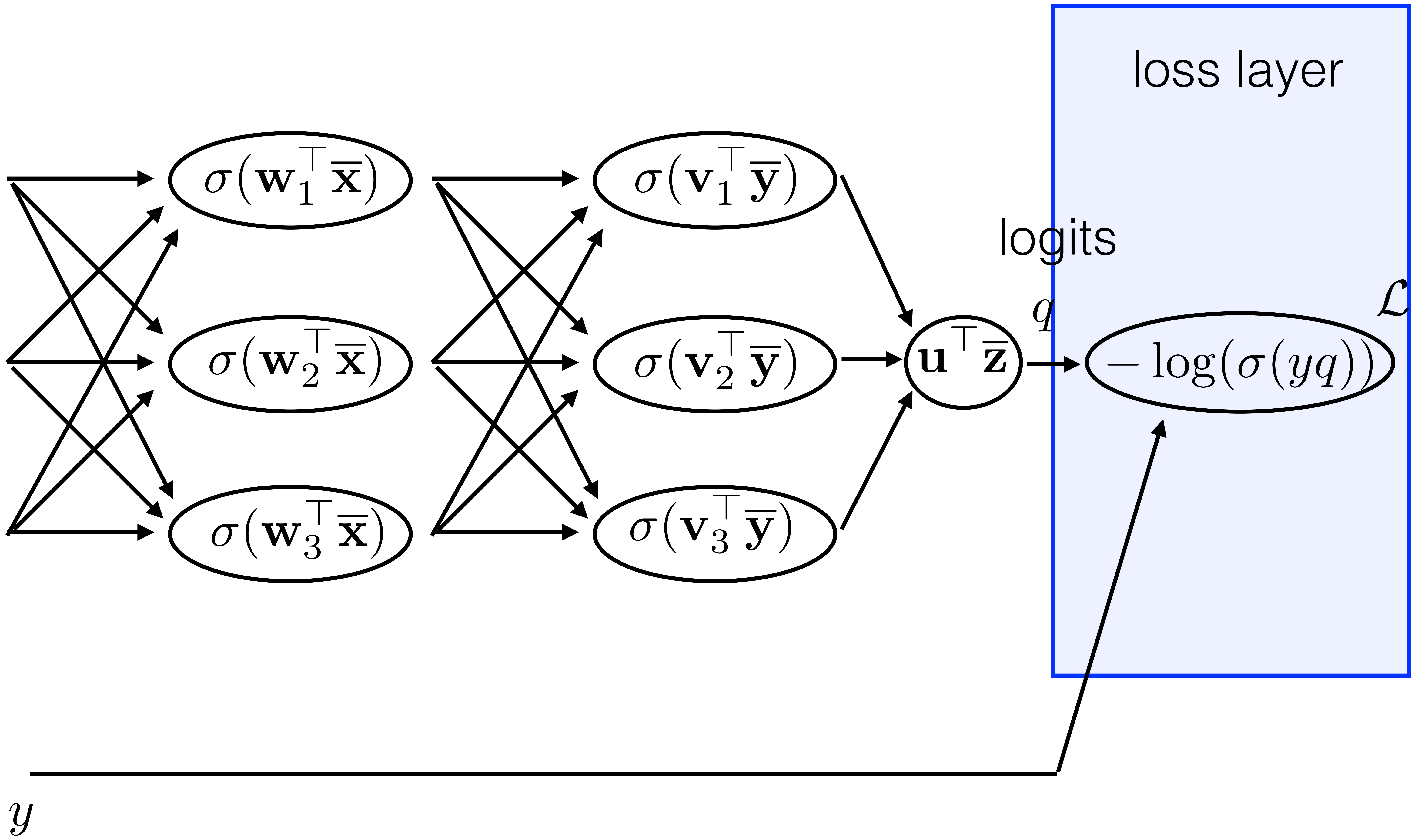
Fully-connected neural network



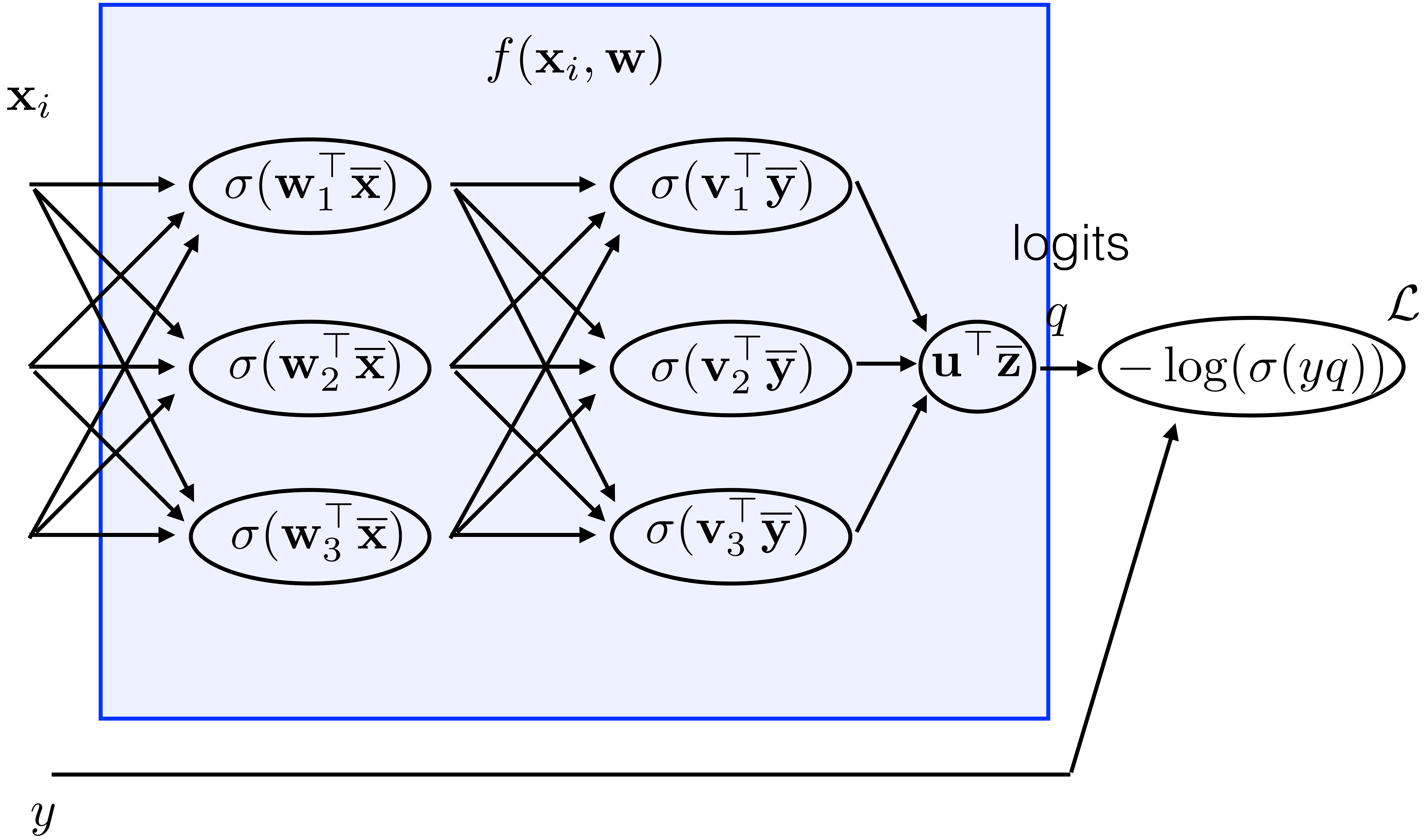
Fully-connected neural network



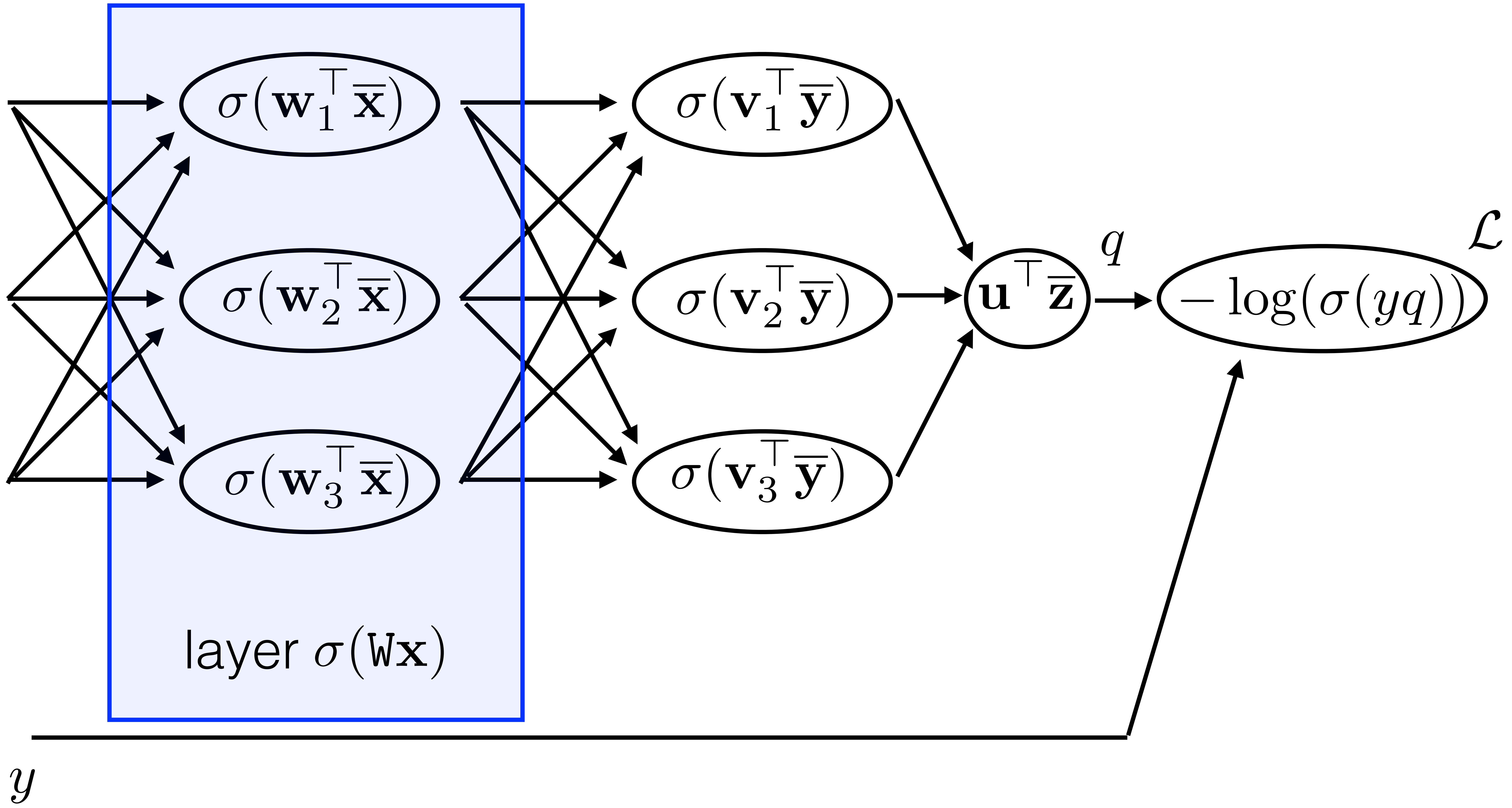
Fully-connected neural network



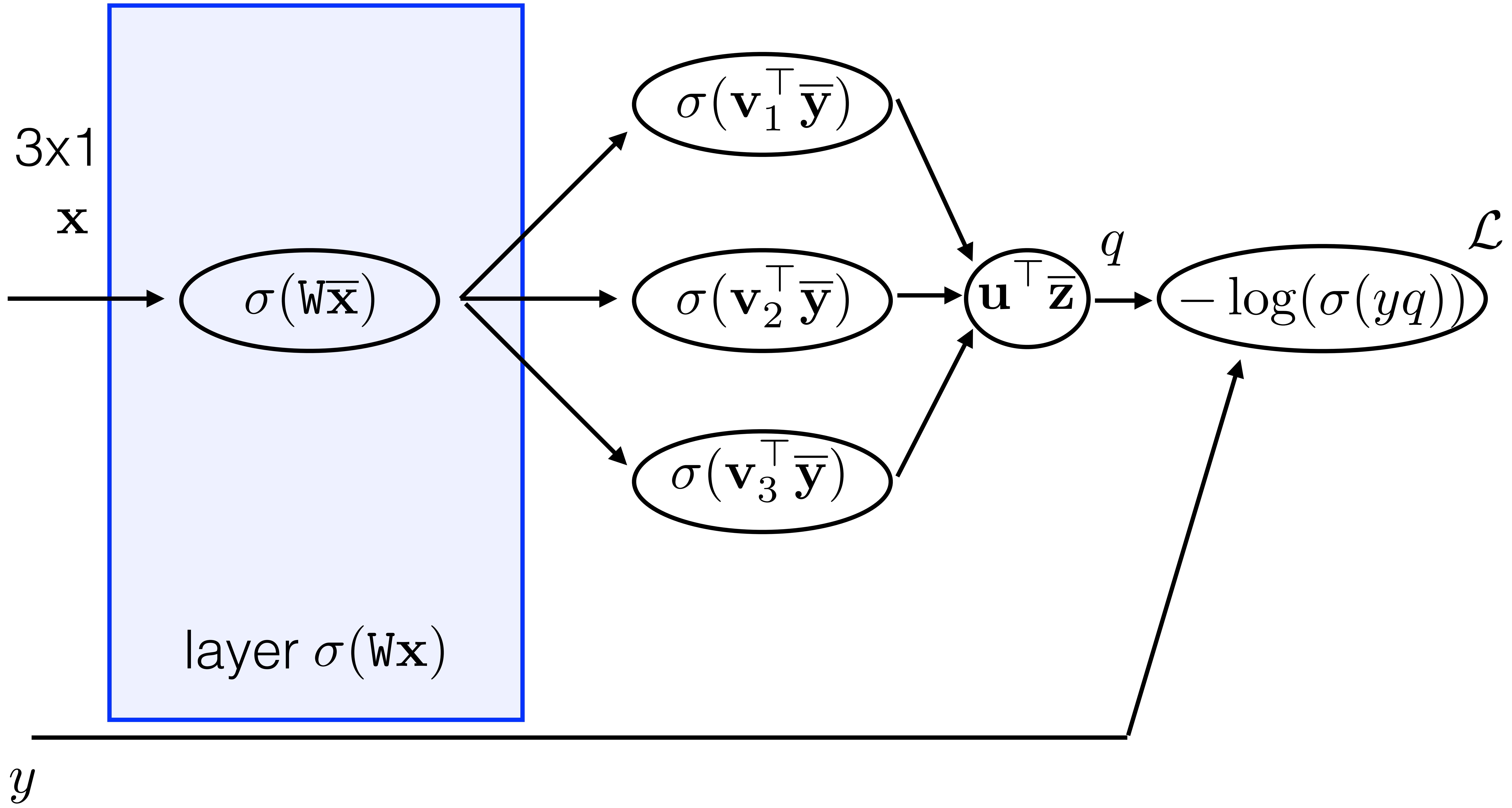
Fully-connected neural network



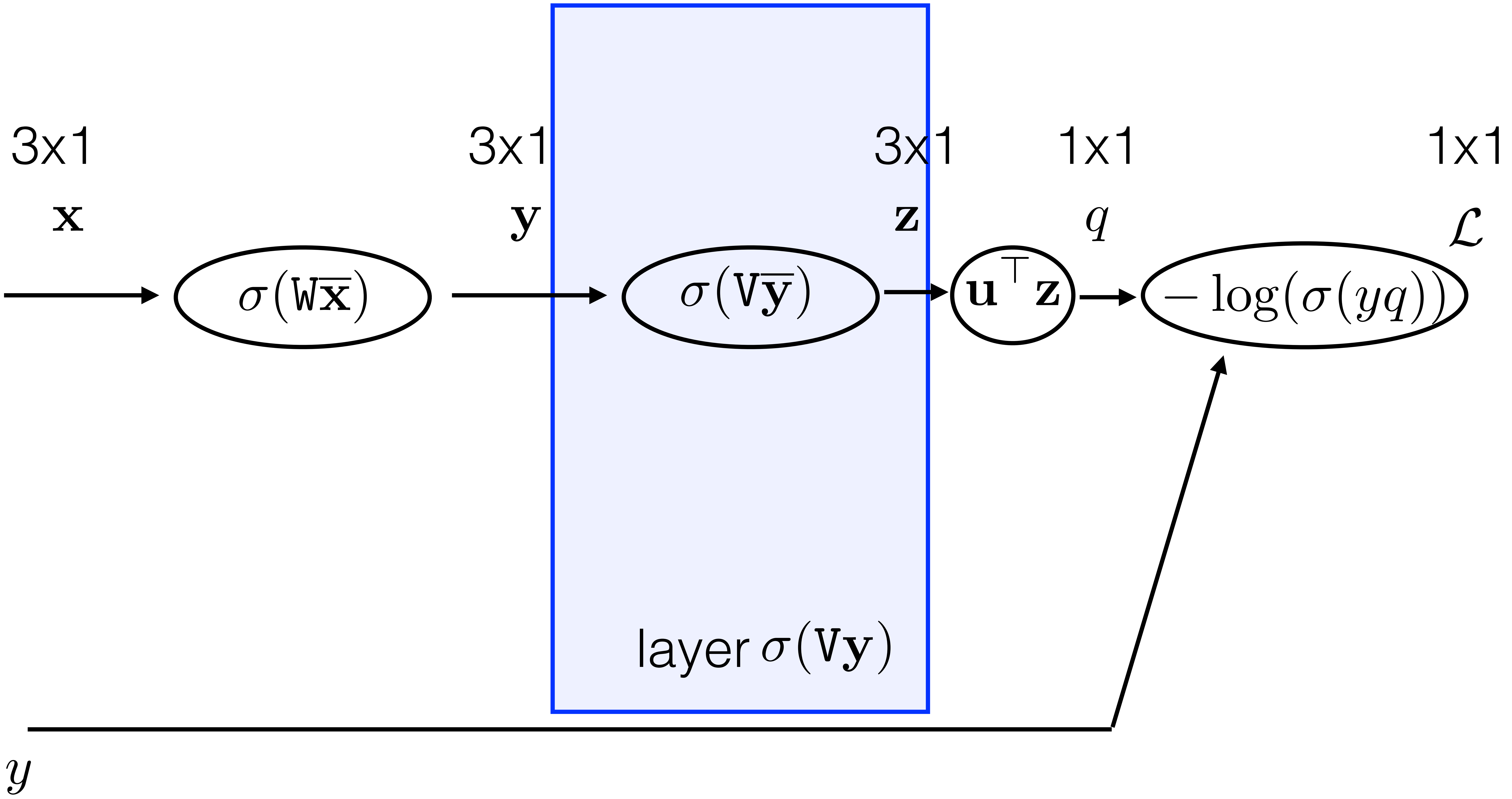
Fully-connected neural network



Fully-connected neural network

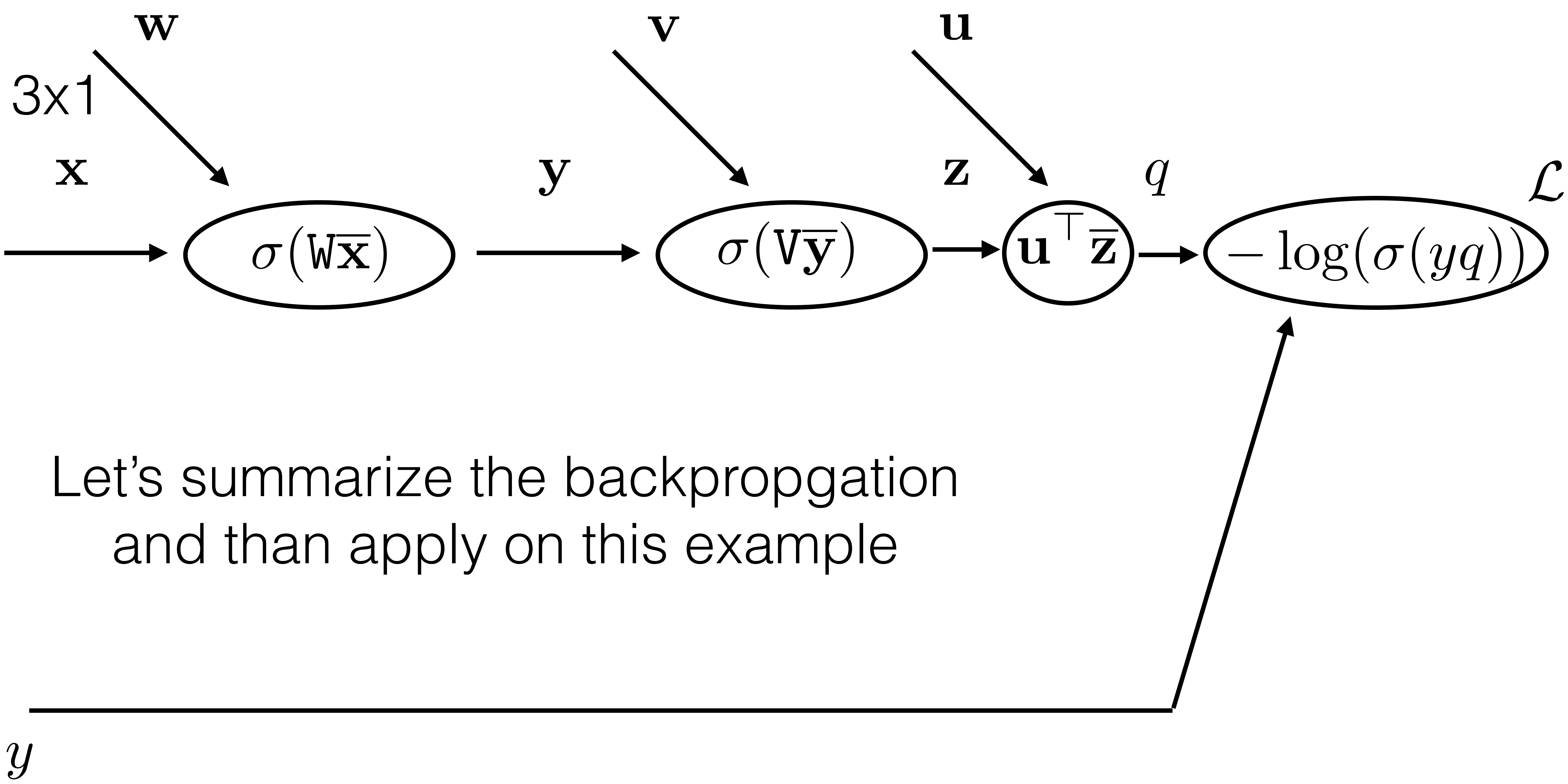


Fully-connected neural network



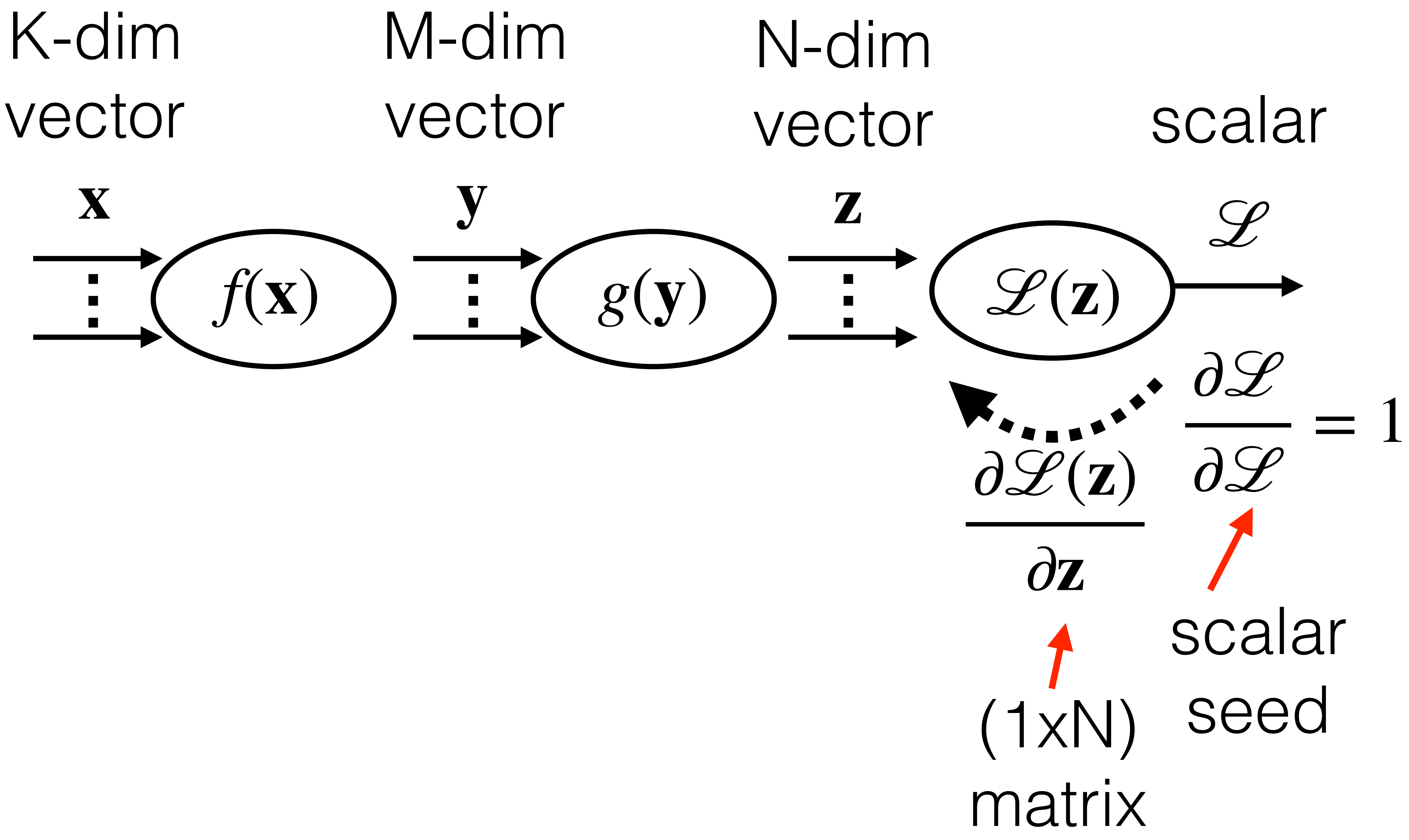
Fully-connected neural network

$\mathbf{w} = \text{vec}(\mathtt{W})$ $\mathbf{v} = \text{vec}(\mathtt{V})$



Let's summarize the backpropgation and than apply on this example

Chain-rule in computational graph and Jacobians



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

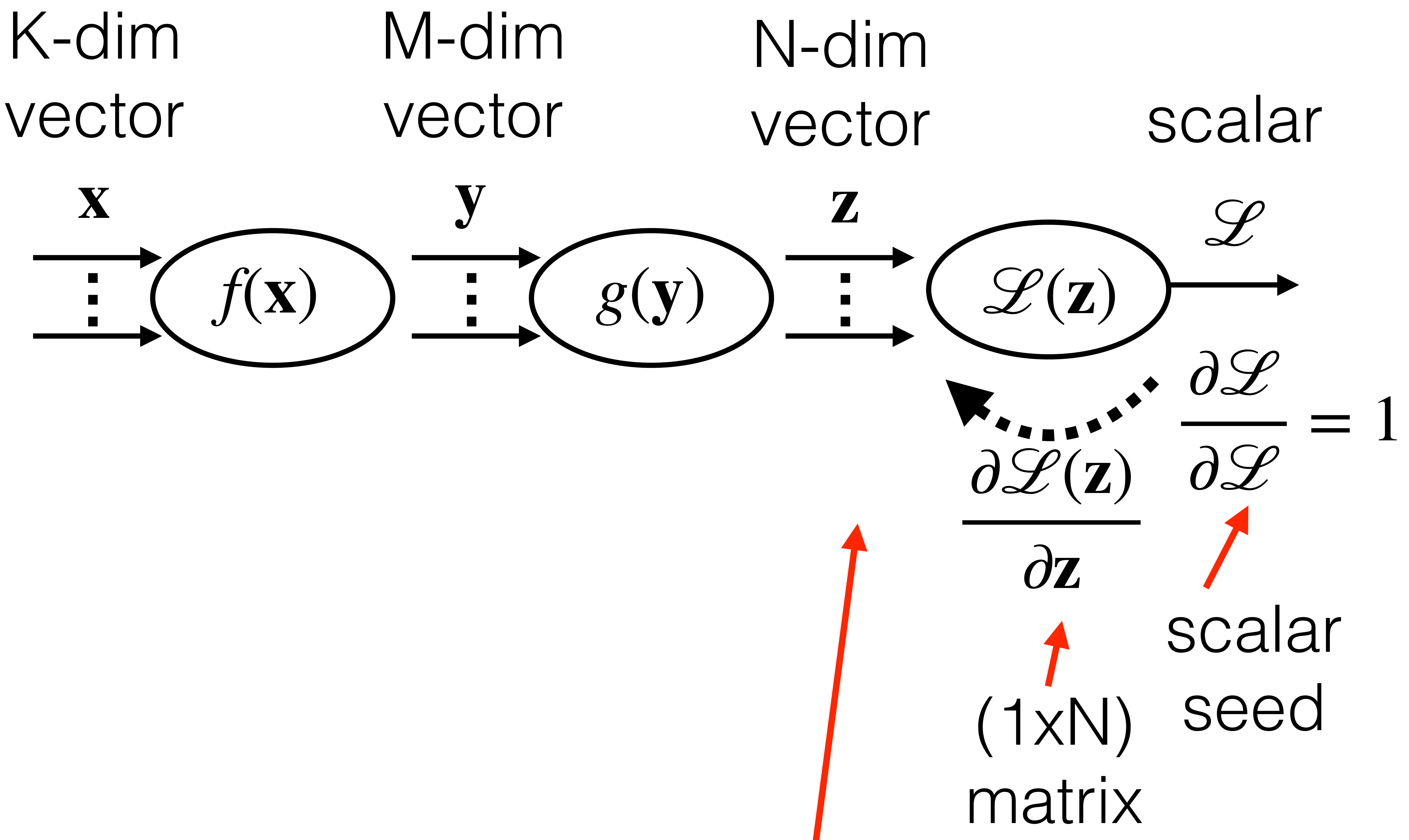
Jacobian: $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} : \mathbb{R}^K \rightarrow \mathbb{R}^{M \times K}$

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

gradient's transpose

1xK		1x1	1xN
Loss jac wrt $\mathbf{x}$ $\frac{\partial \mathscr{L}}{\partial \mathbf{x}}$	=	1	$\frac{\partial \mathscr{L}}{\partial \mathbf{z}}$

Chain-rule in computational graph and Jacobians



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

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

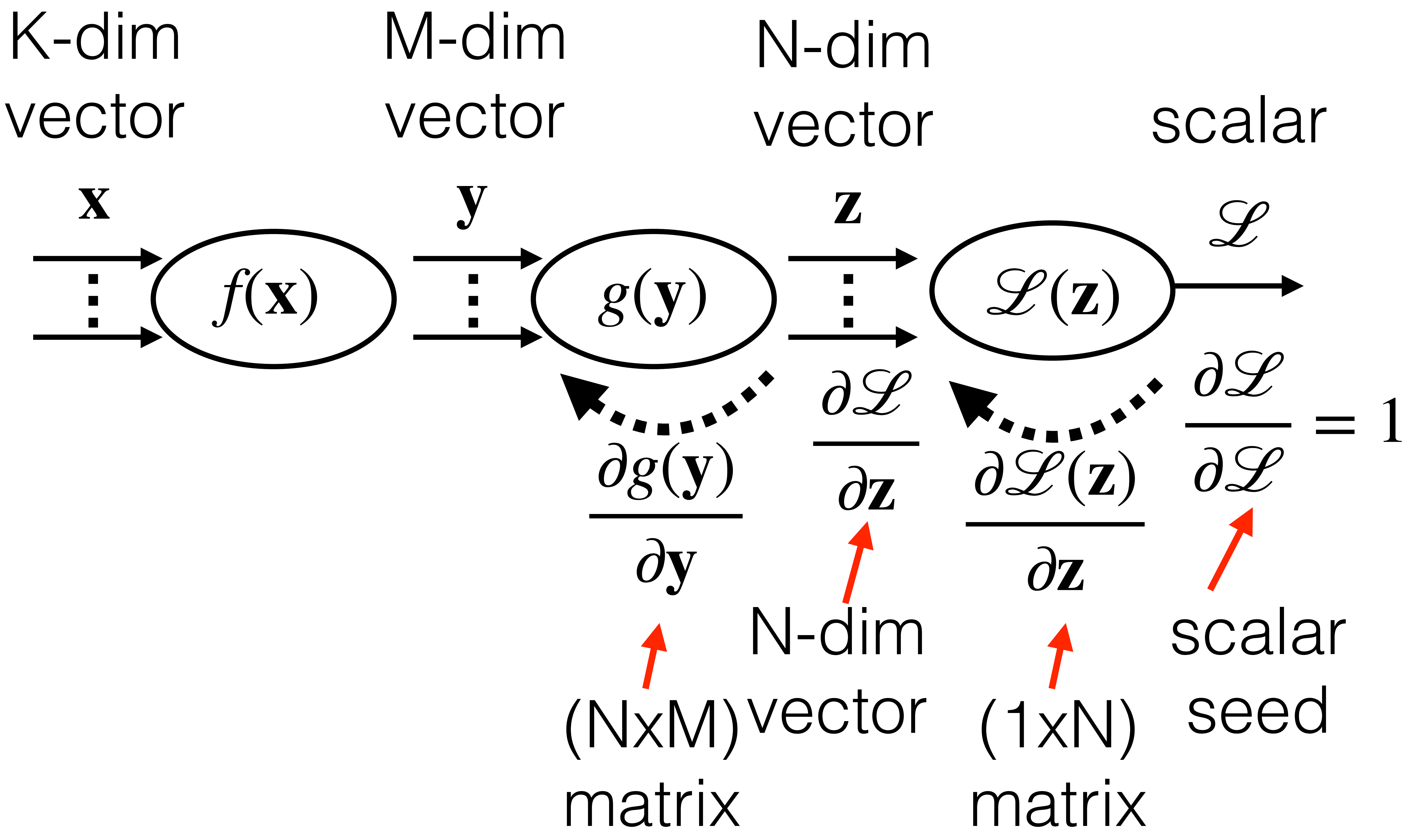
$$= \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_K} \\ \vdots & & \vdots \\ \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_K} \end{bmatrix} =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

gradient's transpose

1xK 1x1 1xN

Loss jac wrt $\mathbf{x}$ $\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = 1 \; \frac{\partial \mathscr{L}}{\partial \mathbf{z}} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}}$ N-dim vector

Chain-rule in computational graph and Jacobians



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

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

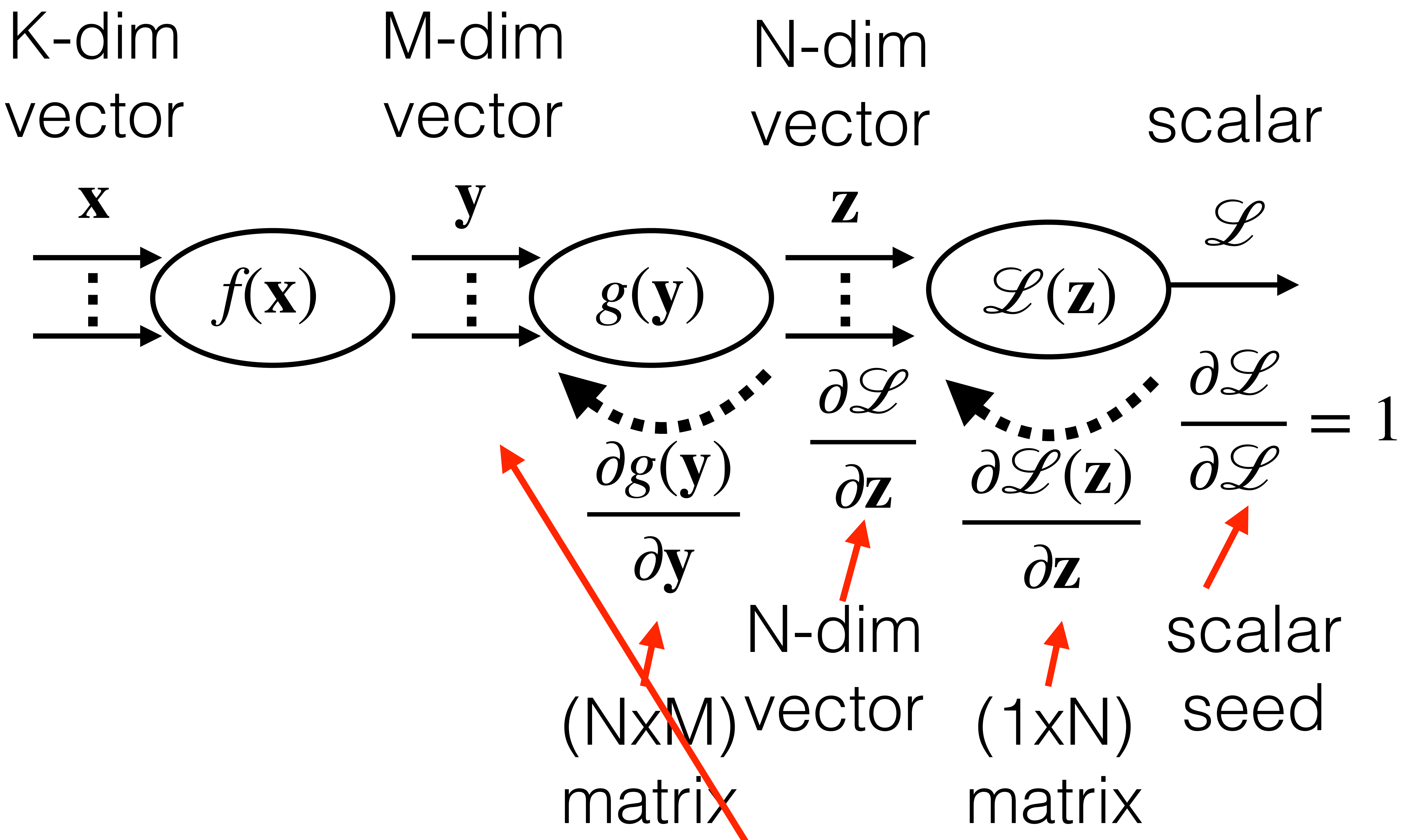
$$= \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_K} \\ \vdots & & \vdots \\ \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_K} \end{bmatrix} =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

gradient's transpose

Loss jac wrt $\mathbf{x}$

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

Chain-rule in computational graph and Jacobians



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

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

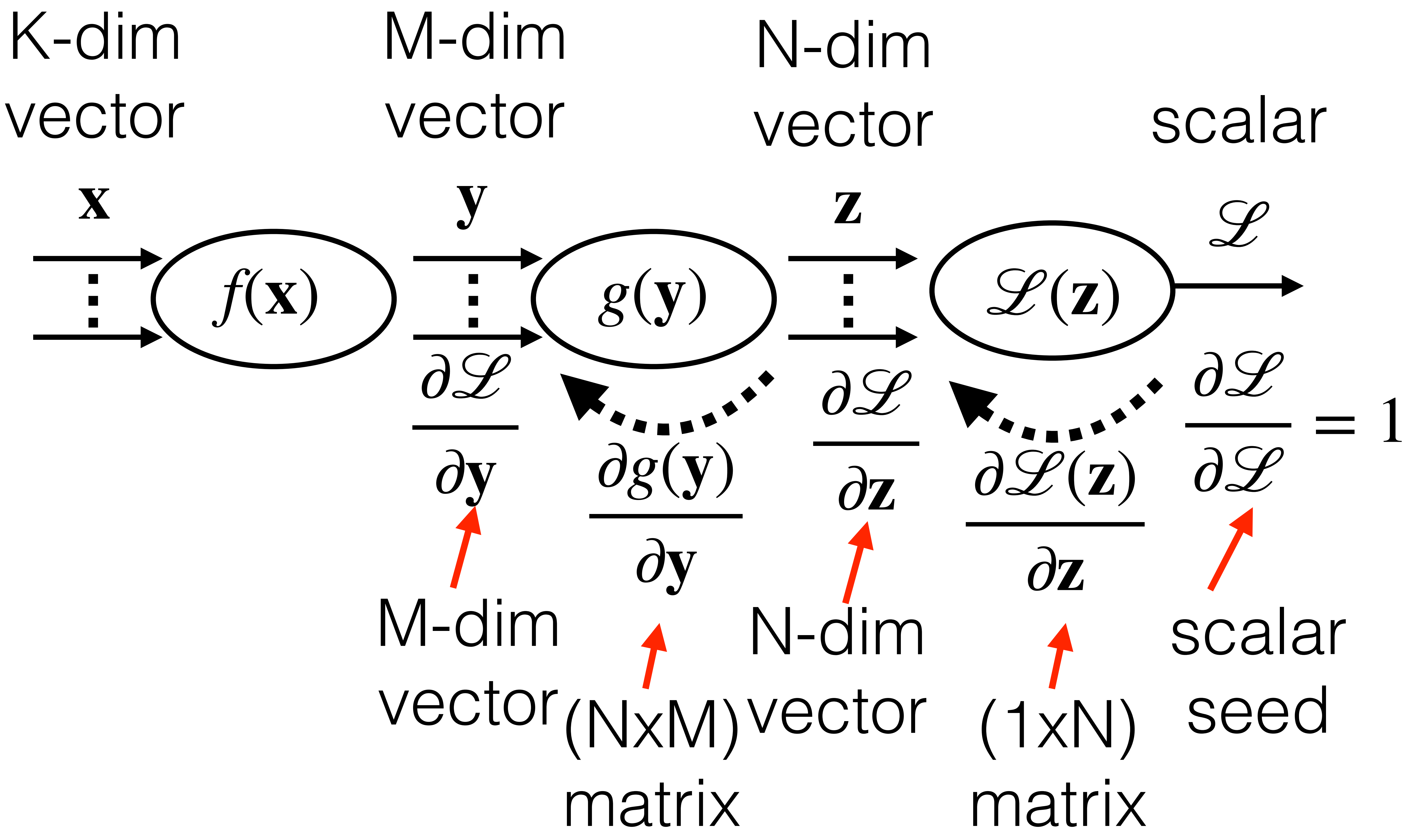
$$= \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_K} \\ \vdots & & \vdots \\ \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_K} \end{bmatrix} =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

gradient's transpose

1xK 1x1 1xN NxM

Loss jac wrt $\mathbf{x}$ $\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = 1 \; \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \; \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} = \frac{\partial \mathscr{L}}{\partial \mathbf{y}}$ M-dim vector

Chain-rule in computational graph and Jacobians



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

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

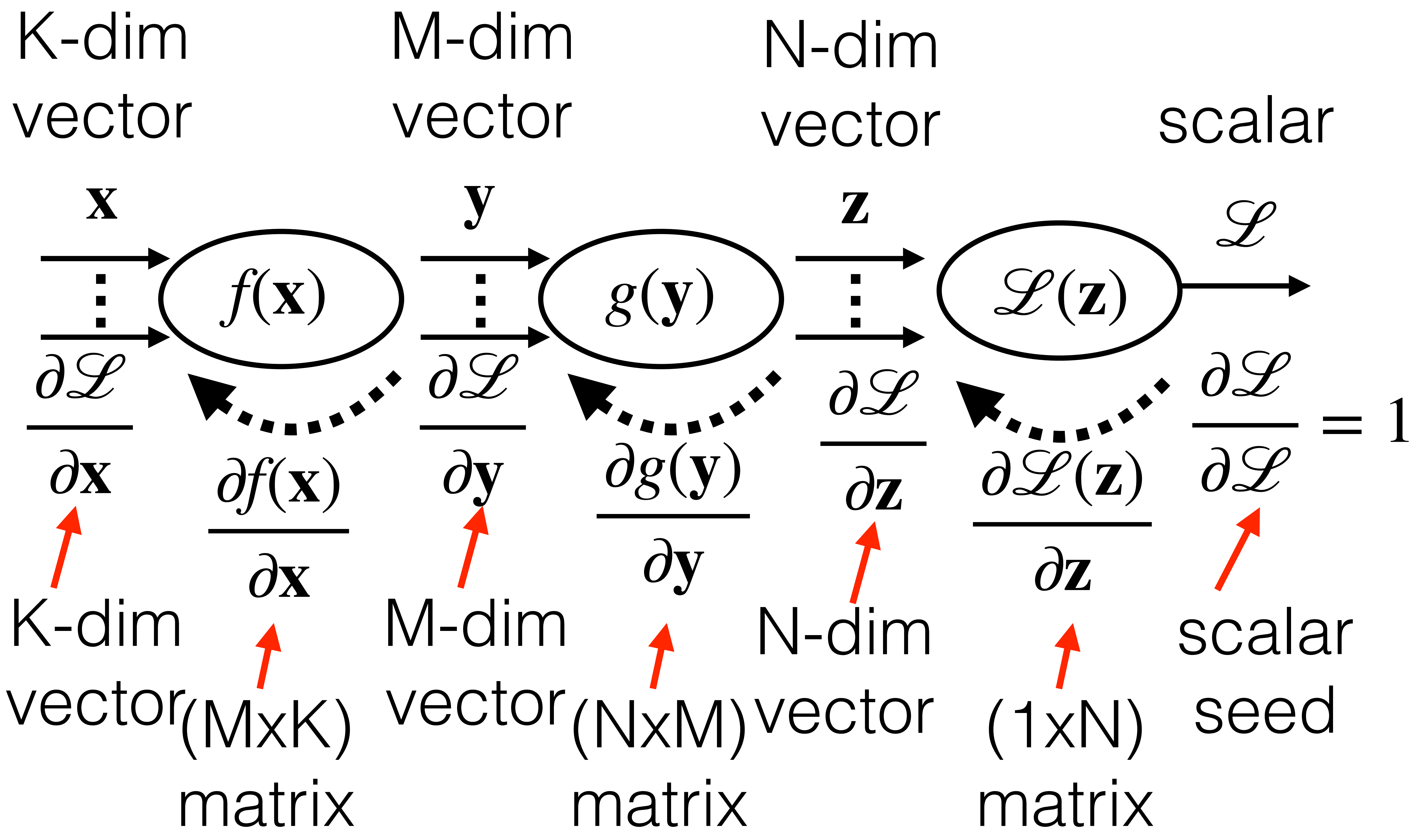
$$= \begin{bmatrix} \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}_K} \\ \vdots & & \vdots \\ \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_1} & \cdots & \frac{\partial f_M(\mathbf{x})}{\partial \mathbf{x}_K} \end{bmatrix} =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

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

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

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

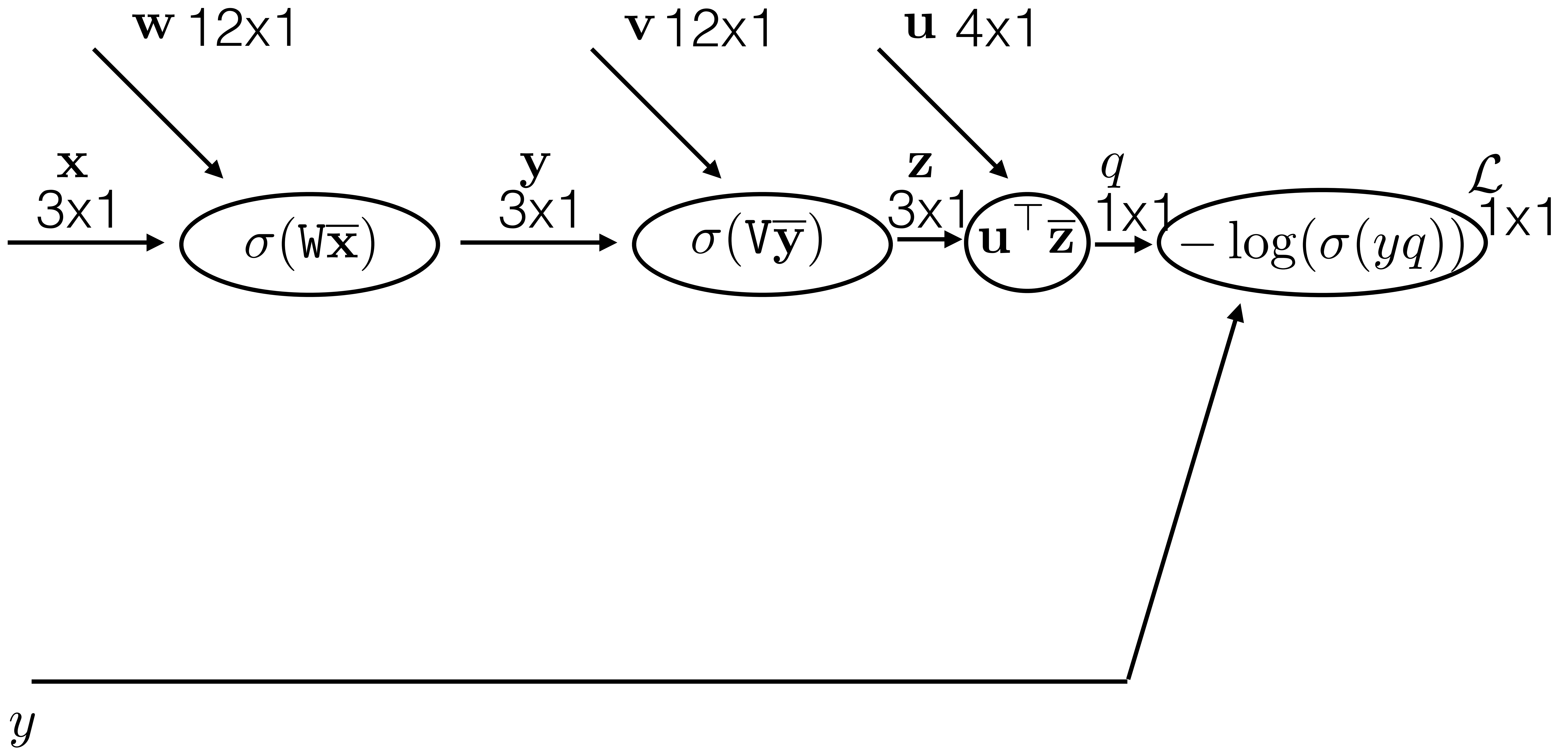
gradient's transpose

Loss jac wrt **x**

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

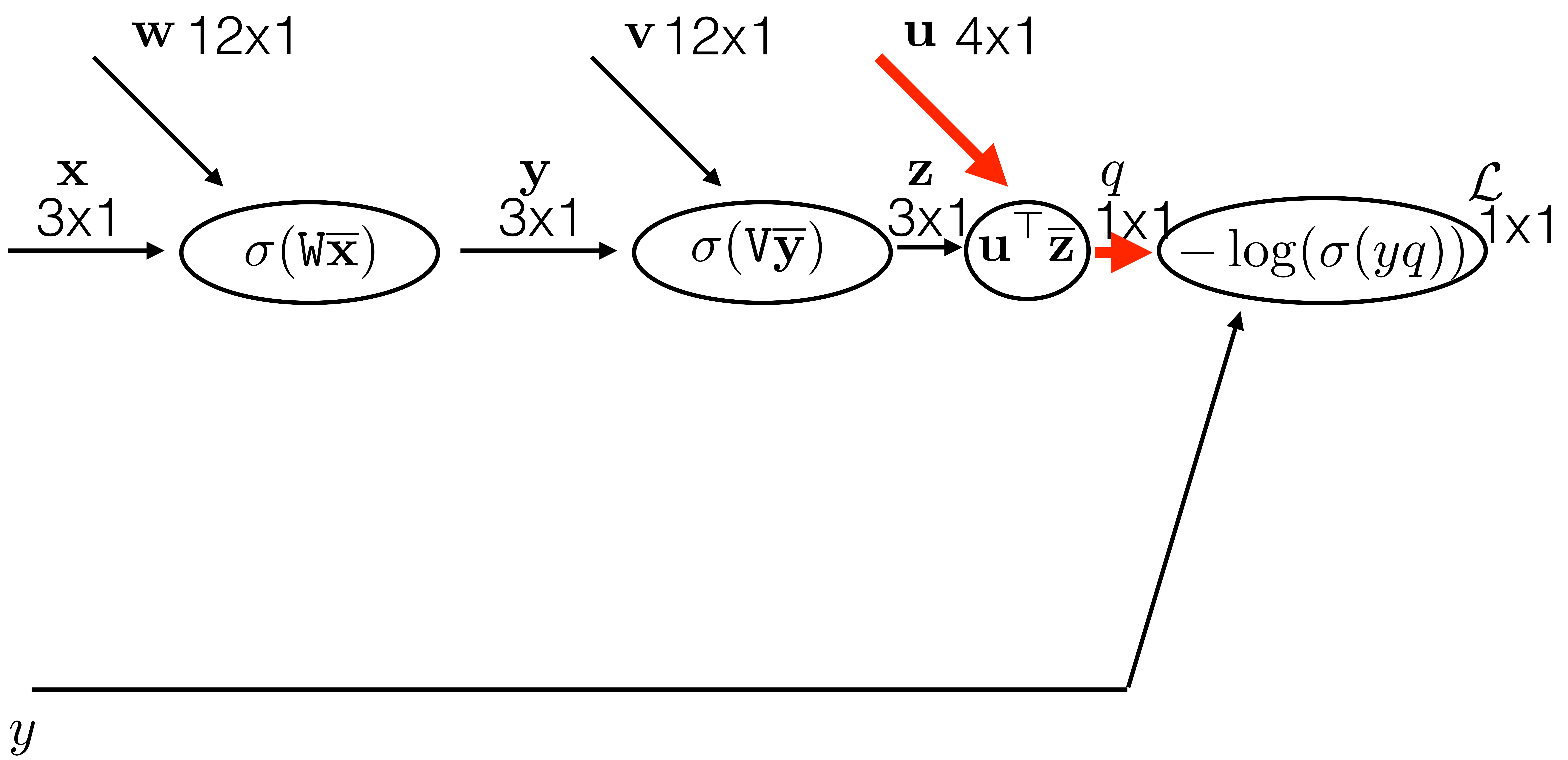
Fully-connected neural network

$\mathbf{w} = \text{vec}(\mathtt{W})$ $\mathbf{v} = \text{vec}(\mathtt{V})$ What is the $\mathbf{w}$-weight dimensionality?



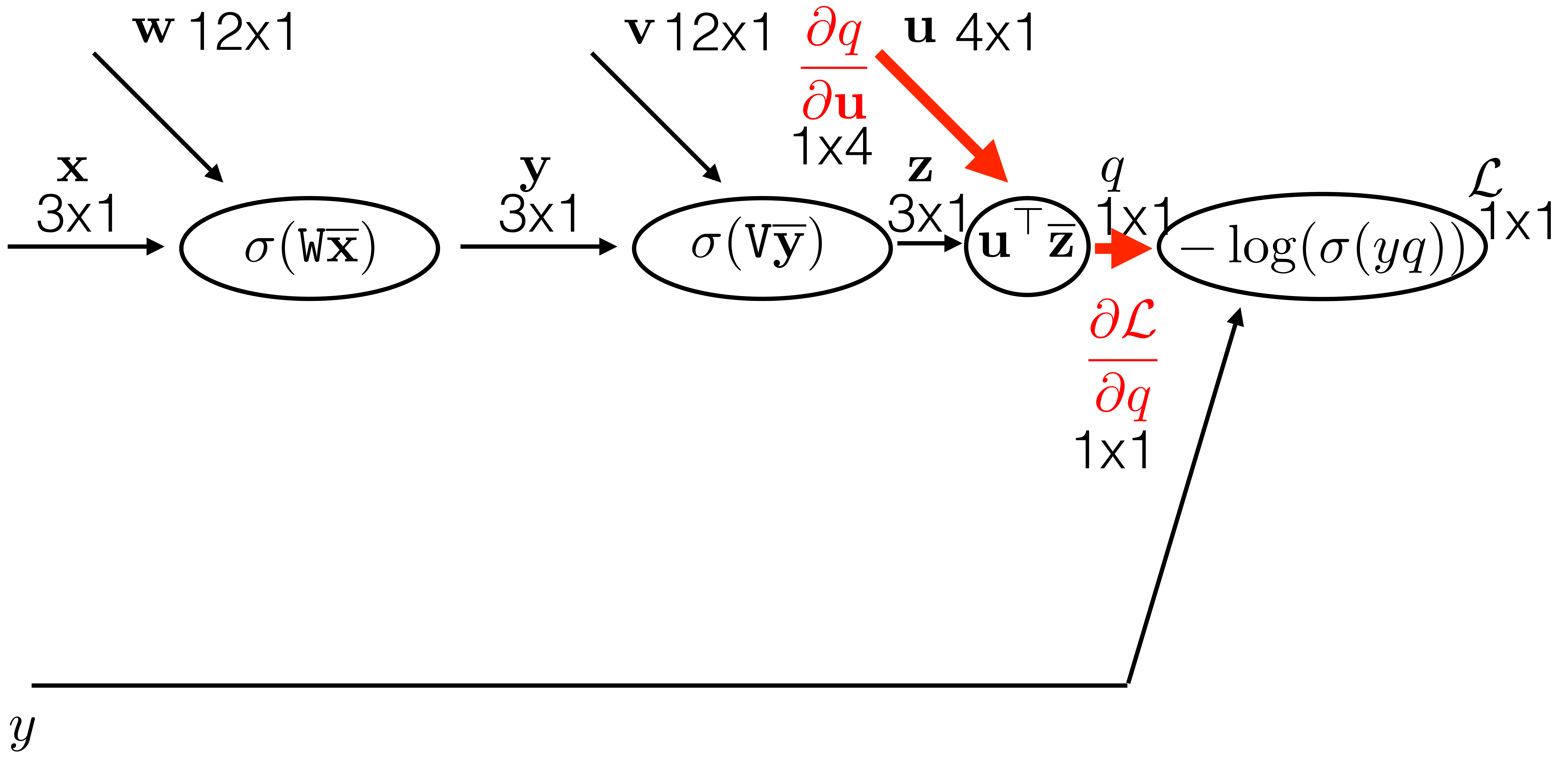
Chainrule in fully-connected neural network

Jacobian wrt $\mathbf{u}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{u}} = ?$

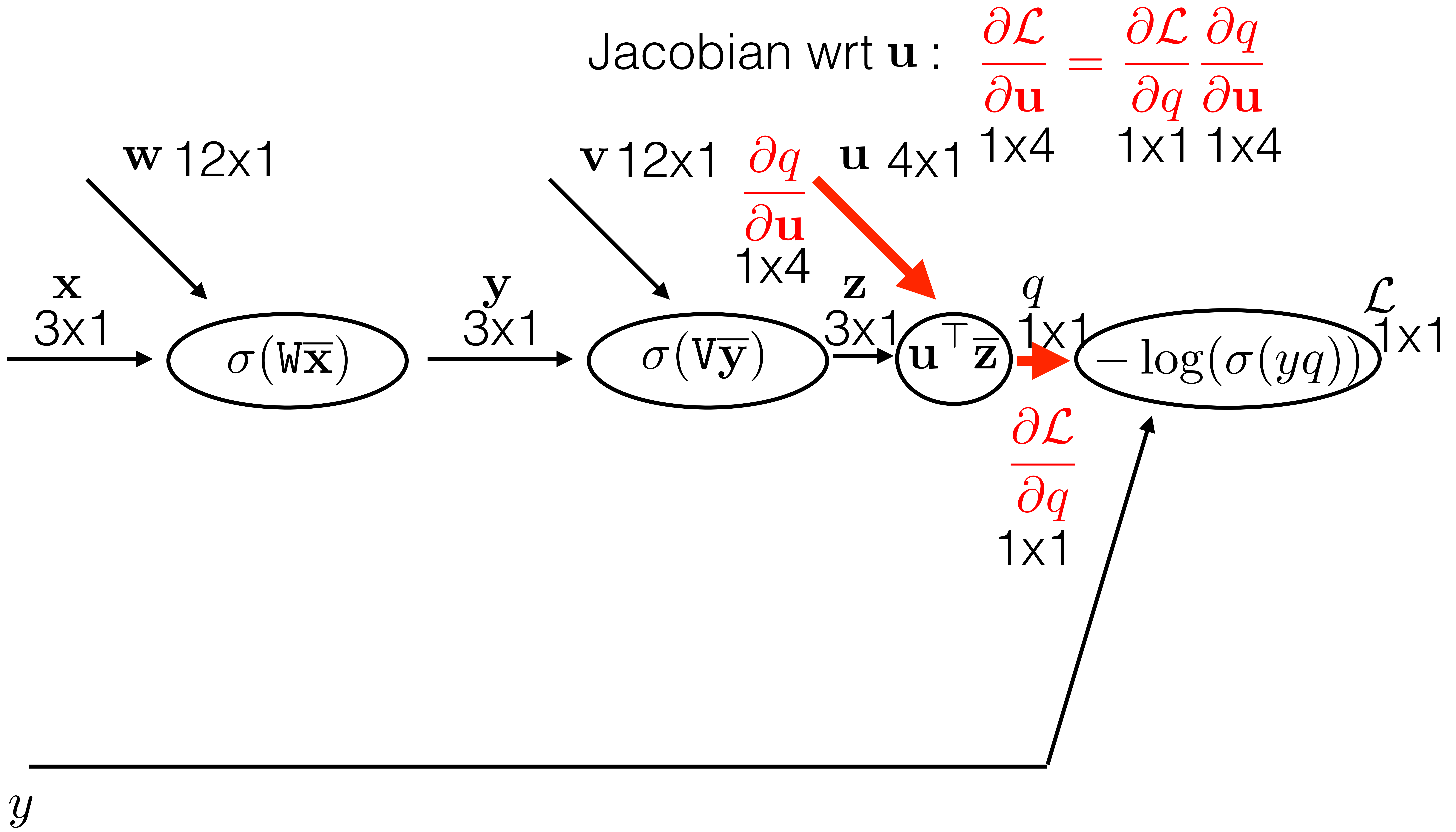


Chainrule in fully-connected neural network

Jacobian wrt $\mathbf{u}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{u}} = \frac{\partial \mathcal{L}}{\partial q} \frac{\partial q}{\partial \mathbf{u}}$

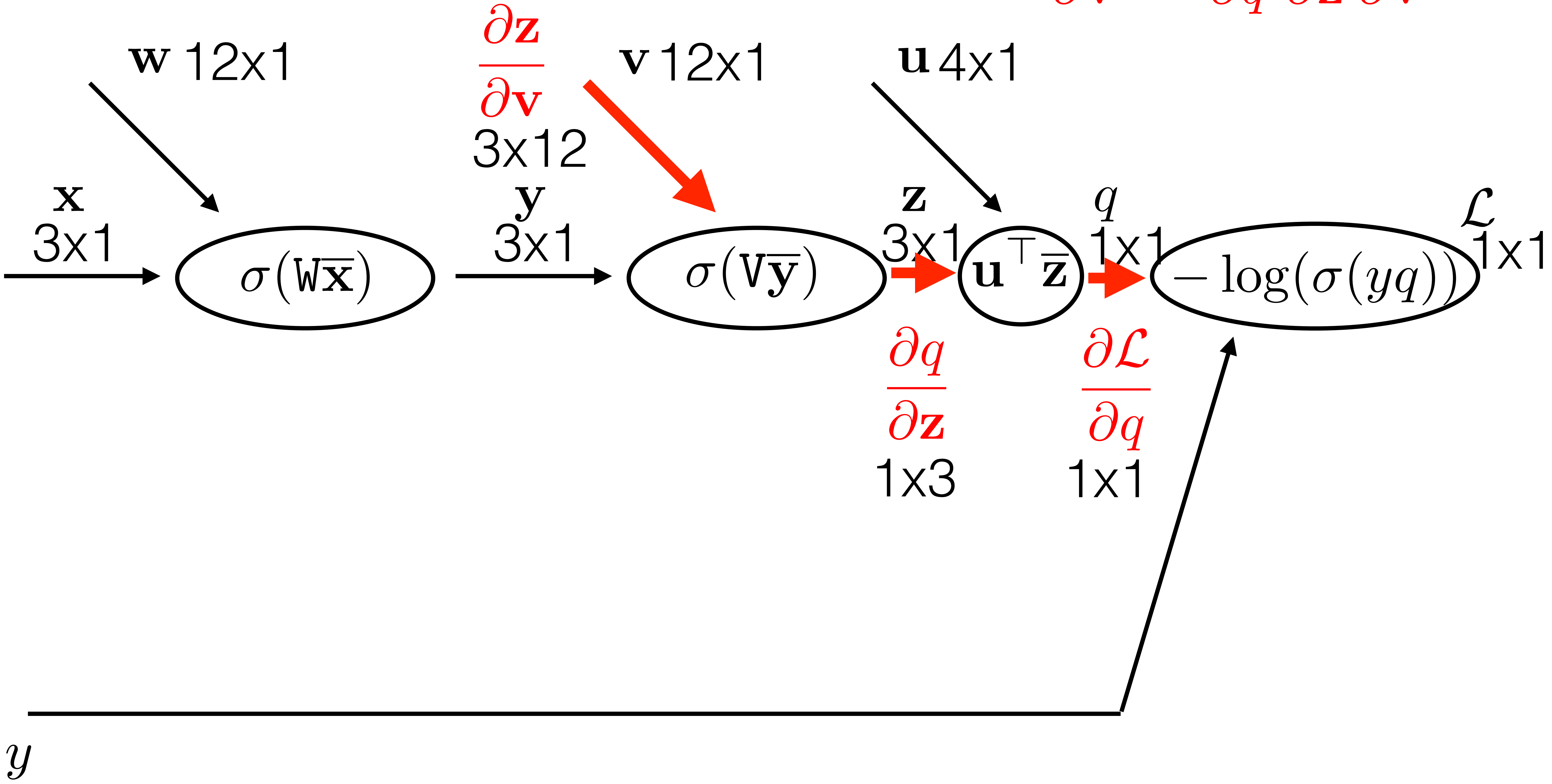


Chainrule in fully-connected neural network



Chainrule in fully-connected neural network

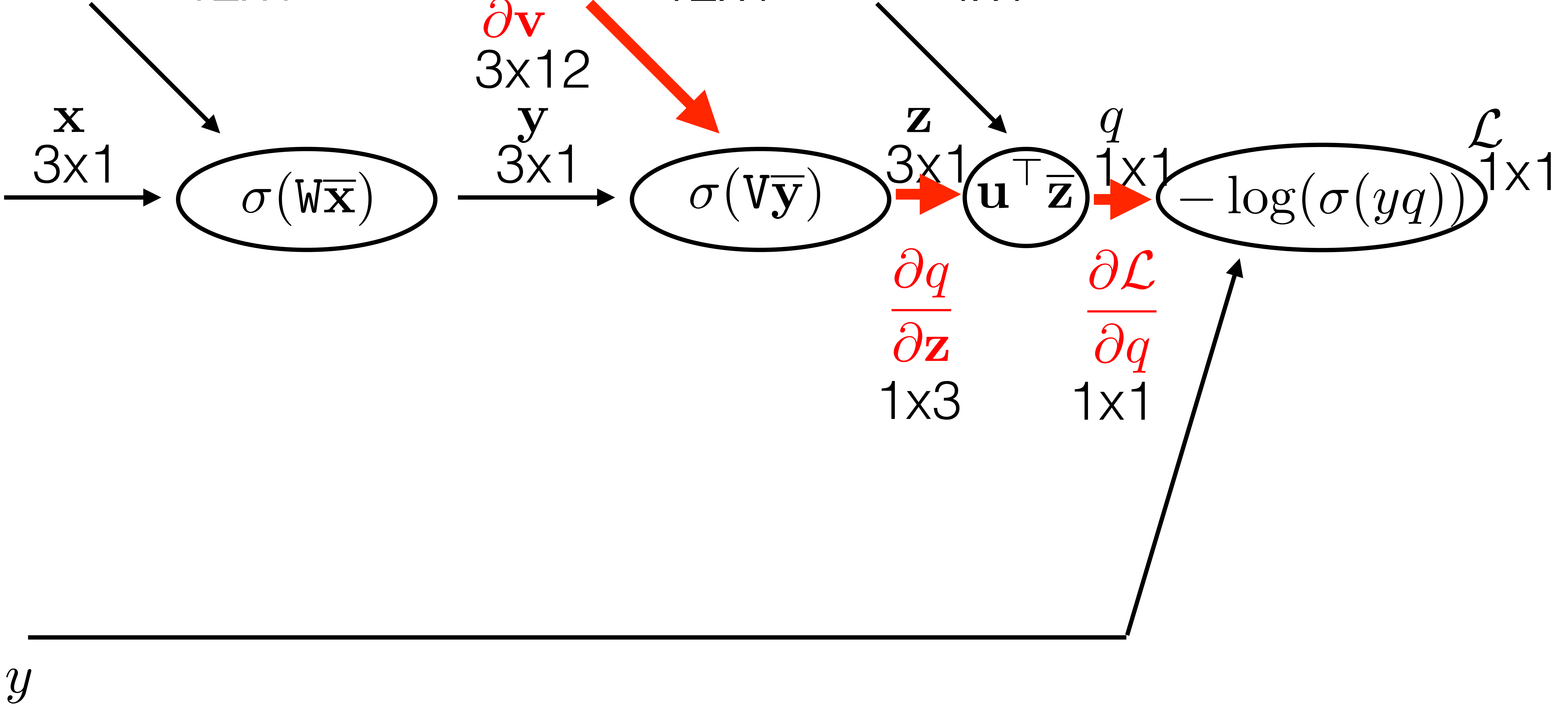
Jacobian wrt $\mathbf{u}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{v}} = \frac{\partial \mathcal{L}}{\partial q} \frac{\partial q}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{v}}$



Chainrule in fully-connected neural network

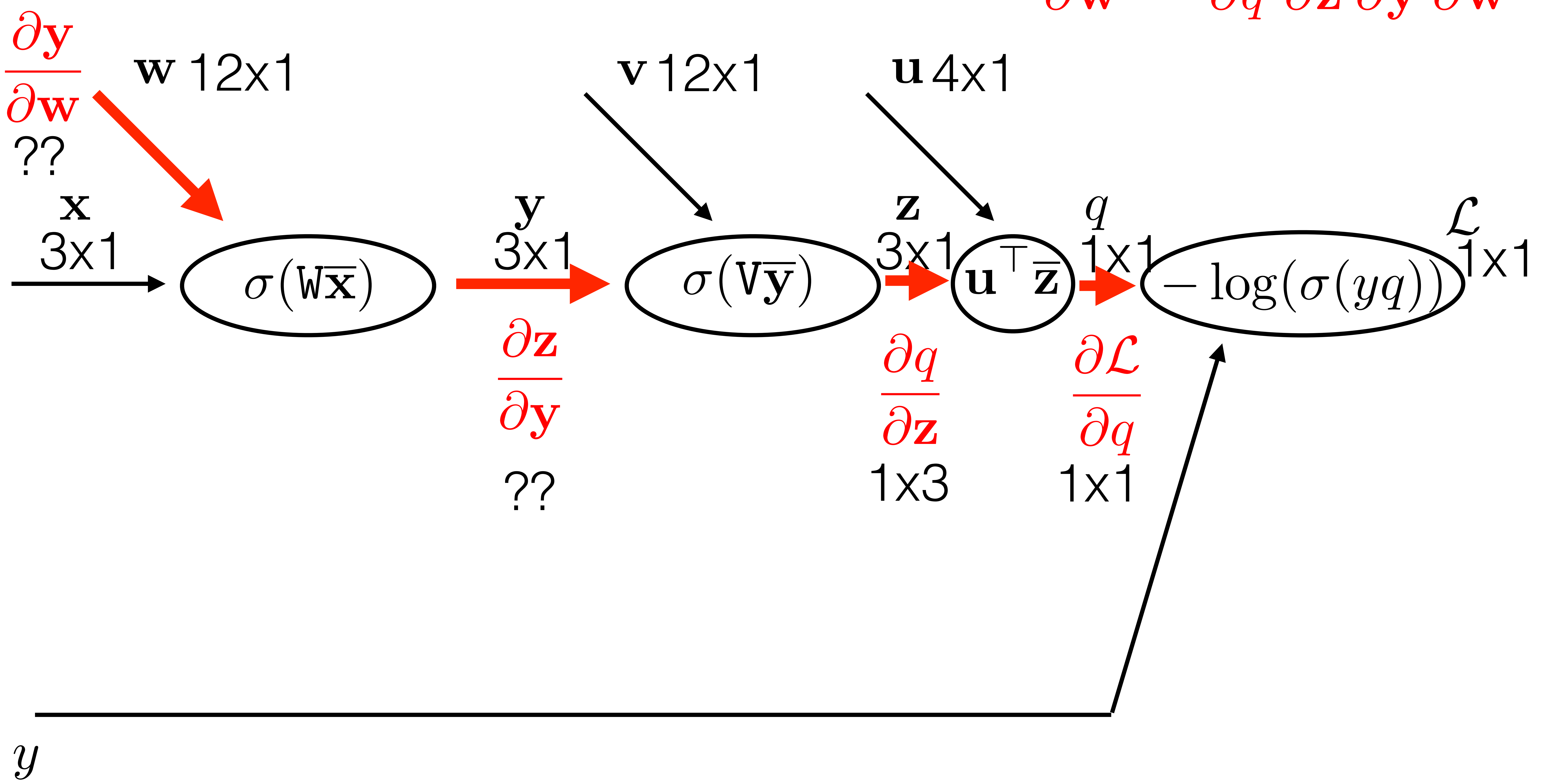
Jacobian wrt $\mathbf{u}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{v}} = \frac{\partial \mathcal{L}}{\partial q} \frac{\partial q}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{v}}$

1x12 1x1 1x3 3x12



Chainrule in fully-connected neural network

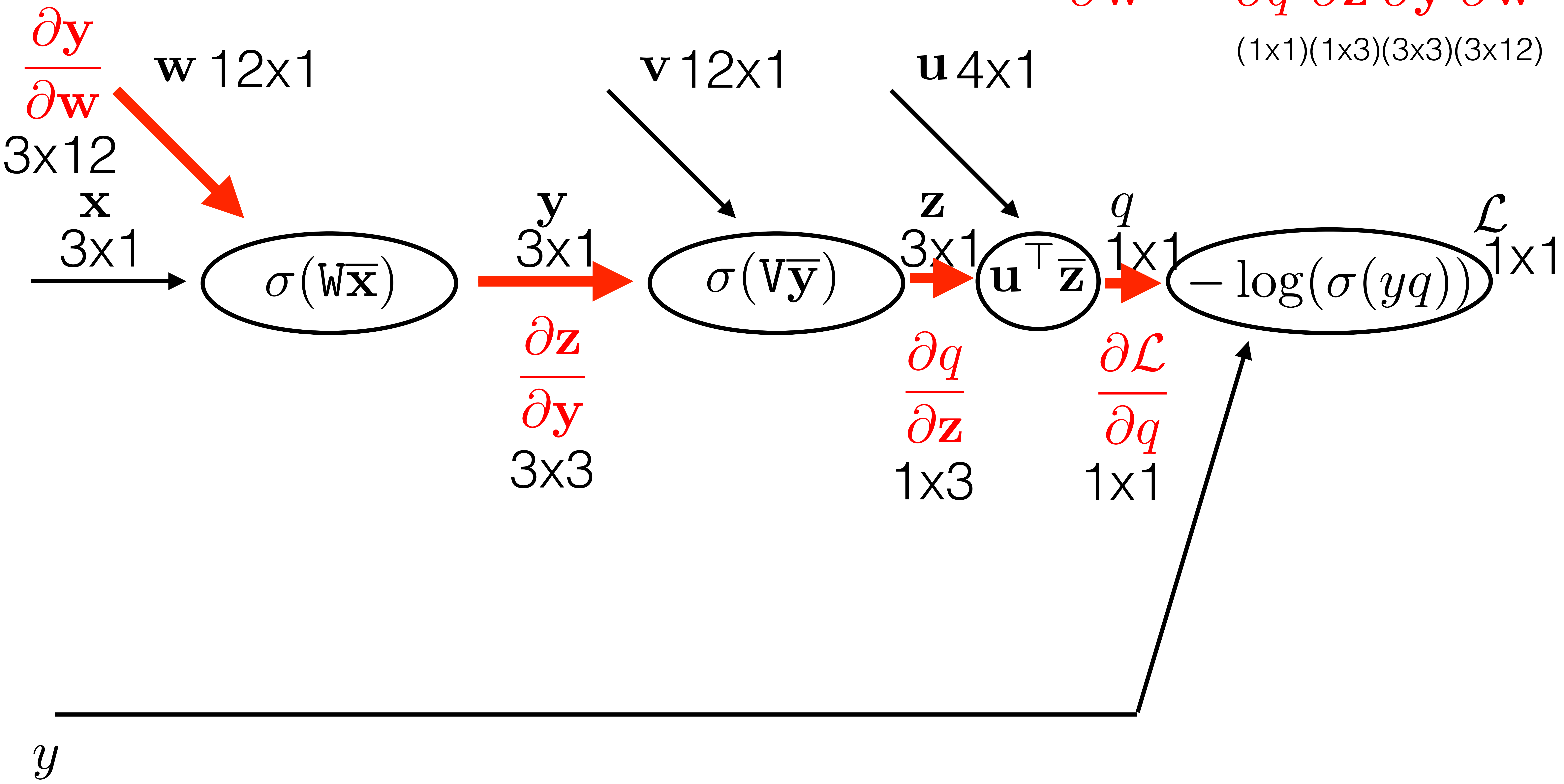
Jacobian wrt $\mathbf{u}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial q} \frac{\partial q}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{w}}$



Chainrule in fully-connected neural network

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

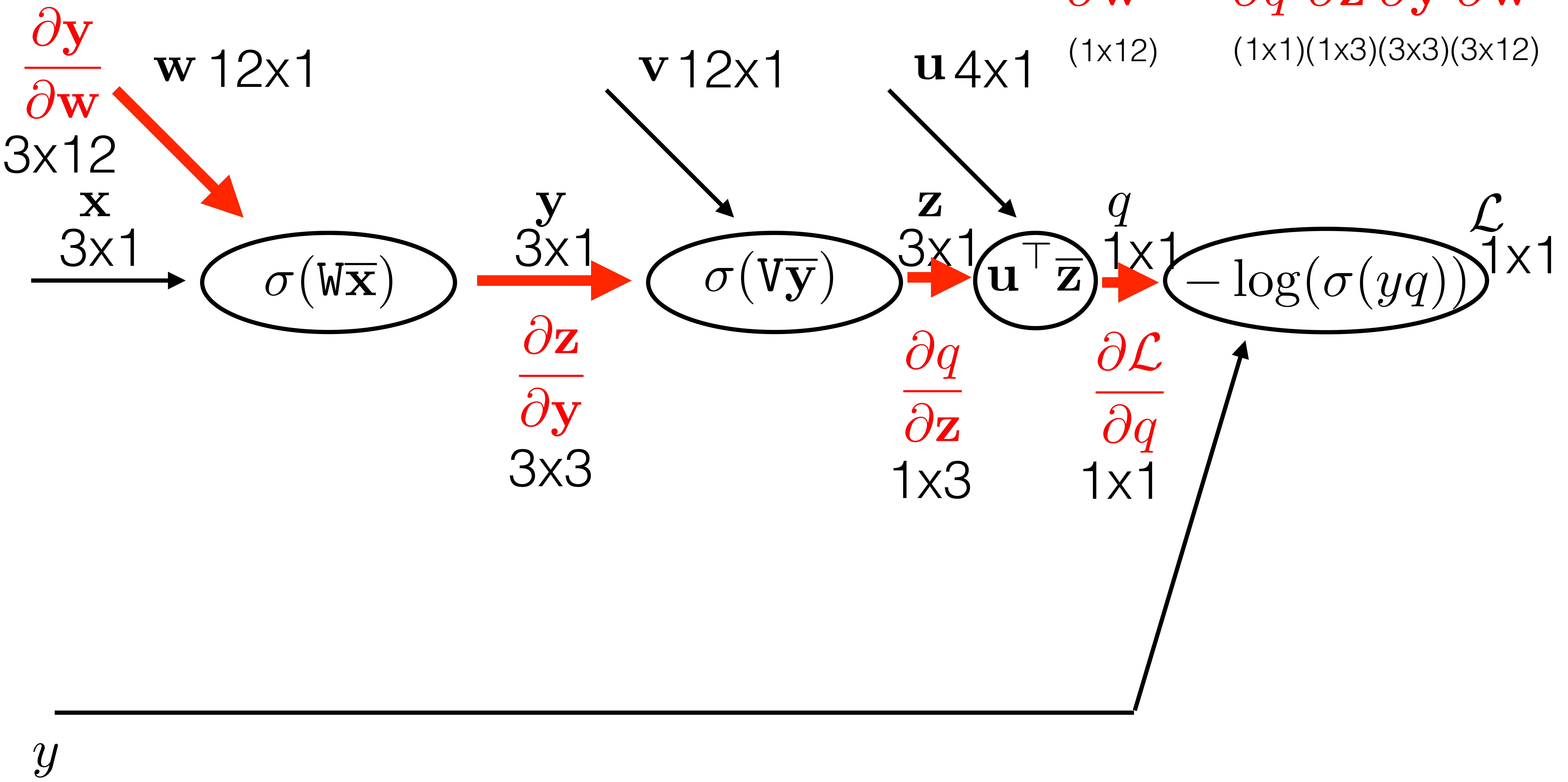
(1x1)(1x3)(3x3)(3x12)



Chainrule in fully-connected neural network

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

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

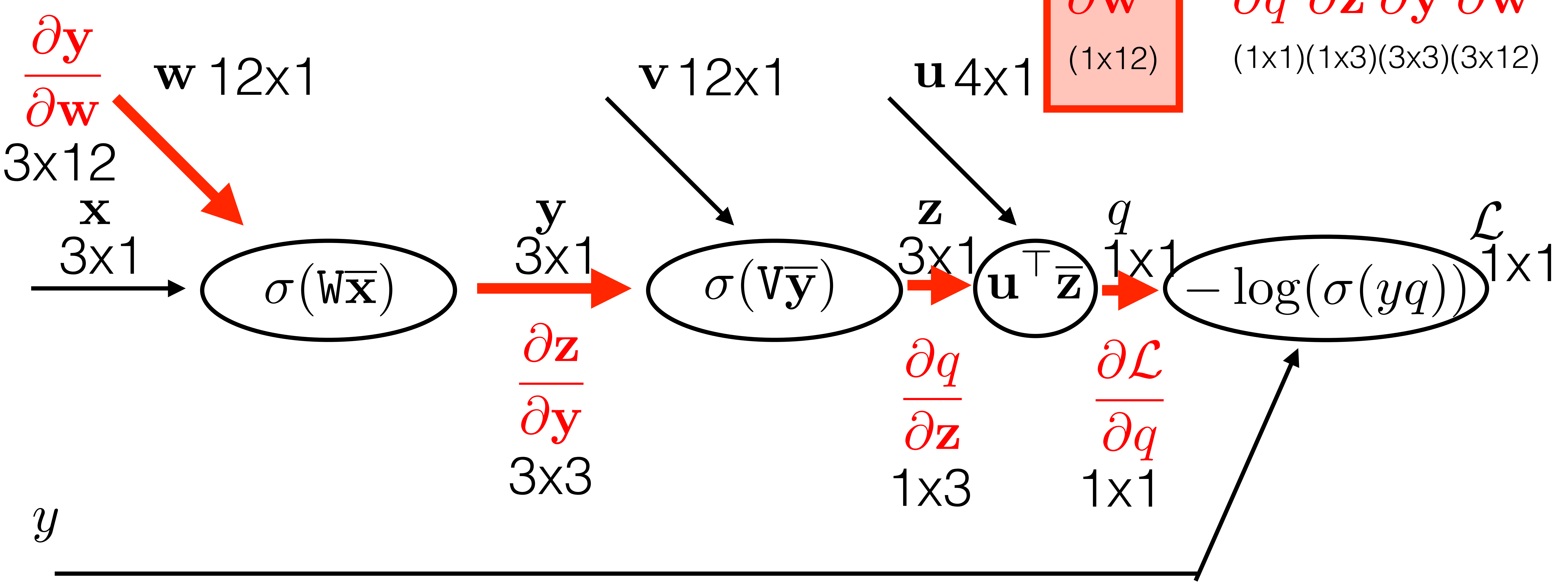


Efficient implementation of autodiff?

The main purpose is estimation of this

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

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

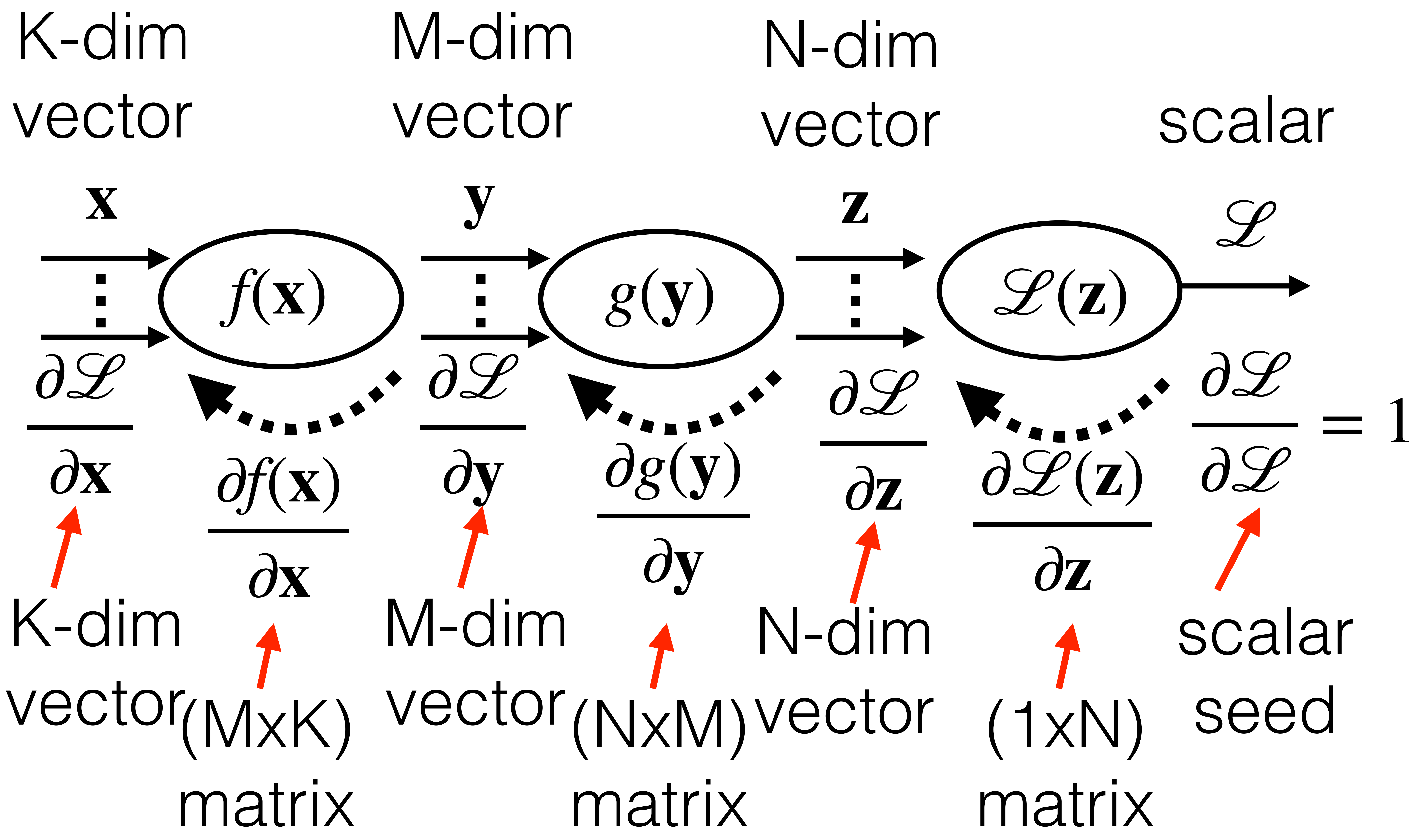


Forward differentiation (JVP): computes Jacobian of the loss one column at a time

Reverse differentiation (VJP): computes Jacobian of the loss one row at a time

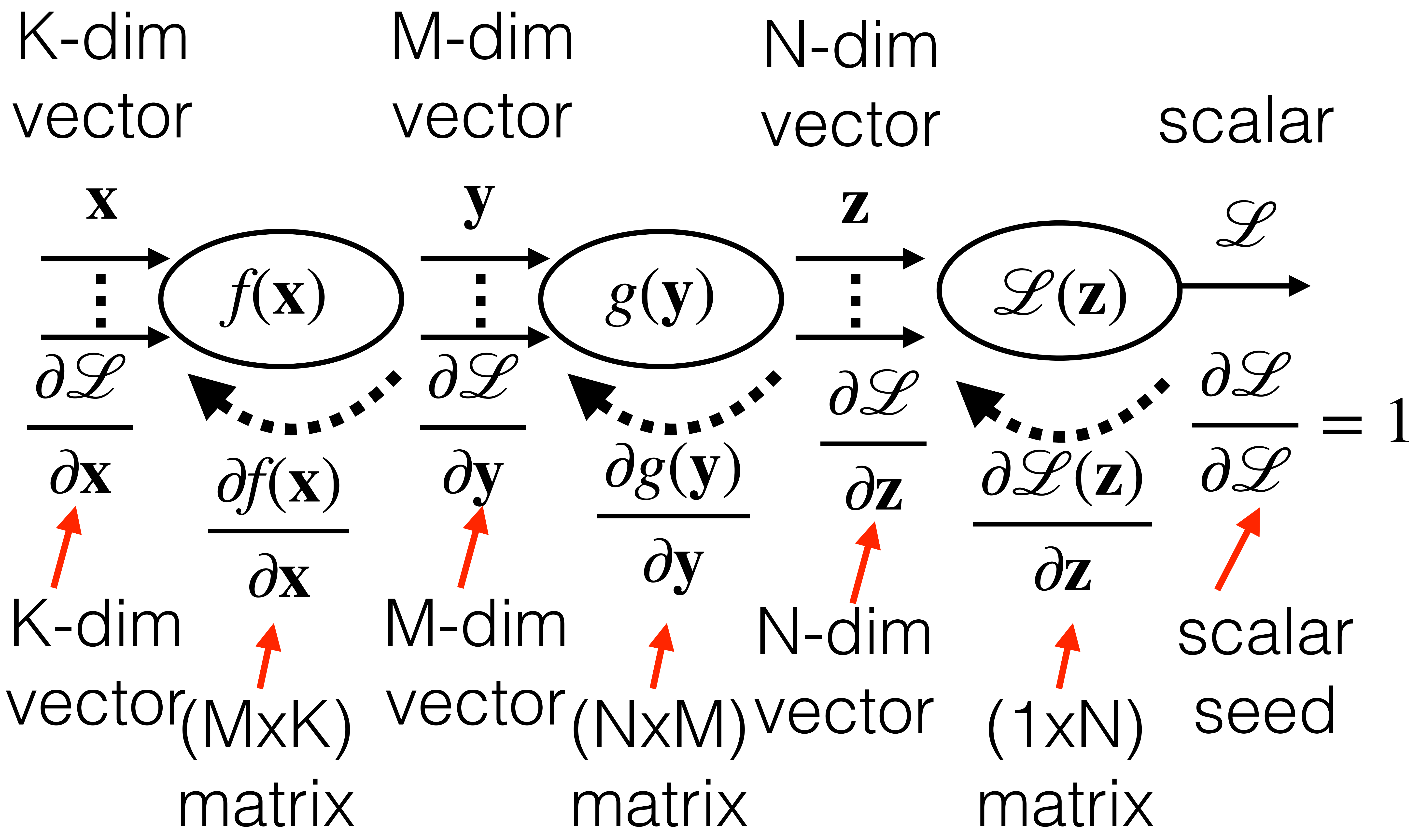
Which one would you choose (we want to compute 1x12 jacobian)?

Efficient implementation of autodiff?



	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?



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

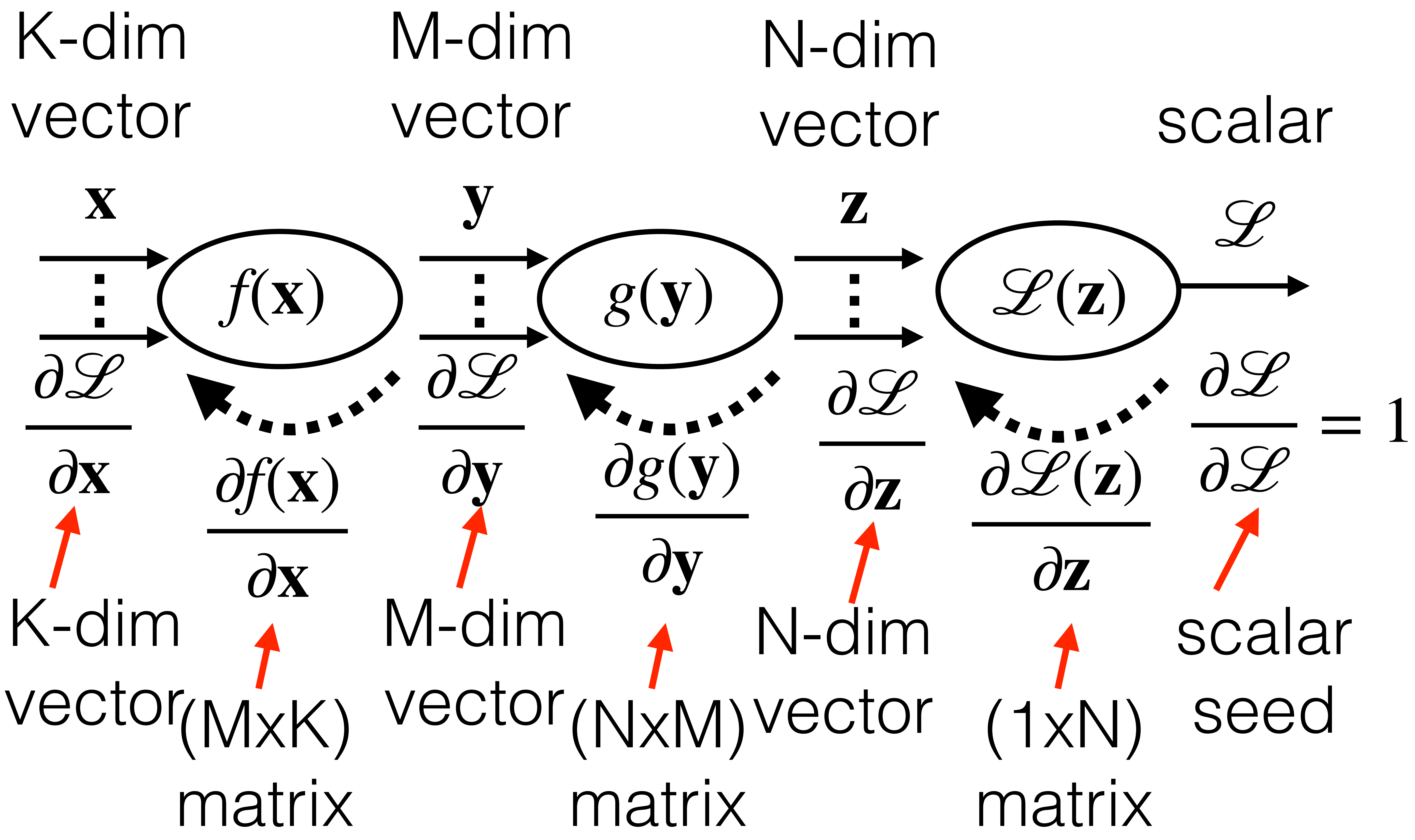
def $\text{vjp}_{g}(\mathbf{v}, \mathbf{y})$:
    return $\mathbf{v}^\top \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

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

Loss jac wrt $\mathbf{x}$

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

Efficient implementation of autodiff?



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

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

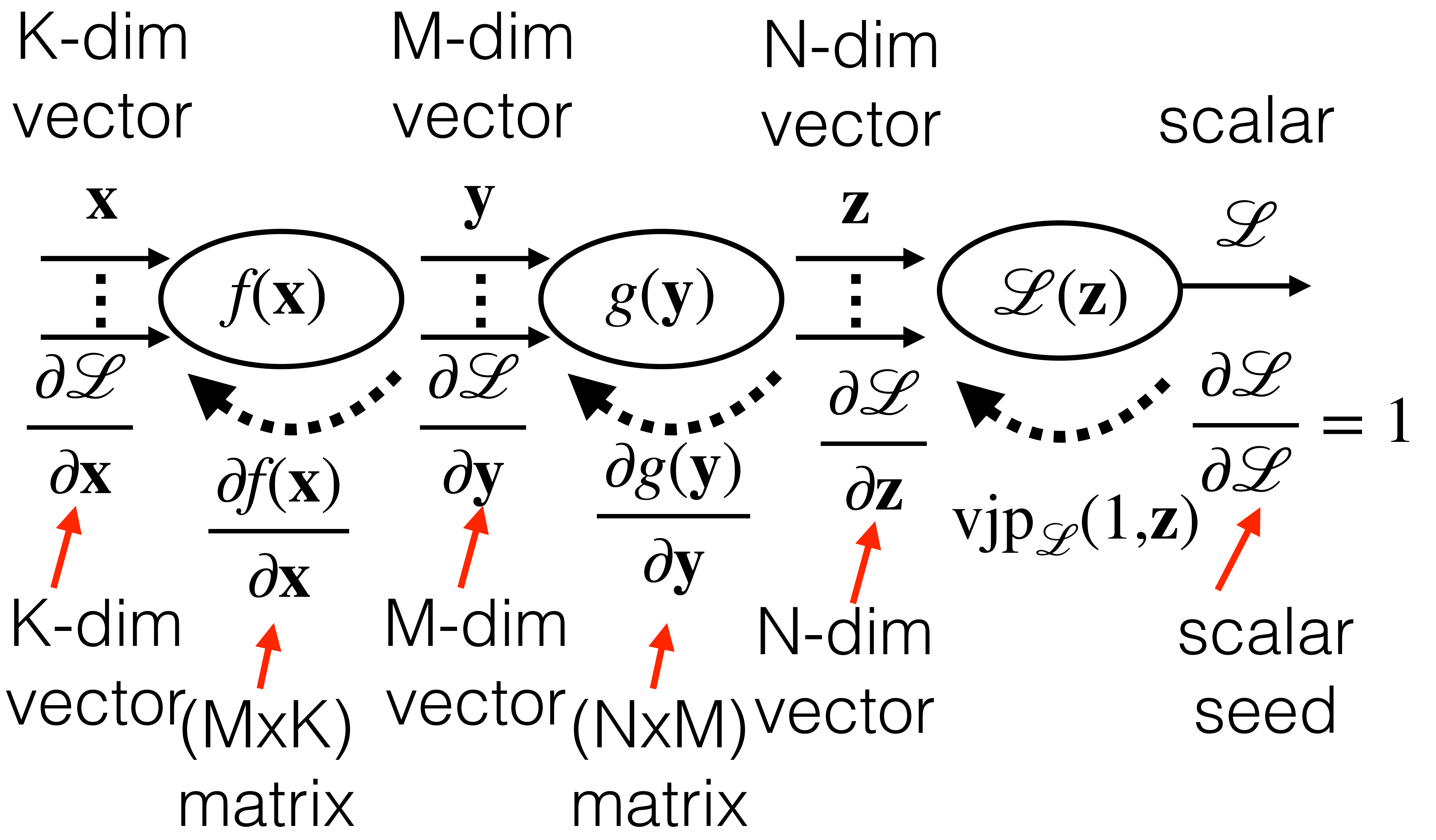
def $\text{vjp}_{f}(\mathbf{v}, \mathbf{x})$:
return $\mathbf{v}^\top \cdot \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$

Loss jac wrt $\mathbf{x}$

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

$\text{vjp}_{\mathscr{L}}(1, \mathbf{z})$

Efficient implementation of autodiff?



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

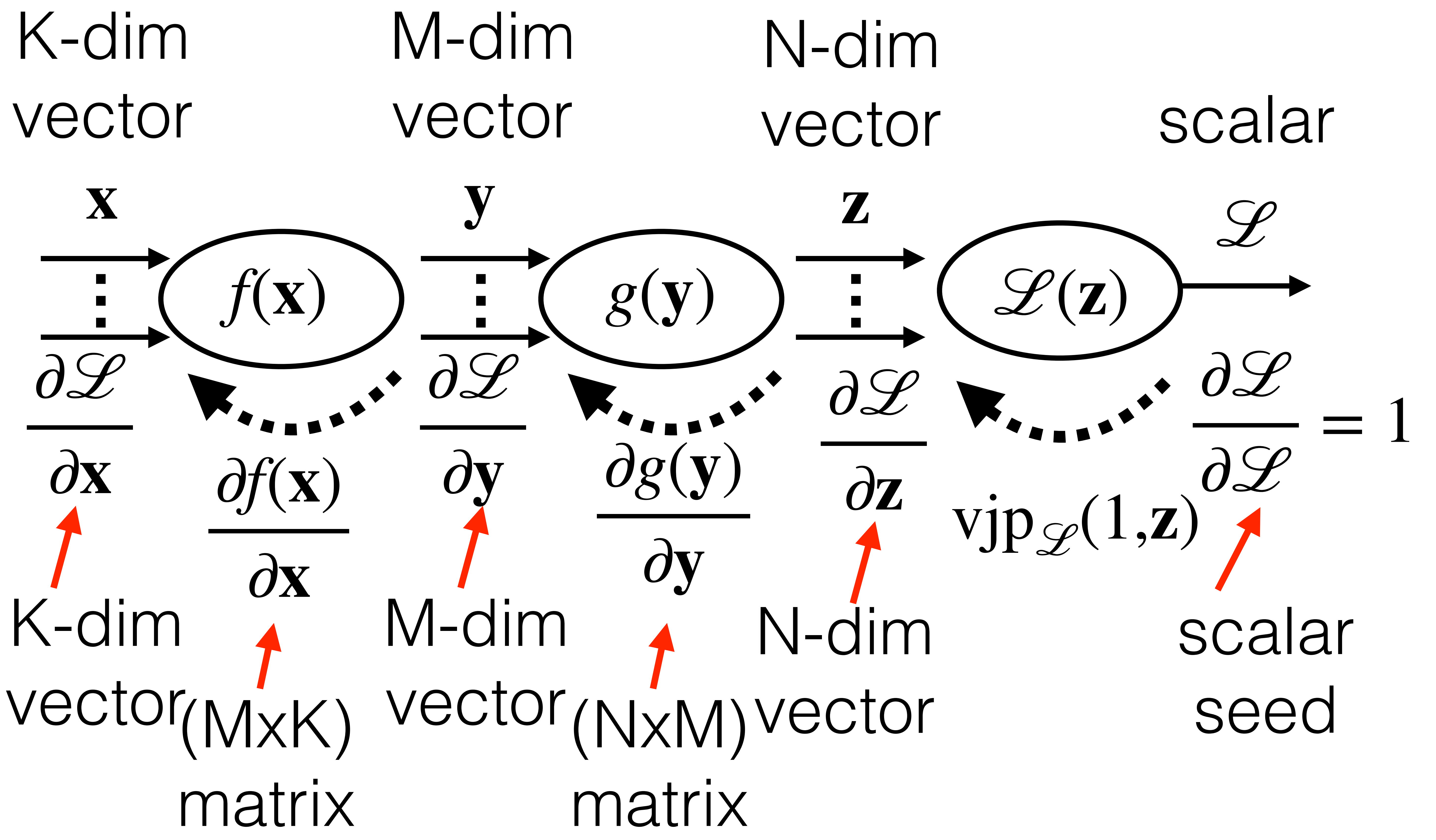
def $\mathrm{vjp}_{g}(\mathbf{v},\mathbf{y})$:
    return $\mathbf{v}^\top \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

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

1xK, NxM, MxK

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

Efficient implementation of autodiff?




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

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

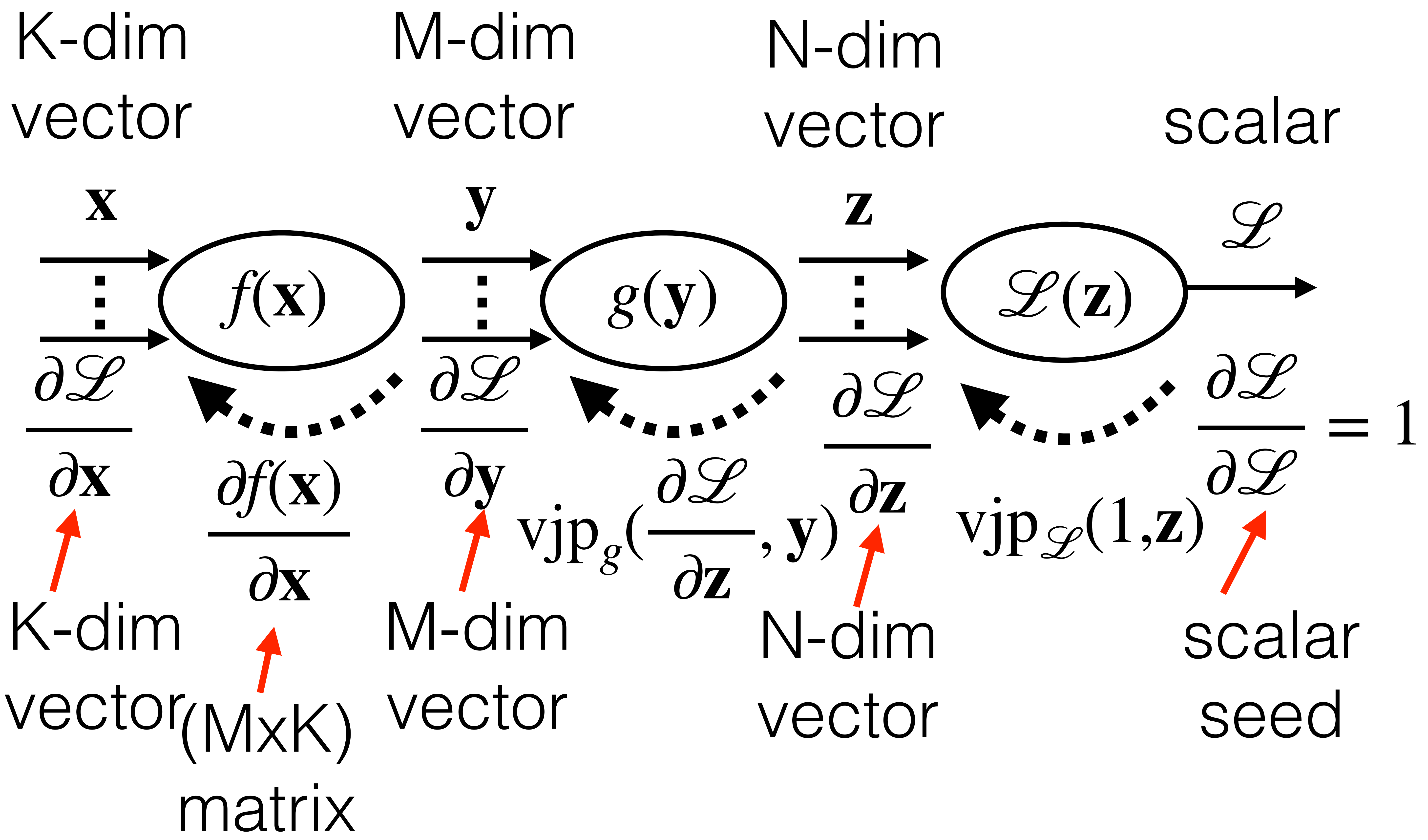
$$\text{def } \text{vjp}_{g}(\mathbf{v}, \mathbf{y}): \quad \text{return } \mathbf{v}^\top \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$$

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

1xK NxM MxK

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

Efficient implementation of autodiff?



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

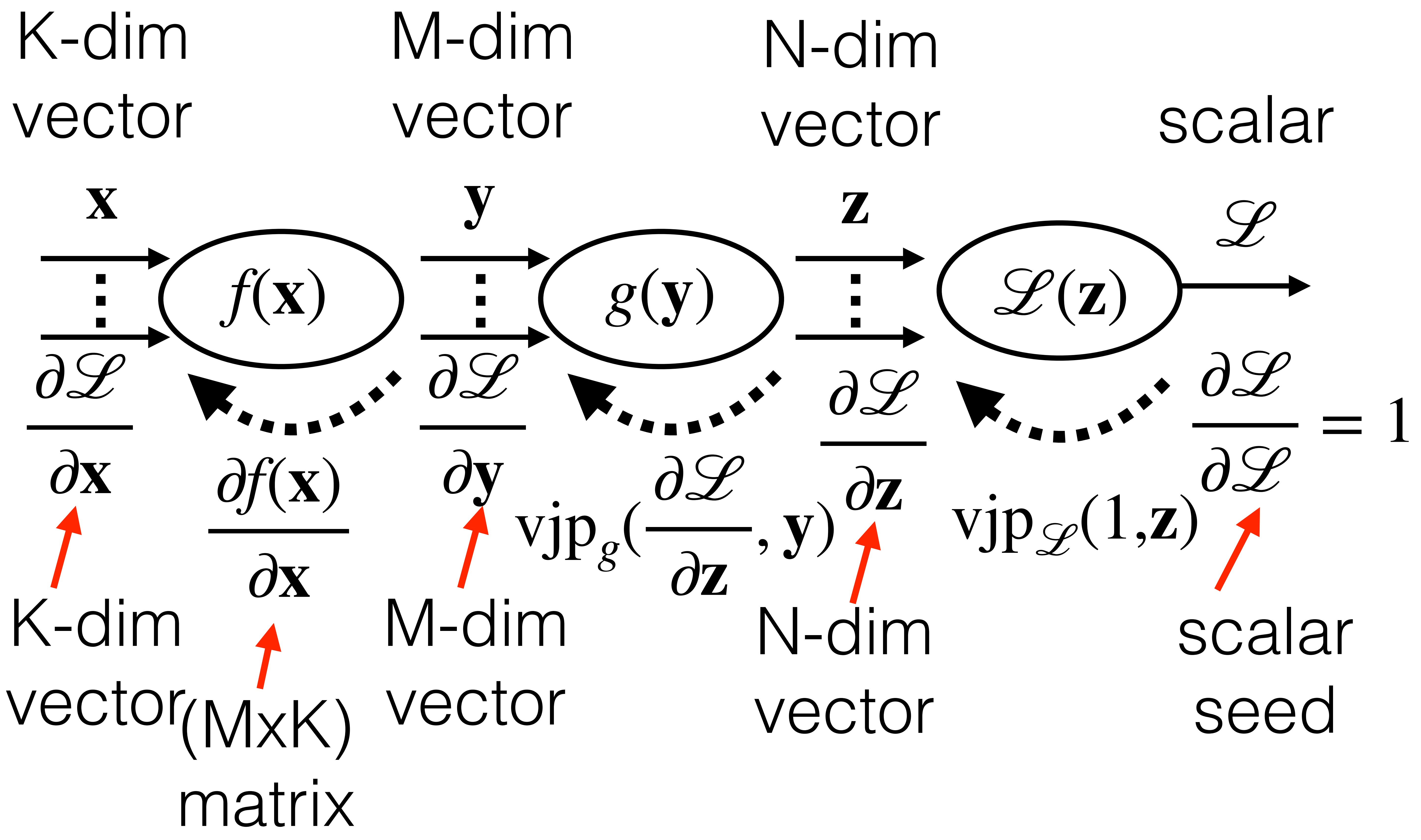
def $\mathrm{vjp}_{g}(\mathbf{v}, \mathbf{y})$:
  return $\mathbf{v}^\top \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

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

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

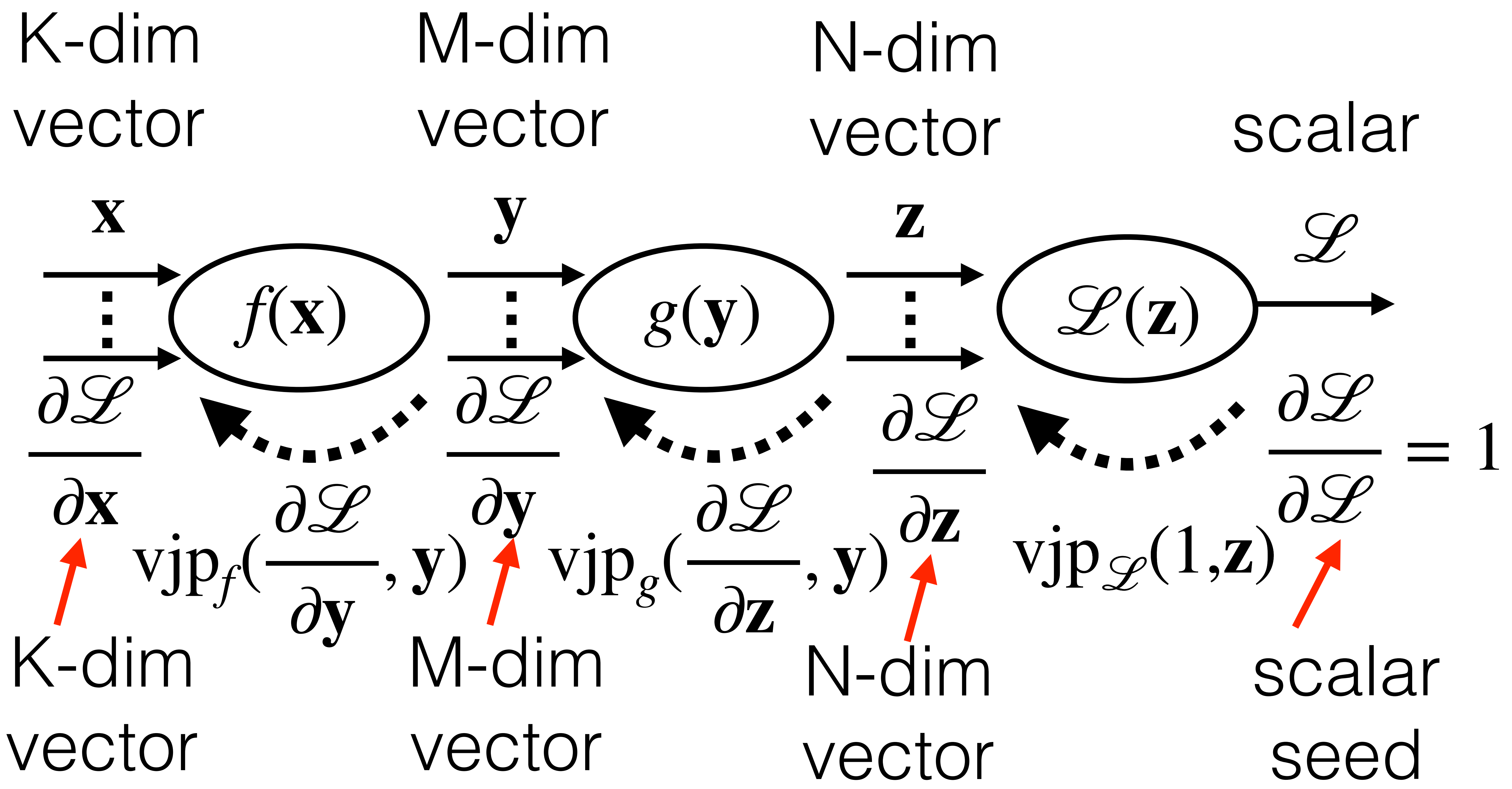
def $\text{vjp}_{g}(\mathbf{v}, \mathbf{y})$:
    return $\mathbf{v}^{\top} \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

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

1xK   MxK

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

Efficient implementation of autodiff?



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

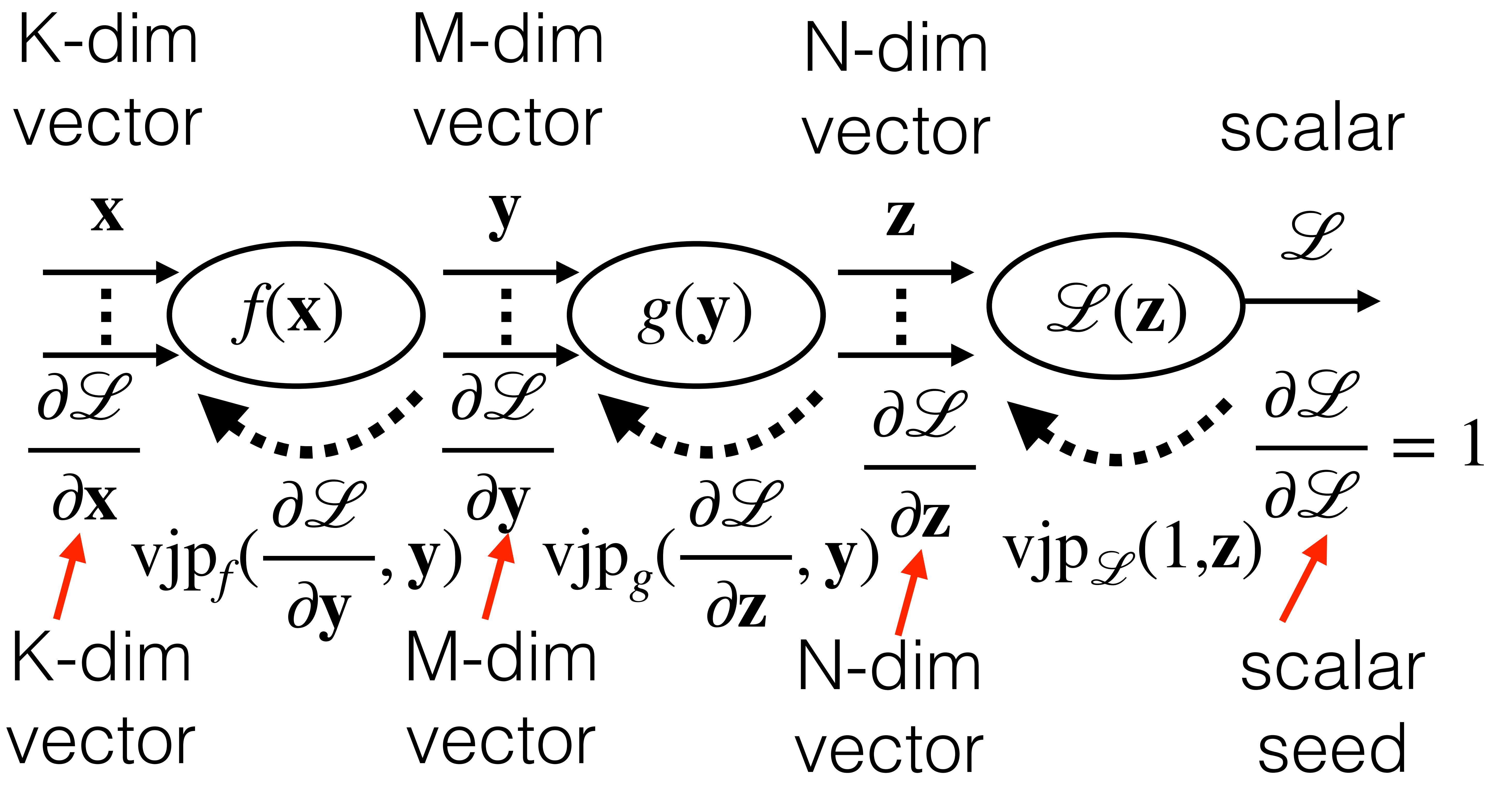
def $\text{vjp}_g(\mathbf{v}, \mathbf{y})$:
    return $\mathbf{v}^\top \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

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

1xK

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

Efficient implementation of autodiff?



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

return $\mathbf{v}^\top \cdot \frac{\partial \mathcal{L}(\mathbf{z})}{\partial \mathbf{z}}$

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

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

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

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

Reverse autodiff algorithm

input: $\mathbf{x}$

forward pass:

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

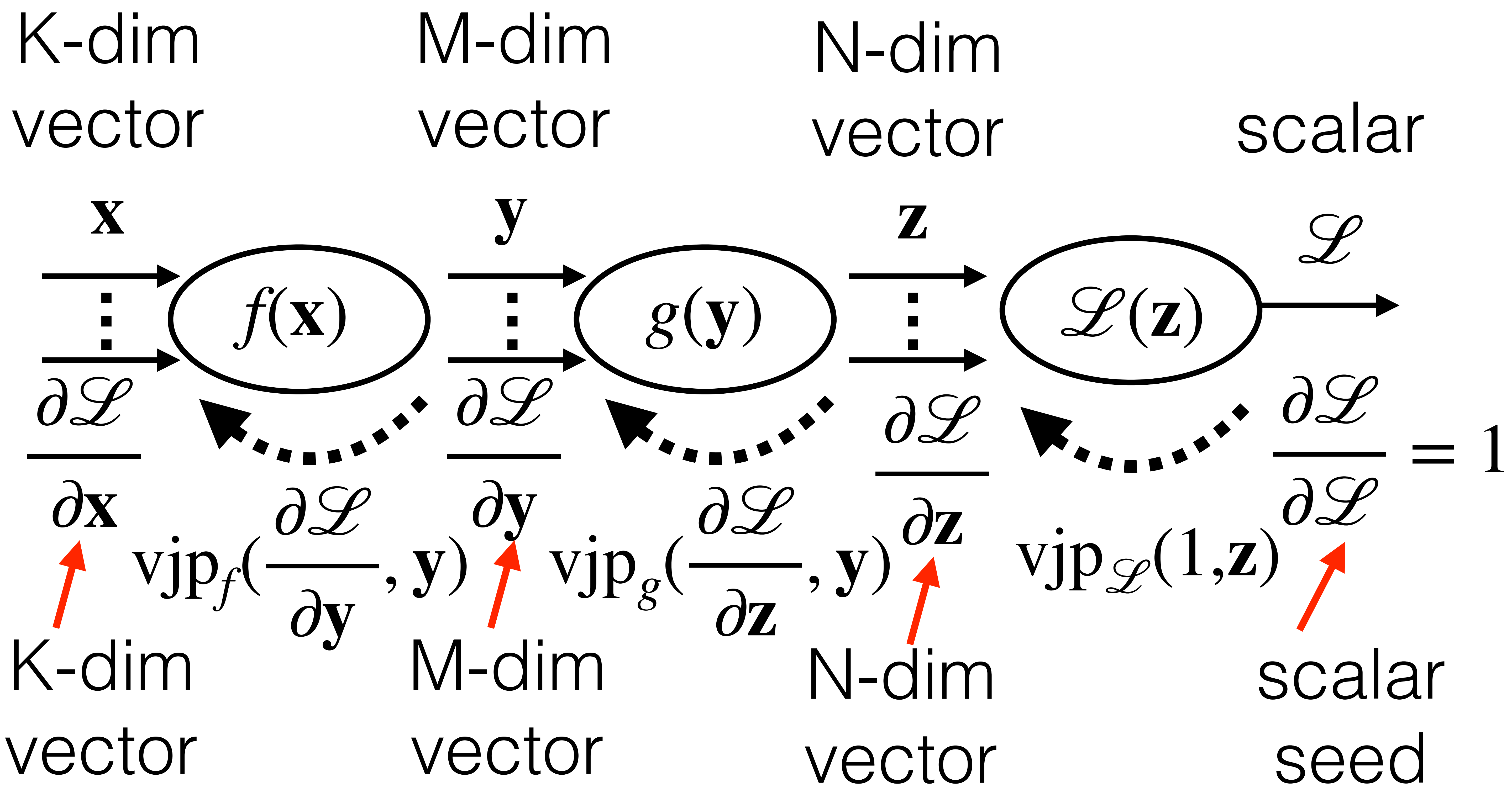
backward pass:

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

output:

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

Efficient implementation of autodiff?



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

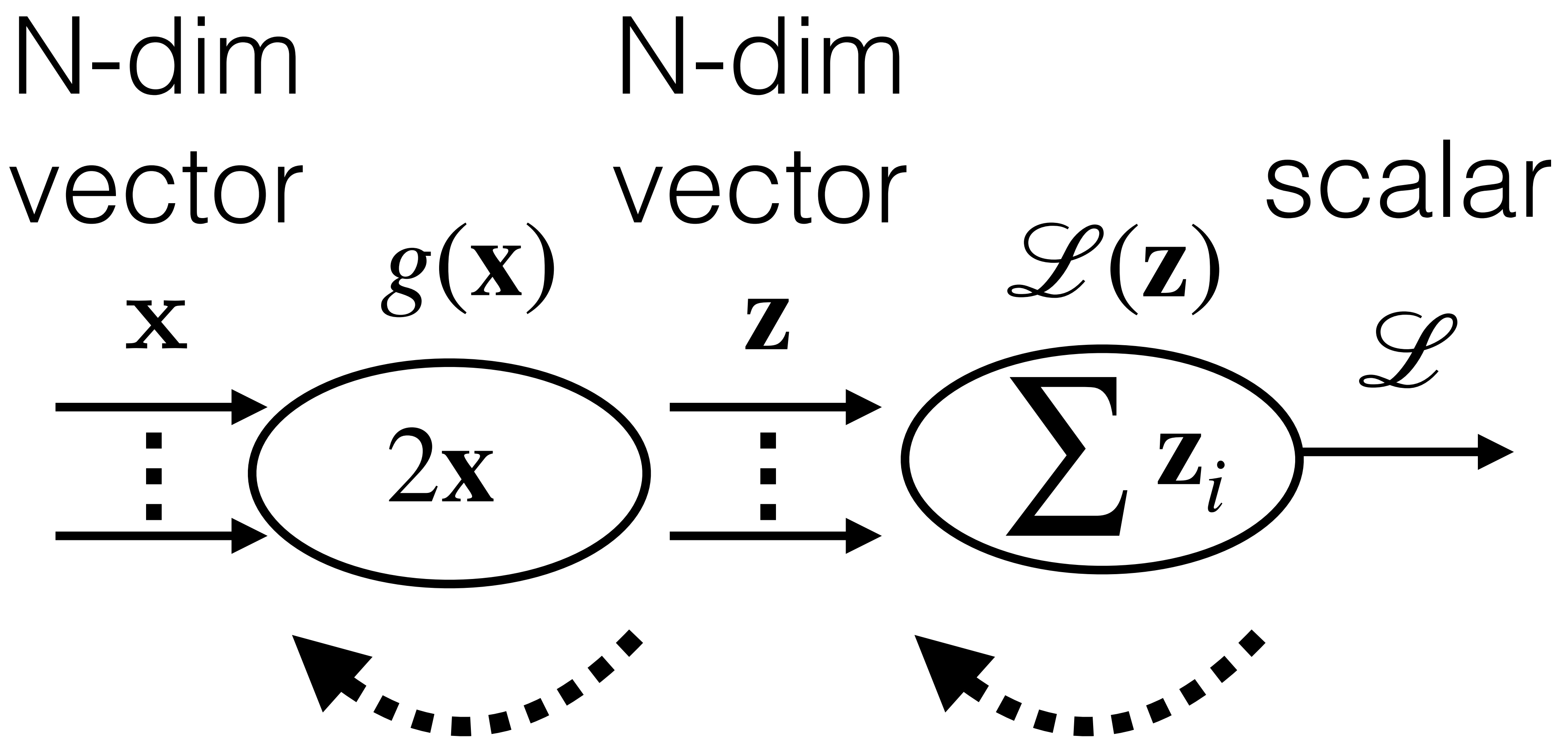
def $\mathrm{vjp}_g(\mathbf{v}, \mathbf{y})$:
    return $\mathbf{v}^\top \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

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

Why the hell should I implement it in such a way?

- vjp usually implemented more efficiently (building the jacobian not required)
- preserves dimensionality of inputs in backward pass

- vjp usually implemented more efficiently (building the jacobian not required)



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

NXN

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

1xN

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

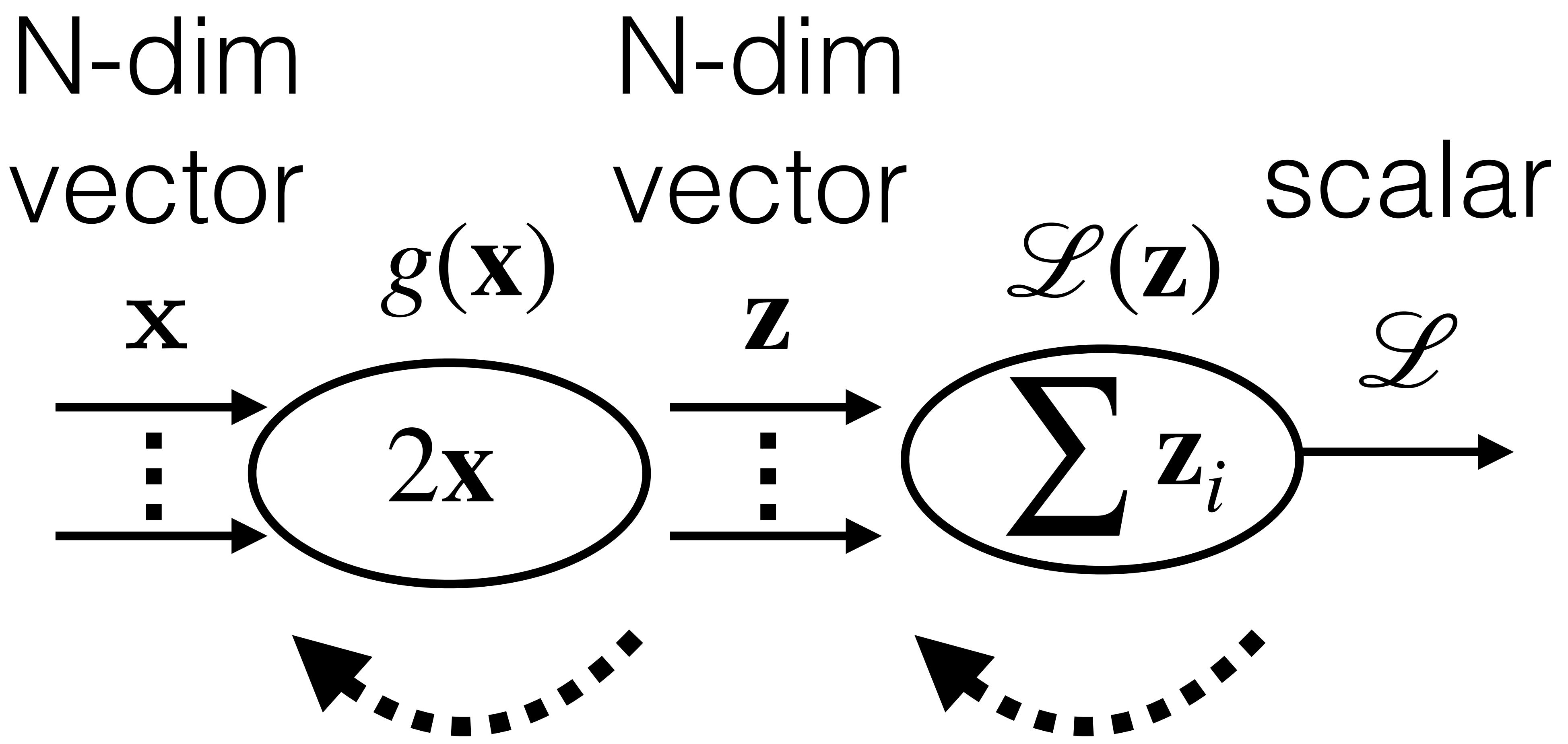
$\texttt{def}\ \mathrm{vjp}_{\mathcal{L}}(\mathbf{v}, \mathbf{z})\texttt{:}$

$\quad \texttt{return}\ \mathbf{v}^\top \cdot \frac{\partial \mathcal{L}(\mathbf{z})}{\partial \mathbf{z}}$

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

$\quad \texttt{return}\ \mathbf{v}^\top \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

- vjp usually implemented more efficiently (building the jacobian not required)



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

NXN

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

1xN

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

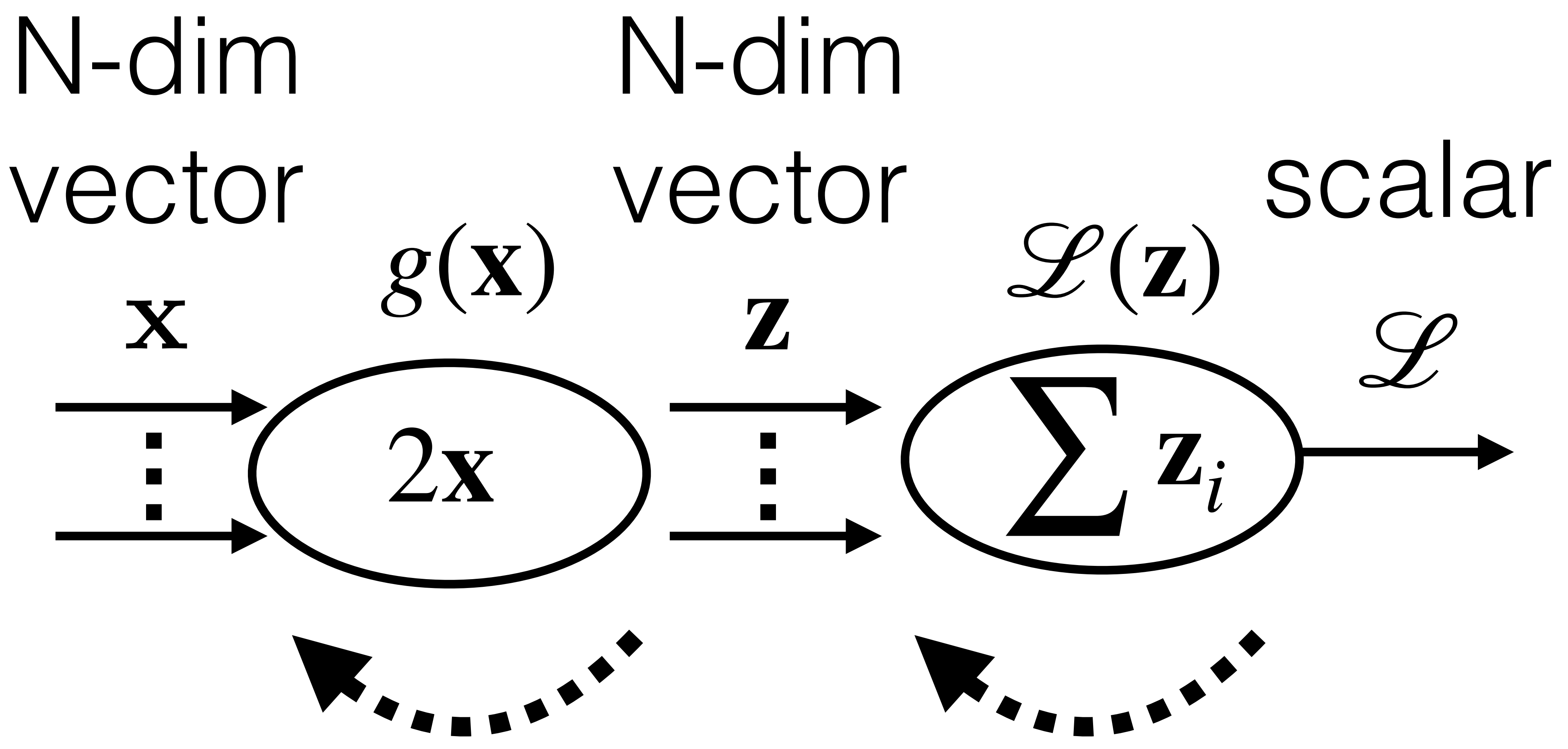
    return $\mathbf{v}^\top \cdot \begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix}$

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

    return $\mathbf{v}^\top \cdot \begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & \dots & \\ & & & & 2 \end{bmatrix}$

Do I really need to construct the jacobian explicitly???

- vjp usually implemented more efficiently (building the jacobian not required)



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

NXN

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

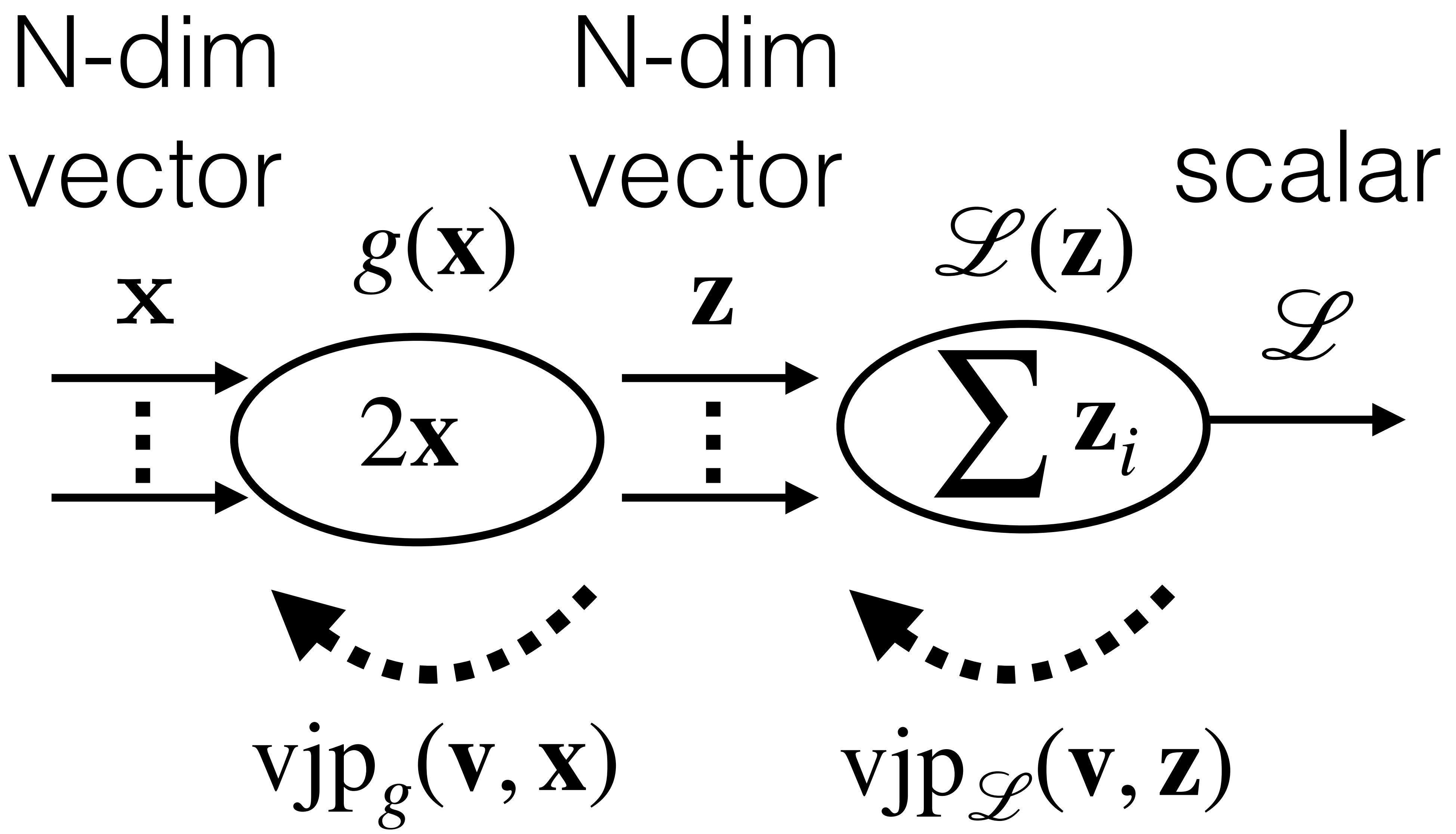
1xN

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

```
def vjp_g(v, x):
    return v^T · 2
```

Do I really need to construct the jacobian explicitly???

- vjp usually implemented more efficiently (building the jacobian not required)



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

    return tile(v,1,N)

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

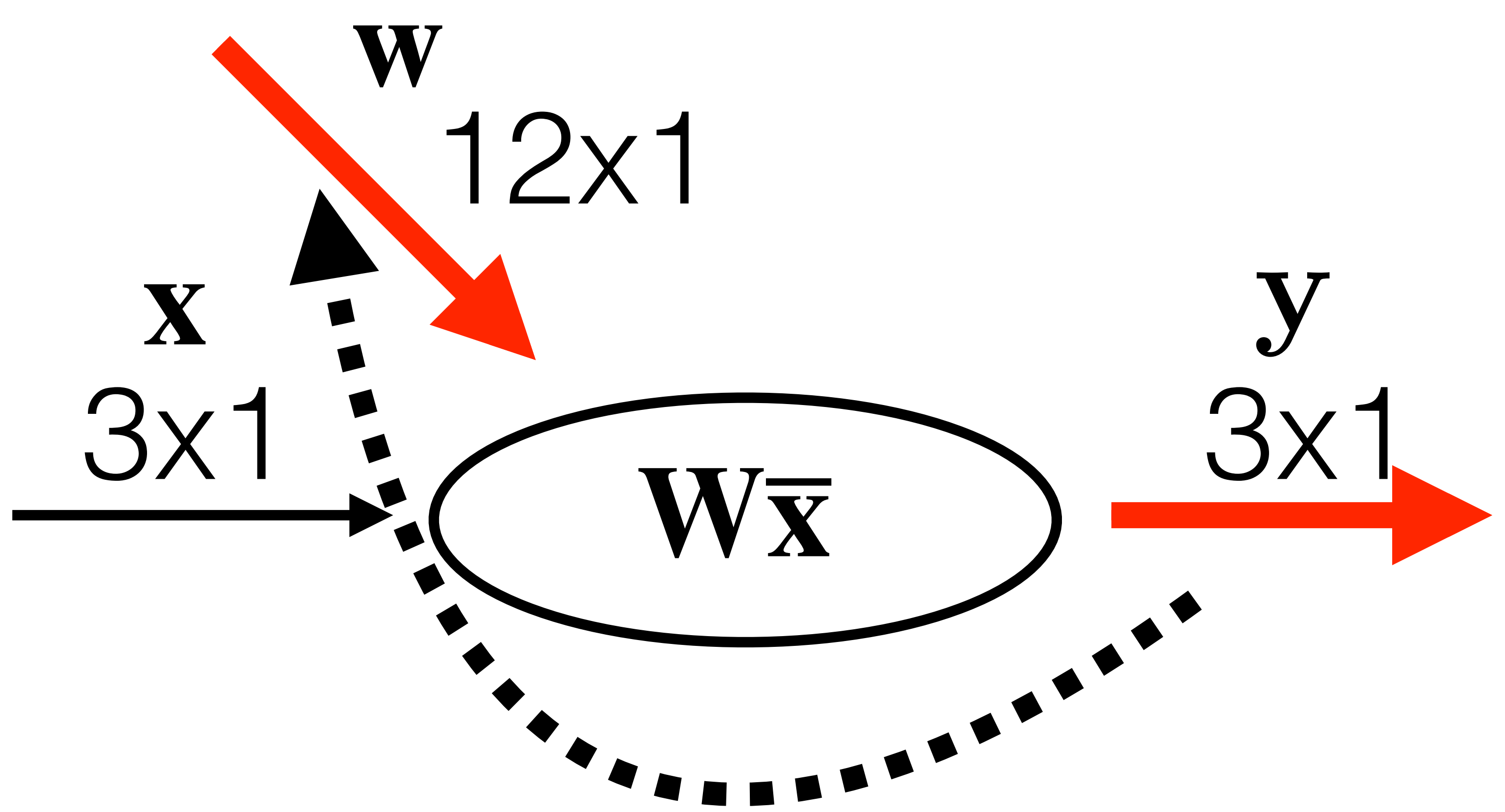
    return $\mathbf{v}^\top \cdot 2$

vjp is automatically added to tensors during the construction of comp.g.

`grad_fn=<fBackward>`

```
x = torch.tensor([[1,2], [3, 4]], dtype=torch.float32, requires_grad=True)
z = x.sum()
print(z)
tensor(10, grad_fn=<SumBackward0>)
```

- vjp usually implemented more efficiently (building the jacobian not required)

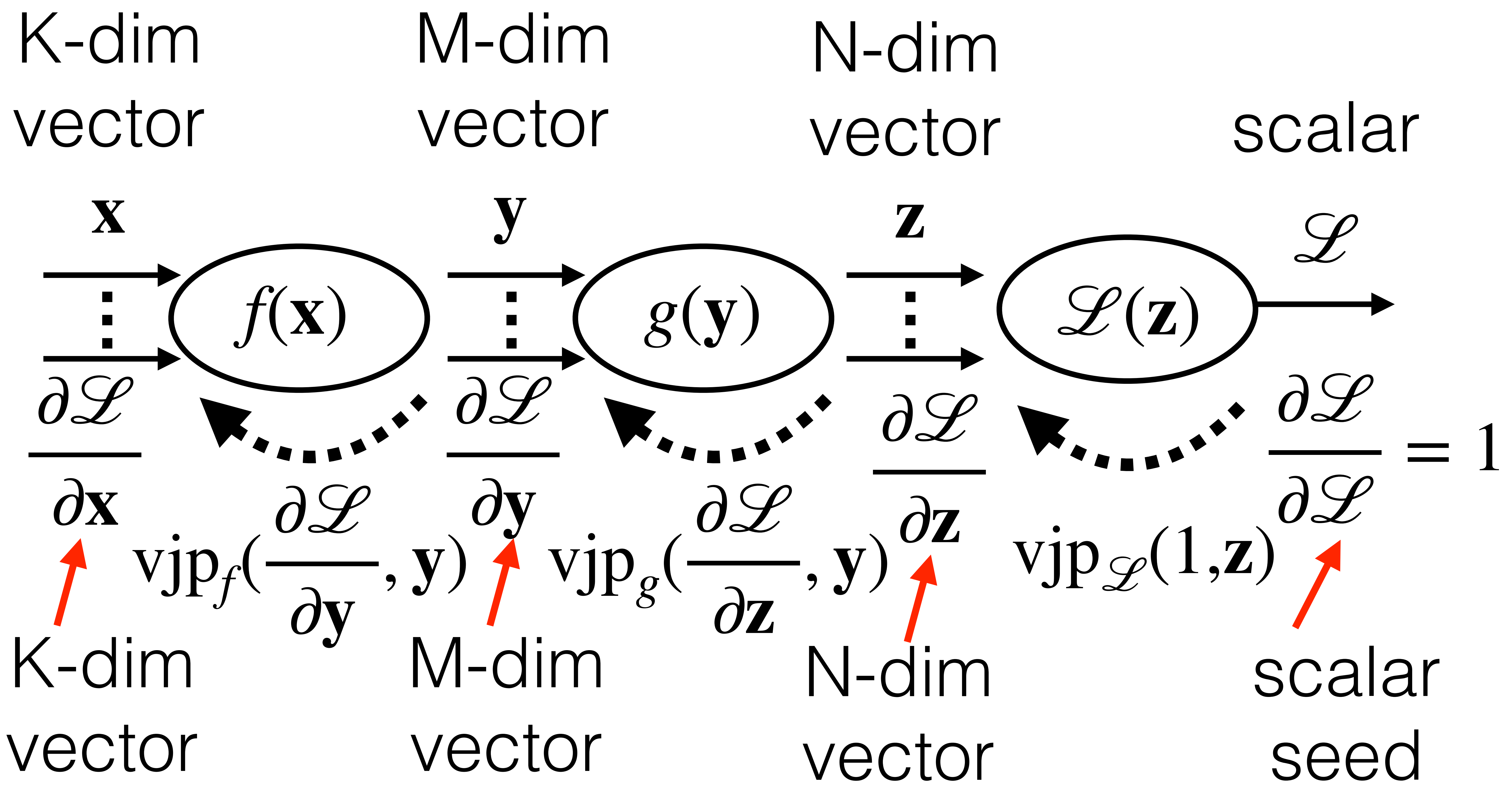


def $\text{vjp}_{\mathbf{w}}\big(\mathbf{v}, (\mathbf{x}, \mathbf{w})\big)$:

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

Efficient implementation of autodiff?



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

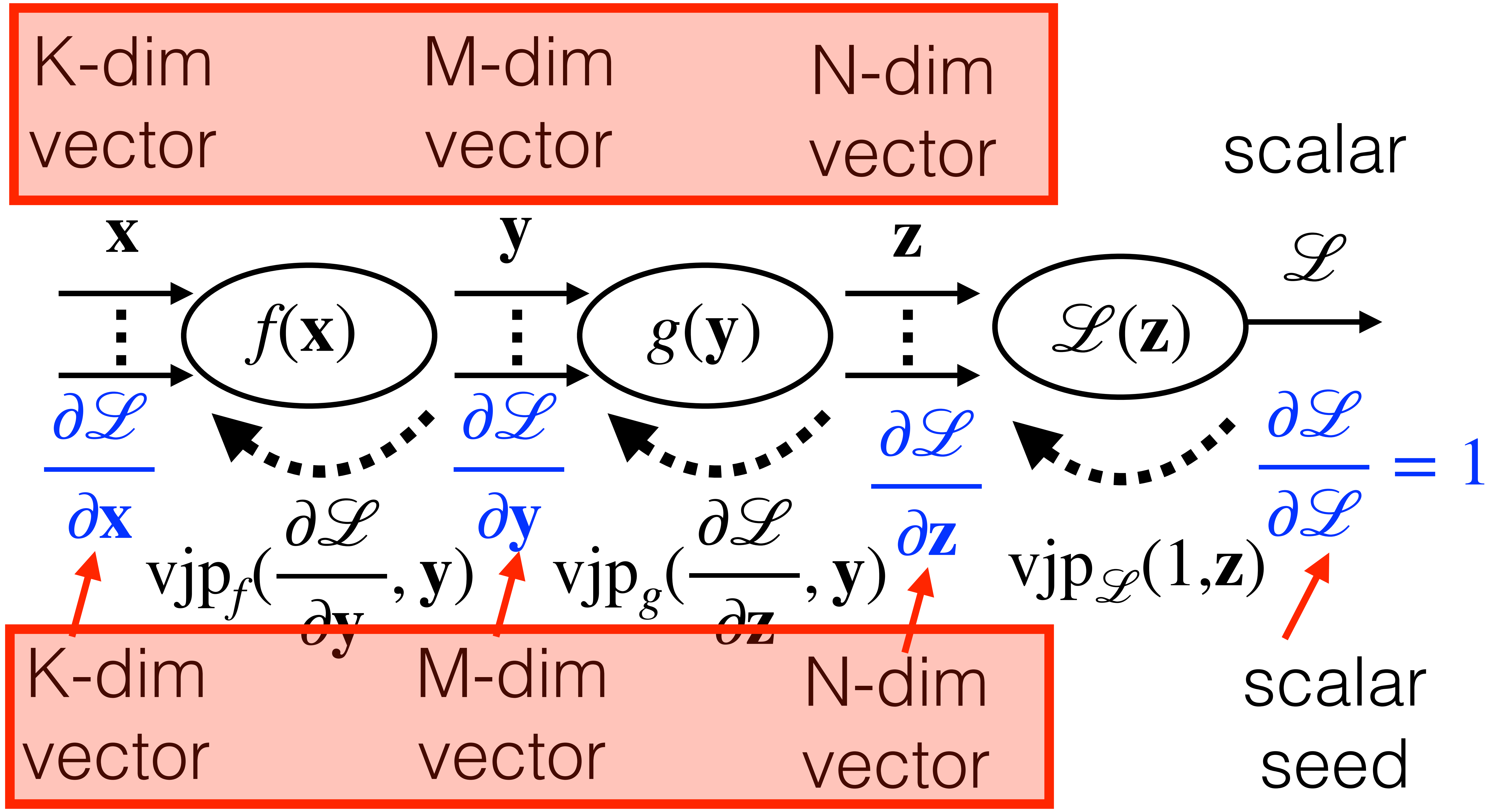
def $\mathrm{vjp}_g(\mathbf{v}, \mathbf{y})$:
    return $\mathbf{v} \cdot \frac{\partial g(\mathbf{y})}{\partial \mathbf{y}}$

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

Why the hell should I implement it in such a way?

- vjp usually implemented more efficiently (building the jacobian not required)
- preserves dimensionality of inputs in backward pass

Efficient implementation of autodiff?




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

```
def vjp_g(v, y):
    return v · ∂g(y)/∂y
```

```
def vjp_f(v, x):
    return v · ∂f(x)/∂x
```

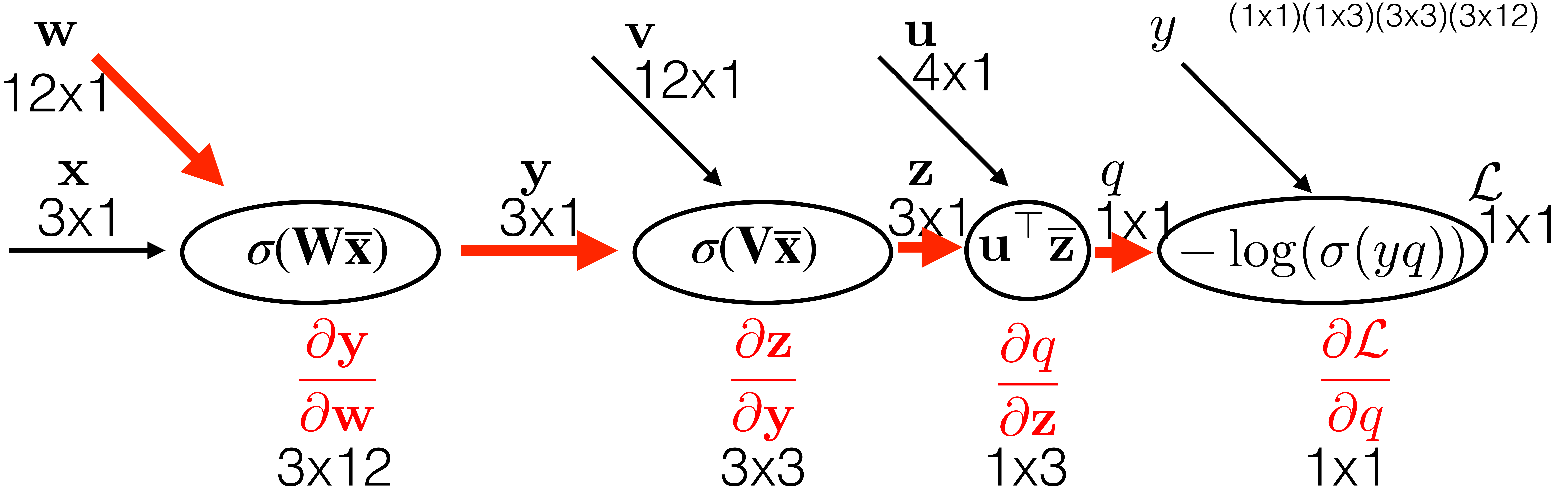
Why the hell should I implement it in such a way?

- vjp usually implemented more efficiently (building the jacobian not required)
- 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)

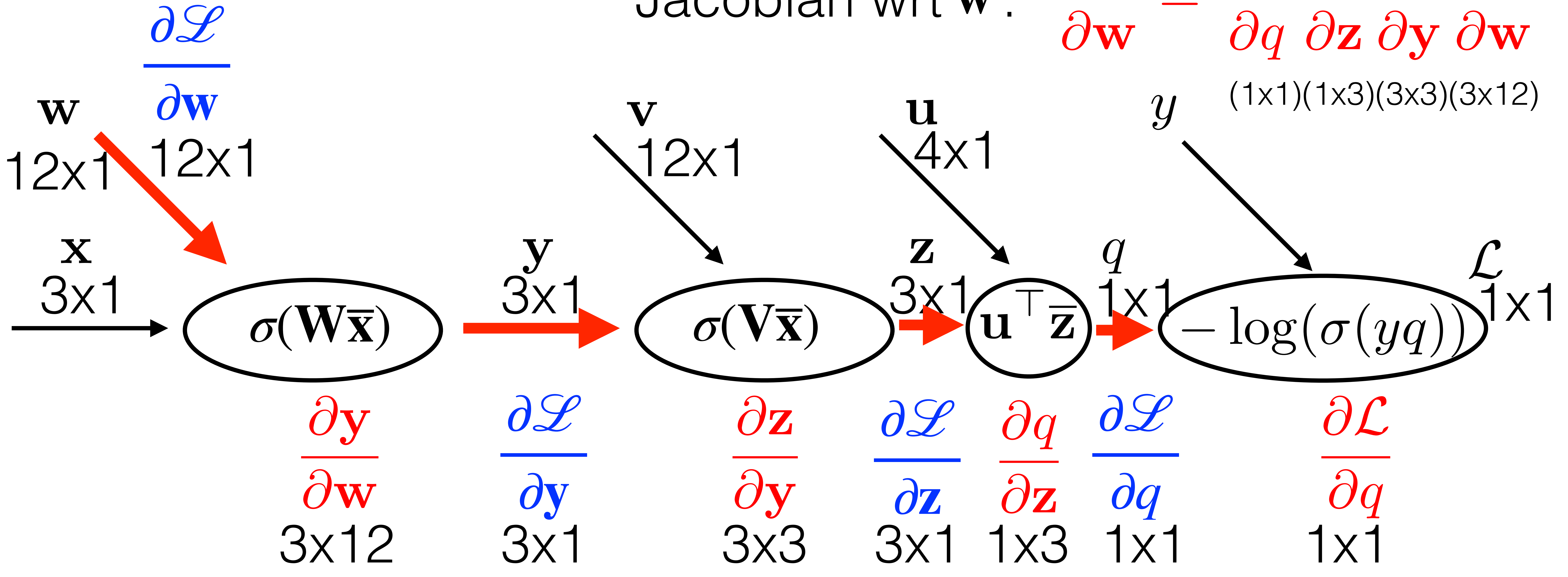


We avoided jacobian of multi-dimensional inputs by explicit vectorization

- preserves dimensionality of inputs in backward pass

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

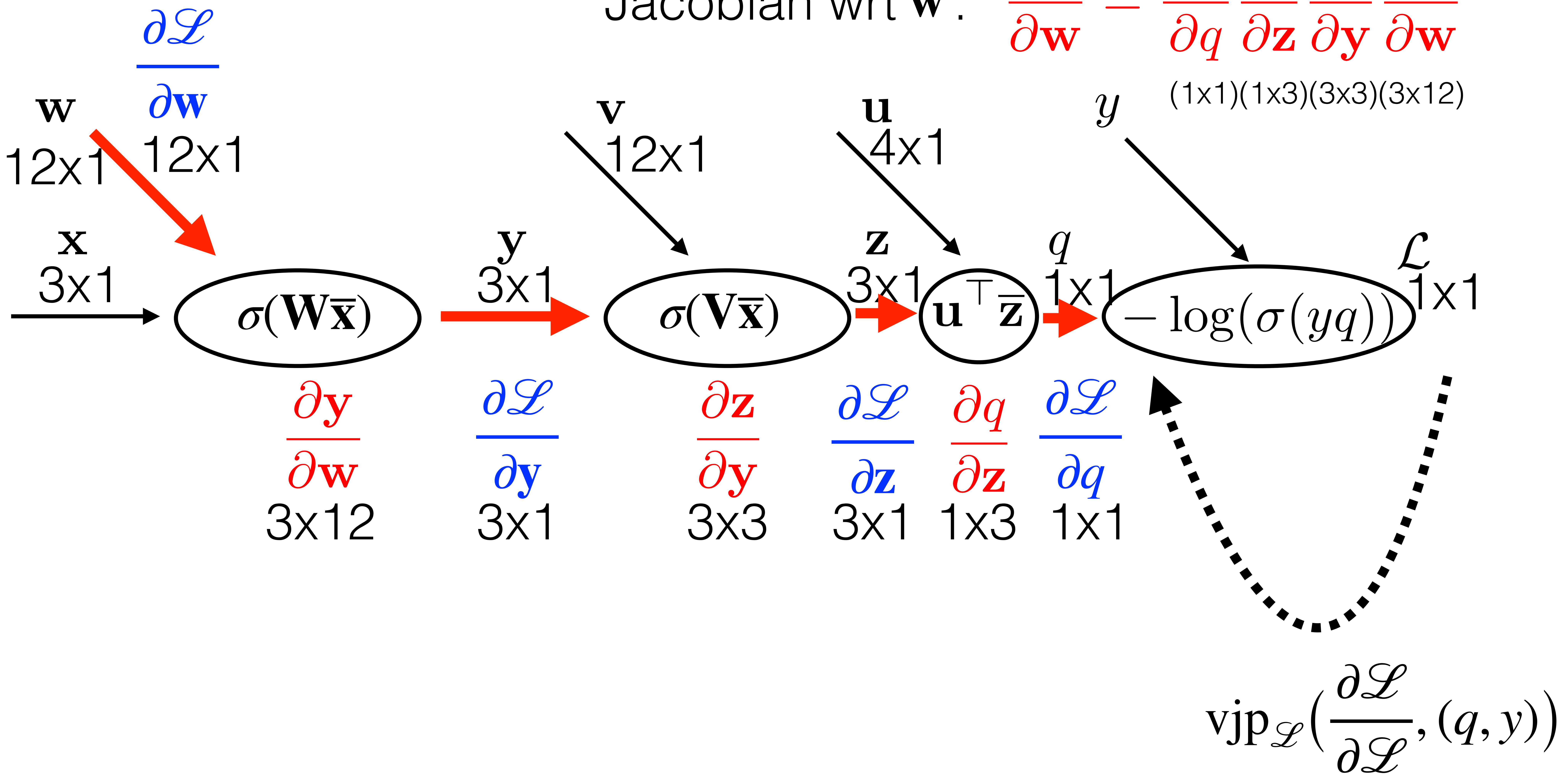
(1x1)(1x3)(3x3)(3x12)



- preserves dimensionality of inputs in backward pass

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

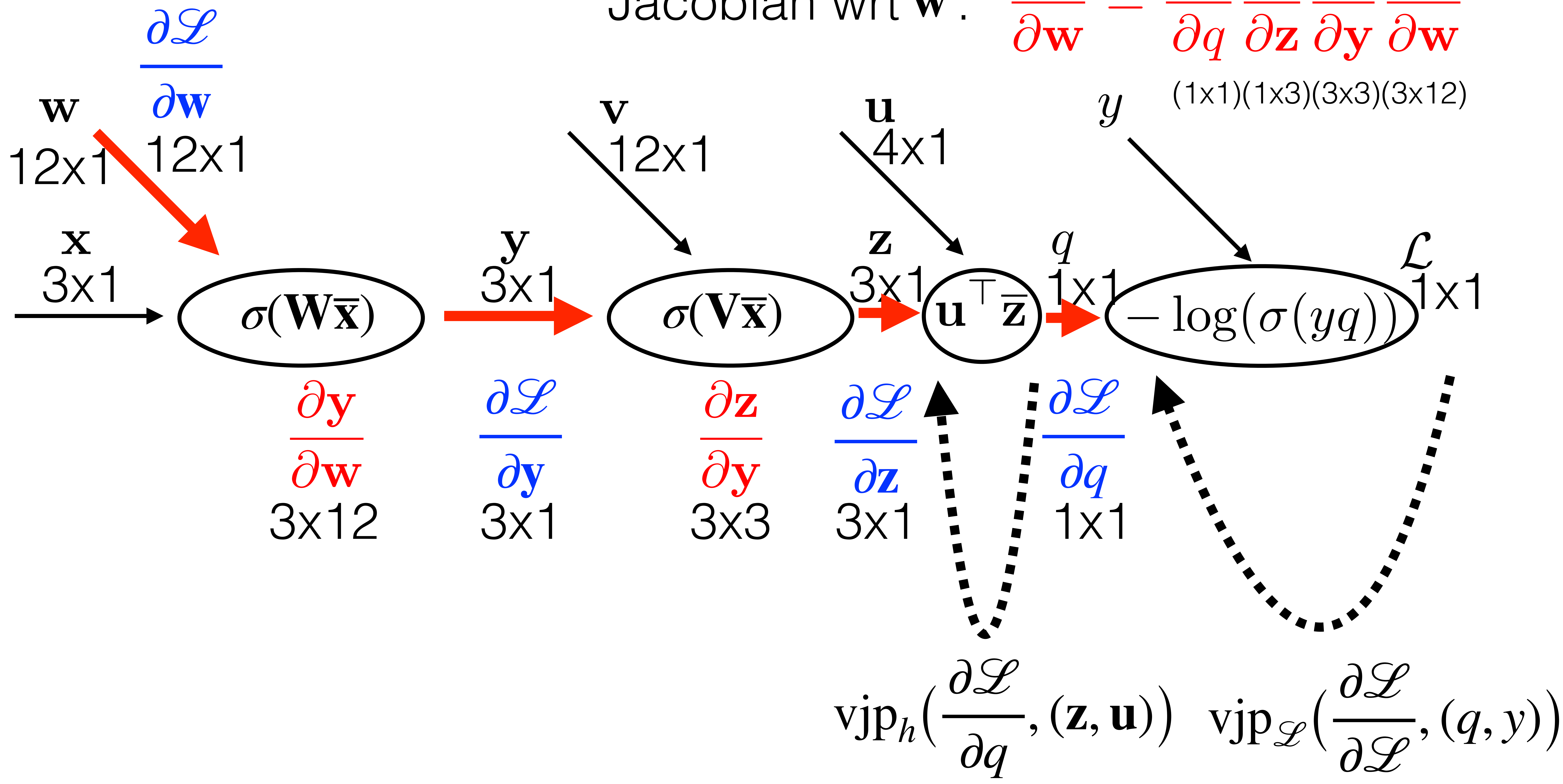
(1x1)(1x3)(3x3)(3x12)



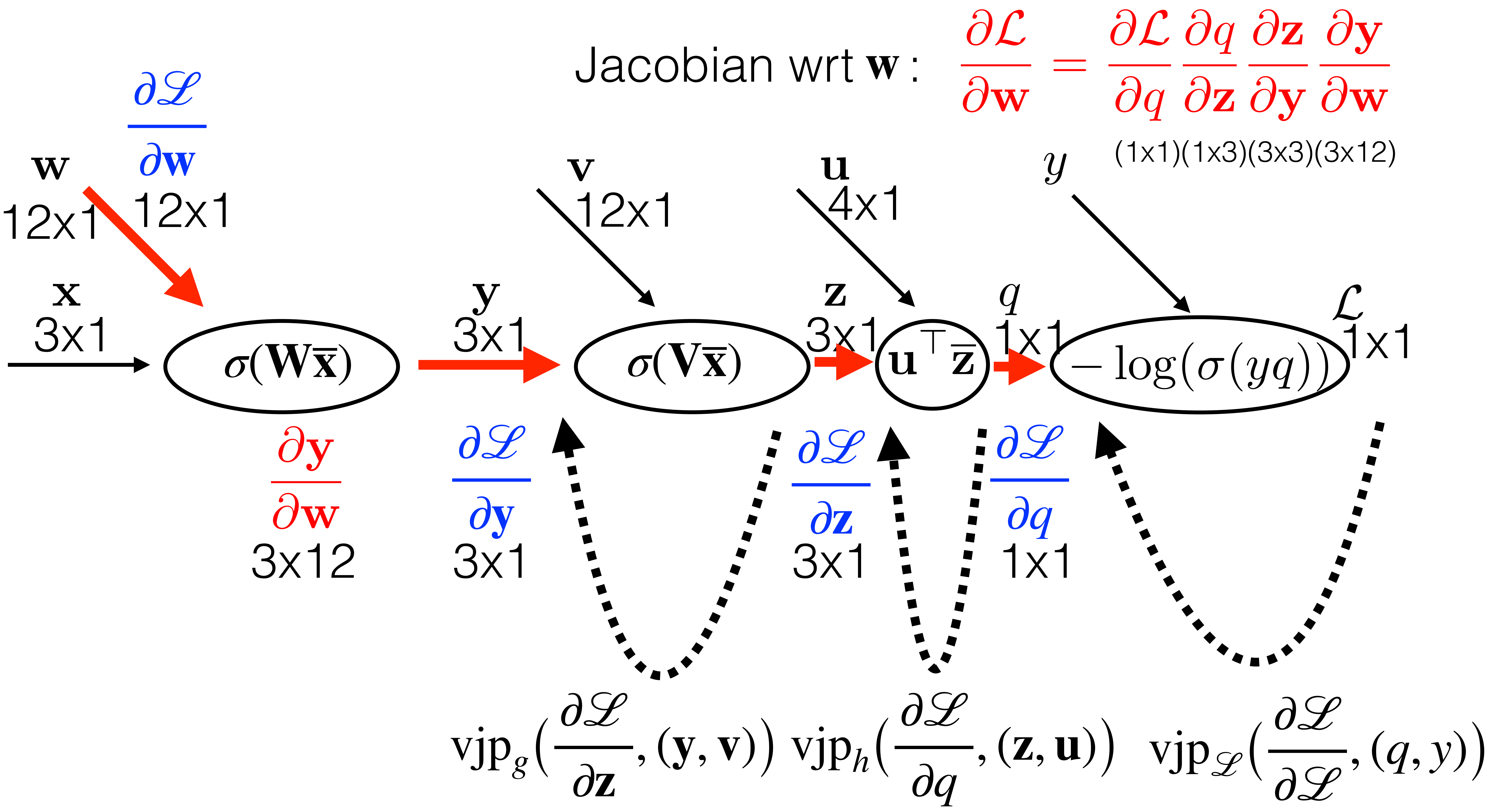
- preserves dimensionality of inputs in backward pass

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

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



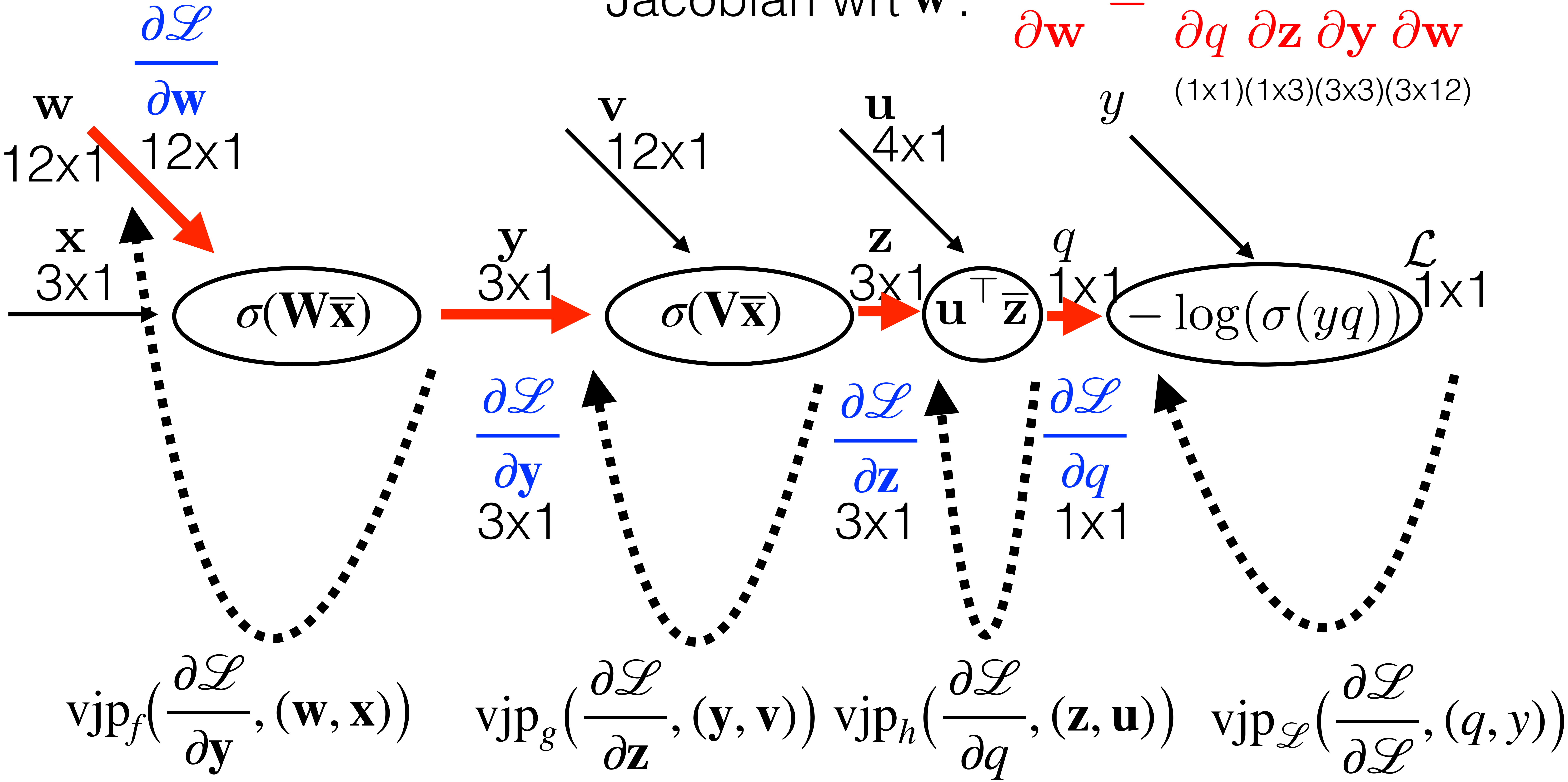
- preserves dimensionality of inputs in backward pass



- preserves dimensionality of inputs in backward pass

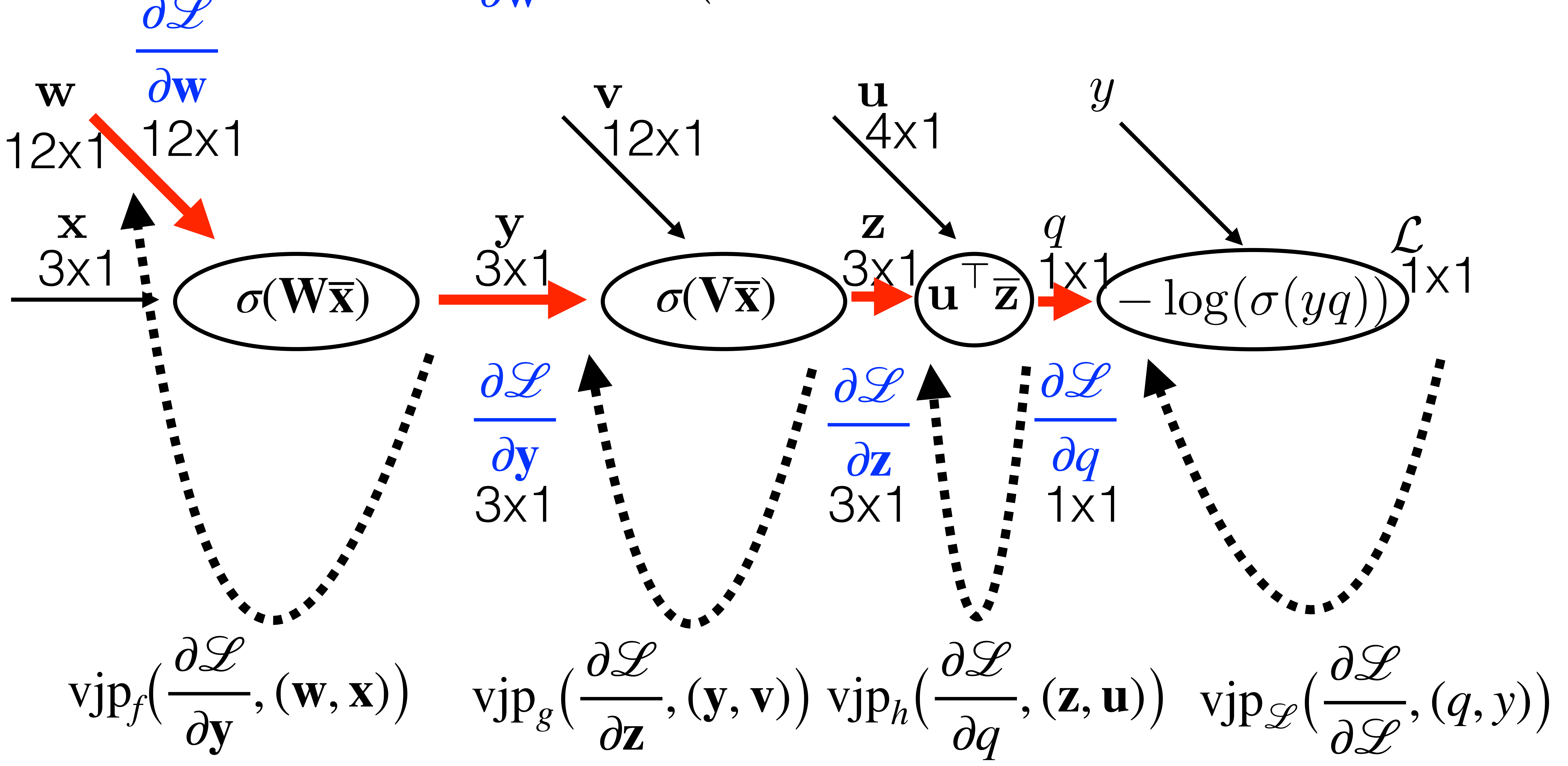
Jacobian wrt $\mathbf{w}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial q}\frac{\partial q}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{w}}$

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



- preserves dimensionality of inputs in backward pass

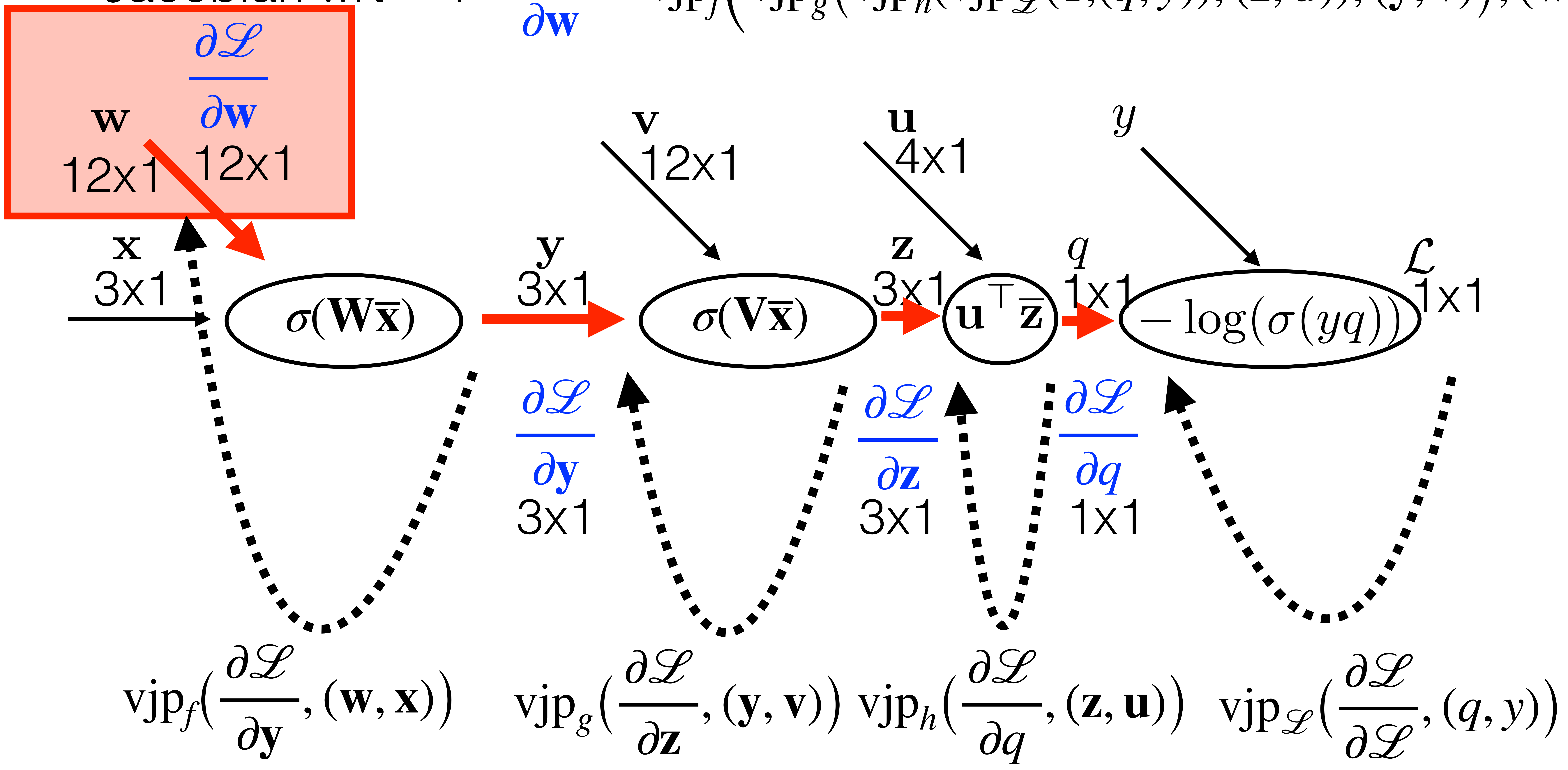
Jacobian wrt $\mathbf{w}$: $\frac{\partial \mathscr{L}}{\partial \mathbf{w}} = \text{vjp}_f\Big(\text{vjp}_g\big(\text{vjp}_h(\text{vjp}_{\mathscr{L}}(1,(q,y)),(\mathbf{z},\mathbf{u})),(\mathbf{y},\mathbf{v})\big),(\mathbf{w},\mathbf{x})\Big)$



There are no more jacobians

- preserves dimensionality of inputs in backward pass

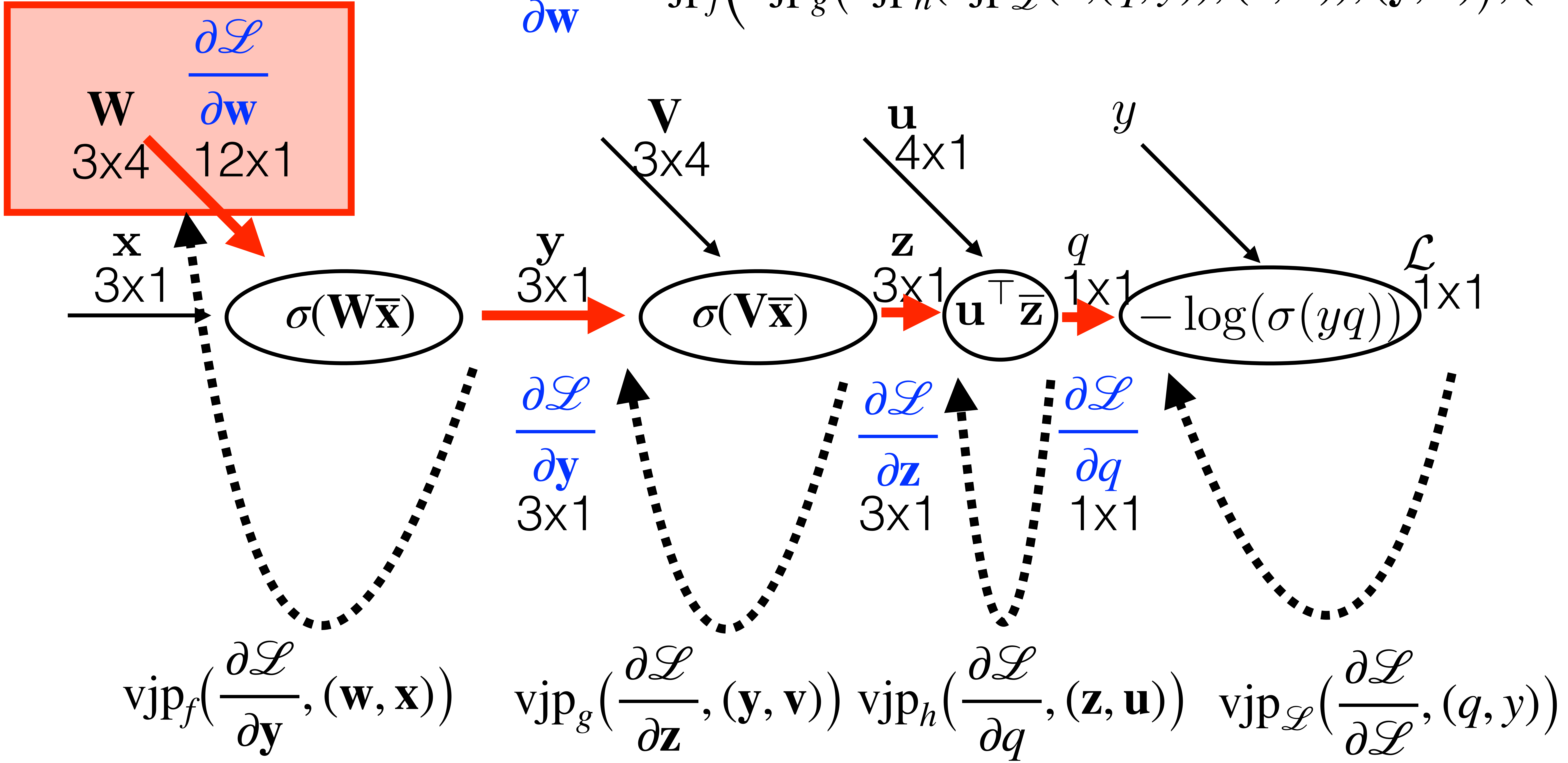
Jacobian wrt $\mathbf{w}$: $\frac{\partial \mathscr{L}}{\partial \mathbf{w}} = \text{vjp}_f\Big(\text{vjp}_g\big(\text{vjp}_h(\text{vjp}_{\mathscr{L}}(1,(q,y)),(\mathbf{z},\mathbf{u})),(\mathbf{y},\mathbf{v})\big),(\mathbf{w},\mathbf{x})\Big)$



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

- preserves dimensionality of inputs in backward pass

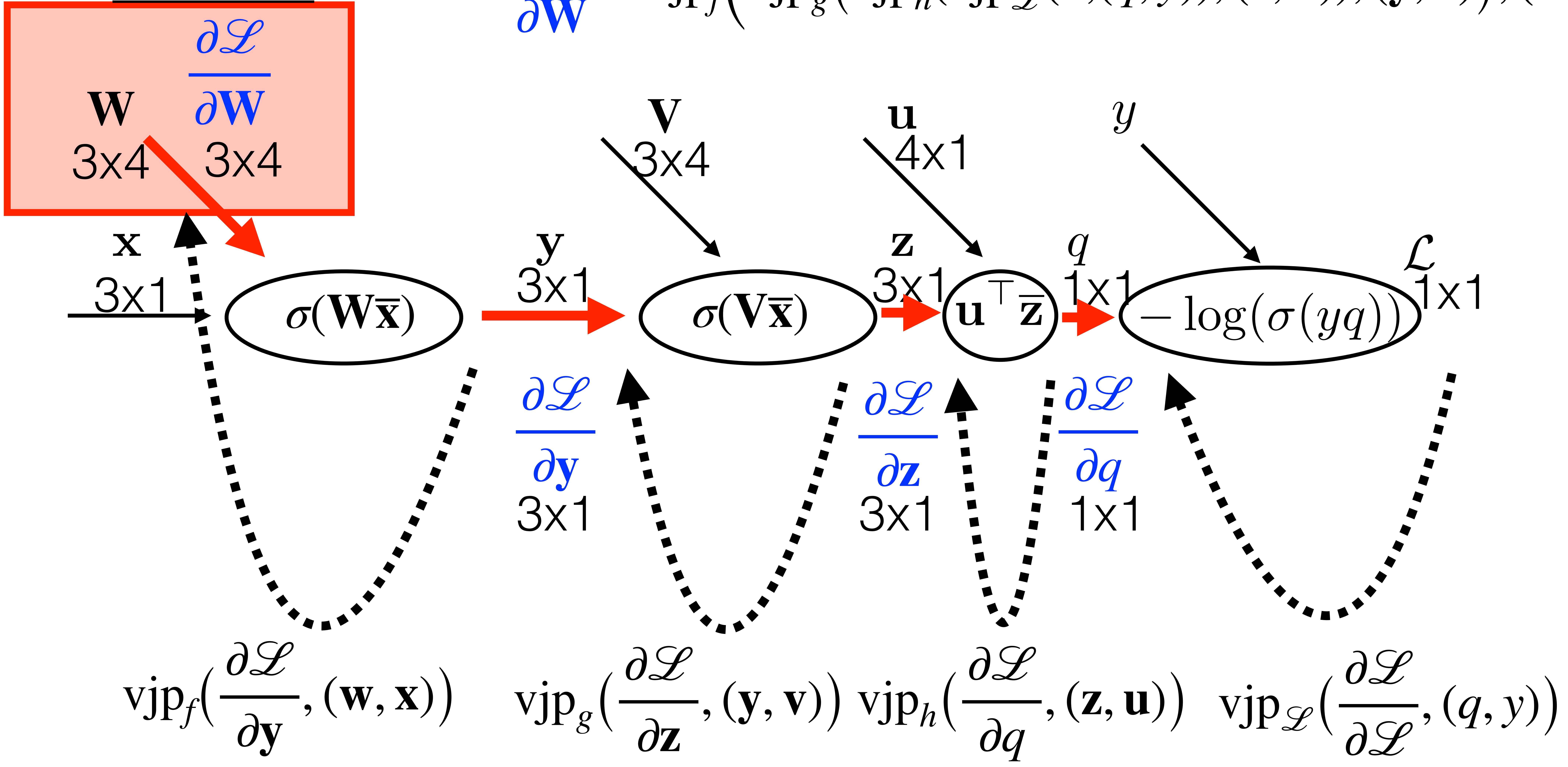
Jacobian wrt $\mathbf{W}$: $\frac{\partial \mathscr{L}}{\partial \mathbf{w}} = \text{vjp}_f\Big(\text{vjp}_g\big(\text{vjp}_h(\text{vjp}_{\mathscr{L}}(1,(q,y)),(\mathbf{z},\mathbf{u})),(\mathbf{y},\mathbf{v})\big),(\mathbf{w},\mathbf{x})\Big)$



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

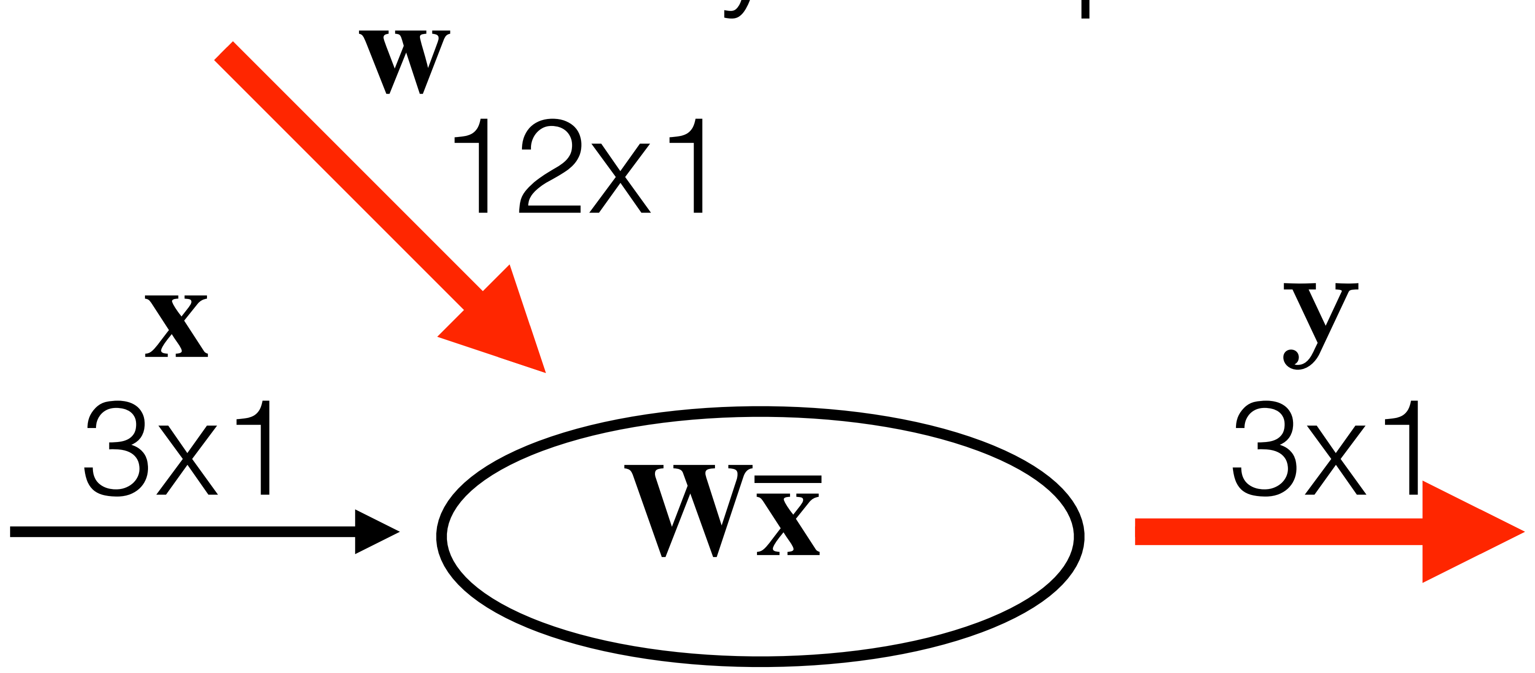
- preserves dimensionality of inputs in backward pass

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



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

- preserves dimensionality of inputs in backward pass



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

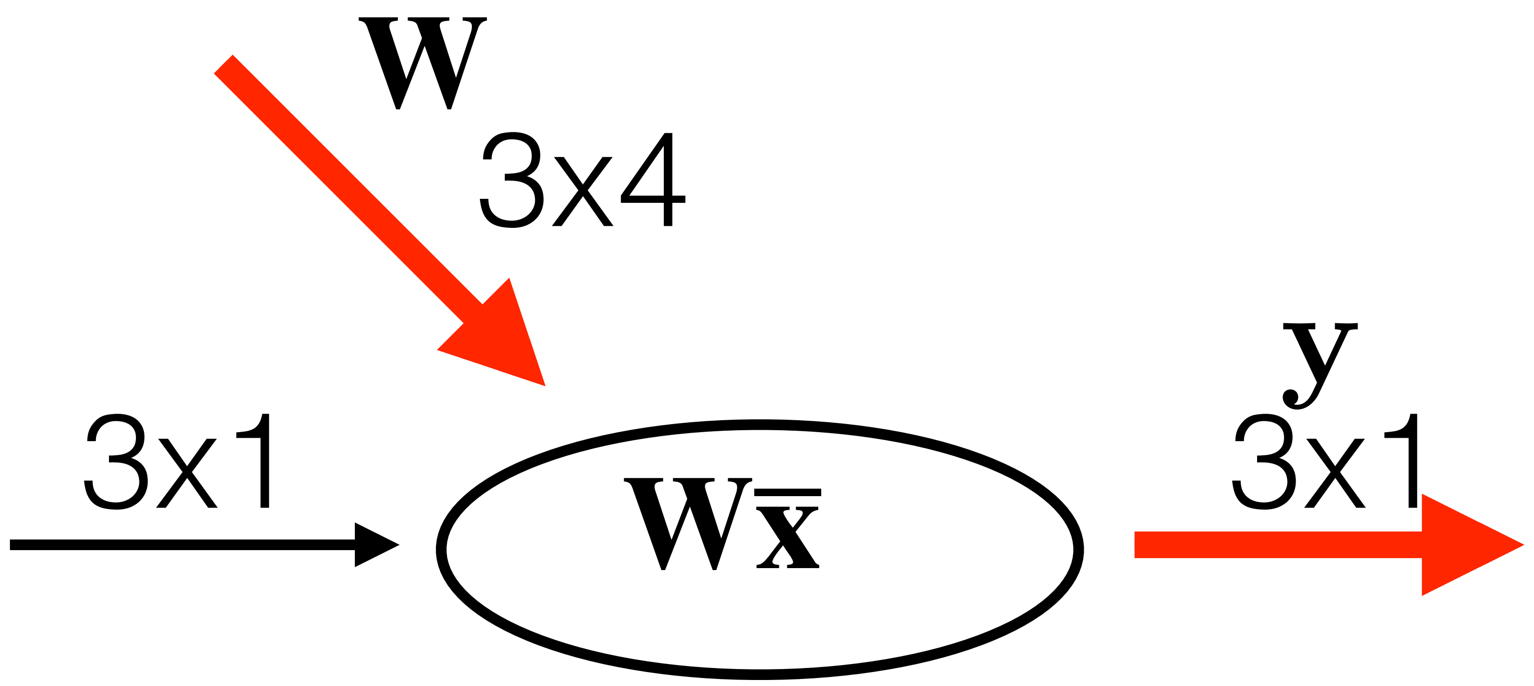
return $\mathbf{v} \cdot \dfrac{\partial \mathbf{y}}{\partial \mathbf{w}} = \mathbf{v} \cdot \begin{bmatrix} -\bar{\mathbf{x}}^\top- & & \\ & -\bar{\mathbf{x}}^\top- & \\ & & -\bar{\mathbf{x}}^\top- \end{bmatrix} = \begin{bmatrix} v_1 \cdot \bar{\mathbf{x}}^\top, & v_2 \cdot \bar{\mathbf{x}}^\top, &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

1x3 3x12 1x12

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

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

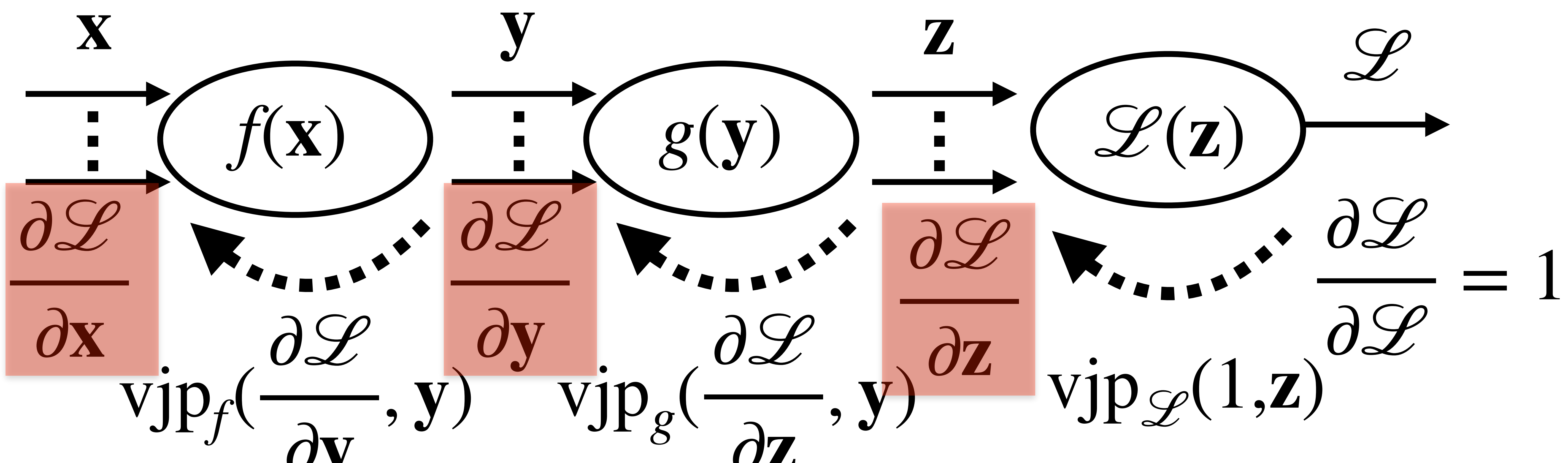


- preserves dimensionality of inputs in backward pass

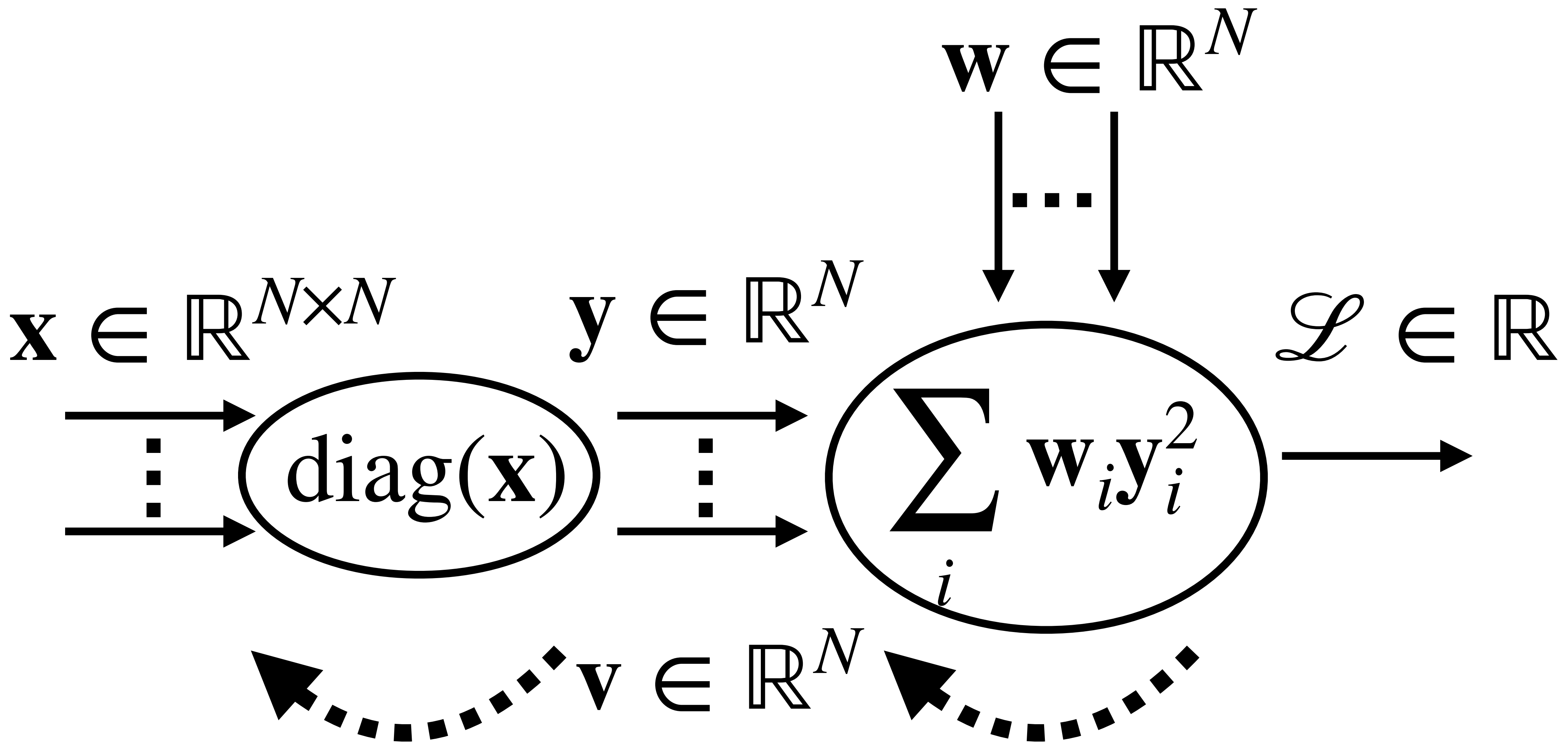
- if $g : \mathbb{R}^{m\times m} \to \mathbb{R}^{n\times n}$, what is its jacobian? $\frac{\partial g(\mathbf{y})}{\partial \mathbf{y}} : \mathbb{R}^{m\times m} \to \mathbb{R}^{n\times n\times m\times m}$
- vjp avoids multiplications of high-dimensional jacobians tensors
- vjp avoids explicit vectorization 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

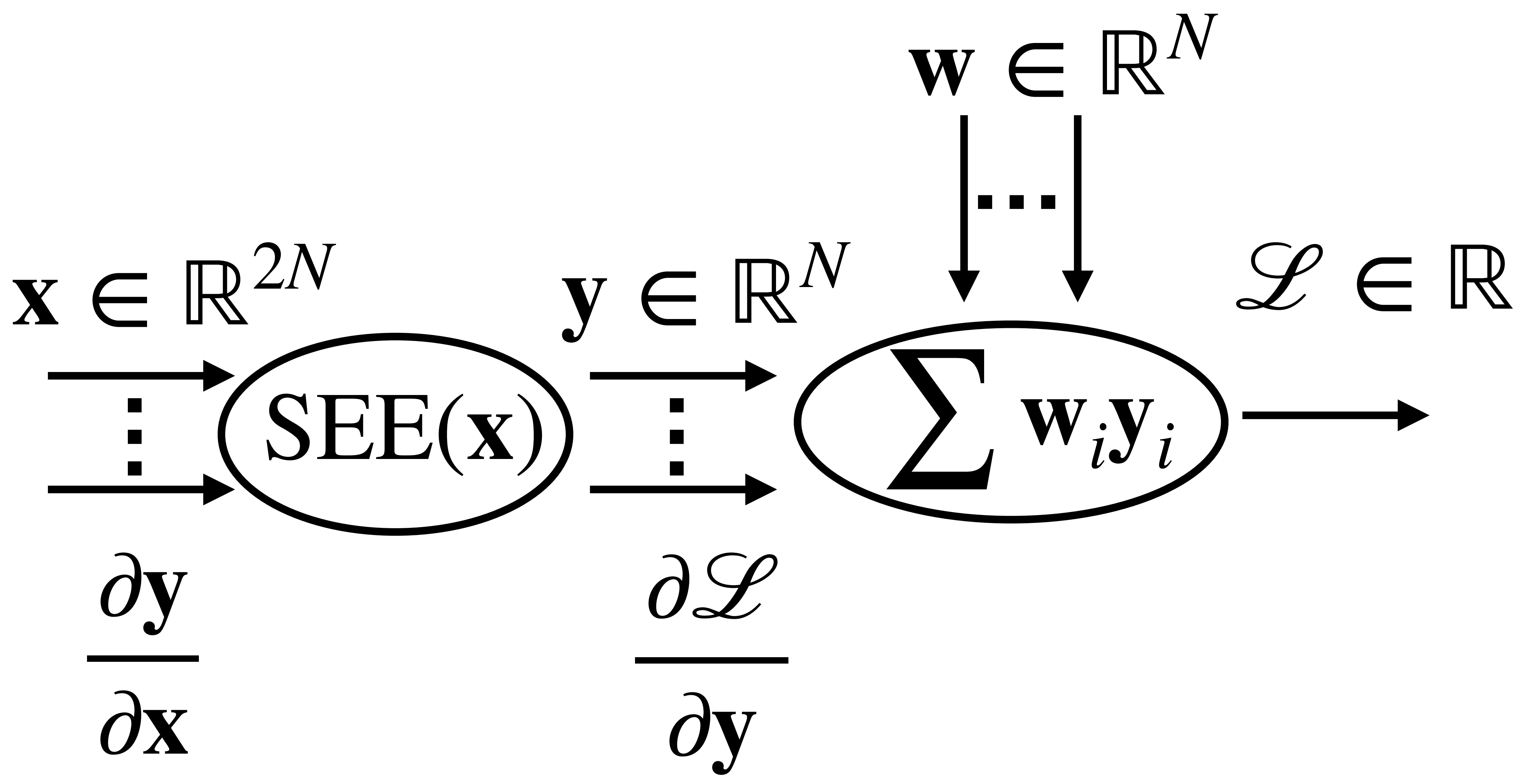


$$\text{vjp}_{\text{diag}}\left(\mathbf{v}\right) = ? \qquad \text{vjp}_{\mathscr{L}}\left(1\right) = ?$$

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

Example: Jacobian vs vector-jacobian-product function

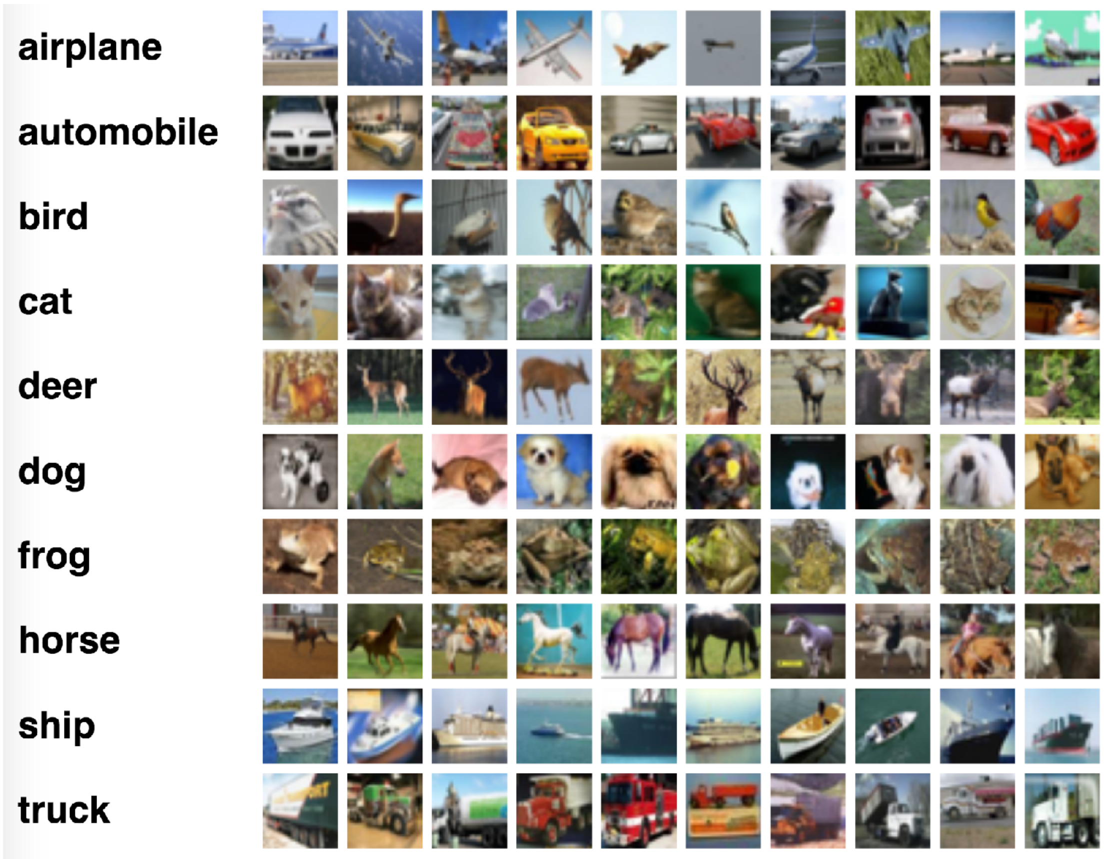
$\mathbf{y} = \text{SEE}(\mathbf{x})$ = Select Even Element from input vector $\mathbf{x}$



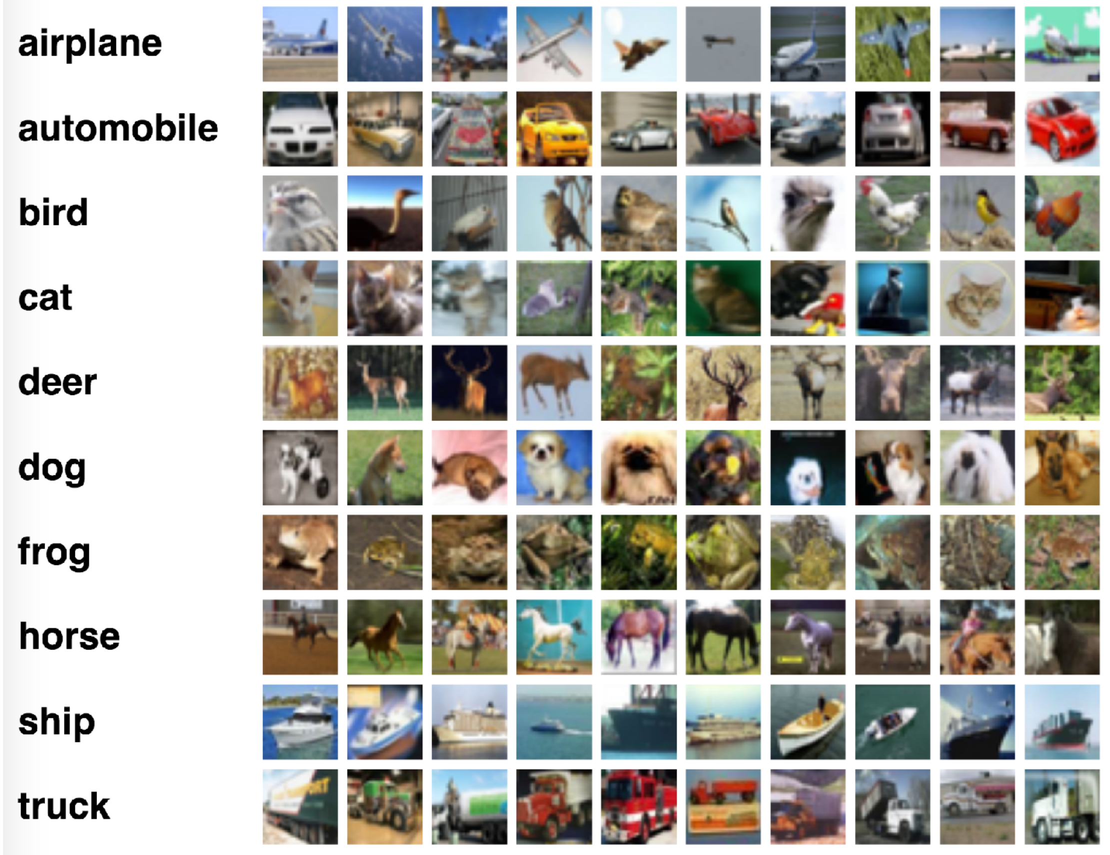
$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = ???$$

Neural nets summary

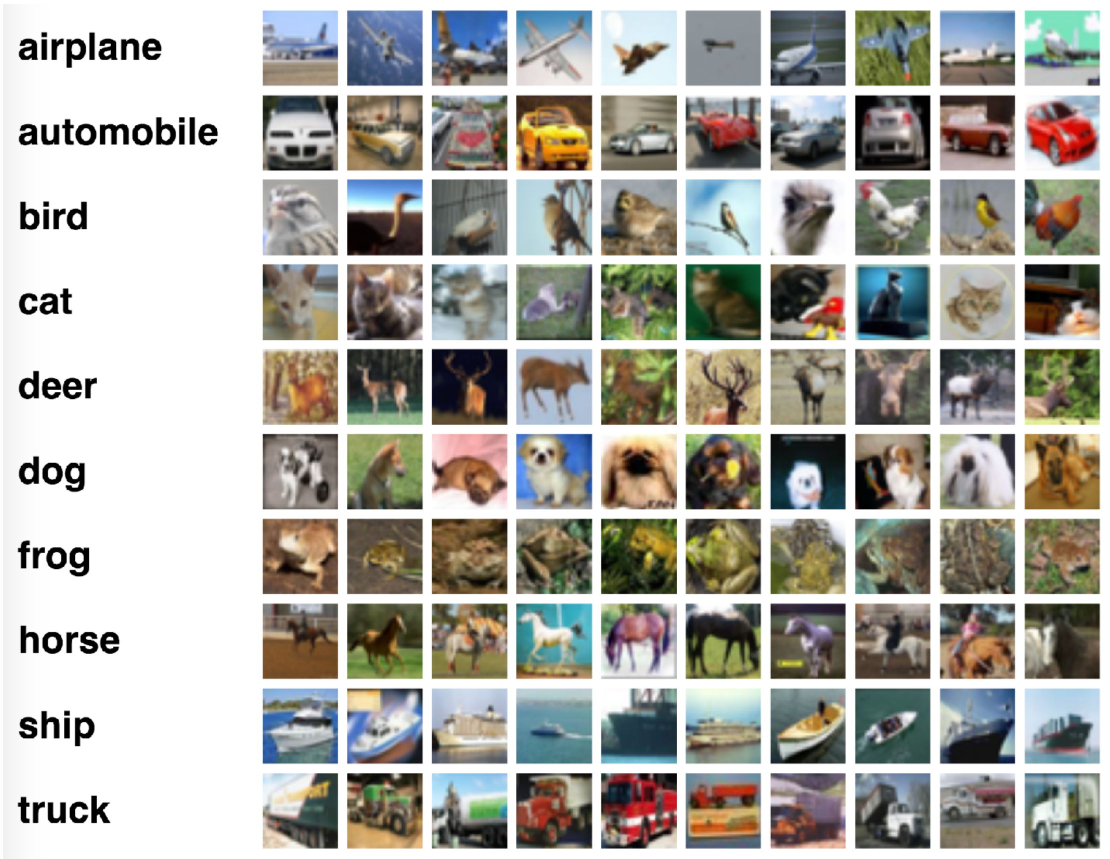
- **Neural net** is a function created as concatenation of simplier functions (e.g. neurons or layers of neurons)
- **Fully connected neural net** is neural network created from neurons, where **all** outputs from previous layer are connected to **all** inputs of the following layer
- **Learning** is gradient optimization of a neural net concatenated with a loss-layer on a training set.
- **Gradient** evaluation is implemented as backward concatenation of 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
- **Deep learning frameworks** (Pytorch, Tensorflow, Caffe) has many predefined layers and takes care about the efficient 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	Error: FCNN
MNIST		8%	**??**
CIFAR-10		63%	**??**

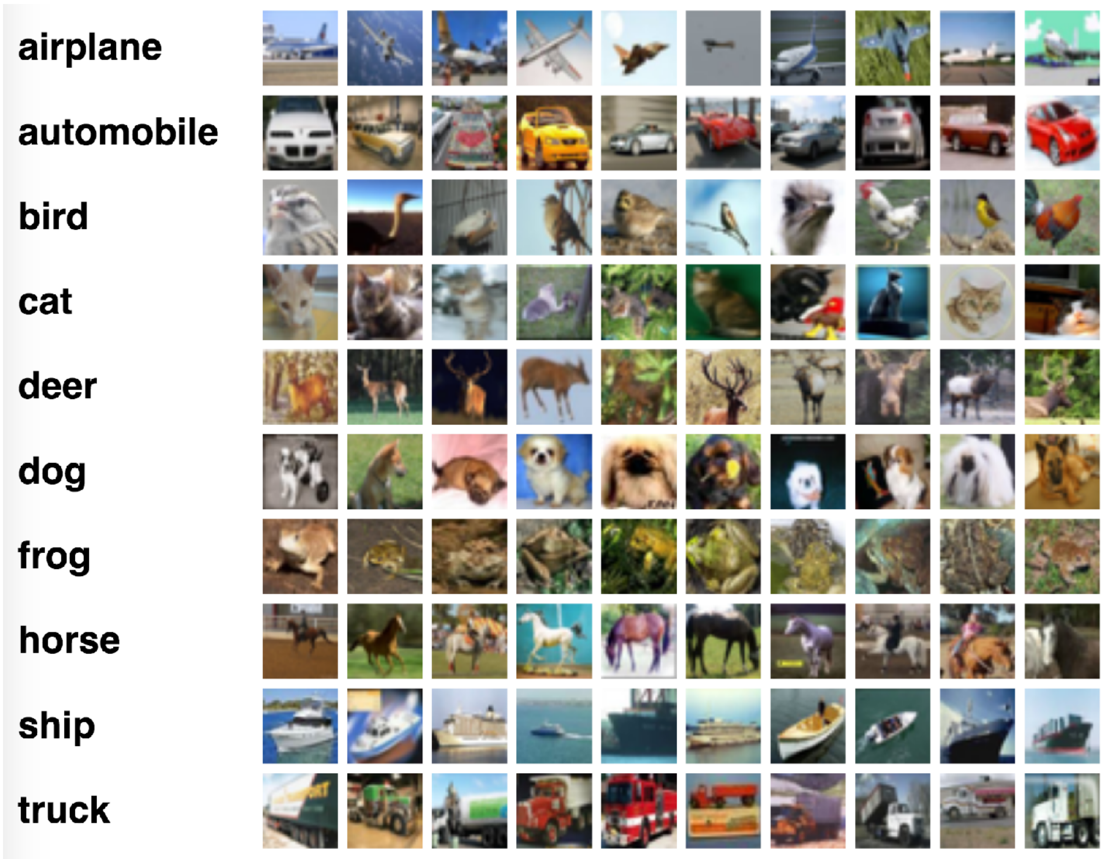
https://benchmarks.ai

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

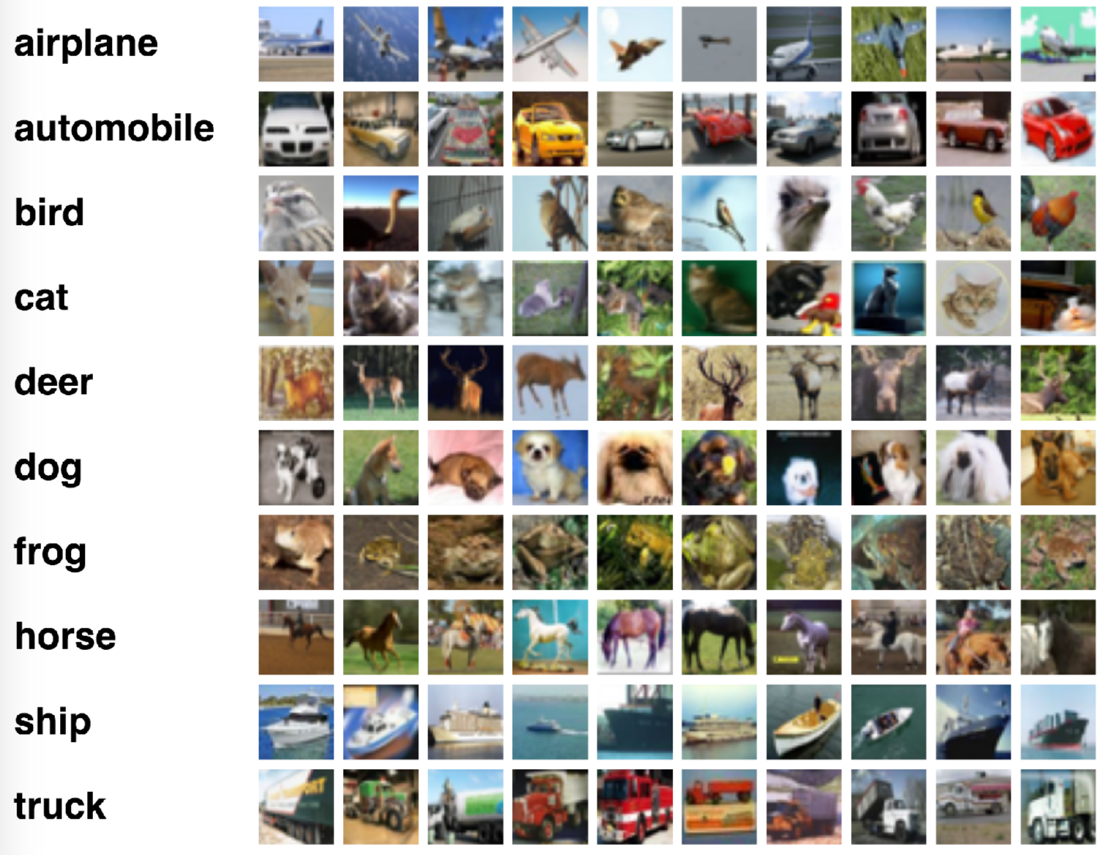
https://benchmarks.ai

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

https://benchmarks.ai

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

https://benchmarks.ai

Dataset		Error: Linear	Error: FCNN	Error: ConvNet
MNIST		8%	2%	0.2% [CVPR 2013]
		underfit	overfit	good fit ;-)
		too simple arch.	too complex arch.	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