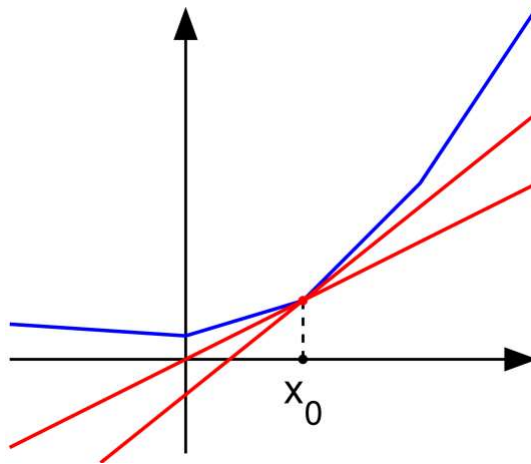
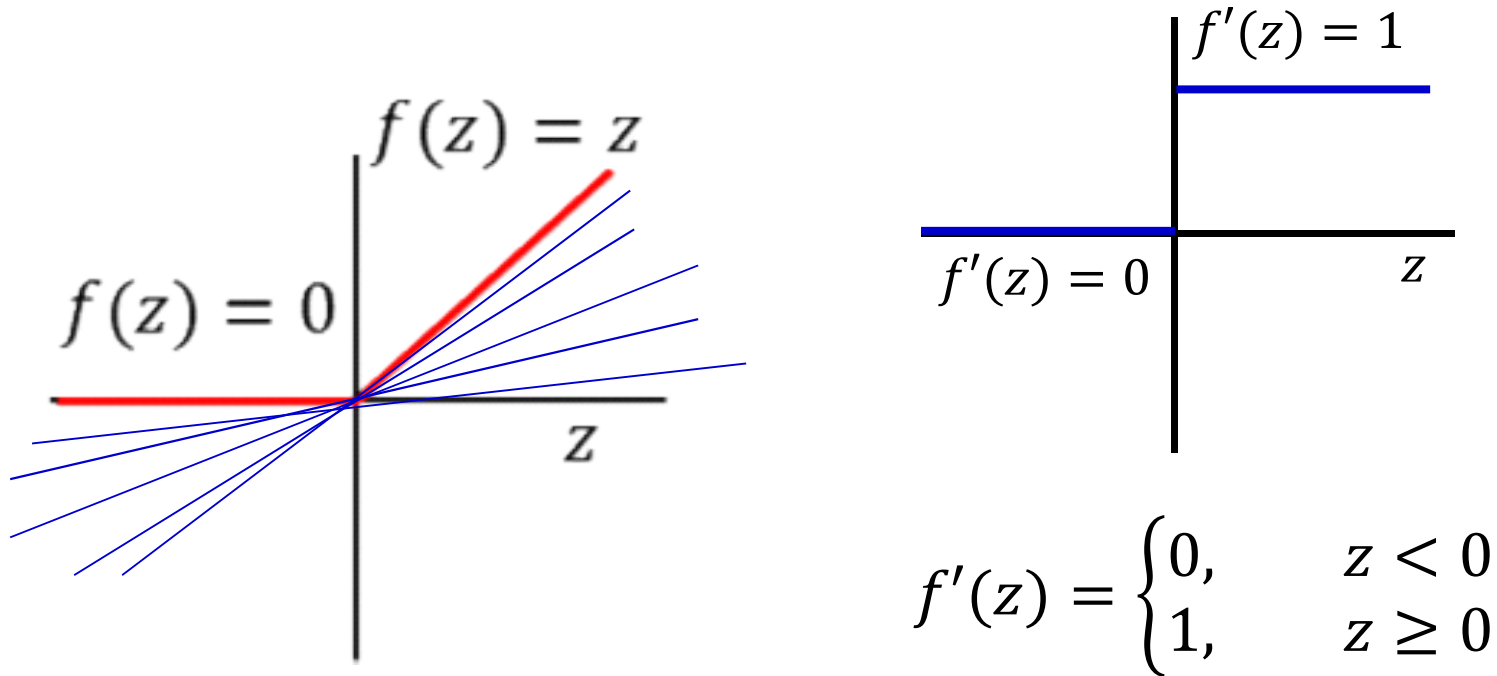


The subgradient



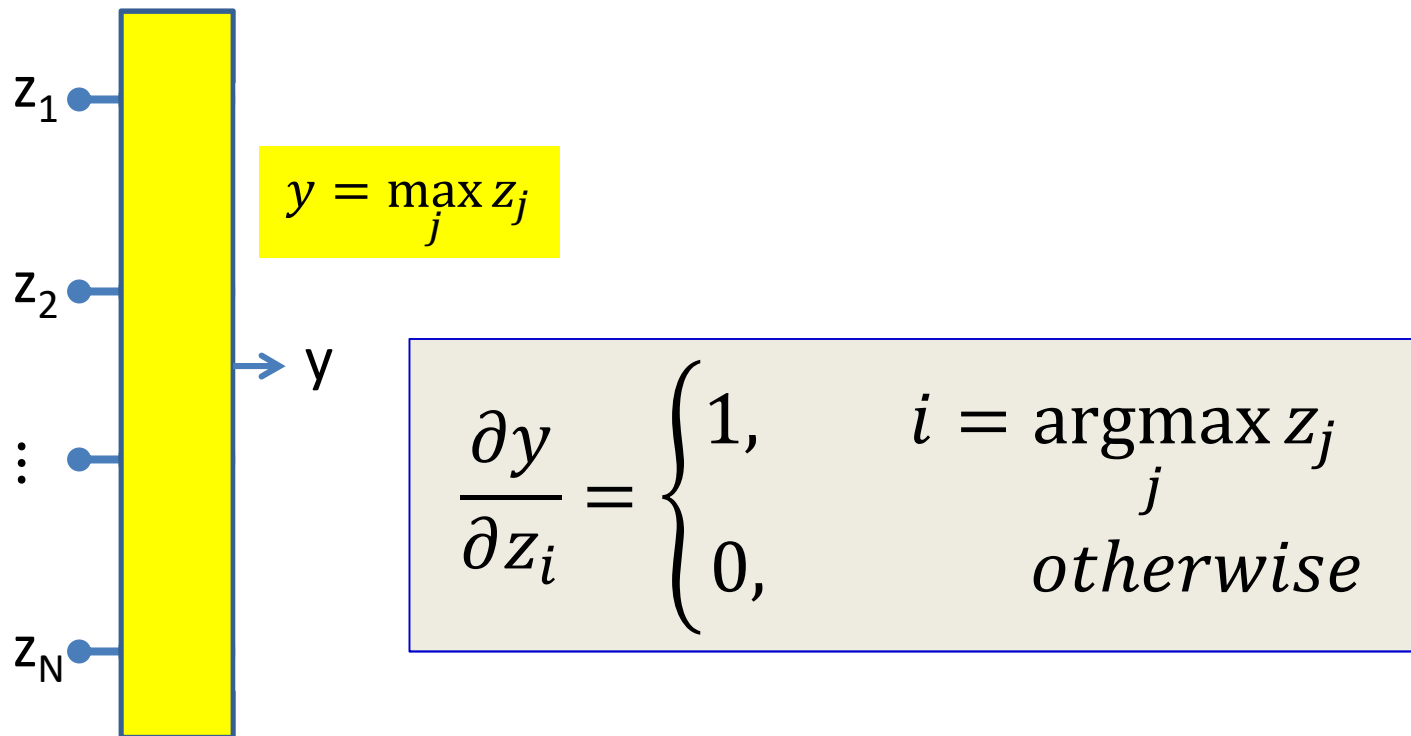
- A subgradient of a function $f(x)$ at a point x_0 is any vector v such that
$$(f(x) - f(x_0)) \geq v^T (x - x_0)$$
 - Any direction such that moving in that direction increases the function
- Guaranteed to exist only for convex functions
 - “bowl” shaped functions
 - For non-convex functions, the equivalent concept is a “quasi-secant”
- The subgradient is a direction in which the function is guaranteed to increase
- If the function is differentiable at x_0 , the subgradient is the gradient
 - The gradient is not always the subgradient though

Subgradients and the RELU



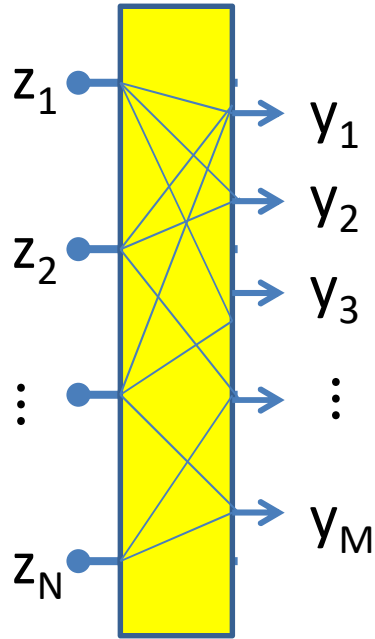
- Can use any subgradient
 - At the differentiable points on the curve, this is the same as the gradient
 - Typically, will use the equation given

Subgradients and the Max



- Vector equivalent of subgradient
 - 1 w.r.t. the largest incoming input
 - Incremental changes in this input will change the output
 - 0 for the rest
 - Incremental changes to these inputs will not change the output

Subgradients and the Max



$$y_i = \operatorname{argmax}_{l \in \mathcal{S}_j} z_l$$

$$\frac{\partial y_j}{\partial z_i} = \begin{cases} 1, & i = \operatorname{argmax}_{l \in \mathcal{S}_j} z_l \\ 0, & \text{otherwise} \end{cases}$$

- Multiple outputs, each selecting the max of a different subset of inputs
 - Will be seen in convolutional networks
- Gradient for any output:
 - 1 for the specific component that is maximum in corresponding input subset
 - 0 otherwise

Backward Pass: Recap

- Output layer (N) :

- For $i = 1 \dots D_N$

- $\frac{\partial Div}{\partial Y_i} = \frac{\partial Div(Y, d)}{\partial y_i^{(N)}}$

- $\frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial Div}{\partial y_i^{(N)}} \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}}$
 - OR $\sum_j \frac{\partial Div}{\partial y_j^{(N)}} \frac{\partial y_j^{(N)}}{\partial z_i^{(N)}}$ (vector activation)

- For layer $k = N - 1$ downto 0

- For $i = 1 \dots D_k$

- $\frac{\partial Div}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial Div}{\partial z_j^{(k+1)}}$

- $\frac{\partial Div}{\partial z_i^{(k)}} = \frac{\partial Div}{\partial y_i^{(k)}} \frac{\partial y_i^{(k)}}{\partial z_i^{(k)}}$
 - OR $\sum_j \frac{\partial Div}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}}$ (vector activation)

- $\frac{\partial Div}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \frac{\partial Div}{\partial z_i^{(k+1)}}$ for $j = 1 \dots D_{k+1}$

These may be subgradients

Overall Approach

- For each data instance
 - **Forward pass:** Pass instance forward through the net. Store all intermediate outputs of all computation
 - **Backward pass:** Sweep backward through the net, iteratively compute all derivatives w.r.t weights
- Actual loss is the sum of the divergence over all training instances

$$\mathbf{Loss} = \frac{1}{|\{X\}|} \sum_X \text{Div}(Y(X), d(X))$$

- Actual gradient is the sum or average of the derivatives computed for each training instance

$$\nabla_W \mathbf{Loss} = \frac{1}{|\{X\}|} \sum_X \nabla_W \text{Div}(Y(X), d(X)) \quad W \leftarrow W - \eta \nabla_W \mathbf{Loss}^T$$

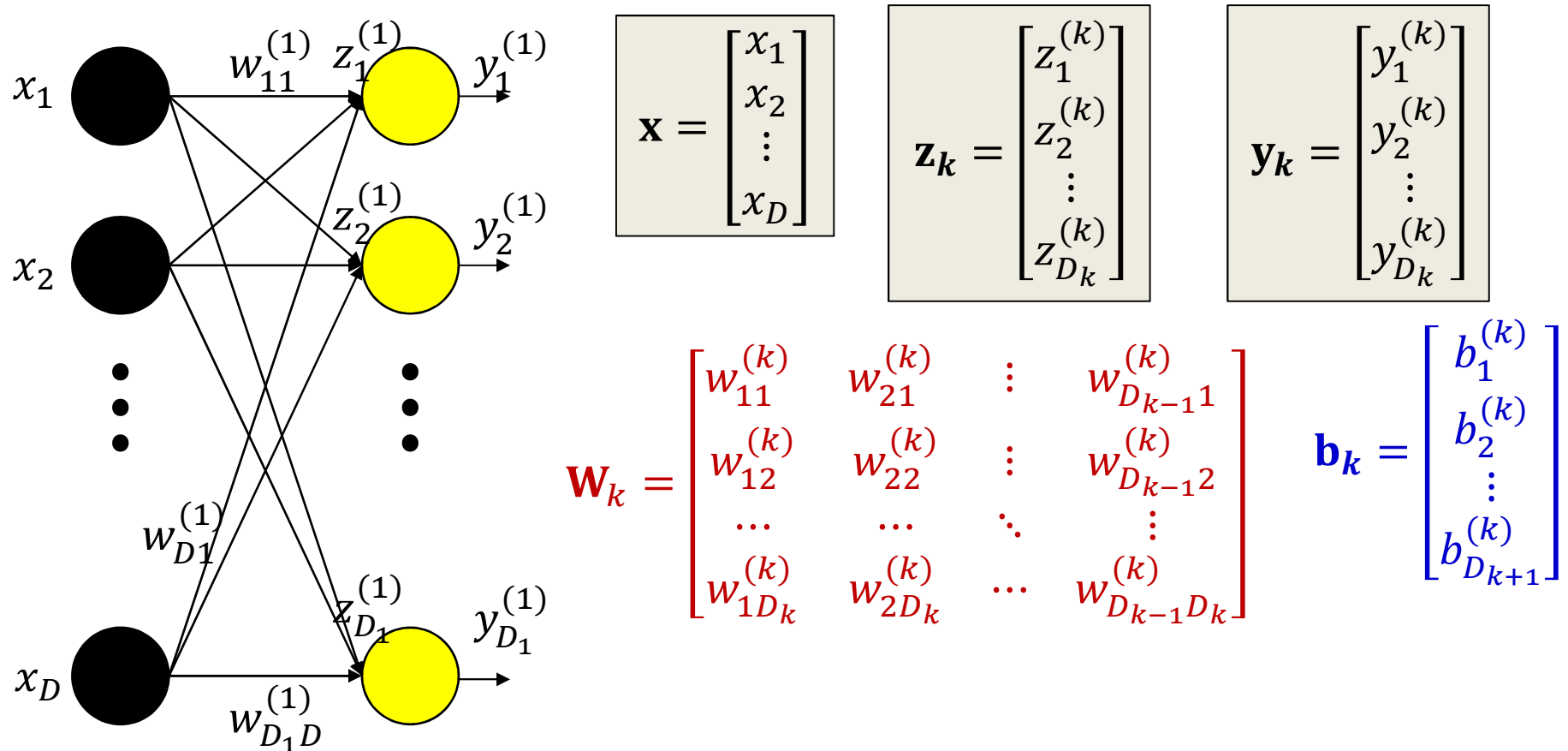
Training by BackProp

- Initialize weights $\mathbf{W}^{(k)}$ for all layers $k = 1 \dots K$
- Do:
 - Initialize $Loss = 0$; For all i, j, k , initialize $\frac{dLoss}{dw_{i,j}^{(k)}} = 0$
 - For all $t = 1:T$ (Loop over training instances)
 - **Forward pass:** Compute
 - Output \mathbf{Y}_t
 - $Loss += Div(\mathbf{Y}_t, \mathbf{d}_t)$
 - **Backward pass:** For all i, j, k :
 - Compute $\frac{dDiv(\mathbf{Y}_t, \mathbf{d}_t)}{dw_{i,j}^{(k)}}$
 - Compute $\frac{dLoss}{dw_{i,j}^{(k)}} += \frac{dDiv(\mathbf{Y}_t, \mathbf{d}_t)}{dw_{i,j}^{(k)}}$
 - For all i, j, k , update:
$$w_{i,j}^{(k)} = w_{i,j}^{(k)} - \frac{\eta}{T} \frac{dLoss}{dw_{i,j}^{(k)}}$$
- Until $Loss$ has converged

Vector formulation

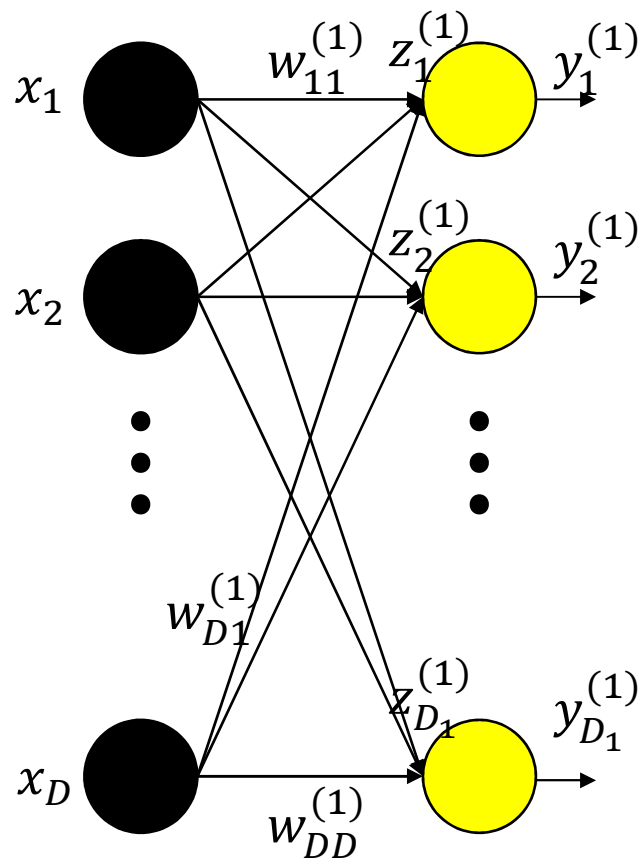
- For layered networks it is generally simpler to think of the process in terms of vector operations
 - Simpler arithmetic
 - Fast matrix libraries make operations *much* faster
- We can restate the entire process in vector terms
 - On slides, please read
 - This is what is *actually* used in any real system
 - Will appear in quiz

Vector formulation



- Arrange all inputs to the network in a vector \mathbf{x}
- Arrange the *inputs* to neurons of the k th layer as a vector \mathbf{z}_k
- Arrange the outputs of neurons in the k th layer as a vector \mathbf{y}_k
- Arrange the weights to any layer as a matrix \mathbf{W}_k
 - Similarly with biases

Vector formulation



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix}$$

$$\mathbf{z}_k = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{D_k}^{(k)} \end{bmatrix}$$

$$\mathbf{y}_k = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_{D_k}^{(k)} \end{bmatrix}$$

$$\mathbf{W}_k = \begin{bmatrix} w_{11}^{(k)} & w_{21}^{(k)} & \vdots & w_{D_{k-1}1}^{(k)} \\ w_{12}^{(k)} & w_{22}^{(k)} & \vdots & w_{D_{k-1}2}^{(k)} \\ \dots & \dots & \ddots & \vdots \\ w_{1D_k}^{(k)} & w_{2D_k}^{(k)} & \dots & w_{D_{k-1}D_k}^{(k)} \end{bmatrix}$$

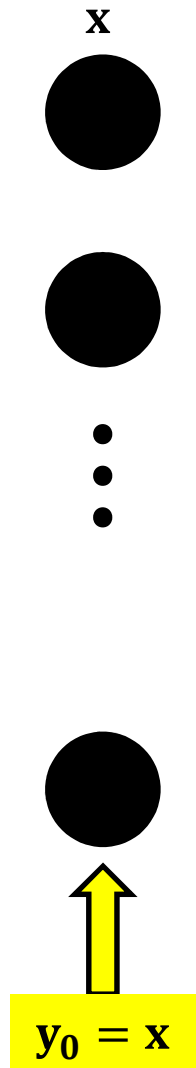
$$\mathbf{b}_k = \begin{bmatrix} b_1^{(k)} \\ b_2^{(k)} \\ \vdots \\ b_{D_{k+1}}^{(k)} \end{bmatrix}$$

- The computation of a single layer is easily expressed in matrix notation as (setting $\mathbf{y}_0 = \mathbf{x}$):

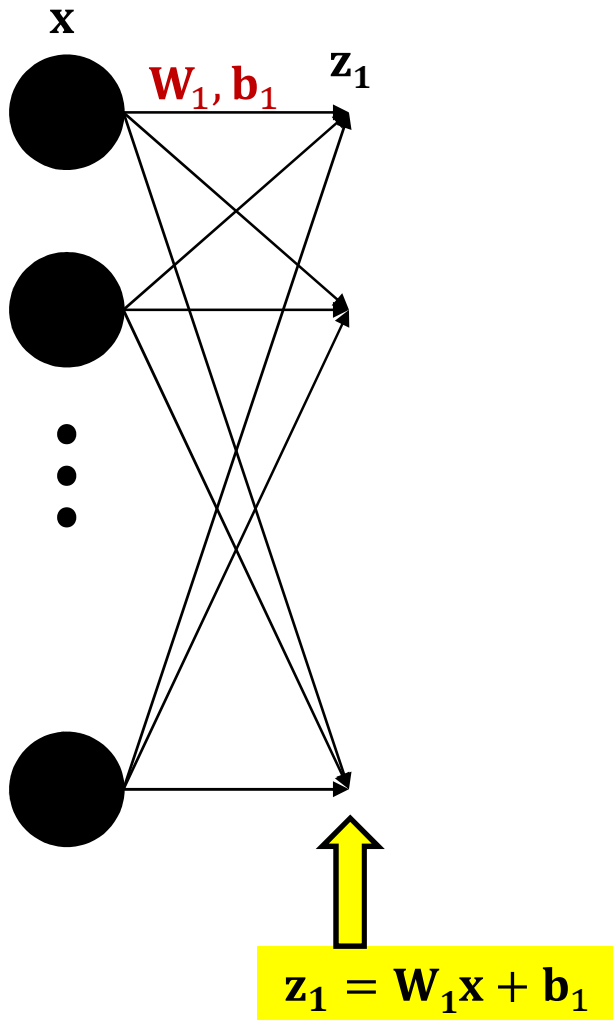
$$\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k$$

$$\mathbf{y}_k = f_k(\mathbf{z}_k)$$

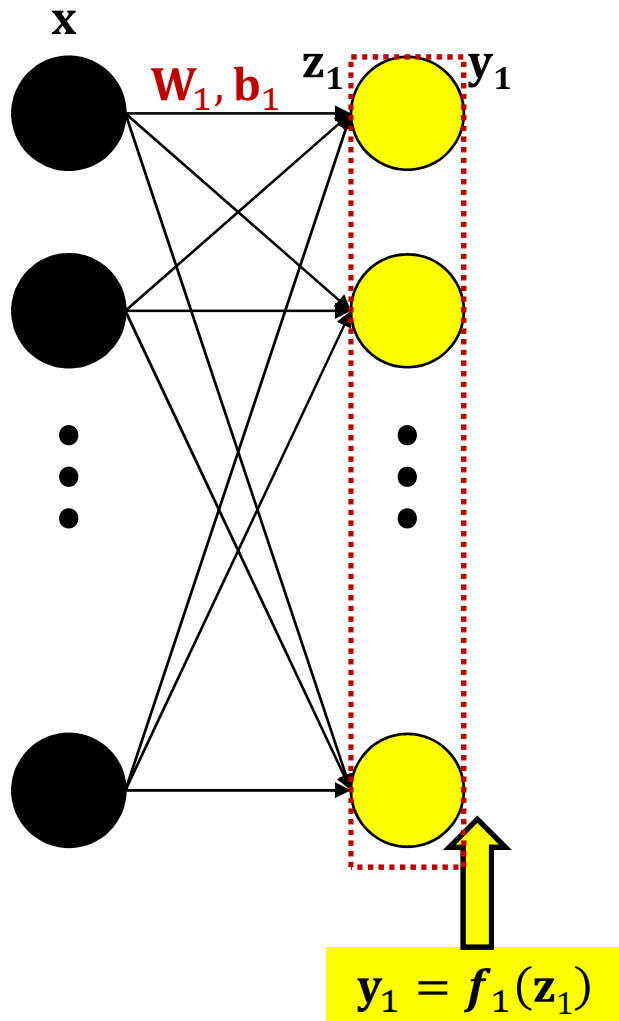
The forward pass: Evaluating the network



The forward pass



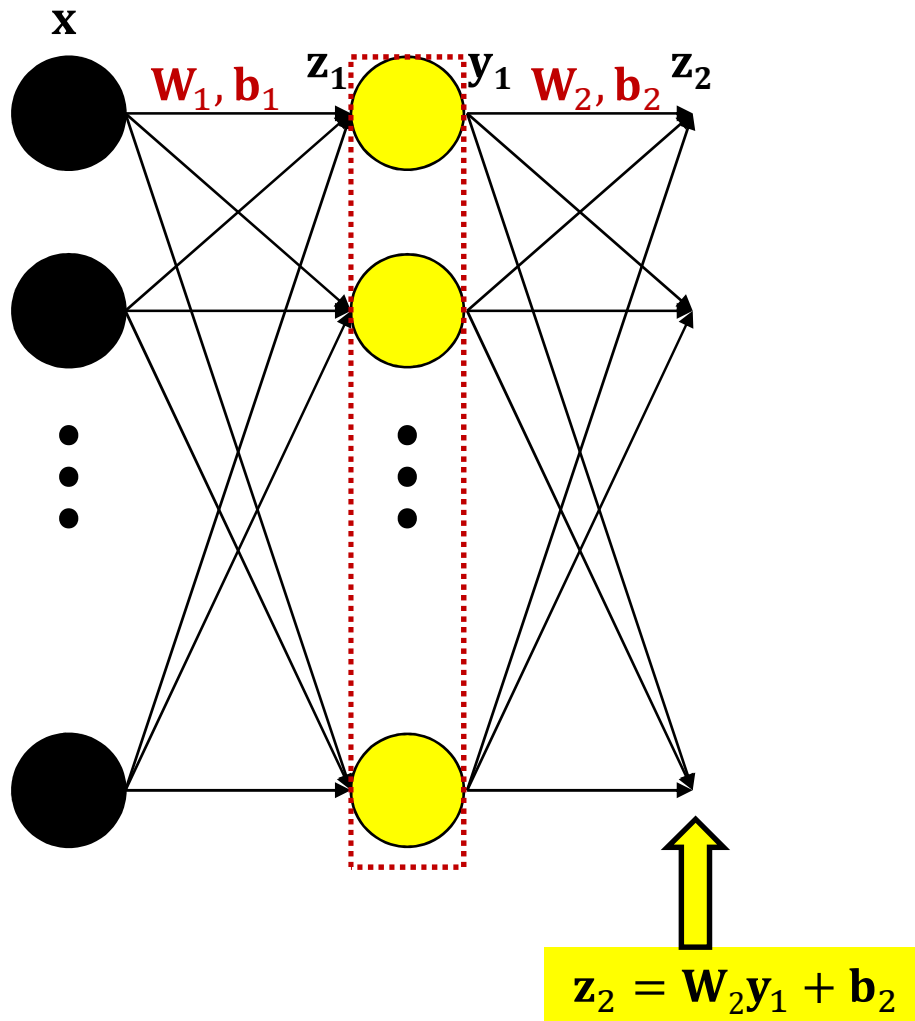
The forward pass



The Complete computation

$$\mathbf{y}_1 = f_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

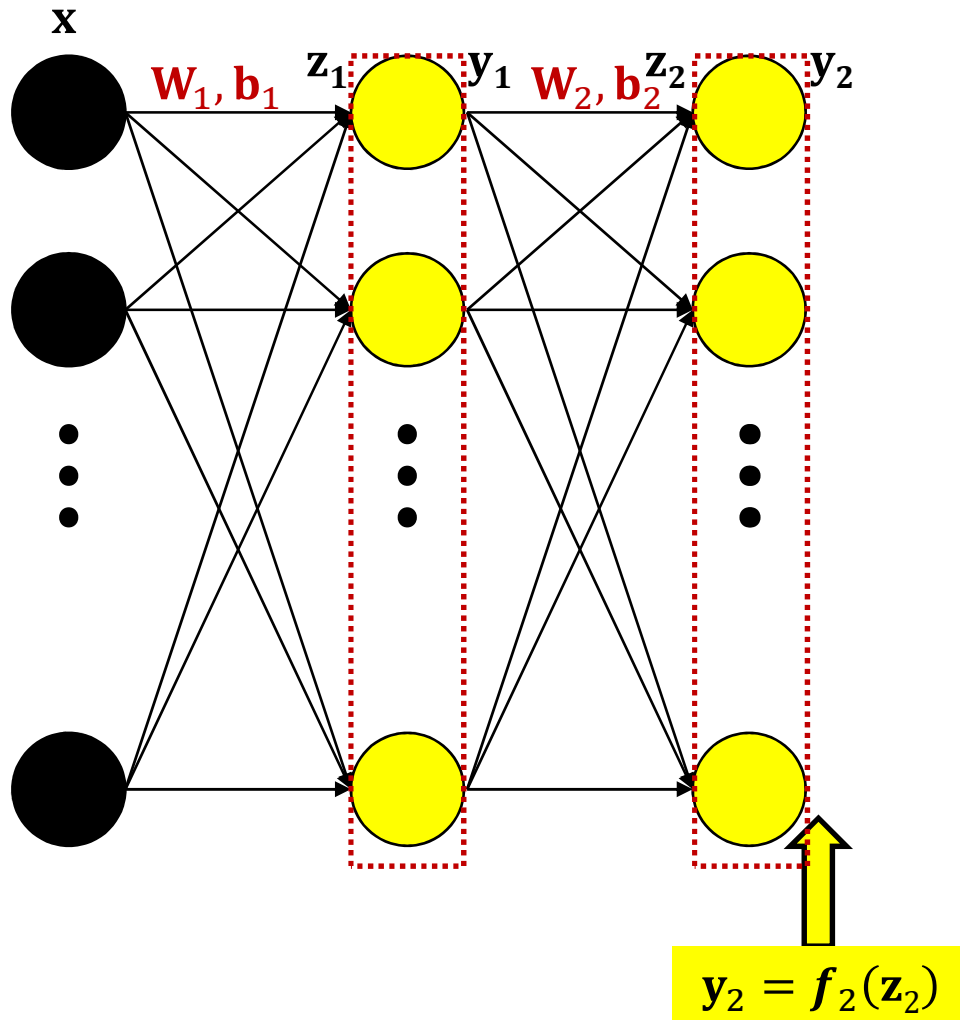
The forward pass



The Complete computation

$$\mathbf{y}_1 = f_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

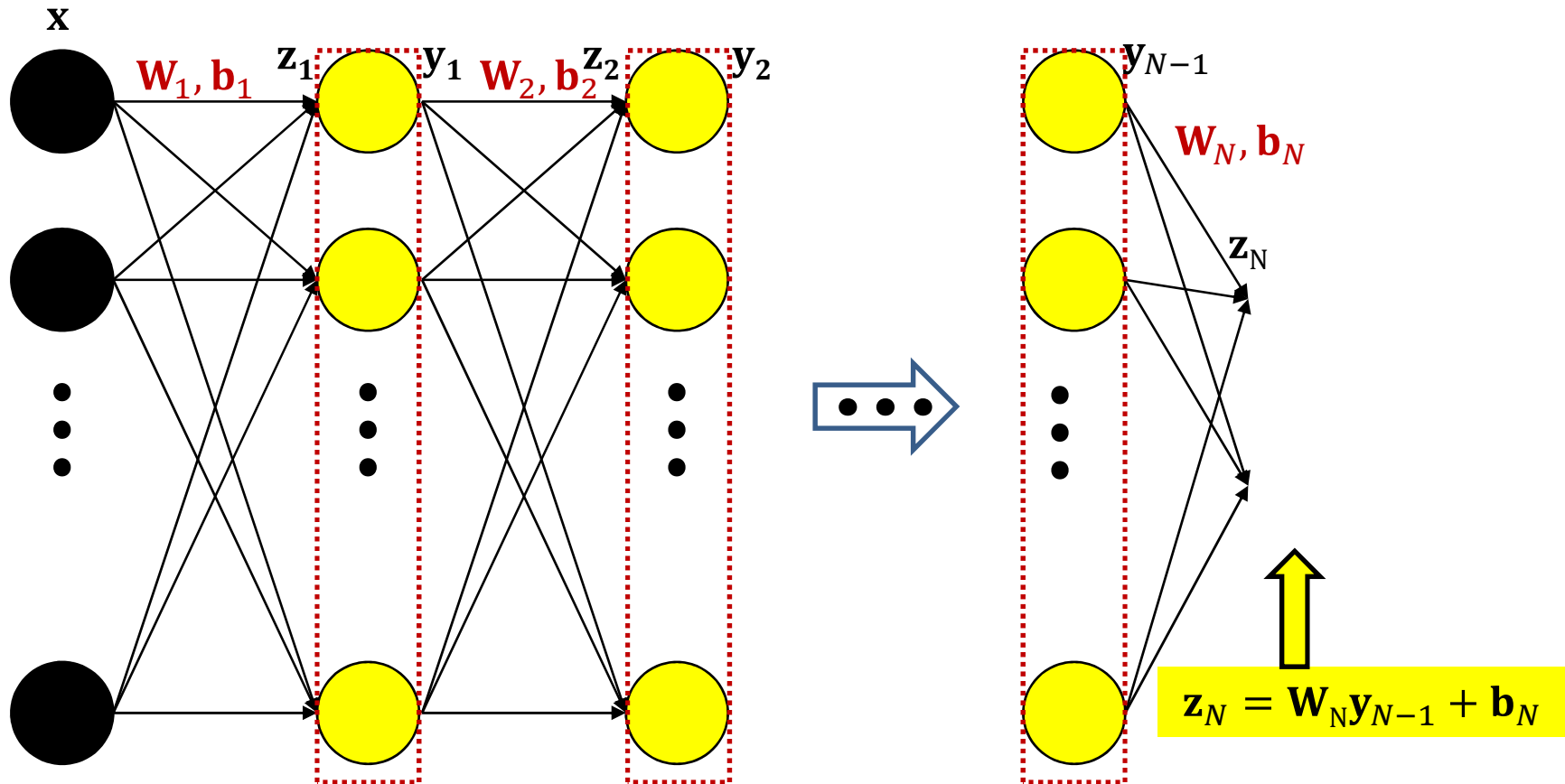
The forward pass



The Complete computation

$$\mathbf{y}_2 = f_2(\mathbf{W}_2 f_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$$

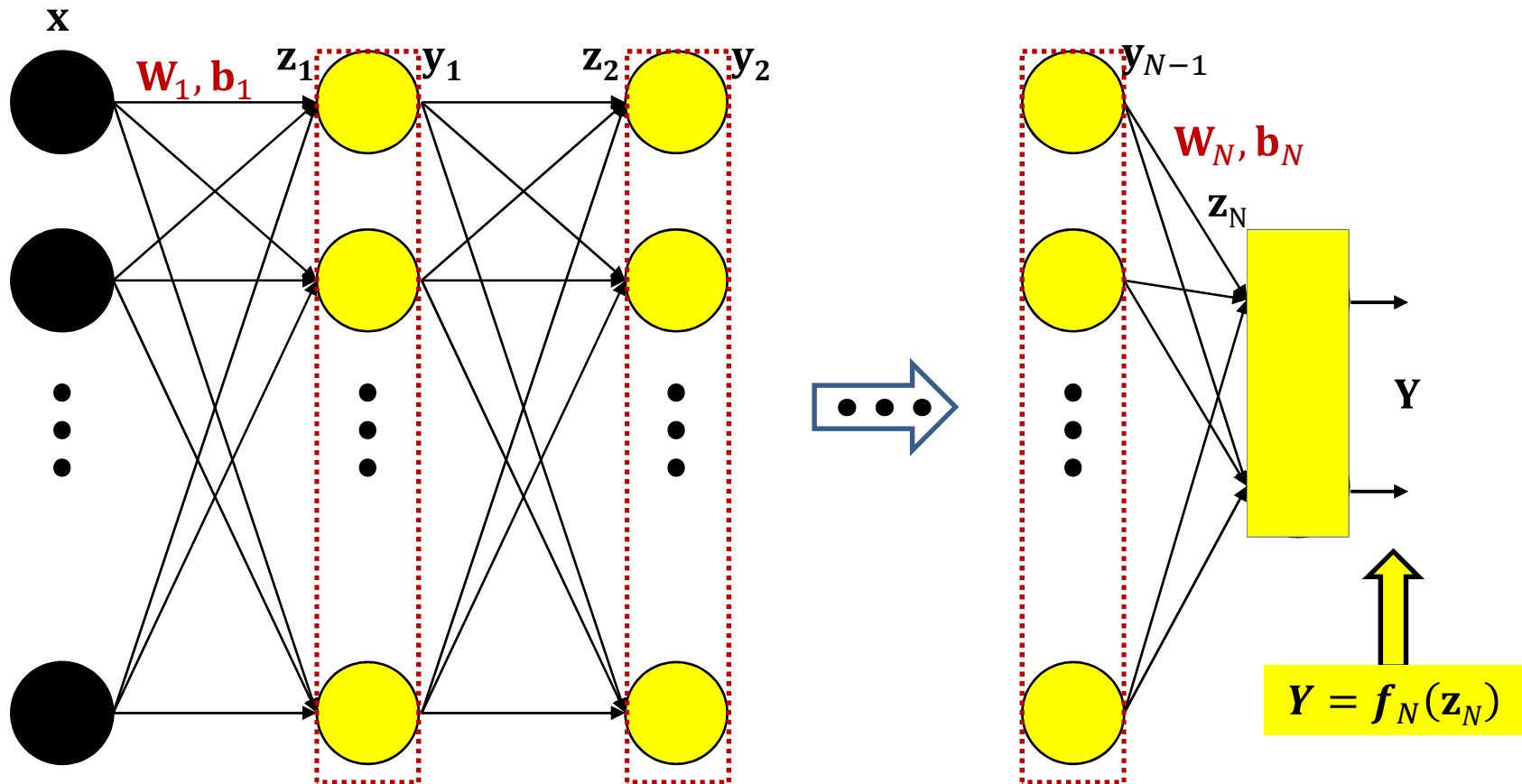
The forward pass



The Complete computation

$$y_2 = f_2(W_2 f_1(W_1 x + b_1) + b_2)$$

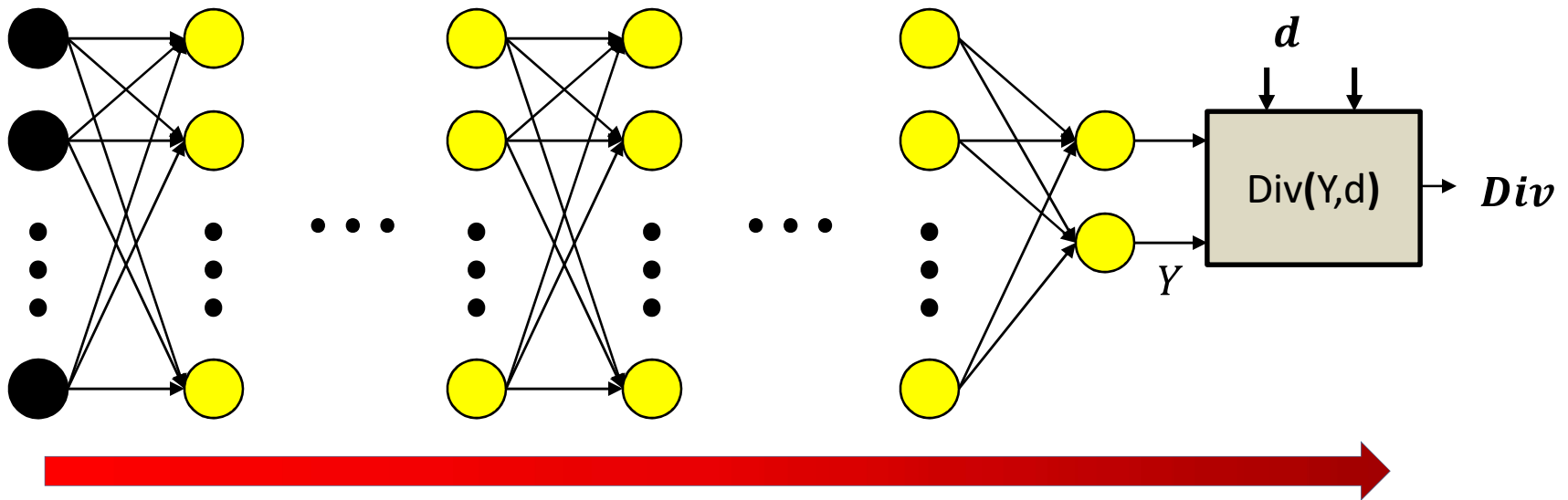
The forward pass



The Complete computation

$$Y = f_N(\mathbf{W}_N f_{N-1}(\dots f_2(\mathbf{W}_2 f_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) \dots) + \mathbf{b}_N)$$

Forward pass



Forward pass:

Initialize

$$\mathbf{y}_0 = \mathbf{x}$$

For $k = 1$ to N :

$$\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k$$

$$\mathbf{y}_k = f_k(\mathbf{z}_k)$$

Output

$$\mathbf{Y} = \mathbf{y}_N$$

The Forward Pass

- Set $\mathbf{y}_0 = \mathbf{x}$
- Recursion through layers:
 - For layer $k = 1$ to N :

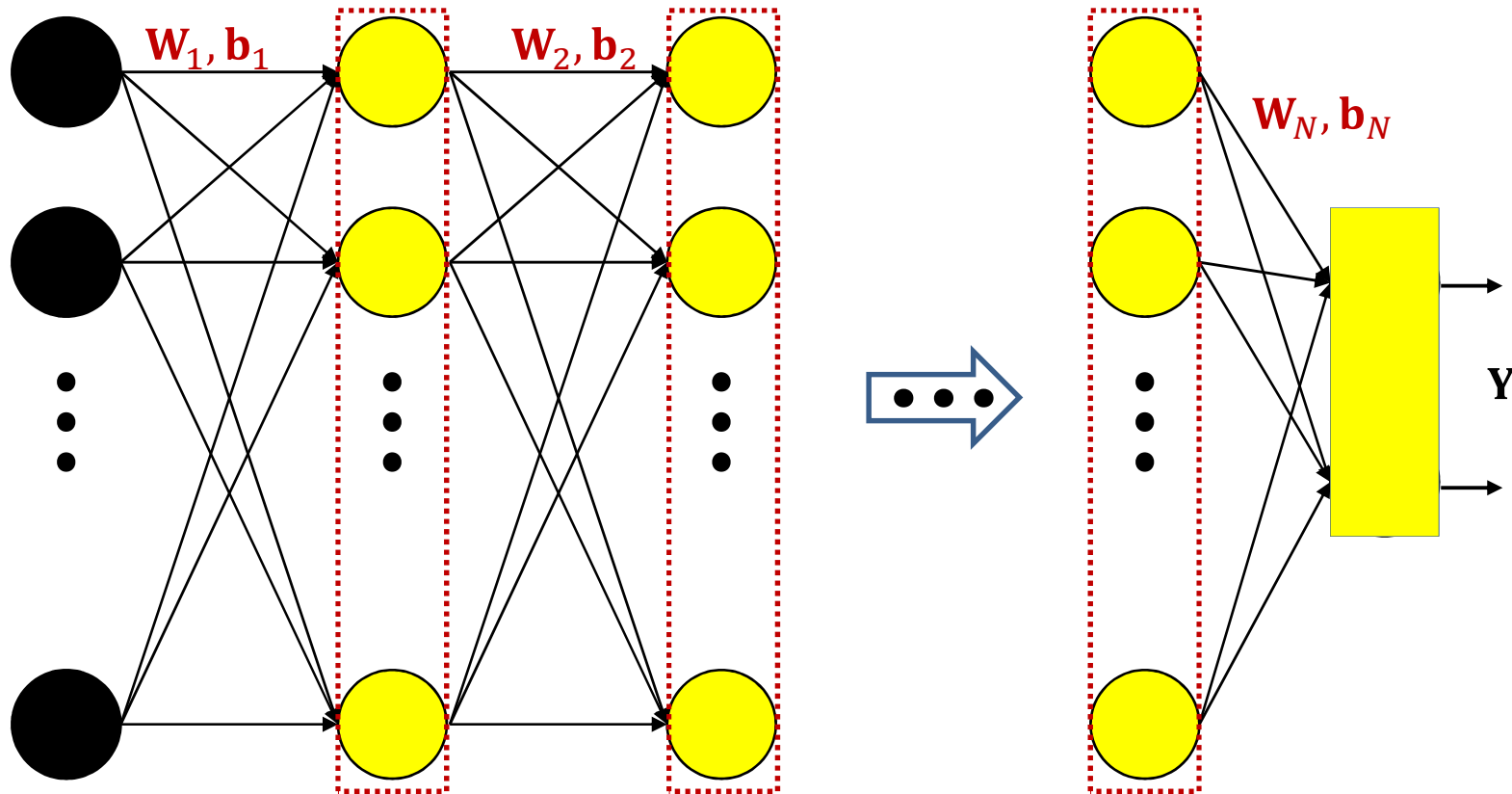
$$\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k$$

$$\mathbf{y}_k = \mathbf{f}_k(\mathbf{z}_k)$$

- Output:

$$\mathbf{Y} = \mathbf{y}_N$$

The backward pass



- The network is a nested function

$$Y = f_N(\mathbf{W}_N f_{N-1}(\dots f_2(\mathbf{W}_2 f_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) \dots) + \mathbf{b}_N)$$

- The error for any \mathbf{x} is also a nested function

$$Div(Y, d) = Div(f_N(\mathbf{W}_N f_{N-1}(\dots f_2(\mathbf{W}_2 f_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) \dots) + \mathbf{b}_N), d)$$

Calculus recap 2: The Jacobian

- The derivative of a vector function w.r.t. vector input is called a *Jacobian*
- It is the matrix of partial derivatives given below

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} = f \left(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_D \end{bmatrix} \right)$$

Using vector notation

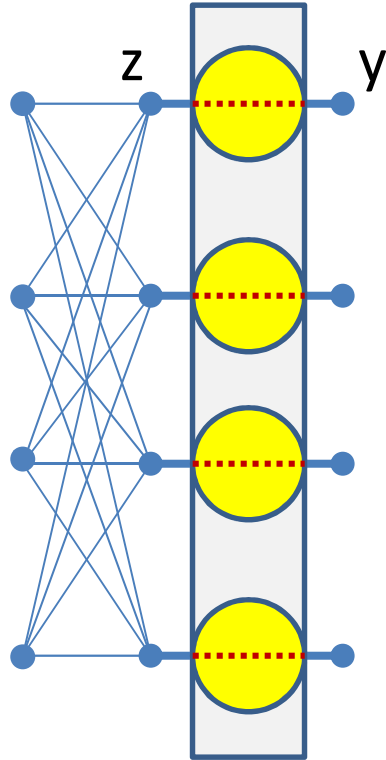
$$\mathbf{y} = f(\mathbf{z})$$

$$J_y(\mathbf{z}) = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \cdots & \frac{\partial y_1}{\partial z_D} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \cdots & \frac{\partial y_2}{\partial z_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_M}{\partial z_1} & \frac{\partial y_M}{\partial z_2} & \cdots & \frac{\partial y_M}{\partial z_D} \end{bmatrix}$$

Check:

$$\Delta \mathbf{y} = J_y(\mathbf{z}) \Delta \mathbf{z}$$

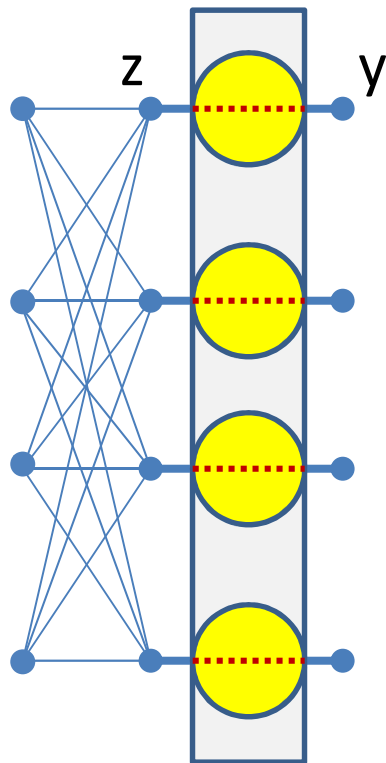
Jacobians can describe the derivatives of neural activations w.r.t their input



$$J_y(\mathbf{z}) = \begin{bmatrix} \frac{dy_1}{dz_1} & 0 & \dots & 0 \\ 0 & \frac{dy_2}{dz_2} & \dots & 0 \\ \dots & \dots & \ddots & \dots \\ 0 & 0 & \dots & \frac{dy_D}{dz_D} \end{bmatrix}$$

- **For Scalar activations**
 - Number of outputs is identical to the number of inputs
- Jacobian is a diagonal matrix
 - Diagonal entries are individual derivatives of outputs w.r.t inputs
 - Not showing the superscript “(k)” in equations for brevity

Jacobians can describe the derivatives of neural activations w.r.t their input

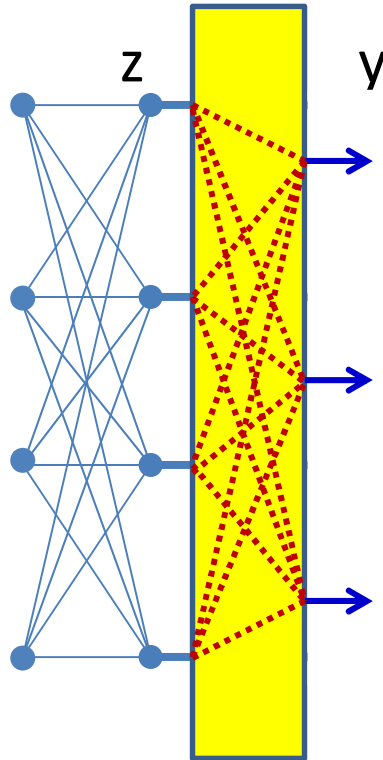


$$y_i = f(z_i)$$

$$J_y(\mathbf{z}) = \begin{bmatrix} f'(z_1) & 0 & \dots & 0 \\ 0 & f'(z_2) & \dots & 0 \\ \dots & \dots & \ddots & \dots \\ 0 & 0 & \dots & f'(z_M) \end{bmatrix}$$

- **For scalar activations (shorthand notation):**
 - Jacobian is a diagonal matrix
 - Diagonal entries are individual derivatives of outputs w.r.t inputs

For *Vector* activations



$$J_{\mathbf{y}}(\mathbf{z}) = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \dots & \frac{\partial y_1}{\partial z_D} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \dots & \frac{\partial y_2}{\partial z_D} \\ \dots & \dots & \ddots & \dots \\ \frac{\partial y_M}{\partial z_1} & \frac{\partial y_M}{\partial z_2} & \dots & \frac{\partial y_M}{\partial z_D} \end{bmatrix}$$

- Jacobian is a full matrix
 - Entries are partial derivatives of individual outputs w.r.t individual inputs

Special case: Affine functions

$$\mathbf{z} = \mathbf{W}\mathbf{y} + \mathbf{b}$$

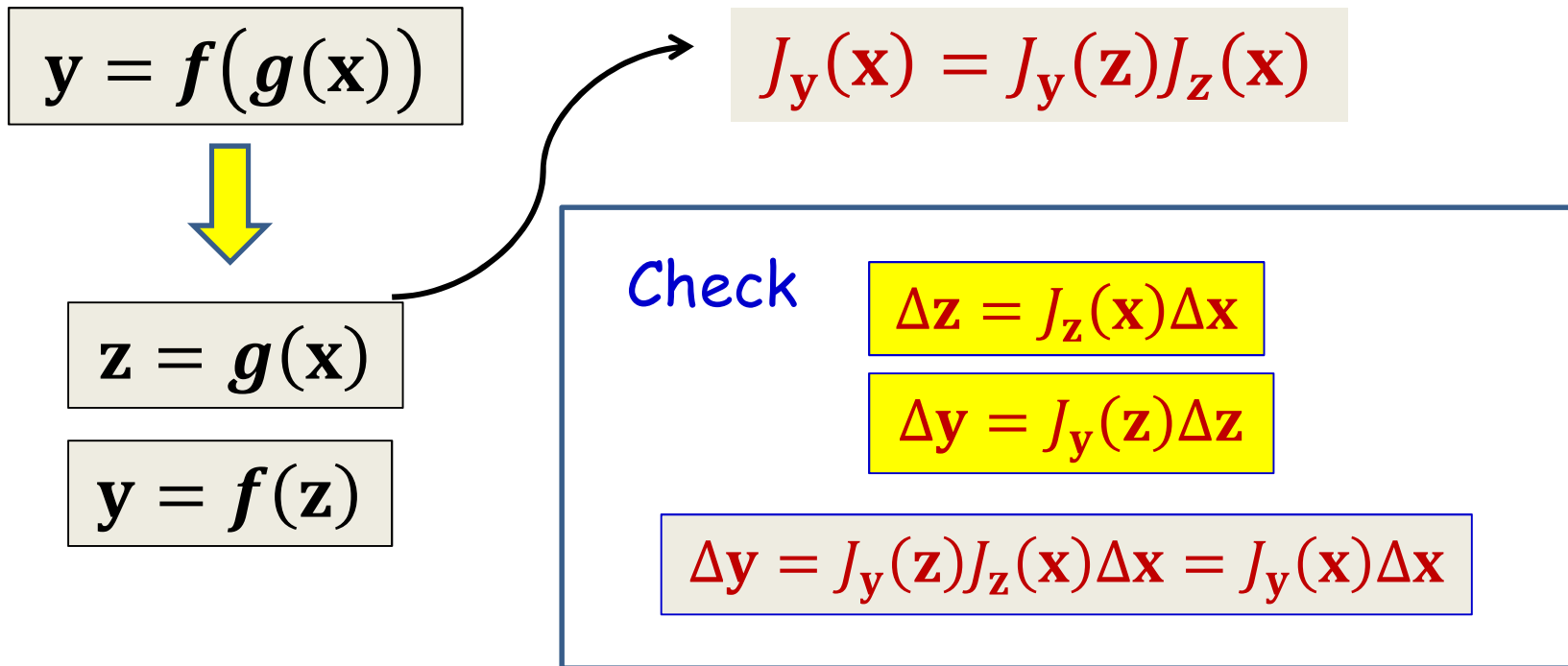


$$J_{\mathbf{z}}(\mathbf{y}) = \mathbf{W}$$

- Matrix \mathbf{W} and bias \mathbf{b} operating on vector \mathbf{y} to produce vector \mathbf{z}
- The Jacobian of \mathbf{z} w.r.t \mathbf{y} is simply the matrix \mathbf{W}

Vector derivatives: Chain rule

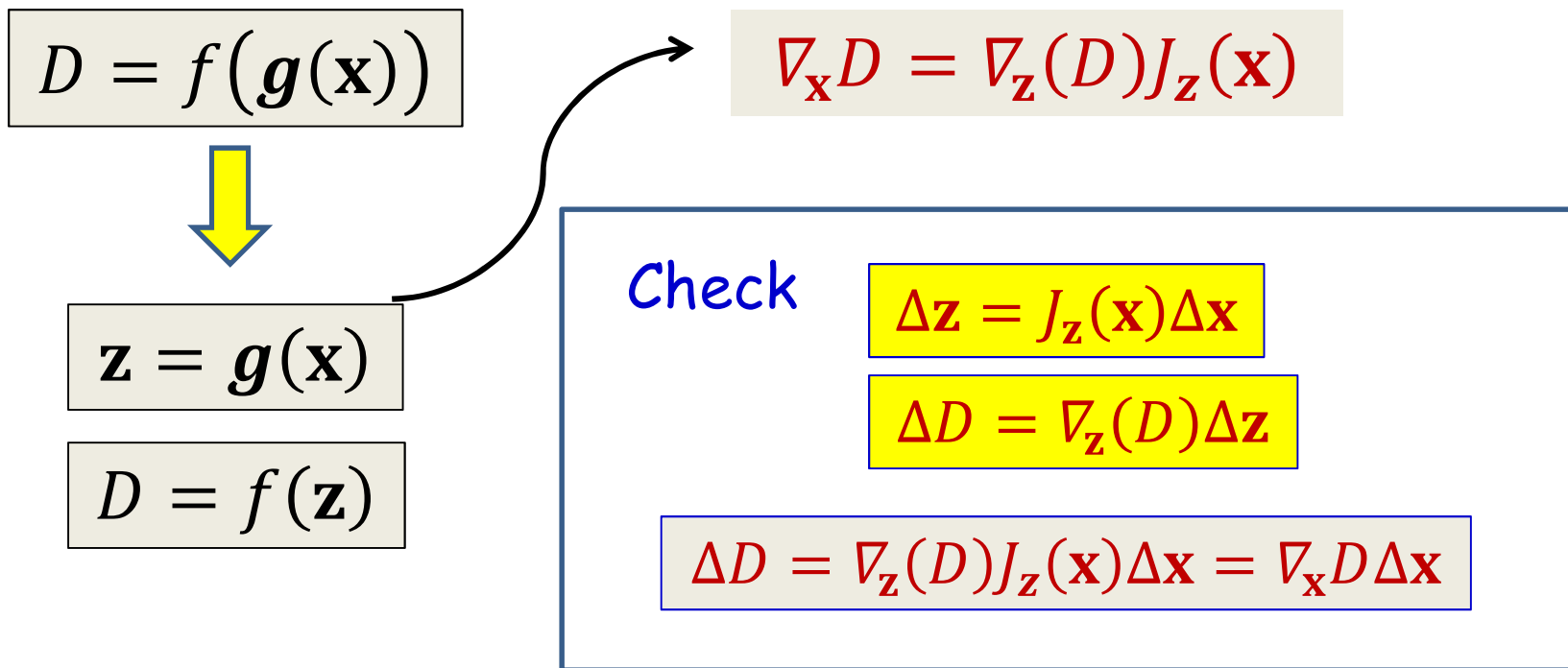
- We can define a chain rule for Jacobians
- **For vector functions of vector inputs:**



Note the order: The derivative of the outer function comes first

Vector derivatives: Chain rule

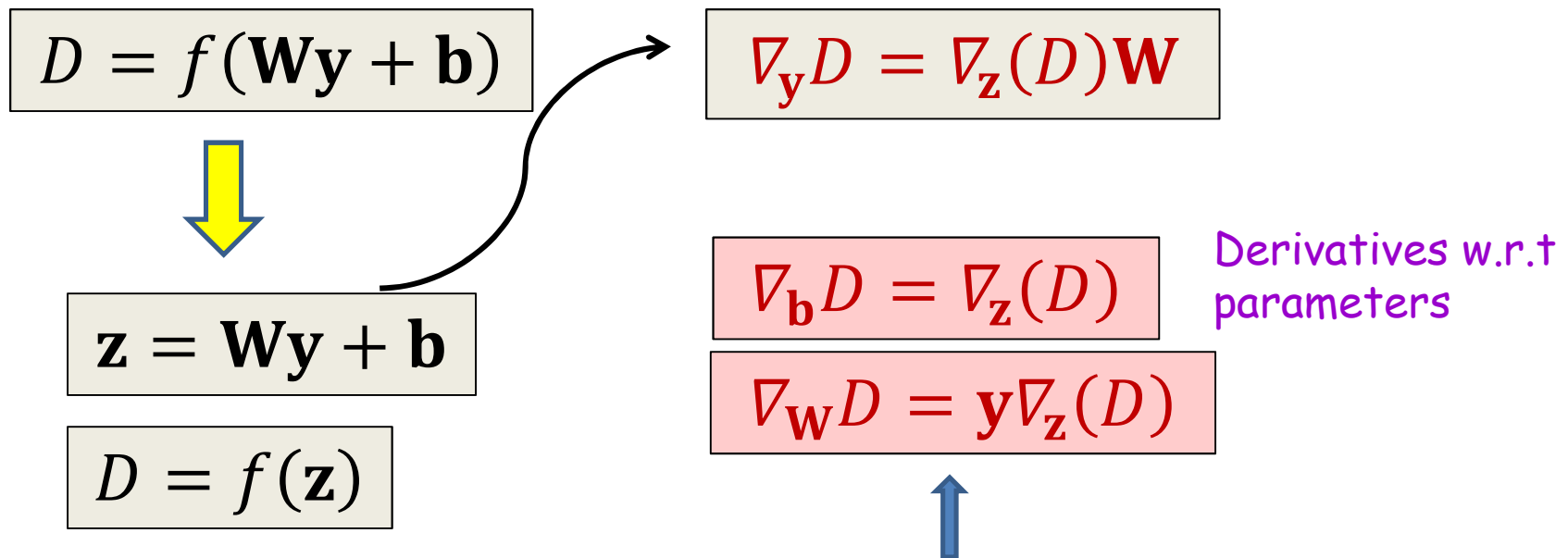
- *The chain rule can combine Jacobians and Gradients*
- **For scalar functions of vector inputs ($g()$ is vector):**



Note the order: The derivative of the outer function comes first

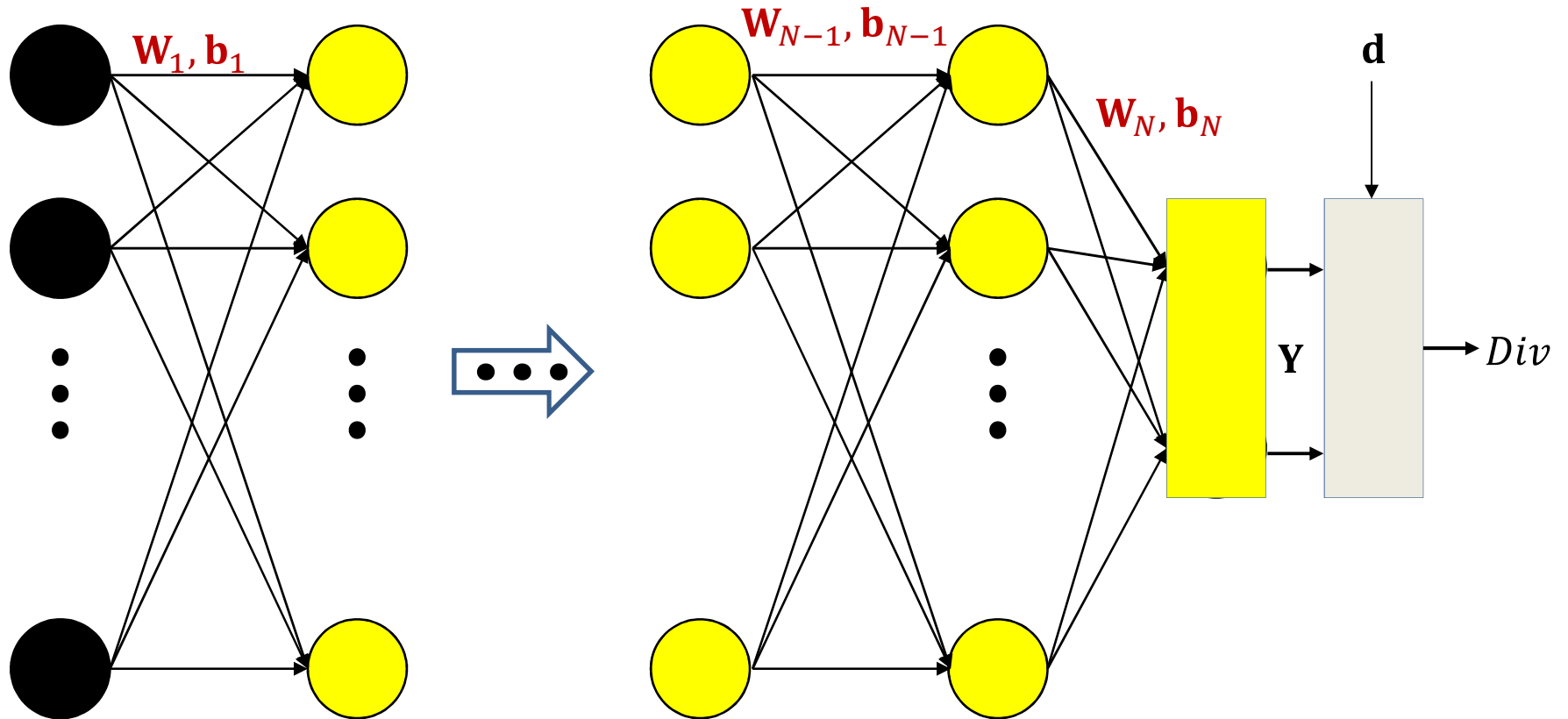
Special Case

- Scalar functions of Affine functions



Note reversal of order. This is in fact a simplification of a product of tensor terms that occur in the *right* order

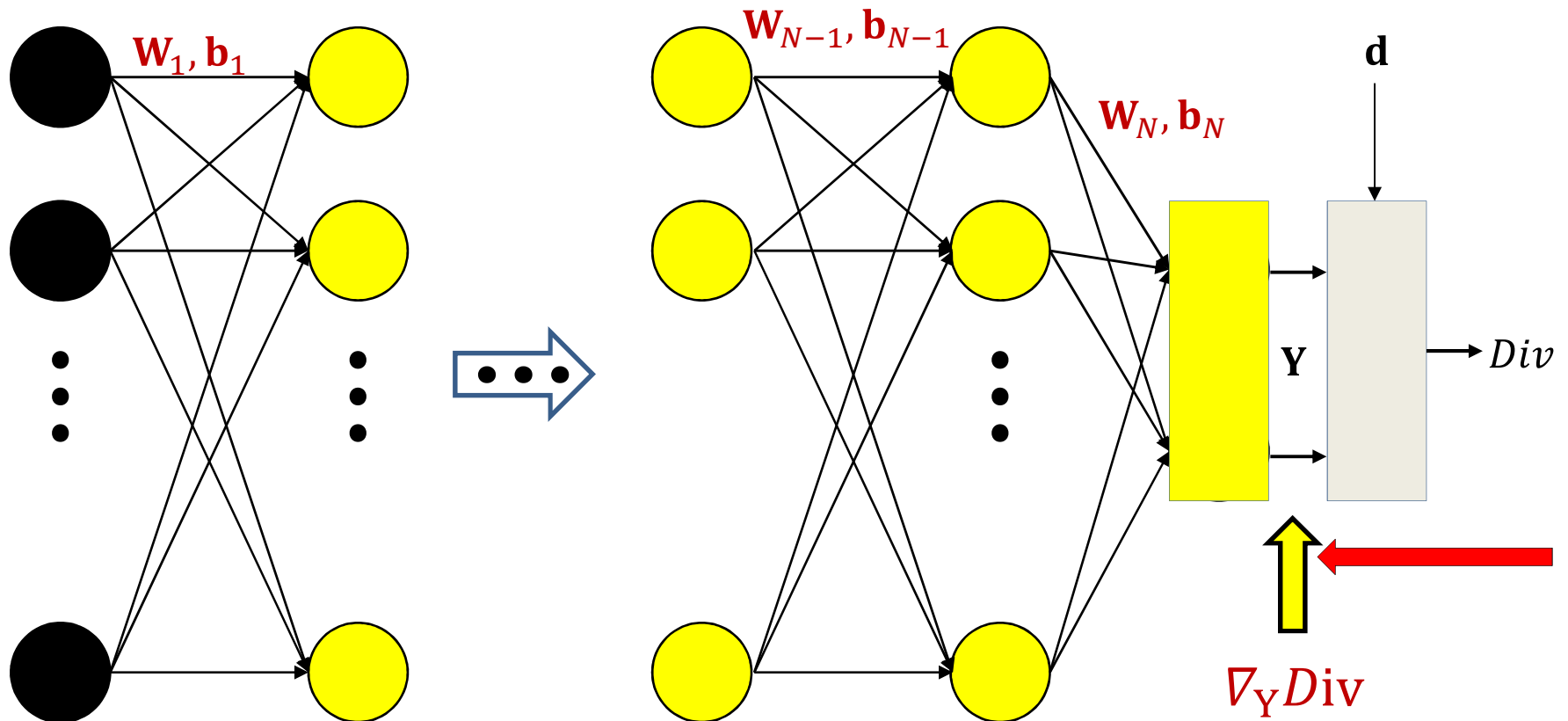
The backward pass



In the following slides we will also be using the notation $\nabla_z Y$ to represent the Jacobian $J_Y(z)$ to explicitly illustrate the chain rule

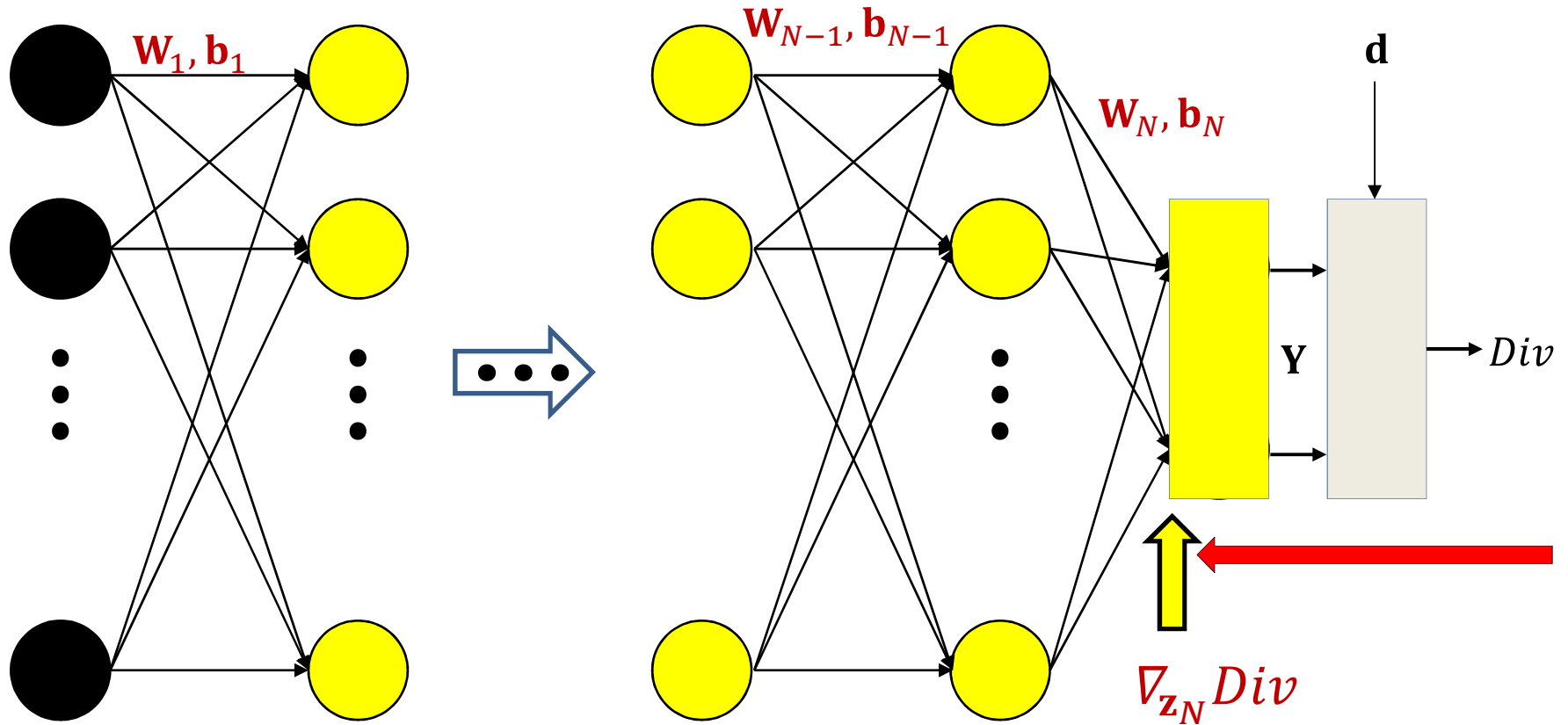
In general $\nabla_a \mathbf{b}$ represents a derivative of \mathbf{b} w.r.t. \mathbf{a} and could be a the transposed gradient (for scalar \mathbf{b}) or a Jacobian (for vector \mathbf{b})

The backward pass



First compute the gradient of the divergence w.r.t. Y .
The actual gradient depends on the divergence function.

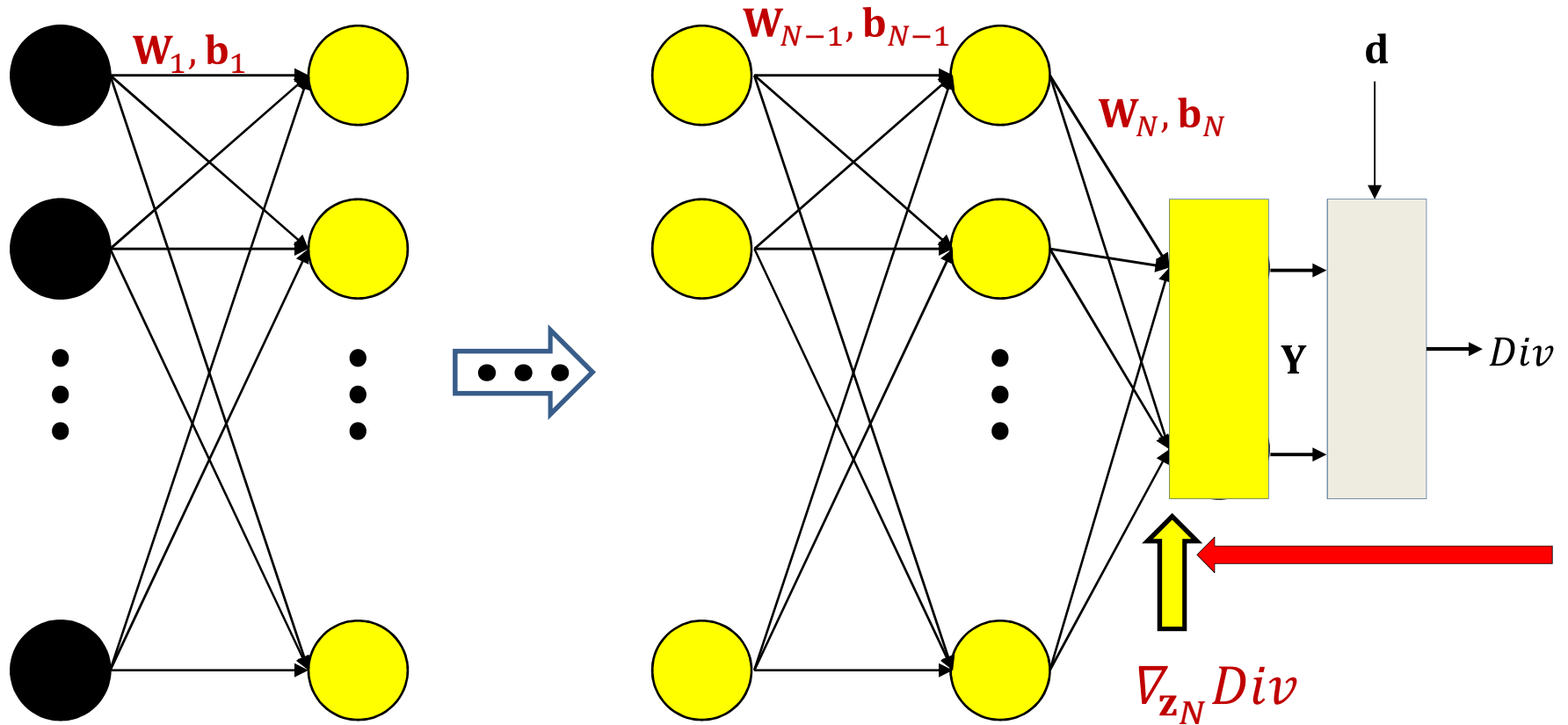
The backward pass



$$\nabla_{z_N} Div = \nabla_Y Div \cdot \nabla_{z_N} Y$$

Already computed New term

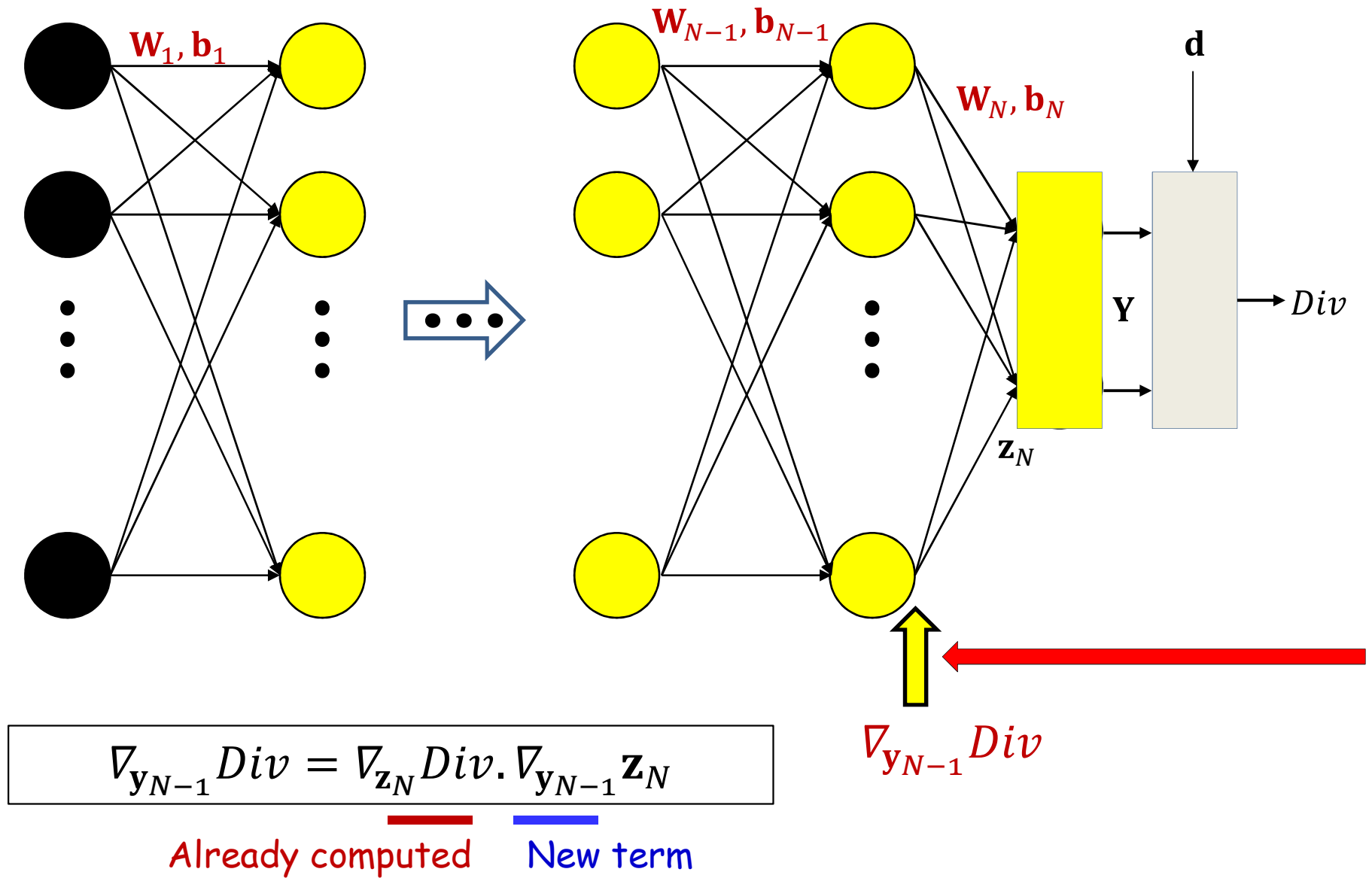
The backward pass



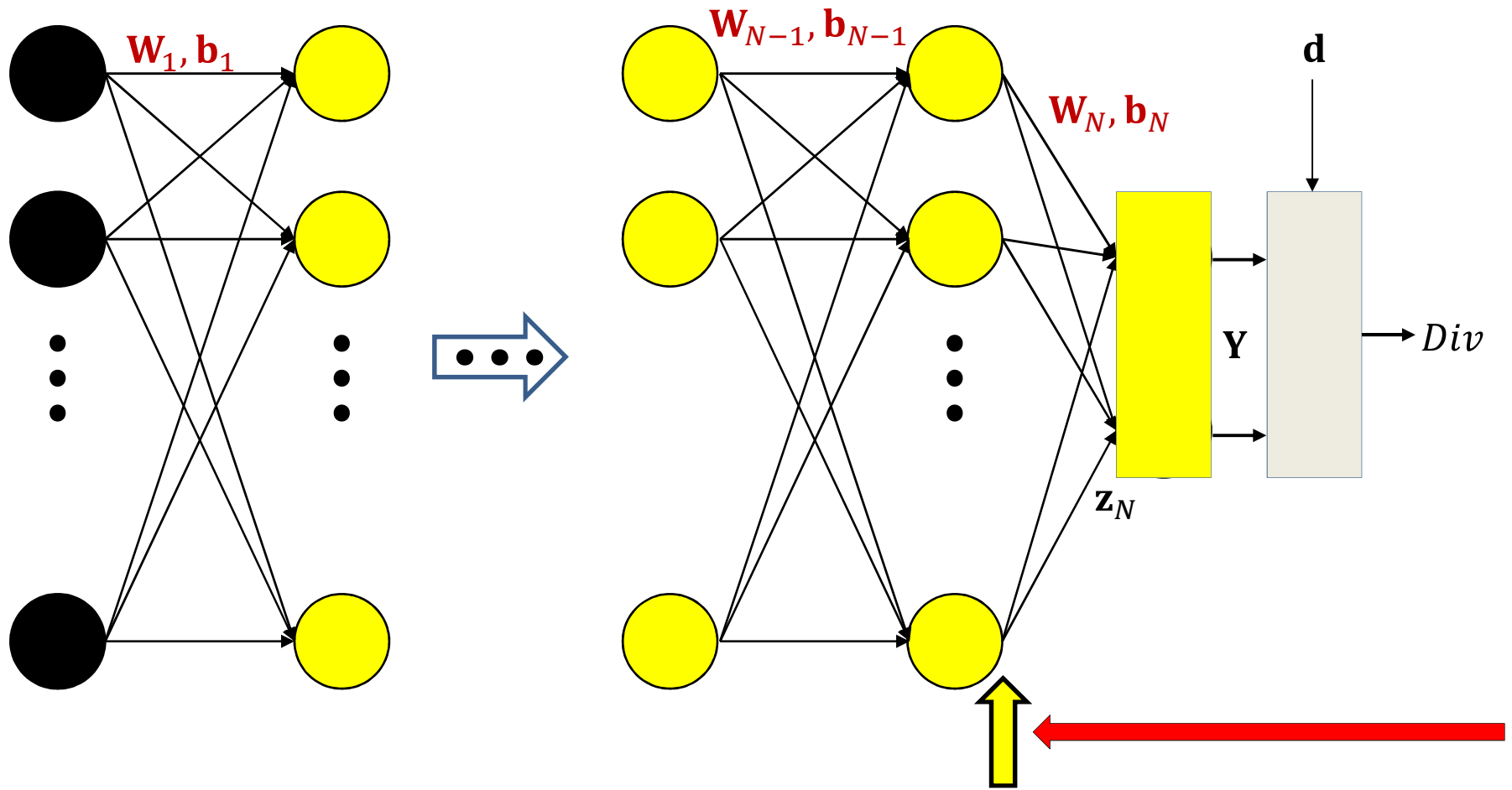
$$\nabla_{z_N} Div = \nabla_Y Div \cdot J_Y(z_N)$$

Already computed New term

The backward pass

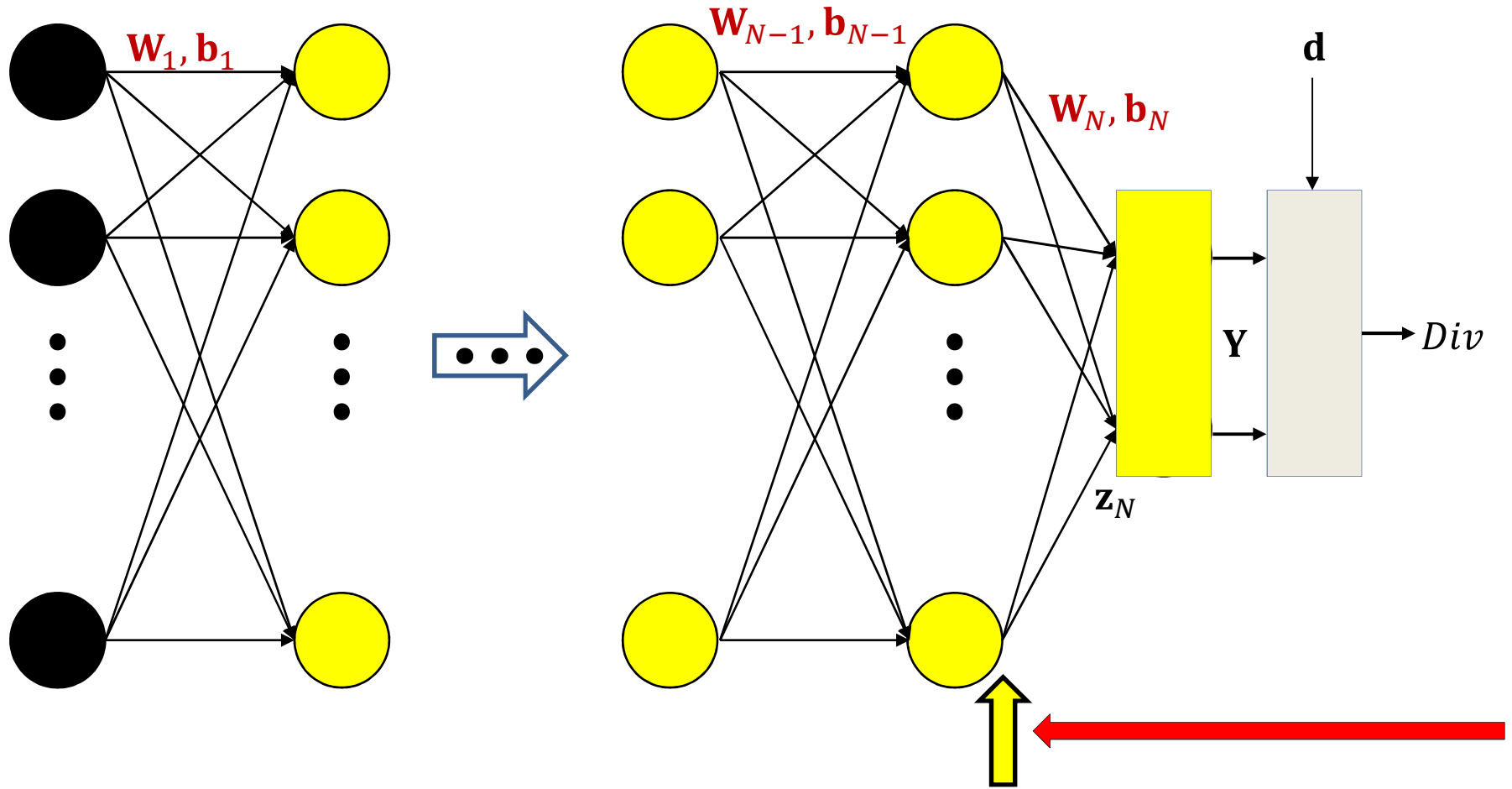


The backward pass



$$\nabla_{y_{N-1}} Div = \underbrace{\nabla_{z_N} Div}_{\text{Already computed}} \underbrace{W_N}_{\text{New term}}$$

The backward pass

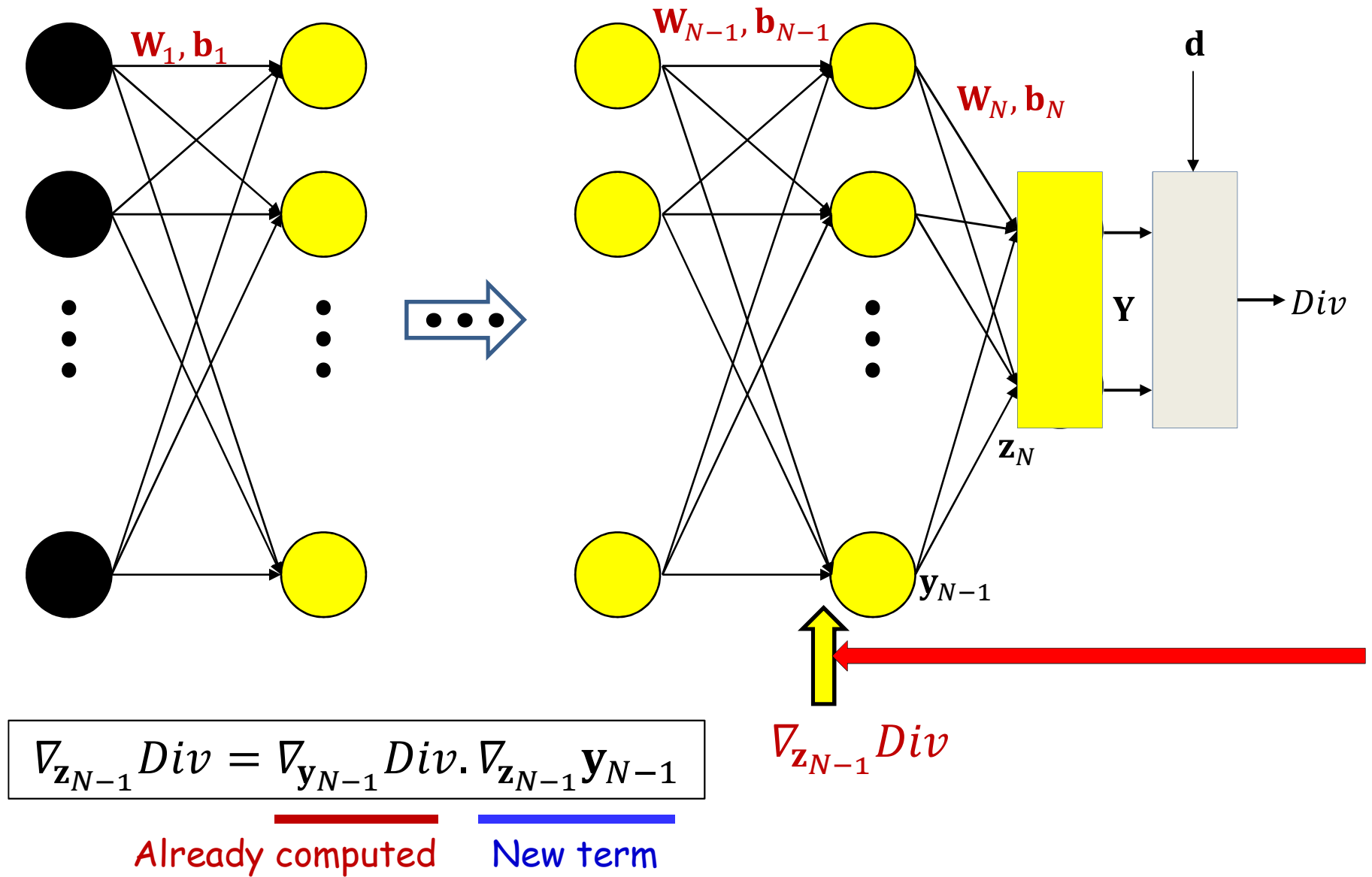


$$\nabla_{y_{N-1}} Div = \nabla_{z_N} Div W_N$$

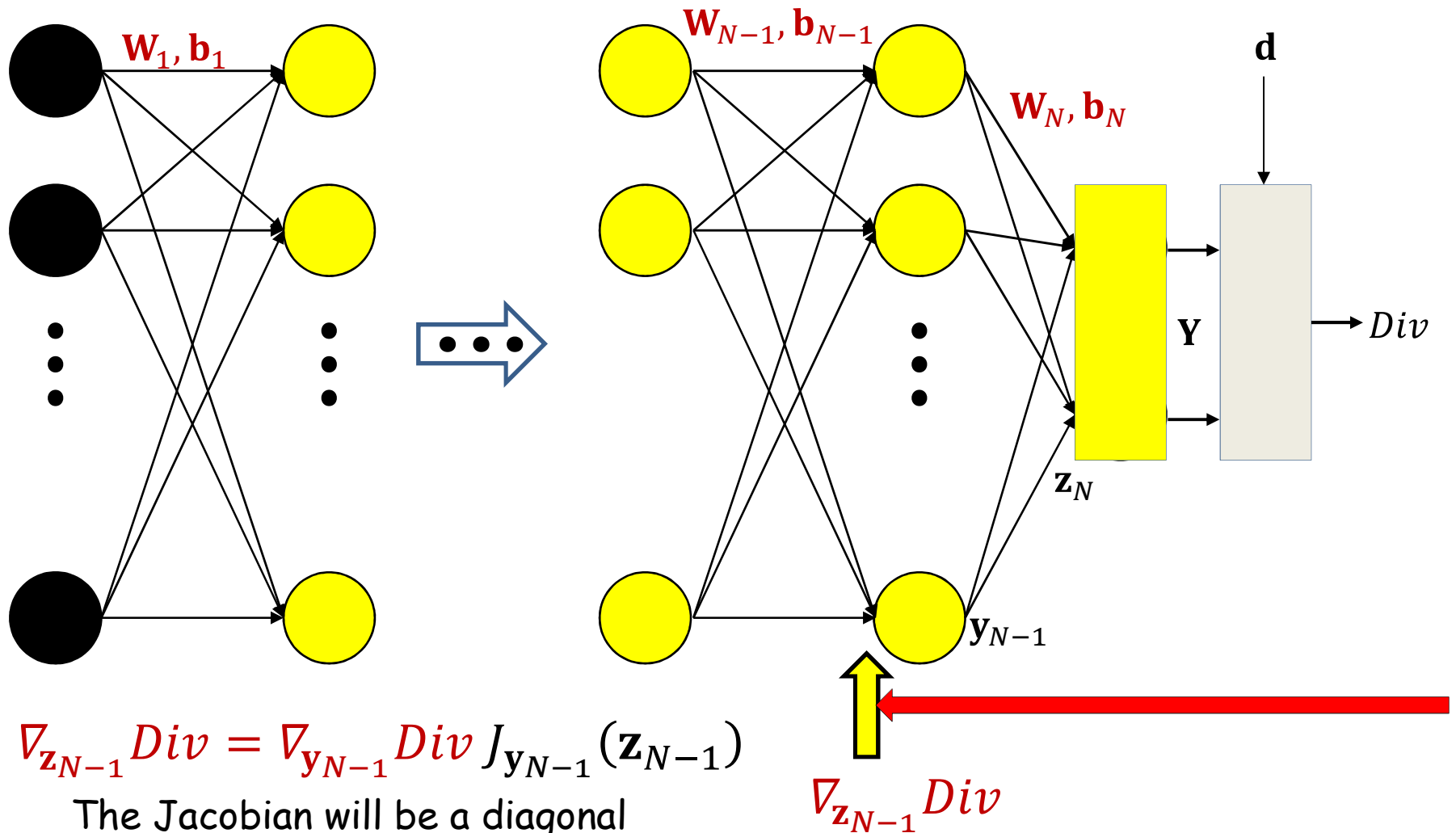
$$\nabla_{W_N} Div = y_{N-1} \nabla_{z_N} Div$$

$$\nabla_{b_N} Div = \nabla_{z_N} Div$$

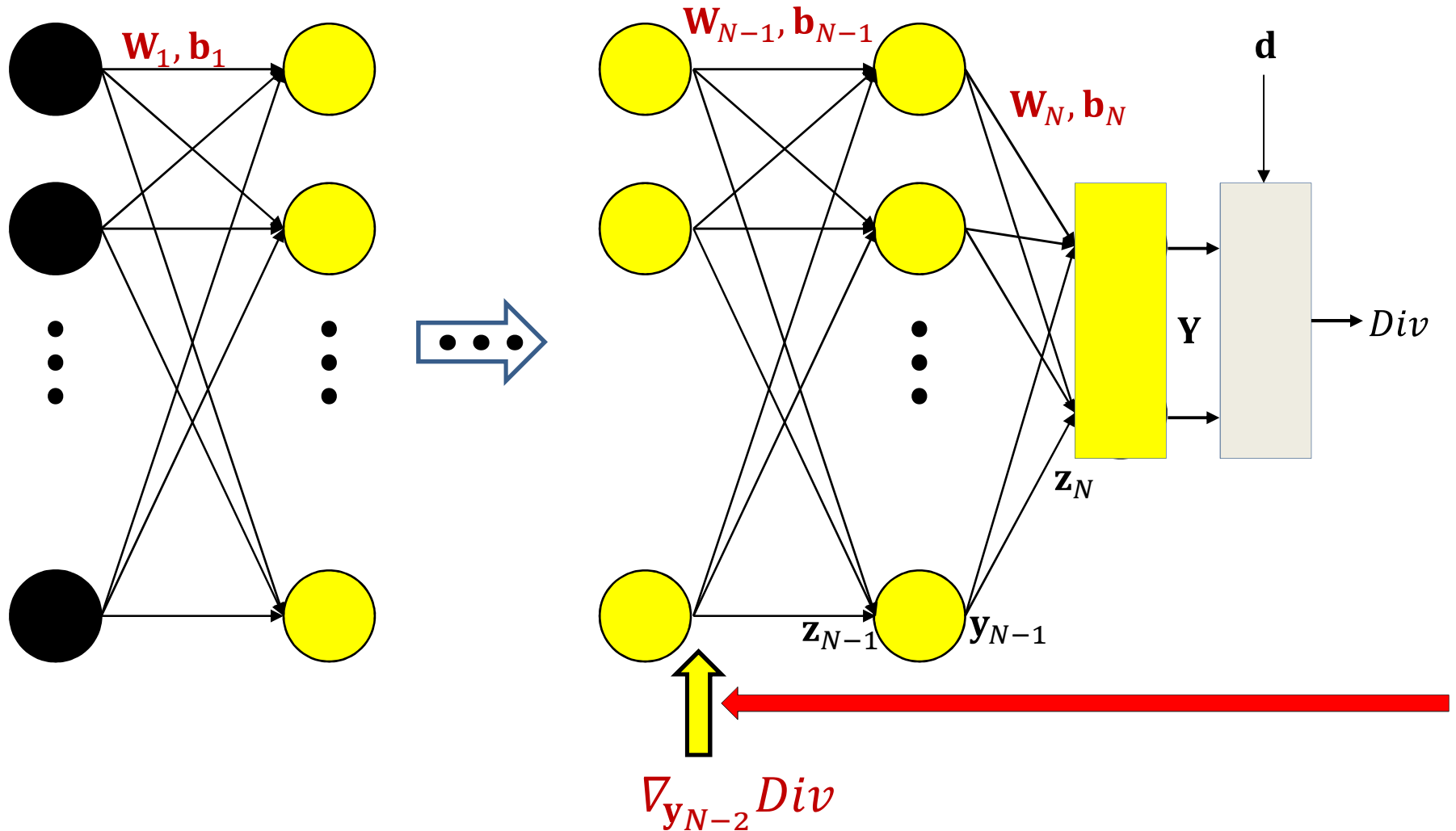
The backward pass



The backward pass

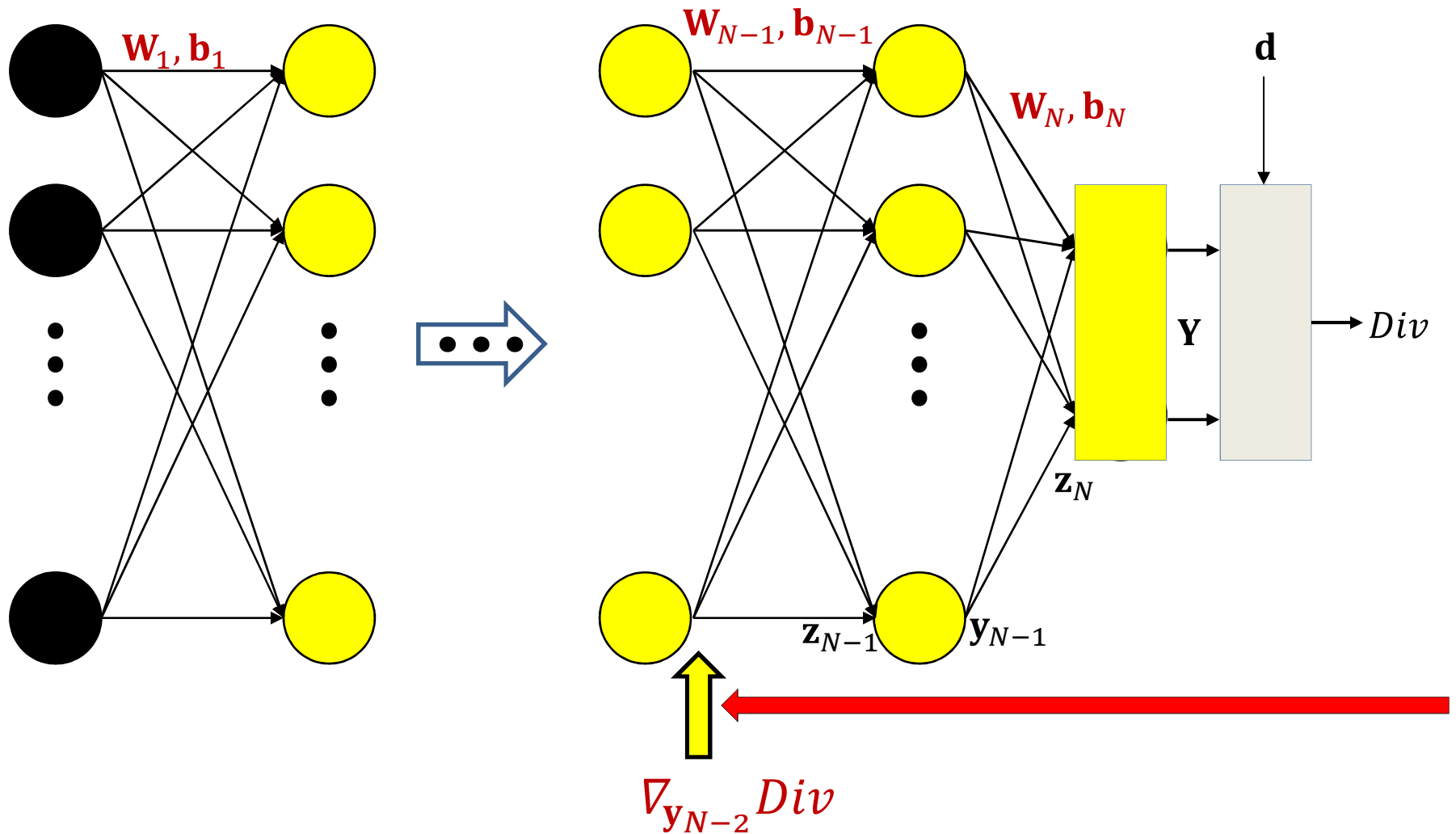


The backward pass



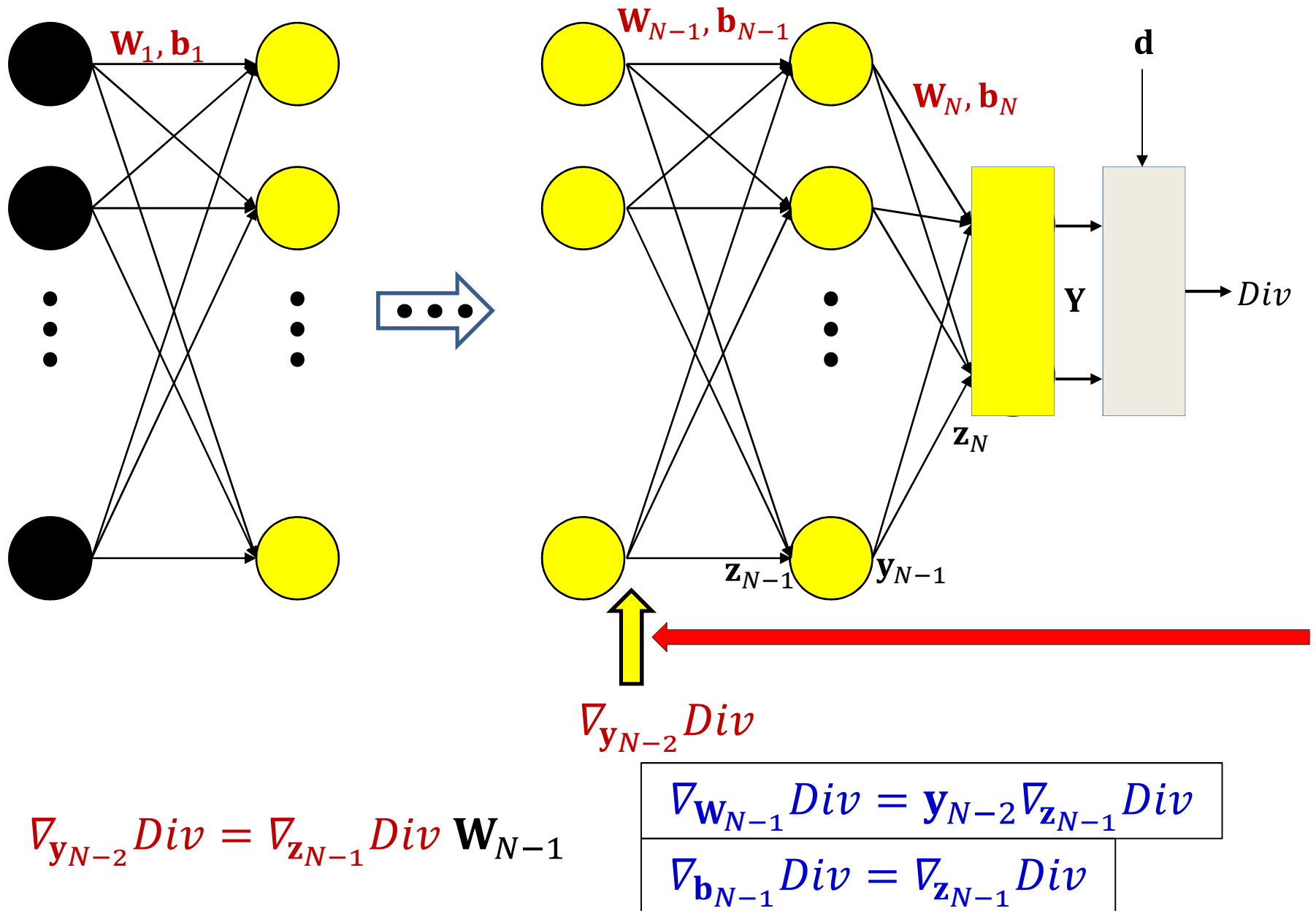
$$\nabla_{y_{N-2}} Div = \nabla_{z_{N-1}} Div \cdot \nabla_{y_{N-2}} z_{N-1}$$

The backward pass

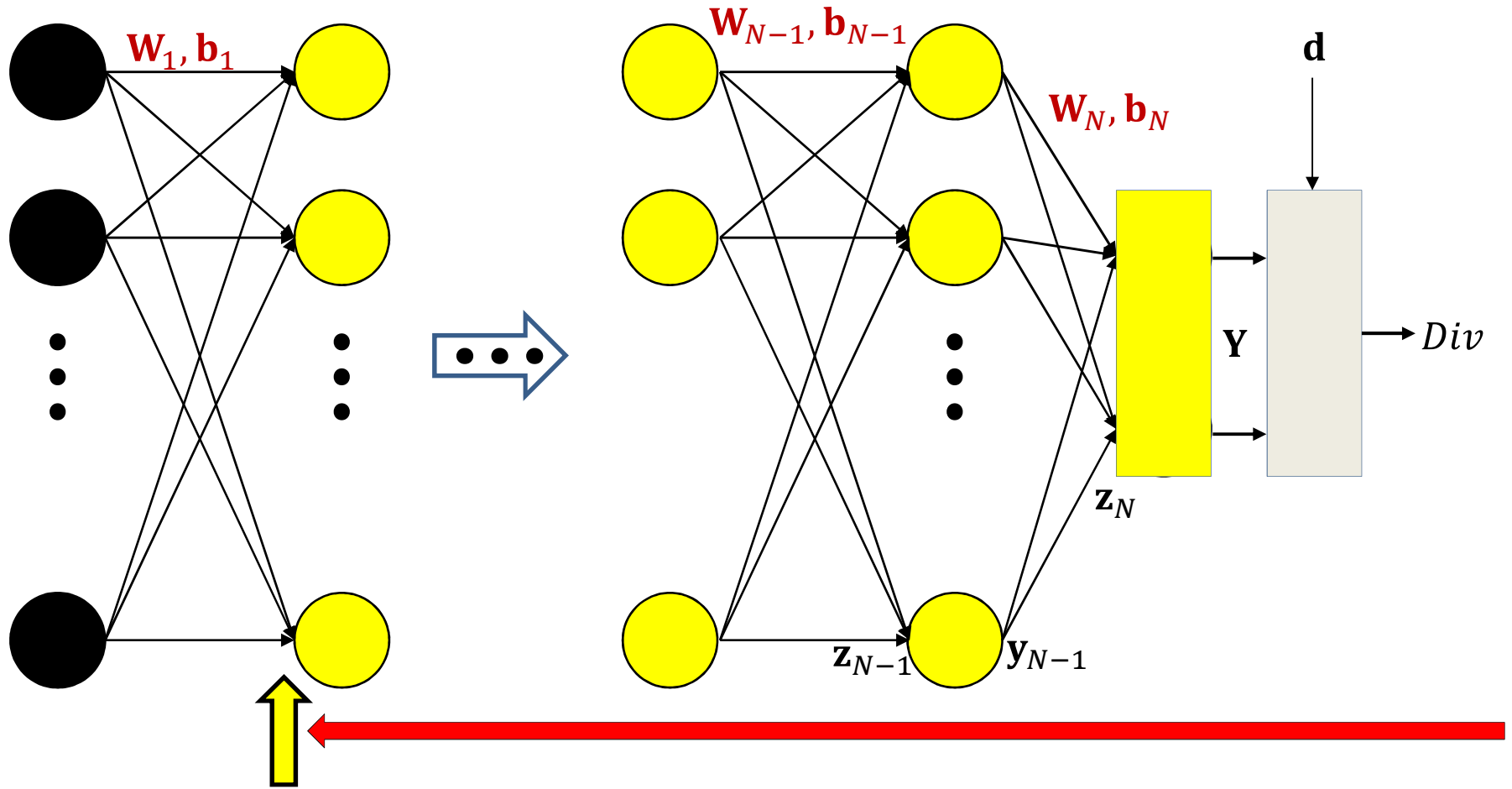


$$\nabla_{\mathbf{y}_{N-2}} Div = \nabla_{\mathbf{z}_{N-1}} Div \mathbf{W}_{N-1}$$

The backward pass

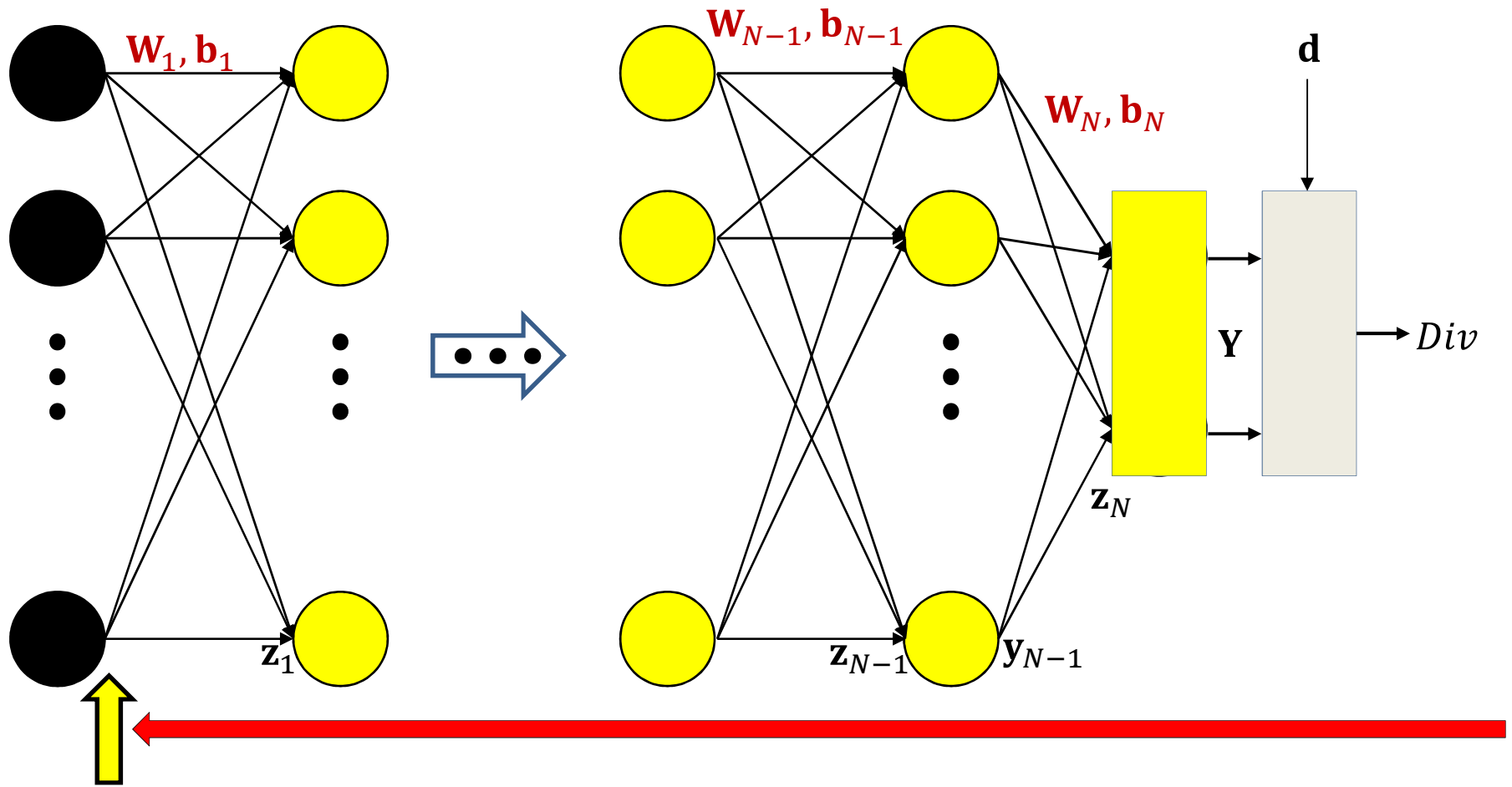


The backward pass



$$\nabla_{z_1} Div = \nabla_{y_1} Div J_{y_1}(z_1)$$

The backward pass



$$\nabla_{w_1} Div = x \nabla_{z_1} Div$$

$$\nabla_{b_1} Div = \nabla_{z_1} Div$$

In some problems we will also want to compute the derivative w.r.t. the input

The Backward Pass

- Set $\mathbf{y}_N = Y, \mathbf{y}_0 = \mathbf{x}$
- Initialize: Compute $\nabla_{\mathbf{y}_N} Div = \nabla_Y Div$
- For layer $k = N$ downto 1:
 - Compute $J_{\mathbf{y}_k}(\mathbf{z}_k)$
 - Will require intermediate values computed in the forward pass
 - Backward recursion step:
$$\nabla_{\mathbf{z}_k} Div = \nabla_{\mathbf{y}_k} Div J_{\mathbf{y}_k}(\mathbf{z}_k)$$
$$\nabla_{\mathbf{y}_{k-1}} Div = \nabla_{\mathbf{z}_k} Div \mathbf{W}_k$$
 - Gradient computation:
$$\nabla_{\mathbf{W}_k} Div = \mathbf{y}_{k-1} \nabla_{\mathbf{z}_k} Div$$
$$\nabla_{\mathbf{b}_k} Div = \nabla_{\mathbf{z}_k} Div$$

The Backward Pass

- Set $\mathbf{y}_N = Y, \mathbf{y}_0 = \mathbf{x}$
- Initialize: Compute $\nabla_{\mathbf{y}_N} Div = \nabla_Y Div$
- For layer $k = N$ downto 1:
 - Compute $J_{\mathbf{y}_k}(\mathbf{z}_k)$
 - Will require intermediate values computed in the forward pass
 - Backward recursion step: Note analogy to forward pass
$$\nabla_{\mathbf{z}_k} Div = \nabla_{\mathbf{y}_k} Div J_{\mathbf{y}_k}(\mathbf{z}_k)$$
$$\nabla_{\mathbf{y}_{k-1}} Div = \nabla_{\mathbf{z}_k} Div \mathbf{W}_k$$
 - Gradient computation:
$$\nabla_{\mathbf{W}_k} Div = \mathbf{y}_{k-1} \nabla_{\mathbf{z}_k} Div$$
$$\nabla_{\mathbf{b}_k} Div = \nabla_{\mathbf{z}_k} Div$$

For comparison: The Forward Pass

- Set $\mathbf{y}_0 = \mathbf{x}$
- For layer $k = 1$ to N :
 - Forward recursion step:

$$\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k$$

$$\mathbf{y}_k = \mathbf{f}_k(\mathbf{z}_k)$$

- Output:

$$\mathbf{Y} = \mathbf{y}_N$$

Neural network training algorithm

- Initialize all weights and biases ($\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \dots, \mathbf{W}_N, \mathbf{b}_N$)

- Do:

- $Loss = 0$

- For all k , initialize $\nabla_{\mathbf{W}_k} Loss = 0, \nabla_{\mathbf{b}_k} Loss = 0$

- For all $t = 1:T$ # Loop through training instances

- Forward pass : Compute

- Output $\mathbf{Y}(\mathbf{X}_t)$

- Divergence $Div(\mathbf{Y}_t, \mathbf{d}_t)$

- $Loss += Div(\mathbf{Y}_t, \mathbf{d}_t)$

- Backward pass: For all k compute:

- $\nabla_{\mathbf{y}_k} Div = \nabla_{\mathbf{z}_{k+1}} Div \mathbf{W}_{k+1}$

- $\nabla_{\mathbf{z}_k} Div = \nabla_{\mathbf{y}_k} Div J_{\mathbf{y}_k}(\mathbf{z}_k)$

- $\nabla_{\mathbf{W}_k} Div(\mathbf{Y}_t, \mathbf{d}_t) = \mathbf{y}_{k-1} \nabla_{\mathbf{z}_k} Div; \nabla_{\mathbf{b}_k} Div(\mathbf{Y}_t, \mathbf{d}_t) = \nabla_{\mathbf{z}_k} Div$

- $\nabla_{\mathbf{W}_k} Loss += \nabla_{\mathbf{W}_k} Div(\mathbf{Y}_t, \mathbf{d}_t); \nabla_{\mathbf{b}_k} Loss += \nabla_{\mathbf{b}_k} Div(\mathbf{Y}_t, \mathbf{d}_t)$

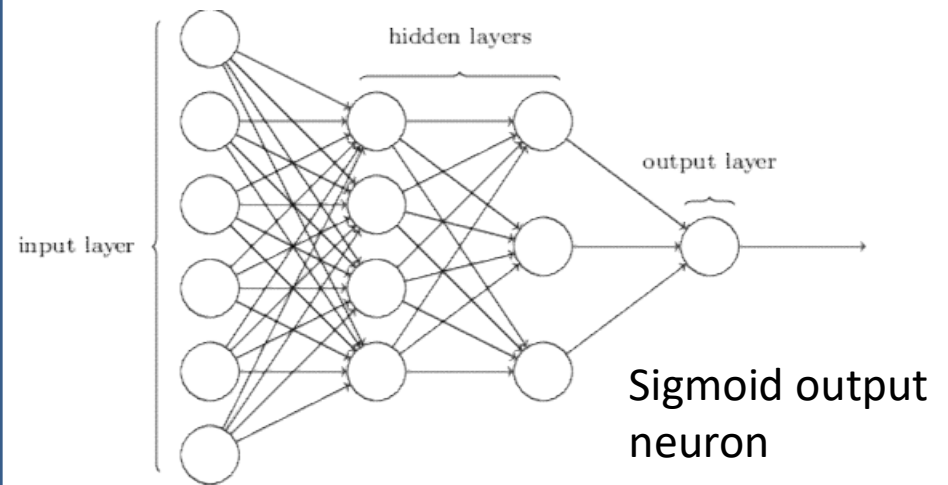
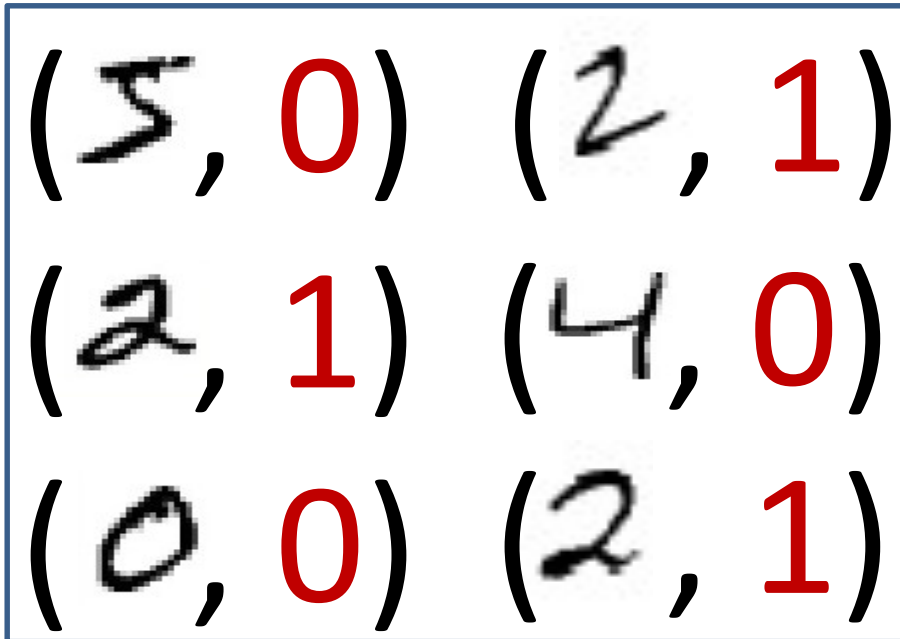
- For all k , update:

$$\mathbf{W}_k = \mathbf{W}_k - \frac{\eta}{T} (\nabla_{\mathbf{W}_k} Loss)^T; \quad \mathbf{b}_k = \mathbf{b}_k - \frac{\eta}{T} (\nabla_{\mathbf{b}_k} Loss)^T$$

- Until $Loss$ has converged

Setting up for digit recognition

Training data

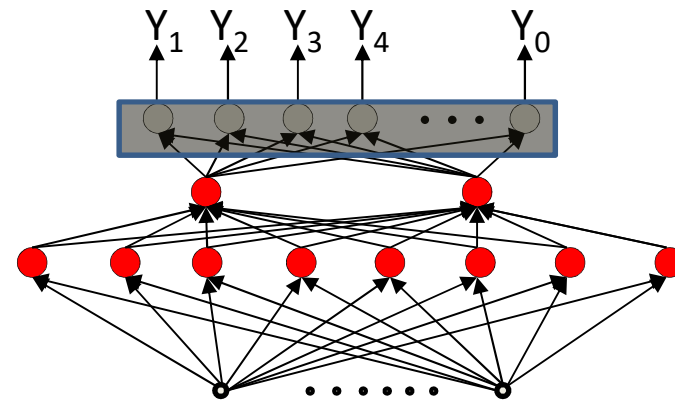


- Simple Problem: Recognizing “2” or “not 2”
- Single output with sigmoid activation
 - $Y \in (0,1)$
 - d is either 0 or 1
- Use KL divergence
- Backpropagation to learn network parameters

Recognizing the digit

Training data

(5, 5)	(2, 2)
(2, 2)	(4, 4)
(0, 0)	(2, 2)



- More complex problem: Recognizing digit
- Network with 10 (or 11) outputs
 - First ten outputs correspond to the ten digits
 - Optional 11th is for none of the above
- Softmax output layer:
 - Ideal output: One of the outputs goes to 1, the others go to 0
- Backpropagation with KL divergence to learn network

Issues

- Convergence: How well does it learn
 - And how can we improve it
- How well will it generalize (outside training data)
- What does the output really mean?
- *Etc..*