Forward and Backward Propagation

CSE 849 Deep Learning

Spring 2025

Zijun Cui

The slides have been revised based on CMU’s 11-785 Introduction to Deep Learning course by Bhiksha Raj

Problem Statement

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$
- Minimize the following function

$$Loss(W) = \frac{1}{N} \sum_i div(f(X_i; W), d_i)$$

w.r.t W

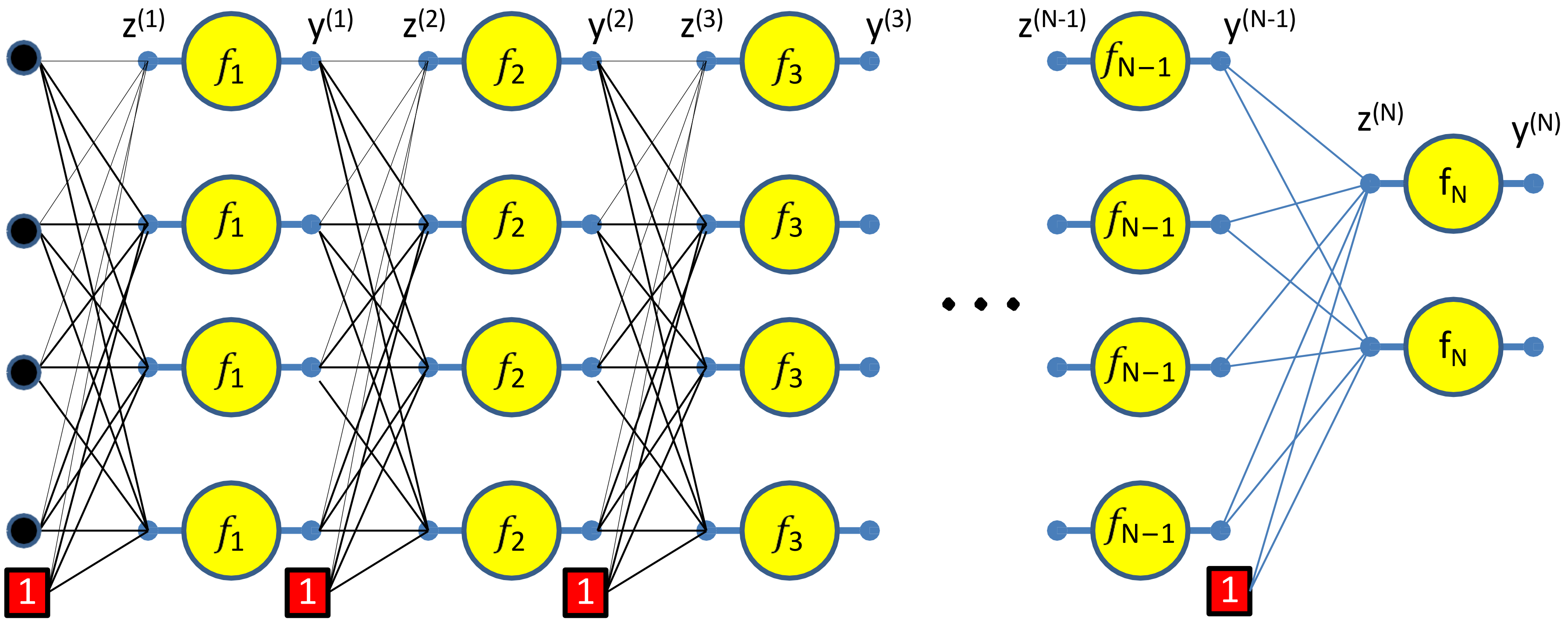
- This is problem of function minimization
  – An instance of optimization

Problem Setup:
Things to define

✓ What is f() and what are its parameters W?
✓ What are these input-output pairs?
✓ What is the divergence div()?

We are ready!
Next, we will start forward pass and backward update

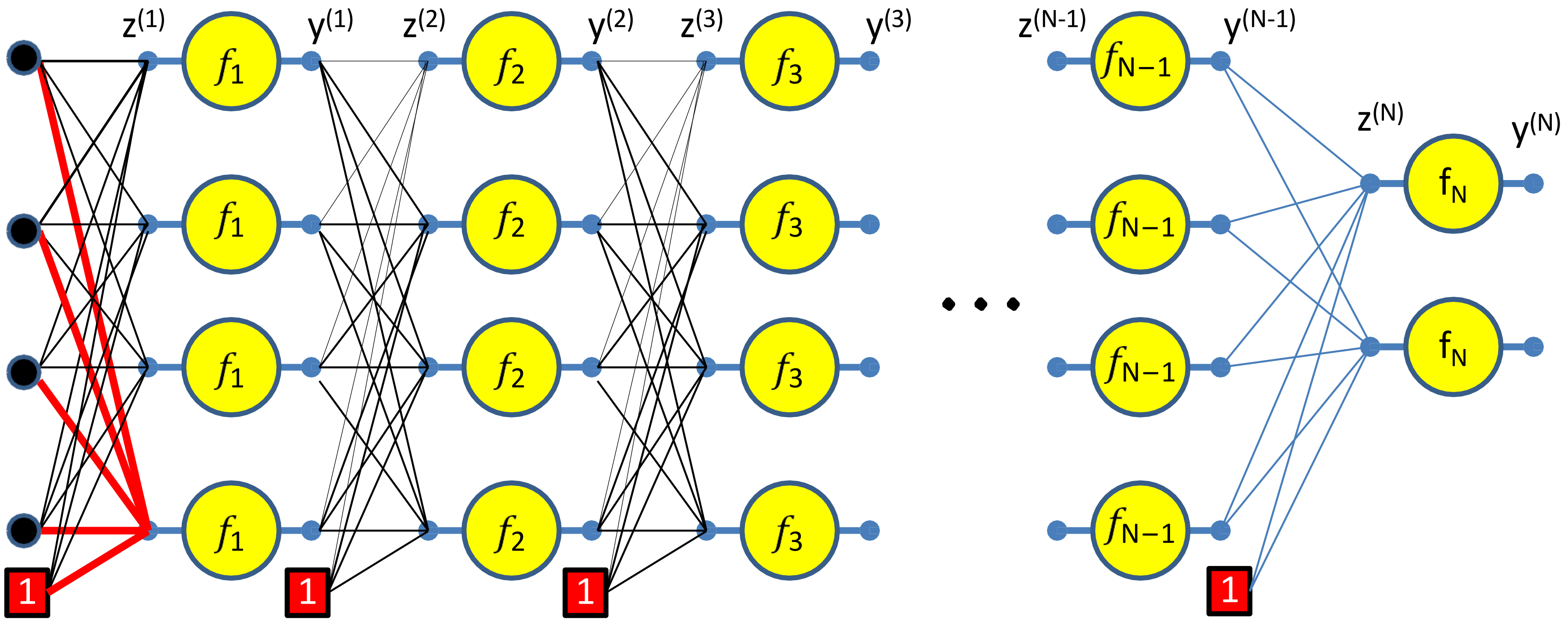
Forward pass



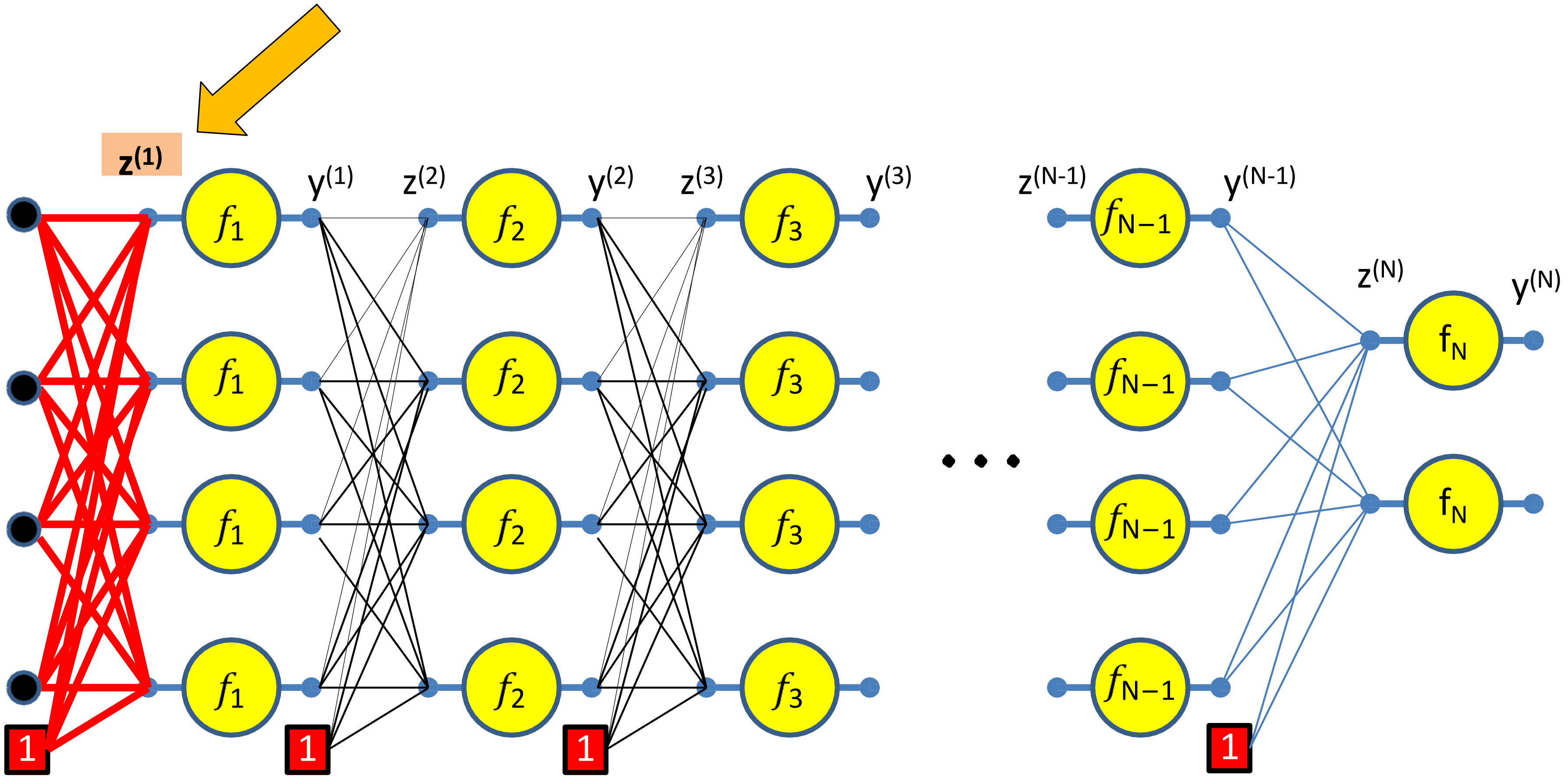
Setting $y_i^{(0)} = x_i$ for notational convenience

Assuming $w_{0j}^{(k)} = b_j^{(k)}$ and $y_0^{(k)} = 1$ -- assuming the bias is a weight and extending the output of every layer by a constant 1, to account for the biases

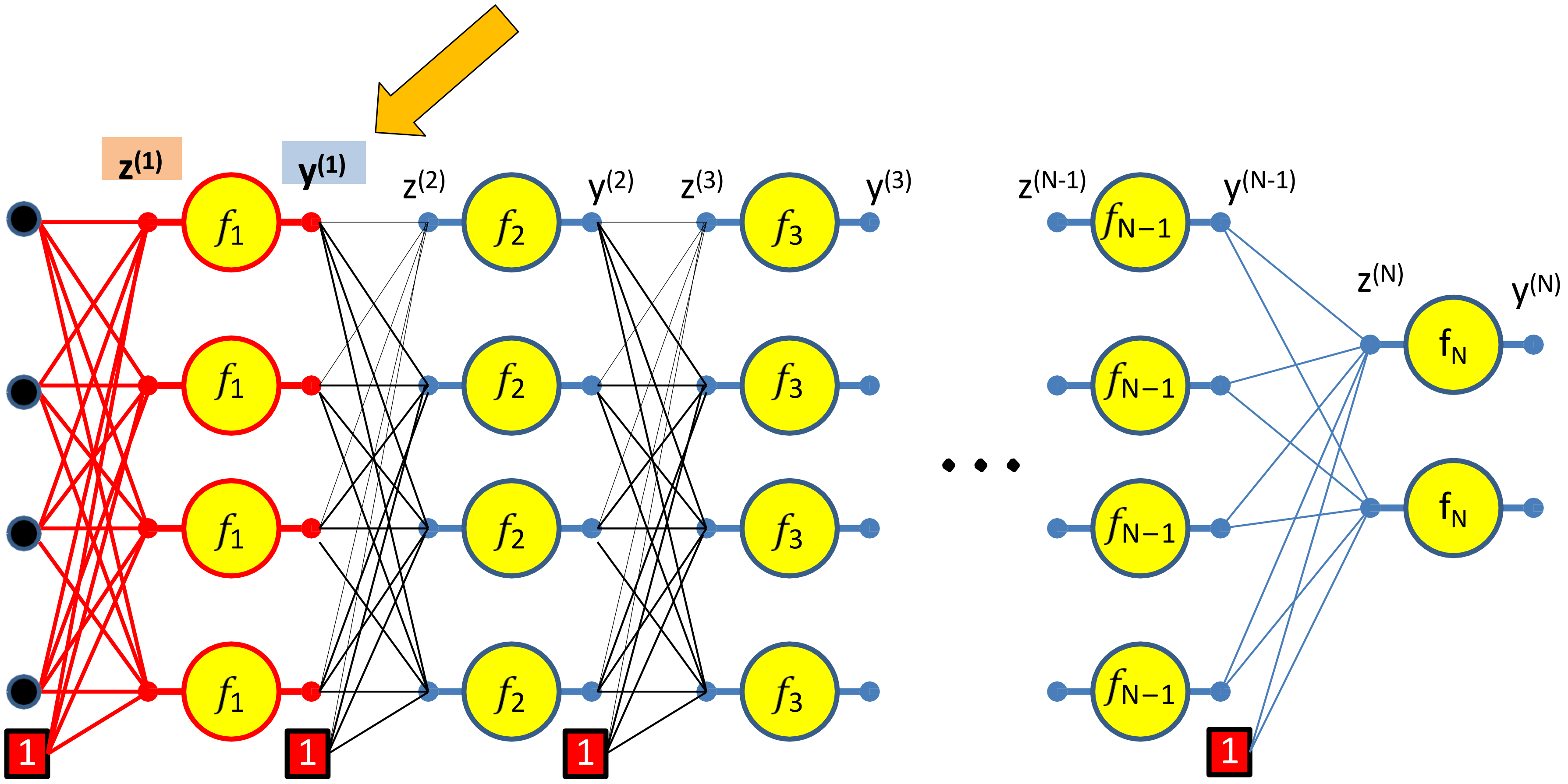
Forward pass



$$z_1^{(1)} = \sum_i w_{i1}^{(1)} y_i^{(0)}$$

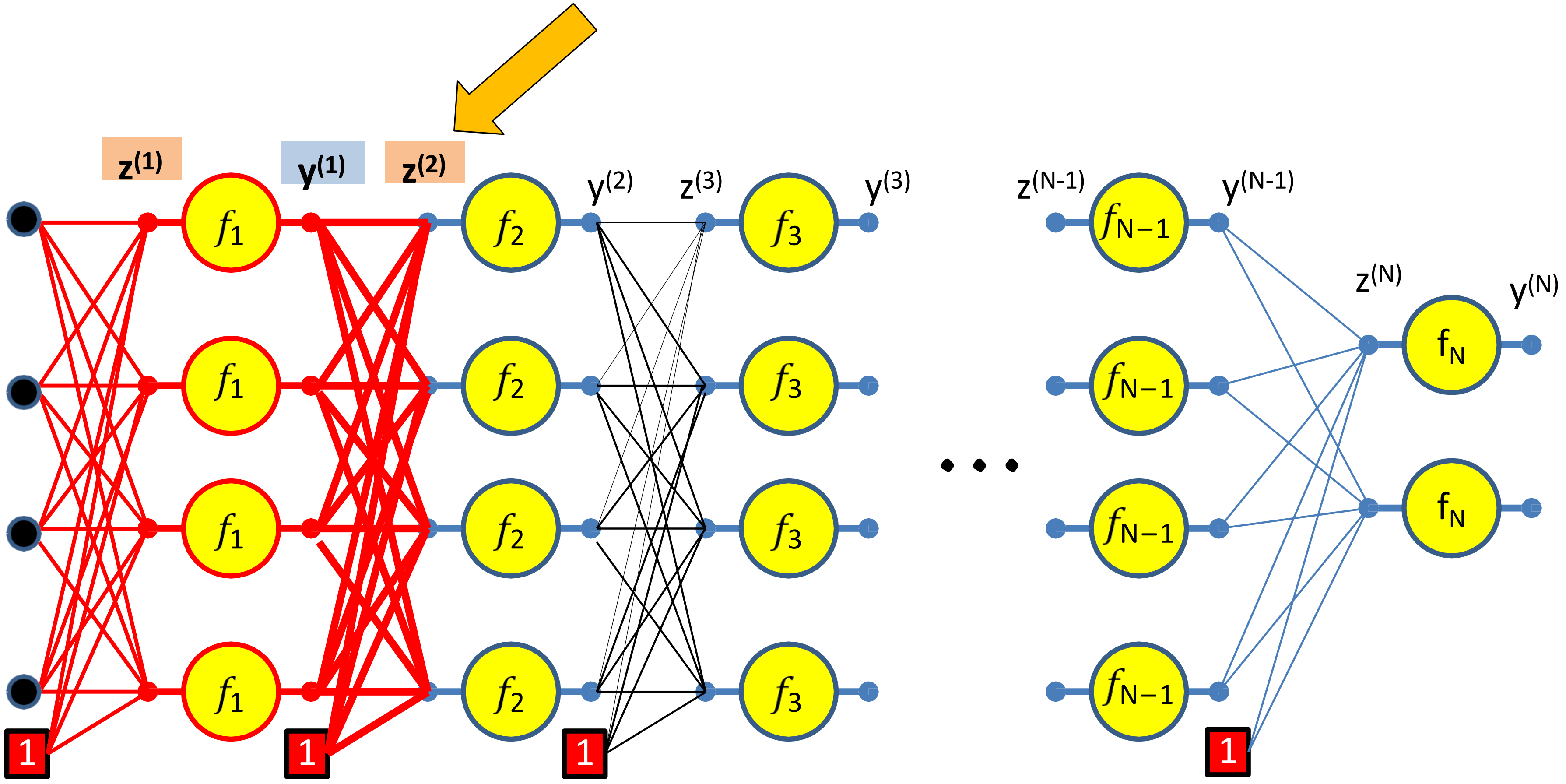


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$



$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

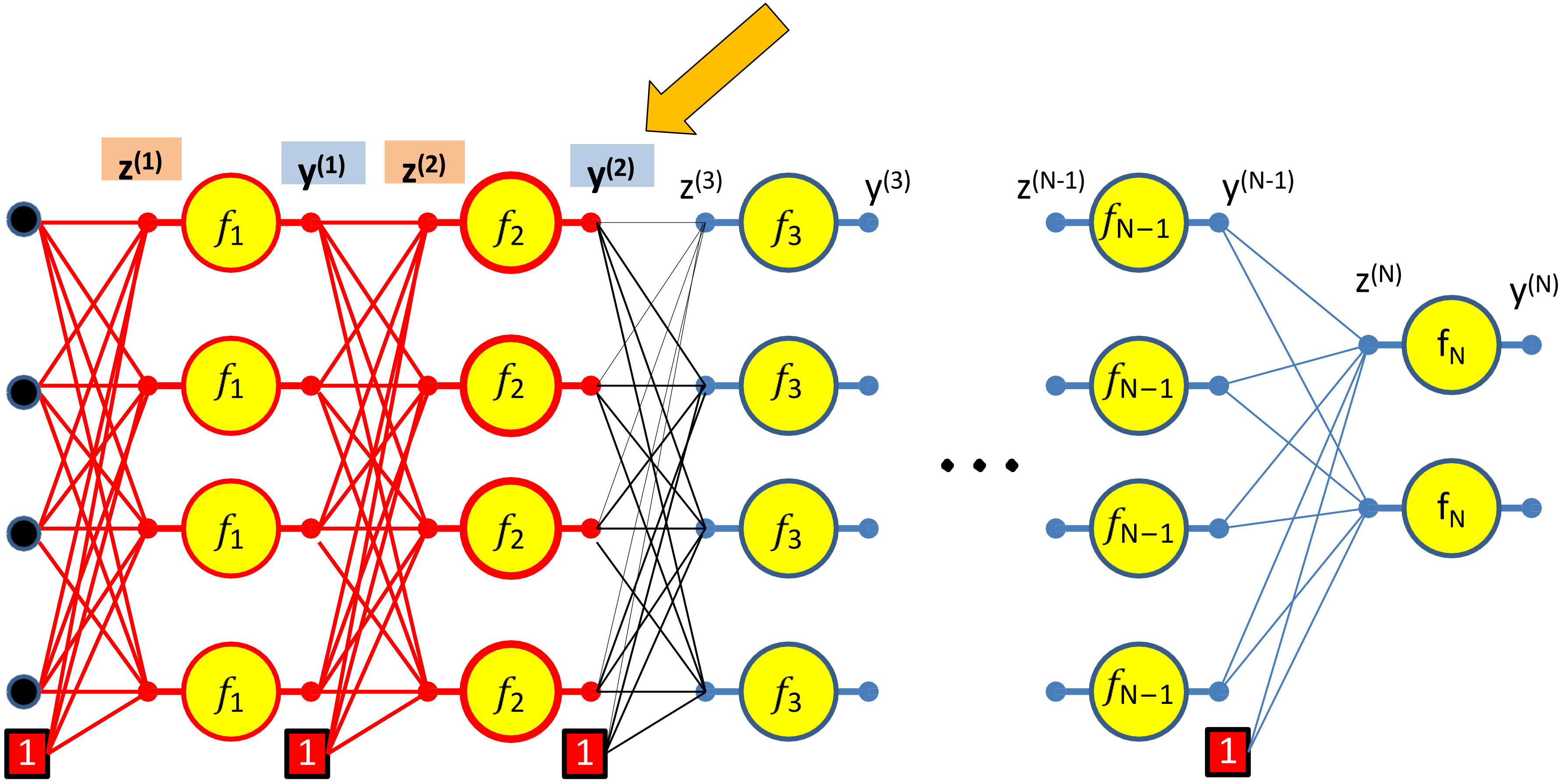
$$\boxed{y_j^{(1)} = f_1\left(z_j^{(1)}\right)}$$



$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

$$y_j^{(1)} = f_1\left(z_j^{(1)}\right)$$

$$\boxed{z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)}}$$

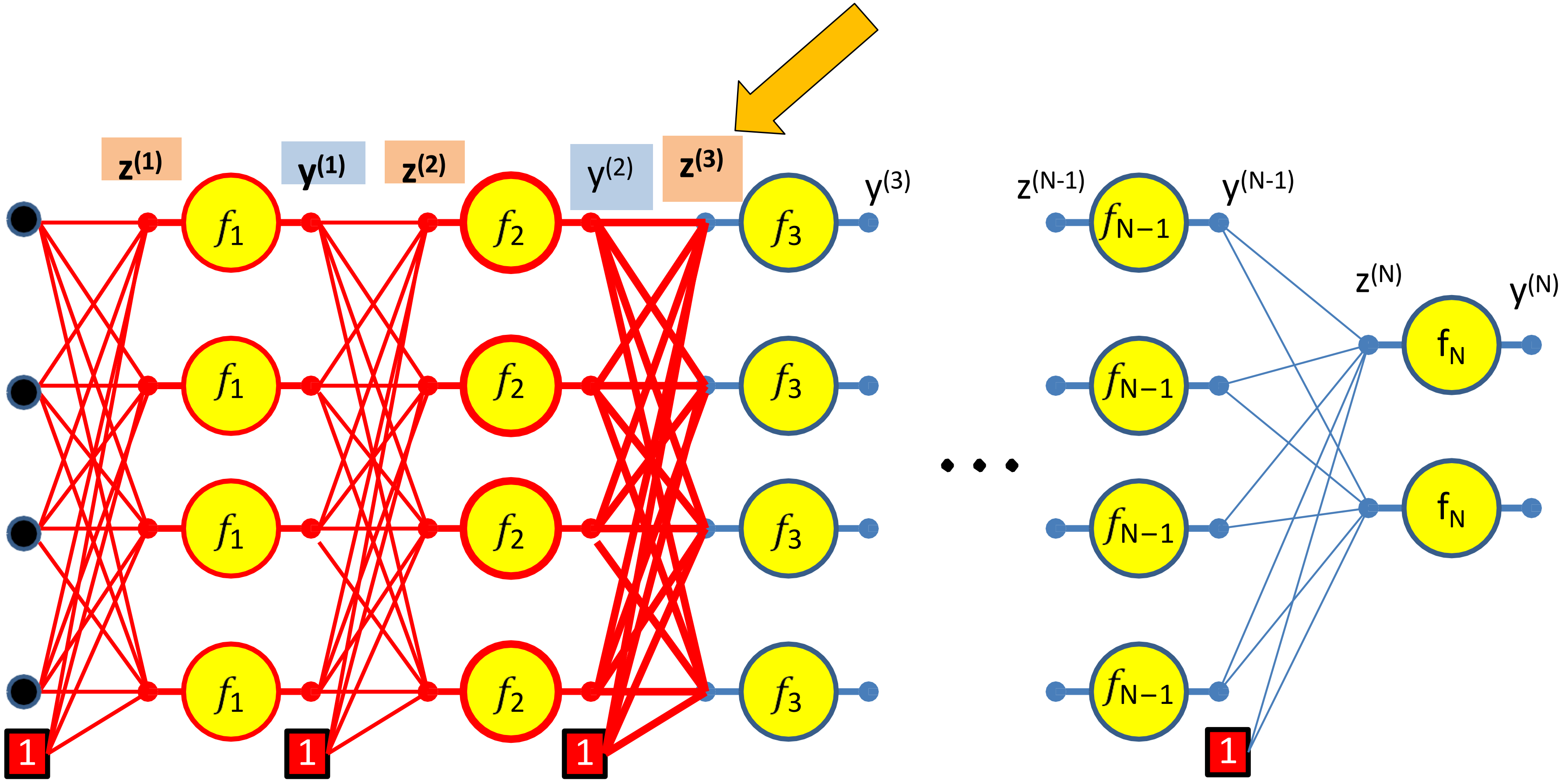


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

$$y_j^{(1)} = f_1\left(z_j^{(1)}\right)$$

$$z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)}$$

$$\boxed{y_j^{(2)} = f_2\left(z_j^{(2)}\right)}$$



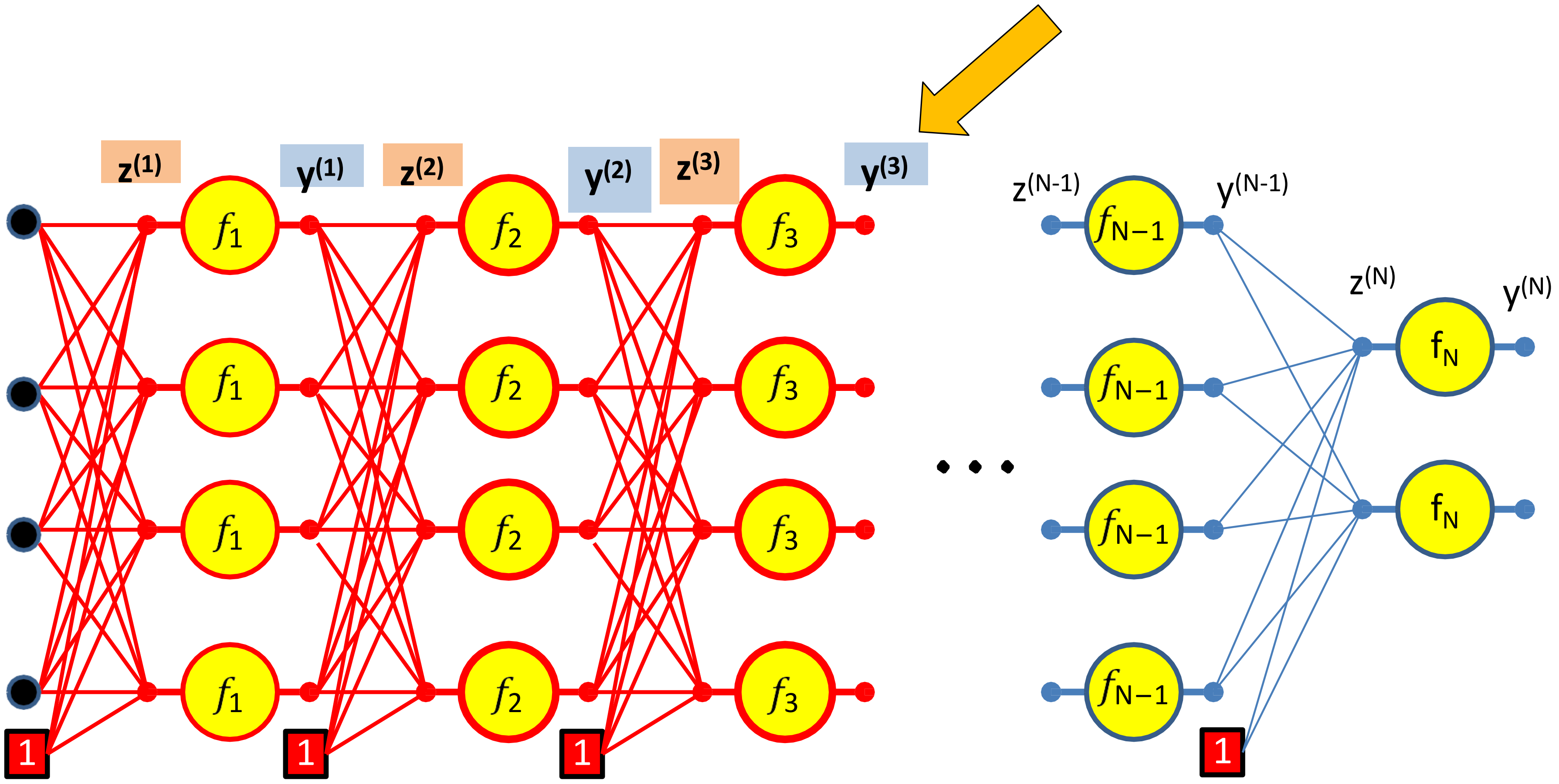
$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

$$y_j^{(1)} = f_1\left(z_j^{(1)}\right)$$

$$z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)}$$

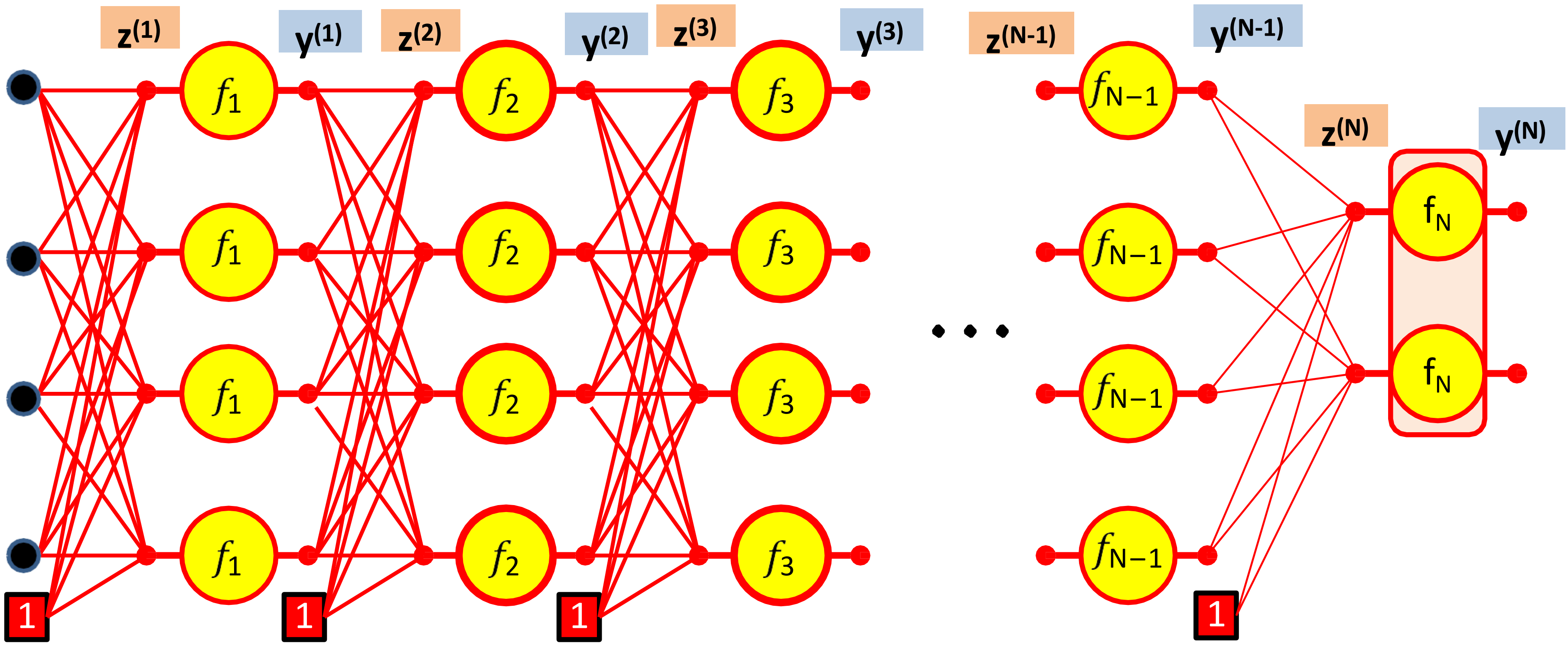
$$y_j^{(2)} = f_2\left(z_j^{(2)}\right)$$

$$\boxed{z_j^{(3)} = \sum_i w_{ij}^{(3)} y_i^{(2)}}$$



$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)} \qquad y_j^{(1)} = f_1\left(z_j^{(1)}\right) \qquad z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)} \qquad y_j^{(2)} = f_2\left(z_j^{(2)}\right)$$

$$z_j^{(3)} = \sum_i w_{ij}^{(3)} y_i^{(2)} \qquad \boxed{y_j^{(3)} = f_3\left(z_j^{(3)}\right)} \qquad \cdots$$



$$y_j^{(N-1)} = f_{N-1}\left(z_j^{(N-1)}\right)$$

$$z_j^{(N)} = \sum_i w_{ij}^{(N)} y_i^{(N-1)}$$

$$\boldsymbol{y}^{(N)} = f_N\left(\boldsymbol{z}^{(N)}\right)$$

Forward "Pass"

- Input: D dimensional vector $\mathbf{x} = [x_j,\ j = 1 \dots D]$
- Set:
  - $D_0 = D,$ is the width of the 0th (input) layer
  - $y_j^{(0)} = x_j,\ j = 1 \dots D;$ $\quad y_0^{(k=1\dots N)} = x_0 = 1$
- For layer $k = 1 \dots N$
  - For $j = 1 \dots D_k$ $\quad D_k$ is the size of the kth layer
    - $z_j^{(k)} = \sum_{i=0}^{D_{k-1}} w_{i,j}^{(k)} y_i^{(k-1)}$
    - $y_j^{(k)} = f_k\left(z_j^{(k)}\right)$
- Output:
  - $Y = y_j^{(N)}, j = 1..D_N$

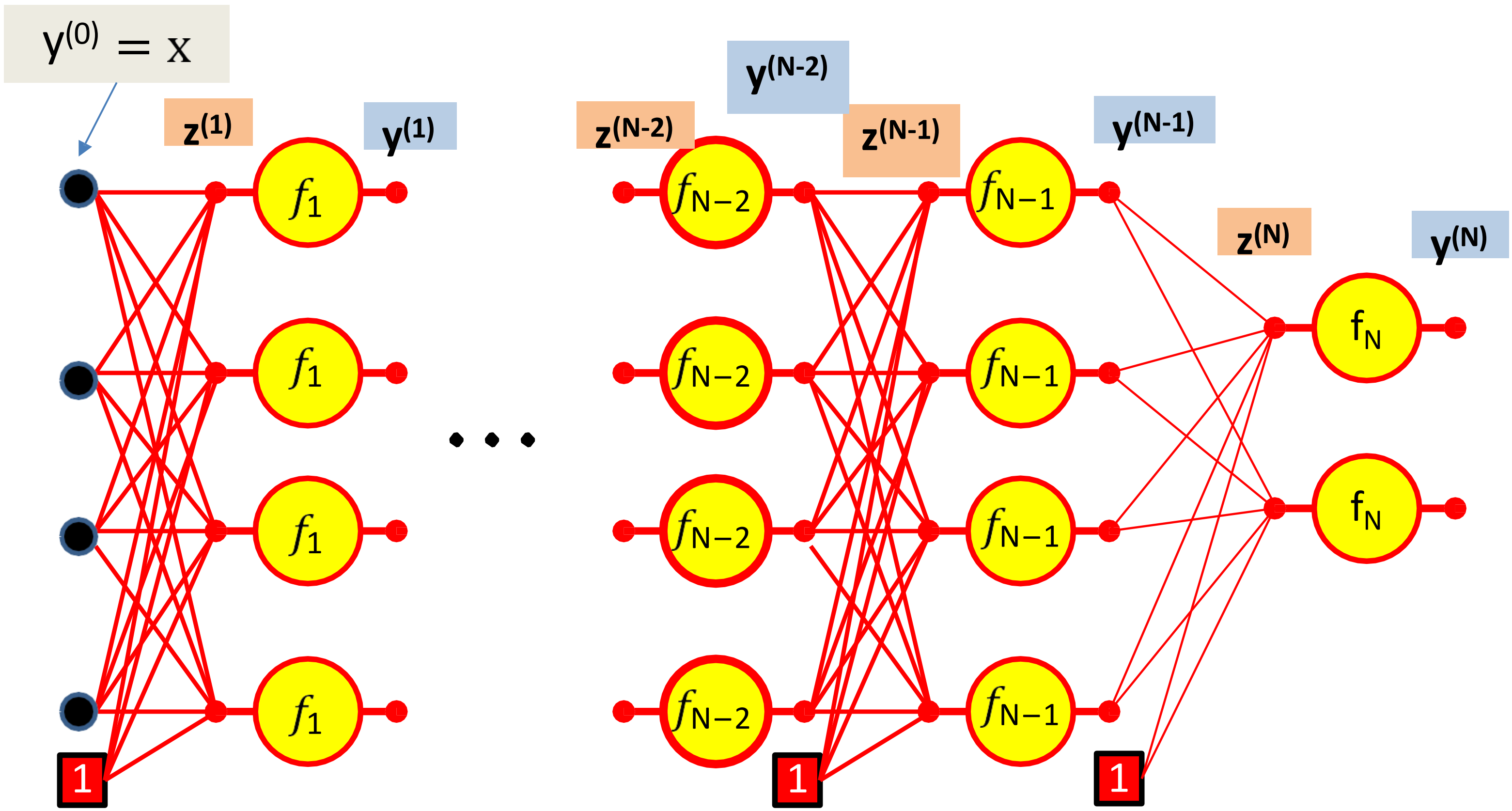
Training Neural Nets through Gradient Descent

Total training Loss:

$$Loss = \frac{1}{T}\sum_t Div(Y_t, d_t)$$

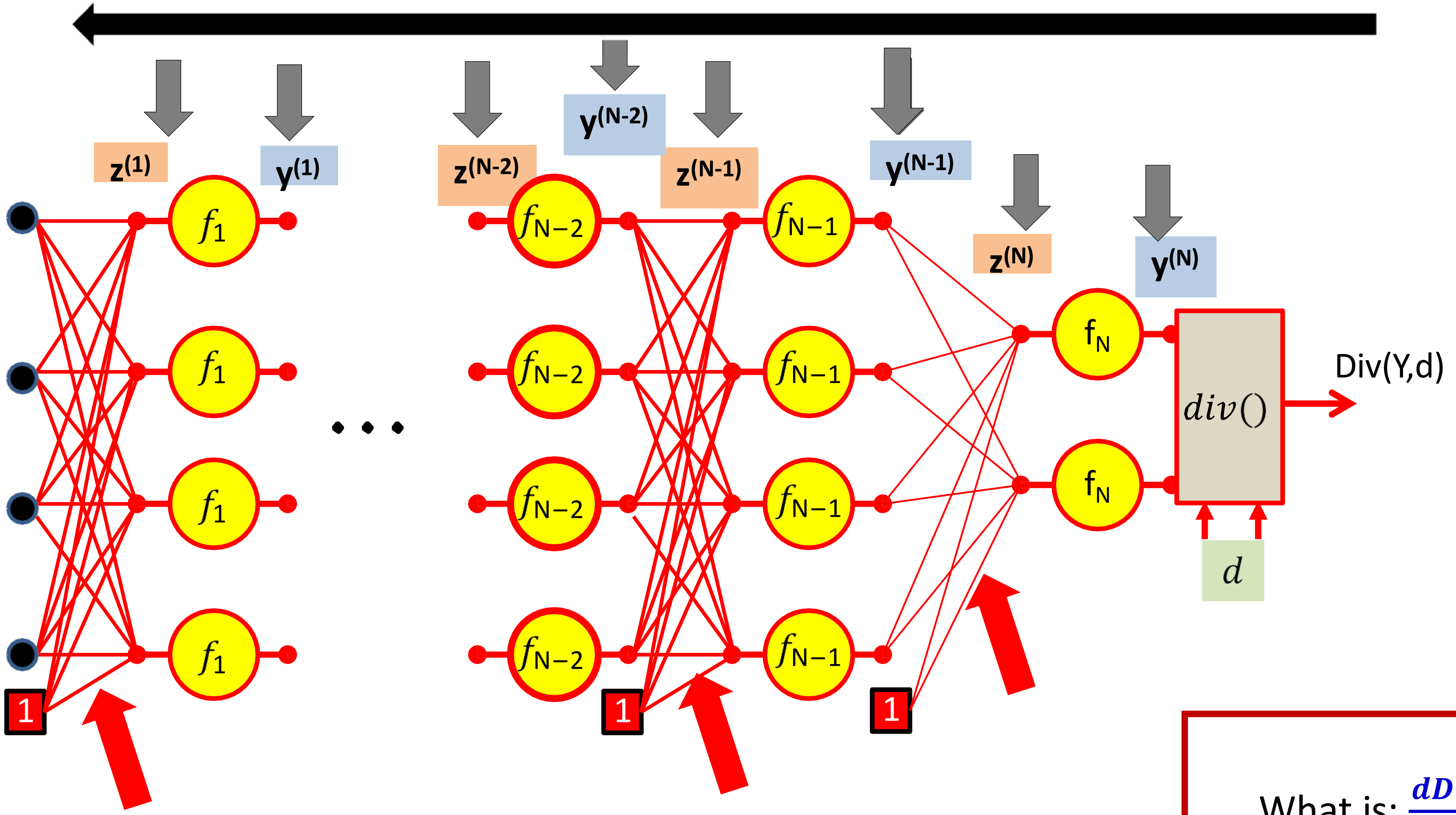
- Gradient descent algorithm:
- Initialize all weights and biases $\{w_{ij}^{(k)}\}$
  - Using the extended notation: the bias is also a weight
- Do:
  - For every layer k for all i, j, update:
    - $w_{i,j}^{(k)} = w_{i,j}^{(k)} - \eta \frac{dLoss}{dw_{i,j}^{(k)}}$
- Until $Loss$ has converged

Computing derivatives



We have computed all these intermediate values in the forward computation

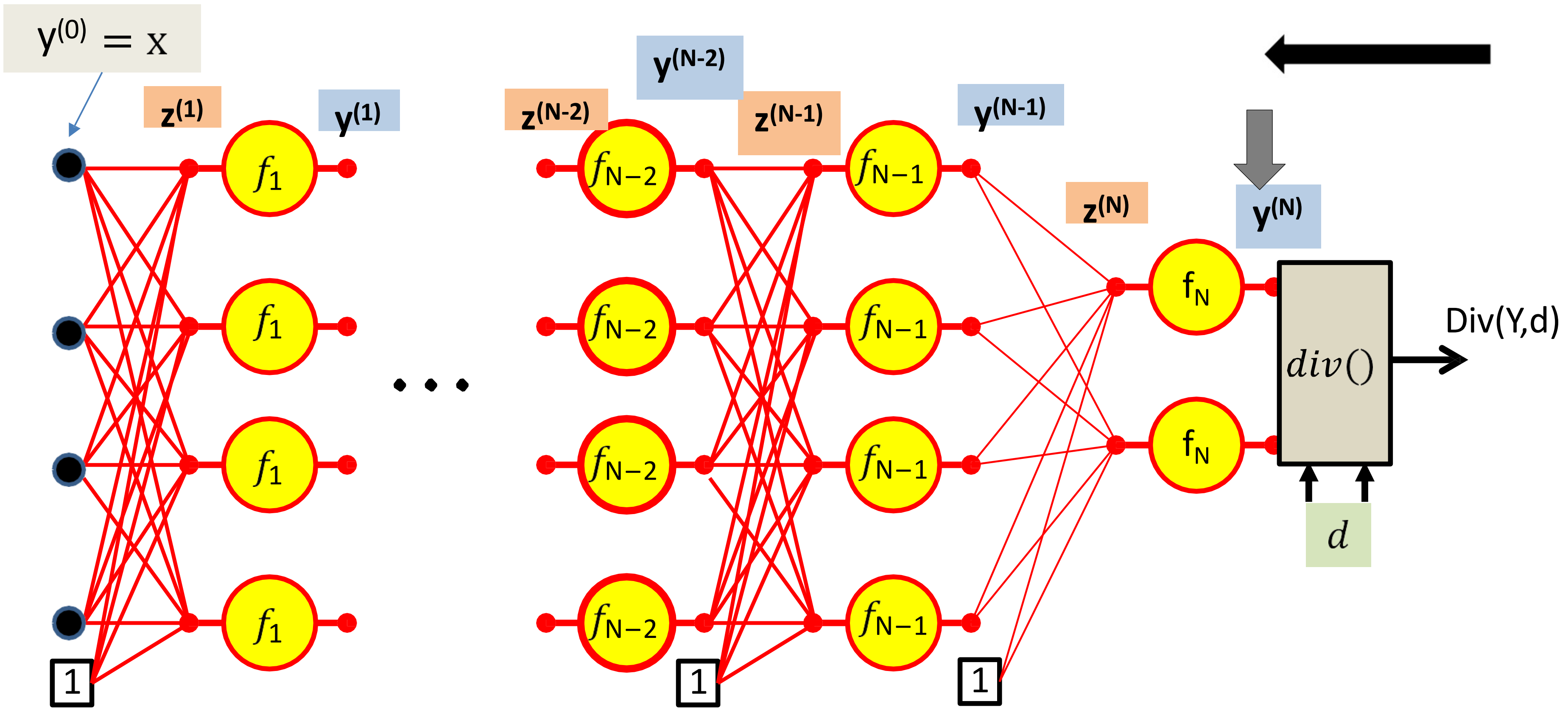
We must remember them – we will need them to compute the derivatives



What is: $\frac{dDiv(Y,d)}{dw_{i,j}^{(k)}}$

Today's Topic

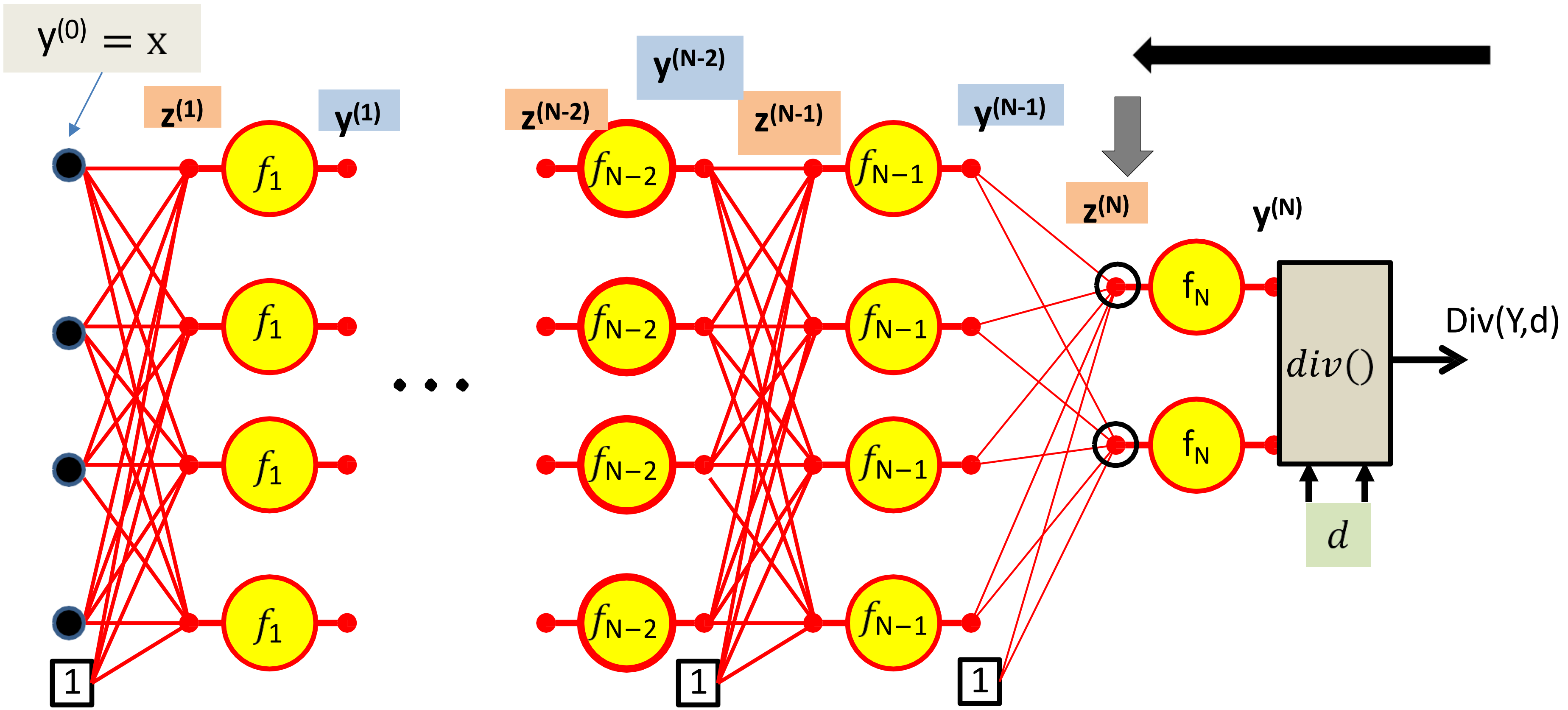
Computing derivatives



The derivative w.r.t the actual output of the final layer of the network is simply the derivative w.r.t to the output of the network

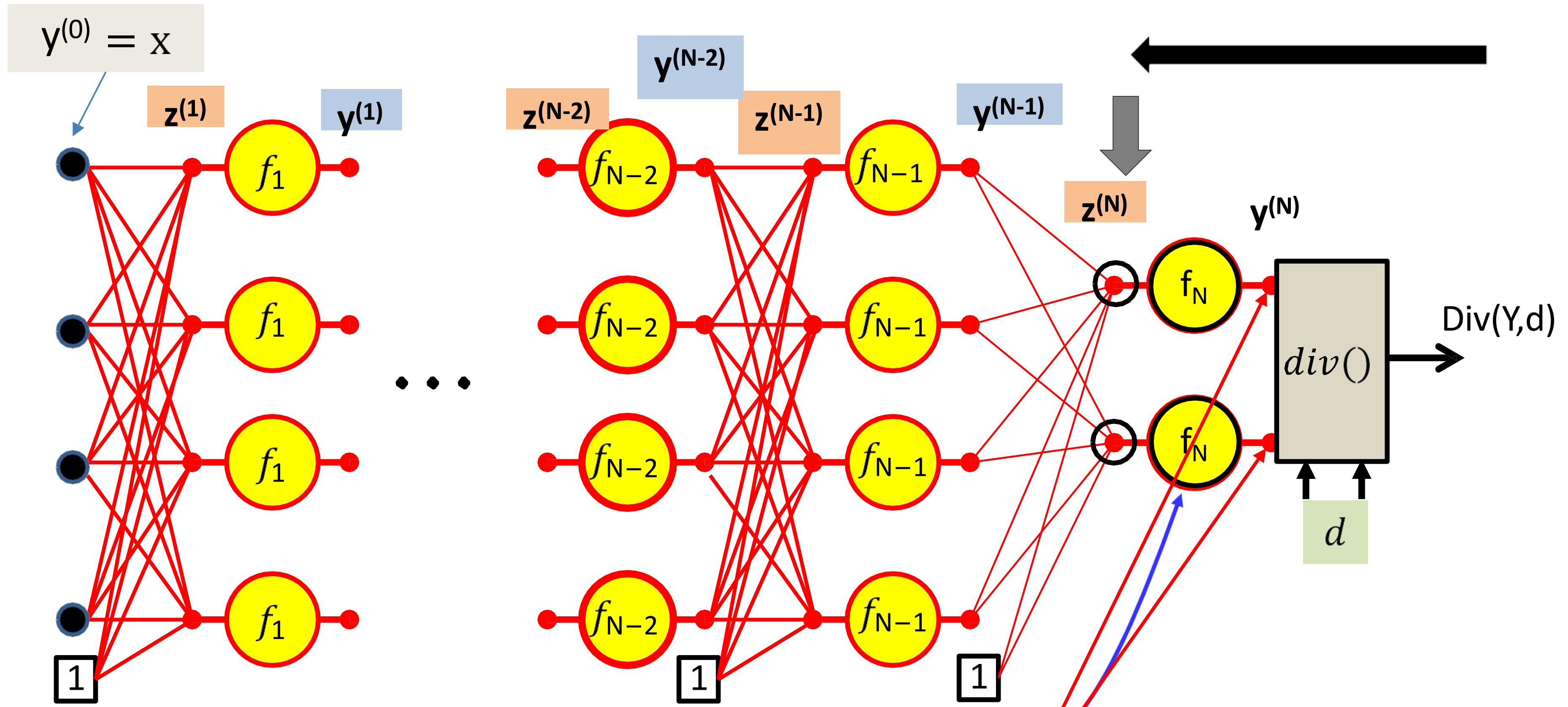
$$\frac{\partial Div(Y,d)}{\partial {y_i}^{(N)}} = \frac{\partial Div(Y,d)}{\partial y_i}$$

Computing derivatives



$$\frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial Div}{\partial y_i^{(N)}}$$

Computing derivatives



$$\frac{\partial Div}{\partial z_i^{(N)}} = f_N'\left(z_i^{(N)}\right)\frac{\partial Div}{\partial y_i^{(N)}} \Longleftarrow \frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}}\frac{\partial Div}{\partial y_i^{(N)}}$$

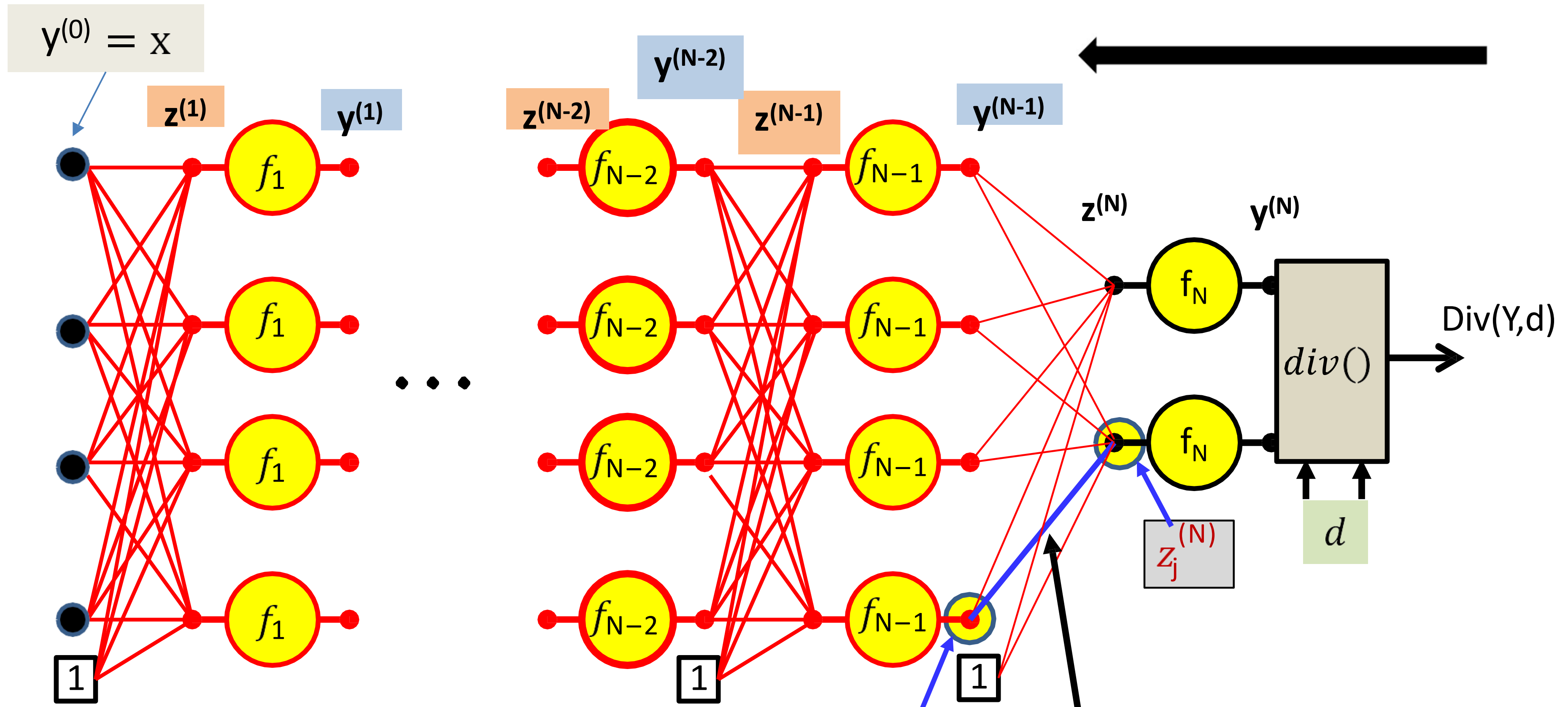
$$= f_N'(z_i^{(N)})$$

Already computed in previous step

Derivative of activation function

And $z_i^{(N)}$ has been computed in forward pass

Computing derivatives



$$\frac{\partial Div}{\partial w_{ij}^{(N)}} = y_i^{(N-1)} \frac{\partial Div}{\partial z_j^{(N)}} \Leftarrow \frac{\partial Div}{\partial w_{ij}^{(N)}} = \frac{\partial z_j^{(N)}}{\partial w_{ij}^{(N)}} \frac{\partial Div}{\partial z_j^{(N)}}$$

Just computed $\boxed{y_i^{(N-1)}}$

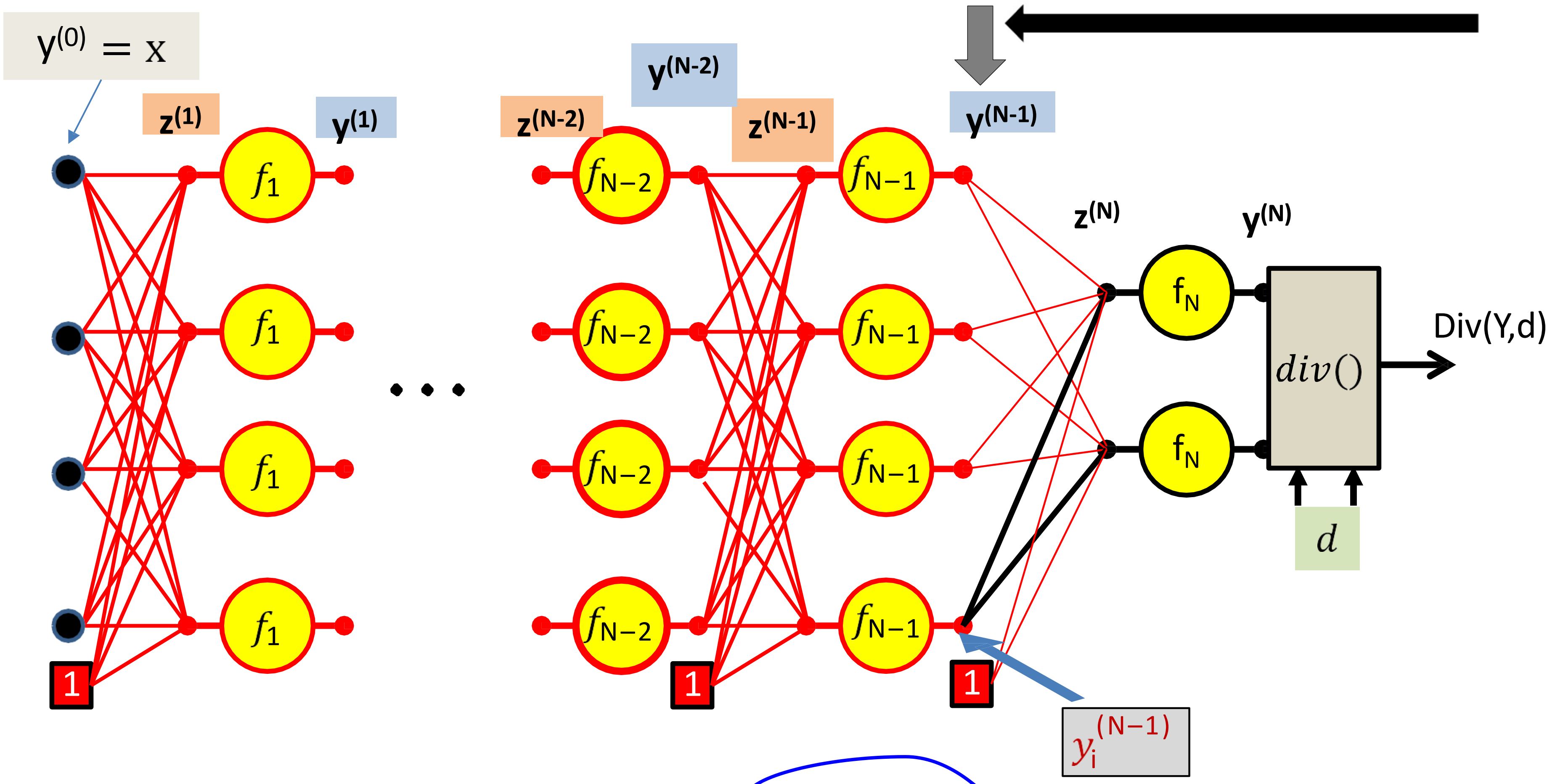
Computed in forward pass

Because

$$z_j^{(N)} = w_{ij}^{(N)} y_i^{(N-1)} + \text{other terms}$$

For the bias term $y_0^{(N-1)} = 1$

Computing derivatives



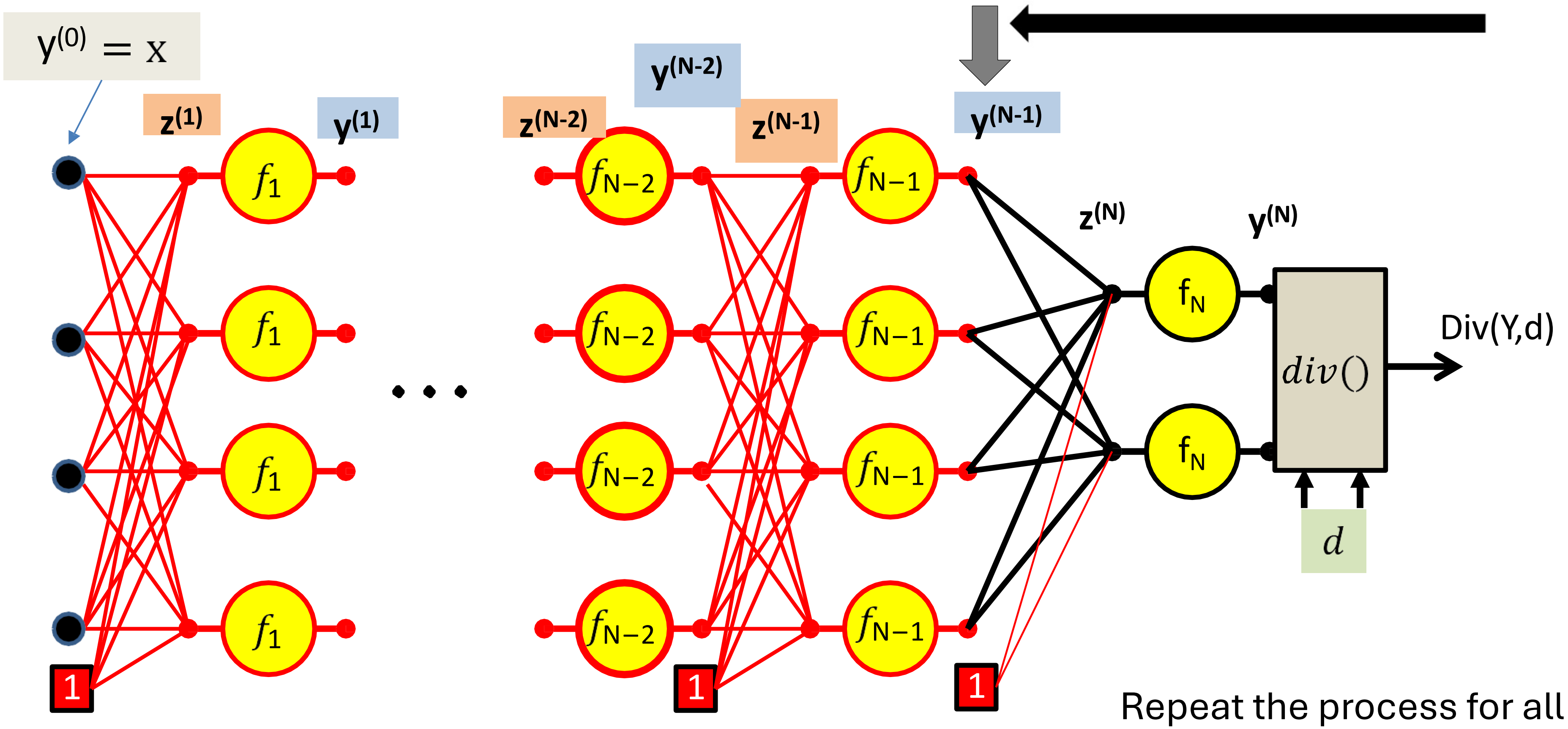
$$\frac{\partial Div}{\partial y_i^{(N-1)}} = \sum_j w_{ij}^{(N)} \frac{\partial Div}{\partial z_j^{(N)}} \quad \Longleftarrow \quad \frac{\partial Div}{\partial y_i^{(N-1)}} = \sum_j \frac{\partial z_j^{(N)}}{\partial y_i^{(N-1)}} \frac{\partial Div}{\partial z_j^{(N)}}$$

Just computed $\boxed{w_{ij}^{(N)}}$

Because

$$z_j^{(N)} = w_{ij}^{(N)} y_i^{(N-1)} + \text{other terms}$$

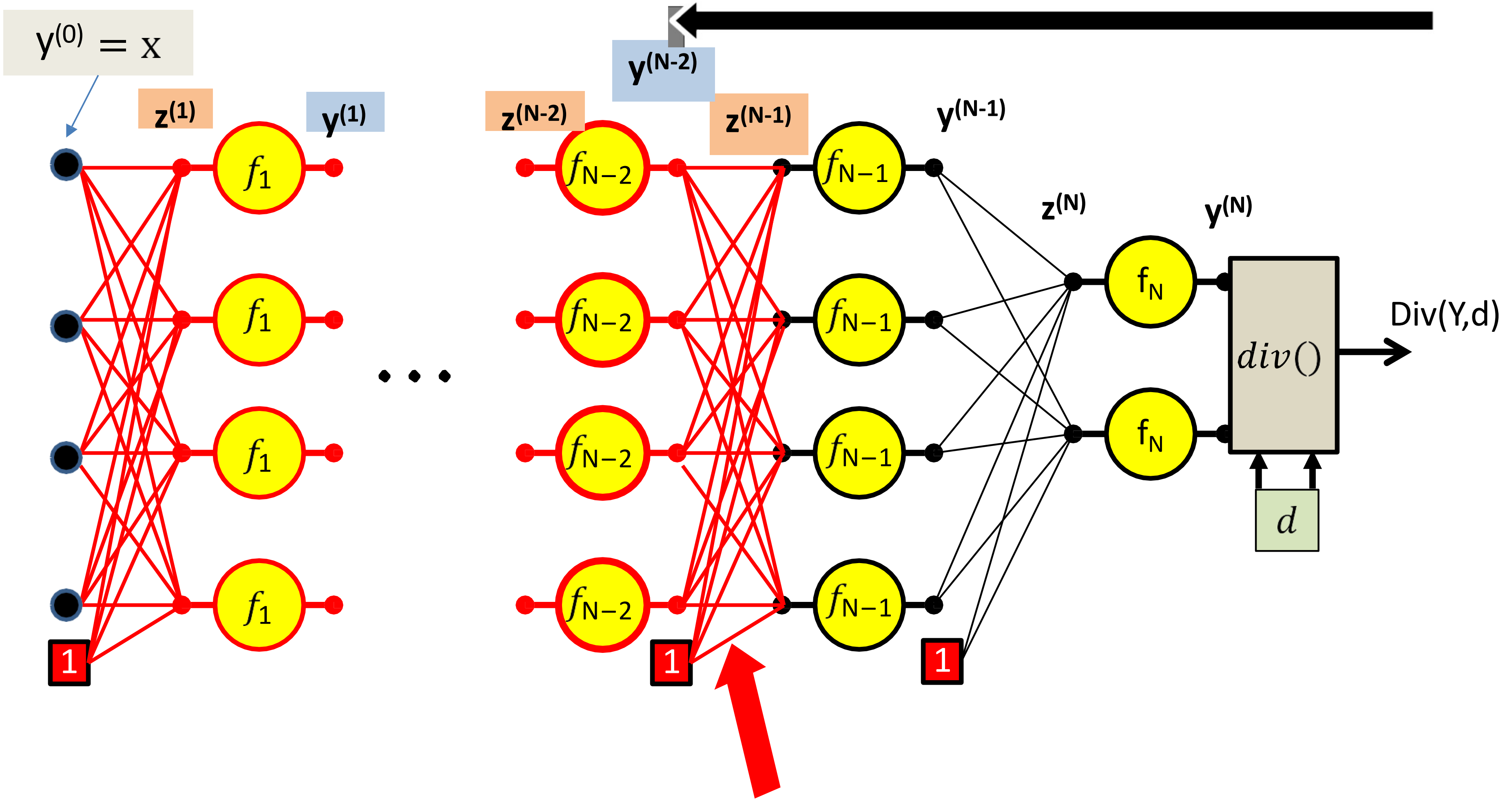
Computing derivatives



Repeat the process for all the y in (N-1) layer

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

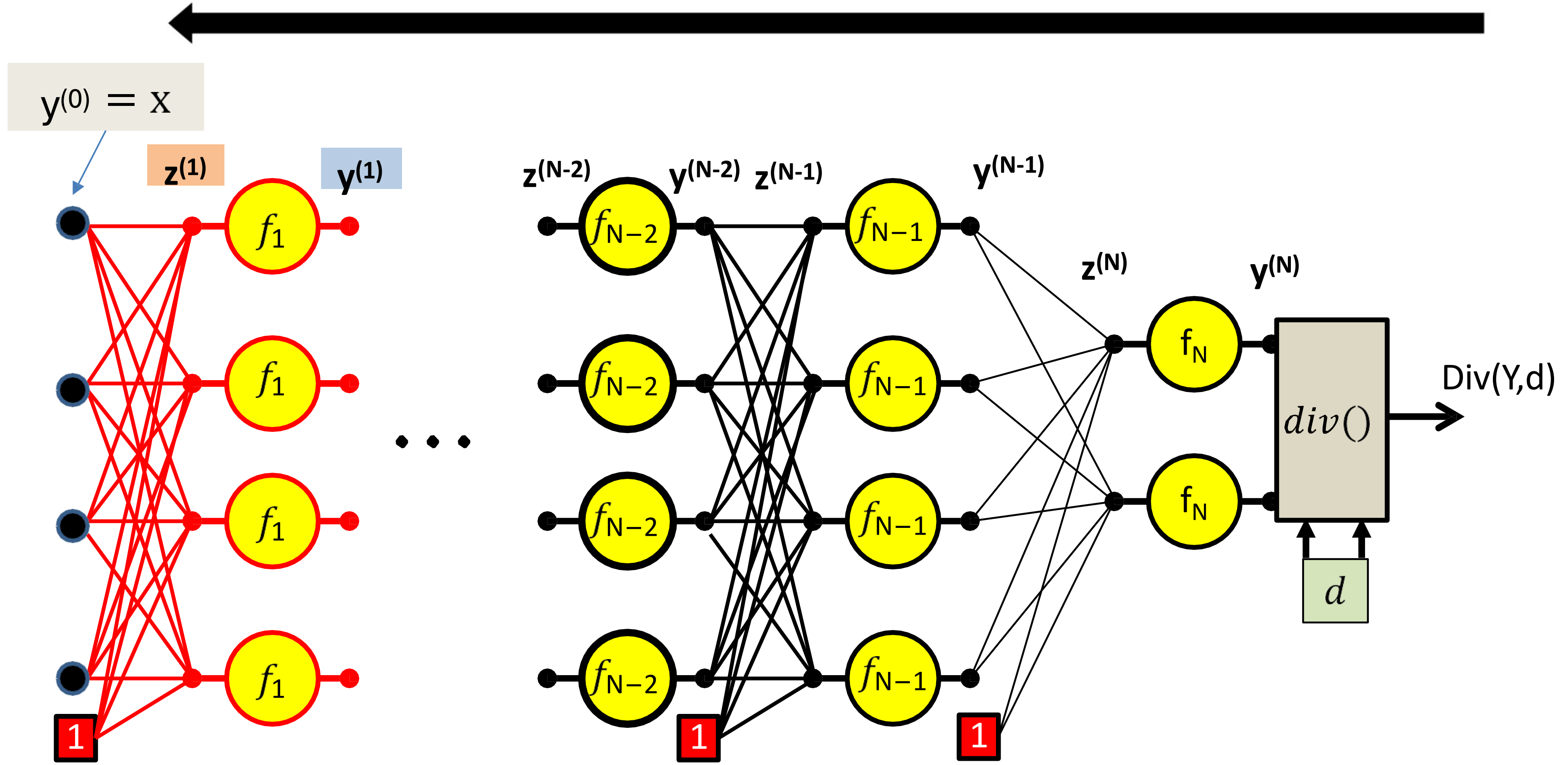
We then can continue the calculation as before but in layer N-1



$$\frac{\partial Div}{\partial y_i^{(N-2)}} = \sum_j w_{ij}^{(N-1)} \frac{\partial Div}{\partial z_j^{(N-1)}}$$

$$\frac{\partial Div}{\partial w_{ij}^{(N-1)}} = y_i^{(N-2)} \frac{\partial Div}{\partial z_j^{(N-1)}} \longleftarrow \frac{\partial Div}{\partial z_i^{(N-1)}} = f_{N-1}'\left(z_i^{(N-1)}\right) \frac{\partial Div}{\partial y_i^{(N-1)}}$$

Finally reach the input layer



$$\frac{\partial Div}{\partial y_1^{(1)}} = \sum_j w_{ij}^{(2)} \frac{\partial Div}{\partial z_j^{(2)}}$$

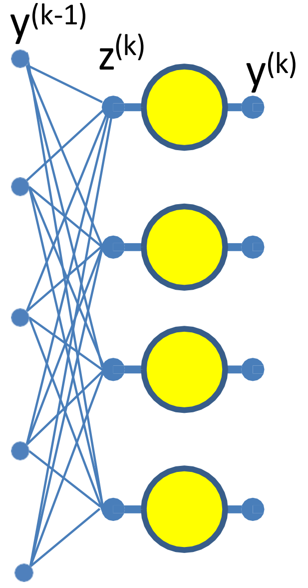
$$\frac{\partial Div}{\partial z_i^{(1)}} = f_1'\left(z_i^{(1)}\right) \frac{\partial Div}{\partial y_i^{(1)}}$$

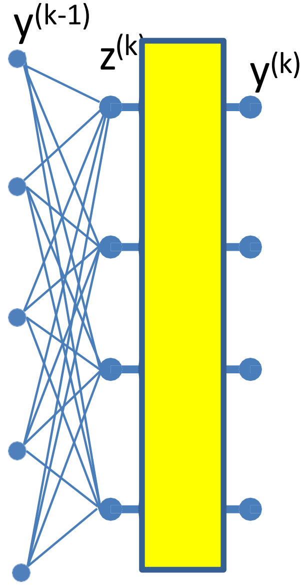
$$\frac{\partial Div}{\partial w_{ij}^{(1)}} = y_i^{(0)} \frac{\partial Div}{\partial z_j^{(1)}}$$

Special cases

- Have assumed so far that
  1. The computation of the output of one neuron does not directly affect computation of other neurons in the same layers
  2. Inputs to neurons only combine through weighted addition
  3. Activations are actually differentiable
  - All of these conditions are frequently not applicable

Special Case 1. Vector activations





- Vector activations: all outputs are functions of all inputs

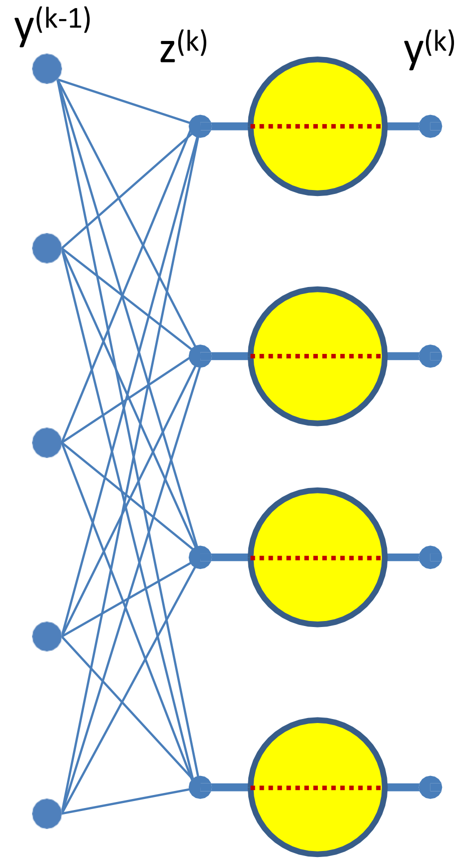
Scalar activation: Modifying a z_i only changes corresponding y_i

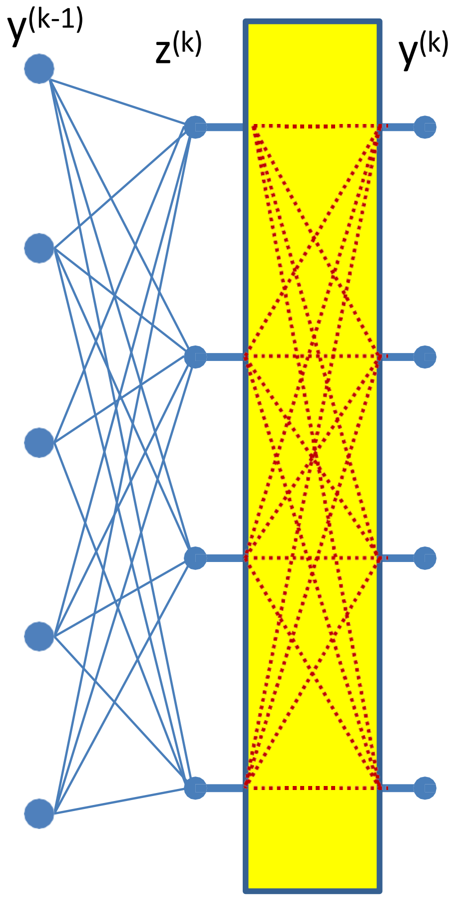
$$y_i^{(k)} = f\left(z_i^{(k)}\right)$$

Vector activation: Modifying a z_i potentially changes all $y_1 \dots y_M$

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

“Influence” diagram





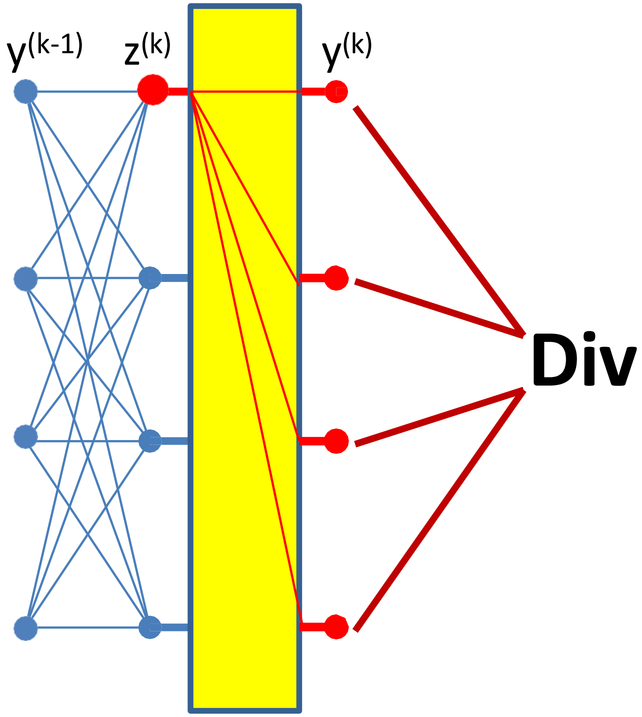
Scalar activation: Each z_i influences *one* y_i

Vector activation: Each z_i influences all, $y_1 \ldots y_M$

Derivatives of vector activation

Scalar activation: Each z_i influences *one* y_i

$$\frac{\partial Div}{\partial z_i^{(k)}} = \frac{dy_i^{(k)}}{dz_i^{(k)}} \frac{\partial Div}{\partial y_i^{(k)}}$$

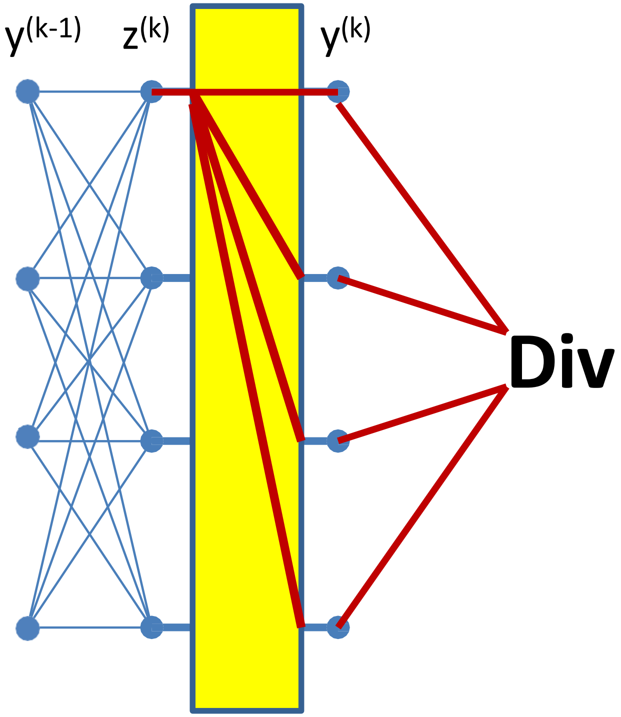


$$\frac{\partial Div}{\partial z_i^{(k)}} = \sum_j \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} \frac{\partial Div}{\partial y_j^{(k)}}$$

Note: derivatives of scalar activations are just a special case of vector activations:

$$\frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} = 0 \quad \text{for} \quad i \neq j$$

Example Vector Activation: Softmax



$$y_i^{(k)} = \frac{exp\left(z_i^{(k)}\right)}{\sum_j exp\left(z_j^{(k)}\right)}$$

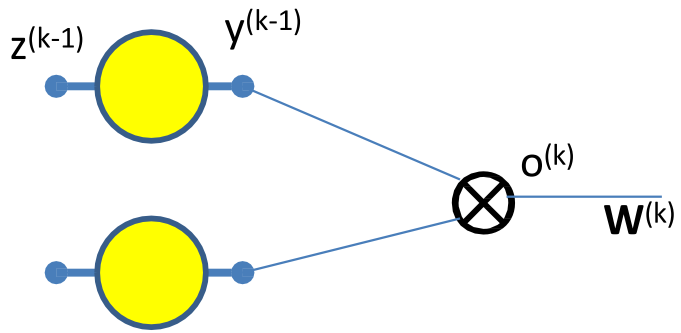
$$\frac{\partial Div}{\partial z_i^{(k)}} = \sum_j \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} \frac{\partial Div}{\partial y_j^{(k)}}$$

$$\Rightarrow \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} = \begin{cases} y_i^{(k)}\left(1 - y_i^{(k)}\right) & \text{if } i = j \\ -y_i^{(k)} y_j^{(k)} & \text{if } i \neq j \end{cases}$$

$$\Rightarrow \frac{\partial Div}{\partial z_i^{(k)}} = \sum_j \frac{\partial Div}{\partial y_j^{(k)}} y_j^{(k)} \left(\delta_{ij} - y_i^{(k)}\right)$$

$$\delta_{ij} = 1 \text{ if } i = j, \quad 0 \text{ if } i \neq j$$

Special Case 2: Multiplicative networks



Forward:

$$o_i^{(k)} = y_j^{(k-1)} y_l^{(k-1)}$$

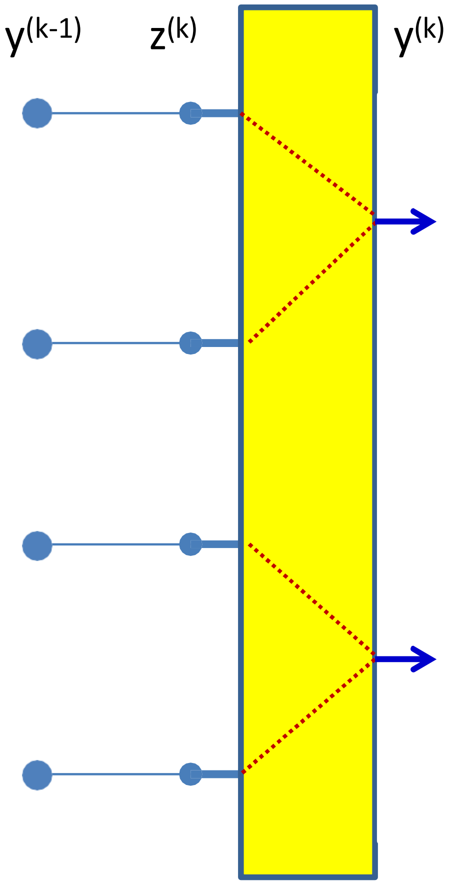
- Some types of networks have *multiplicative* combination
  - In contrast to the *additive* combination we have seen so far
- Seen in networks such as LSTMs, GRUs, attention models, etc.

Backward:

$$\frac{\partial Div}{\partial y_j^{(k-1)}} = \frac{\partial o_i^{(k)}}{\partial y_j^{(k-1)}} \frac{\partial Div}{\partial o_i^{(k)}} = y_l^{(k-1)} \frac{\partial Div}{\partial o_i^{(k)}} \qquad \frac{\partial Div}{\partial y_l^{(k-1)}} = \frac{\partial o_i^{(k)}}{\partial y_l^{(k-1)}} \frac{\partial Div}{\partial o_i^{(k)}} = y_j^{(k-1)} \frac{\partial Div}{\partial o_i^{(k)}}$$

suppose $\frac{\partial Div}{\partial o_i^{(k)}}$ is already calculated

Multiplicative combination as a case of vector activations

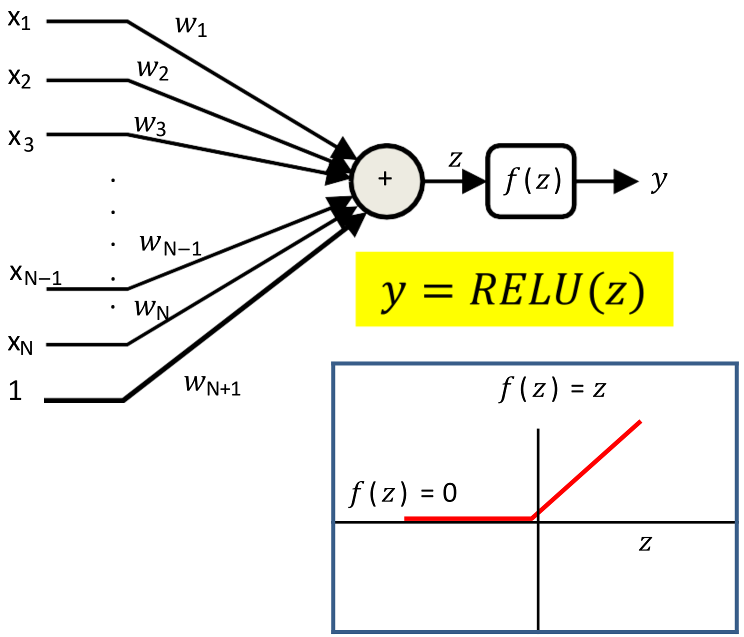


$$z_i^{(k)} = y_i^{(k-1)}$$

$$y_i^{(k)} = z_{2i-1}^{(k)} z_{2i}^{(k)}$$

- A layer of multiplicative combination is a special case of vector activation

Special Case 3: Non-differentiable Activations

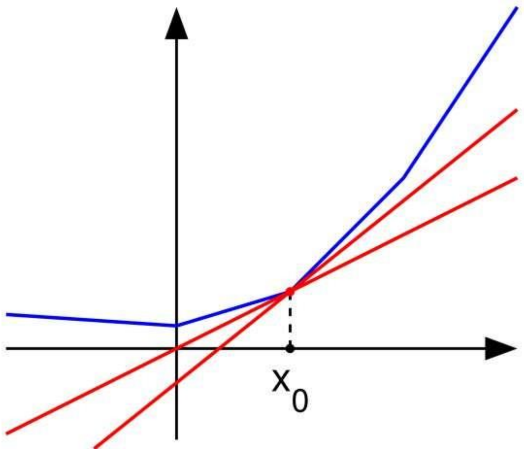


- Activation functions are sometimes not actually differentiable
  - E.g. The RELU (Rectified Linear Unit)
    - And its variants: leaky RELU, randomized leaky RELU
  - E.g. The "max" function

$$y = \max_j z_j$$

- Must use "subgradients" where available

The subgradient



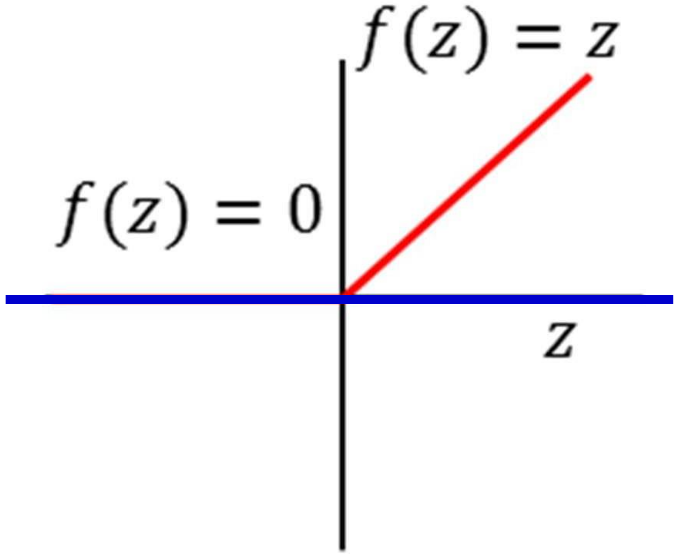
- A subgradient of a function $f(x)$ at a point x_0 is any vector v such that

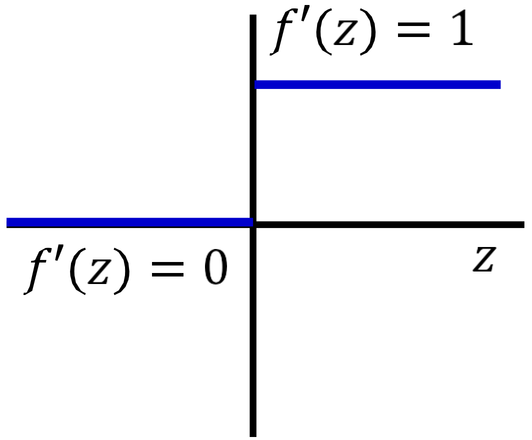
$$(f(x) - f(x_0)) \geq v^{\mathsf{T}} (x - x_0)$$

  - 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

Non-differentiability: RELU

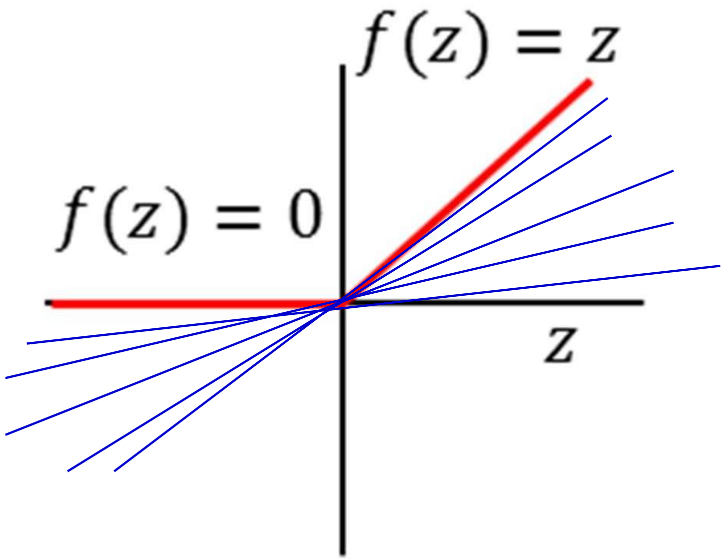




$$f'(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

$\Delta f(z) = \alpha \Delta z$

- At 0, a *negative* perturbation $\Delta z < 0$ results in no change of $f(z)$
  - $\alpha = 0$
- A *positive* perturbation $\Delta z > 0$ results in $\Delta f(z) = \Delta z$
  - $\alpha = 1$
- we can imagine that the curve is rotating continuously from slope = 0 to slope = 1 at z = 0
  - So any slope between 0 and 1 is valid



- The *subderivative* of a RELU is the slope of any line that lies entirely under it
- Can use any subgradient at 0
  - Typically, will use the equation given
  - Deep learning frameworks, such as TensorFlow and PyTorch, directly choose the gradient as 1 at the point z=0z = 0z=0 for ReLU.

Subgradients and the Max

- Multiple outputs, each selecting the max of a different subset of inputs

$$y_j = \max_{l \in \delta_j} z_l$$

  - Will be seen in convolutional networks
- Gradient for any output:
  - 1 for the specific component that is maximum in corresponding input subset
  - 0 otherwise

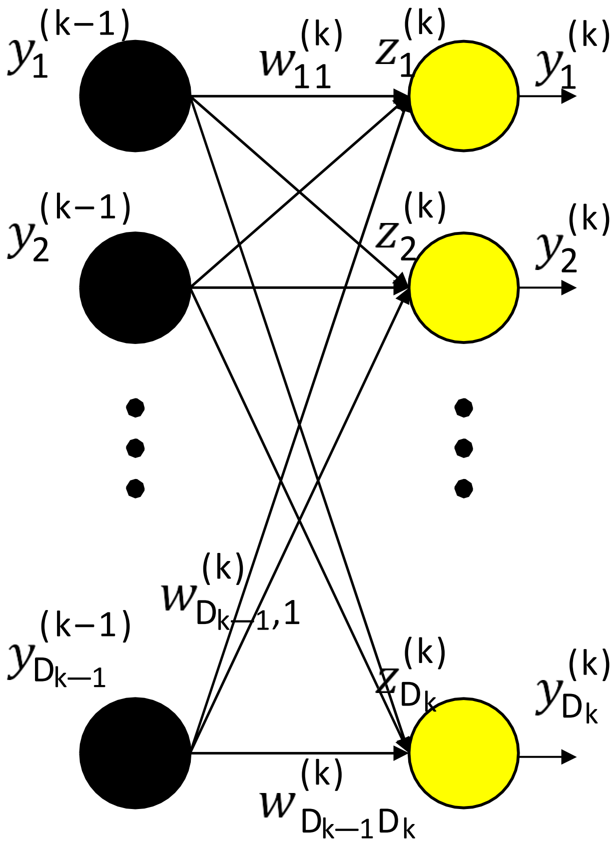
$$\frac{\partial y_j}{\partial z_i} = 1 \text{ if } i = arg\max_{l \in \delta_j} z_l, \text{ 0 otherwise}$$

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
  - This is what is *actually* used in any real system

Vector formulation

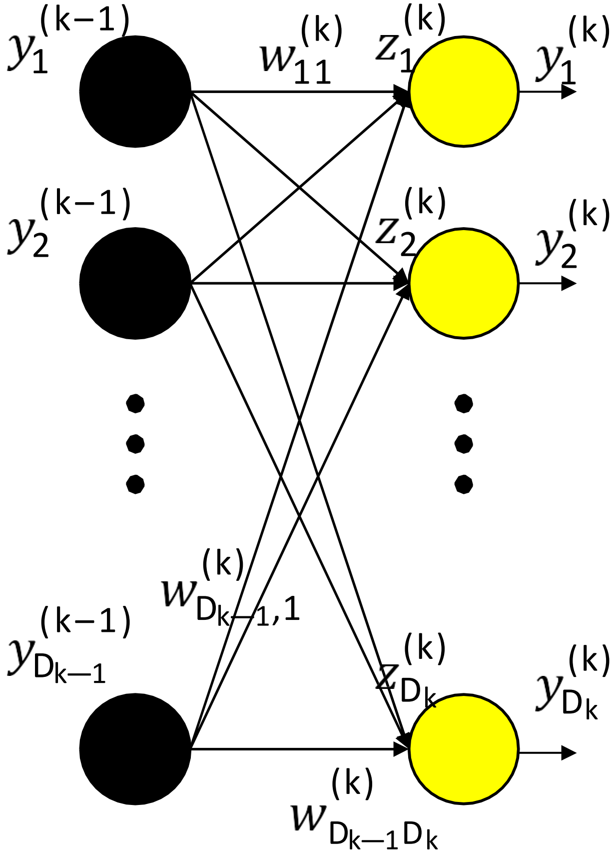


$$\mathbf{z}_k = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{D_k}^{(k)} \end{bmatrix} \qquad \mathbf{y}_k = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_{D_k}^{(k)} \end{bmatrix}$$

$$\mathbf{W}_k = \begin{bmatrix} w_{11}^{(k)} & w_{21}^{(k)} & \vdots & w_{D_{k-1}1}^{(k)} \\ w_{12}^{(k)} & w_{22}^{(k)} & \vdots & w_{D_{k-1}2}^{(k)} \\ \cdots & \cdots & \ddots & \vdots \\ w_{1D_k}^{(k)} & w_{2D_k}^{(k)} & \cdots & w_{D_{k-1}D_k}^{(k)} \end{bmatrix} \qquad \mathbf{b}_k = \begin{bmatrix} b_1^{(k)} \\ b_2^{(k)} \\ \vdots \\ b_{D_k}^{(k)} \end{bmatrix}$$

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

Vector formulation



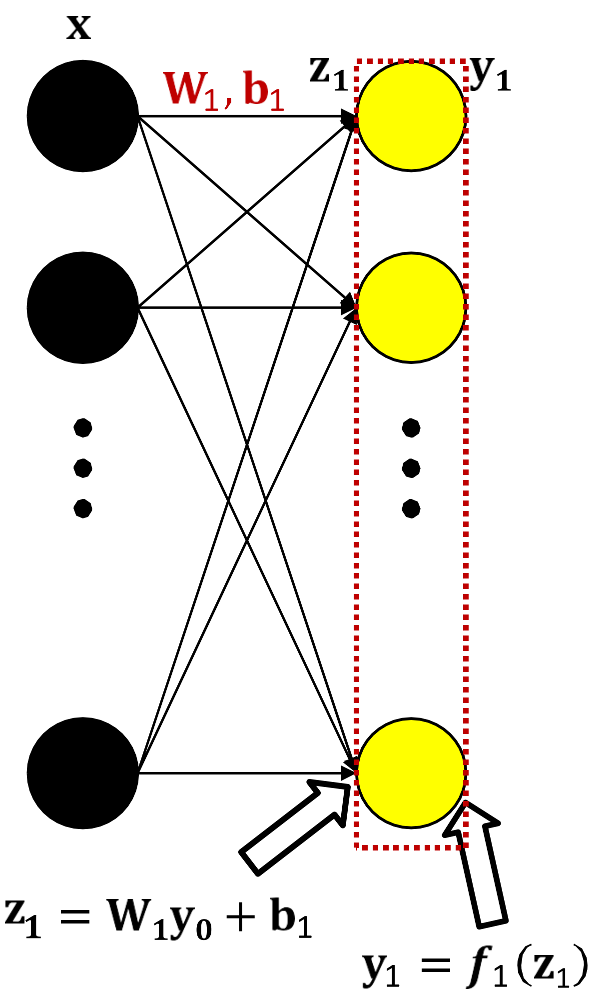
$$\mathbf{z}_k = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{D_k}^{(k)} \end{bmatrix} \qquad \mathbf{y}_k = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_{D_k}^{(k)} \end{bmatrix}$$

$$\mathbf{W}_k = \begin{bmatrix} w_{11}^{(k)} & w_{21}^{(k)} & \vdots & w_{D_{k-1}1}^{(k)} \\ w_{12}^{(k)} & w_{22}^{(k)} & \vdots & w_{D_{k-1}2}^{(k)} \\ \cdots & \cdots & \ddots & \vdots \\ w_{1D_k}^{(k)} & w_{2D_k}^{(k)} & \cdots & w_{D_{k-1}D_k}^{(k)} \end{bmatrix} \qquad \mathbf{b}_k = \begin{bmatrix} b_1^{(k)} \\ b_2^{(k)} \\ \vdots \\ b_{D_k}^{(k)} \end{bmatrix}$$

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

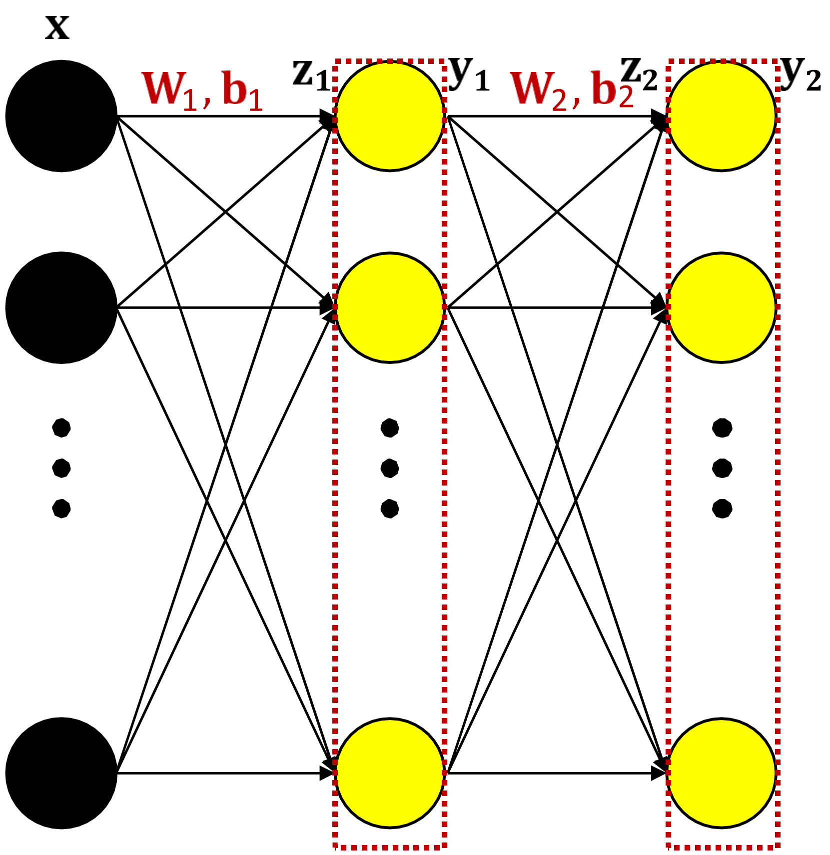
$$\boxed{\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k} \qquad \boxed{\mathbf{y}_k = f_k(\mathbf{z}_k)}$$

The forward pass



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

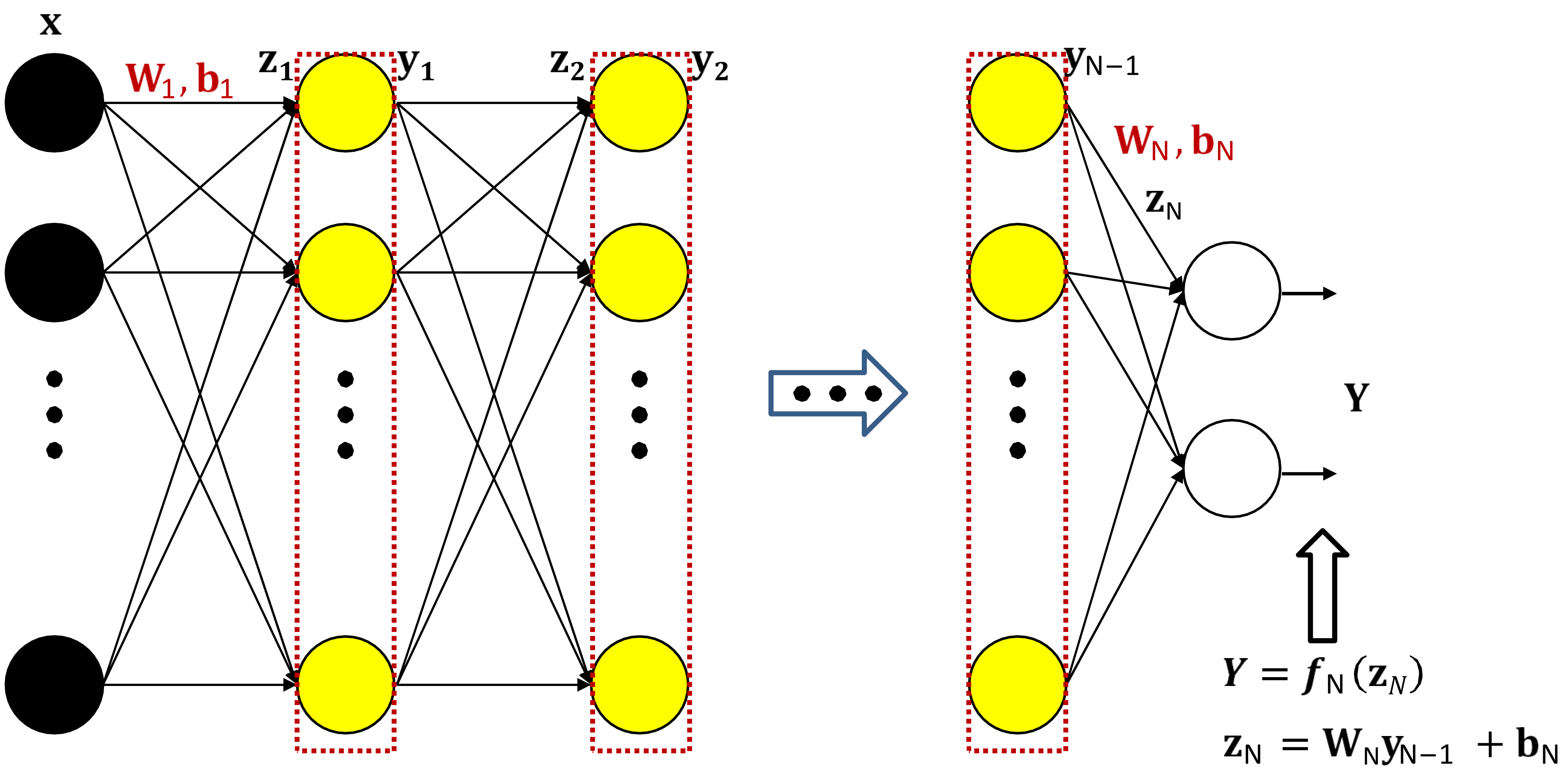
The forward pass



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

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

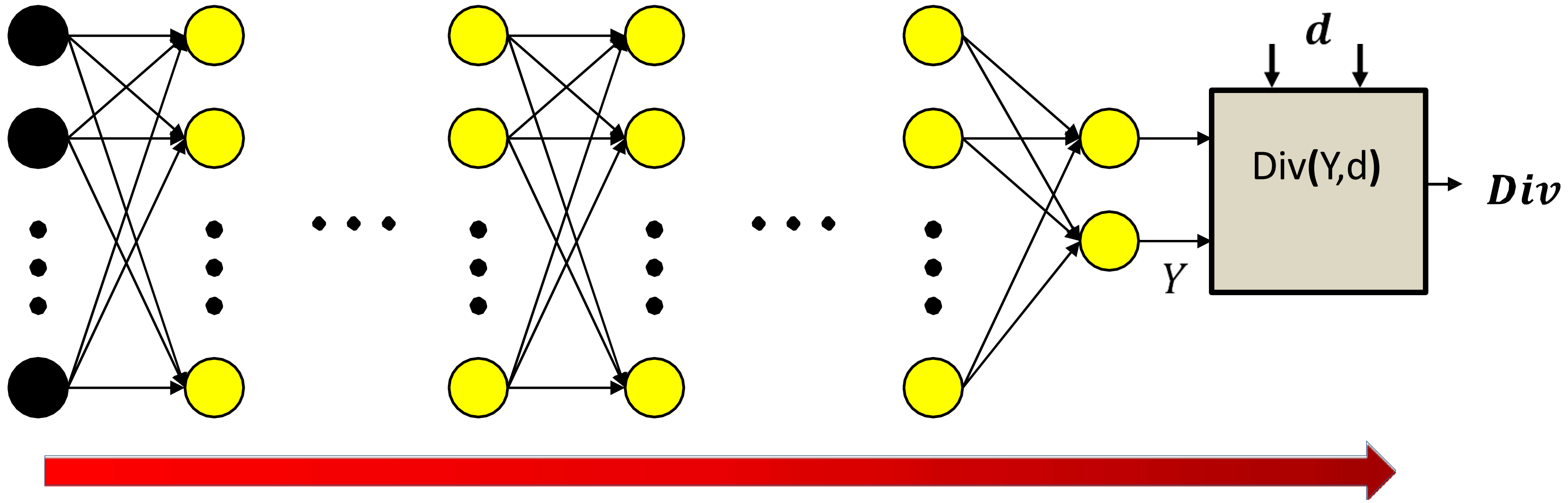
The forward pass



$$\mathbf{z}_N = \mathbf{W}_N f_{N-1}\left(\ldots f_2(\mathbf{W}_2 f_1(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) \ldots\right) + \mathbf{b}_N$$

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

Forward pass



Forward pass:

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

For k = 1 to N: $\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k \qquad \mathbf{y}_k = \boldsymbol{f}_k(\mathbf{z}_k)$

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

The Backward Pass

- Have completed the forward pass
- Before presenting the backward pass, some more calculus…
  - *Vector* calculus this time

Vector Calculus Notes: Definitions

- **For a scalar function of a vector argument**

$$y = f(\mathbf{z})$$
$$\Delta y = \nabla_{\mathbf{z}} y \, \Delta \mathbf{z}$$

If $\boldsymbol{z}$ is an $\mathrm{D} \times 1$ vector, and y is a scalar
$\nabla_z y$ is a $1 \times D$ vector

The shape of the derivative is the transpose of the shape of $\boldsymbol{z}$

$\nabla_z y^T$ is called the **gradient** of y w.r.t. $\boldsymbol{z}$

Otherwise, the dimension won't match

- **For a vector function of a vector argment**

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

$$\Delta \mathbf{y} = \nabla_{\mathbf{z}} y \, \Delta \mathbf{z}$$

If $\boldsymbol{z}$ is an $\mathrm{D} \times 1$ vector, y is an $\mathrm{M} \times 1$ vector
$\nabla_z y$ is a $\mathrm{M} \times D$ matrix

$\nabla_z y$ is called the **Jacobian** of y w.r.t. $\boldsymbol{z}$

Vector Calculus Notes: The Jacobian

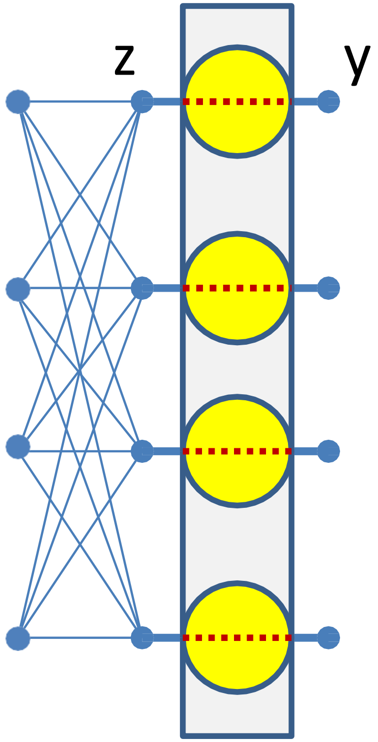
- The derivative of a vector function w.r.t. vector input is called a *Jacobian*
- It is the matrix of partial derivatives given below

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

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

$\nabla_z y$ is a $\mathrm{M} \times D$ matrix

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



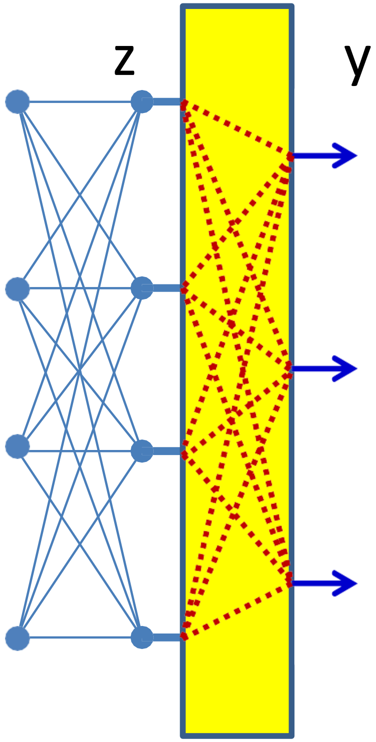
For scalar activations

$$y_i = f(z_i)$$

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

- Jacobian is a diagonal matrix
- Diagonal entries are individual derivatives of outputs w.r.t inputs

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

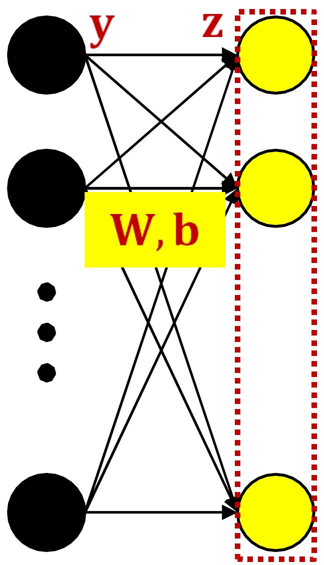


For vector activations

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

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

Special case: Affine functions



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

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

- 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 Calculus Notes 2: Chain rule

- For nested functions we have the following chain rule

$$\mathbf{y} = \boldsymbol{y}(\boldsymbol{z}(\mathbf{x})) \Longrightarrow \nabla_{\mathbf{x}}\mathbf{y} = \nabla_{\mathbf{z}}\boldsymbol{y}\nabla_{\mathbf{x}}\mathbf{z}$$

$$\Delta\mathbf{y} = \nabla_{\mathbf{z}}\boldsymbol{y}\nabla_{\mathbf{x}}\mathbf{z}\Delta\mathbf{x} = \nabla_{\mathbf{x}}\mathbf{y}\Delta\mathbf{x}$$

- Chain rule for Jacobians (vector functions of vector inputs):

$$\mathbf{y} = \boldsymbol{y}(\boldsymbol{z}(\mathbf{x})) \Longrightarrow J_{\mathbf{y}}(\mathbf{x}) = J_{\mathbf{y}}(\mathbf{z})J_{\mathbf{z}}(\mathbf{x})$$

$$\Delta\mathbf{y} = J_{\mathbf{y}}(\mathbf{z})J_{\mathbf{z}}(\mathbf{x})\Delta\mathbf{x} = J_{\mathbf{y}}(\mathbf{x})\Delta\mathbf{x}$$

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

- Combining Jacobians and Gradients

$$D = D(\boldsymbol{y}(\mathbf{z})) \Longrightarrow \nabla_{\mathbf{z}}D = \nabla_{\boldsymbol{y}}(D)J_{\boldsymbol{y}}(\mathbf{z})$$

More calculus: Special Case

- Scalar functions of Affine functions

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

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

$$\nabla_{\mathbf{y}} D = \nabla_{\mathbf{z}}(D)\mathbf{W}$$

$$\nabla_{\mathbf{b}} D = \nabla_{\mathbf{z}}(D)$$

Derivatives w.r.t parameters?

Today's Star

$$\nabla_{\mathbf{W}} D = \mathbf{y}\nabla_{\mathbf{z}}(D)$$

- Note: the derivative shapes are the *transpose* of the shapes of $\mathbf{W}$ and $\mathbf{b}$

$$\mathbf{z}^\top = \mathbf{y}^\top\mathbf{W}^\top + \mathbf{b}^\top$$ Writing the transpose

$$\nabla_{W^\top} \mathbf{z}^\top = \mathbf{y}^\top$$

$$\nabla_{W^\top} D = \nabla_{z^\top} D \, \nabla_{W^\top} \mathbf{z}^\top = \nabla_{z^\top} D \;\, \mathbf{y}^\top$$

$$\nabla_{W} D = (\nabla_{W^\top} D)^\top = \mathbf{y}\nabla_{z} D$$

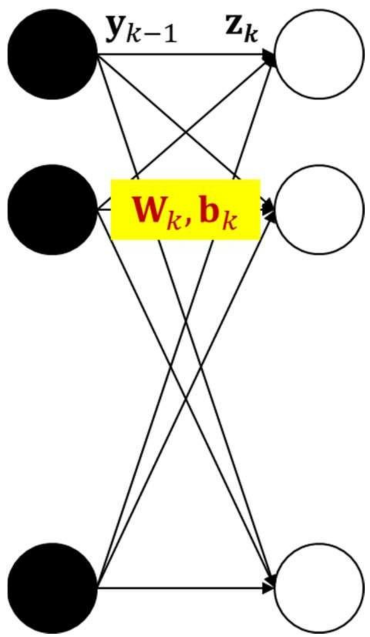
$$\boxed{\nabla_{\mathbf{W}} D = \mathbf{y}\nabla_{\mathbf{z}}(D)}$$

Special Case: Application to a network

- Scalar functions of Affine functions

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

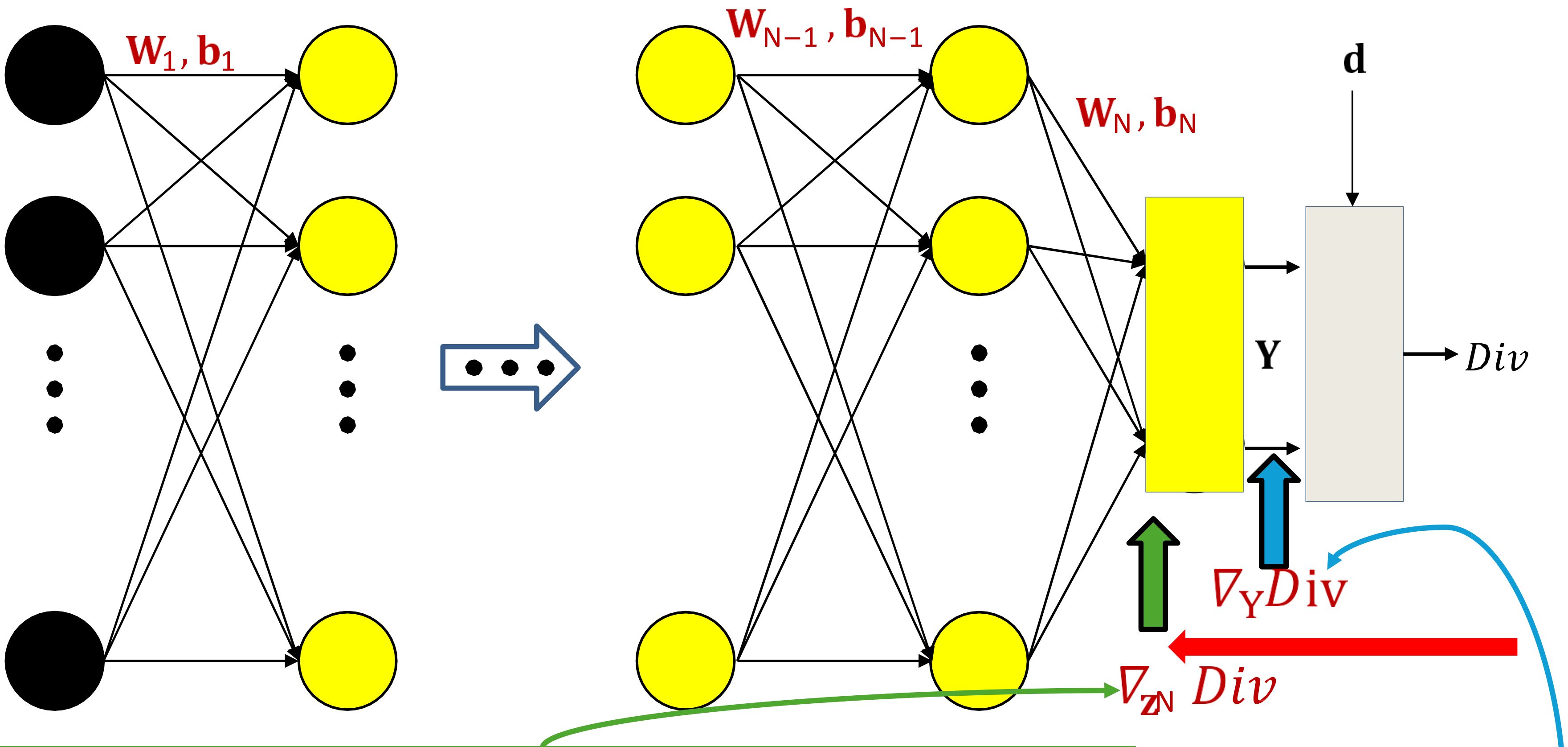
$$Div = Div(\mathbf{z}_k) \quad \Rightarrow \quad \nabla_{\mathbf{y}_{k-1}} Div = \nabla_{\mathbf{z}_k} Div \mathbf{W}_k$$



$$\nabla_{\mathbf{b}_k} Div = \nabla_{\mathbf{z}_k} Div$$

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

The backward pass



The divergence is a nested function: $Div(\mathbf{Y}(\mathbf{z}_N))$

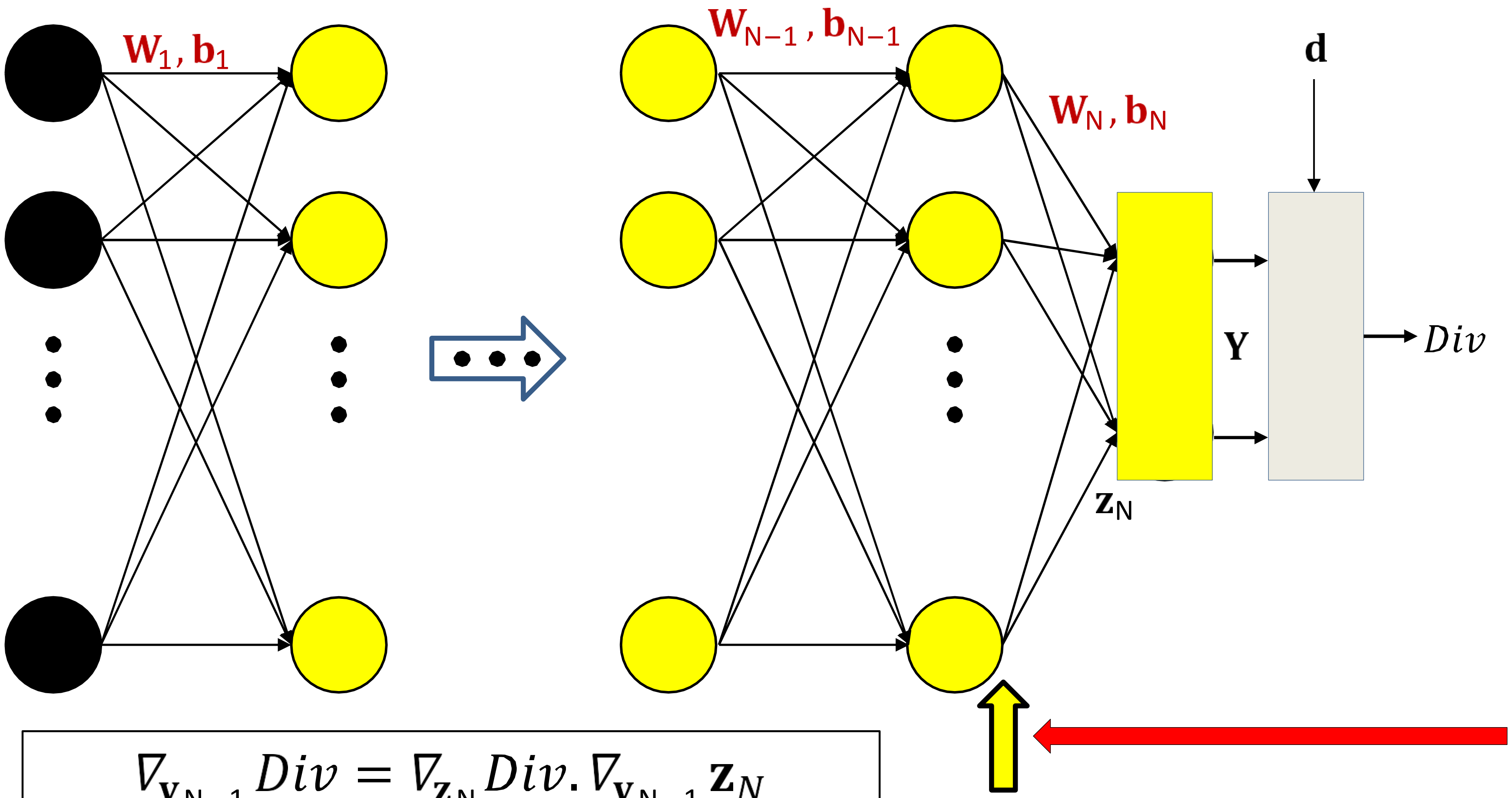
$$\nabla_{\mathbf{z}_N} Div = \nabla_{\mathrm{Y}} Div . \nabla_{\mathbf{z}_N} \mathbf{Y} = \nabla_{\mathrm{Y}} Div . J_{\mathbf{Y}}(\mathbf{z}_N)$$

Already computed

New term

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

The backward pass



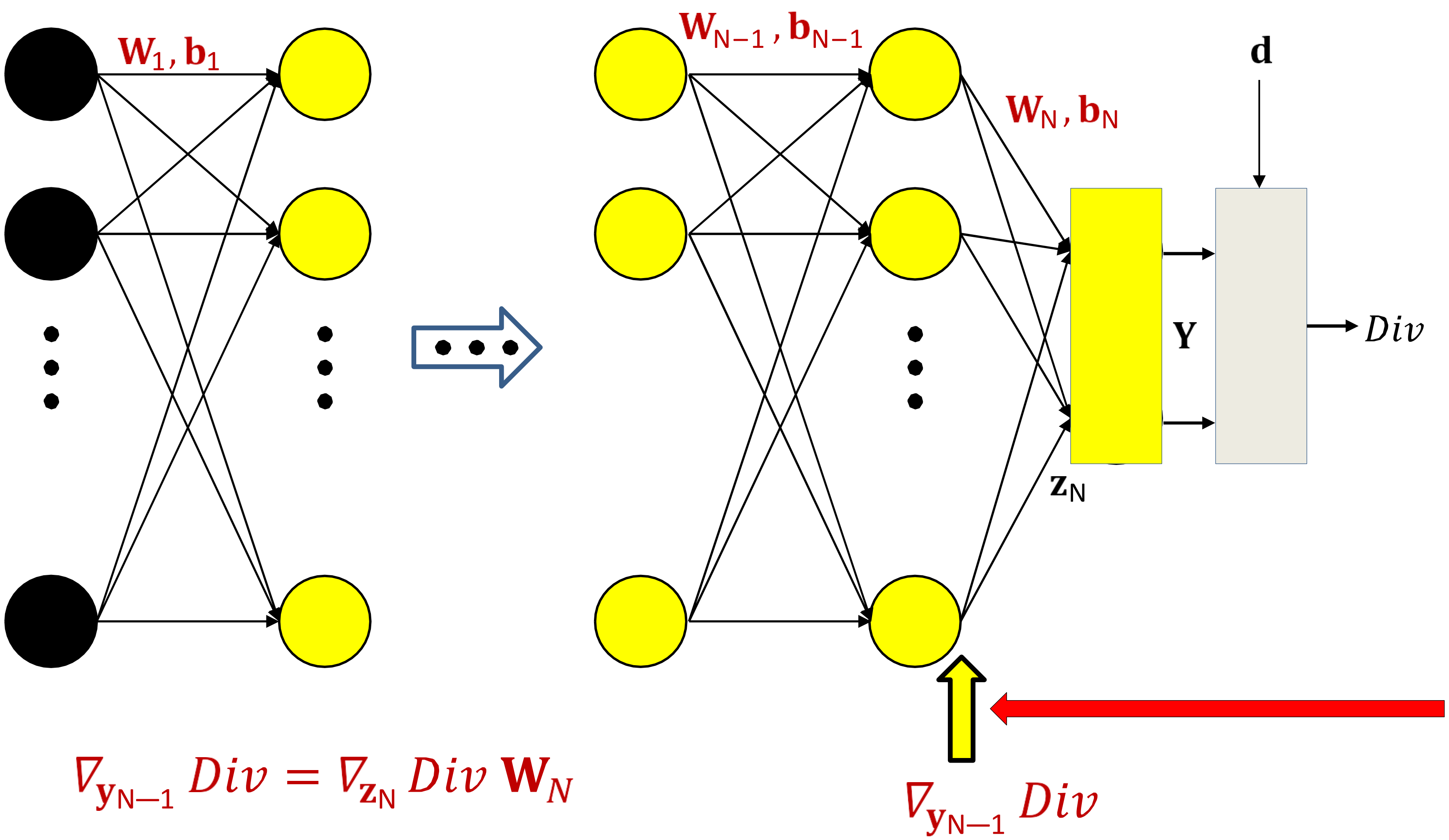
$$\boxed{\nabla_{\mathbf{y}_{N-1}} Div = \nabla_{\mathbf{z}_N} Div . \nabla_{\mathbf{y}_{N-1}} \mathbf{z}_N}$$

$\nabla_{\mathbf{y}_{N-1}} Div$

$$\mathbf{z}_N = \mathbf{W}_N \mathbf{y}_{N-1} + \mathbf{b}_N \quad \Rightarrow \quad \nabla_{\mathbf{y}_{N-1}} \mathbf{z}_N = \mathbf{W}_N$$

$$\nabla_{\mathbf{y}_{N-1}} Div = \underbrace{\nabla_{\mathbf{z}_N} Div}_{\text{Already computed}} \; \underbrace{\mathbf{W}_N}_{\text{New term}}$$

The backward pass

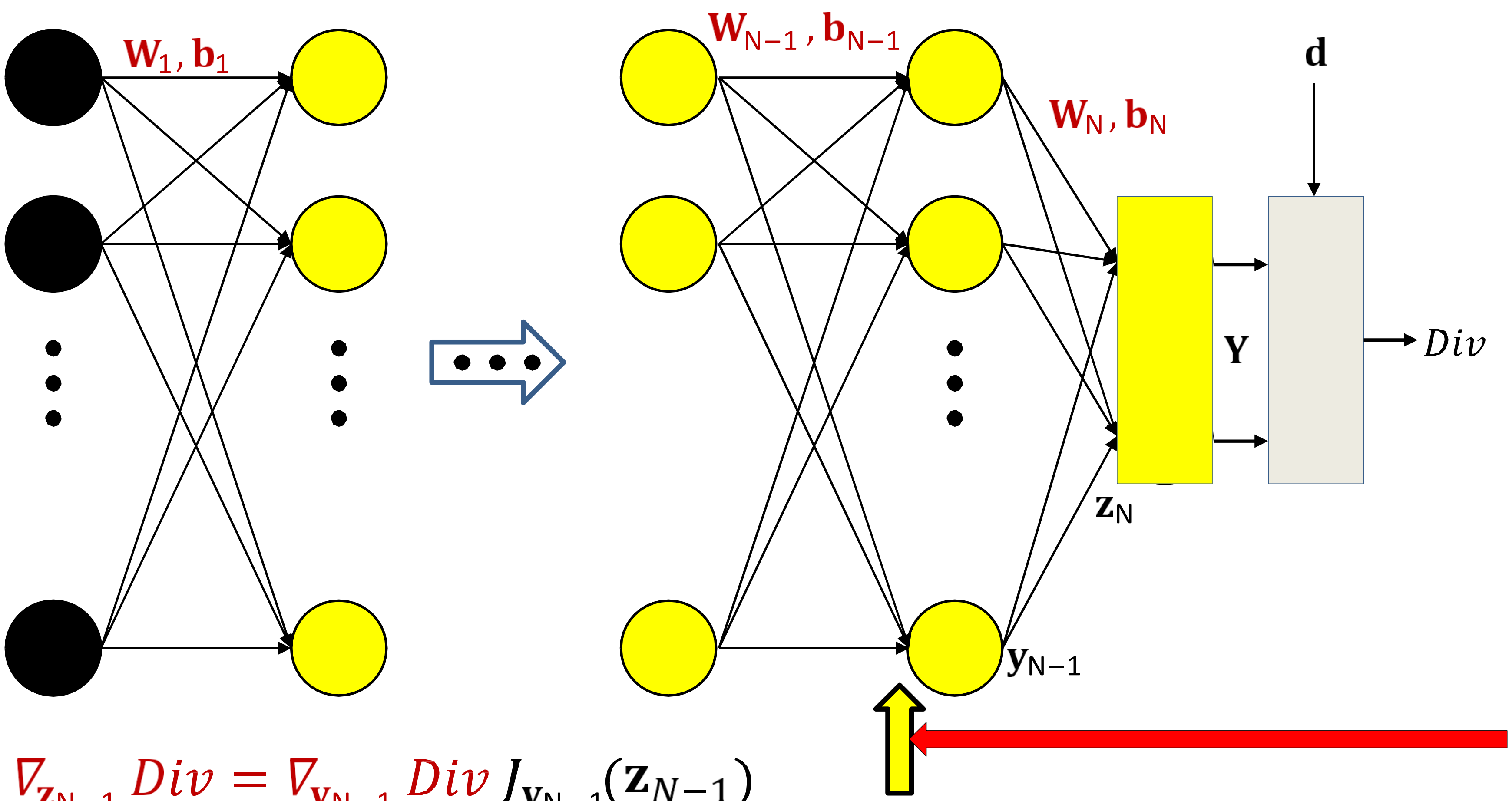


$$\nabla_{\mathbf{y}_{N-1}} Div = \nabla_{\mathbf{z}_N} Div\, \mathbf{W}_N$$

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

$$\boxed{\nabla_{\mathbf{b}_N} Div = \nabla_{\mathbf{z}_N} Div}$$

The backward pass



$$\nabla_{\mathbf{z}_{N-1}} Div = \nabla_{\mathbf{y}_{N-1}} Div \, J_{\mathbf{y}_{N-1}}(\mathbf{z}_{N-1})$$

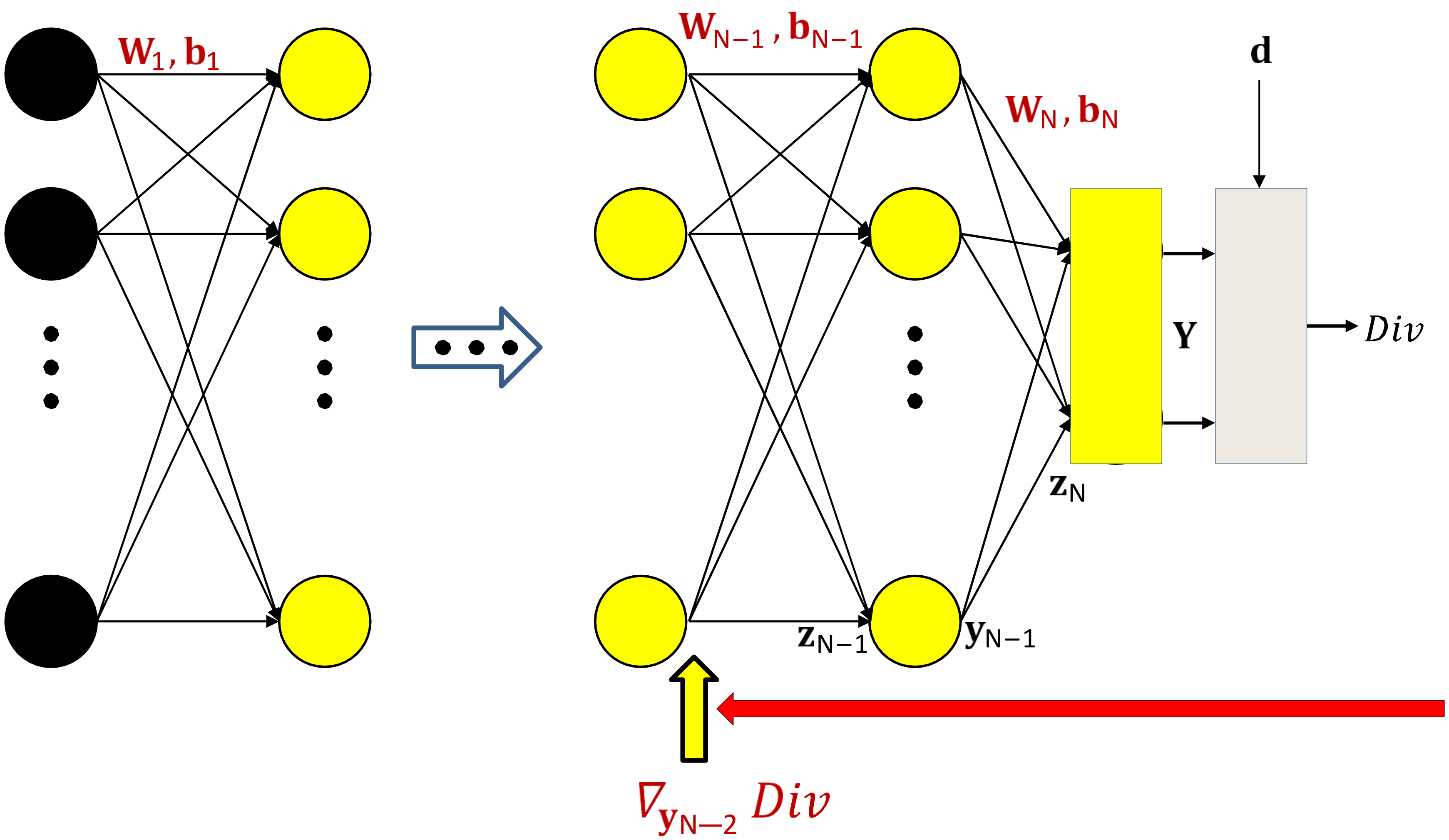
The Jacobian will be a diagonal matrix for scalar activations

$$\boxed{\nabla_{\mathbf{z}_{N-1}} Div = \nabla_{\mathbf{y}_{N-1}} Div . \nabla_{\mathbf{z}_{N-1}} \mathbf{y}_{N-1}}$$

Already computed

New term

The backward pass



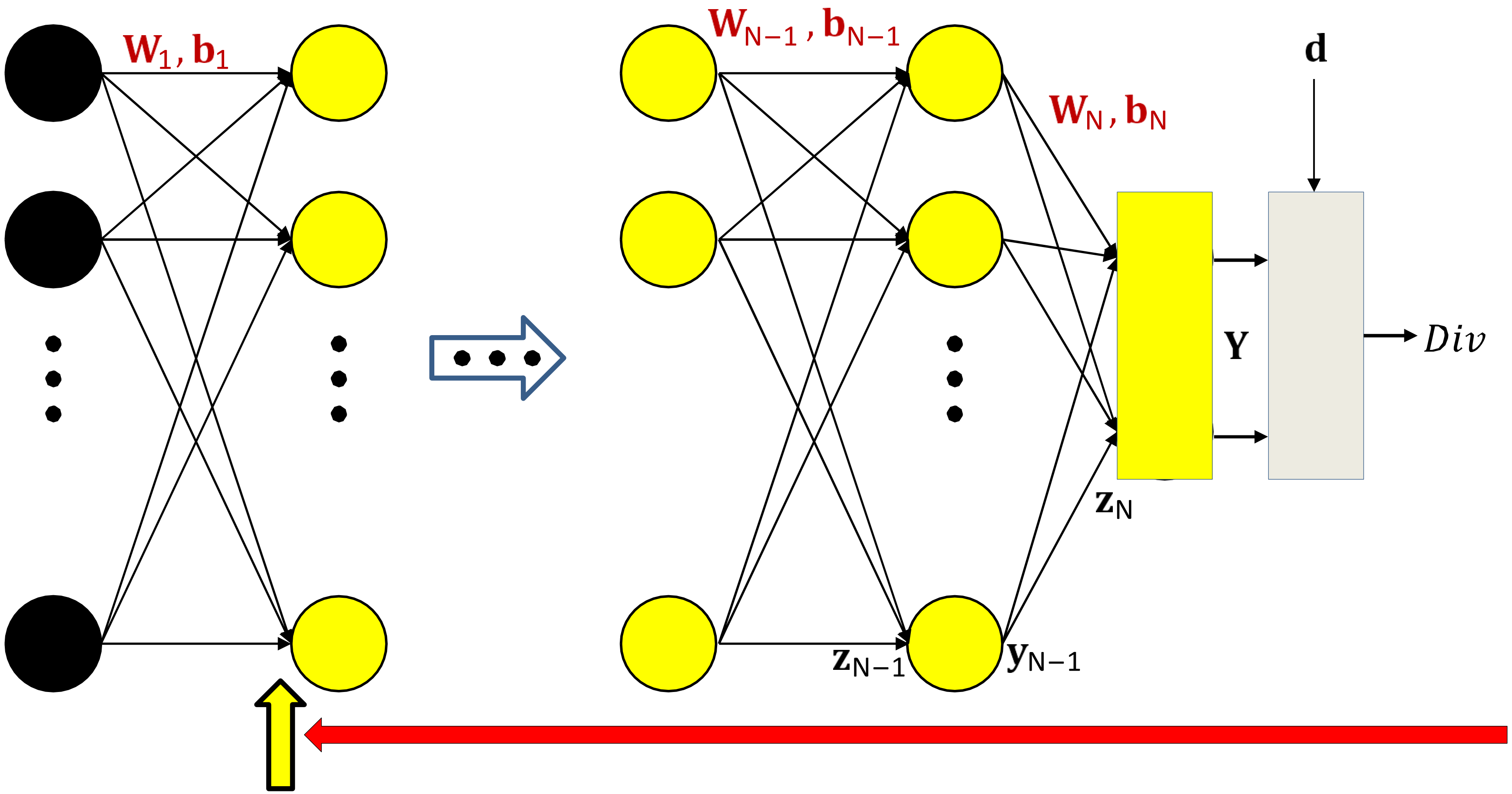
$$\boxed{\nabla_{\mathbf{y}_{N-2}} Div = \nabla_{\mathbf{z}_{N-1}} Div . \nabla_{\mathbf{y}_{N-2}} \mathbf{z}_{N-1}}$$

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

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

$$\boxed{\nabla_{\mathbf{b}_{N-1}} Div = \nabla_{\mathbf{z}_{N-1}} Div}$$

The backward pass



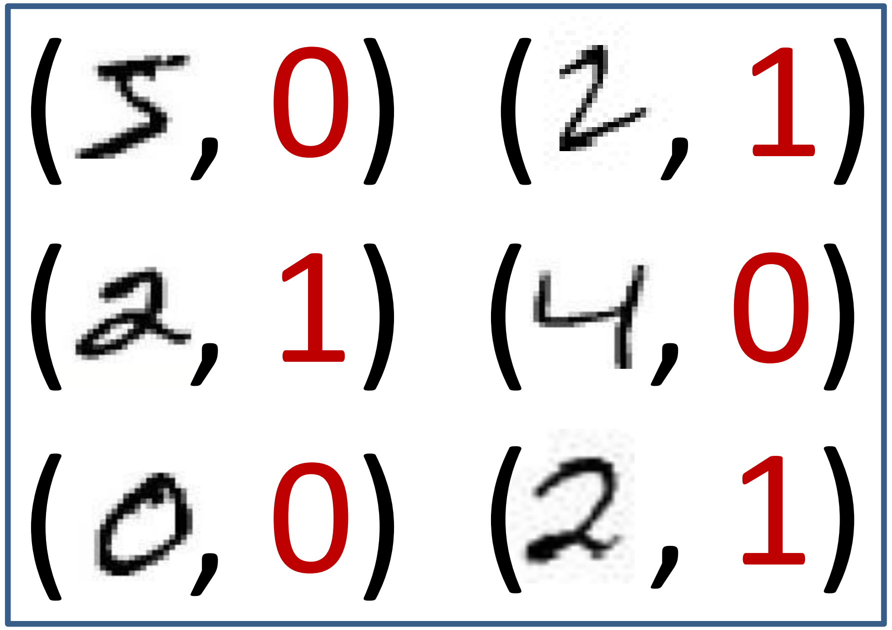
$$\nabla_{\mathbf{z}_1} Div = \nabla_{\mathbf{y}_1} Div \, J_{\mathbf{y}_1}(\mathbf{z}_1)$$

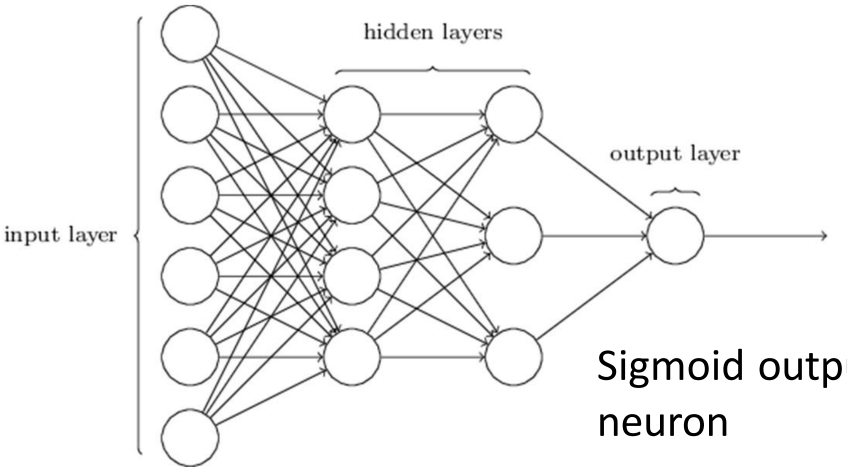
$$\boxed{\nabla_{\mathbf{W}_1} Div = \mathbf{x} \nabla_{\mathbf{z}_1} Div}$$

$$\boxed{\nabla_{\mathbf{b}_1} Div = \nabla_{\mathbf{z}_1} Div}$$

Setting up for digit recognition

Training data



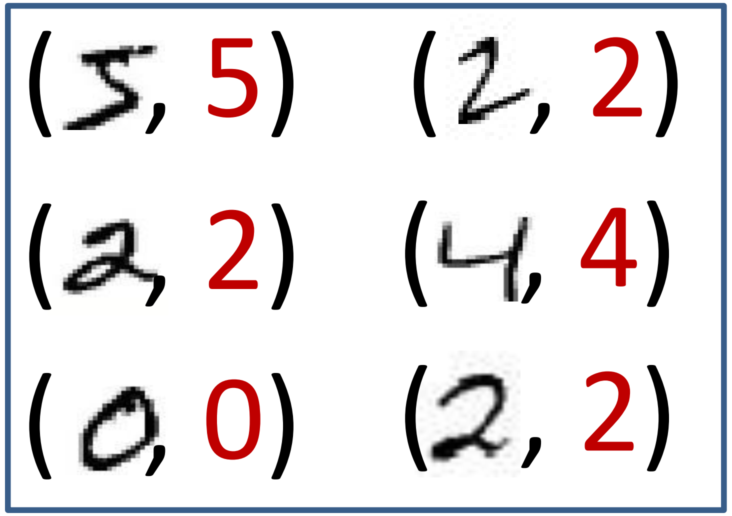


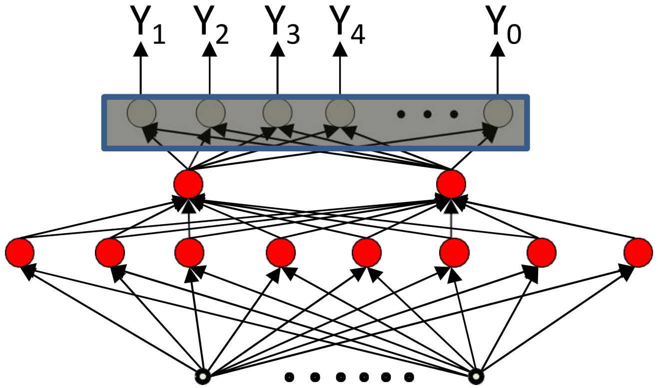
Sigmoid output neuron

- 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 compute derivatives
  - To apply in gradient descent to learn network parameters

Recognizing the digit

Training data





- 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 compute derivatives for gradient descent updates to learn network