Fully connected networks

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

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

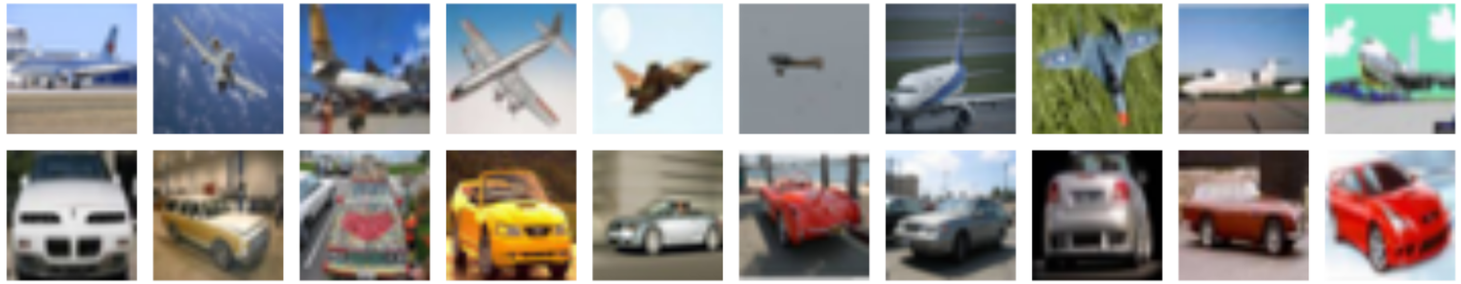
Linear classifier and neuron

Labels

RGB images

$+1$

-1



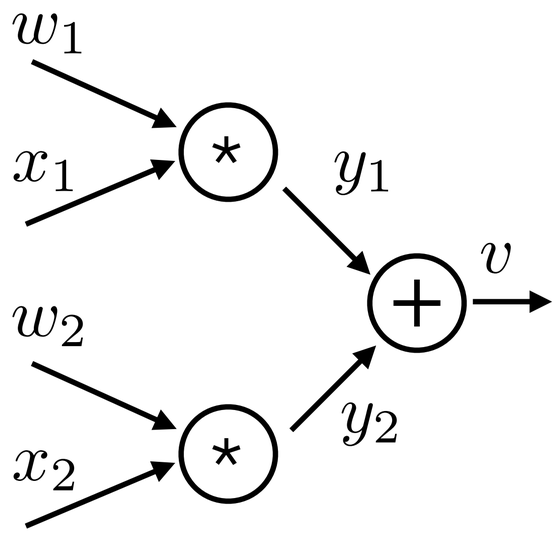
def classify():

 # Linear classifier

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

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

Computational graph of linear classifier



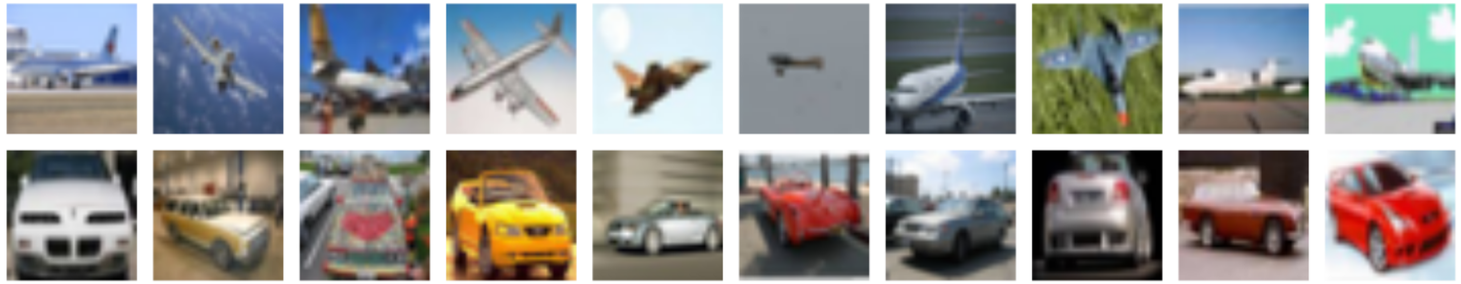
Linear classifier and neuron

Labels

RGB images

$+1$

-1



def classify():

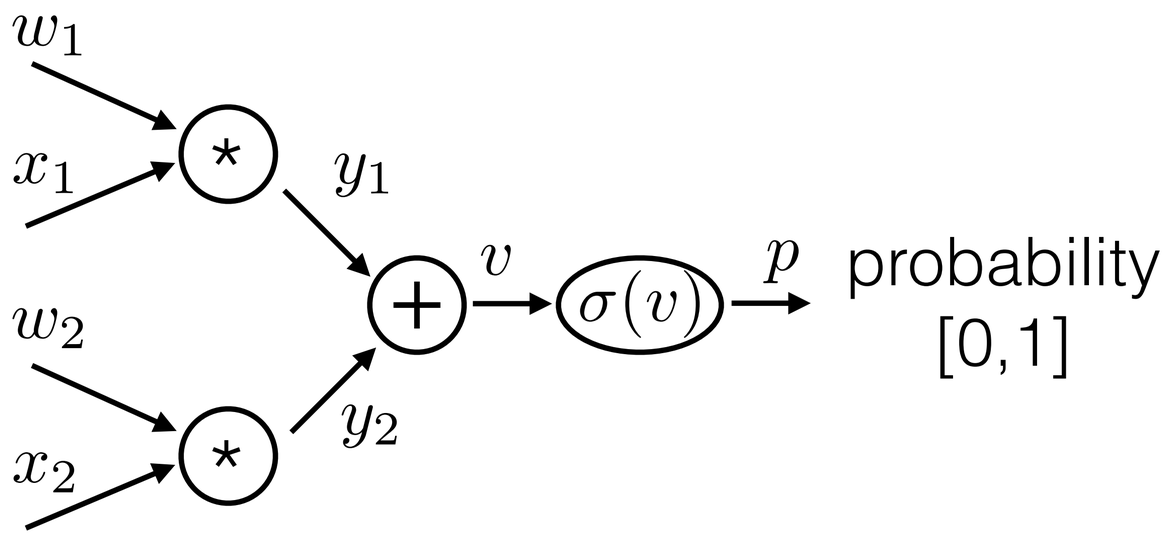
 # Neuron

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

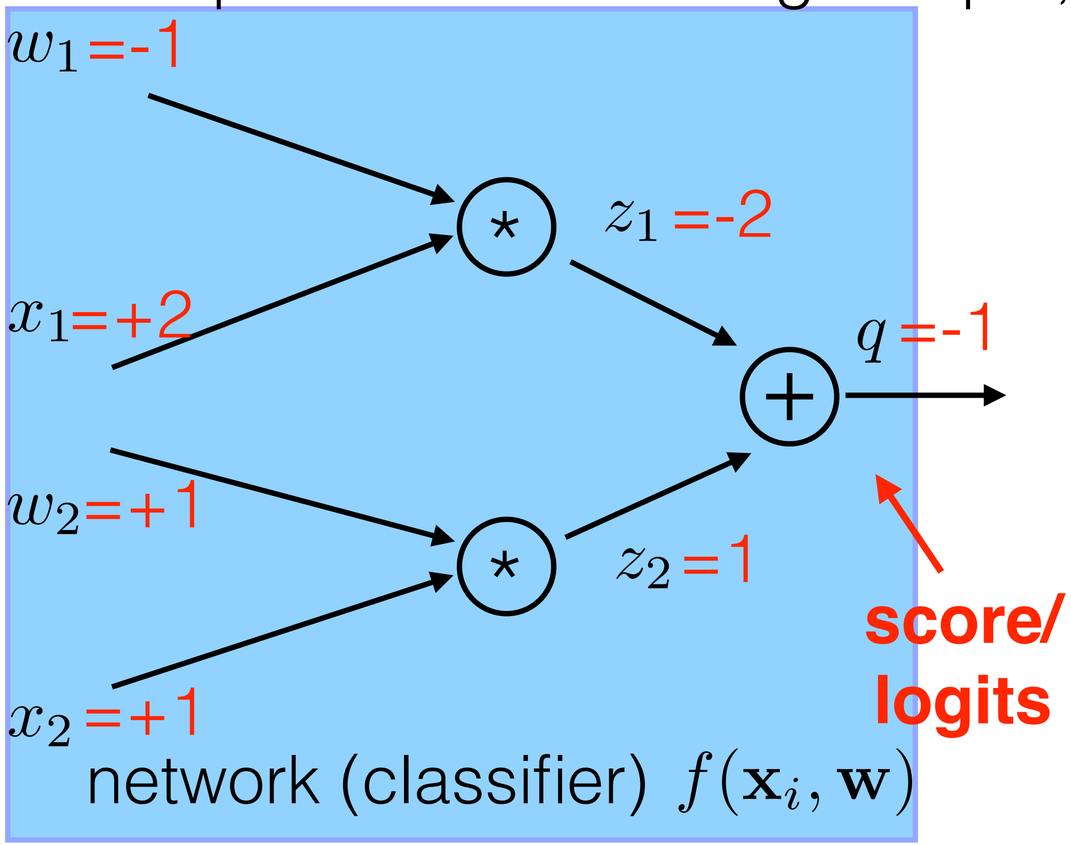
$p = \sigma\left(\mathbf{w}^\top \mathbf{x}\right)$

return p

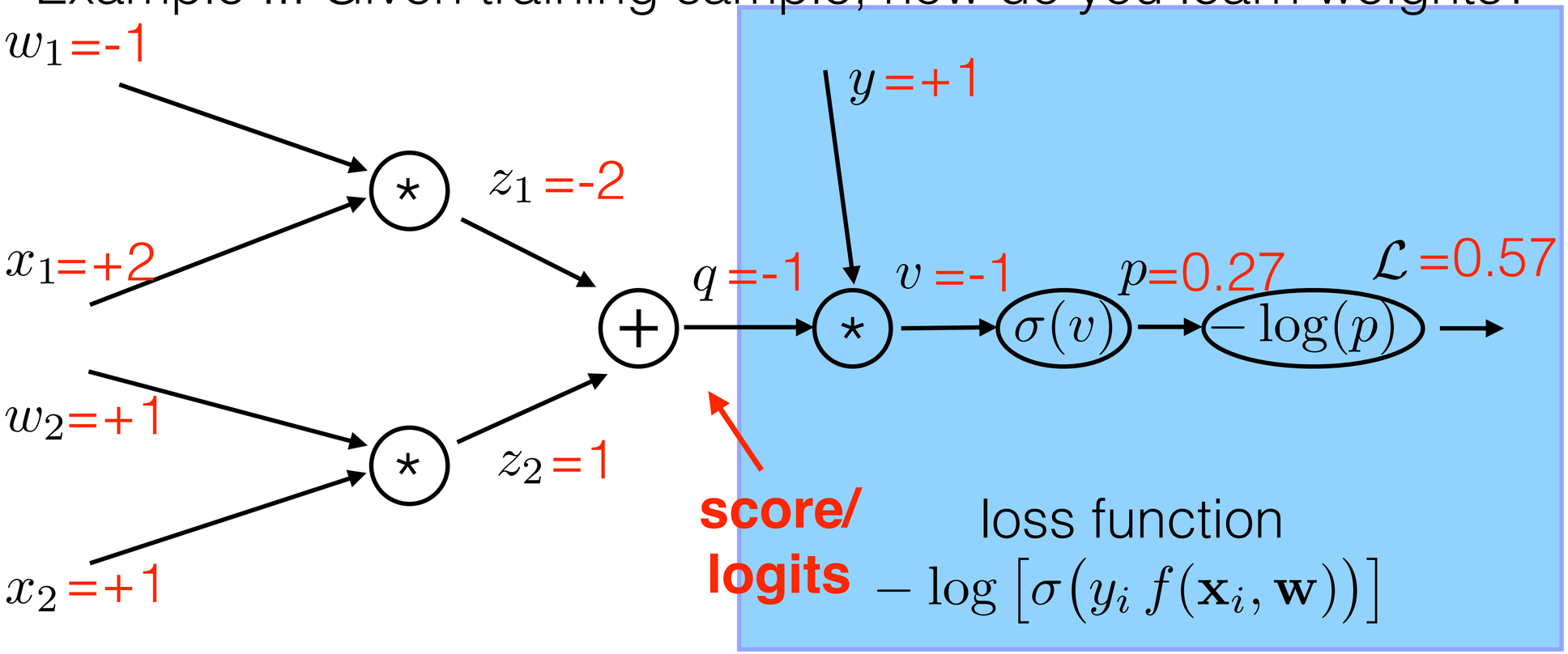
Computational graph of neuron



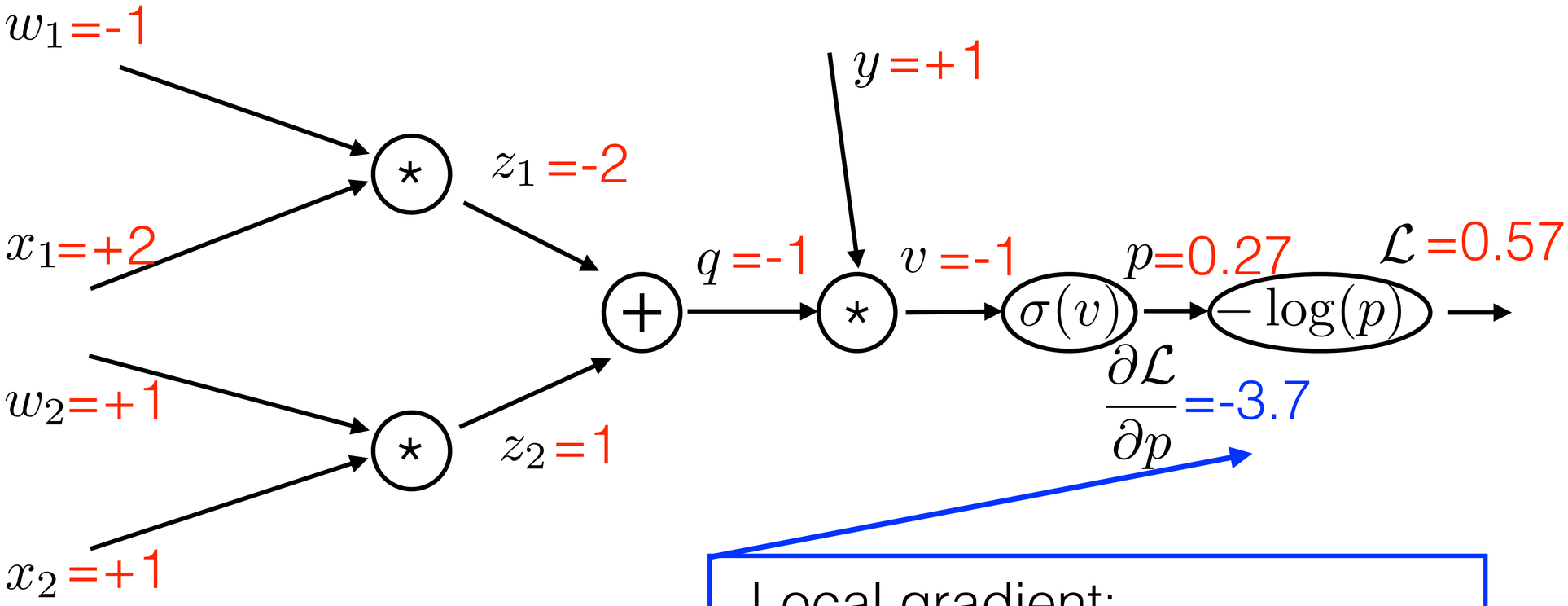
Example II: Given training sample, how do you learn weights?



Example II: Given training sample, how do you learn weights?



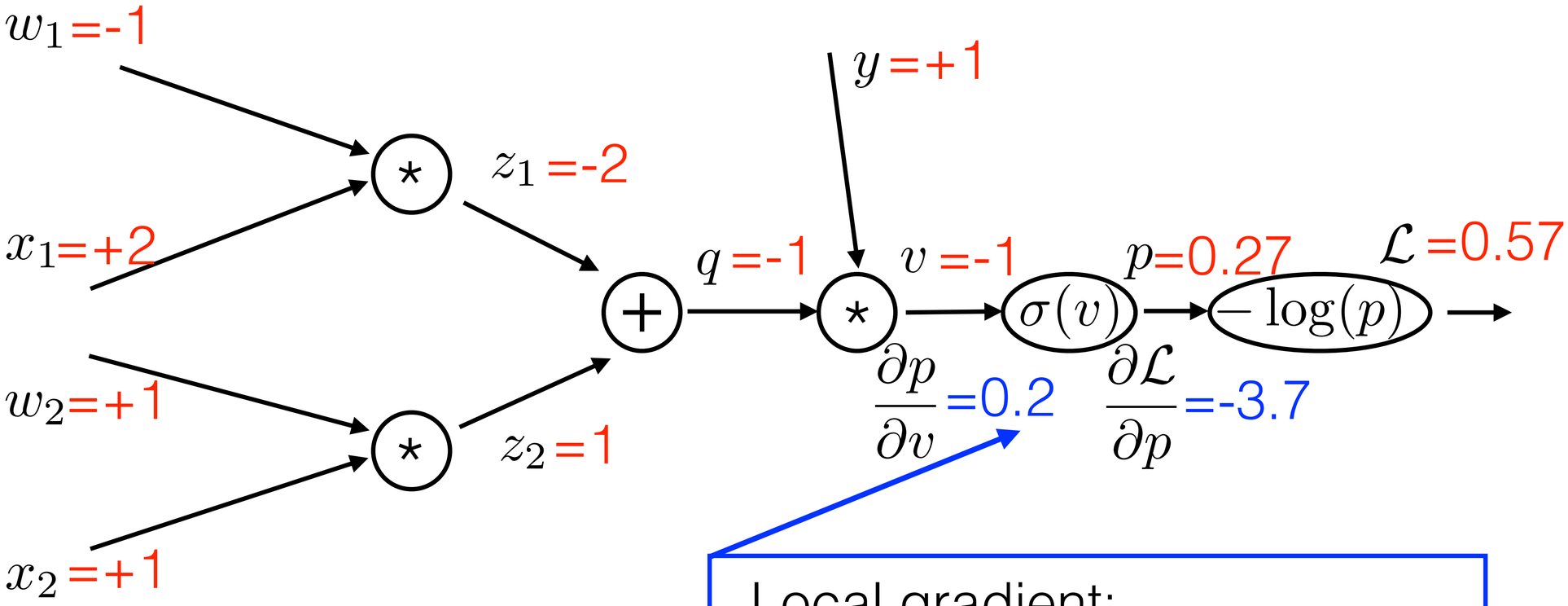
Example II: Given training sample, how do you learn weights?



Local gradient:

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

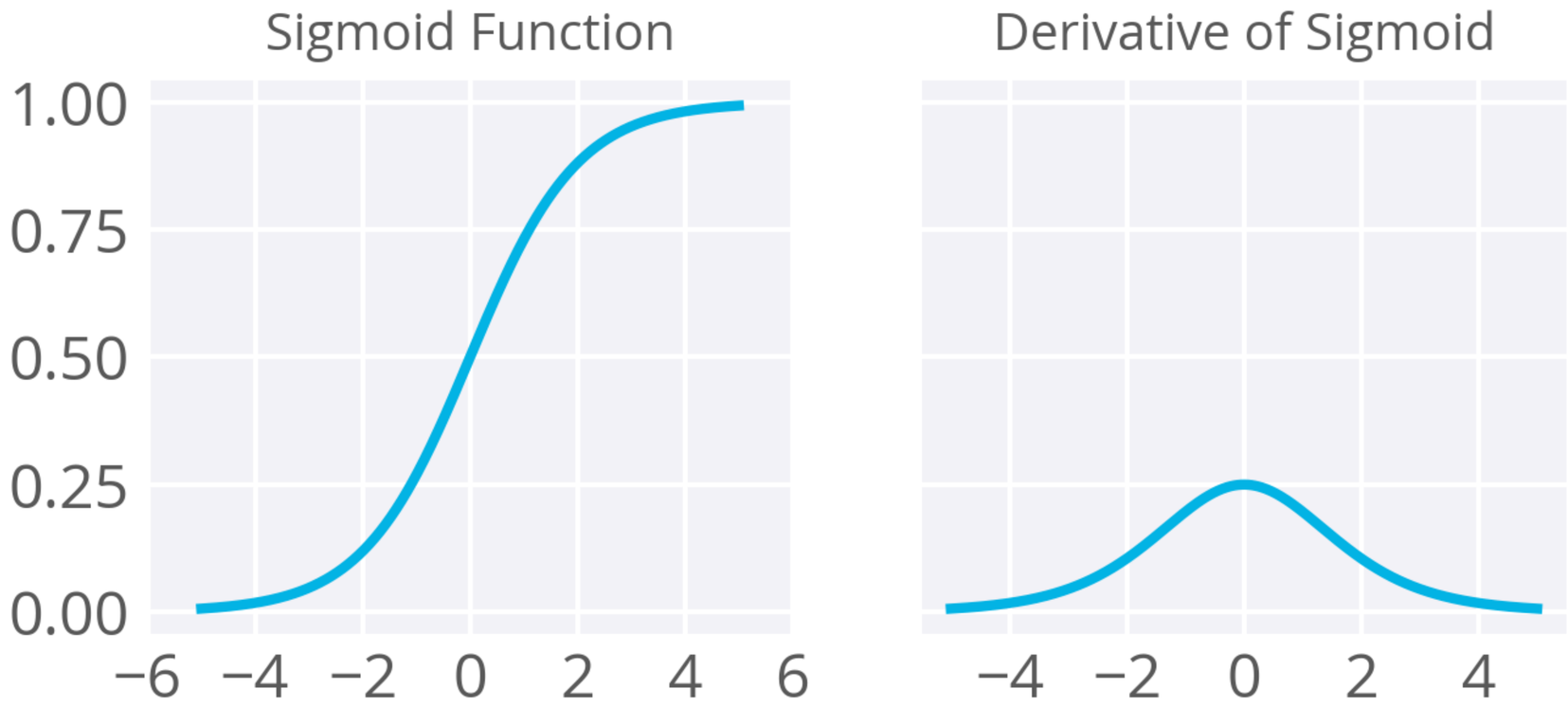
Example II: Given training sample, how do you learn weights?



Local gradient:

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

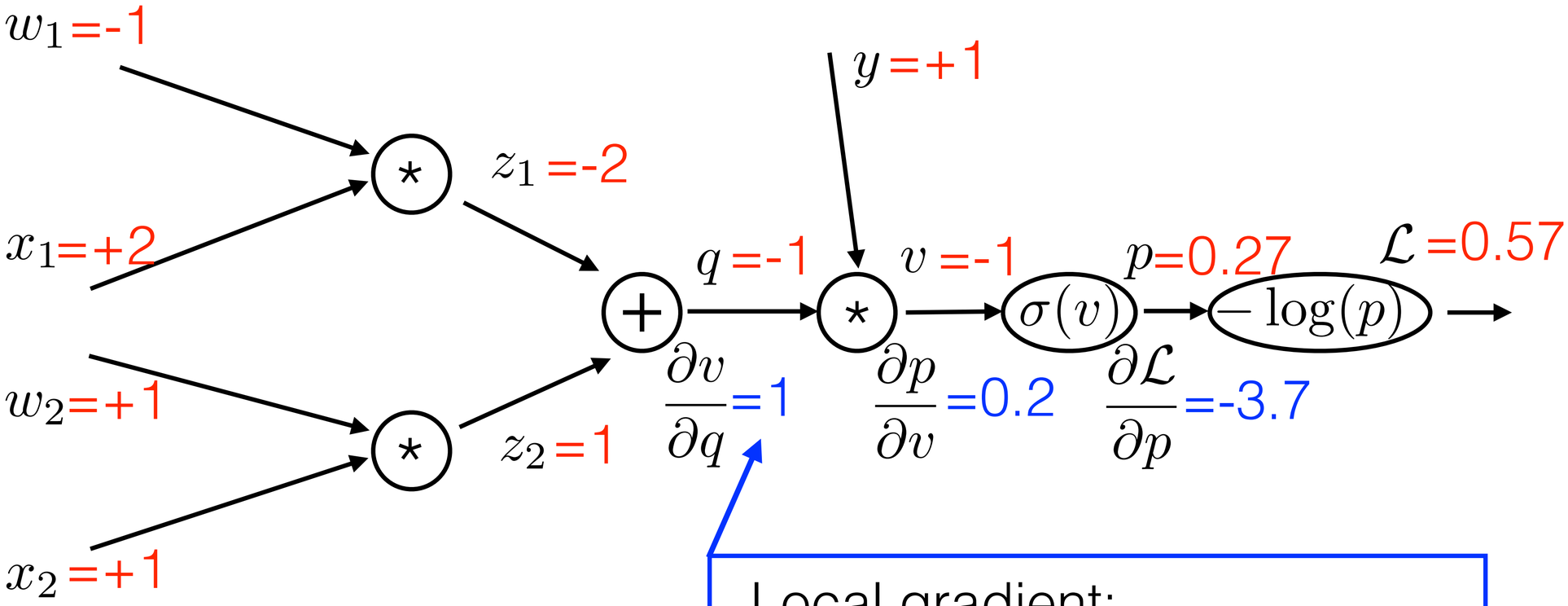
Example II: Given training sample, how do you learn weights?



Local gradient:

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

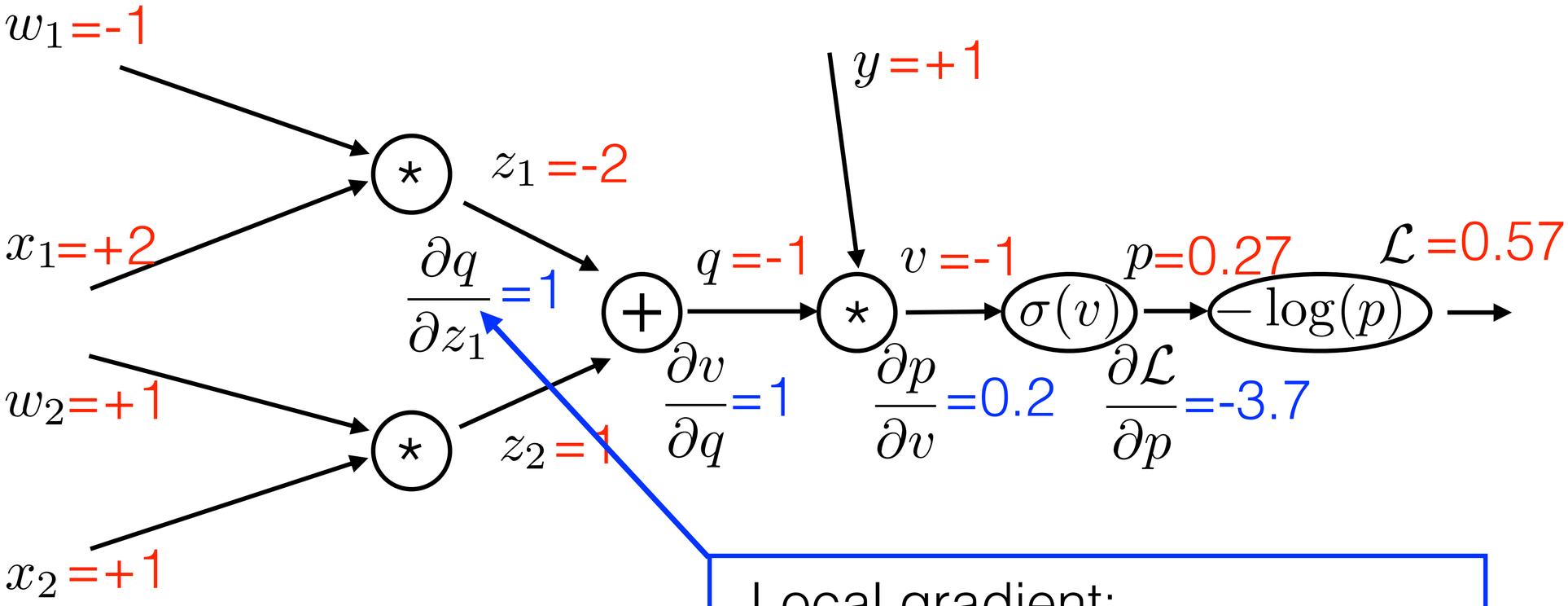
Example II: Given training sample, how do you learn weights?



Local gradient:

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

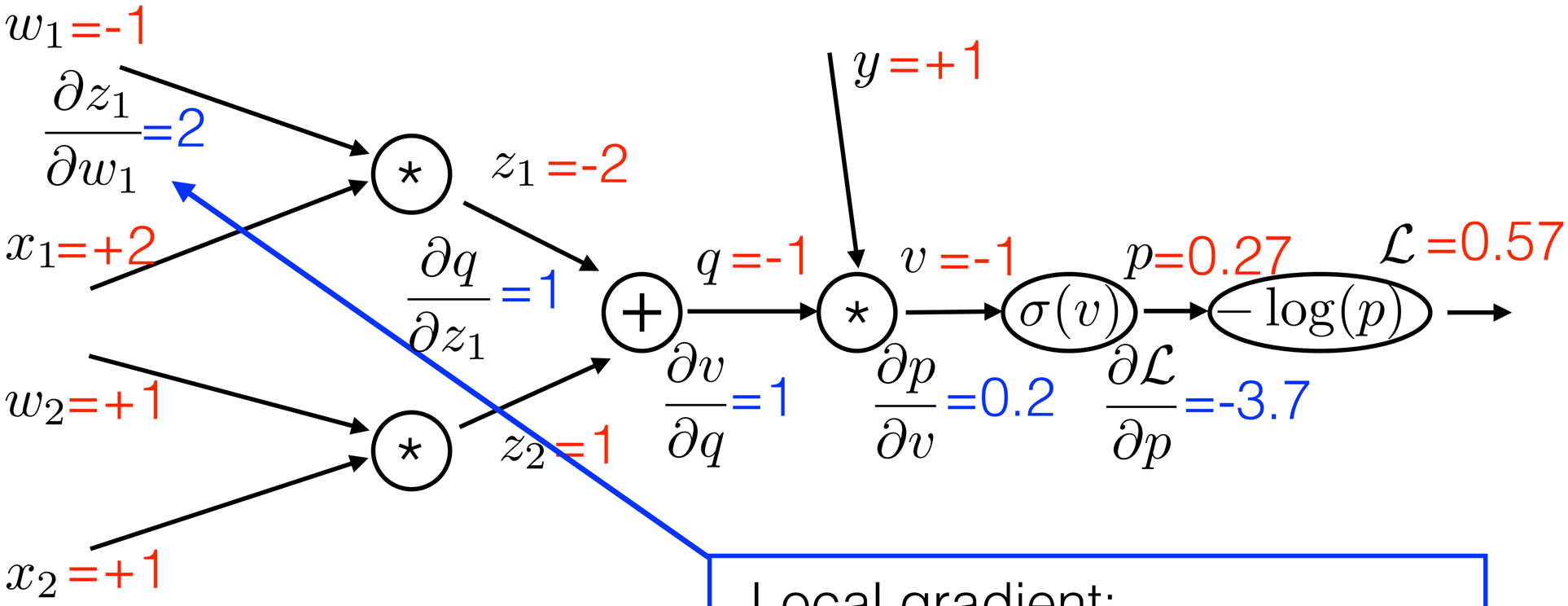
Example II: Given training sample, how do you learn weights?



Local gradient:

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

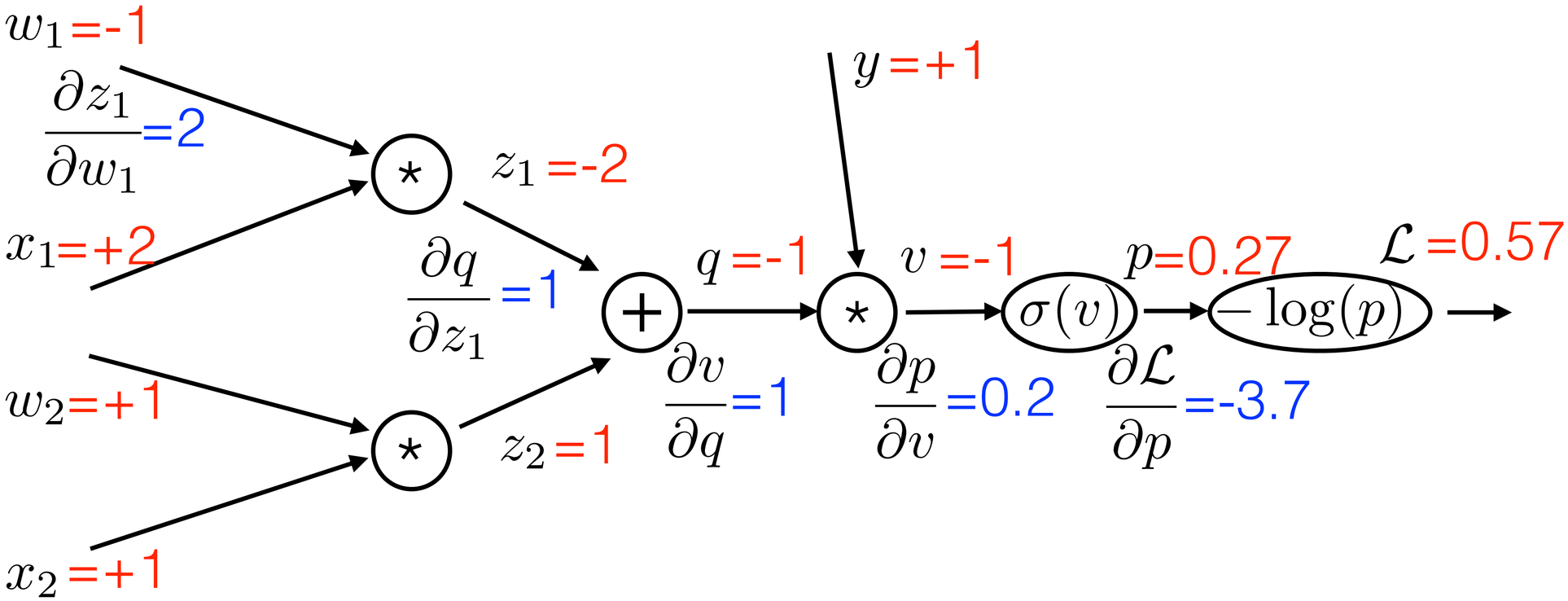
Example II: Given training sample, how do you learn weights?



Local gradient:

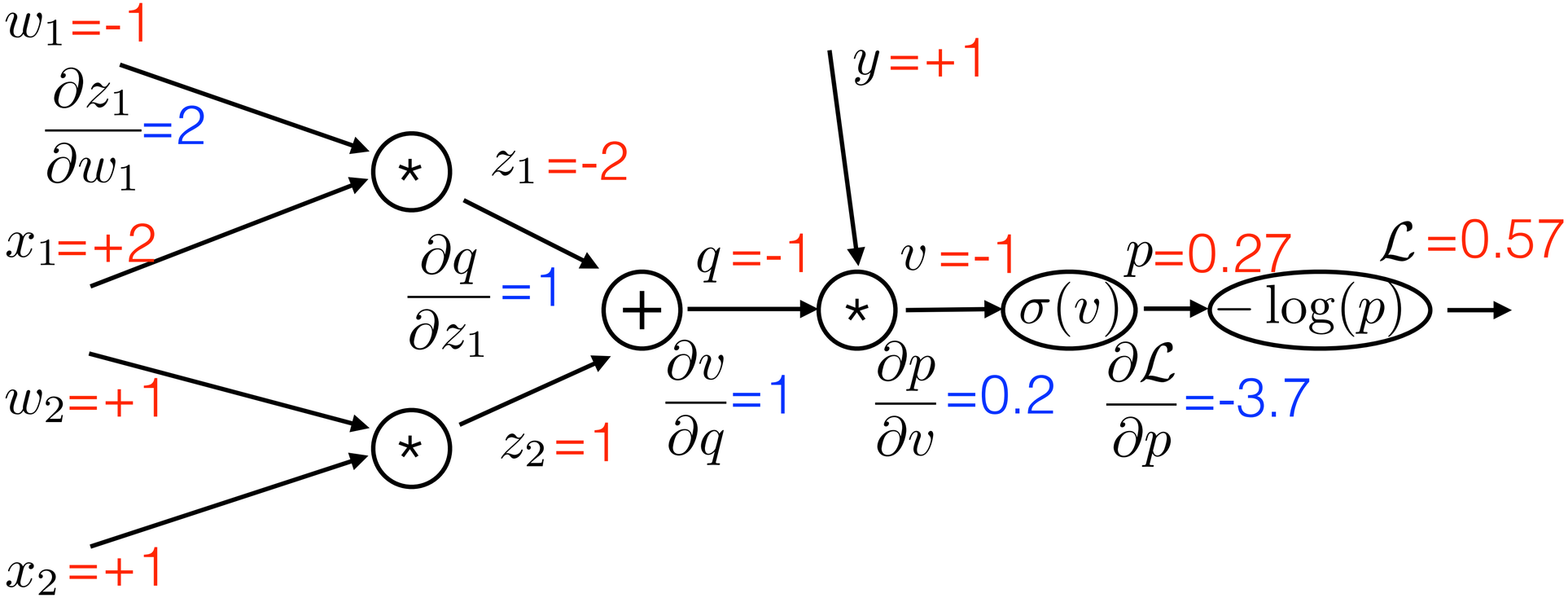
$$\frac{\partial z_1}{\partial w_1} = \frac{\partial (w_1 x_1)}{\partial w_1} = x_1$$

Example II: Given training sample, how do you learn weights?



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

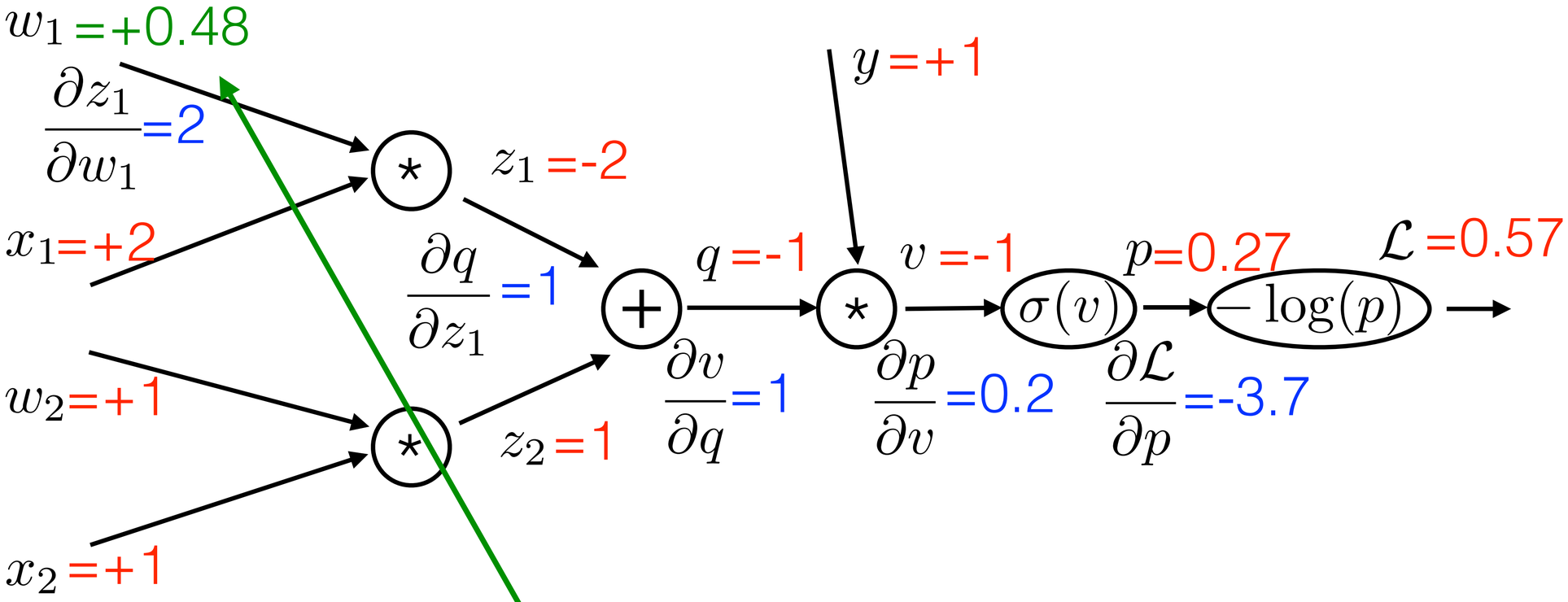
Example II: Given training sample, how do you learn weights?



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

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

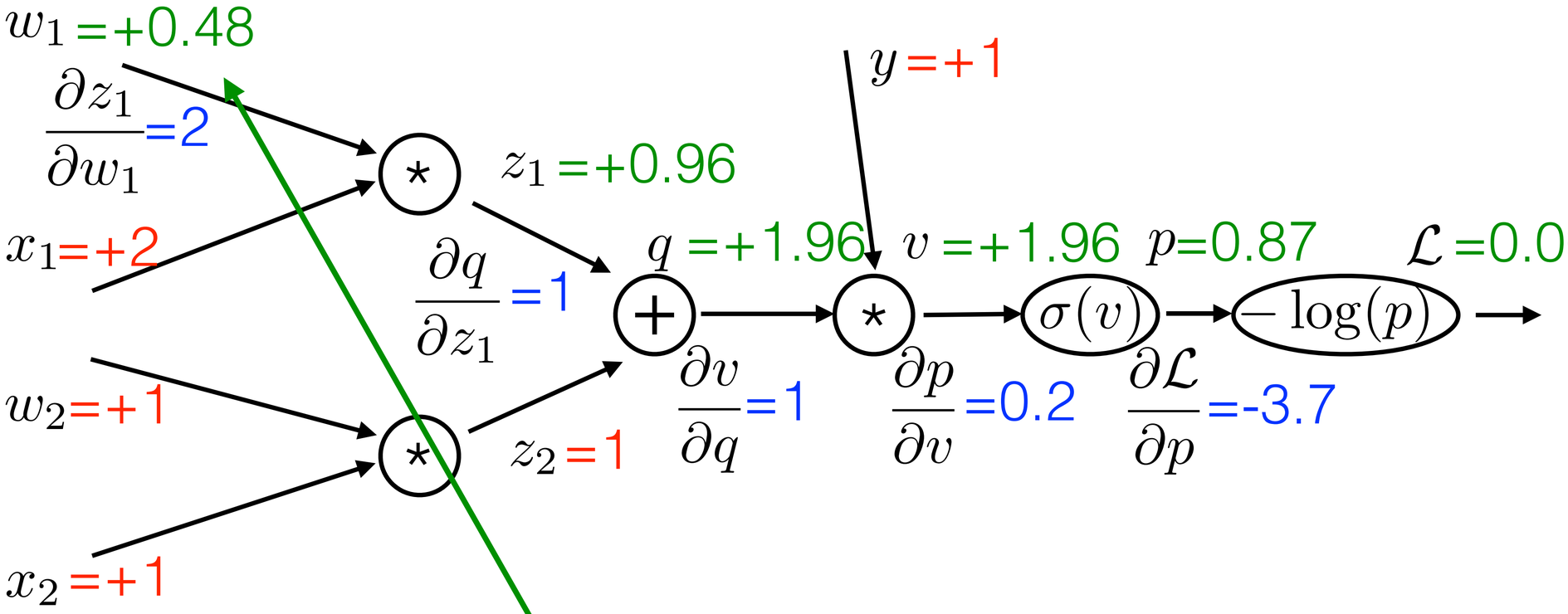
Example II: Given training sample, how do you learn weights?



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

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

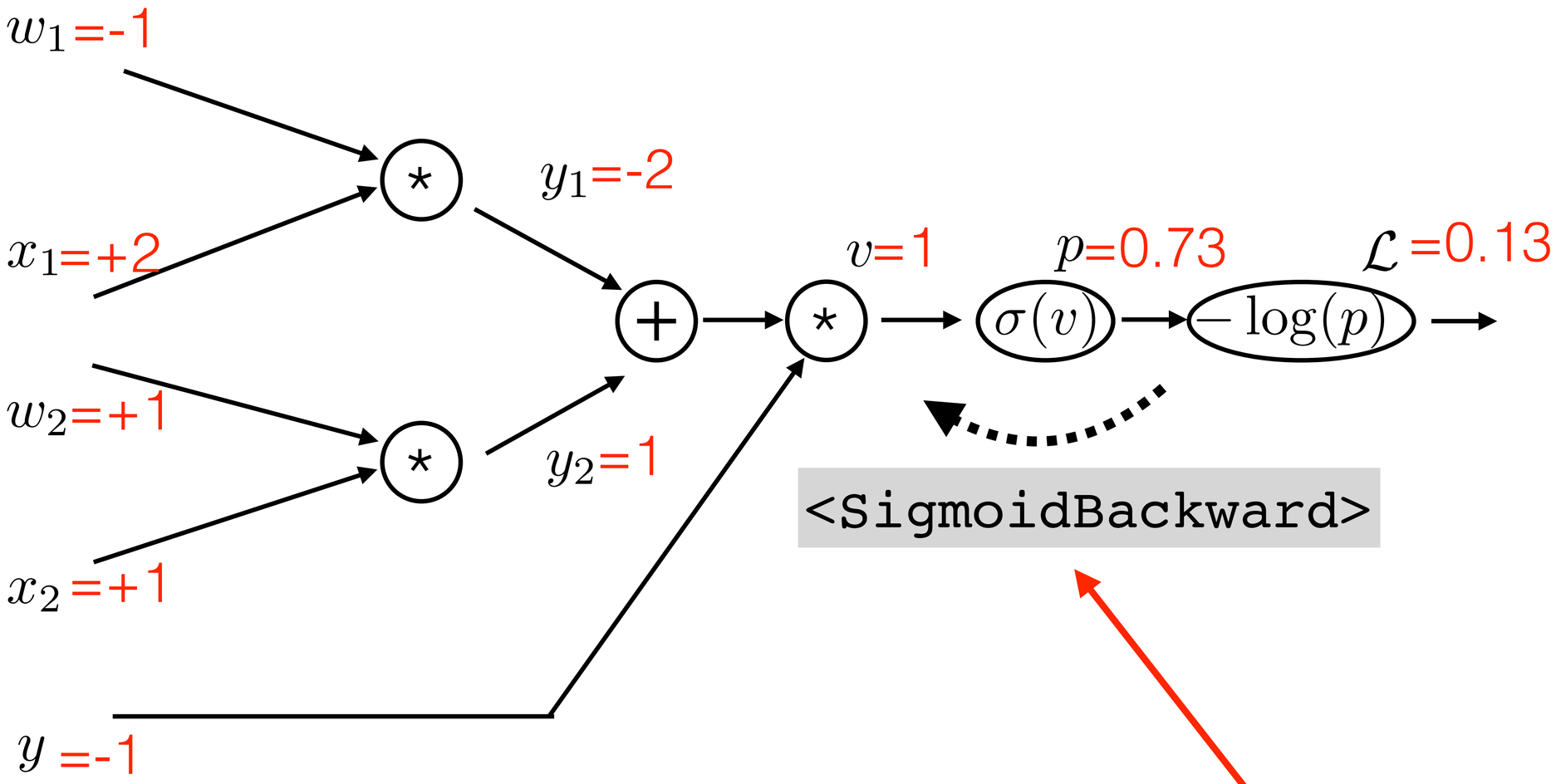
Example II: Given training sample, how do you learn weights?



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

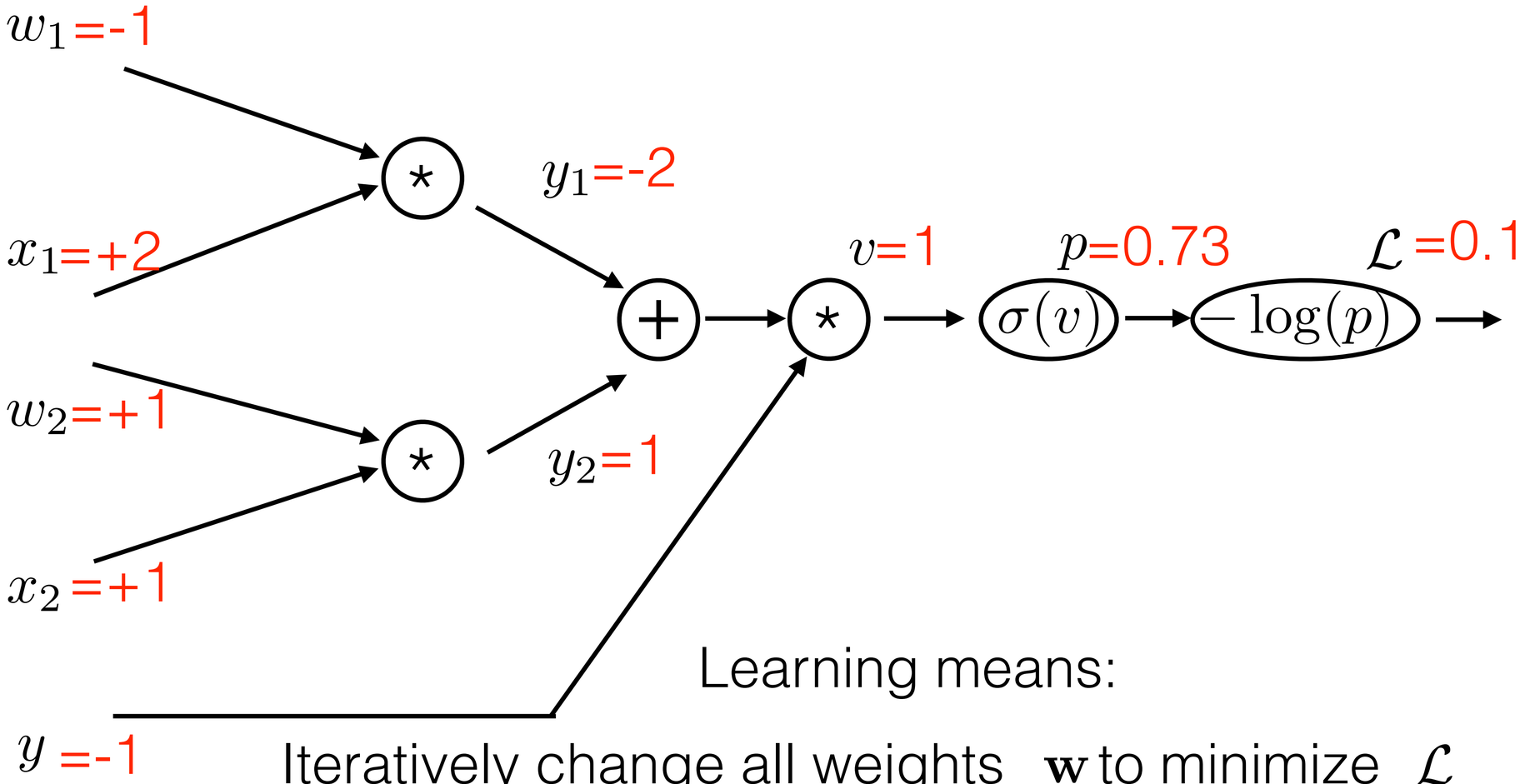
$$w_1 = w_1 - \alpha\frac{\partial \mathcal{L}}{\partial w1} \text{ = +0.48}$$

PyTorch representation




```
p = torch.sigmoid(v)
 tensor(0.7311, dtype=torch.float64, grad_fn=<SigmoidBackward>)
```

Example III: vector representation

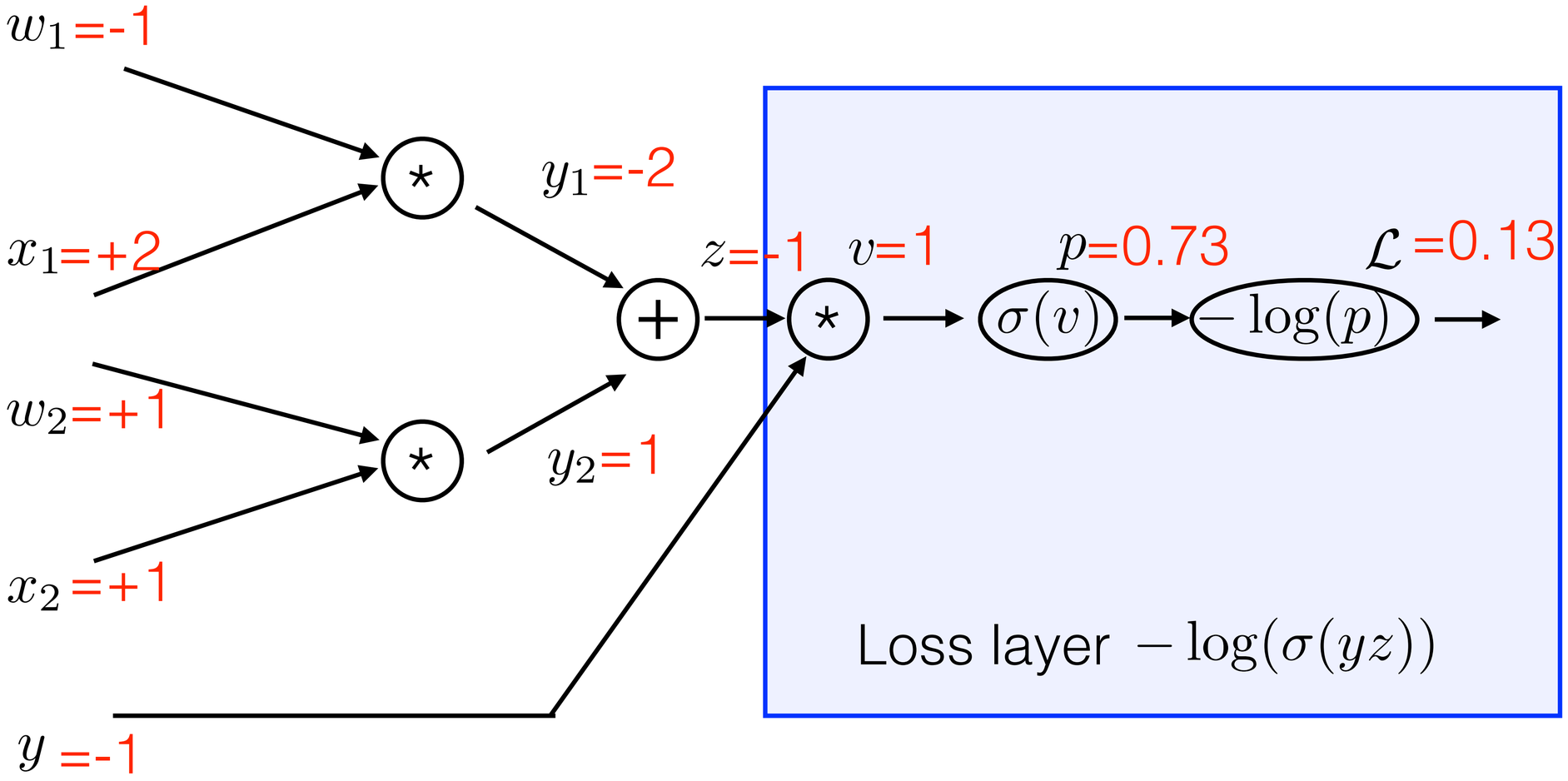


Learning means:

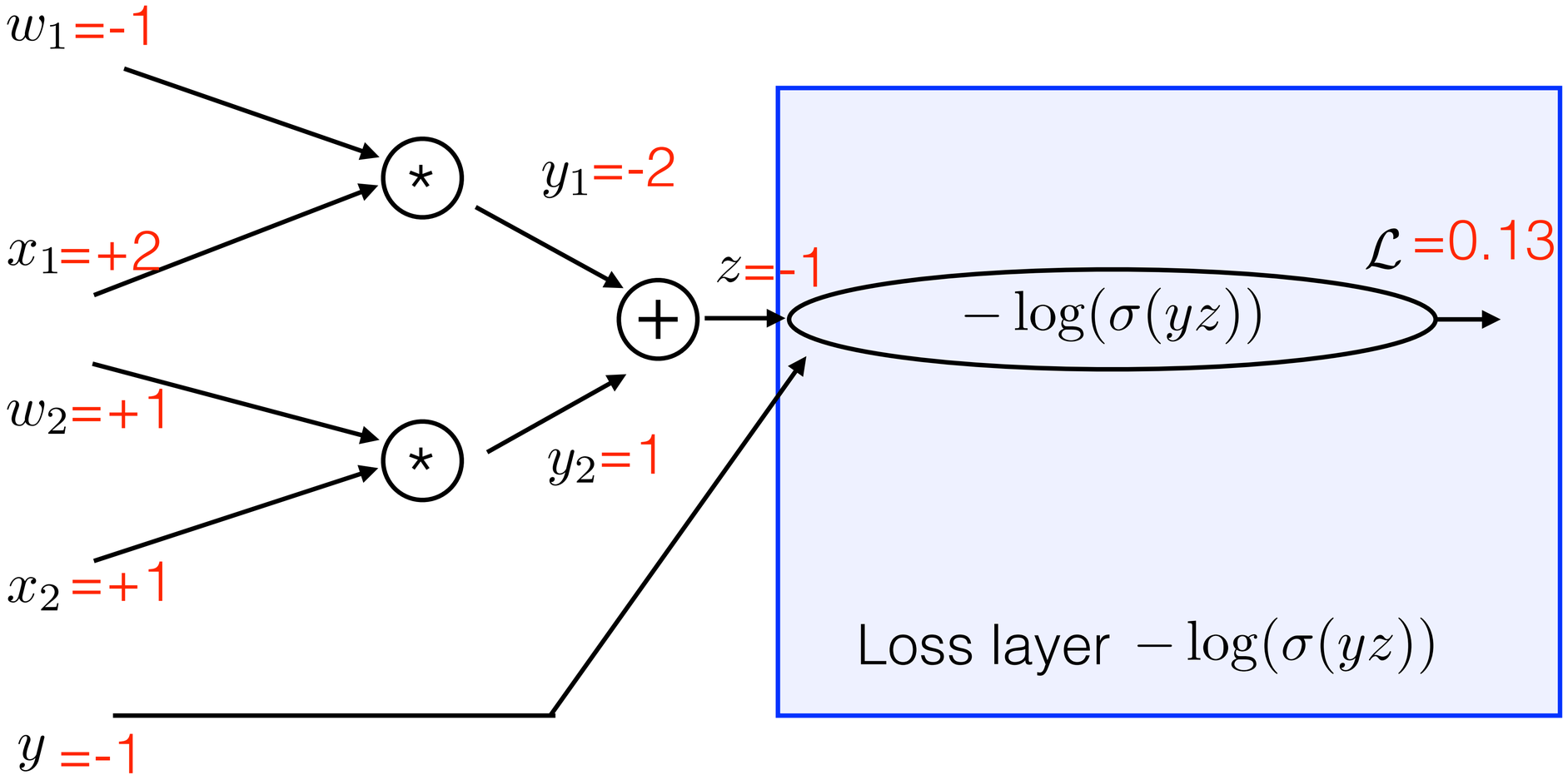
Iteratively change all weights $\mathbf{w}$ to minimize $\mathcal{L}$

$$\mathbf{w} = \mathbf{w} - \alpha \left[\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}}\right]^\top \quad \text{where} \quad \frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \left[\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \dots\right]$$

Computational graph of the learning



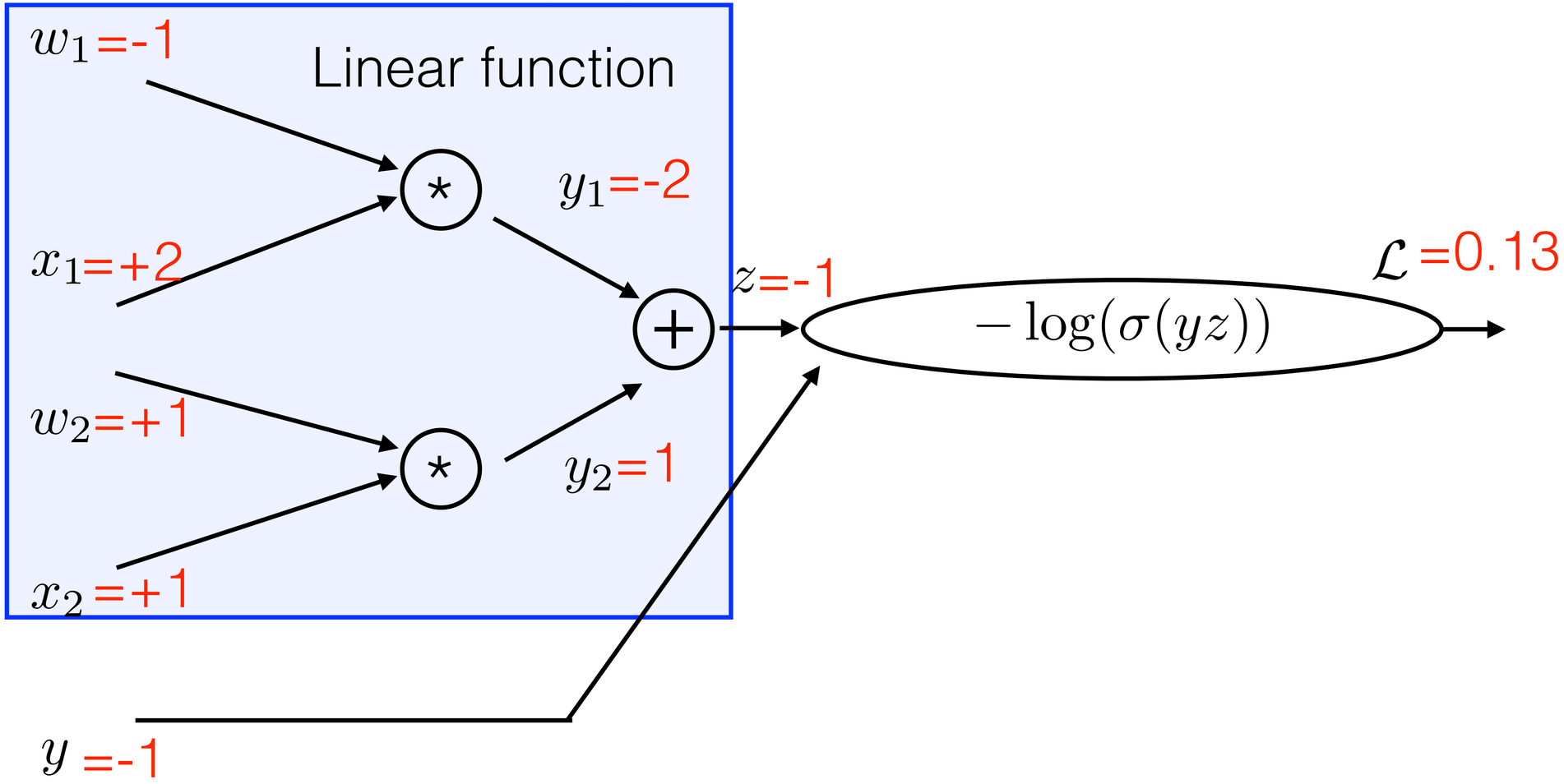
Backprop in vector representation



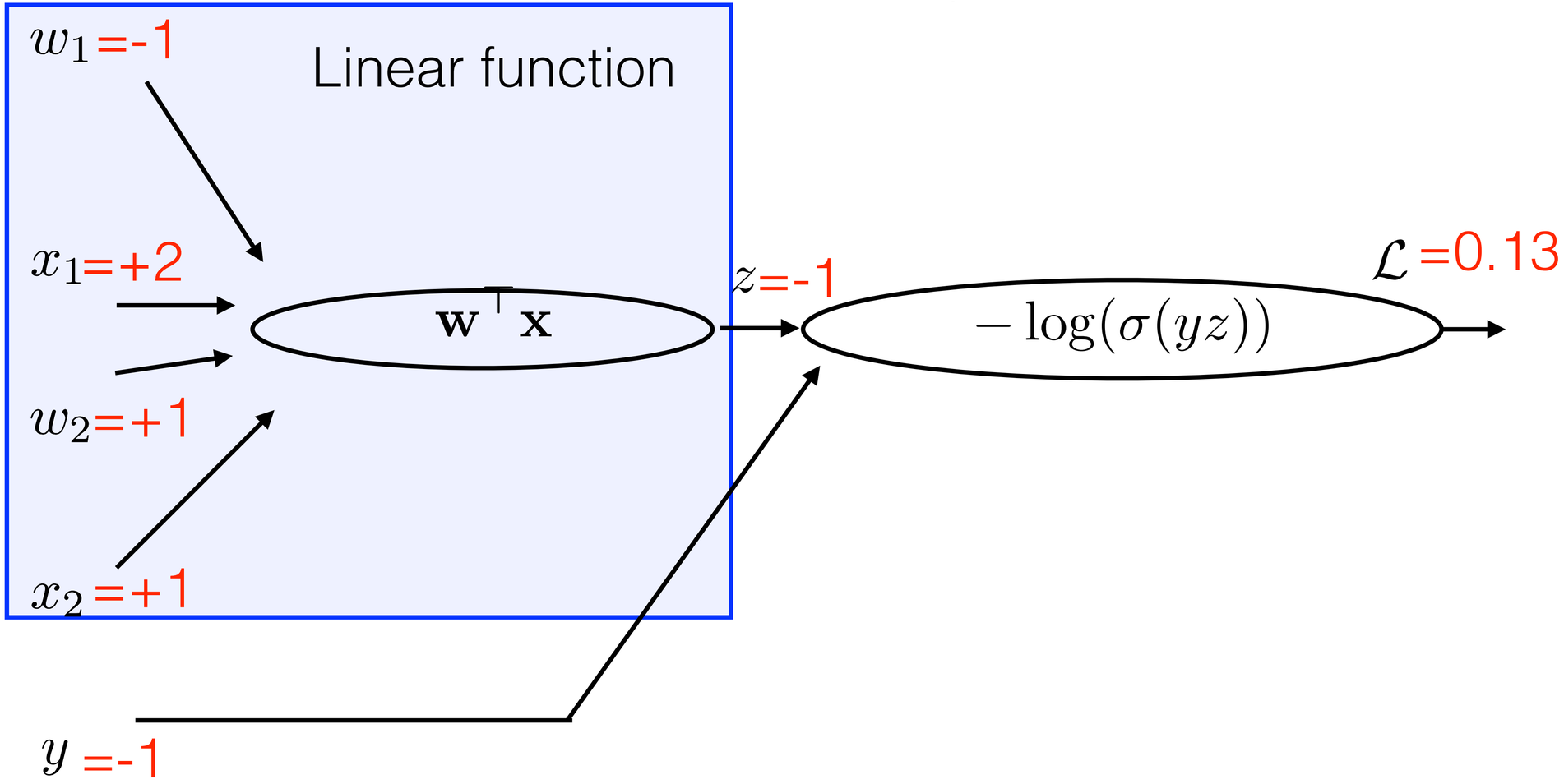
This is the logistic loss!

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

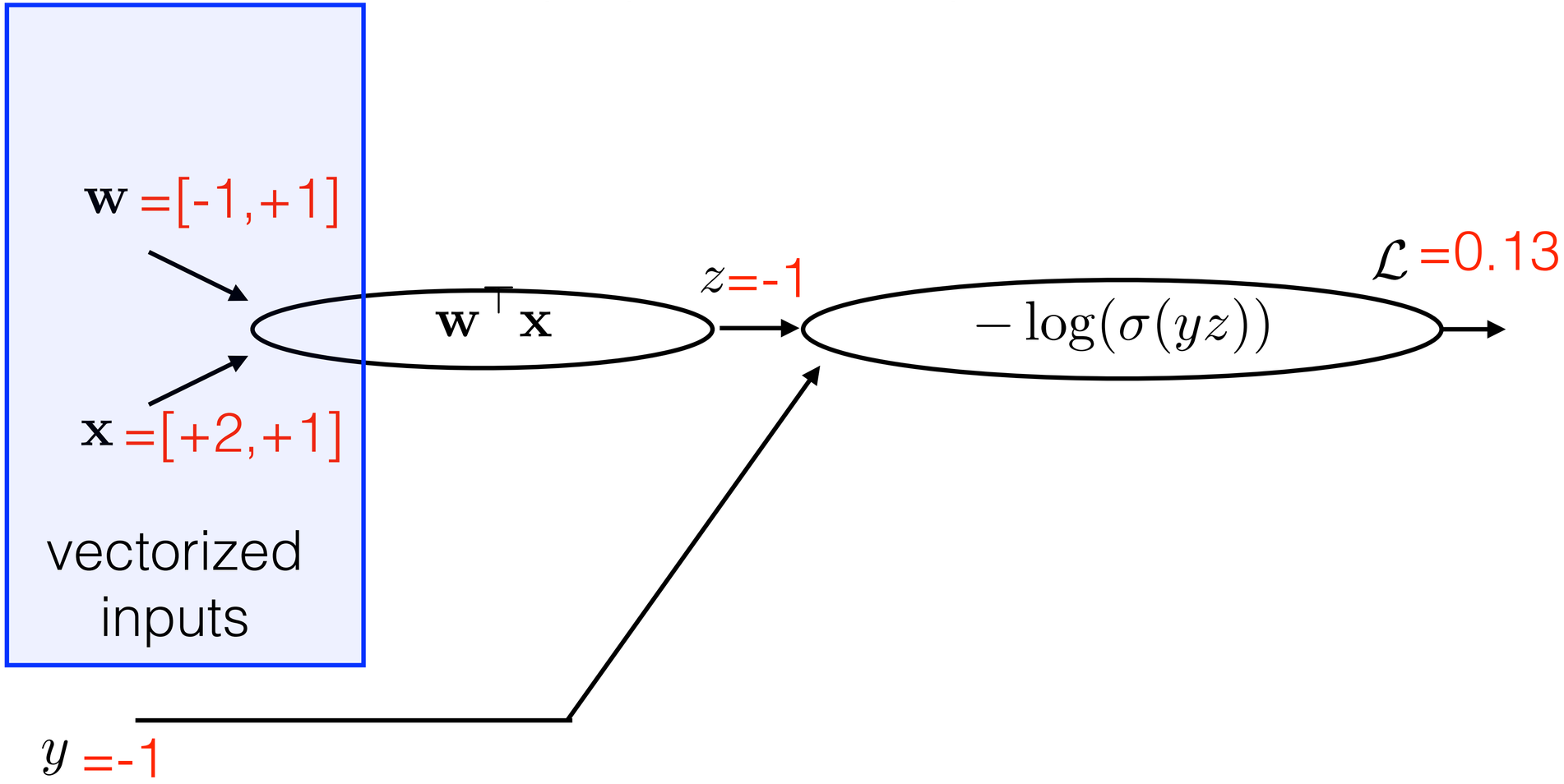
Backprop in vector representation



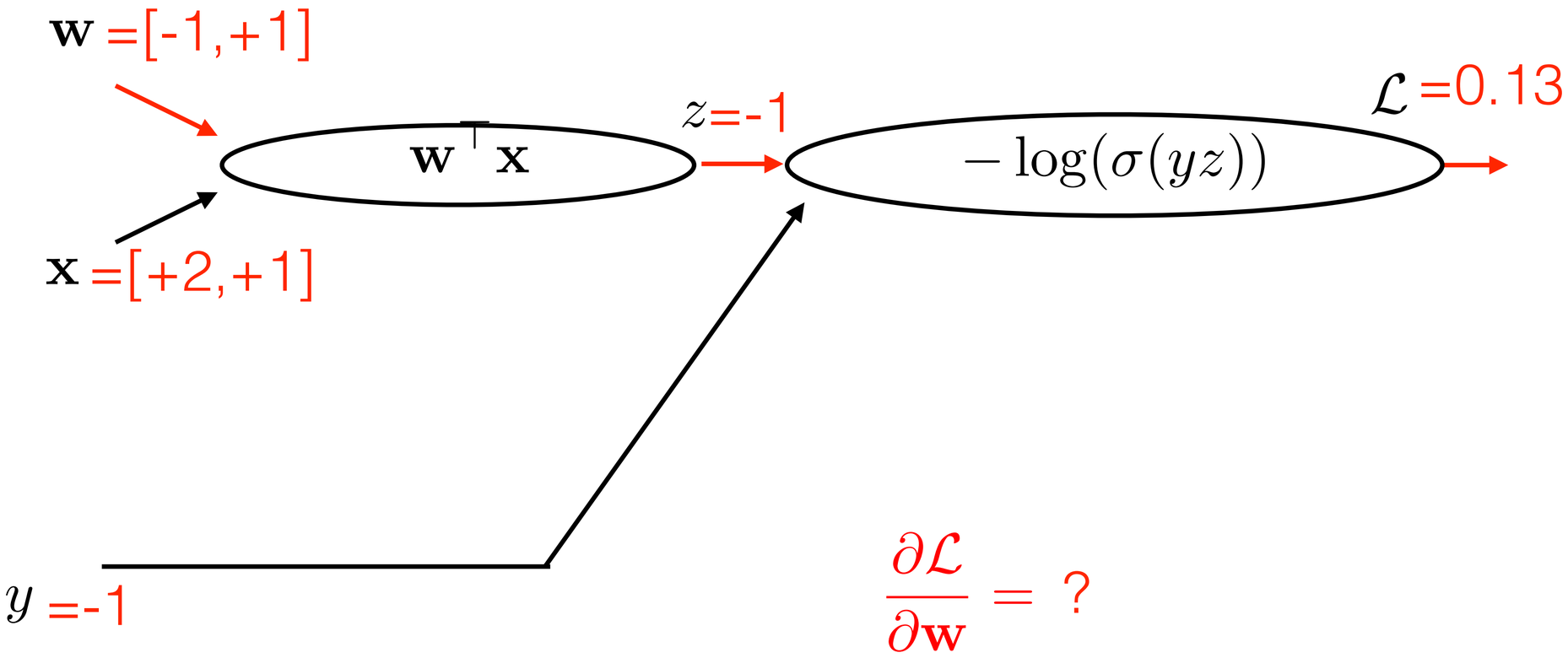
Backprop in vector representation



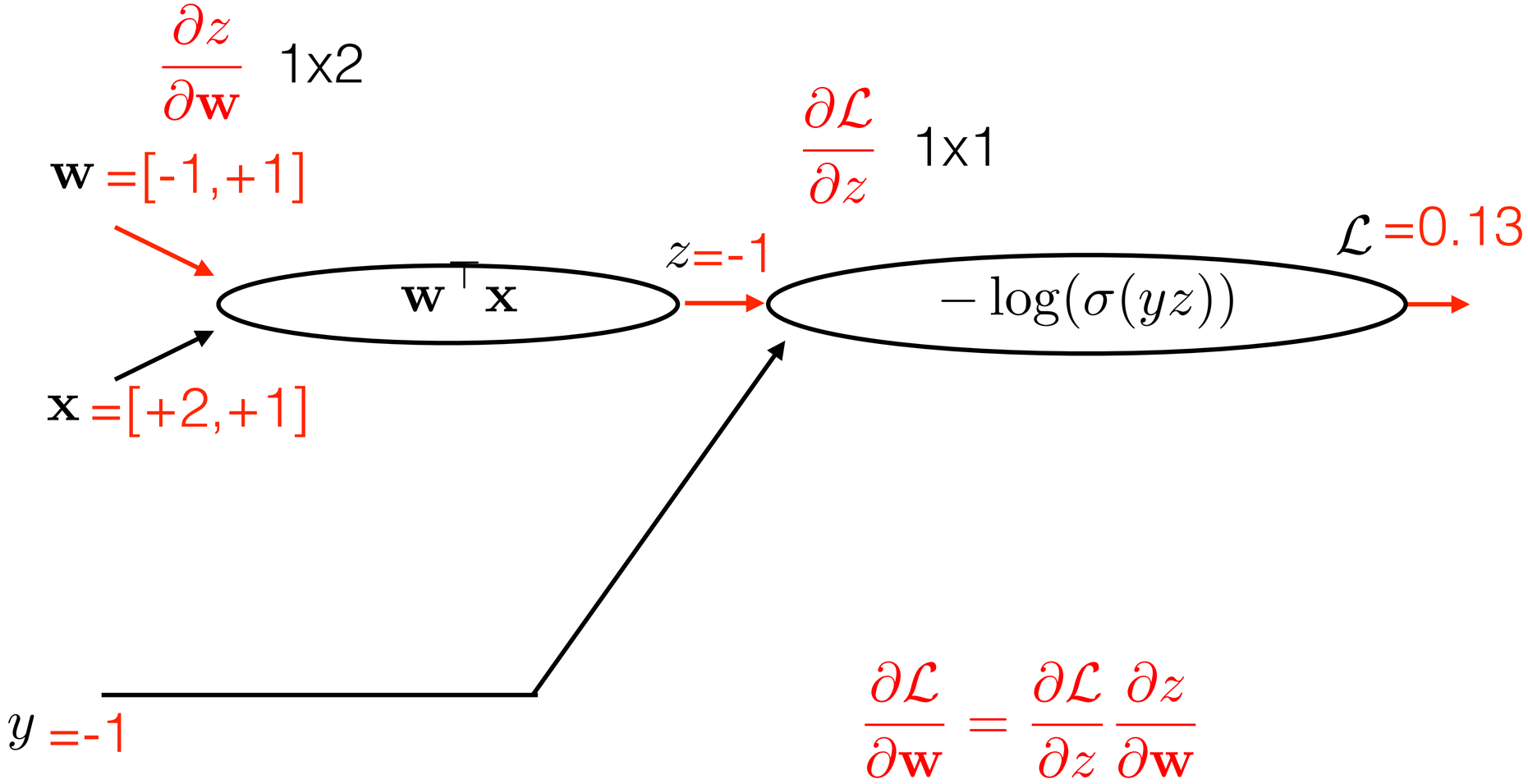
Backprop in vector representation



Backprop in vector representation

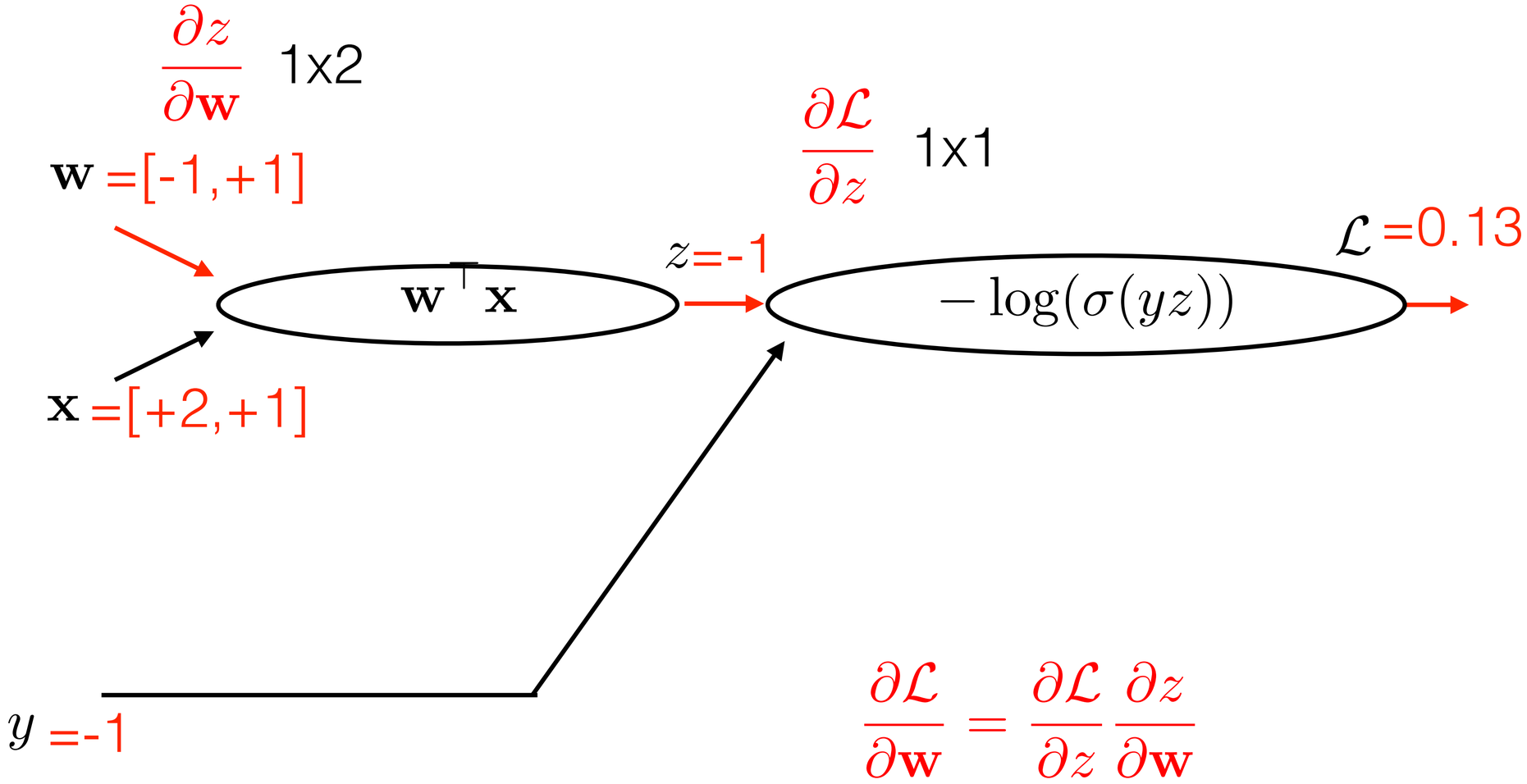


Backprop in vector representation



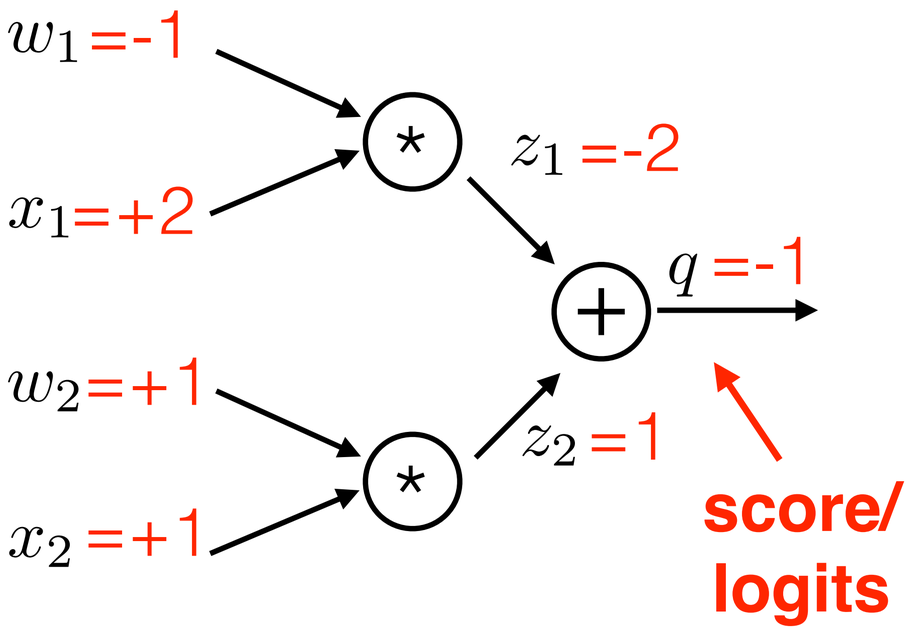
$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial \mathbf{w}}$$

Backprop in vector representation

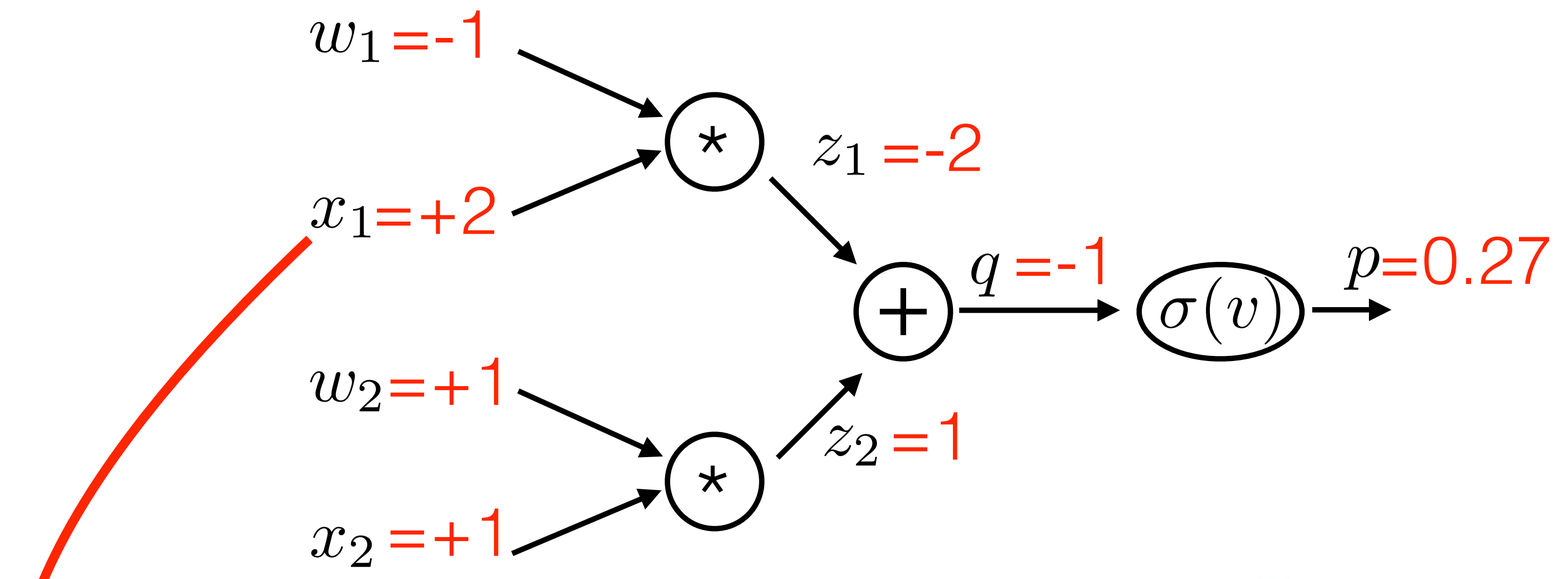


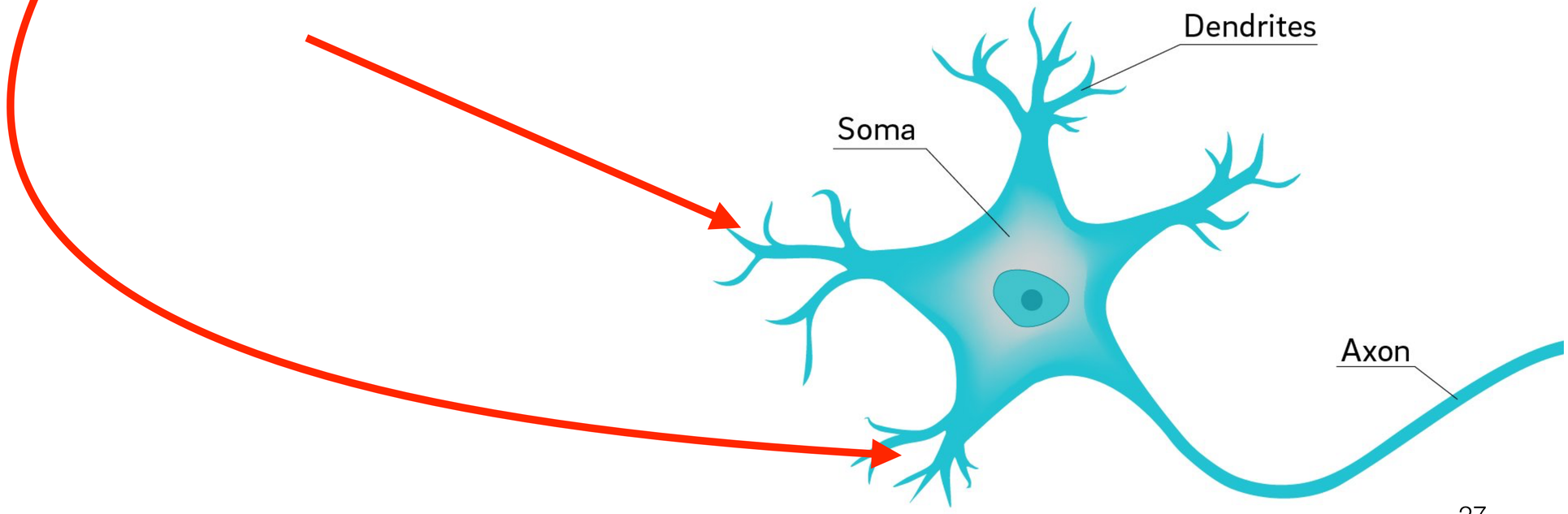
Learning from multiple training samples means summing up the partial derivative over all samples

Linear classifier + sigmoid = neuron

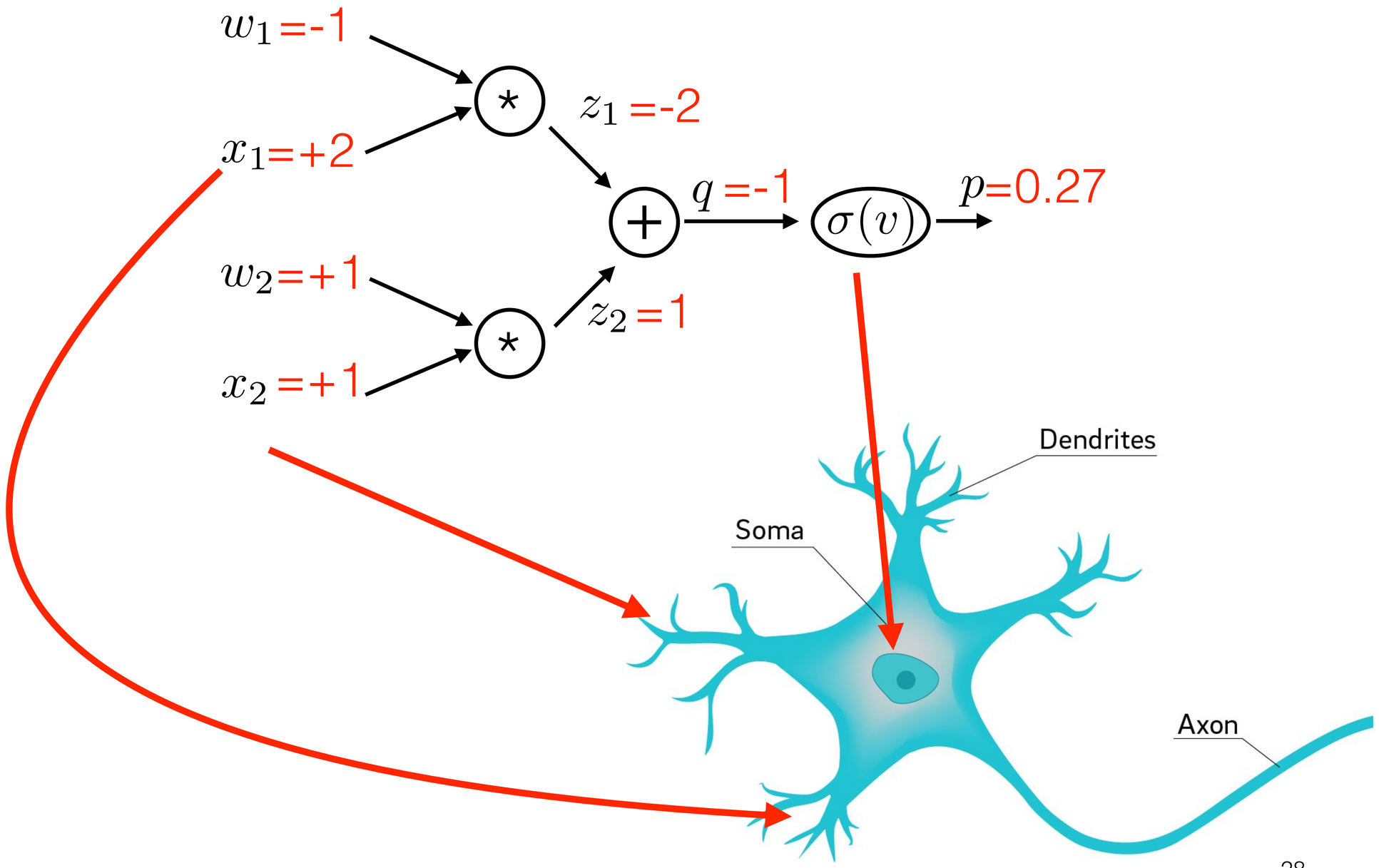


Relation to biological neuron

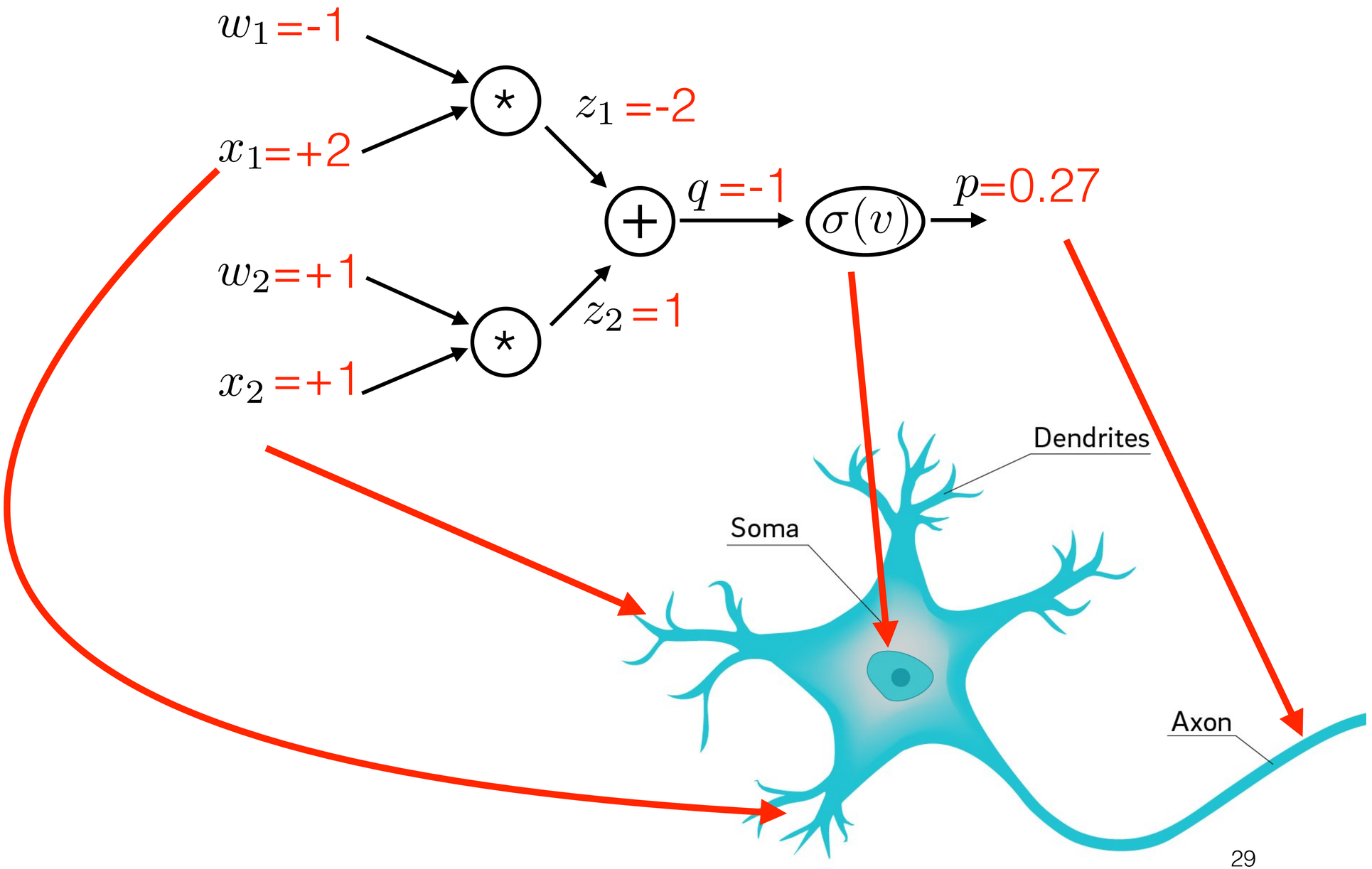




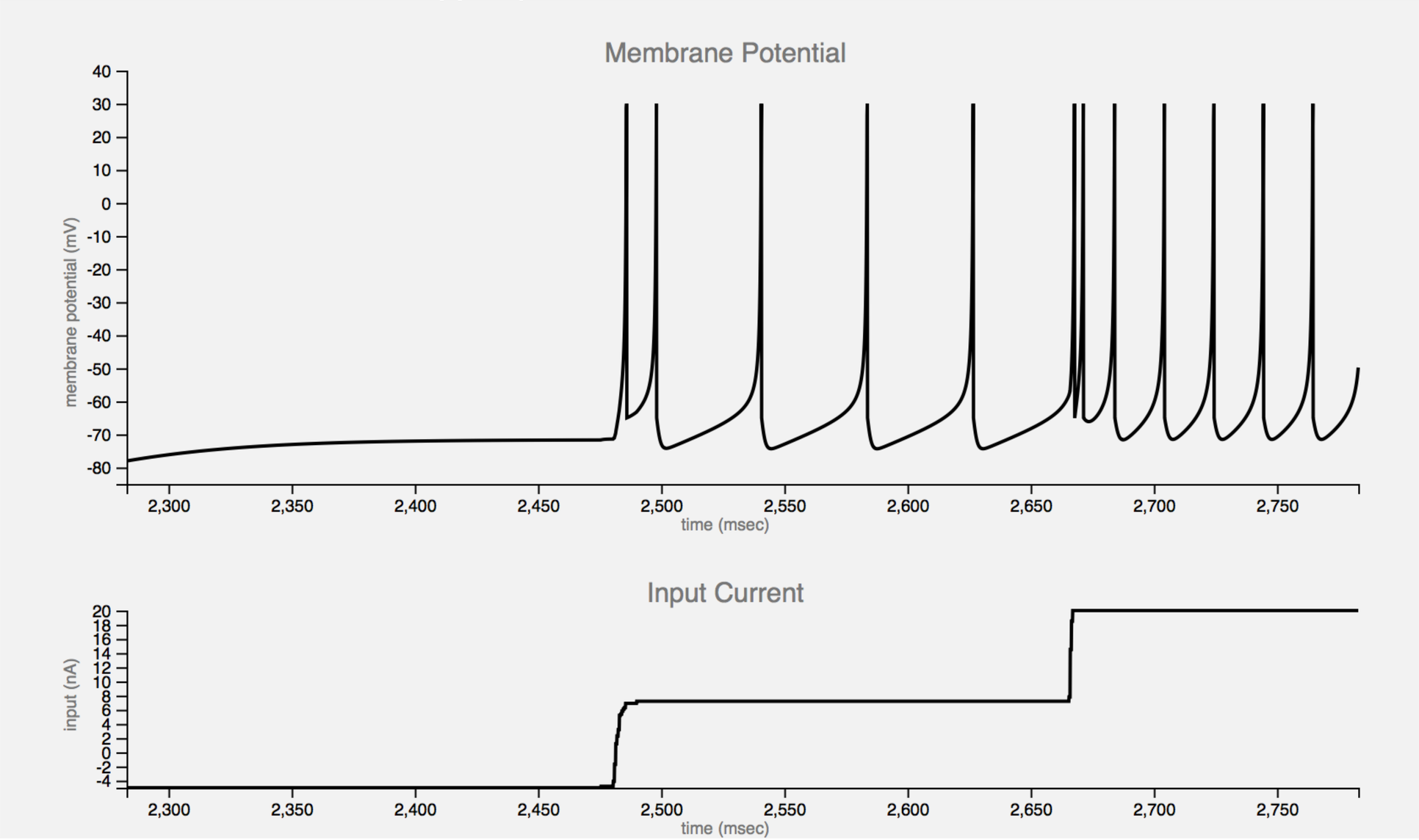
Relation to biological neuron



Relation to biological neuron

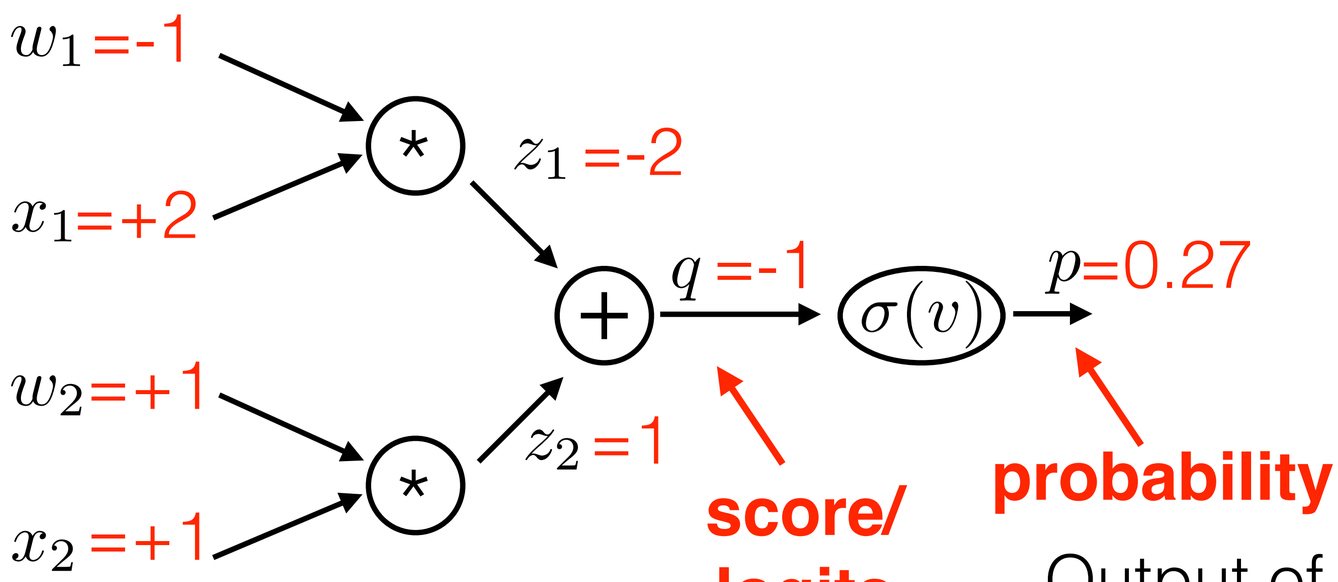


Modeling dynamic neuron behaviour

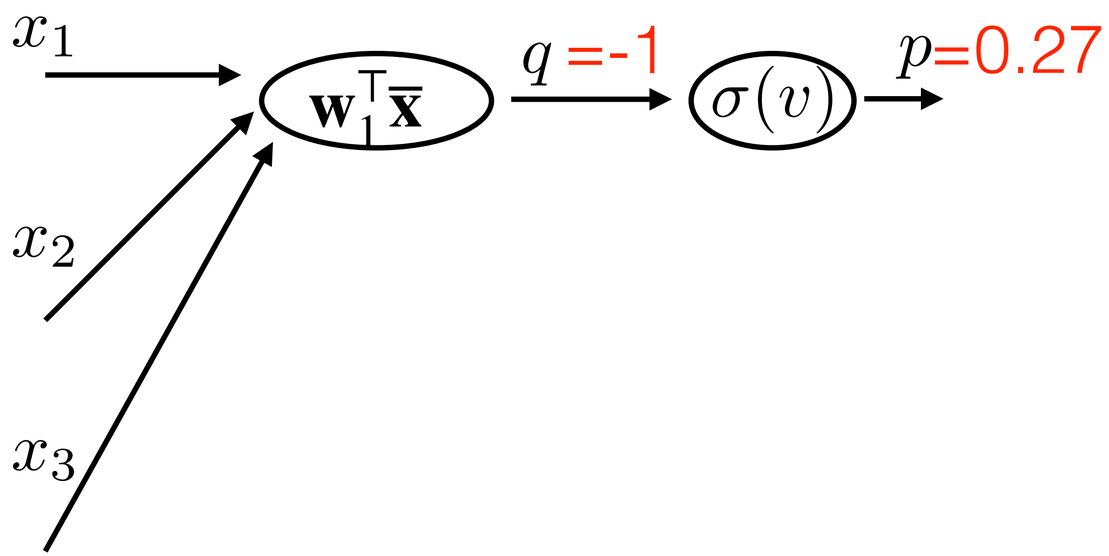


http://jackterwilliger.com/biological-neural-networks-part-i-spiking-neurons/

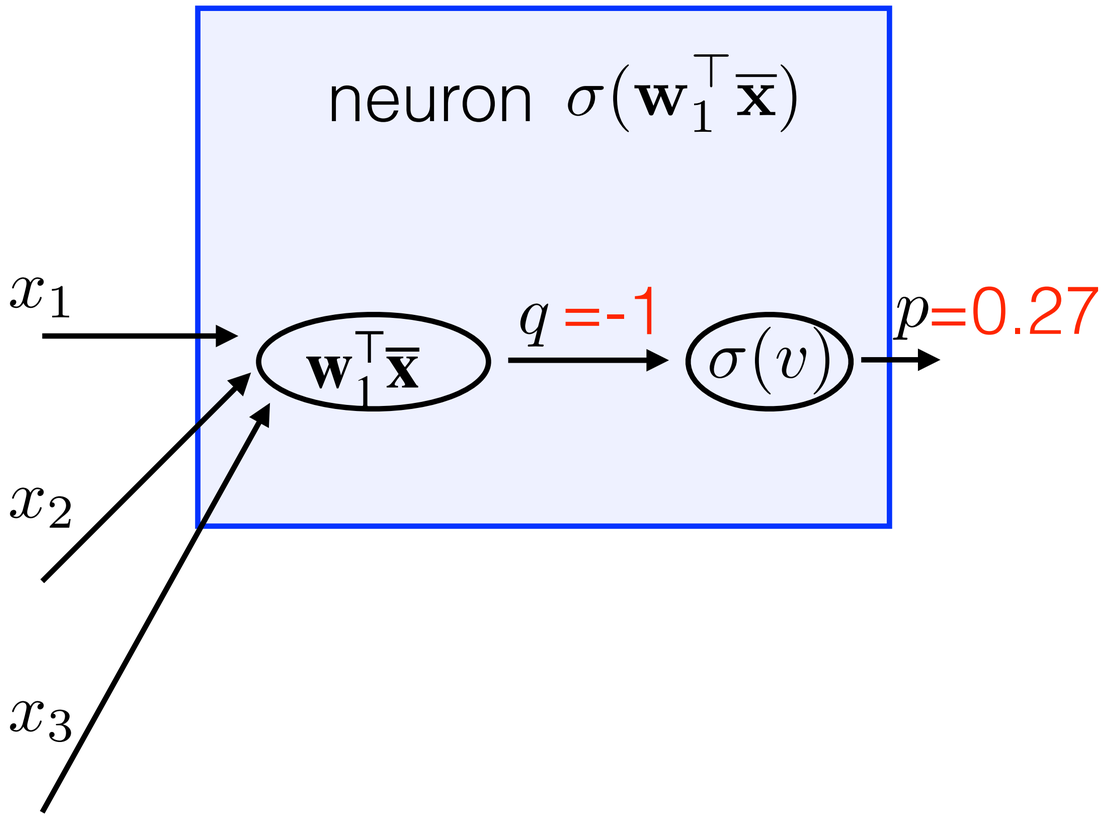
Neuron


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
score/ logits
probability
Output of linear classifier
Output of neuron

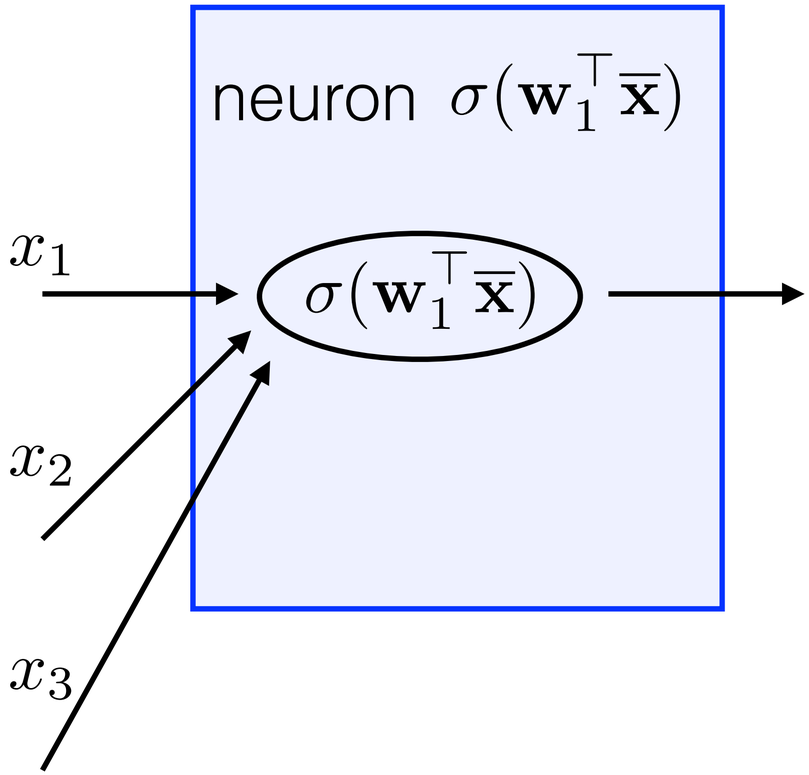
Neuron



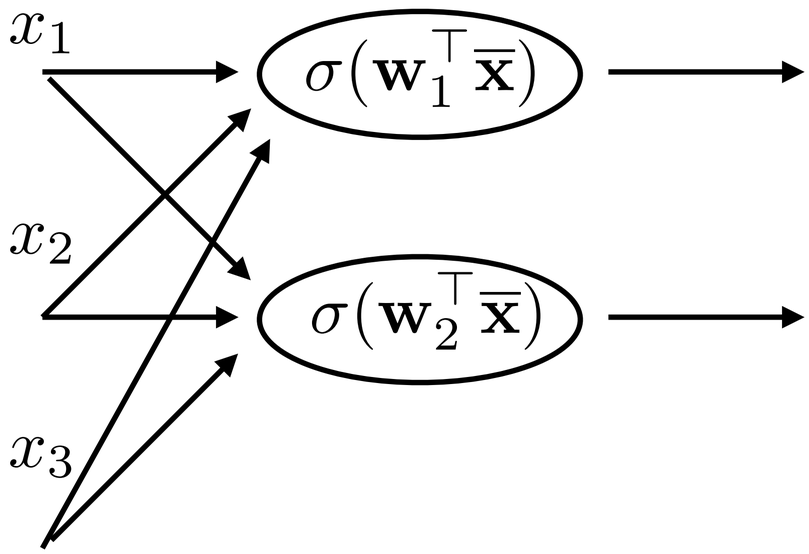
Neuron


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

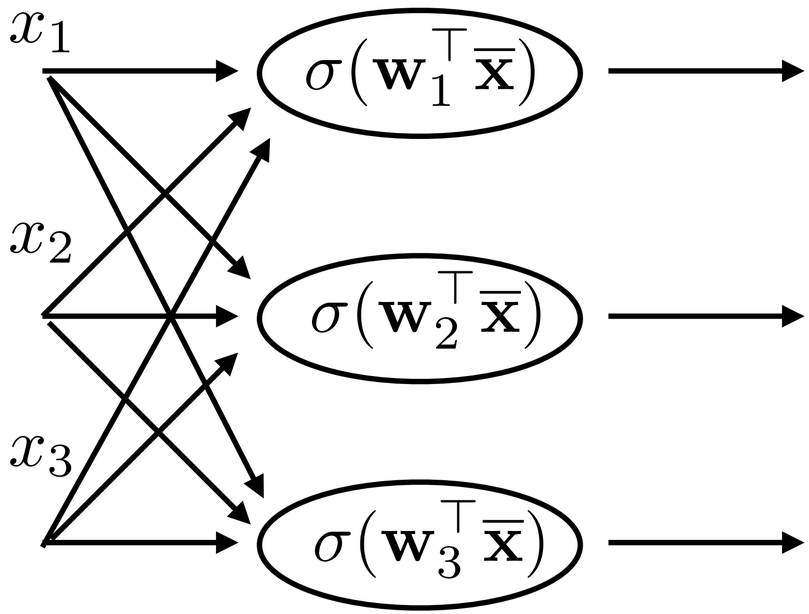
Fully-connected neural network



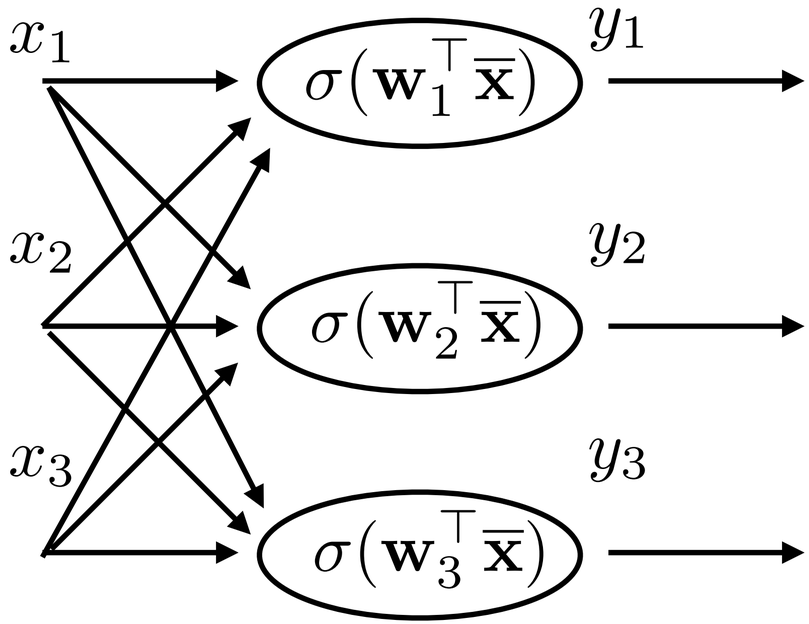
Fully-connected neural network



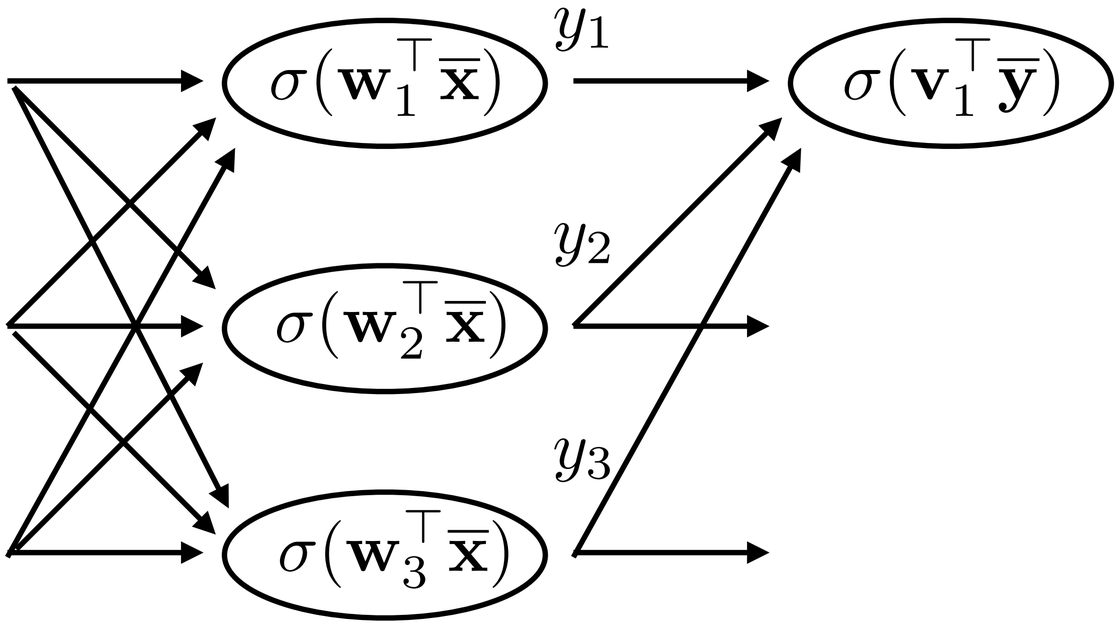
Fully-connected neural network



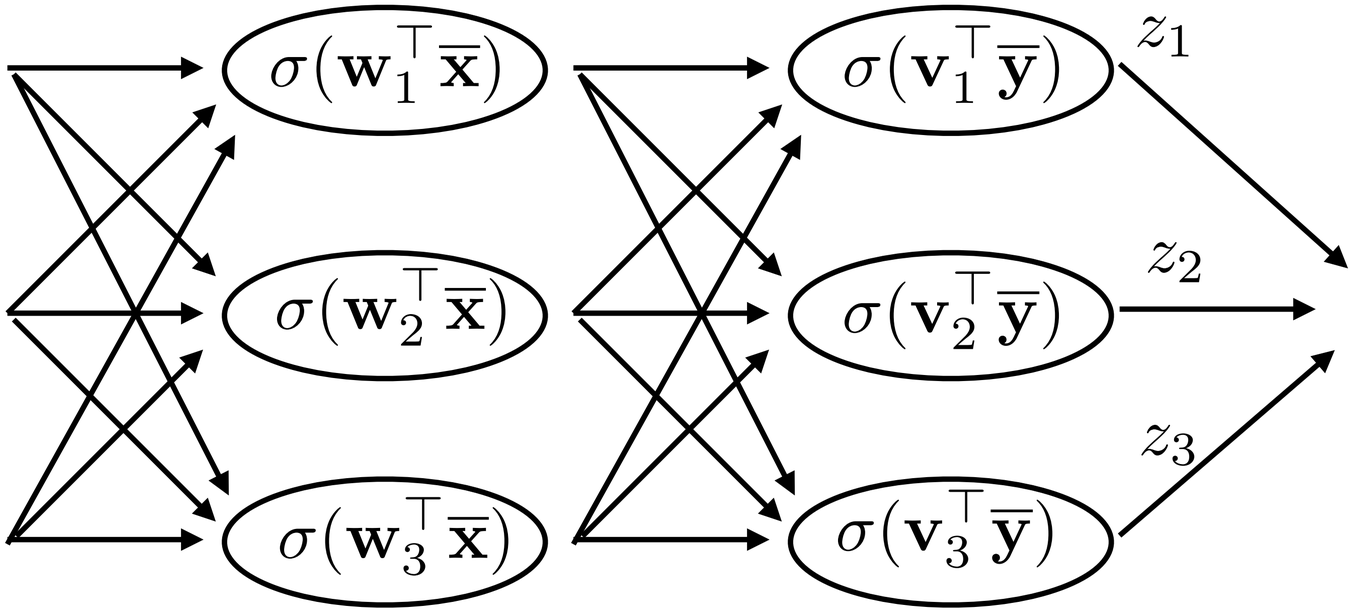
Fully-connected neural network



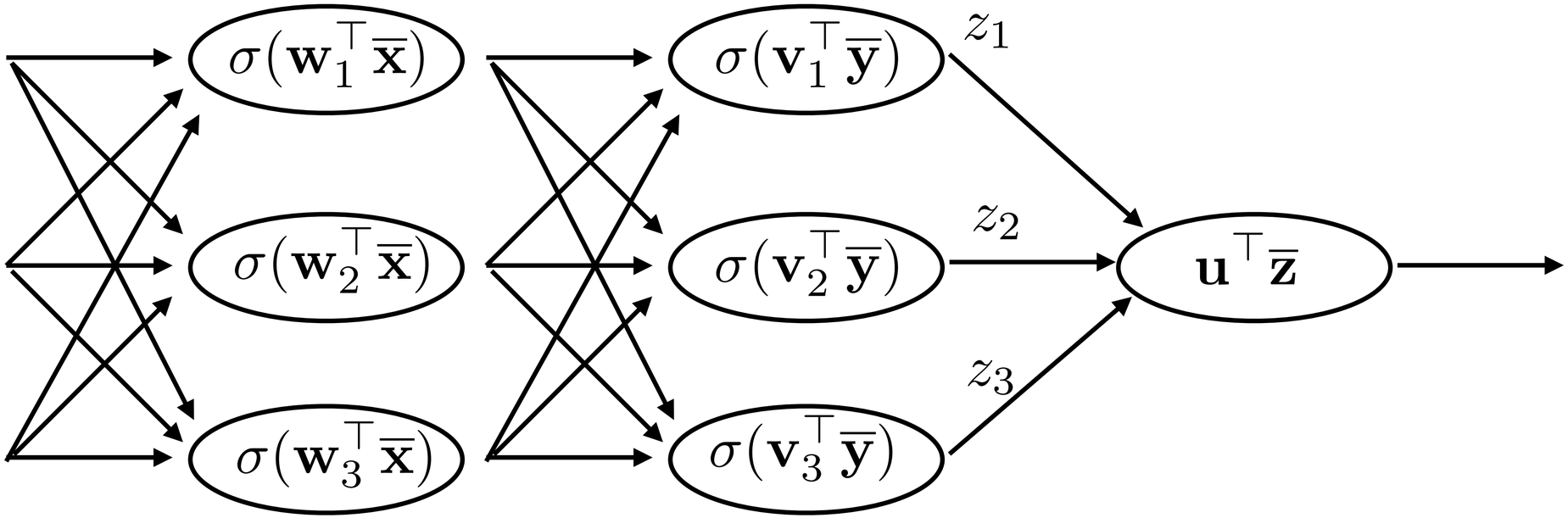
Fully-connected neural network



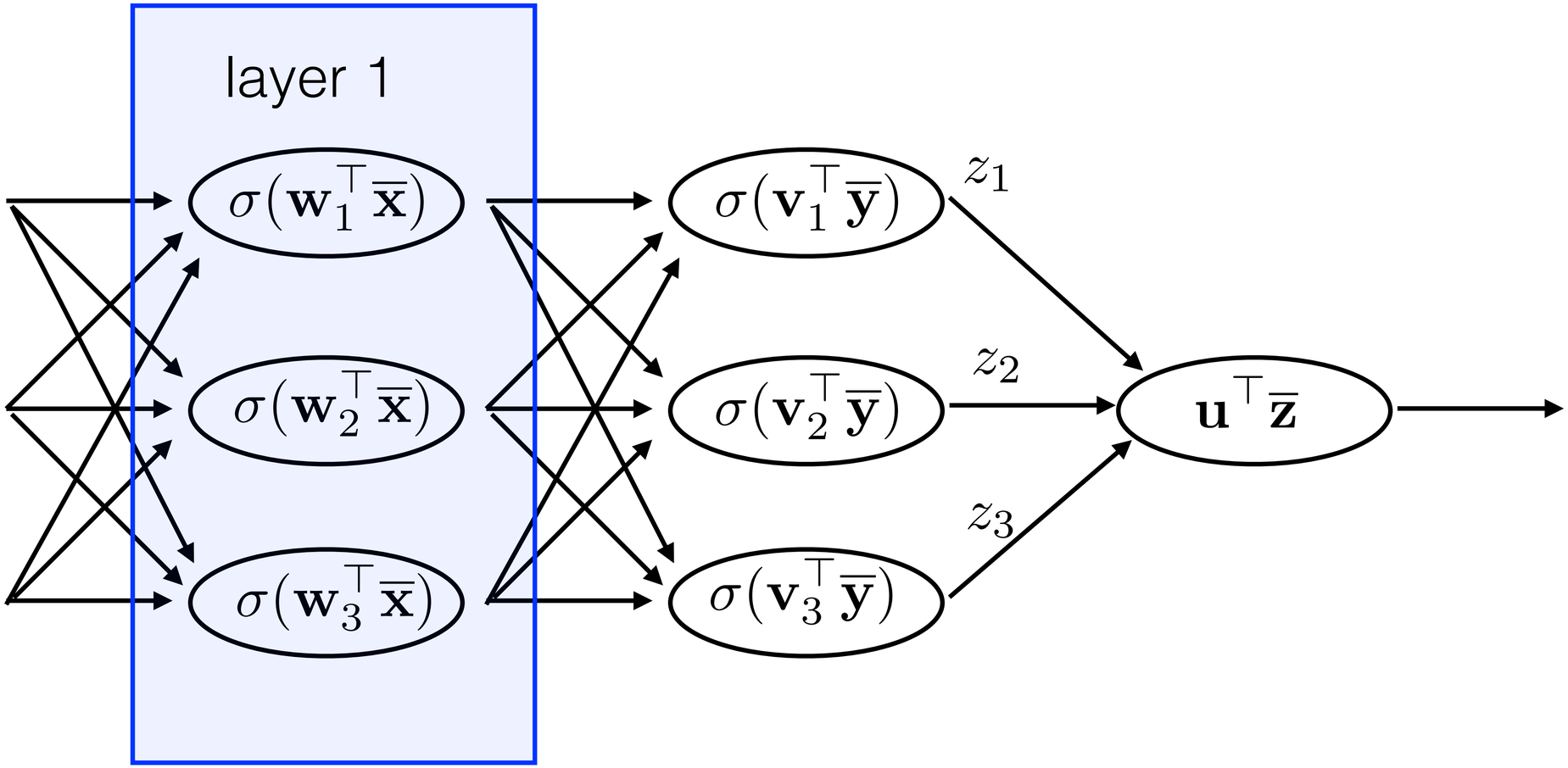
Fully-connected neural network



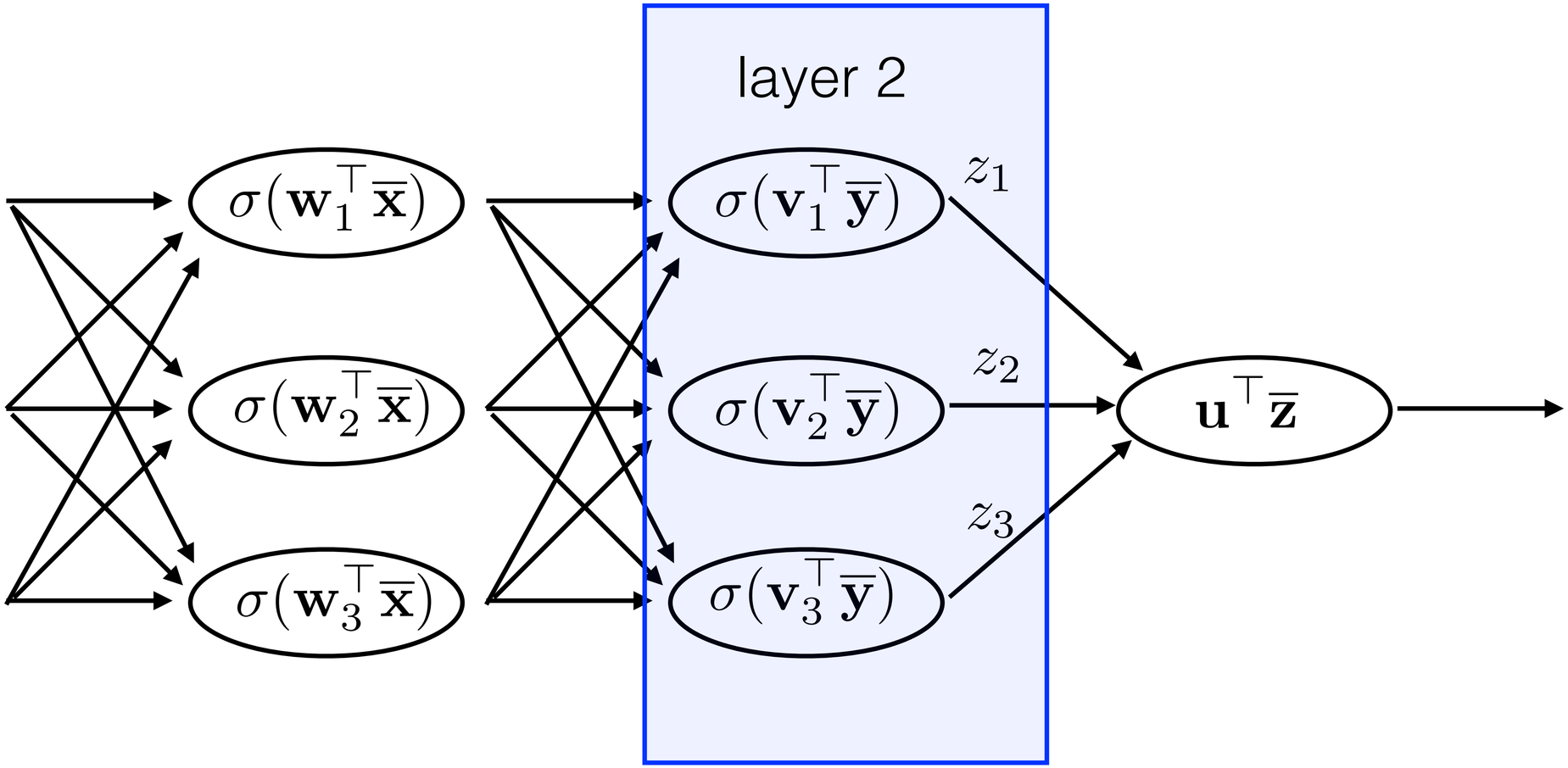
Fully-connected neural network



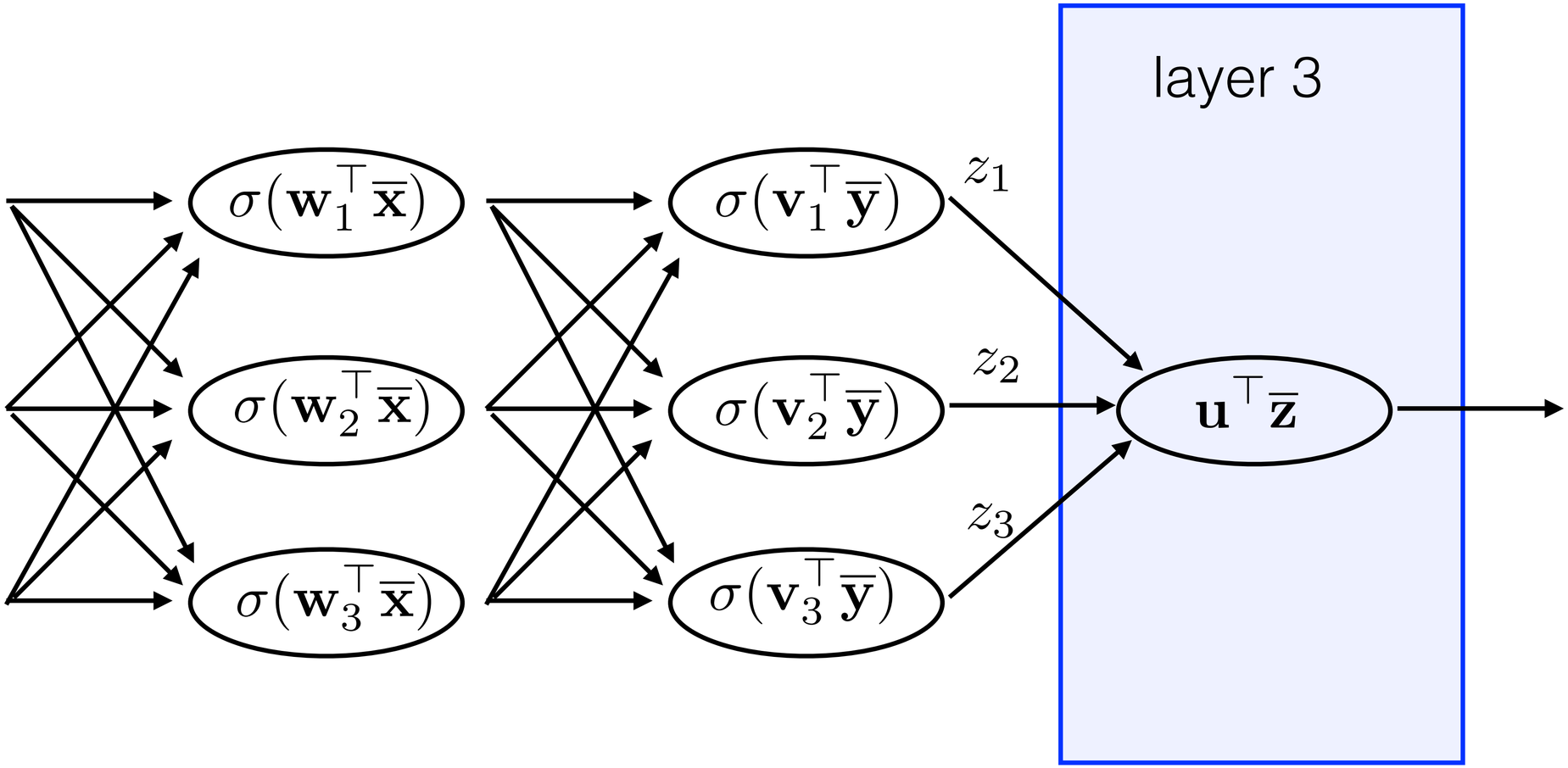
Fully-connected neural network



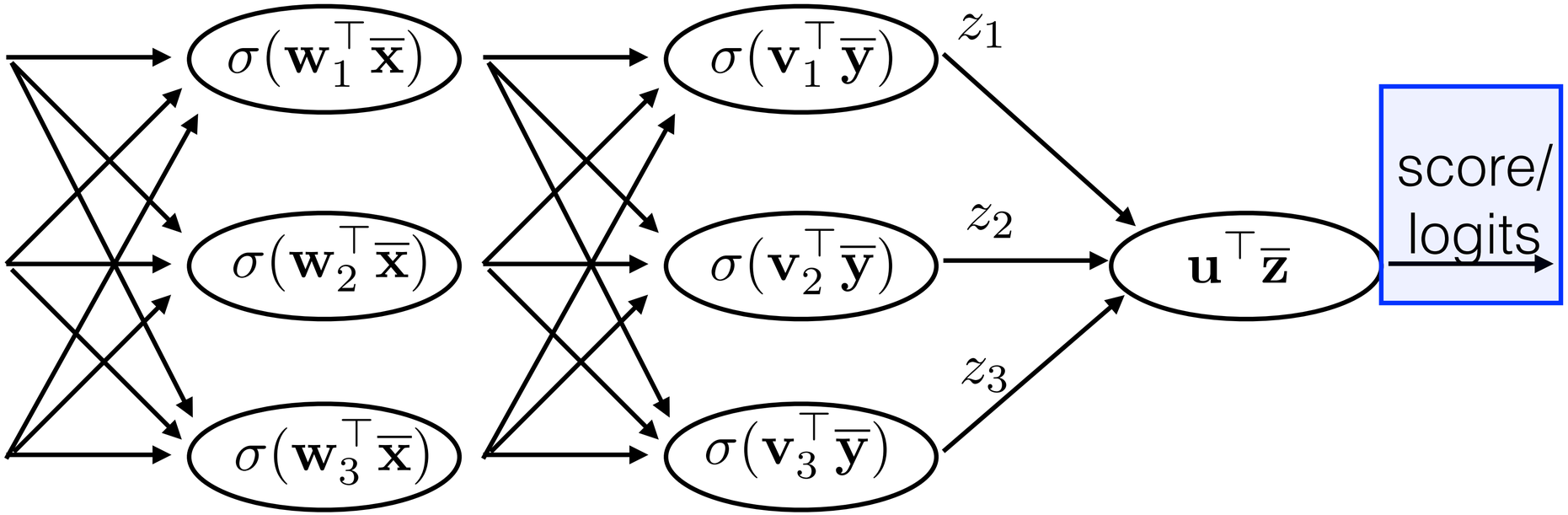
Fully-connected neural network



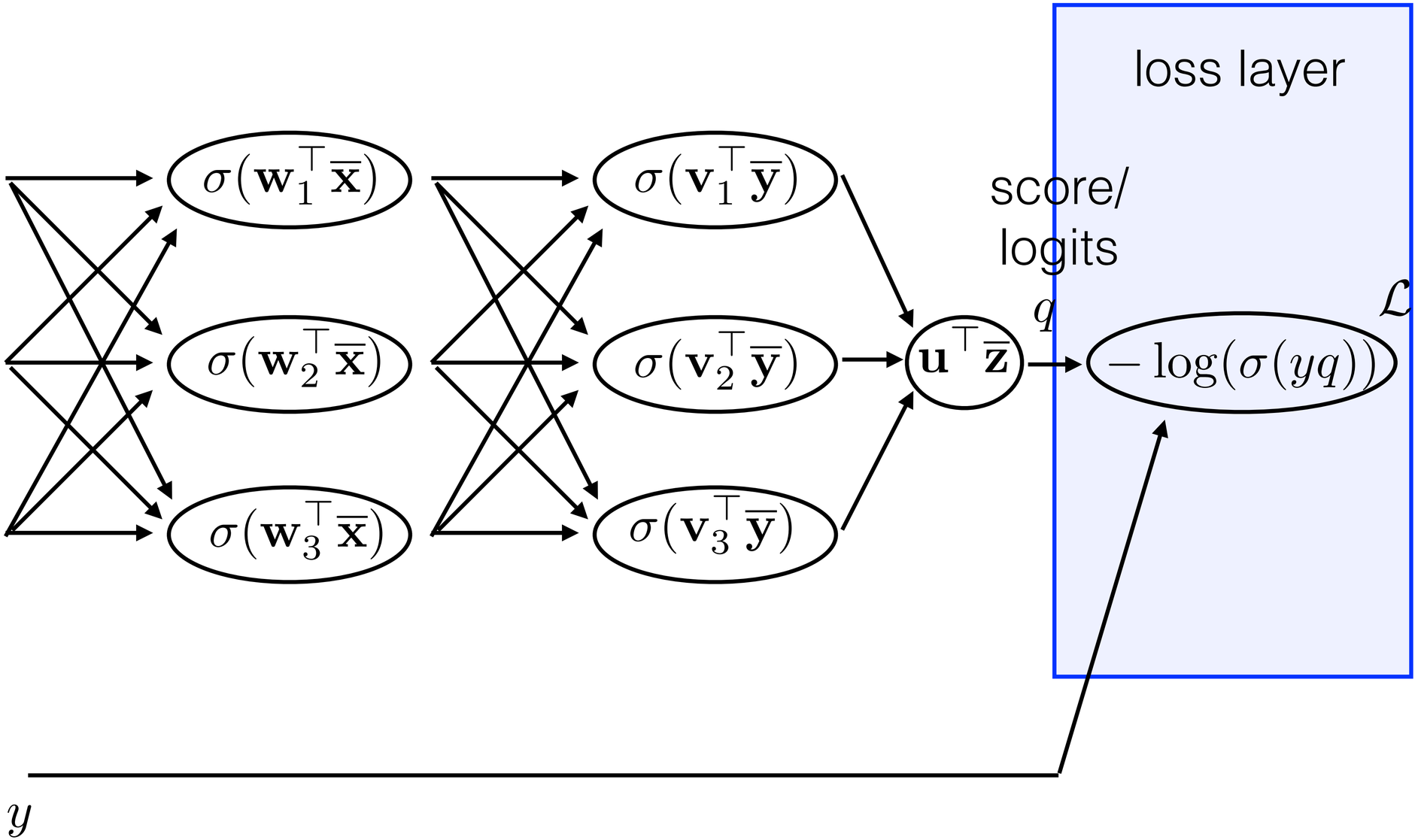
Fully-connected neural network



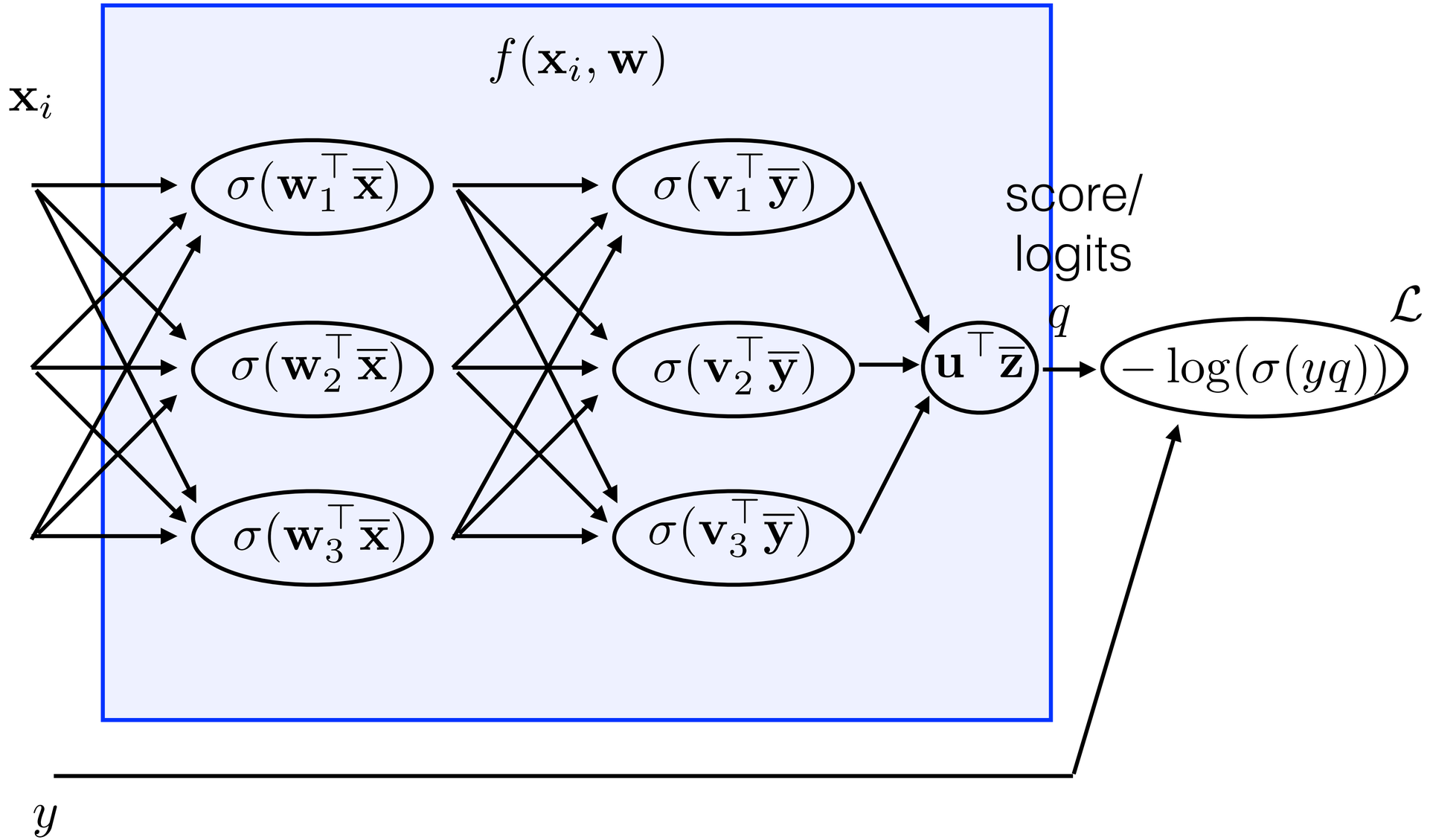
Fully-connected neural network



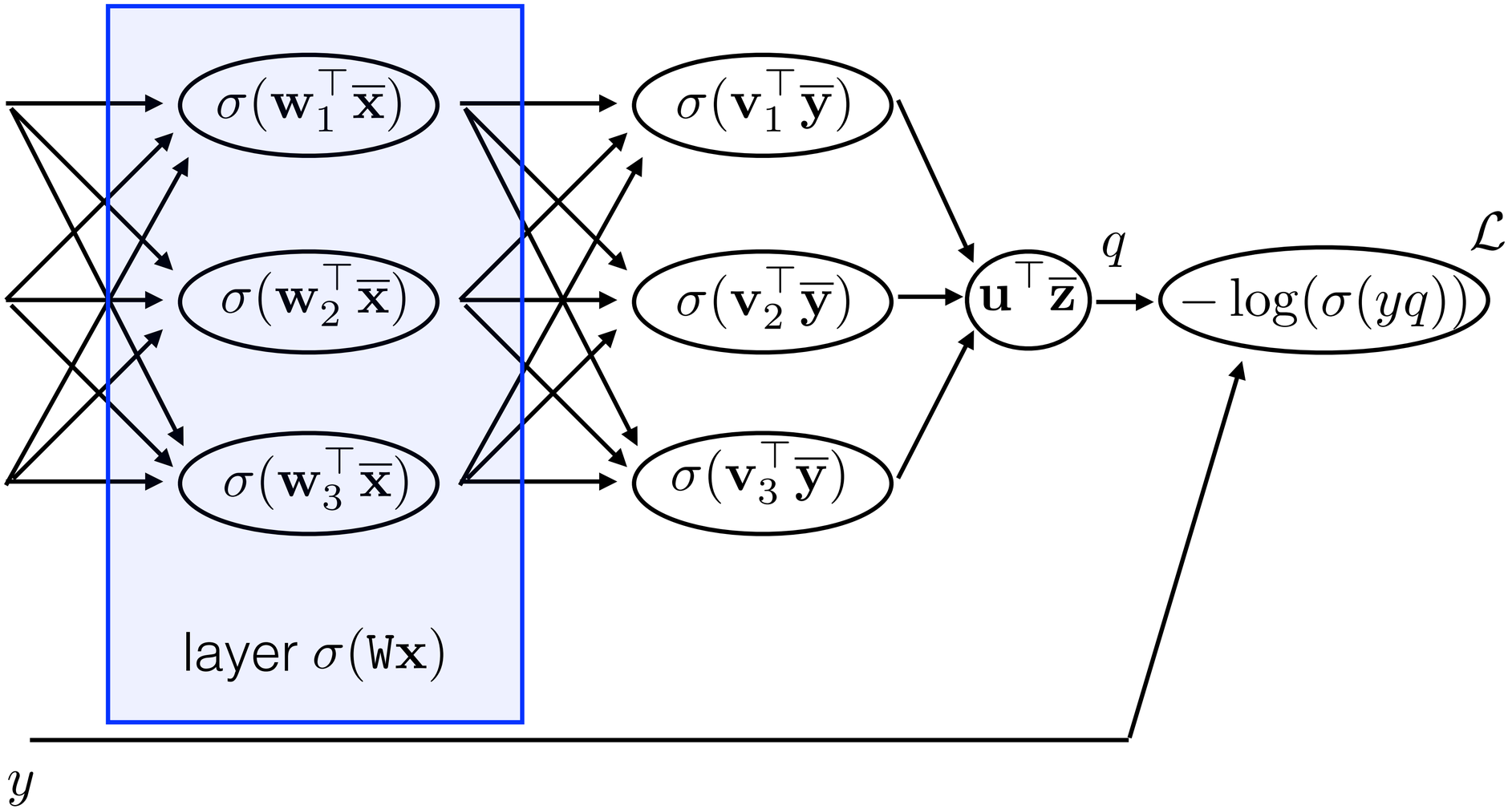
Fully-connected neural network



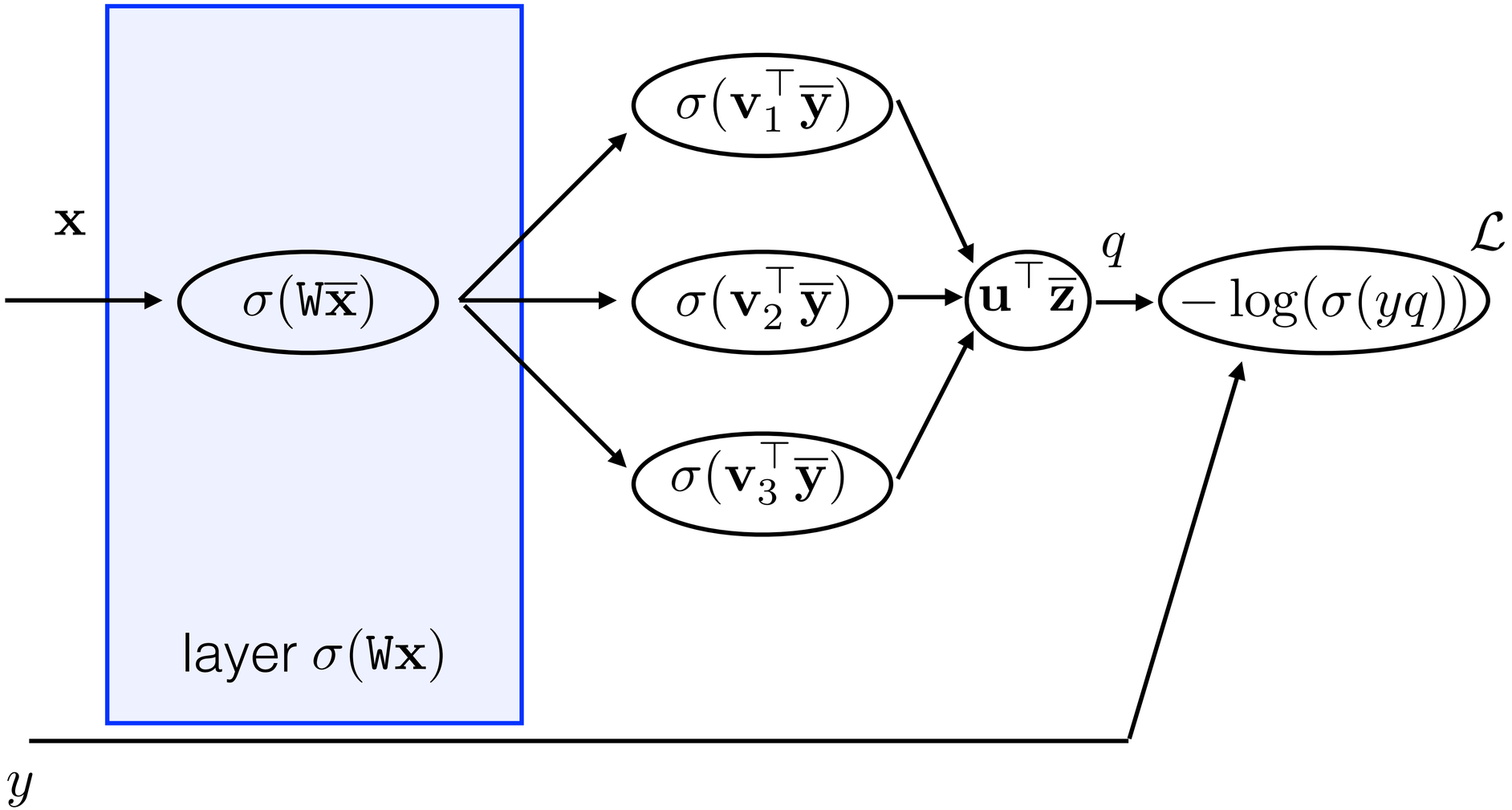
Fully-connected neural network



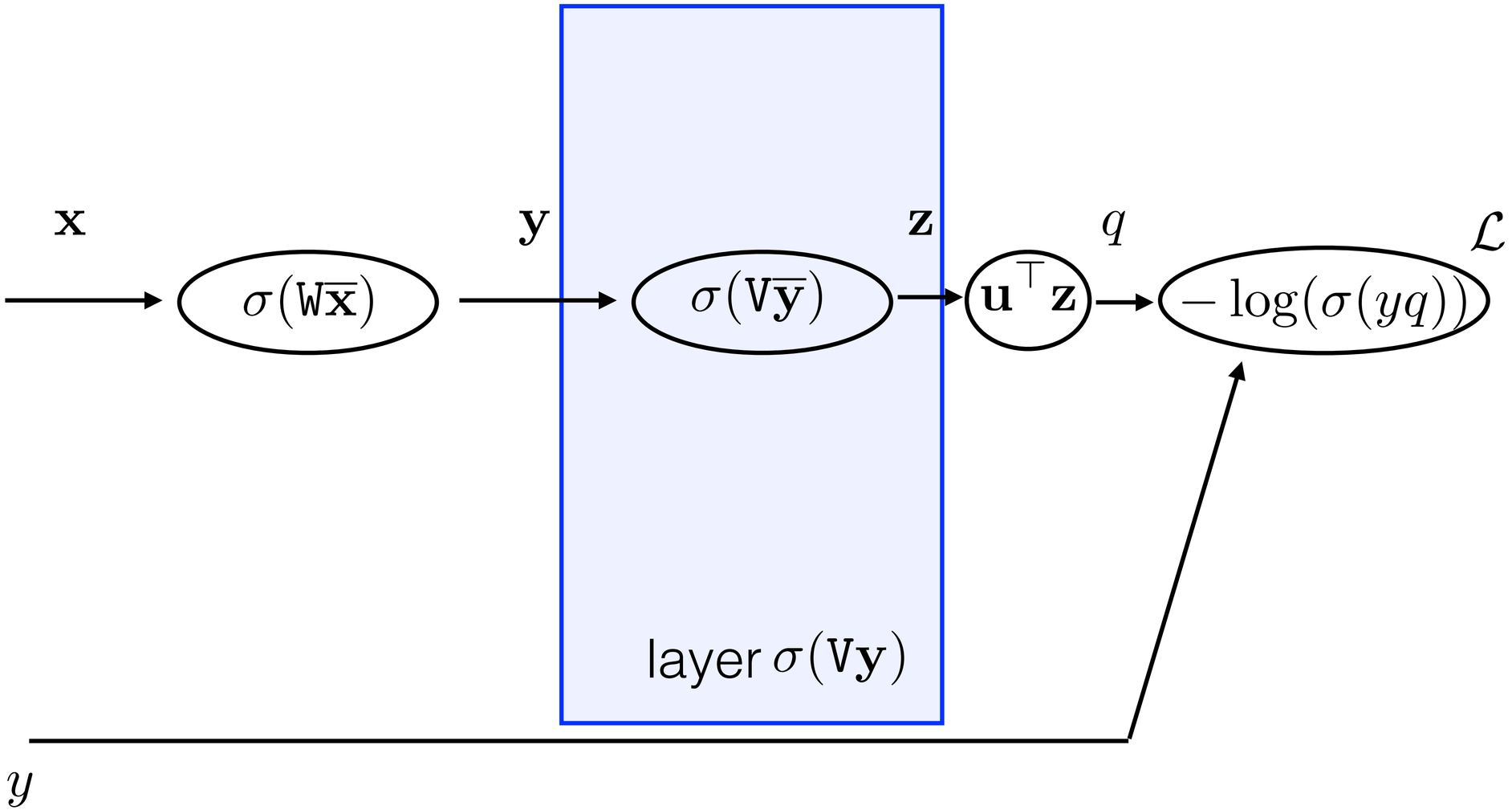
Fully-connected neural network



Fully-connected neural network

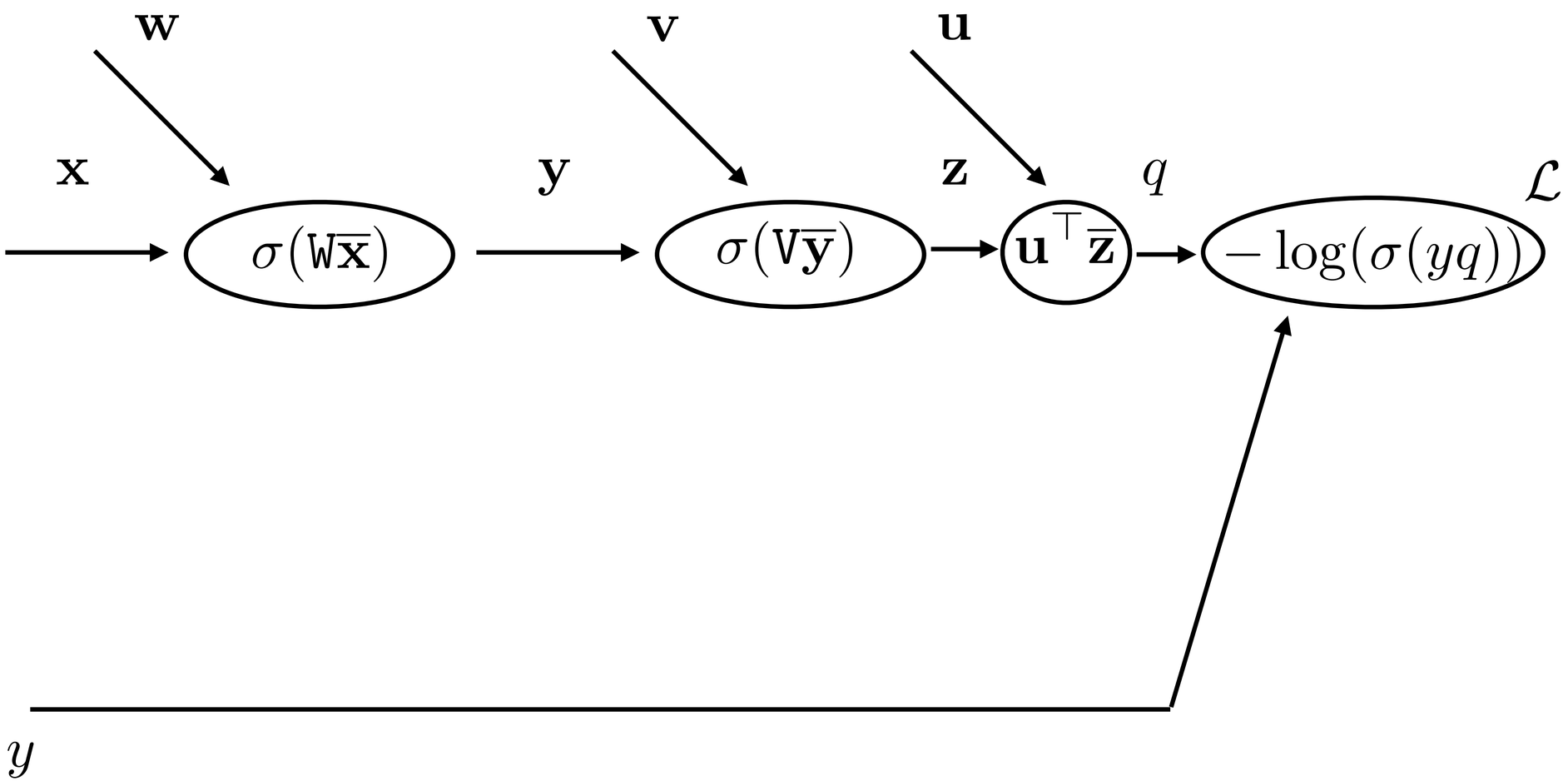


Fully-connected neural network



Fully-connected neural network

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

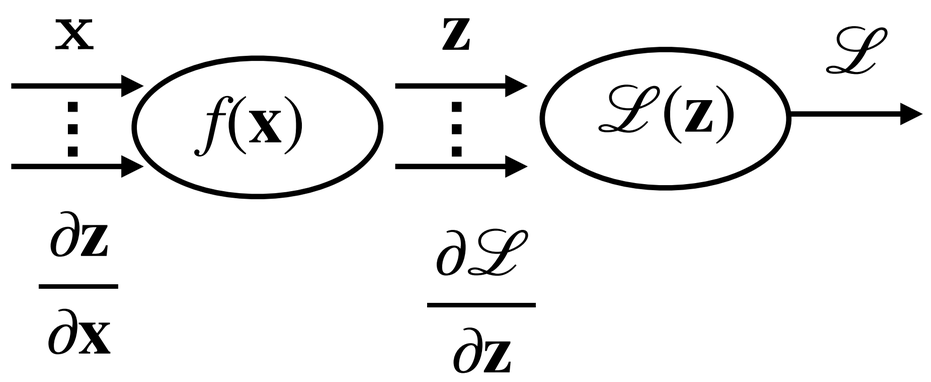


Chainrule and Jacobian

M-dim vector

N-dim vector

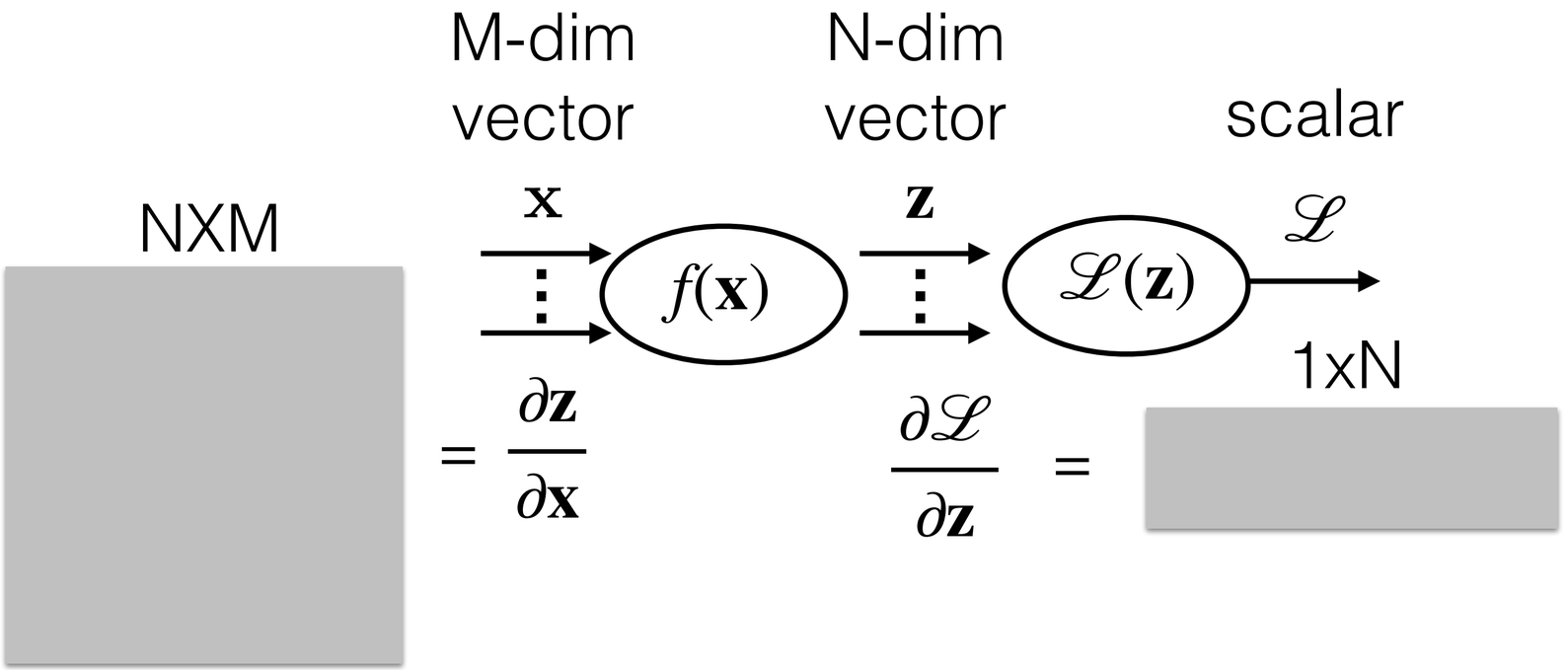
scalar



$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{x}}$$

Jacobian 1xM	Jacobian 1xN	Jacobian NxN

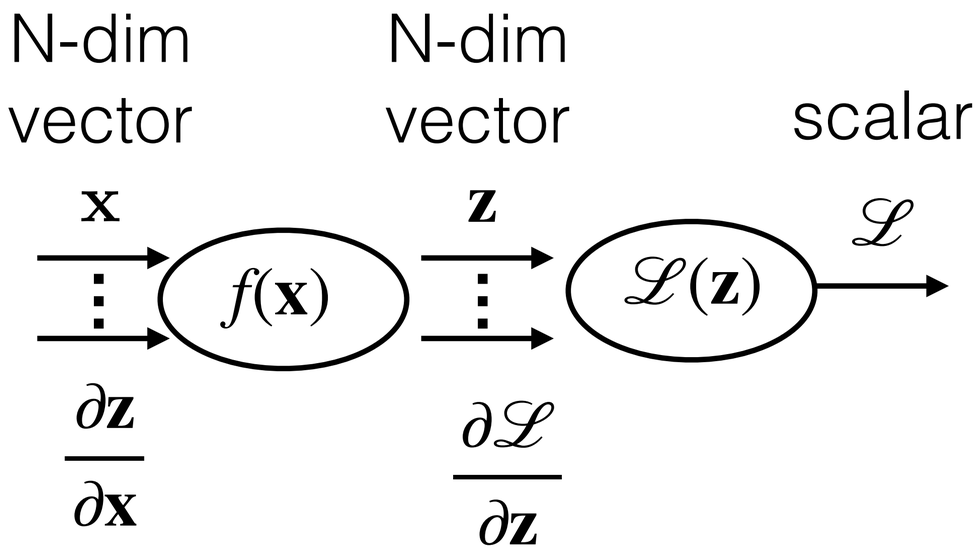
Chainrule and Jacobian

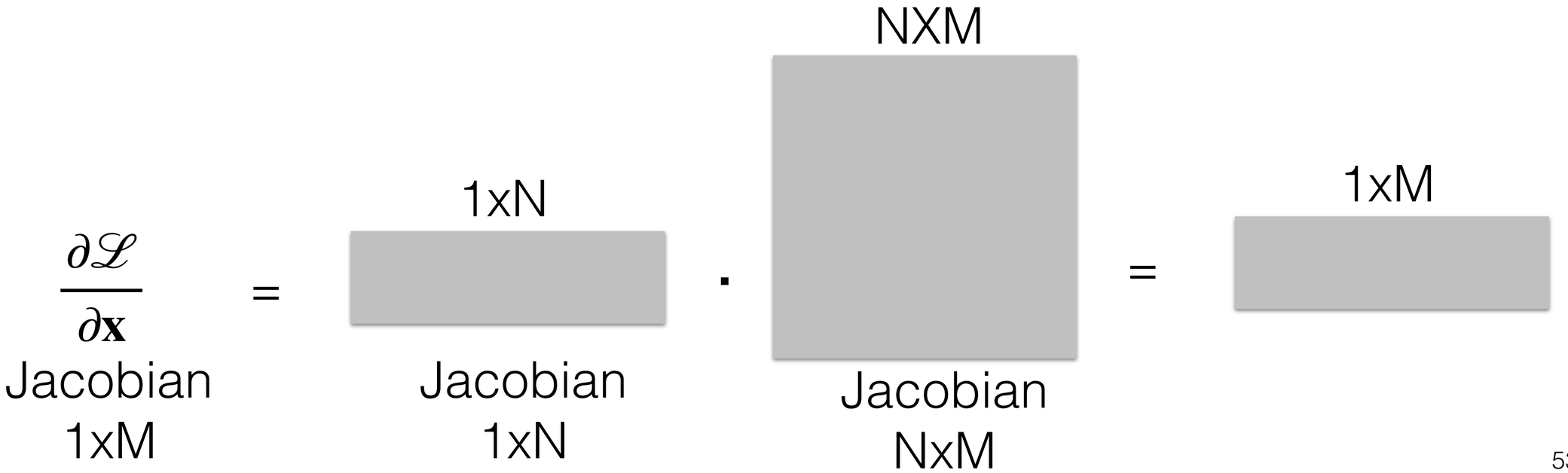


$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{x}}$$

Jacobian 1xM	Jacobian 1xN	Jacobian NxM

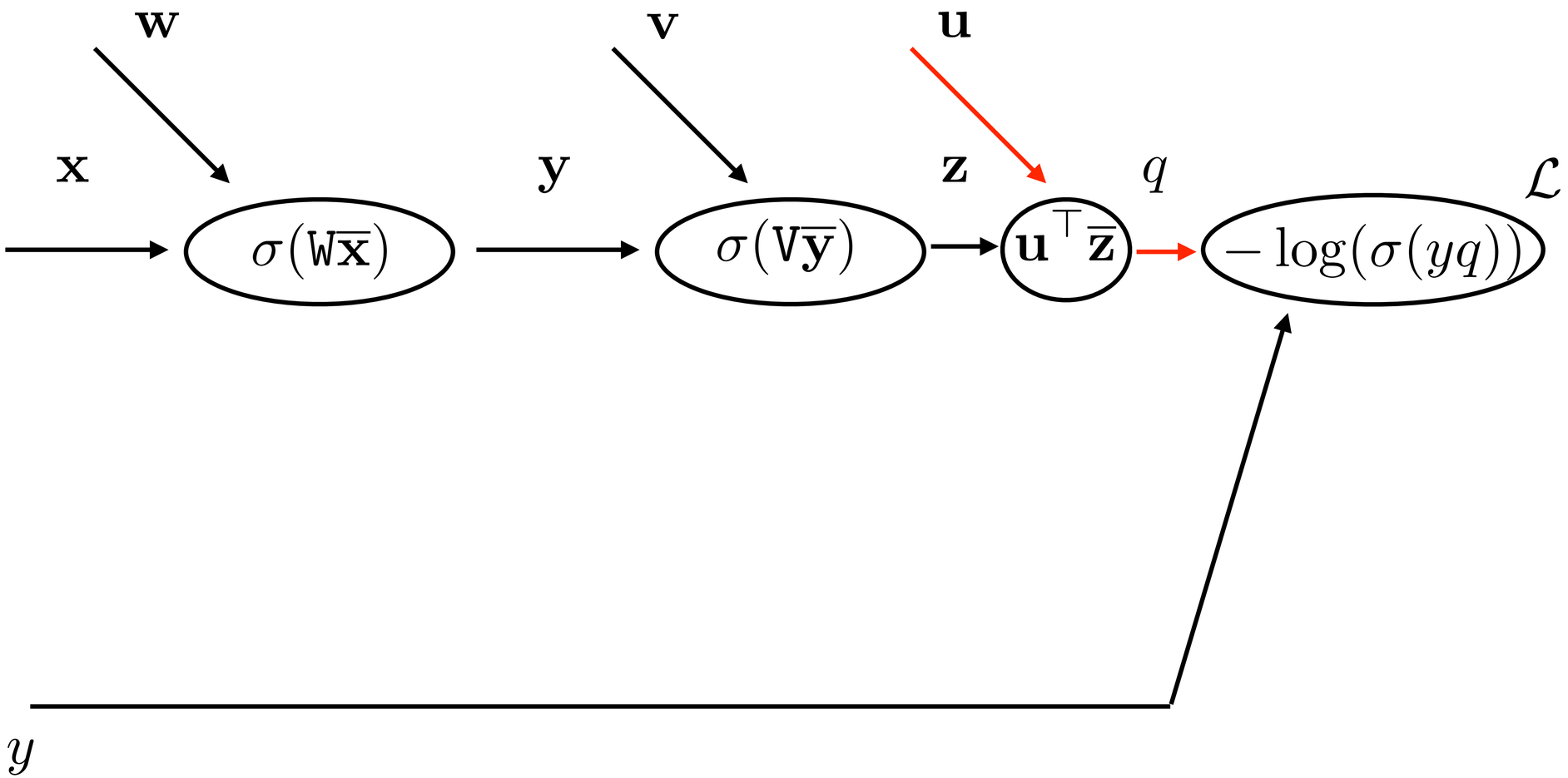
Chainrule and Jacobian





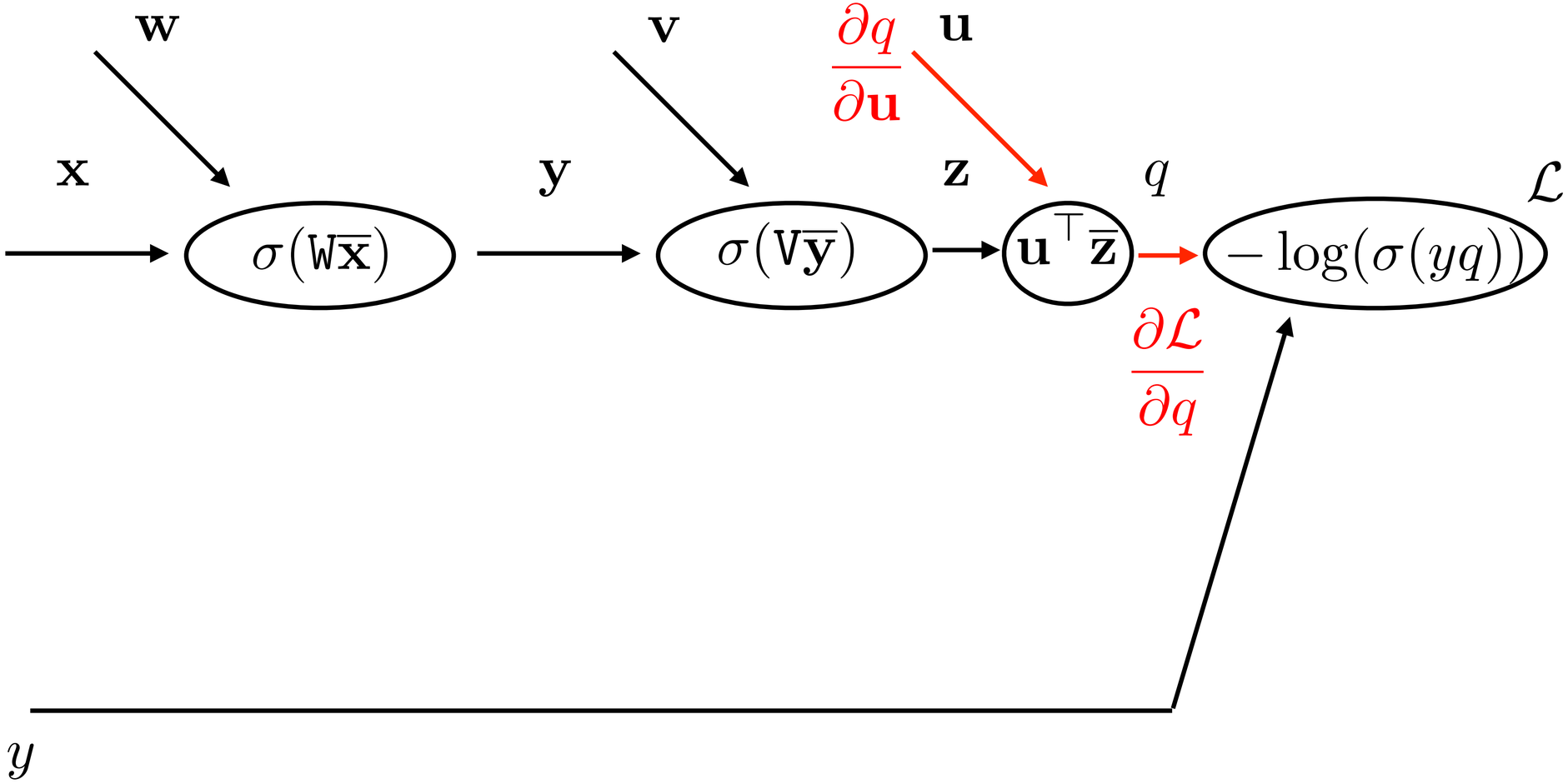
Chainrule in fully-connected neural network

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



Chainrule in fully-connected neural network

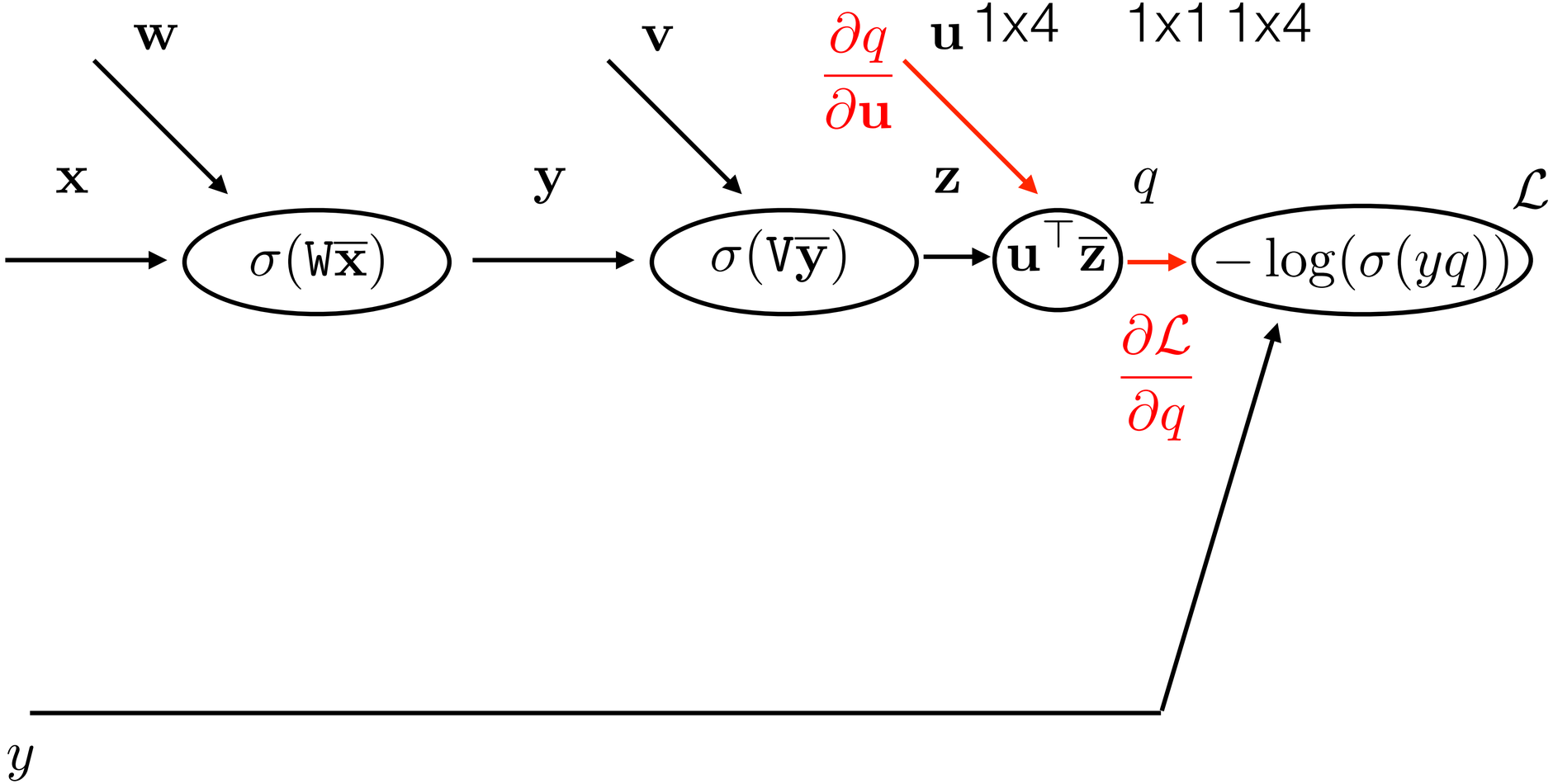
Derivative wrt $\mathbf{u}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{u}} = \frac{\partial \mathcal{L}}{\partial q} \frac{\partial q}{\partial \mathbf{u}}$



Chainrule in fully-connected neural network

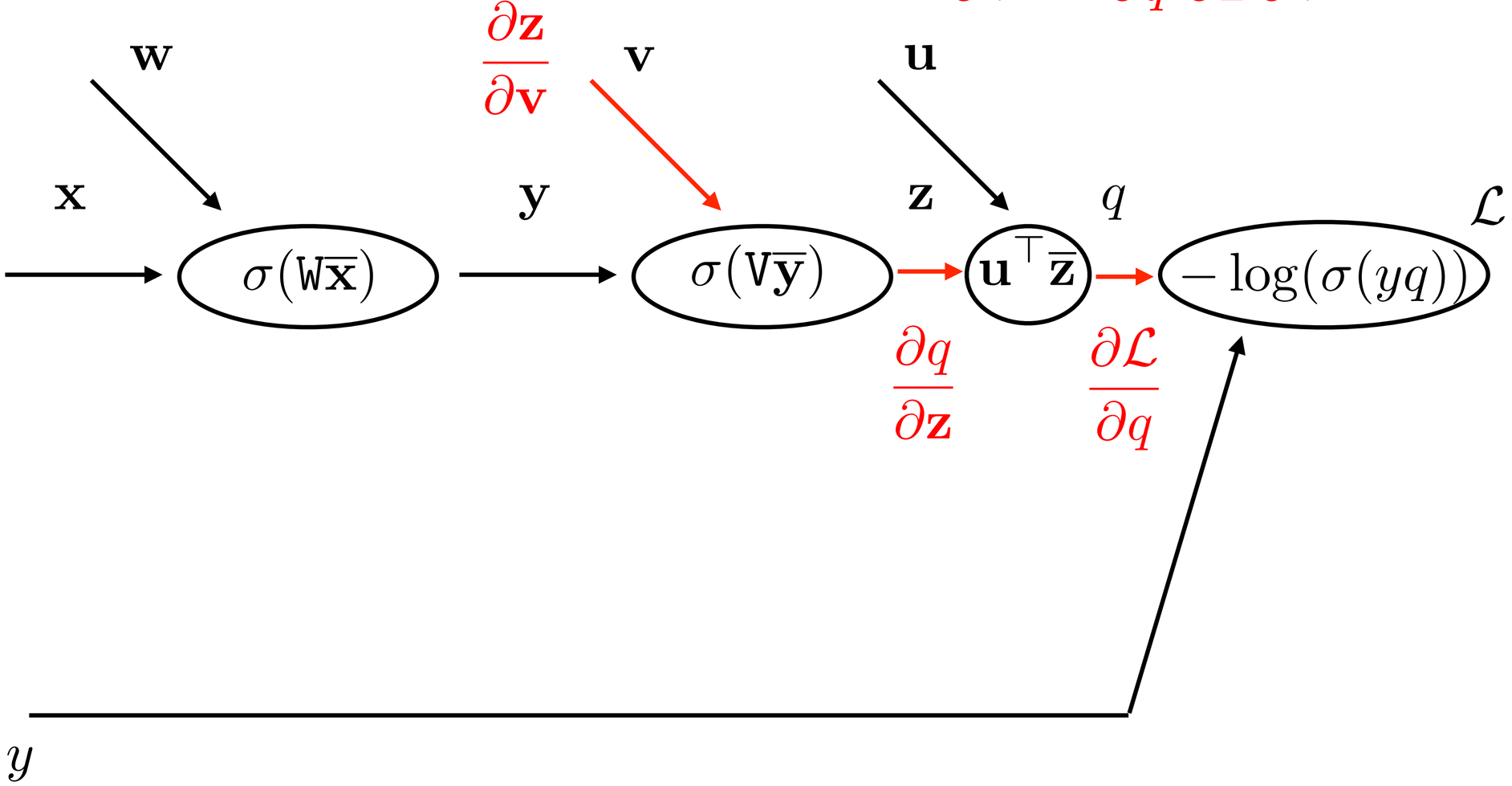
Derivative wrt $\mathbf{u}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{u}} = \frac{\partial \mathcal{L}}{\partial q}\frac{\partial q}{\partial \mathbf{u}}$

1x4 1x1 1x4



Chainrule in fully-connected neural network

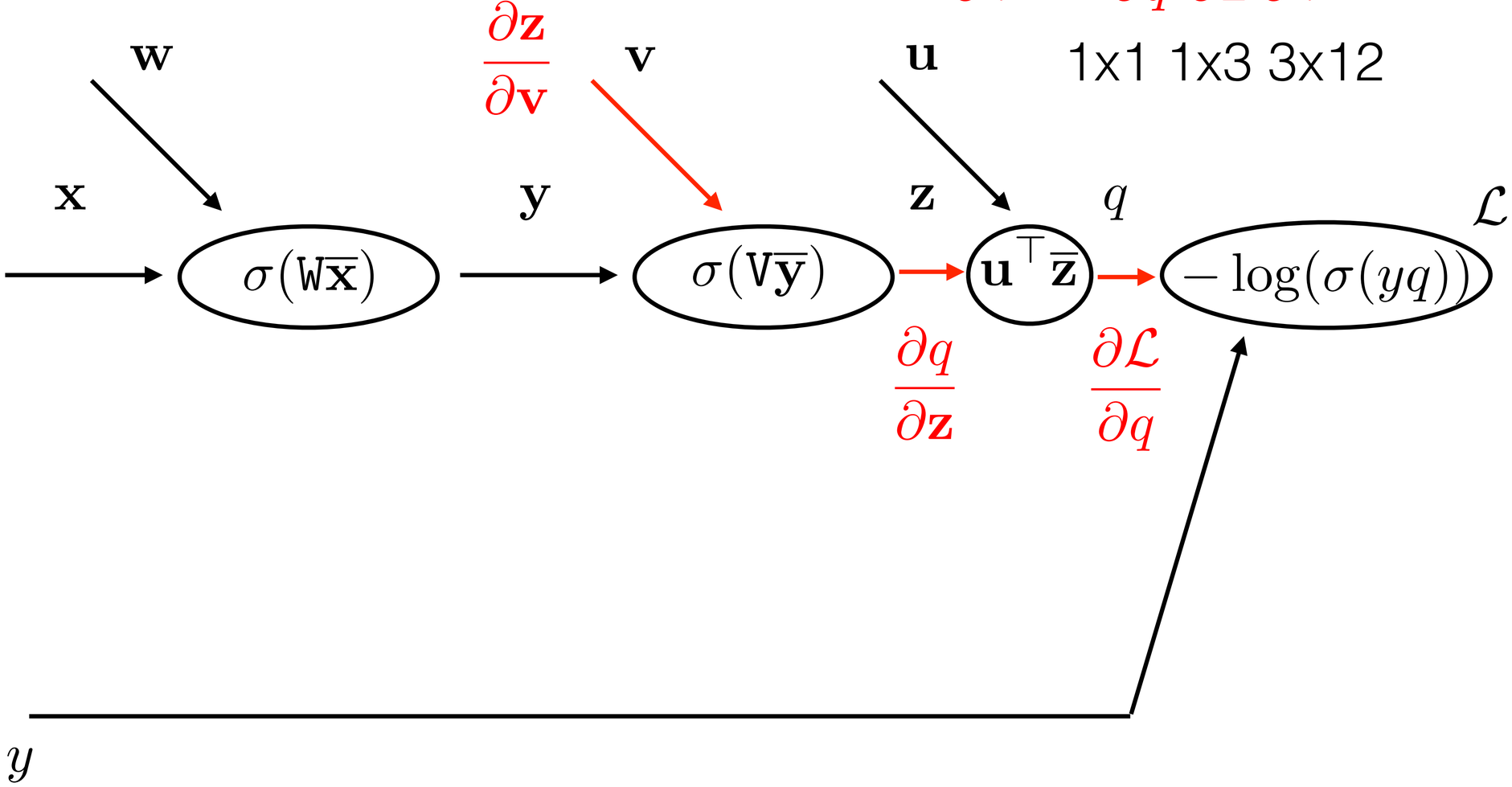
Derivative wrt $\mathbf{v}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{v}} = \frac{\partial \mathcal{L}}{\partial q} \frac{\partial q}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{v}}$



Chainrule in fully-connected neural network

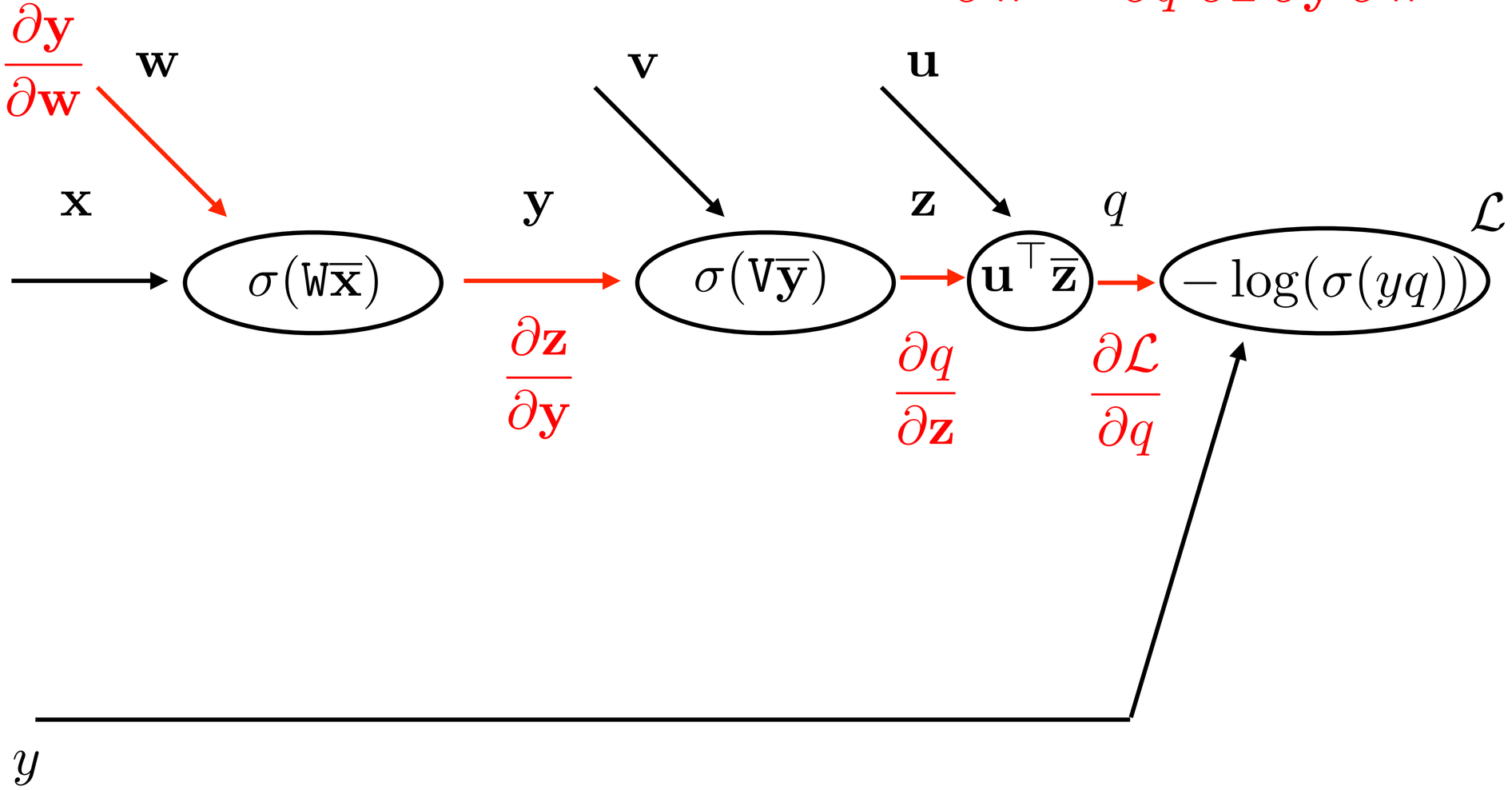
Derivative wrt $\mathbf{v}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{v}} = \frac{\partial \mathcal{L}}{\partial q}\frac{\partial q}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{v}}$

1x1 1x3 3x12



Chainrule in fully-connected neural network

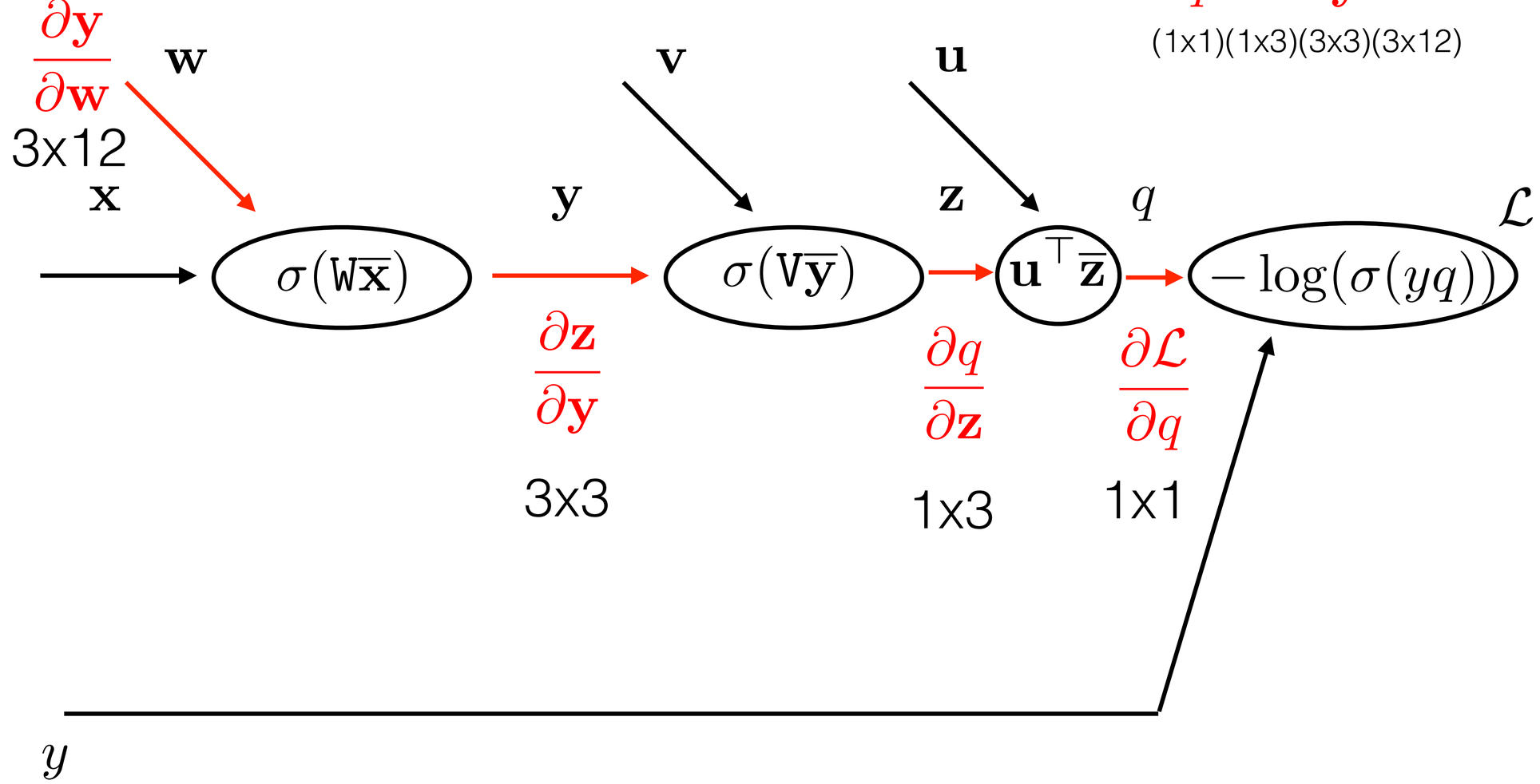
Derivative wrt $\mathbf{w}$: $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial q}\frac{\partial q}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{w}}$



Chainrule in fully-connected neural network

Derivative 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)



Chainrule in fully-connected neural network

1. Estimate all required jacobians
2. Update weights:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{u}} = \frac{\partial \mathcal{L}}{\partial q}\frac{\partial q}{\partial \mathbf{u}} \qquad \mathbf{u} = \mathbf{u} - \alpha \left[\frac{\partial \mathcal{L}}{\partial \mathbf{u}}\right]^\top$$

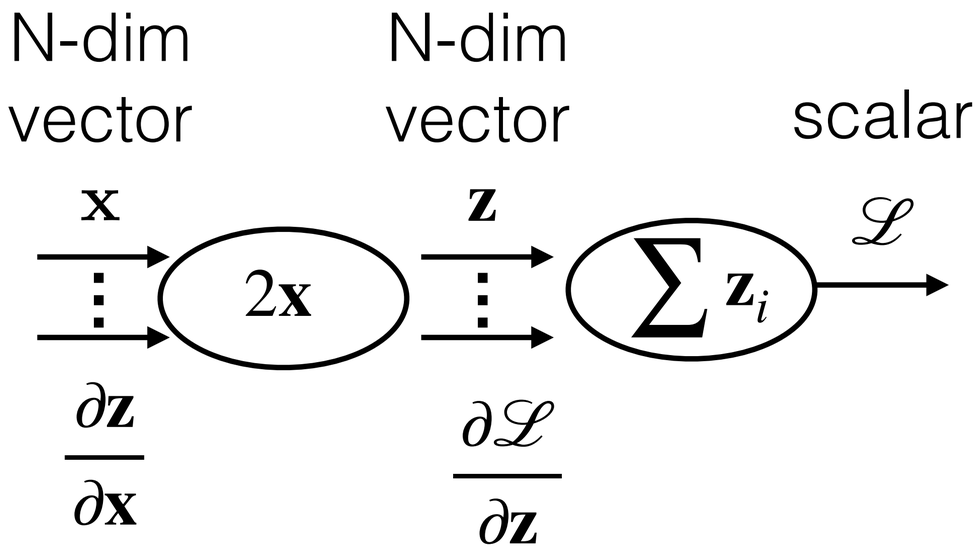
$$\frac{\partial \mathcal{L}}{\partial \mathbf{v}} = \frac{\partial \mathcal{L}}{\partial q}\frac{\partial q}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{v}} \qquad \mathbf{v} = \mathbf{v} - \alpha \left[\frac{\partial \mathcal{L}}{\partial \mathbf{v}}\right]^\top$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial q}\frac{\partial q}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{w}} \qquad \mathbf{w} = \mathbf{w} - \alpha \left[\frac{\partial \mathcal{L}}{\partial \mathbf{w}}\right]^\top$$

3. Optionally update learning rate α
4. Repeat until convergence

Particular gradients can be obtained by multiplying jacobians, however it is inefficient => slightly different implementation

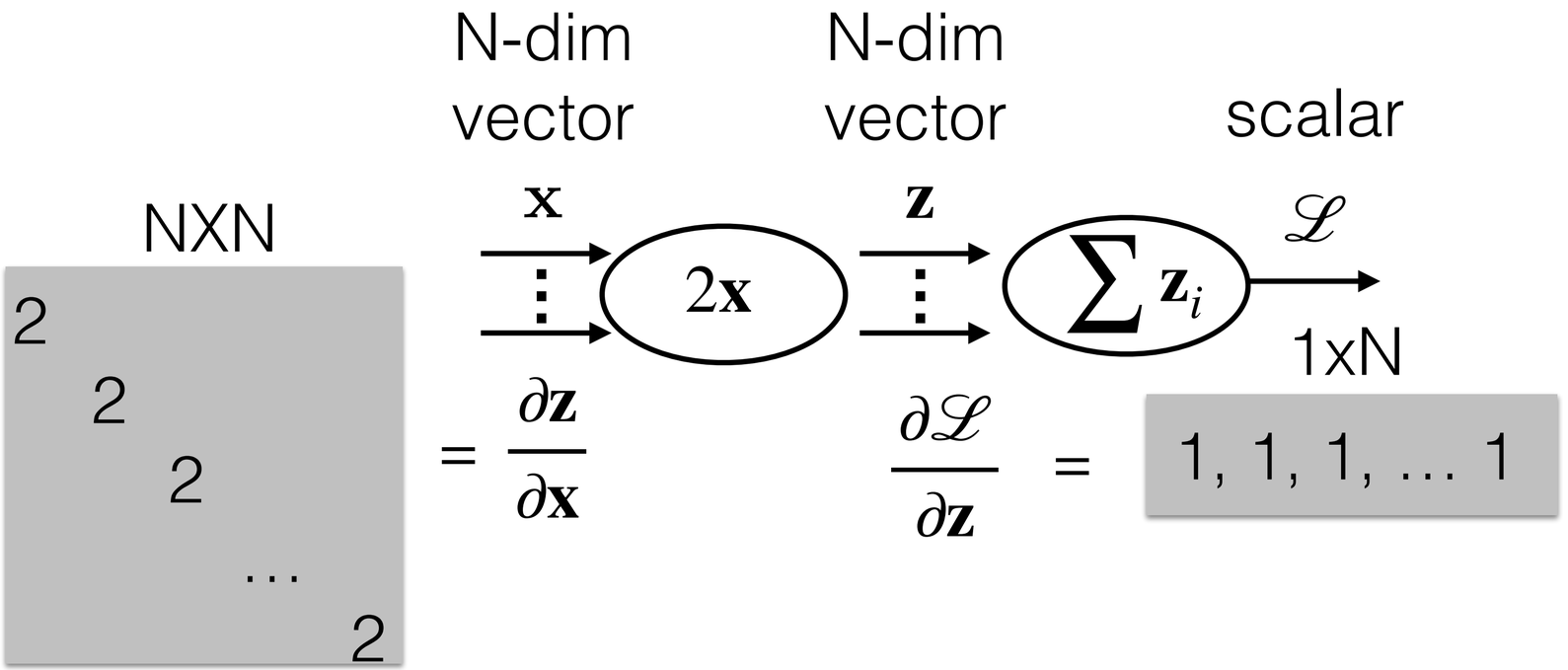
Backprop in Pytorch avoids Jacobians



$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{x}}$$

Jacobian 1xN = Jacobian 1xN · Jacobian NxN

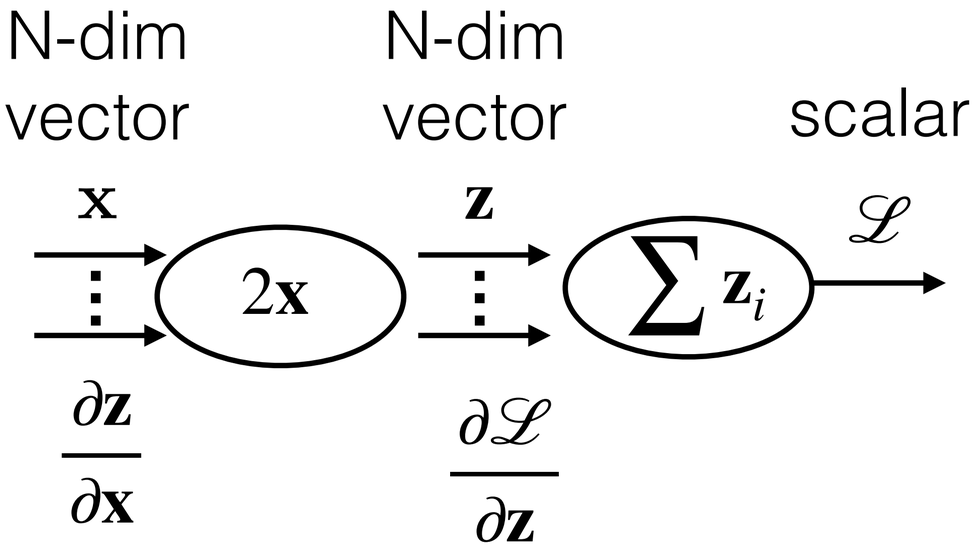
Backprop in Pytorch avoids Jacobians



$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{x}}$$

Jacobian 1xN = Jacobian 1xN · Jacobian NxN

Backprop in Pytorch avoids Jacobians



NXN

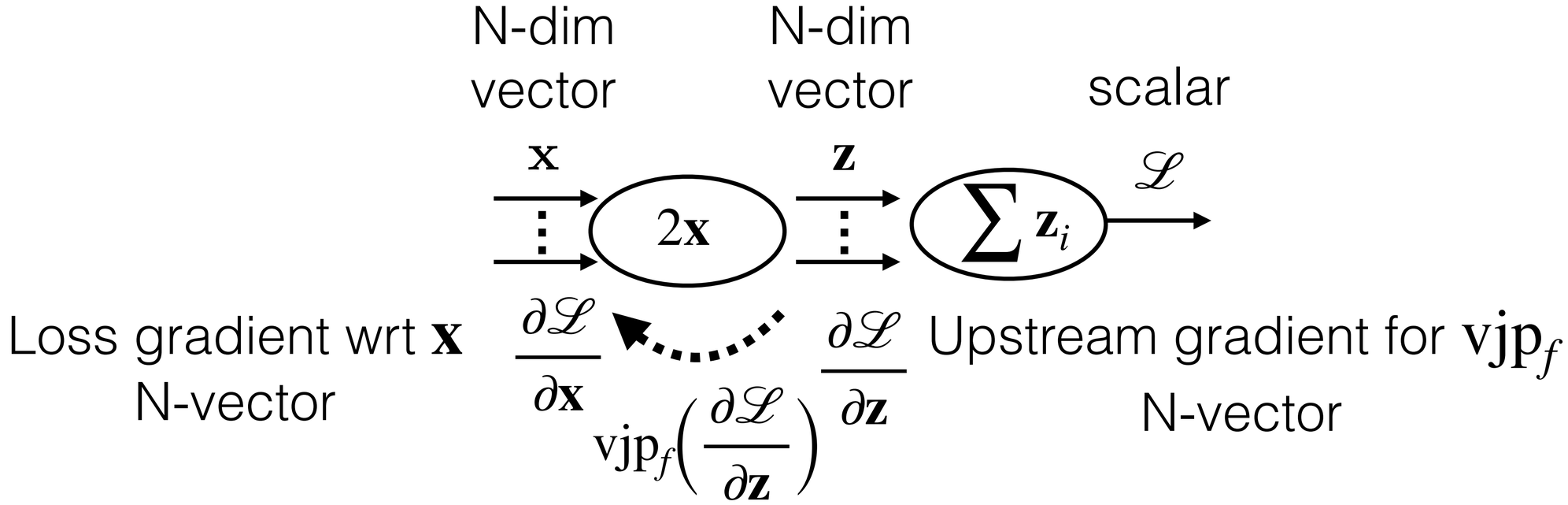
1xN

1xN

$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \begin{bmatrix} 1, 1, 1, \dots 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & & & \\ & 2 & & \\ & & 2 & \\ & & & \dots \\ & & & & 2 \end{bmatrix} = \begin{bmatrix} 2, 2, 2, \dots 2 \end{bmatrix}$$

Jacobian 1xN = Jacobian 1xN · Jacobian NxN

Backprop in Pytorch avoids Jacobians



$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \mathrm{vjp}_f\left(\ 1, 1, 1, \ldots 1\ \right) = 2, 2, 2, \ldots 2$$

Loss gradient
N-vector

vector-Jacobian product
N-vector -> N-vector

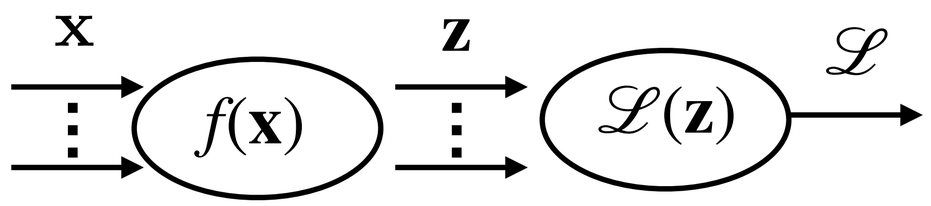
Loss gradient
N-vector

Backprop in Pytorch avoids Jacobians

M-dim vector

N-dim vector

scalar



Prohibitively time-consuming & memory-consuming

N=M=1000=>10^6

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} \qquad \frac{\partial \mathscr{L}}{\partial \mathbf{z}}$$

$$\mathrm{vjp}_f\left(\frac{\partial \mathscr{L}}{\partial \mathbf{z}}\right)$$

This product can be implemented

significantly faster in $\mathrm{vjp}_f\left(\frac{\partial \mathscr{L}}{\partial \mathbf{z}}\right)$

$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{x}}$$

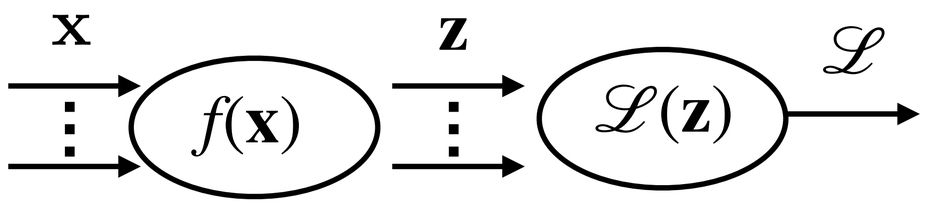
Jacobian 1xM = Jacobian 1xN · Jacobian NxM

Backprop in Pytorch avoids Jacobians

M-dim vector

N-dim vector

scalar



Loss gradient M-dim vector

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

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

Upstream gradient N-dim vector

$\text{vjp}_f\left(\frac{\partial \mathscr{L}}{\partial \mathbf{z}}\right)$

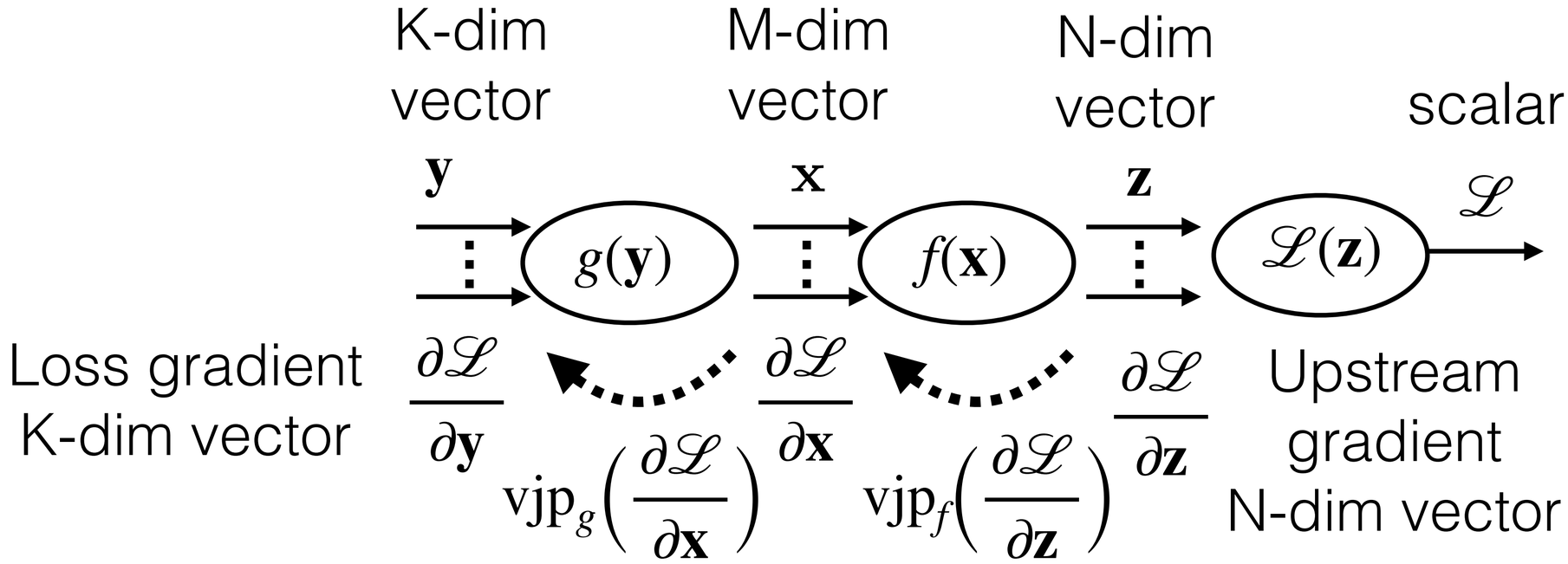
`grad_fn=<fBackward>`

$$\frac{\partial \mathscr{L}}{\partial \mathbf{x}} = \text{vjp}_f\left(\frac{\partial \mathscr{L}}{\partial \mathbf{z}}\right)$$

M-dim vector

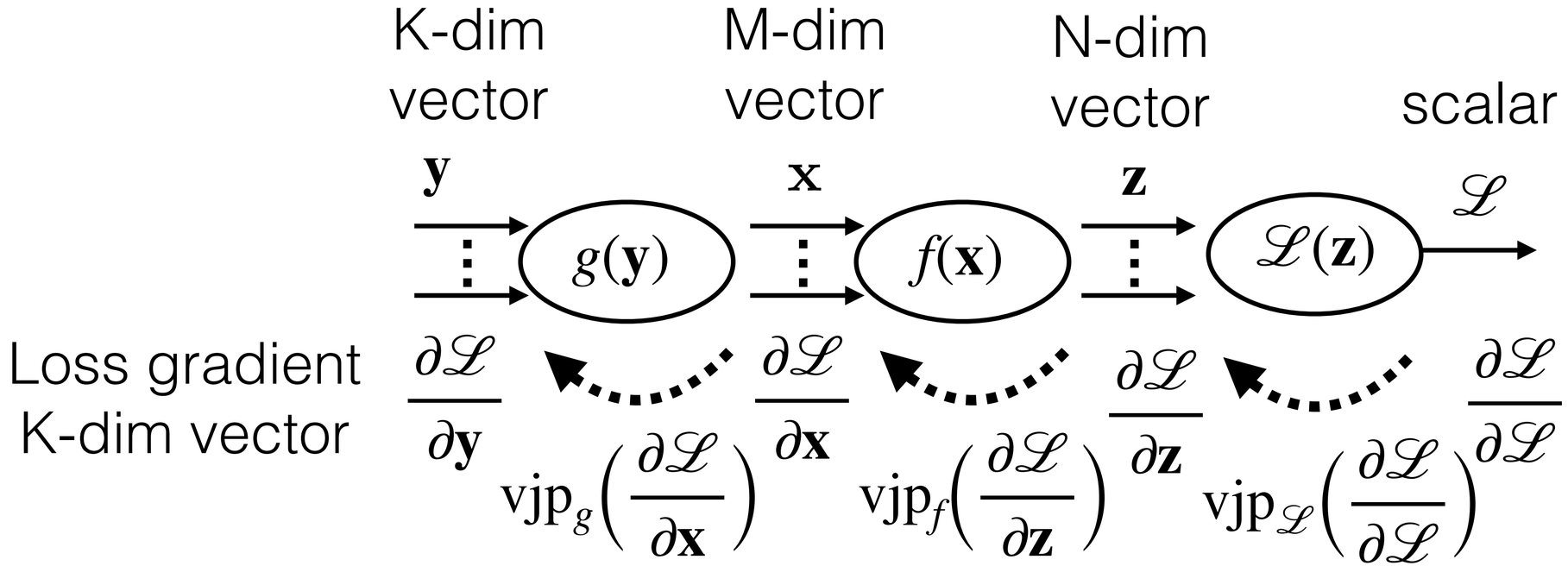
vector-Jacobian product

Backprop in Pytorch avoids Jacobians



$$\underbrace{\frac{\partial \mathscr{L}}{\partial \mathbf{y}}}_{\text{K-dim vector}} = \underbrace{\text{vjp}_g\left(\text{vjp}_f\left(\frac{\partial \mathscr{L}}{\partial \mathbf{z}}\right)\right)}_{\text{vector-Jacobian product}}$$

Backprop in Pytorch avoids Jacobians

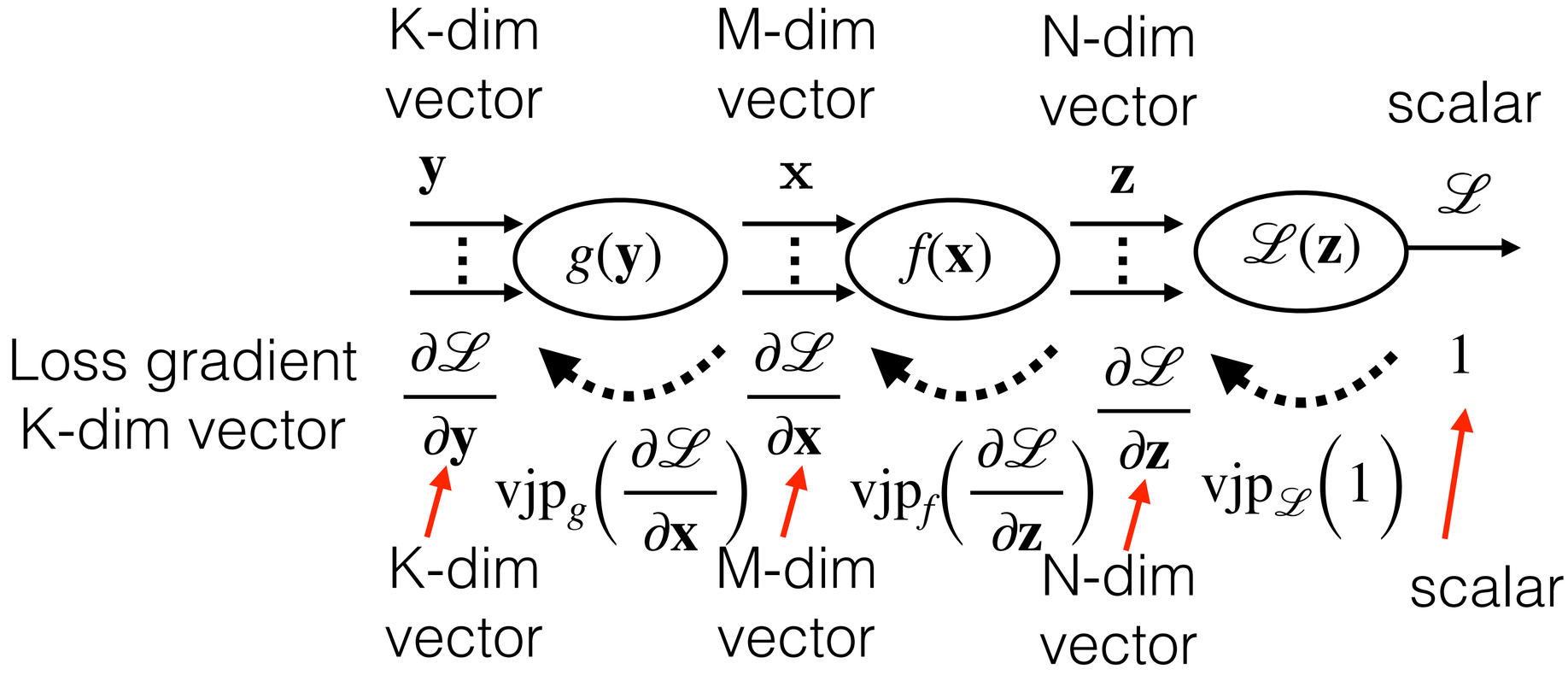


$$\frac{\partial \mathscr{L}}{\partial \mathbf{y}} = \text{vjp}_g\left(\text{vjp}_f\left(\text{vjp}_{\mathscr{L}}\left(\frac{\partial \mathscr{L}}{\partial \mathscr{L}}\right)\right)\right)$$

K-dim vector

vector-Jacobian product

Backprop in Pytorch avoids Jacobians



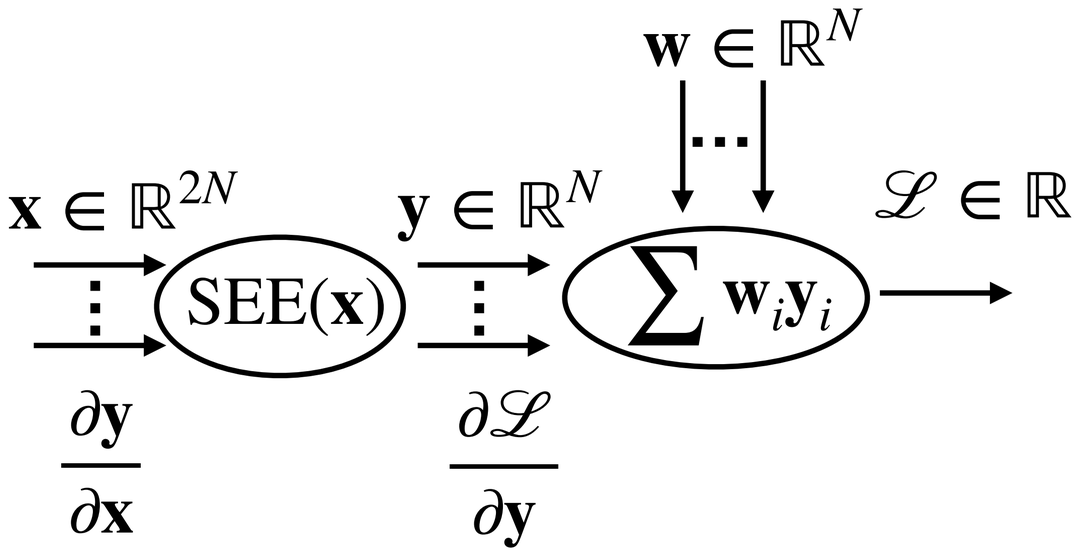
$$\frac{\partial \mathscr{L}}{\partial \mathbf{y}} = \text{vjp}_g\left(\text{vjp}_f\left(\text{vjp}_{\mathscr{L}}(1)\right)\right)$$

K-dim vector

vector-Jacobian product

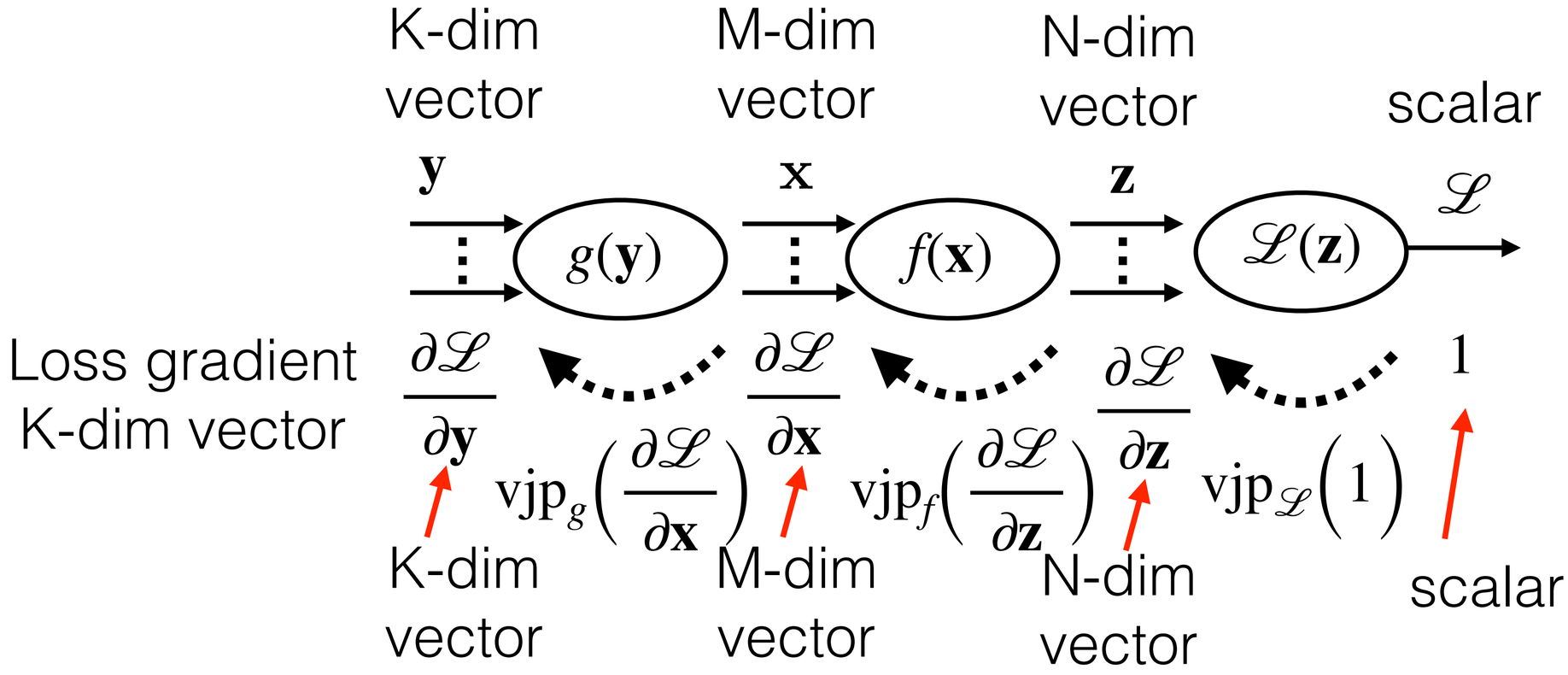
Example: Jacobian vs vector-jacobian-product function

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



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

Backprop in Pytorch avoids Jacobians



$$\frac{\partial \mathscr{L}}{\partial \mathbf{y}} = \text{vjp}_g\left(\text{vjp}_f\left(\text{vjp}_{\mathscr{L}}(1)\right)\right)$$

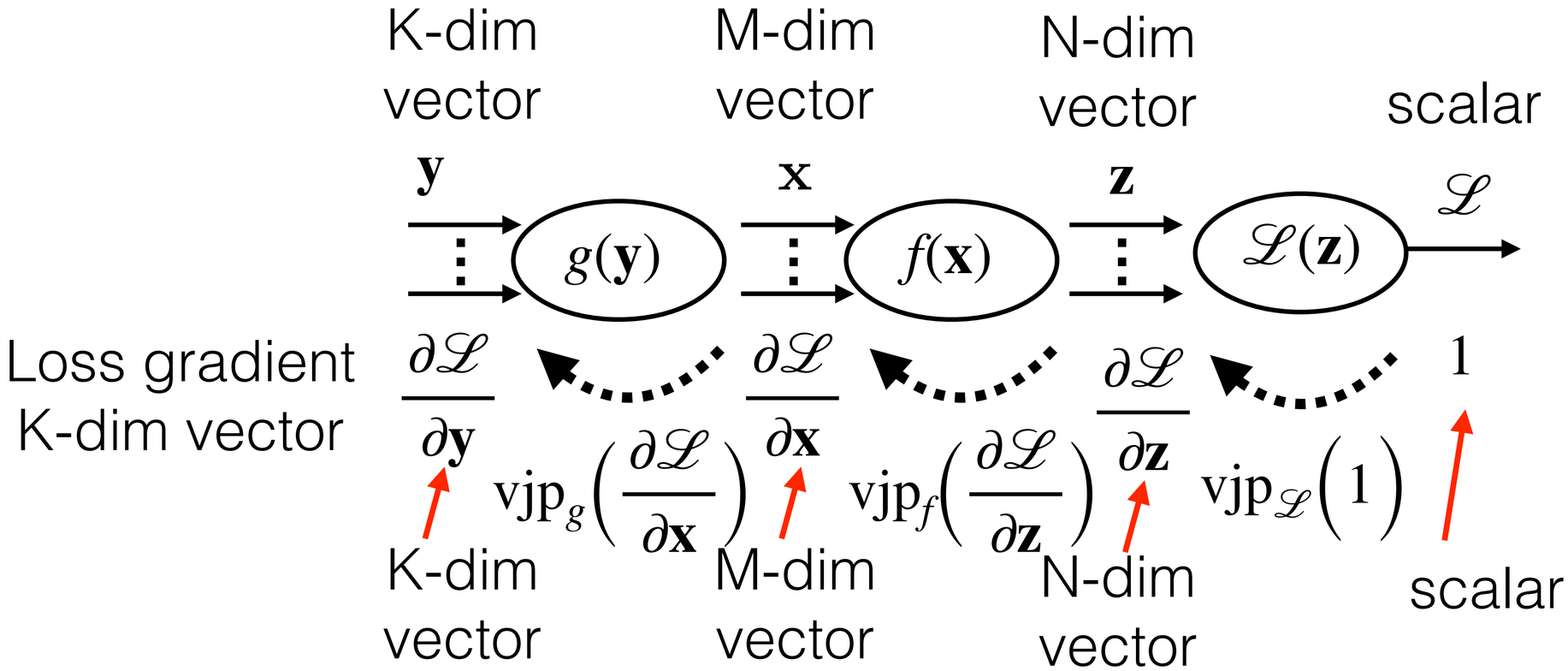
K-dim vector

vector-Jacobian product

Backprop in Pytorch avoids Jacobians

Vector-Jacobian products allows to preserve data dimensionality (avoid vectorization)

=>Gradient in Pytorch is the tensor of the same dimensionality as the input.

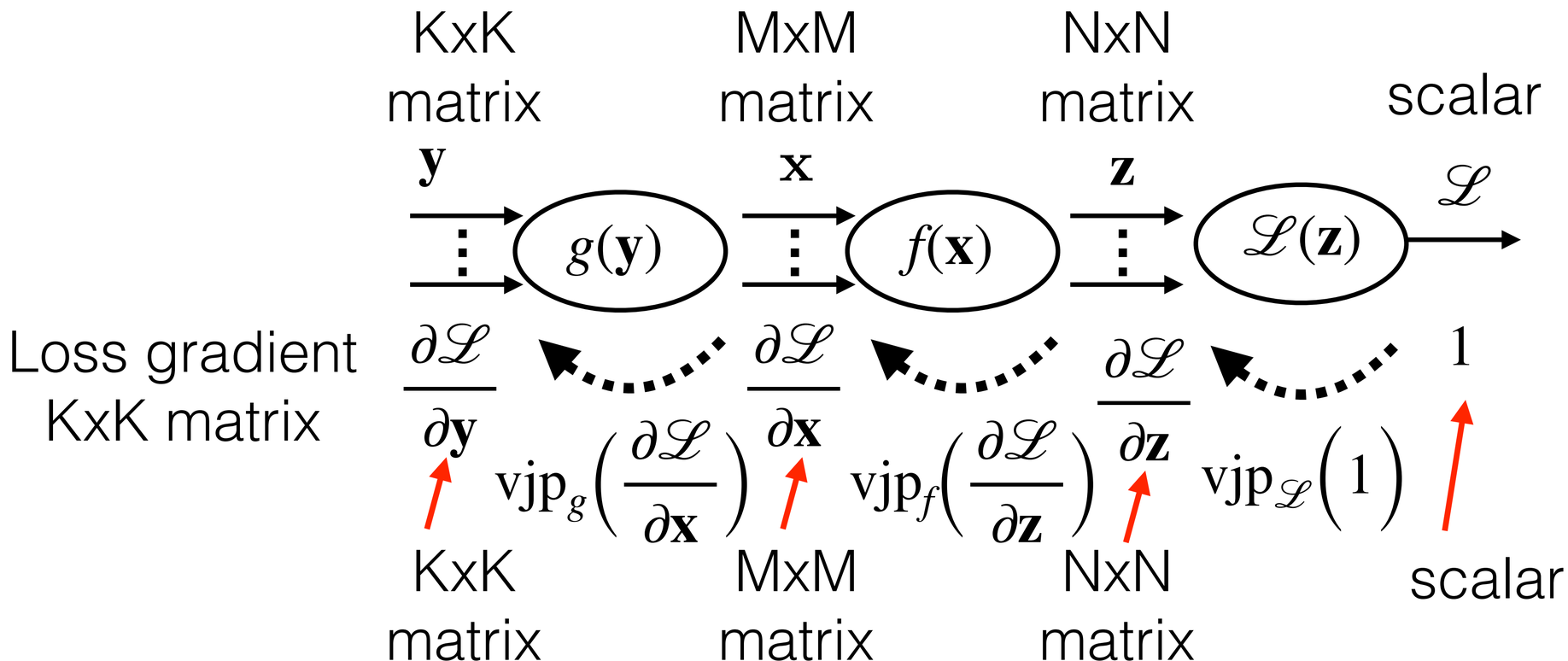


Backprop in Pytorch avoids Jacobians

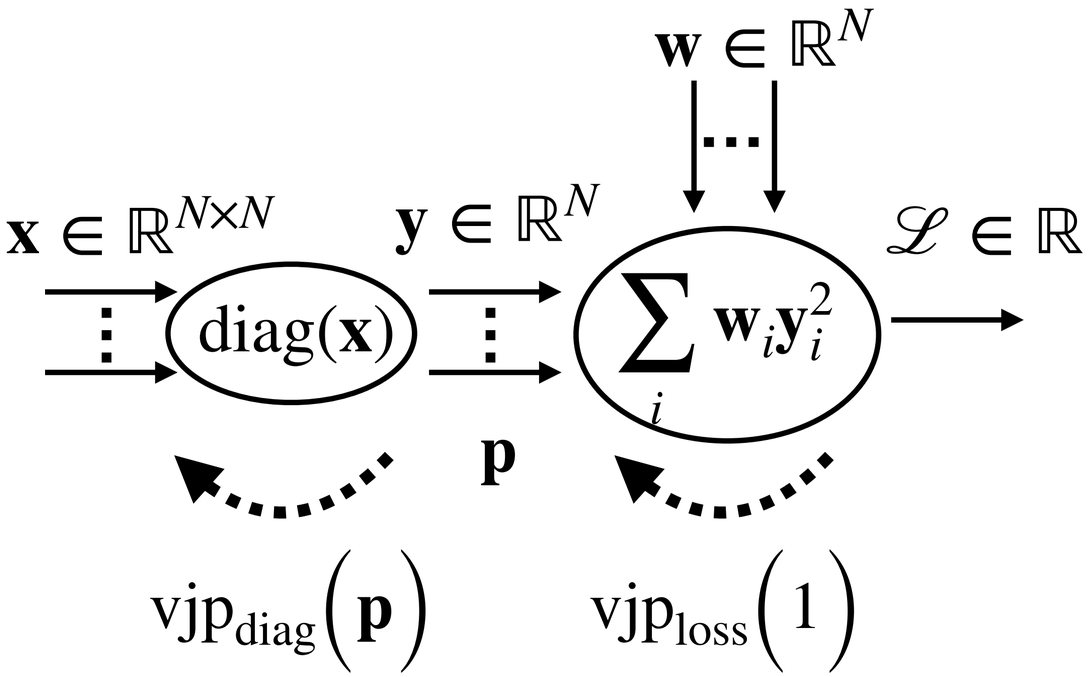
Vector-Jacobian products allows to preserve data dimensionality (avoid vectorization)

=>Gradient in Pytorch is the tensor of the same dimensionality as the input.

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

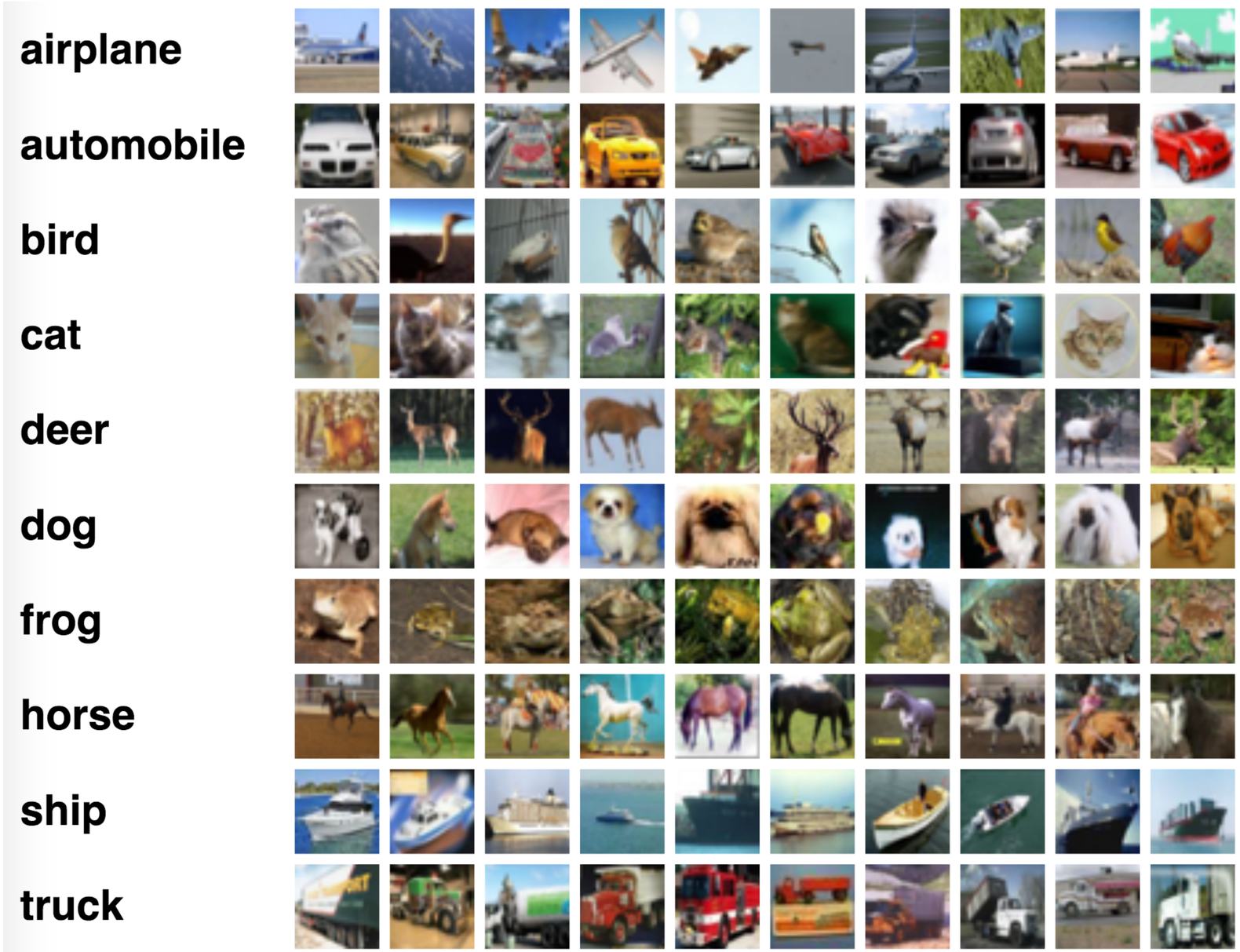


Example: Jacobian vs vector-jacobian-product function

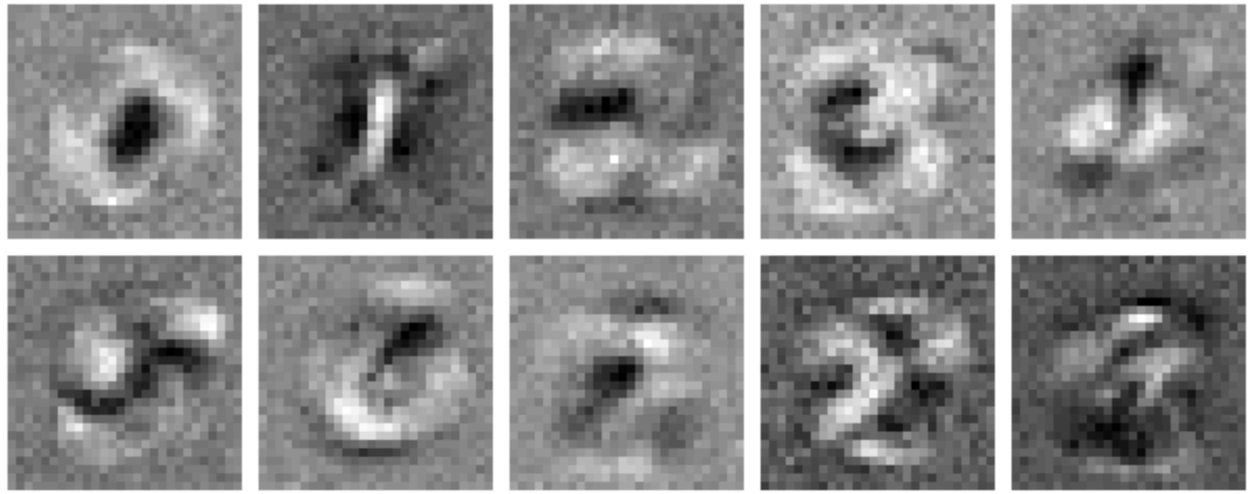
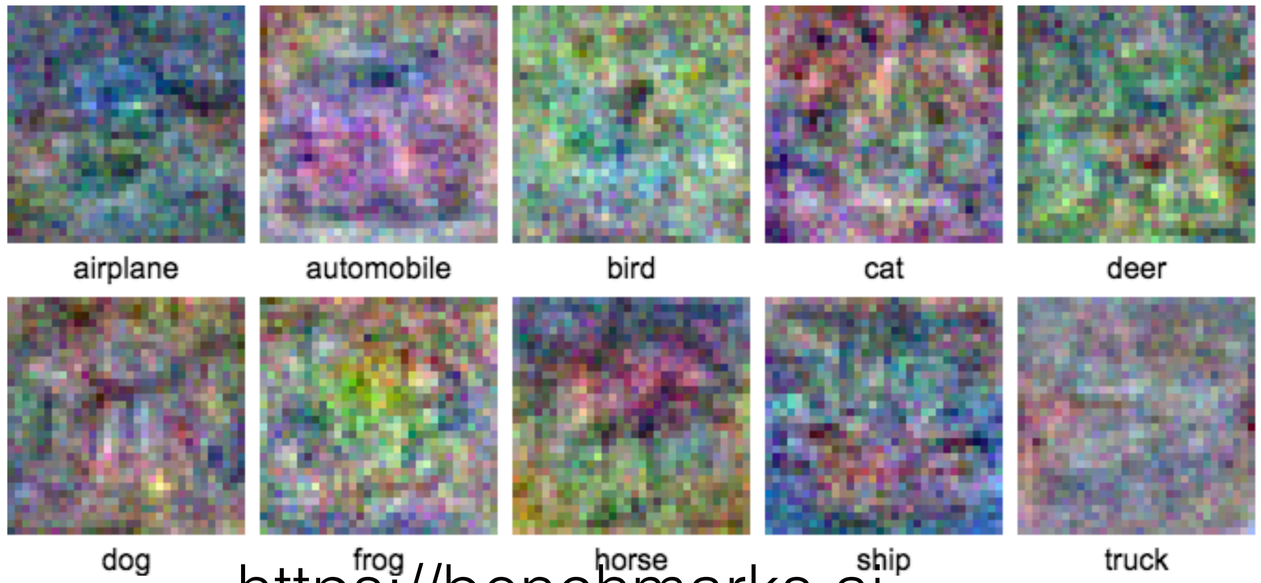


Neural nets summary

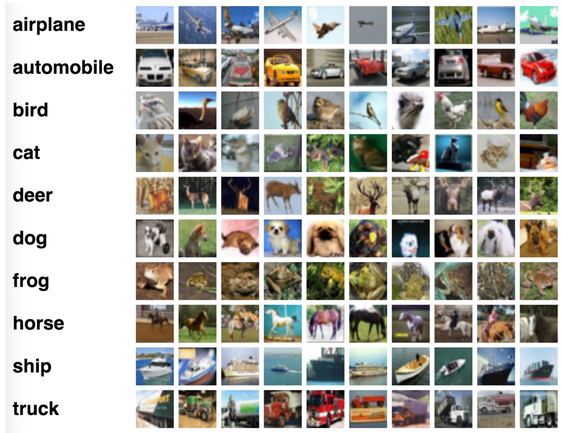
- **Neural net** is a function created as concatenation of simplier functions (e.g. neurons or layers of neurons)
- **Fully connected neural net** is neural network created from neurons, where **all** outputs from previous layer are connected to **all** inputs of the following layer
- **Backpropagation** is gradient optimization of a neural net concatenated with a loss-layer on a training set.
- It is impleneted as backward concatenation of vector-jacobian functions.
- **Deep learning frameworks** (Pytorch, Tensorflow) has many predefined layers.
- **Spoiler alert:** Fully connected NN does not work on structural data (images, sound) well.



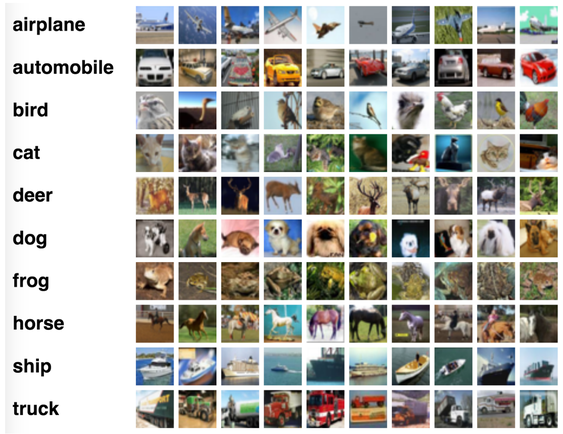
CIFAR-10: classify 32x32 RGB images into 10 categories
https://www.cs.toronto.edu/~kriz/cifar.html

Dataset	Learned weights of linear classifier	Error
MNIST		8%
CIFAR-10		63%

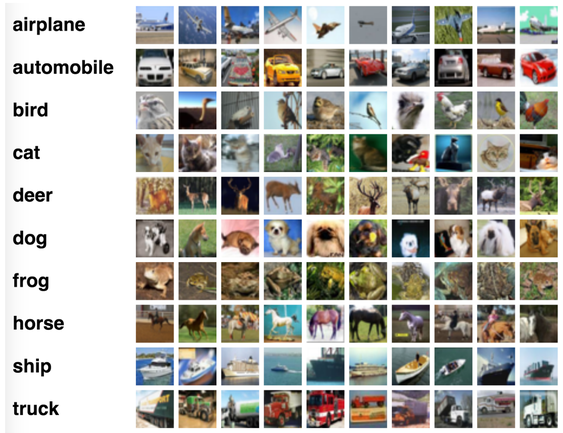
https://benchmarks.ai

Dataset	Linear Error
MNIST 	8%
CIFAR-10 	63%

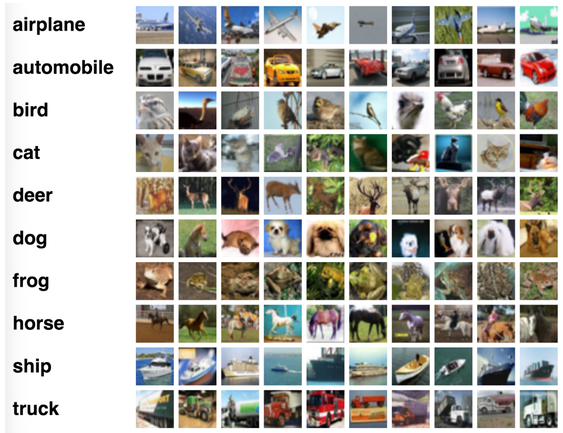
https://benchmarks.ai

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

https://benchmarks.ai

Dataset		Error		
		Linear	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	overfit	cortex inspired structure
CIFAR-10		63%	55%	1% [EfficientNet, 2018]

https://benchmarks.ai

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \left(\sum_i -\log(p(y_i|\mathbf{x}_i, \mathbf{w})) \right) + (-\log p(\mathbf{w}))$$

loss function prior/regulariser

- Class of function represented by a FCNN is too general (it allows for many leprechauns).
- Naive regulariser helps a bit, but dimensionality/wildness is huge => curse-of-dimensionality, overfitting,…
- What is number of weights between two 1000-neuron layers?
- **Next lecture:** study animal cortex to find a stronger prior on the class of suitable functions.
- **Spoiler alert 2:**
  reduce very general class of functions ”neuron layer” to very specific sub-class of functions “convolution layer”

Competencies required for the test T1

- Ability to draw a computational graph.
- Compute edge gradients/jacobians.
- Perform one step of backpropagation in a vectorized form