EMBEDDED MACHINE LEARNING LECTURE 04 - AUTOMATIC DIFFERENTIATION & OPTIMIZATION

Holger Fröning
holger.froening@ziti.uni-heidelberg.de
HAWAII Group (formerly Computing Systems), Institute of Computer Engineering
Heidelberg University

RECAP: BACK PROP ON ONE SLIDE

Data set containing N input-target pairs: $\mathscr{D} = \{(\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_N, t_N)\}$

Training ANNs: adjust randomly initialized weights $\mathbf{W}$ to solve a given task by minimizing a loss function $\mathscr{L}$ using gradient-based optimization

$$\mathscr{L}(\mathbf{W}; \mathscr{D}) = \sum_{n=1}^{N} l(y(\mathbf{W}, \mathbf{x}_n), t_n) + \lambda r(\mathbf{W});$$

based on a data term l that penalizes wrong prediction (error function); and

for a regularizer $r(\mathbf{W})$ such as ℓ^1–norm or ℓ^2–norm and a trade-off hyperparameter λ

Backpropagation: compute gradient for input-target pair and minimize the loss function by iteratively calculating

$\mathbf{W} := \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathscr{L}(\mathbf{W}; \mathscr{D})$, for $\nabla_{\mathbf{x}} = \left(\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n}\right)$ and learning rate η

Key operations: partial derivative and all-reduce

RECAP: GRADIENT BEHAVIOR

add gate: gradient distributor

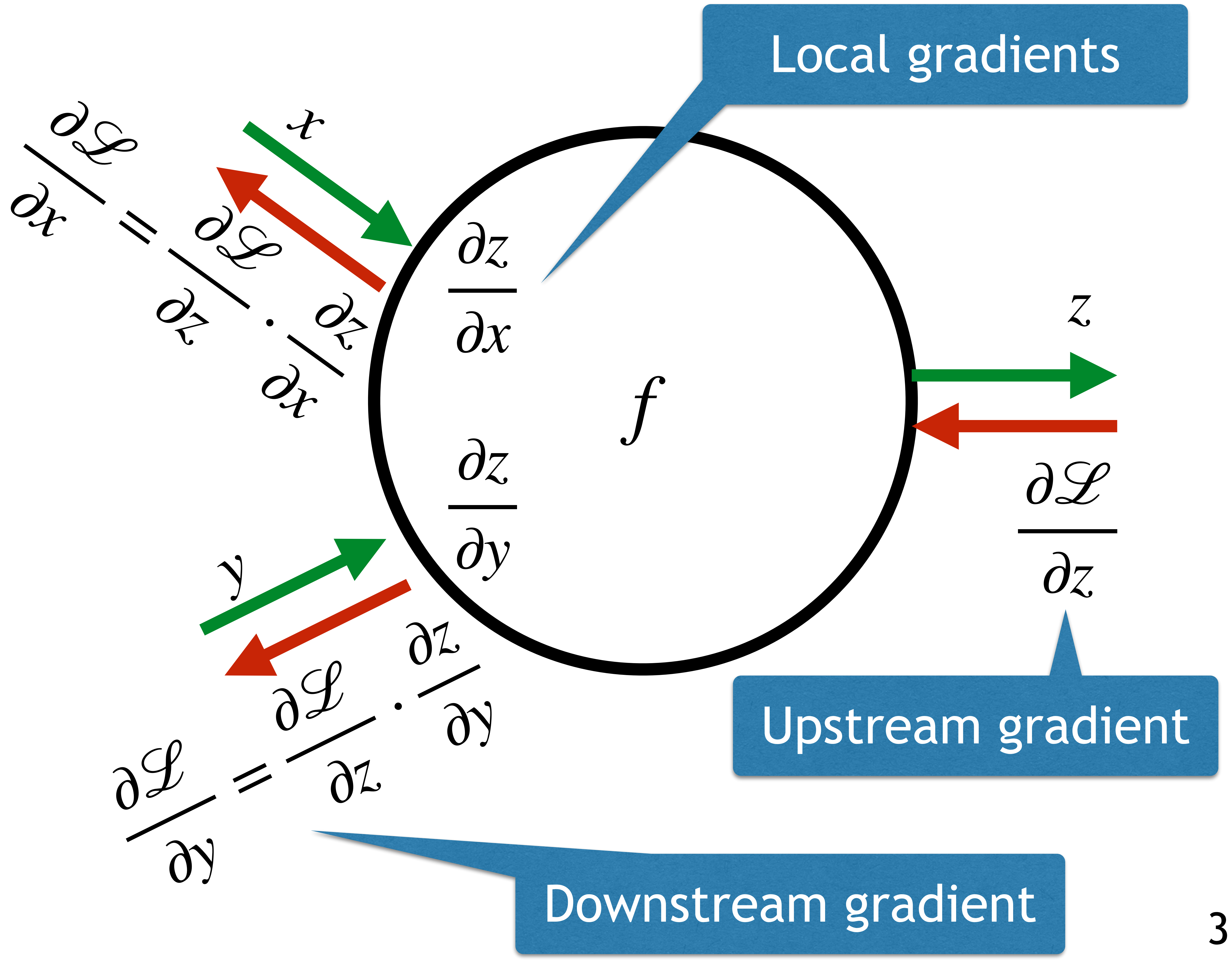
- Both downstream gradients equal upstream gradient

mul gate: gradient switcher

- Downstream gradients multiplied with other input

max gate: gradient router

- Downstream gradients depend on comparison of inputs


Local gradients
Upstream gradient
Downstream gradient
x
y
z
f
$\frac{\partial z}{\partial x}$
$\frac{\partial z}{\partial y}$
$\frac{\partial \mathscr{L}}{\partial z}$
$\frac{\partial \mathscr{L}}{\partial x} = \frac{\partial \mathscr{L}}{\partial z} \cdot \frac{\partial z}{\partial x}$
$\frac{\partial \mathscr{L}}{\partial y} = \frac{\partial \mathscr{L}}{\partial z} \cdot \frac{\partial z}{\partial y}$

BACK TO NEURAL NETWORKS

$$\mathbf{y}(\mathbf{W}, \mathbf{x}_0) = \mathbf{x}_L = f(\mathbf{W}_L \oplus f(\mathbf{W}_{L-1} \oplus (\ldots \oplus f(\mathbf{W}_1 \oplus \mathbf{x}_0)\ldots)))$$

$$\underbrace{\underbrace{\underbrace{\frac{\partial l}{\partial \mathbf{x}_0} = \frac{\partial l}{\partial \mathbf{x}_L}}_{} \cdot \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}}}_{} \cdot \ldots \cdot \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}}}_{} \cdot \ldots \cdot \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} \cdot \frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_0}$$

What are efficient forms to compute the gradient for the parameters ($\mathbf{W}$ and $\mathbf{b}$)?

(AUTOMATIC) DIFFERENTIATION

With material from pytorch.org and Roger Grosse (U. Toronto, CSC421/2516)

DIFFERENTIATION IN PYTORCH

Autograd calculates & stores the gradients for each parameter in the parameter's `.grad` attribute

Gradient descent: adjust each parameter by its gradient

```
import torch
from torchvision.models import resnet18, ResNet18_Weights

model = resnet18(weights = ResNet18_Weights.DEFAULT)
# hidden: w = torch.tensor(..., requires_grad=True)

data = torch.rand(1, 3, 64, 64) # example image
labels = torch.rand(1, 1000) # label shape in this example

prediction = model(data) # forward pass

loss = (prediction - labels).sum()
loss.backward() # backward pass

optim = torch.optim.SGD(model.parameters(), lr=1e-2, momentum=0.9)

optim.step() # gradient descent
```

As neural networks are nested functions, we can simplify the calculation of the overall derivative using the chain rule (actually we will need the intermediate derivates anyway)

DIFFERENTIATION IN AUTOGRAD

Creating tensors with `requires_grad=True` signals autograd that every operation on them should be tracked

```
a = torch.tensor([2., 3.], requires_grad=True)
b = torch.tensor([6., 4.], requires_grad=True)
```

Example calculation: `q = 3*a**3 - b**2` resp. $\mathbf{q} = 3\mathbf{a}^3 - \mathbf{b}^2$

Sought gradients $d\mathbf{q}/d\mathbf{a} = 9\mathbf{a}^2$ and $d\mathbf{q}/d\mathbf{b} = -2\mathbf{b}$ are calculated by autograd upon the `.backward` call

As in this example $\mathbf{q}$ is a vector, we need to also make the gradient a vector

```
external_grad = torch.tensor([1., 1.])
q.backward(gradient=external_grad)
```

NEED FOR AUTOMATIC DIFFERENTIATION

Consider a simple model and loss function (scalar values for simplicity)

$$z = wx + b$$

$$y = \sigma(z)$$

$$\mathscr{L}_{reg} = \mathscr{L} + \lambda\mathscr{R} = \frac{1}{2}(y-t)^2 + \frac{\lambda}{2}w^2$$

For calculus class, see right side

- Cumbersome calculations
- Redundant steps (green brackets)
- Redundant terms in the final expressions (red)

Neural networks have many layers and many parameters, thus many ∂w's and ∂b's

$$\mathscr{L}_{reg} = \frac{1}{2}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}w^2$$

$$\begin{aligned}
\frac{\partial \mathscr{L}_{reg}}{\partial w} &= \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}w^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial w}\Big(\sigma(wx+b)-t\Big)^2 + \frac{\lambda}{2}\frac{\partial}{\partial w}w^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial w}(\sigma(wx+b)-t) + \lambda w \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial w}(wx+b) + \lambda w \\
&= (\sigma(\underline{wx+b})-t)\sigma'(\underline{wx+b})x + \lambda w
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \mathscr{L}_{reg}}{\partial b} &= \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx+b)-t)^2 + \frac{\lambda}{2}w^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial b}\Big(\sigma(wx+b)-t\Big)^2 + \frac{\lambda}{2}\frac{\partial}{\partial b}w^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial b}(\sigma(wx+b)-t) + 0 \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial b}(wx+b) \\
&= (\sigma(\underline{wx+b})-t)\sigma'(\underline{wx+b})
\end{aligned}$$

COMPUTATIONAL GRAPH

Consider a computational graph based on vertices and edges

- A vertex is the result of an operation resp. a variable
- An edge is a dependency to a variable (directed from input to output)
- Topological ordering: parents (incl. inputs) come before children (incl. output)

$$z = wx + b,\ y = \sigma(z),\ \mathscr{R} = \frac{1}{2}w^2$$

$$\mathscr{L}_{reg} = \mathscr{L} + \lambda\mathscr{R} = \frac{1}{2}(y - t)^2 + \frac{\lambda}{2}w^2$$

$$\begin{tikzcd}
x \arrow[dr] & & t \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & y \arrow[r] & \mathscr{L} \arrow[r] & \mathscr{L}_{reg} \\
w \arrow[ur, red] \arrow[rrr, red] & & & \mathscr{R} \arrow[ur] &
\end{tikzcd}$$

Start with children (result of the expression)

- Working **backward** through the graph

Note the need for a multivariate derivate for w (red)

RECAP: CHAIN RULE

Backprop requires to know the gradient of loss with respect to various parameters

Chain rule of calculus (with a new notation): $\frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial u}\frac{\partial u}{\partial v} = \overline{u}\frac{\partial u}{\partial v} = \overline{v}$

Emphasizes that $\overline{v}$ ("partial derivate to v") is a quantity we compute, not a mathematical expression to evaluate

Also emphasizes data reuse, as we will store e.g. $\overline{v}$ for future uses

Simplifies for instance the multivariate chain rule

$$\frac{\partial}{\partial v} f\big(a(v), b(v)\big) = \frac{\partial f}{\partial a}\frac{\partial a}{\partial v} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial v}$$

=> $\overline{v} = \frac{\partial \mathcal{L}}{\partial v} = \frac{\partial \mathcal{L}}{\partial a}\frac{\partial a}{\partial v} + \frac{\partial \mathcal{L}}{\partial b}\frac{\partial b}{\partial v} = \overline{a}\frac{\partial a}{\partial v} + \overline{b}\frac{\partial b}{\partial v}$

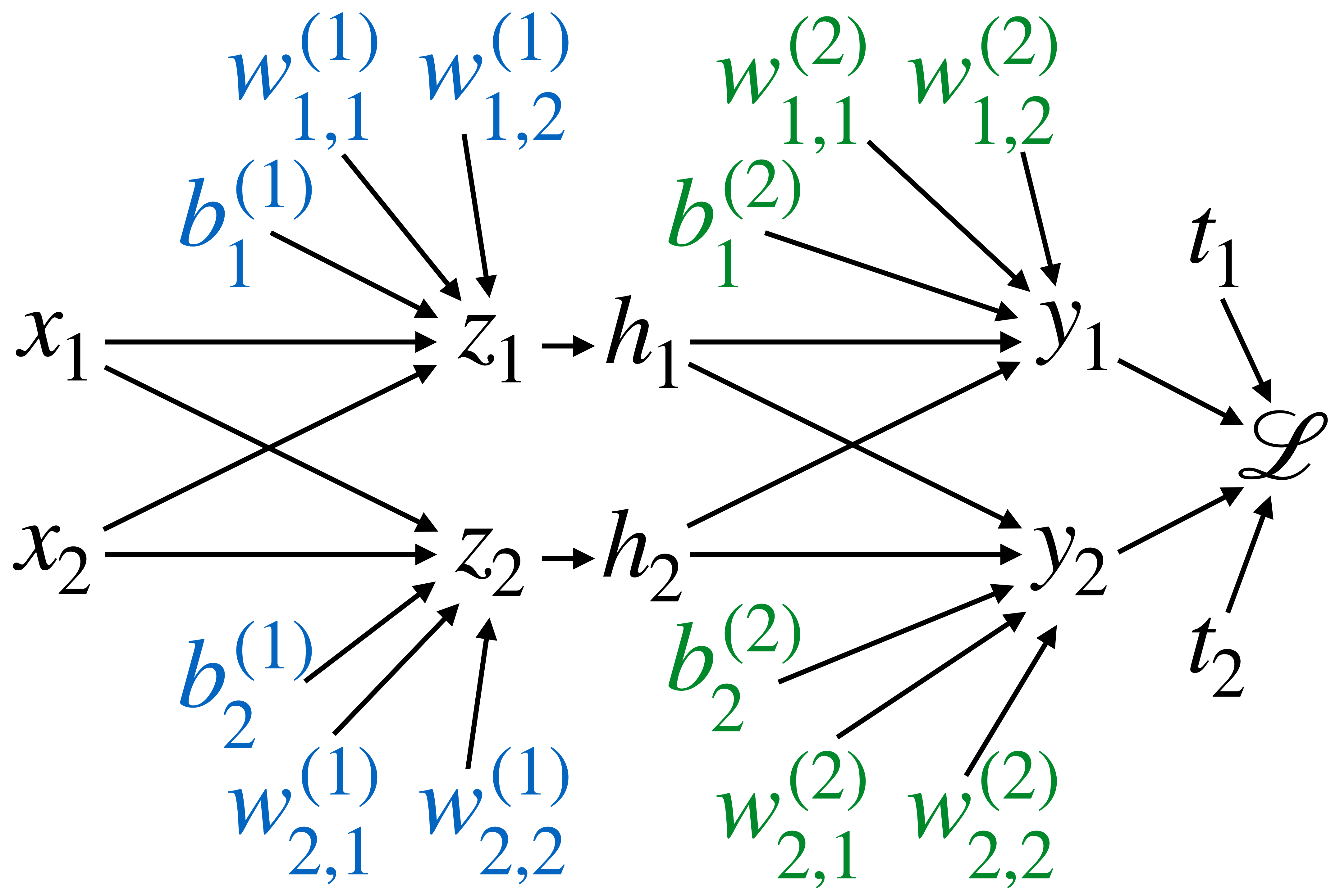
$\frac{\partial \mathcal{L}}{\partial v}$ is an expression to evaluate, while $\overline{v}$ is a previously computed value

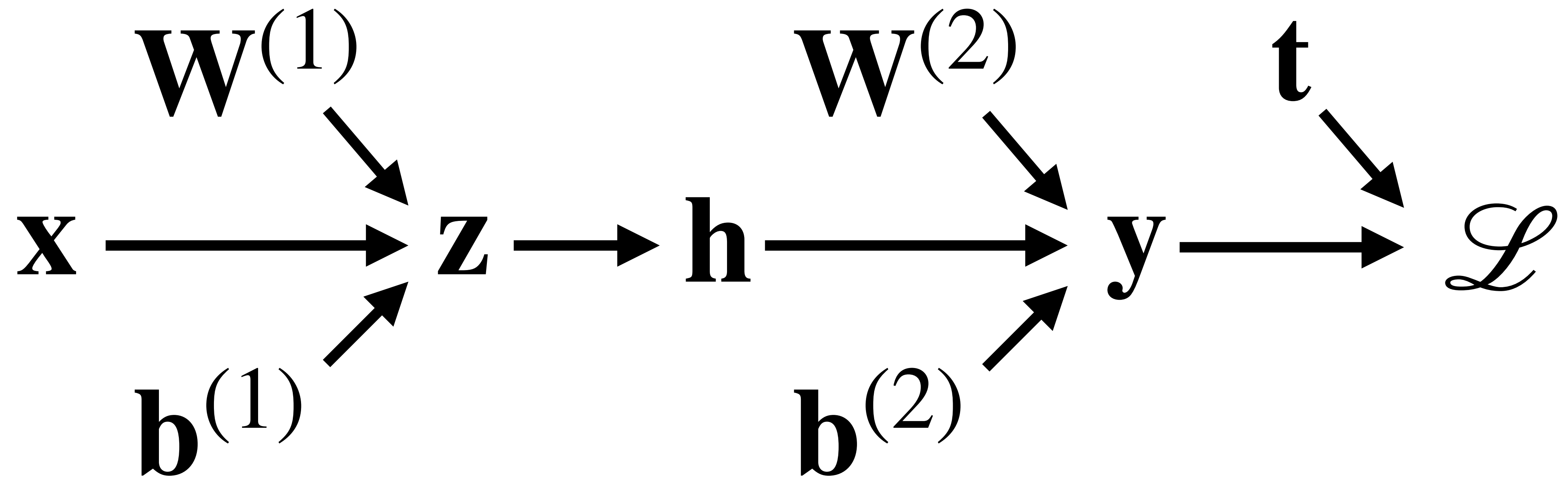
MULTILAYER NETWORKS - SCALAR VIEW

Computing the loss derivatives for multilayer neural networks

- Essentially identical to previous slide; nothing new
- Just more cluttered as networks tend to be deep and consist of millions of parameters

Example of a two-layer NN: $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$; $\mathbf{h} = \sigma(\mathbf{z})$; $\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$; $\mathscr{L} = \frac{1}{2}\|\mathbf{y} - \mathbf{t}\|^2$




$$\begin{tikzcd}
& \mathbf{W}^{(1)} \arrow[dr] & & & \mathbf{W}^{(2)} \arrow[dr] & & \mathbf{t} \arrow[dr] & \\
\mathbf{x} \arrow[rr] & & \mathbf{z} \arrow[r] & \mathbf{h} \arrow[rr] & & \mathbf{y} \arrow[rr] & & \mathscr{L} \\
& \mathbf{b}^{(1)} \arrow[ur] & & & \mathbf{b}^{(2)} \arrow[ur] & & &
\end{tikzcd}$$

MULTILAYER NETWORKS - SCALAR VIEW (1)

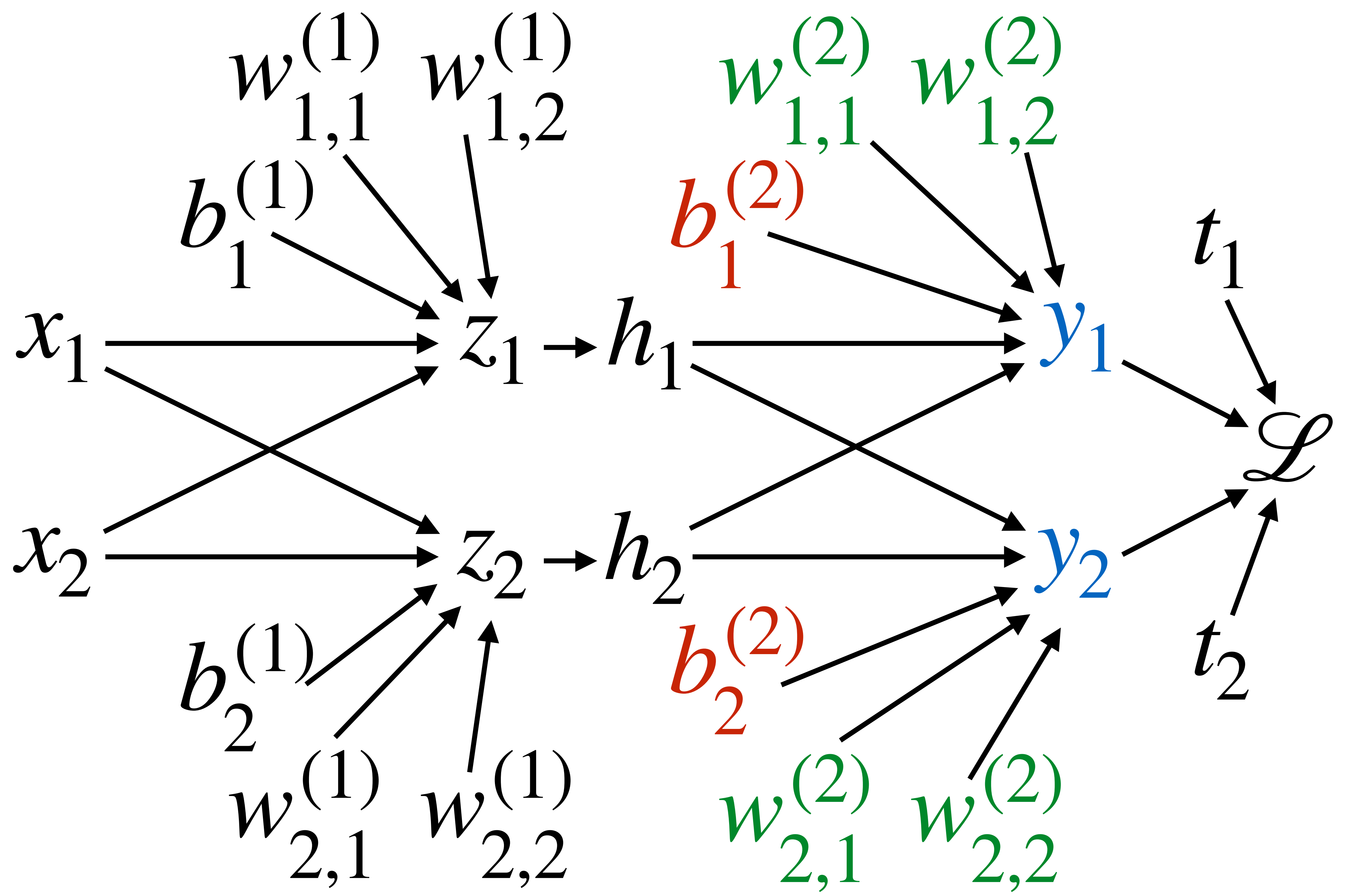
$$y_j = \sum_i (w_{j,i}^{(2)} h_i) + b_j^{(2)}$$

Computing the loss derivatives for multilayer neural networks

- Essentially identical to previous slide; nothing new
- Just more cluttered as networks tend to be deep and consist of millions of parameters

Example of a two-layer NN: $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$; $\mathbf{h} = \sigma(\mathbf{z})$; $\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$; $\mathscr{L} = \frac{1}{2}\|\mathbf{y} - \mathbf{t}\|^2$

Backward path starts with $\overline{\mathscr{L}} = 1$; $\forall k, j$ of the corresponding vector/matrix



$$\overline{y_k} = \frac{\partial \mathscr{L}}{\partial y_k} = \frac{\partial}{\partial y_k}\Big(\frac{1}{2}\sum_i (y_i - t_i)^2\Big) = y_k - t_k$$

$$\overline{w_{k,j}^{(2)}} = \frac{\partial \mathscr{L}}{\partial y_k} \cdot \frac{\partial y_k}{\partial w_{k,j}^{(2)}} = \overline{y_k} \cdot \frac{\partial}{\partial w_{k,j}^{(2)}}\Big(\sum_i (w_{k,i}^{(2)} h_i) + b_k^{(2)}\Big) = \overline{y_k} \cdot h_j$$

$$\overline{b_k^{(2)}} = \frac{\partial \mathscr{L}}{\partial y_k} \cdot \frac{\partial y_k}{\partial b_k^{(2)}} = \overline{y_k} \cdot \frac{\partial}{\partial b_k^{(2)}}\Big(\sum_i (w_{k,i}^{(2)} h_i) + b_k^{(2)}\Big) = \overline{y_k}$$

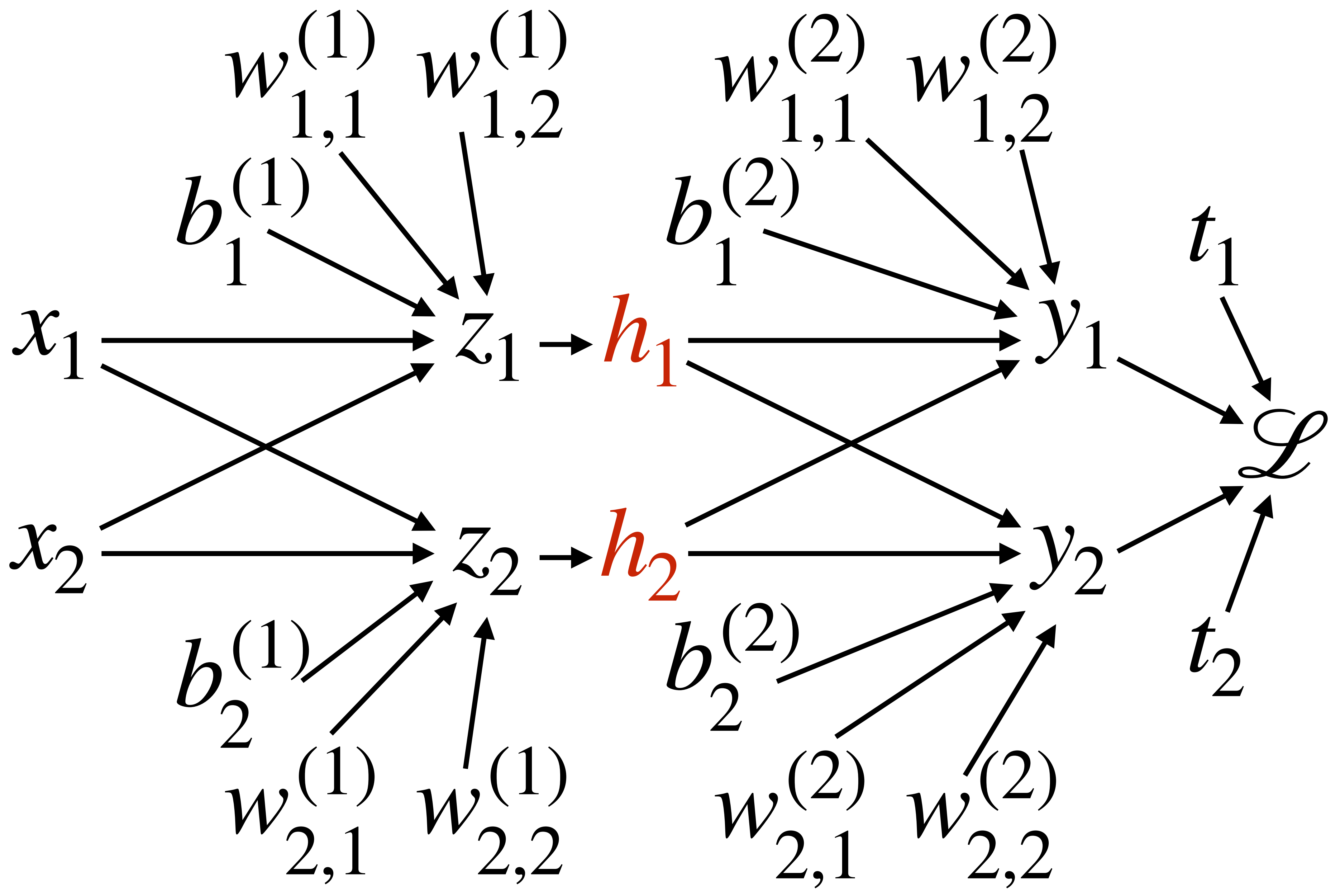
MULTILAYER NETWORKS - SCALAR VIEW (2)

Computing the loss derivatives for multilayer neural networks

- Essentially identical to previous slide; nothing new
- Just more cluttered as networks tend to be deep and consist of millions of parameters

Example of a two-layer NN: $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$; $\mathbf{h} = \sigma(\mathbf{z})$; $\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$; $\mathscr{L} = \frac{1}{2}\|\mathbf{y} - \mathbf{t}\|^2$

Backward path starts with $\overline{\mathscr{L}} = 1$; $\forall k, j$ of the corresponding vector/matrix



$$\overline{h_k} = \sum_j \frac{\partial \mathscr{L}}{\partial y_j} \cdot \frac{\partial y_j}{\partial h_k}$$
$$= \sum_j \overline{y_j} \cdot \frac{\partial}{\partial h_k}\Big(\sum_i (w^{(2)}_{j,i} h_i) + b^{(2)}_j\Big) = \sum_j \overline{y_j} \cdot w^{(2)}_{j,k}$$
$$= \sum_j \overline{y_j} \cdot w^{(2)}_{j,k}$$

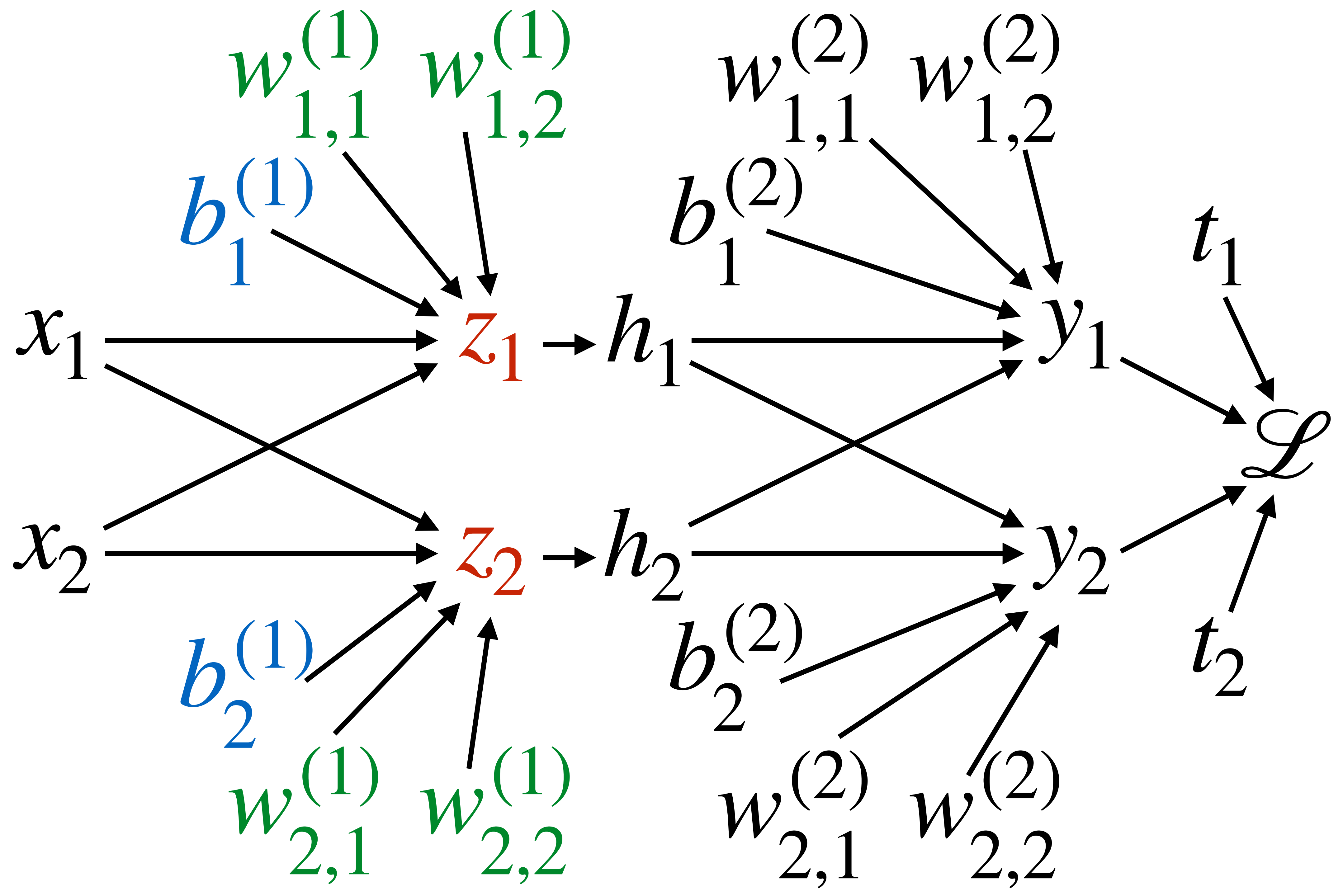
MULTILAYER NETWORKS - SCALAR VIEW (3)

Computing the loss derivatives for multilayer neural networks

- Essentially identical to previous slide; nothing new
- Just more cluttered as networks tend to be deep and consist of millions of parameters

Example of a two-layer NN: $\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$; $\mathbf{h} = \sigma(\mathbf{z})$; $\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$; $\mathscr{L} = \frac{1}{2}\|\mathbf{y} - \mathbf{t}\|^2$

Backward path starts with $\overline{\mathscr{L}} = 1$; $\forall k, j$ of the corresponding vector/matrix

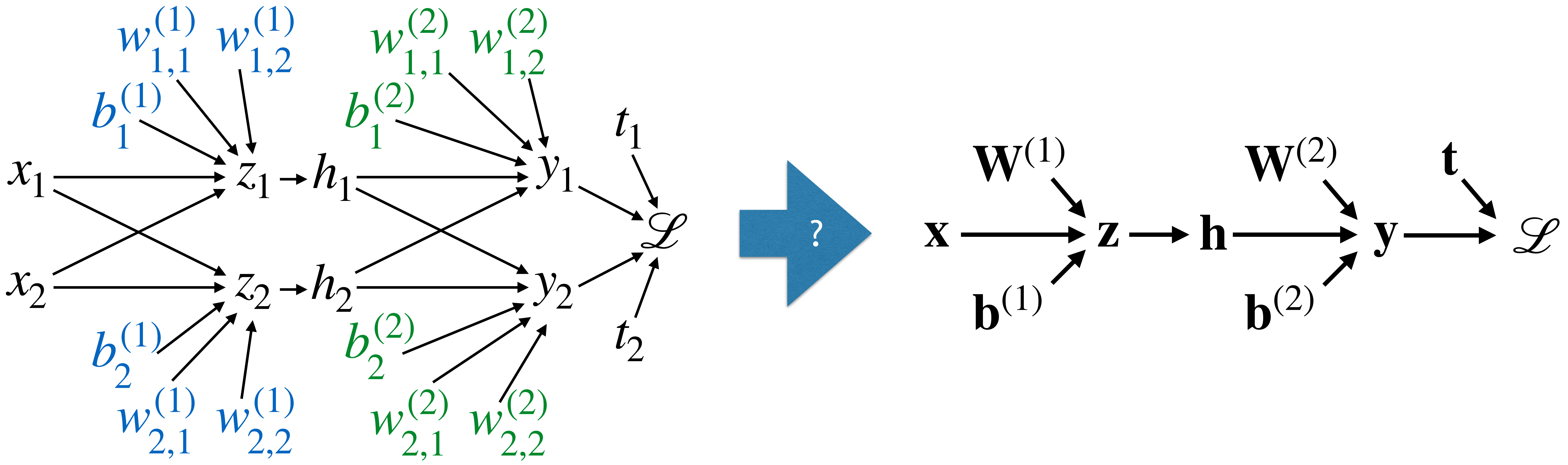


$$\overline{z_k} = \frac{\partial \mathscr{L}}{\partial h_k} \cdot \frac{\partial h_k}{\partial z_k} = \overline{h_k} \cdot \frac{\partial}{\partial z_k}\sigma(z_k) = \overline{h_k}\sigma'(z_k)$$

$$\overline{w_{k,j}^{(1)}} = \frac{\partial \mathscr{L}}{\partial z_k} \cdot \frac{\partial z_k}{\partial w_{k,j}^{(1)}} = \overline{z_k} \cdot \frac{\partial}{\partial w_{k,j}^{(1)}}\Big(\sum_i (w_{k,i}^{(1)} x_i) + b_k^{(1)}\Big) = \overline{z_k} \cdot x_j$$

$$\overline{b_k^{(1)}} = \frac{\partial \mathscr{L}}{\partial z_k} \cdot \frac{\partial z_k}{\partial b_k^{(1)}} = \overline{z_k} \cdot \frac{\partial}{\partial b_k^{(1)}}\Big(\sum_i (w_{k,i}^{(1)} x_i) + b_k^{(1)}\Big) = \overline{z_k}$$

VECTOR CALCULUS



VECTOR CALCULUS

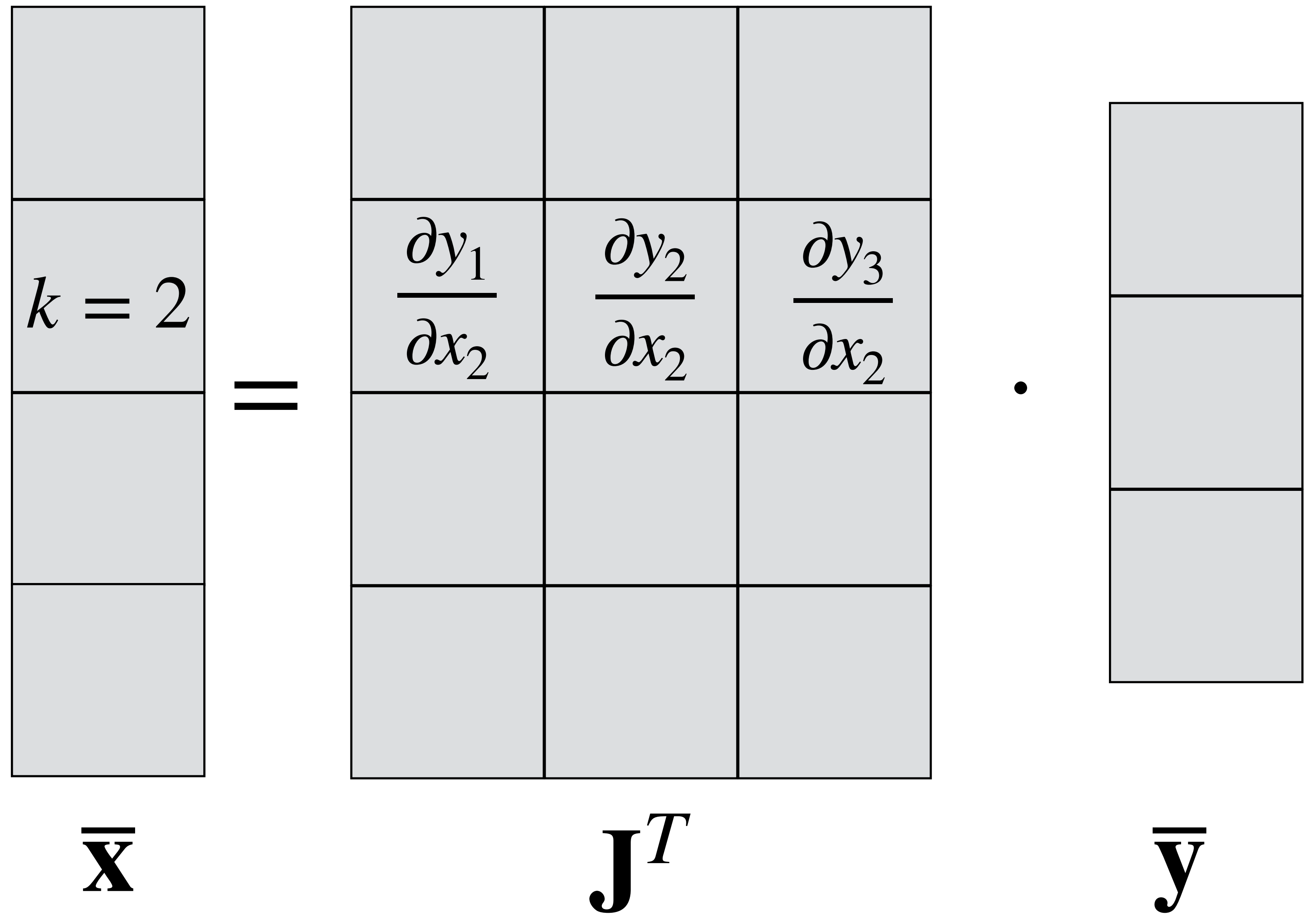
Consider a vector-valued function $(y_1 \dots y_m)^T = f(x_1 \dots x_n)^T$

Or $\mathbf{y} = f(\mathbf{x})$

Then the partial derivatives are $\overline{x_k} = \sum_i \overline{y_i} \frac{\partial y_i}{\partial x_k}, \ \forall k$

Based on a Jacobian matrix (in numerator layout)

$$\mathbf{J} = \left(\frac{\partial \mathbf{y}}{\partial x_1} \dots \frac{\partial \mathbf{y}}{\partial x_n} \right) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$



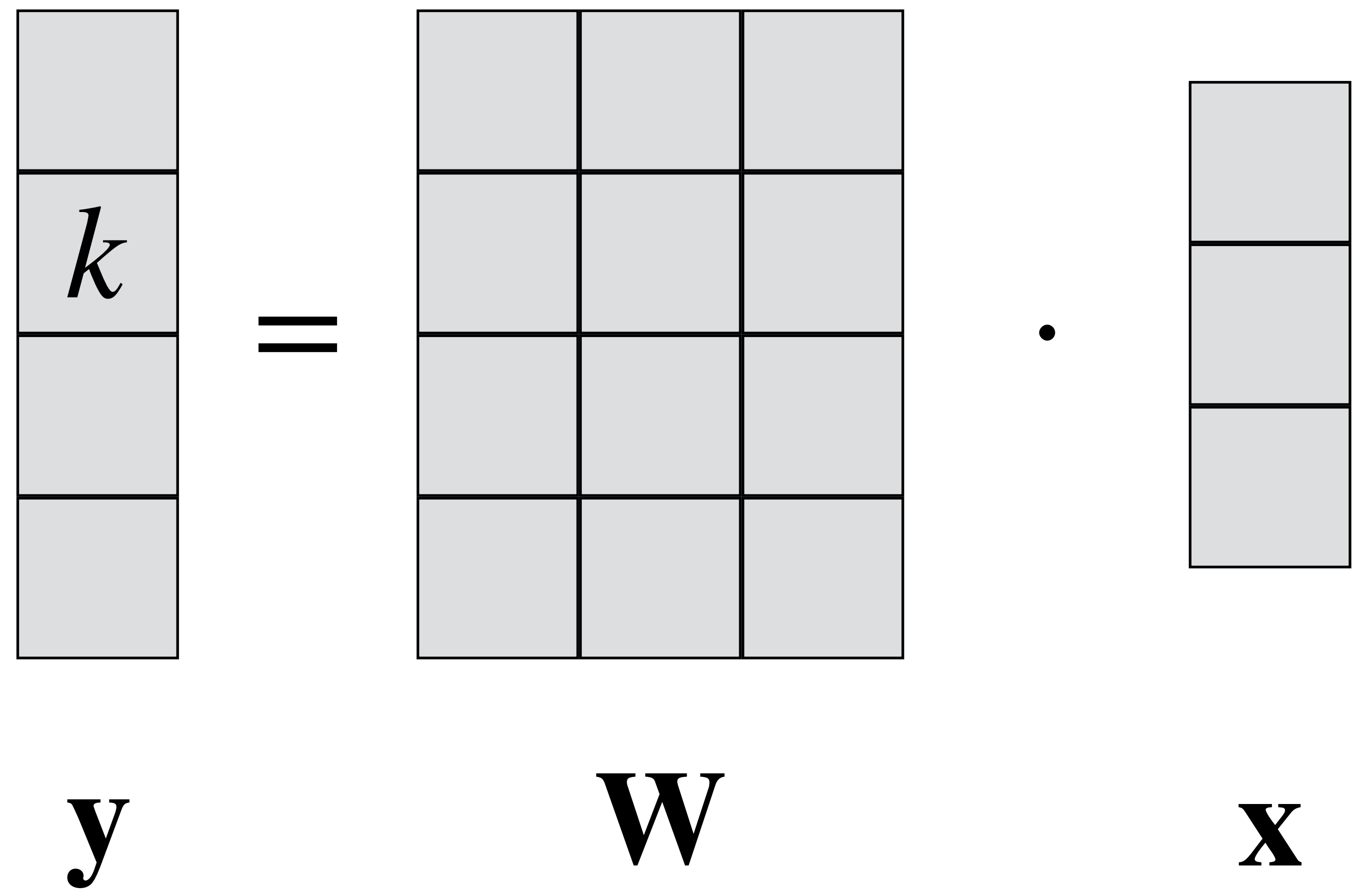
$k=2$
$=$
$\frac{\partial y_1}{\partial x_2}$
$\frac{\partial y_2}{\partial x_2}$
$\frac{\partial y_3}{\partial x_2}$
$\cdot$
$\overline{\mathbf{x}}$
$\mathbf{J}^T$
$\overline{\mathbf{y}}$

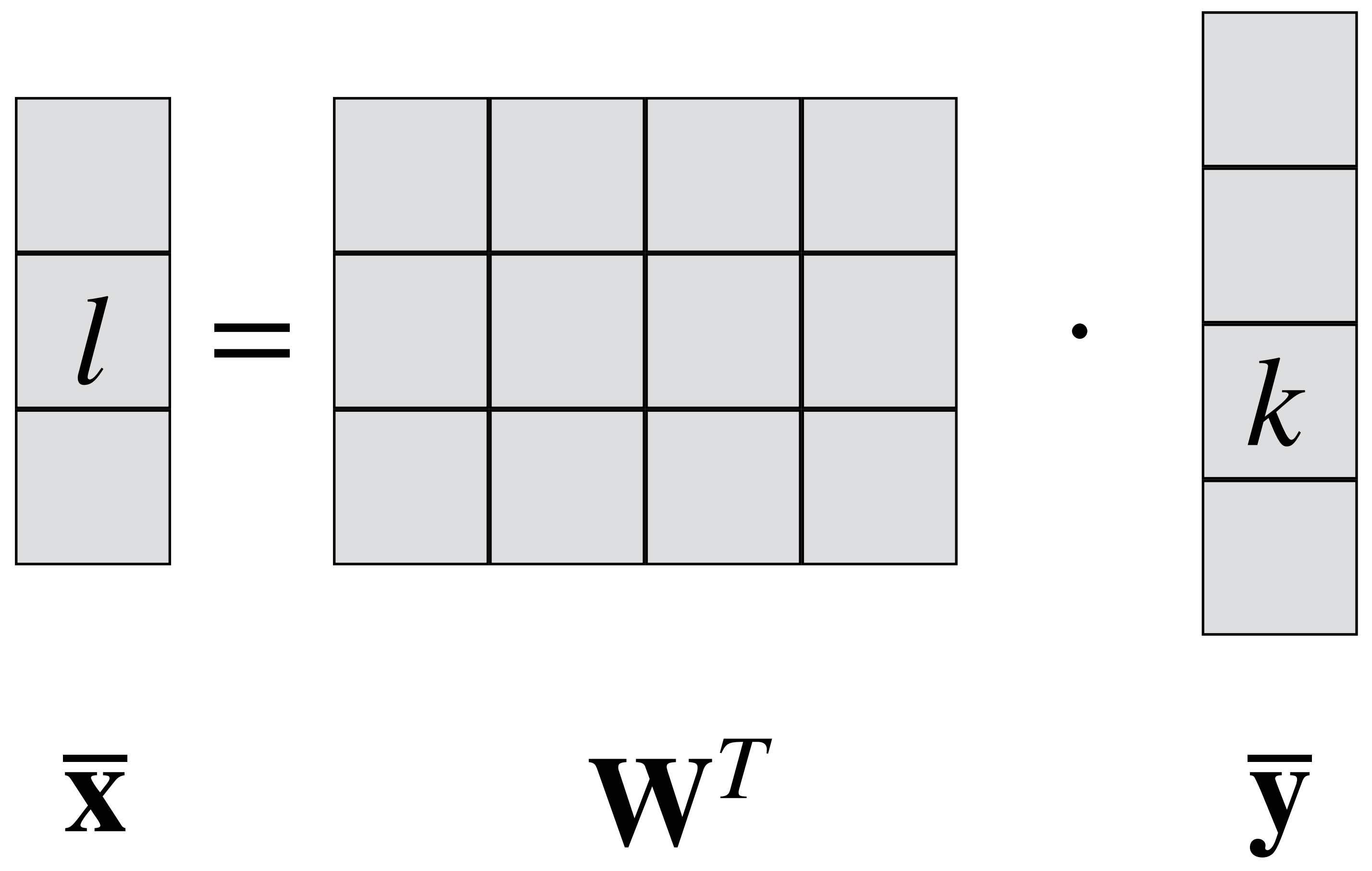
the partial derivatives can be formulated as $\overline{\mathbf{x}} = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^T \overline{\mathbf{y}} = \mathbf{J}^T \overline{\mathbf{y}}$

VECTOR CALCULUS - EXAMPLE 1

Matrix-vector multiplication $\mathbf{y} = \mathbf{W}\mathbf{x}$ resp. $y_k = \sum_i w_{k,i} x_i$

$\frac{\partial y_k}{\partial x_l} = \frac{\partial}{\partial x_l} \sum_i w_{k,i} x_i = w_{k,l}$ => $\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{W}$ => $\bar{\mathbf{x}} = \mathbf{W}^T \bar{\mathbf{y}}$ (chain rule)

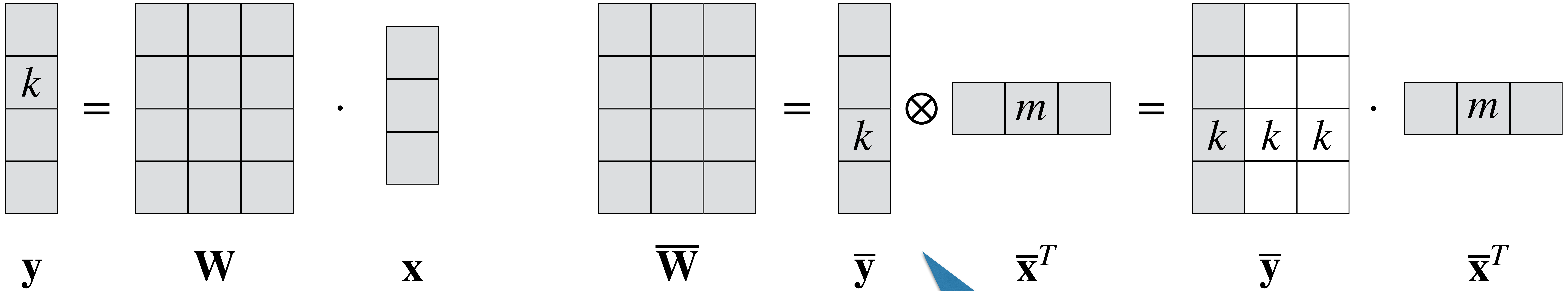




VECTOR CALCULUS - EXAMPLE 2

Matrix-vector multiplication $\mathbf{y} = \mathbf{W}\mathbf{x}$ resp. $y_k = \sum_i w_{k,i} x_i$

$$\frac{\partial y_k}{\partial w_{l,m}} = \frac{\partial}{\partial w_{l,m}} \sum_i w_{k,i} x_i = \begin{cases} x_m;\, k = l \\ 0;\ \text{else} \end{cases}$$ => $\overline{\mathbf{W}} = \overline{\mathbf{y}}\mathbf{x}^T = \overline{\mathbf{y}} \otimes \mathbf{x}$ (chain rule)



Dyadic/outer product

VECTOR CALCULUS - EXAMPLE 3

Elementwise operation $\mathbf{y} = e^{\mathbf{x}}$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} e^{x_0} & & 0 \\ & \ddots & \\ 0 & & e^{x_n} \end{bmatrix} \text{ and thus } \overline{\mathbf{x}} = e^{\mathbf{x}} \circ \overline{\mathbf{y}}$$

Only for $k = l$, k being an element of $\mathbf{y}$ and l an element of $\mathbf{x}$, the partial derivate is nonzero

Hadamard product $\circ$ (elementwise multiplication)

MULTILAYER NETWORKS - VECTORIZED VIEW

$$\overline{y_k} = \frac{\partial \mathcal{L}}{\partial y_k} = \frac{\partial}{\partial y_k}\Big(\frac{1}{2}\sum_i (y_i - t_i)^2\Big) = y_k - t_k$$

$$\overline{\mathbf{y}} = (\mathbf{y} - \mathbf{t})$$

$$\overline{w_{k,j}^{(2)}} = \frac{\partial \mathcal{L}}{\partial y_k} \cdot \frac{\partial y_k}{\partial w_{k,j}^{(2)}} = \overline{y_k} \cdot \frac{\partial}{\partial w_{k,j}^{(2)}}\Big(\sum_i (w_{k,i}^{(2)} h_i) + b_k^{(2)}\Big) = \overline{y_k} \cdot h_j$$

$$\overline{\mathbf{W}^{(2)}} = \overline{\mathbf{y}}\mathbf{h}^T$$

$$\overline{b_k^{(2)}} = \frac{\partial \mathcal{L}}{\partial y_k} \cdot \frac{\partial y_k}{\partial b_k^{(2)}} = \overline{y_k} \cdot \frac{\partial}{\partial b_k^{(2)}}\Big(\sum_i (w_{k,i}^{(2)} h_i) + b_k^{(2)}\Big) = \overline{y_k}$$

$$\overline{\mathbf{b}^{(2)}} = \overline{\mathbf{y}}$$

$$\overline{h_k} = \sum_j \frac{\partial \mathcal{L}}{\partial y_j} \cdot \frac{\partial y_j}{\partial h_k} = \sum_j \overline{y_j} \cdot \frac{\partial}{\partial h_k}\Big(\sum_i (w_{j,i}^{(2)} h_i) + b_j^{(2)}\Big) = \sum_j \overline{y_j} \cdot w_{j,k}^{(2)}$$

$$\overline{\mathbf{h}} = \mathbf{W}^{(2)T}\overline{\mathbf{y}}$$

$$\overline{z_k} = \frac{\partial \mathcal{L}}{\partial h_k} \cdot \frac{\partial h_k}{\partial z_k} = \overline{h_k} \cdot \frac{\partial}{\partial z_k}\sigma(z_k) = \overline{h_k}\sigma'(z_k)$$

$$\overline{\mathbf{z}} = \overline{\mathbf{h}} \circ \sigma'(\mathbf{z})$$

$$\overline{w_{k,j}^{(1)}} = \frac{\partial \mathcal{L}}{\partial z_k} \cdot \frac{\partial z_k}{\partial w_{k,j}^{(1)}} = \overline{z_k} \cdot \frac{\partial}{\partial w_{k,j}^{(1)}}\Big(\sum_i (w_{k,i}^{(1)} x_i) + b_k^{(1)}\Big) = \overline{z_k} \cdot x_j$$

$$\overline{\mathbf{W}^{(1)}} = \overline{\mathbf{z}}\mathbf{x}^T$$

$$\overline{b_k^{(1)}} = \frac{\partial \mathcal{L}}{\partial z_k} \cdot \frac{\partial z_k}{\partial b_k^{(1)}} = \overline{z_k} \cdot \frac{\partial}{\partial b_k^{(1)}}\Big(\sum_i (w_{k,i}^{(1)} x_i) + b_k^{(1)}\Big) = \overline{z_k}$$

$$\overline{\mathbf{b}^{(1)}} = \overline{\mathbf{z}}$$

VECTOR CALCULUS FOR GRADIENTS

Suppose now that $\mathbf{v}$ is the gradient vector of the scalar loss l with respect to a (neural network) function's vector-valued result

$$\mathbf{y} = f(\mathbf{x}) \text{: } \mathbf{v} = \left(\frac{\partial l}{\partial y_1} \cdots \frac{\partial l}{\partial y_m}\right)^T$$

To obtain the loss gradient with respect to the weights $\mathbf{x}$, simply:

$$\frac{\partial l}{\partial \mathbf{x}} = \mathbf{v} \cdot \mathbf{J}^{\mathrm{T}} = \begin{pmatrix} \frac{\partial l}{\partial y_1} \\ \vdots \\ \frac{\partial l}{\partial y_m} \end{pmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} = \begin{pmatrix} \frac{\partial l}{\partial x_1} \\ \vdots \\ \frac{\partial l}{\partial x_n} \end{pmatrix}$$

$\mathbf{v}$ = `external_grad`

"Vector Jacobian Product" (VJP)

AUTOGRAD ALGORITHM

```
Let $v_1$ ... $v_N$ denote all vertices (topological ordering)

Let Pa($v_i$) be the parents of $v_i$

Let Ch($v_i$) be the children of $v_i$

For i = 1,...,N
    Compute $v_i$ as a function of Pa($v_i$)

Set $\bar{v}_N = 1$ (by convention $d\mathscr{L}_{reg}/d\mathscr{L}_{reg} = 1$)

For i = N-1,...,1
    $\bar{v}_i = \sum_{j \in \mathrm{Ch}(v_i)} \bar{v}_j \frac{dv_j}{dv_i}$
```

Topological ordering: parents (incl. inputs) come before children (incl. output)

$$\begin{tikzcd}
x \arrow[dr] & & t \arrow[dr] & & \\
b \arrow[r] & z \arrow[r] & y \arrow[r] & \mathscr{L} \arrow[r] & \mathscr{L}_{reg} \\
w \arrow[ur] \arrow[rrr] & & & \mathscr{R} \arrow[ur] &
\end{tikzcd}$$

AUTOGRAD OVERVIEW

1. Trace the computations to construct the computational graph

   Each node of the graph contains information about value (result), function (primitive op), arguments and parents of this node

2. Implement the VJP for each node (primitive operation)

   Note that often the VJP is computed without constructing a Jacobian matrix to save time and space, as element-wise operations result in a diagonal Jacobian

3. Backprop

   Given the previous work, pretty much straight-forward

AUTOGRAD DETAILS

Different granularity: operates on primitive operations, not complex expressions

- Many similarities with a processor's microcode and register allocation

No redundant computations

- Modular: broken into small pieces that can be used elsewhere

Works well with vector notation

Original program

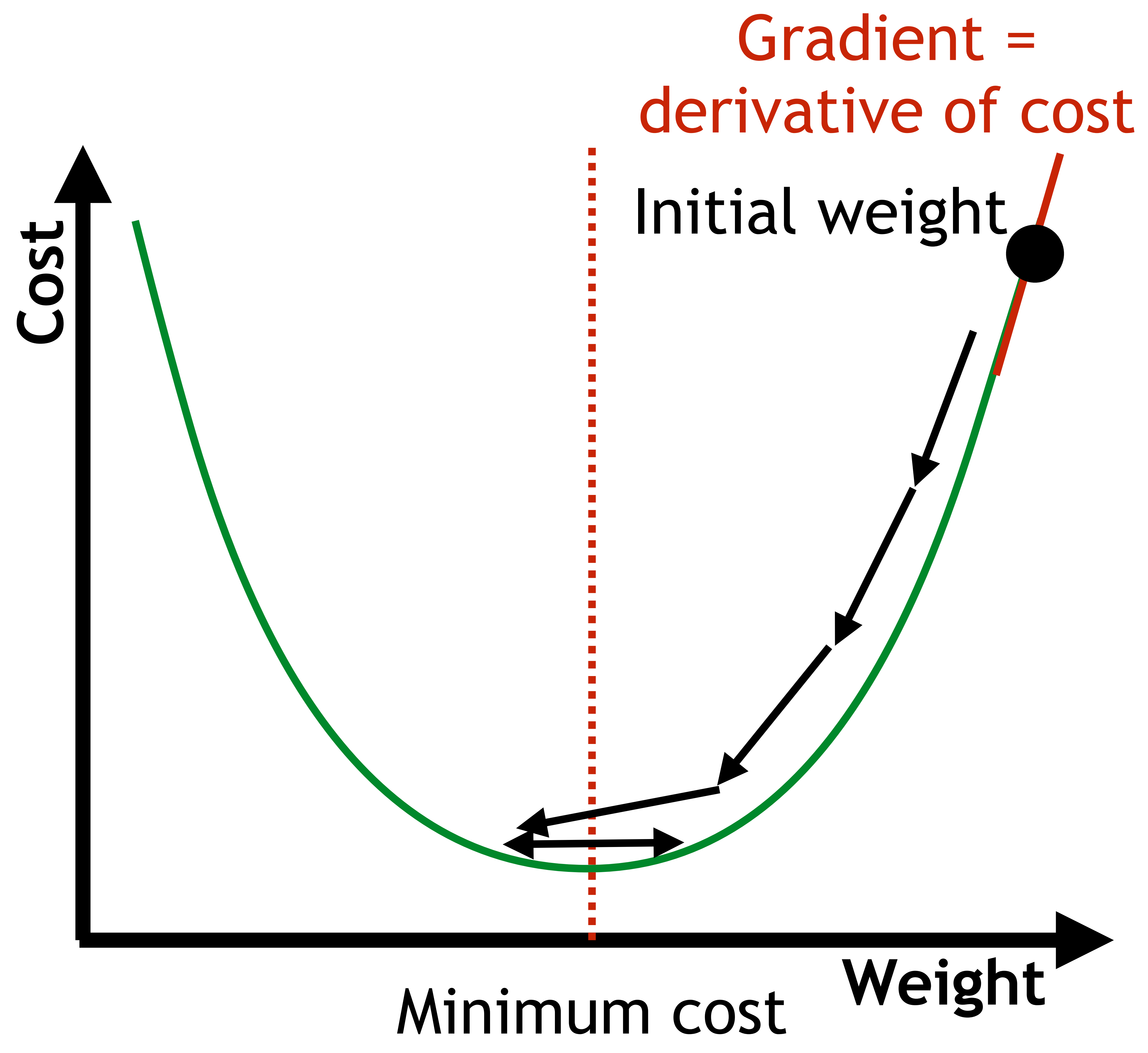
$$z = wx + b$$
$$y = \frac{1}{1 + e^{-z}}$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

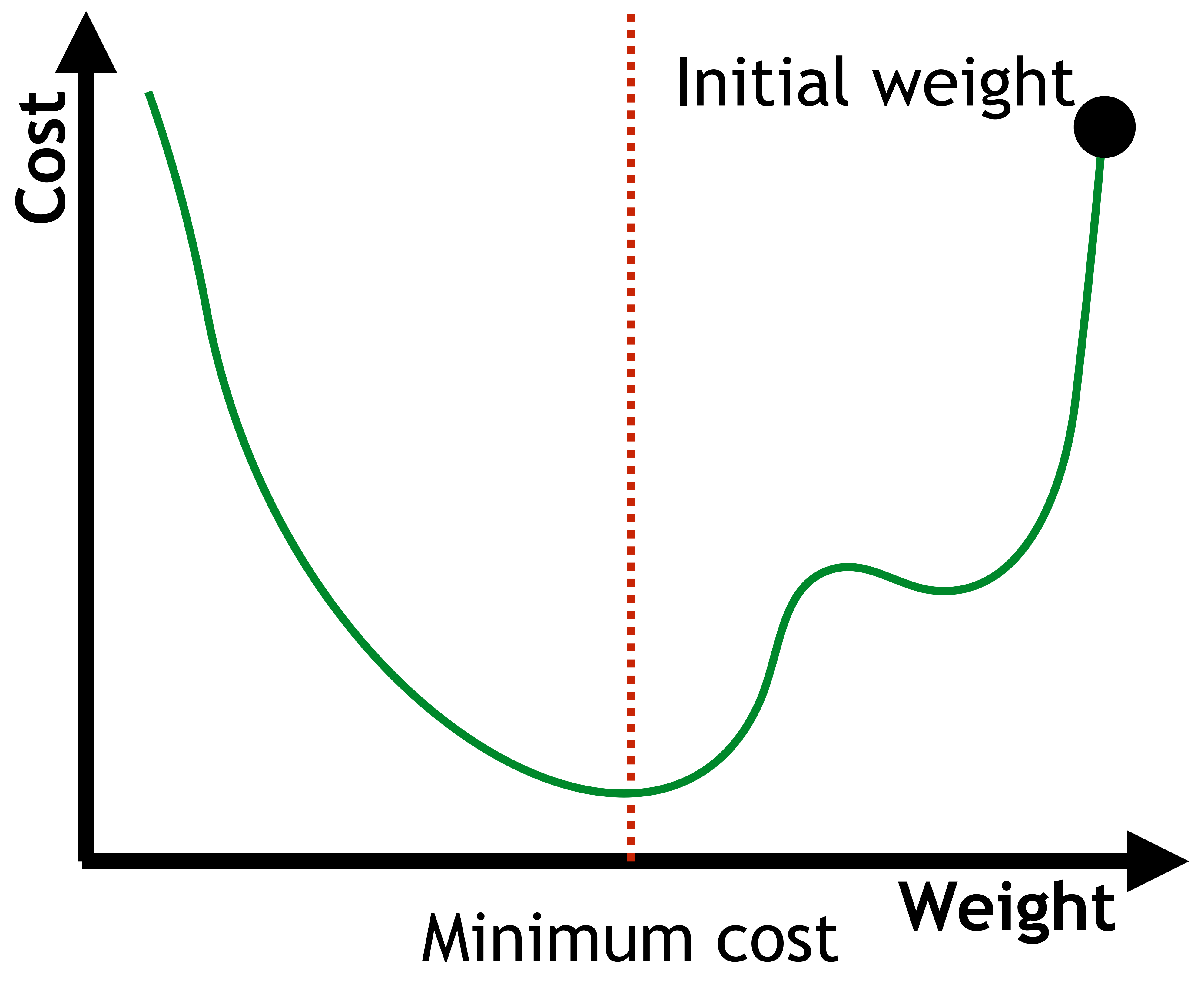
Sequence of operations

```
t1 = w * x
z  = t1 + b
t3 = -z
t4 = exp(t3)
t5 = 1 + t4
y  = 1 / t5
t6 = y - t
t7 = t6 * t6
L  = t7 / 2
```

OPTIMIZATION

GRADIENT DESCENT





$$\mathbf{W} := \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathscr{L}(\mathbf{W}; \mathscr{D})$$

GRADIENT DESCENT VARIANTS

Batch Gradient Descent can be slow and memory/compute intensive

Few but expensive updates

- Looks at every training sample on every step

$$\theta := \theta - \eta \nabla_\theta J(\theta)$$

- Guaranteed to be optimal, but expensive (memory) when dataset is large
- When used, rather with large learning rate (make big steps that are guaranteed to be optimal)

Stochastic Gradient Descent (SGD) updates weights for each training sample

Many but cheap updates

- Stochasticity can be of help to overcome local minima and saddle points
- Requires shuffling the training set (stochasticity)
- Usually small learning rates

$$\theta := \theta - \eta \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

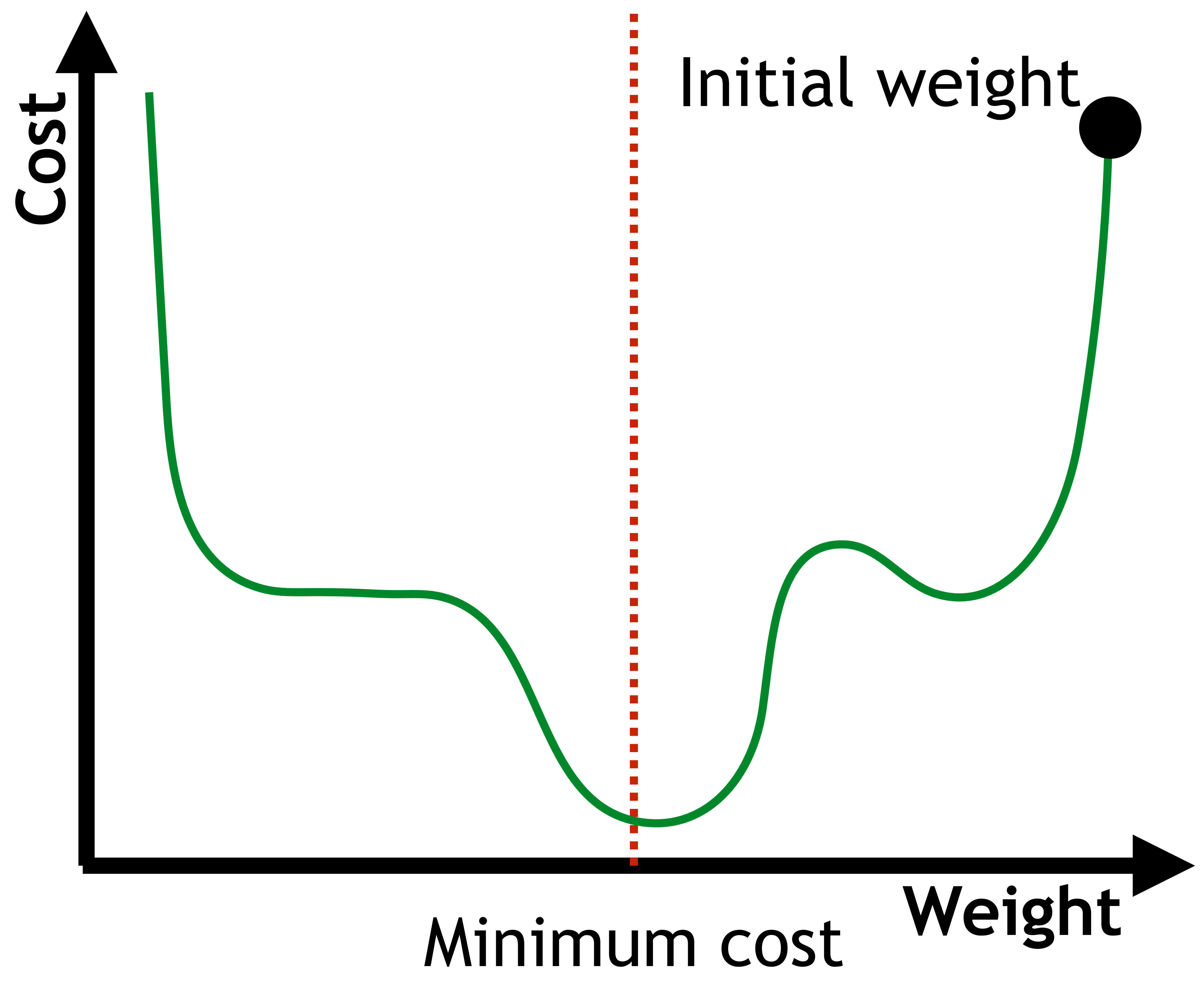
Mini-batch Stochastic Gradient Descent considers a subset of the training set for each update (so-called mini-batch)

- Reduces the variance of weight updates, can thus be more stable in converging
- Mini-batch size N often a power of two (GPU efficiency)
- Plain SGD: equals mini-batch size of 1

$$\theta := \theta - \eta \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

CHOOSING A GOOD LEARNING RATE IS DIFFICULT

- Learning rate schedules adjust the learning rate during training
  - But often add more hyperparameters (schedules and thresholds)
- The same learning rate is applied to all parameter updates, but different parameters might be of different importance
  - In particular if data is sparse or if features have different frequencies
- Neural network loss landscapes are ugly
  - A plethora of local minima and saddle points
  - It is an open discussion which of these two is actually more painful



MOMENTUM

Computes gradient (v_t) for every (mini)-batch and based on earlier gradients (v_{t-1})

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta := \theta - v_t$$

Faster convergence by dampening of "fluctuations"

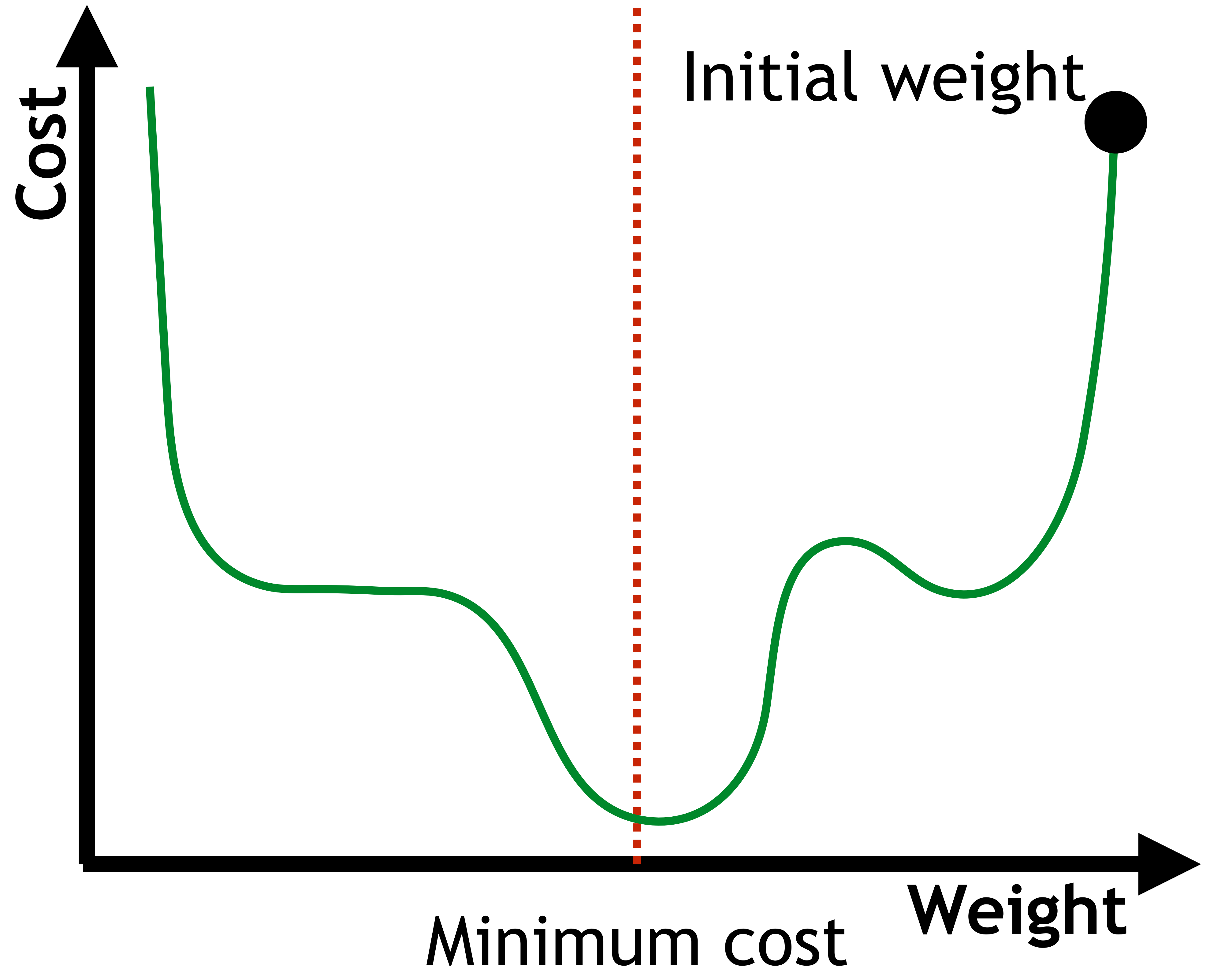
- "Rolling down a ball"

Equal to subtracting the exponentially decaying average of gradients

$$\theta_{t+1} = \eta\left(\nabla_{\theta_t} J(\theta_t) + \gamma \nabla_{\theta_{t-1}} J(\theta_{t-1}) + \gamma^2 \nabla_{\theta_{t-2}} J(\theta_{t-2}) + \ldots \right)$$

Issues

- Convergence is slowed down by static learning rate
- Another hyperparameter



ADAGRAD

Adapts the learning rate to the parameters

- A different learning rate η for every parameter θ_i at every time step t
- Small learning rate for frequent features, large learning rate for infrequent features

Adagrad considers all past gradients for a given parameter θ_i

- Classical update rule is $\theta_{t+1,i} = \theta_{t,i} - \eta g_{t,i}$, for $g_{t,i} = \nabla_\theta J(\theta_{t,i})$,
- Form the sum of the squares of the gradients for θ_i up to time step t into a diagonal matrix (accumulator)

$$\mathbf{G}_t = \sum_{\tau=1}^{t} g_\tau g_\tau^T$$

Initial learning rate

Then: $\theta_{t+1,i} = \theta_{t,i} - \dfrac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$

dampening factor (e.g., 1e-8) to avoid division by zero

No need to fine-tune learning rate, usually use $\eta = 0.01$

Since squared gradients are always positive, $\mathbf{G}$ becomes larger over time and effective learning rate shrinks => stalled convergence

ADADELTA (AND RMSPROP)

ADADELTA (and RMSPROP) restrict the considered history of gradients to overcome ADAGRAD's problem of diminishing learning rate

- Only consider gradients $g = g_{[t:t-w]}$ of a fixed window of size w
- Instead of inefficiently storing w previous gradients, a decaying average is used (similar to momentum) based on a trade-off parameter γ: $E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$

Consider $\theta_{t+1} = \theta_t - \eta g_{t,i} = \theta_t + \Delta\theta_t$, and $RMS[x] = \sqrt{x^2 + \epsilon}$:

- Then: $\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t = -\frac{\eta}{RMS[g]_t} g_t$
- I.e. the diagonal matrix $\mathbf{G}_t$ (all past gradients) is replaced by selected squared gradients

Due to a unit mismatch, also consider an exponentially decaying average for squared parameter updates: $E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$

- Then: $\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$

No more learning rate :)

ADAPTIVE MOMENT ESTIMATION (ADAM)

- ADAM combines Adadelta resp. RMSPROP with momentum
  - Thus extends exponentially decaying average of past squared gradients v_t (e.g., Adadelta & RMSPROP) with an exponentially decaying average of past gradients m_t (similar to momentum)
  - $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$ and $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
- Due to zero initialization there is a a bias towards zero
  - Correction: $\hat{m}_t = m_t / (1 - \beta_1^t)$ and $\hat{v}_t = v_t / (1 - \beta_2^t)$, usually with $\beta_1^t = 0.9$ and $\beta_2^t = 0.999$
- Then: $\theta_{t+1} = \theta_t - \dfrac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$
- “Behaves like a heavy ball with friction”
  - Accelerate in flatter regions, allowing it to progress efficiently toward a minimum
  - Slow down in steeper regions, preventing overshooting and making the convergence more stable
  - Avoid oscillations near the minima or along sharp gradients due to the “damping” effect of friction

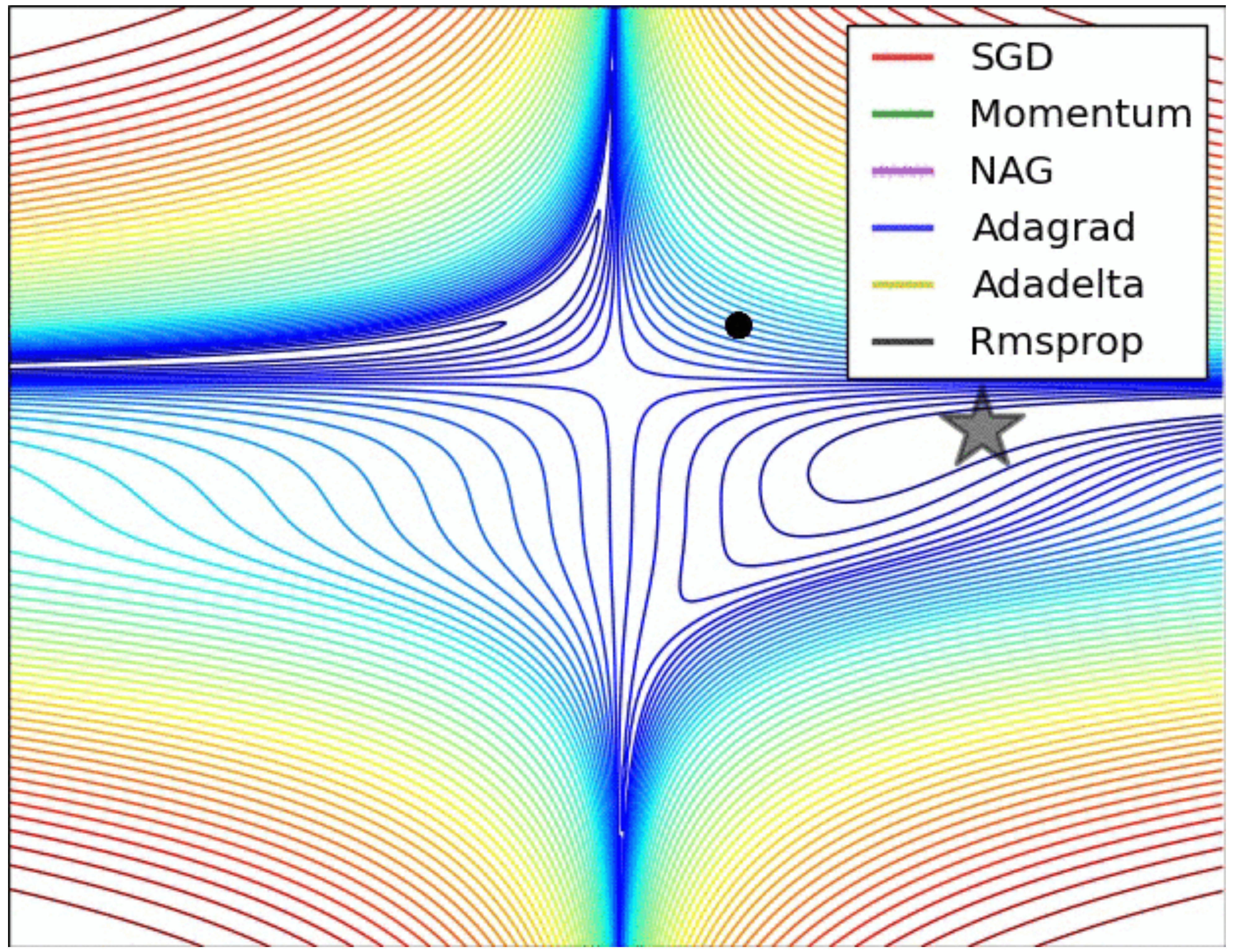
EXAMPLES

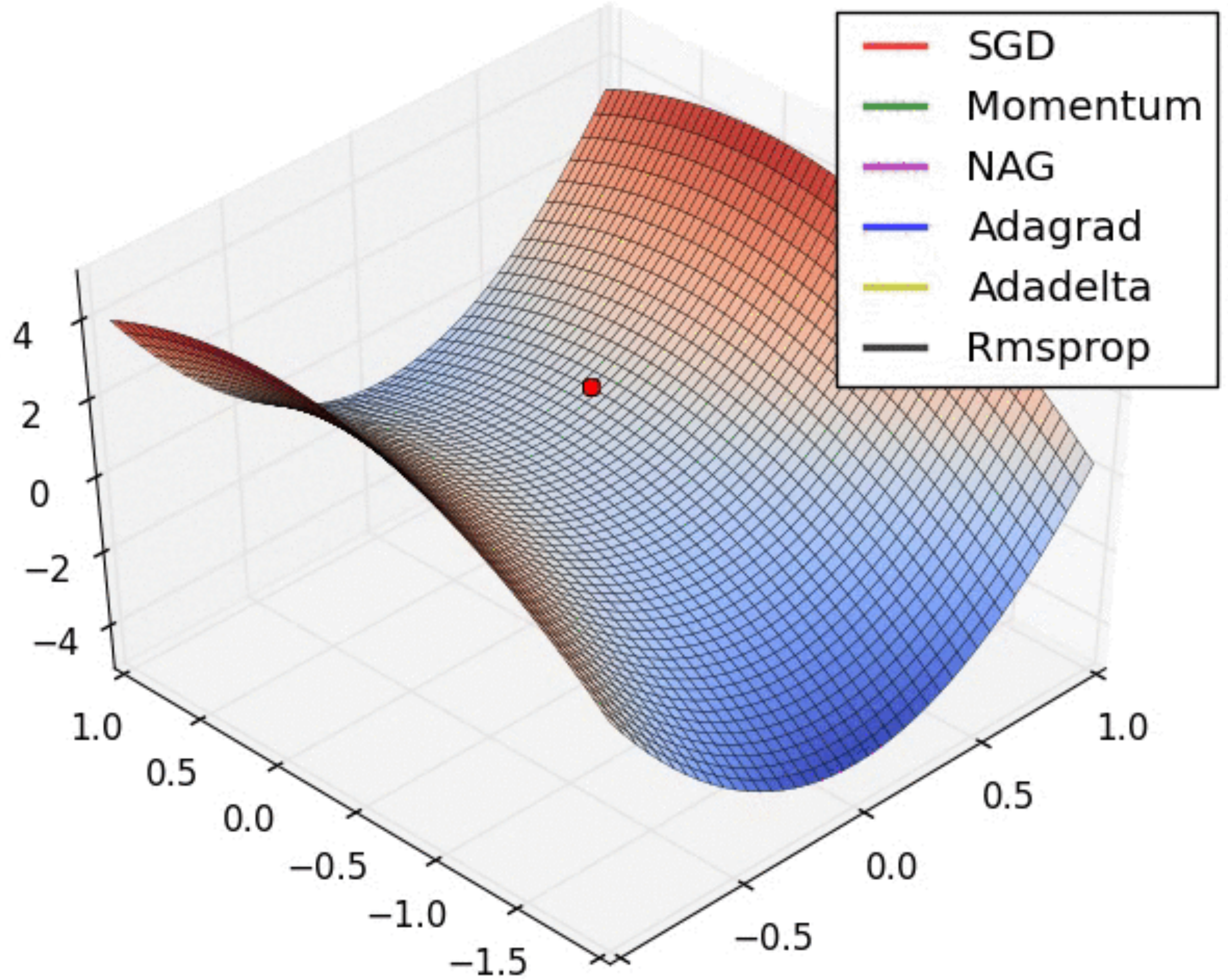
Plain SGD is very robust but does not converge fast

Momentum: consider history of gradients

Adagrad/Adadelta: per-parameter learning rate (infinite/restricted history)

ADAM: combine Momentum and Adadelta (most recommended)





Source: Sebastian Ruder, "An overview of gradient descent optimization algorithms", https://www.ruder.io/optimizing-gradient-descent/

WRAPPING UP

SUMMARY

Backprop and SGD are powerful optimization methods

- Super easy to use and robust
- Actually often hide major implementation mistakes
- Main requirement: differentiability -> restricts choice of NN components

Auto differentiation in PyTorch

- Actually shares many similarities with various other ML tools (TensorFlow, (Theano), TVM, TensorComprehensions, MLIR, …)
- Different layers (intermediate representations), computational graph and corresponding transformations, …

Optimization

- Methods beyond plain SGD to improve convergence and hyperparameter search
- ADAM is strongly recommended