

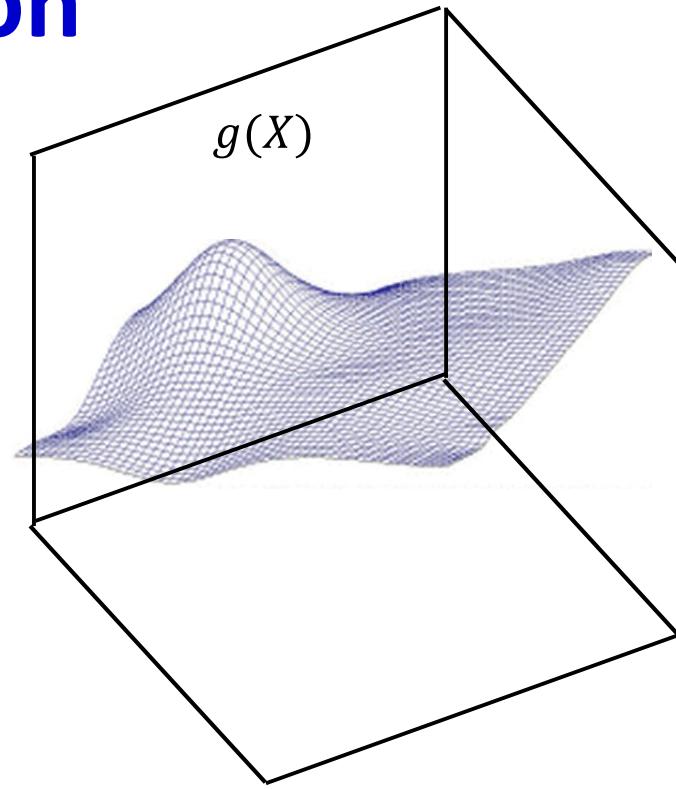
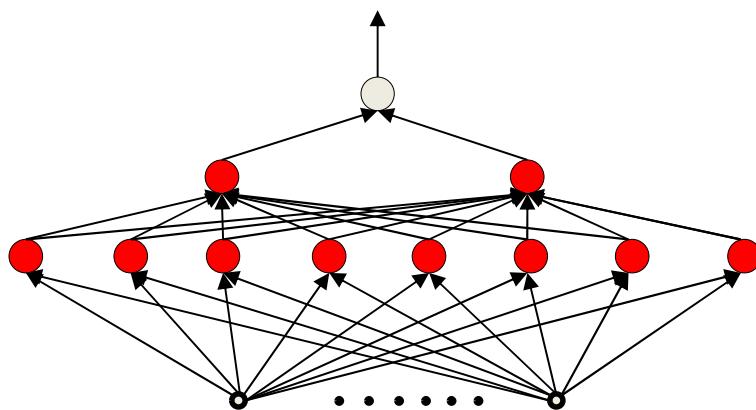
Neural Networks

Learning the network: Backprop

11-785, Fall 2020

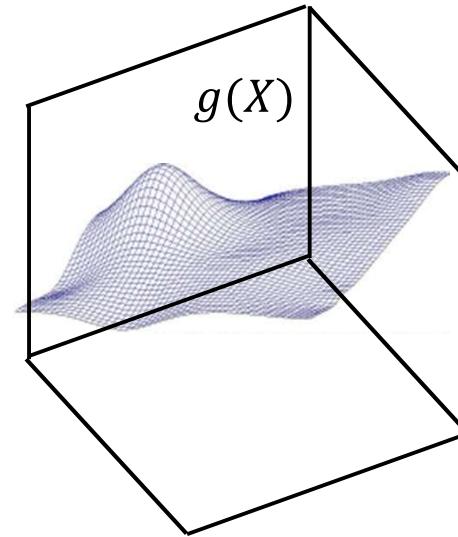
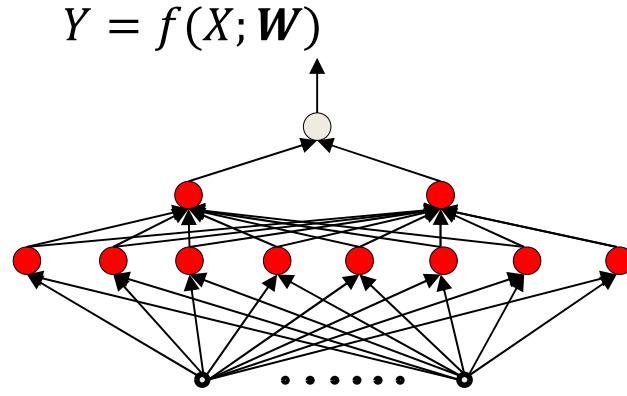
Lecture 4

Recap: The MLP *can* represent any function



- The MLP *can be constructed* to represent anything
- But *how* do we construct it?
 - *I.e.* how do we determine the weights (and biases) of the network to best represent a target function
 - *Assuming that the architecture of the network is given*

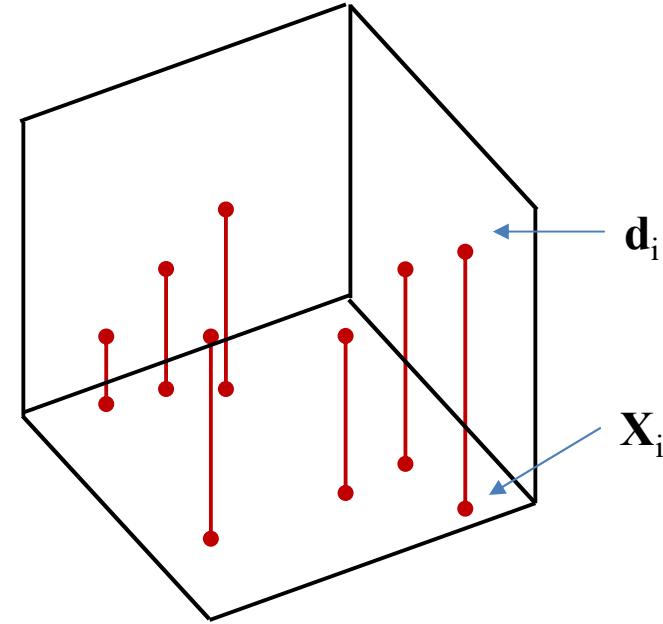
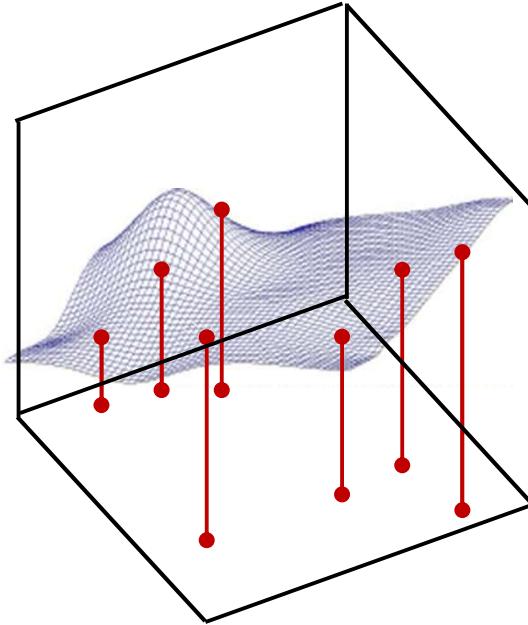
Recap: How to learn the function



- By minimizing expected divergence

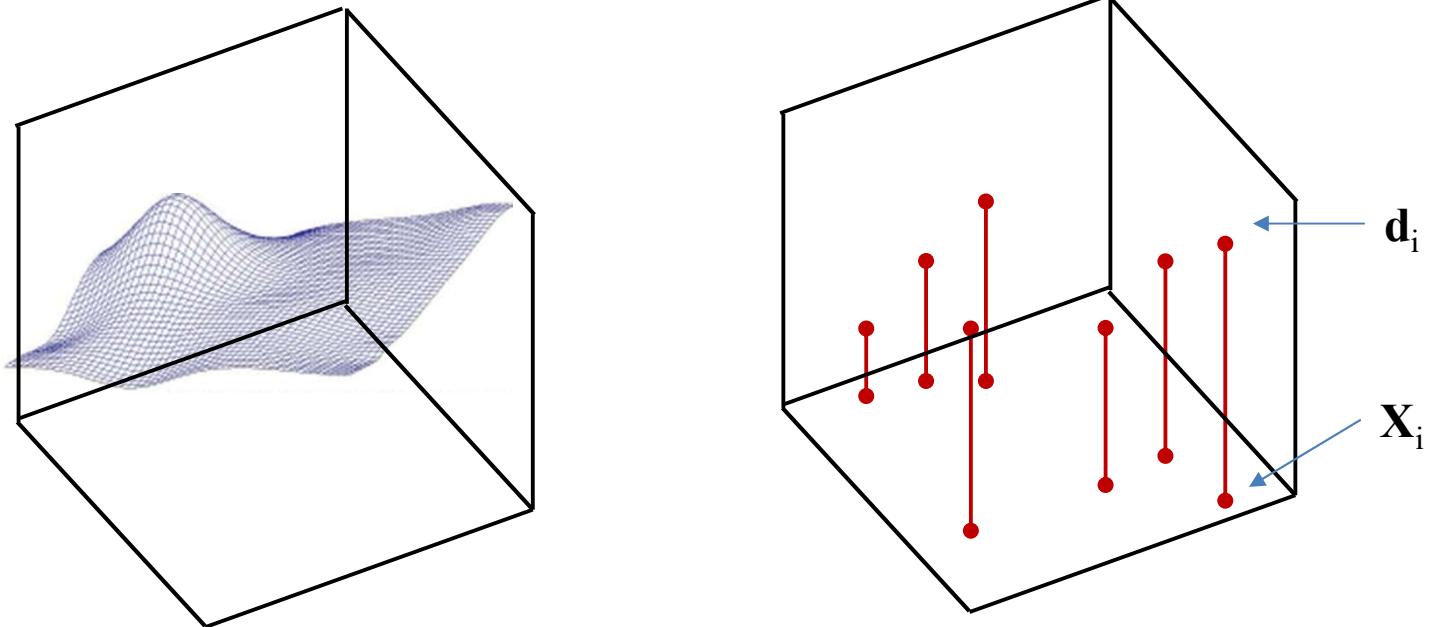
$$\begin{aligned}\widehat{\mathbf{W}} &= \operatorname{argmin}_{\mathbf{W}} \int_X \operatorname{div}(f(X; \mathbf{W}), g(X)) P(X) dX \\ &= \operatorname{argmin}_{\mathbf{W}} E[\operatorname{div}(f(X; \mathbf{W}), g(X))]\end{aligned}$$

Recap: Sampling the function



- $g(X)$ is unknown, so sample it
 - Basically, get input-output pairs for a number of samples of input X_i
 - Good sampling: the samples of X will be drawn from $P(X)$
- Estimate network from the samples

The *Empirical* risk

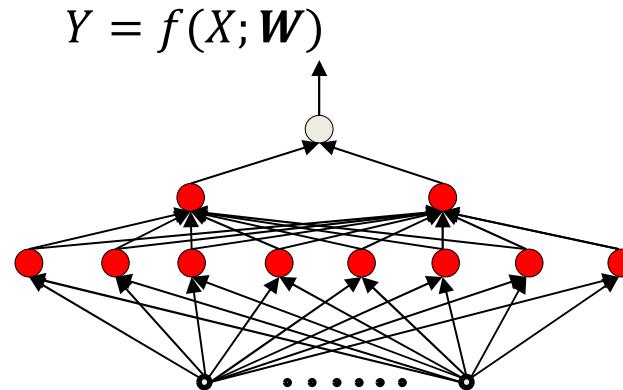


- The *empirical estimate* of the expected divergence is the *average* divergence over the samples

$$E[div(f(X; W), g(X))] \approx \frac{1}{T} \sum_{i=1}^T div(f(X_i; W), d_i)$$

- This approximation is an unbiased estimate of the *expected* divergence that we *actually* want to estimate
 - We can *hope* that minimizing the empirical loss will minimize the true loss
 - Caveat: This hope is generally not based on anything but, well, hope..

Empirical Risk Minimization



- Given a training set of input-output pairs $(\mathbf{X}_1, \mathbf{d}_1), (\mathbf{X}_2, \mathbf{d}_2), \dots, (\mathbf{X}_T, \mathbf{d}_T)$
 - Divergence on the i -th instance: $\text{div}(f(\mathbf{X}_i; \mathbf{W}), \mathbf{d}_i)$
 - Empirical average divergence on all training data:

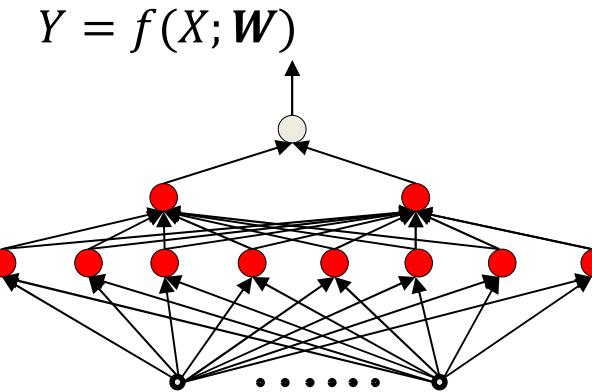
$$\text{Loss}(\mathbf{W}) = \frac{1}{T} \sum_i \text{div}(f(\mathbf{X}_i; \mathbf{W}), \mathbf{d}_i)$$

- Estimate the parameters to minimize the empirical estimate of expected divergence

$$\widehat{\mathbf{W}} = \operatorname{argmin}_{\mathbf{W}} \text{Loss}(\mathbf{W})$$

- I.e. minimize the *empirical risk* over the drawn samples

Empirical Risk Minimization



This is an instance of
function minimization
(optimization)

- Given a training set of input-output pairs $(\mathbf{X}_1, \mathbf{d}_1), (\mathbf{X}_2, \mathbf{d}_2), \dots, (\mathbf{X}_T, \mathbf{d}_T)$
 - Error on the i-th instance: $\text{div}(f(\mathbf{X}_i; \mathbf{W}), \mathbf{d}_i)$
 - Empirical average error on all training data:

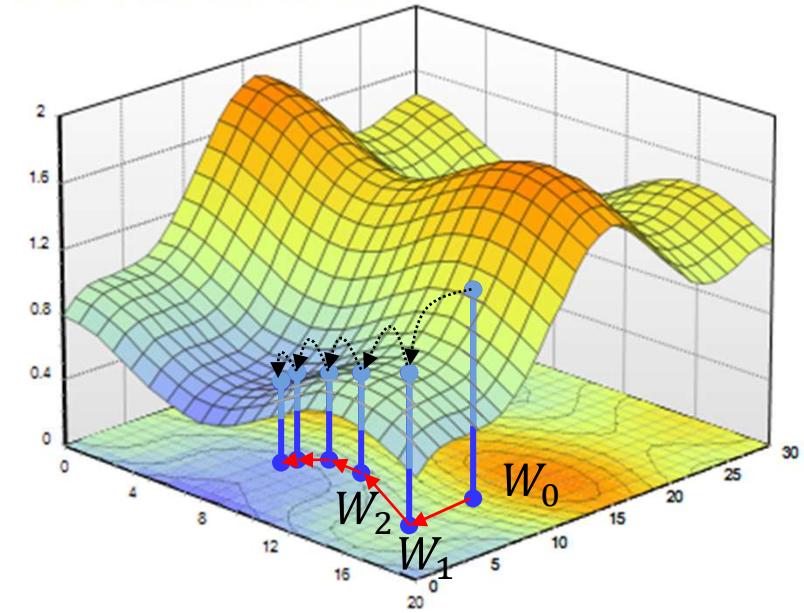
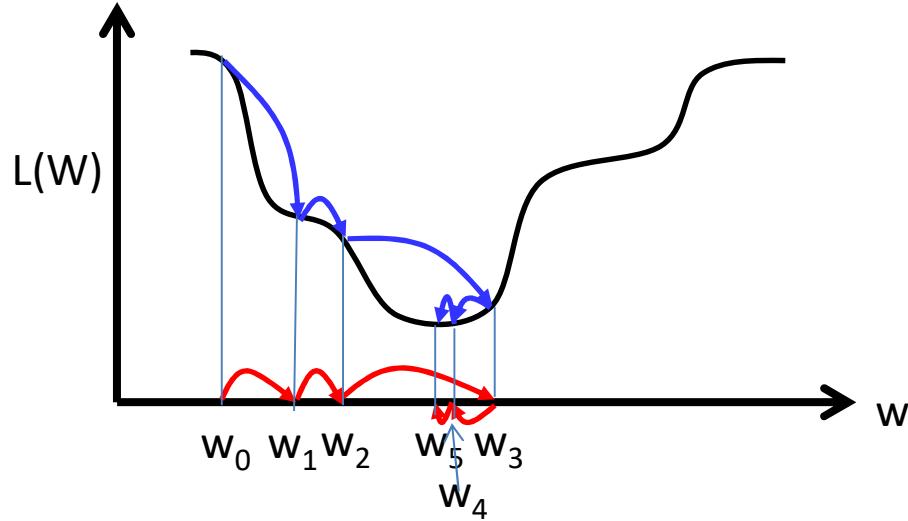
$$\text{Loss}(\mathbf{W}) = \frac{1}{T} \sum_i \text{div}(f(\mathbf{X}_i; \mathbf{W}), \mathbf{d}_i)$$

- Estimate the parameters to minimize the empirical estimate of expected error

$$\widehat{\mathbf{W}} = \operatorname{argmin}_{\mathbf{W}} \text{Loss}(\mathbf{W})$$

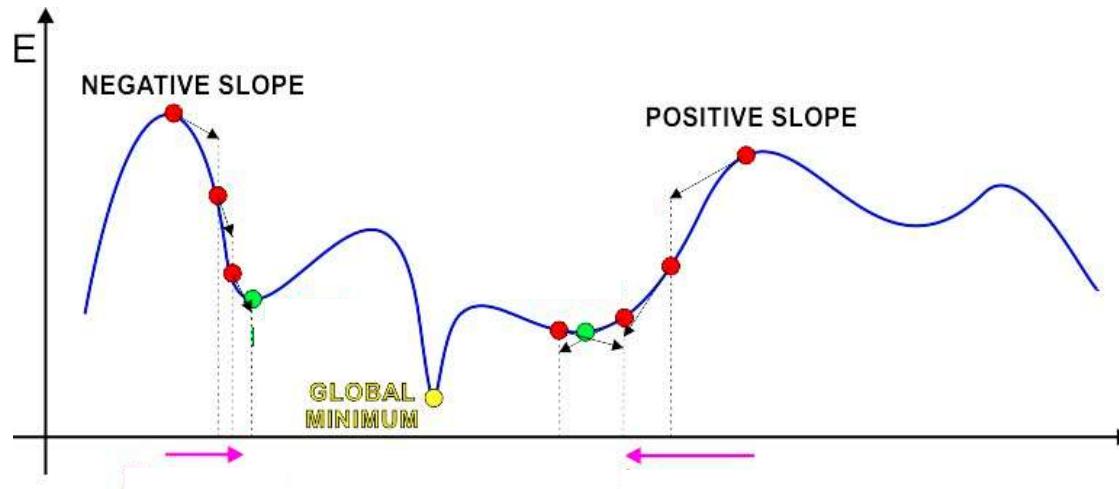
- I.e. minimize the *empirical error* over the drawn samples

Iterative solution



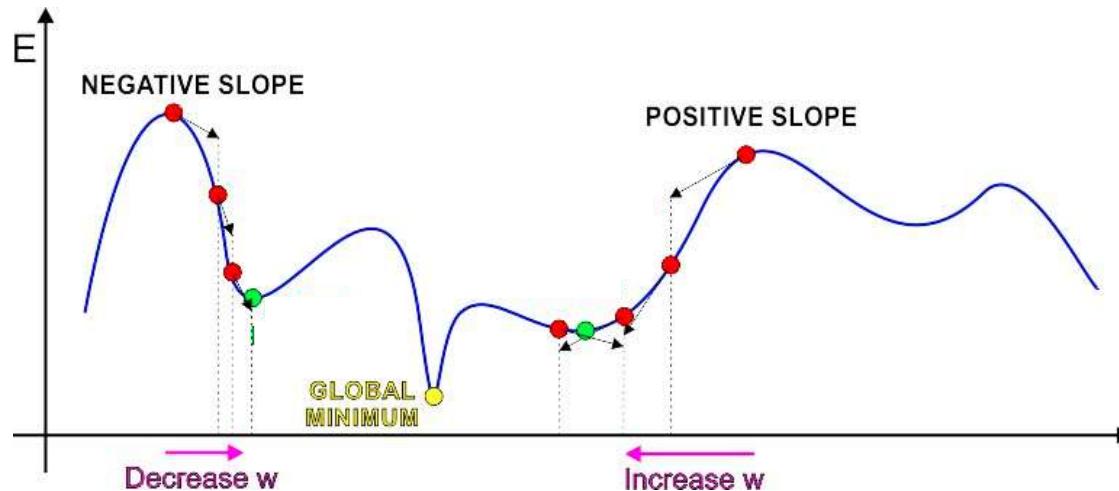
- $\underset{W}{\operatorname{argmin}} \text{Loss}(W)$ does not have a closed form solution
 - Instead we will use an iterative procedure to solve it
- Iterative solutions
 - Start from an initial guess w_0 for the optimal w
 - Update the guess towards a (hopefully) “better” value of $L(w)$
 - Stop when $L(w)$ no longer decreases

The Approach of Gradient Descent



- Iterative solution:
 - Start at some point
 - Find direction in which to shift this point to decrease error
 - This can be found from the derivative of the function
 - A positive derivative → moving left decreases error
 - A negative derivative → moving right decreases error
 - Shift point in this direction

The Approach of Gradient Descent

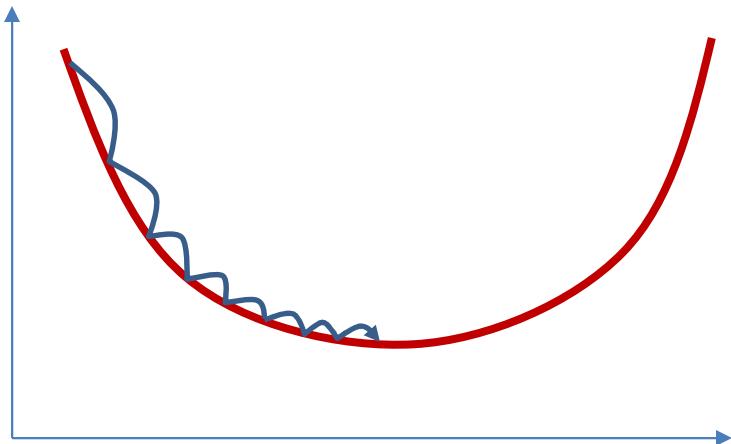


- Iterative solution:
 - Initialize w^0
 - While $L'(w^k) \neq 0$
 - Shift estimate in direction that is expected to decrease the function
 - This is *opposite* to the direction of the derivative (gradient)
 - *The derivative points in the direction of steepest increase of the function*

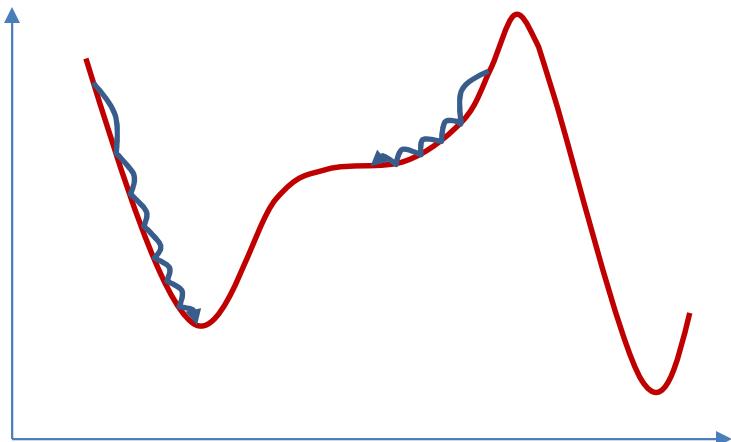
Overall Gradient Descent Algorithm

- Initialize:
 - W^0
 - $k = 0$
- do
 - $W^{k+1} = W^k - \eta^k \nabla_W L(W^k)^T$
 - $k = k + 1$
- while $|L(W^{k+1}) - L(W^k)| > \varepsilon$

Convergence of Gradient Descent



- For appropriate step size, for convex (bowl-shaped) functions gradient descent will always find the minimum.



- For non-convex functions it will find a local minimum or an inflection point

- Returning to our problem..

Problem Statement

- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Minimize the following function

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

w.r.t W

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

Preliminaries

- Before we proceed: the problem setup

Problem Setup: Things to define

- Given a training set of input-output pairs
 $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- What are these input-output pairs?

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

Problem Setup: Things to define

- Given a training set of input-output pairs
 $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- What are these input-output pairs?

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

What is $f()$ and
what are its
parameters W ?

Problem Setup: Things to define

- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- What are these input-output pairs?

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

What is the divergence $div()$?

What is $f()$ and what are its parameters W ?

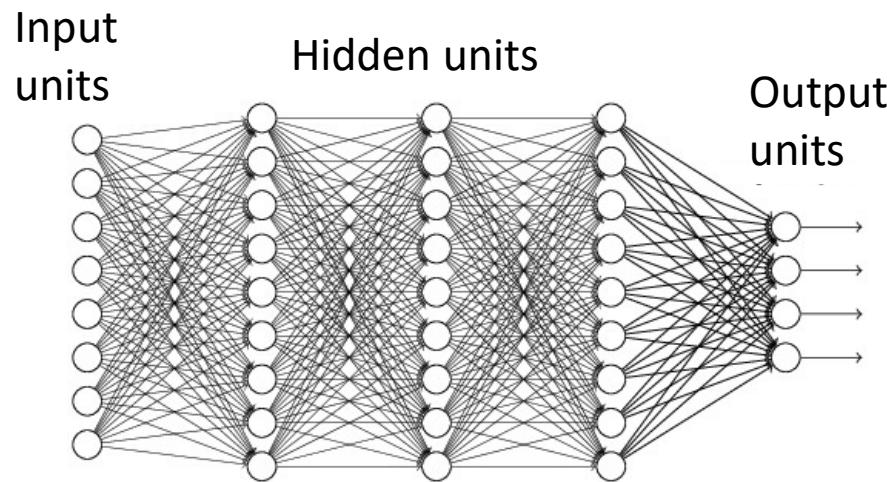
Problem Setup: Things to define

- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Minimize the following function

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

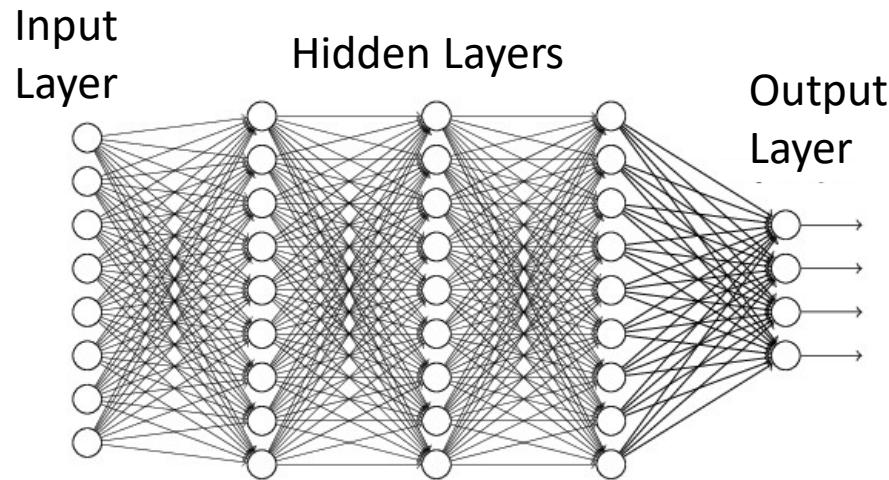
What is $f()$ and
what are its
parameters W ?

What is $f()$? Typical network



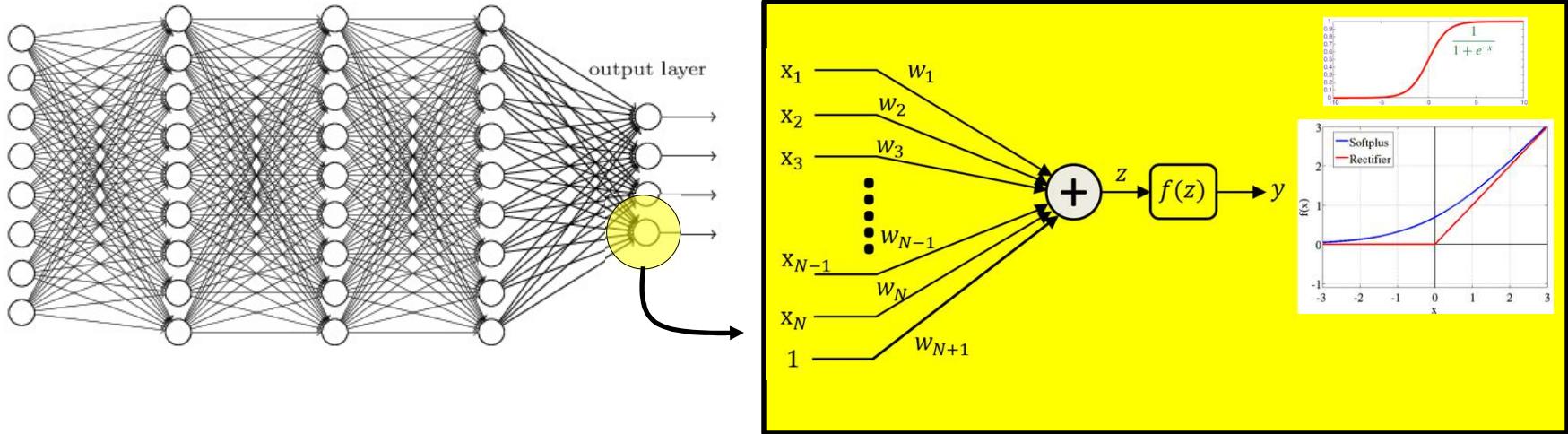
- Multi-layer perceptron
- A *directed* network with a set of inputs and outputs
 - No loops

Typical network



- We assume a “layered” network for simplicity
 - Each “layer” of neurons only gets inputs from the earlier layer(s) and outputs signals only to later layer(s)
 - We will refer to the inputs as the ***input layer***
 - No neurons here – the “layer” simply refers to inputs
 - We refer to the outputs as the ***output layer***
 - Intermediate layers are ***“hidden” layers***

The individual neurons



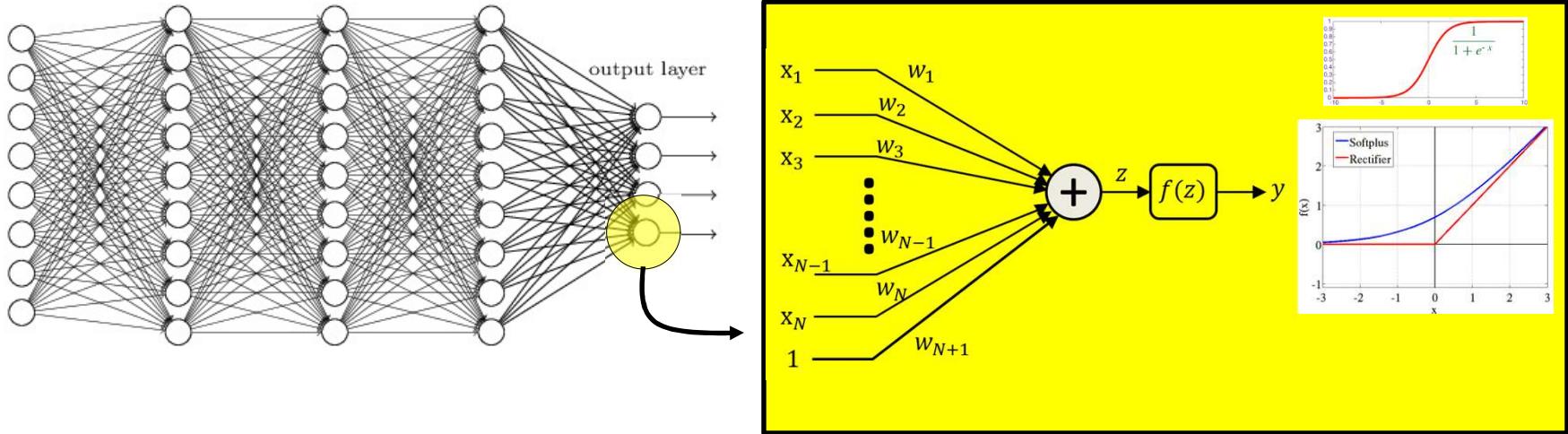
- Individual neurons operate on a set of inputs and produce a single output
 - **Standard setup:** A continuous activation function applied to an affine combination of the inputs

$$y = f \left(\sum_i w_i x_i + b \right)$$

- More generally: *any* differentiable function

$$y = f(x_1, x_2, \dots, x_N; W)$$

The individual neurons



- Individual neurons operate on a set of inputs and produce a single output
 - **Standard setup:** A continuous activation function applied to an affine combination of the input

$$y = f \left(\sum_i w_i x_i + b \right)$$

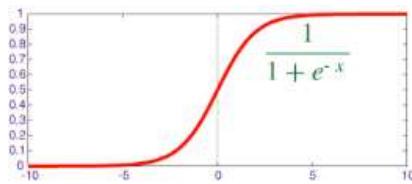
- More generally: *any* differentiable function

$$y = f(x_1, x_2, \dots, x_N; W)$$

We will assume this unless otherwise specified

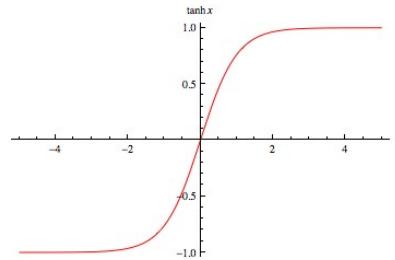
Parameters are weights w_i and bias b

Activations and their derivatives



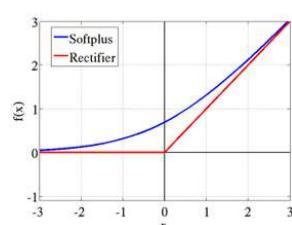
$$f(z) = \frac{1}{1 + \exp(-z)}$$

$$f'(z) = f(z)(1 - f(z))$$



$$f(z) = \tanh(z)$$

$$f'(z) = (1 - f^2(z))$$



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

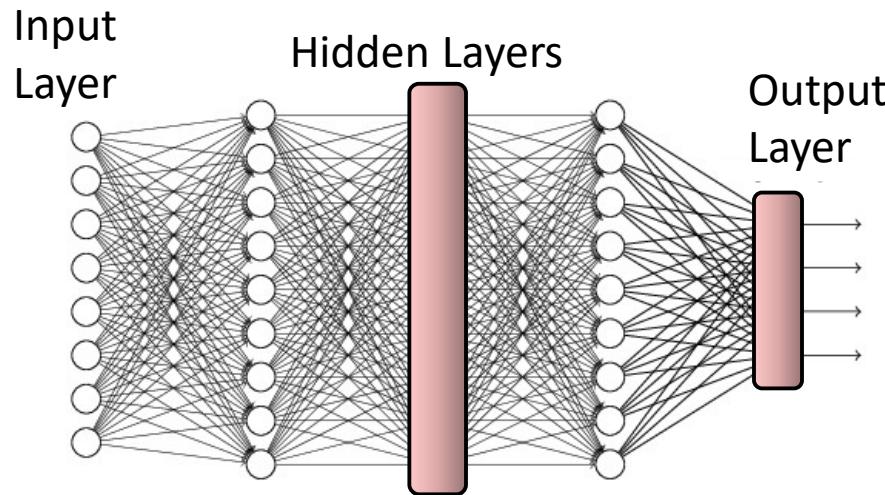
$$f(z) = \log(1 + \exp(z))$$

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

$$f'(z) = \frac{1}{1 + \exp(-z)}$$

- Some popular activation functions and their derivatives

Vector Activations

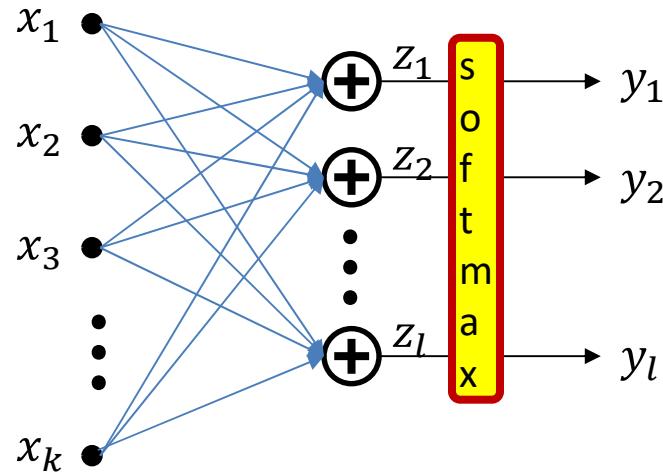


- We can also have neurons that have *multiple coupled outputs*

$$[y_1, y_2, \dots, y_l] = f(x_1, x_2, \dots, x_k; W)$$

- Function $f()$ operates on set of inputs to produce set of outputs
- Modifying a single parameter in W will affect *all* outputs

Vector activation example: Softmax



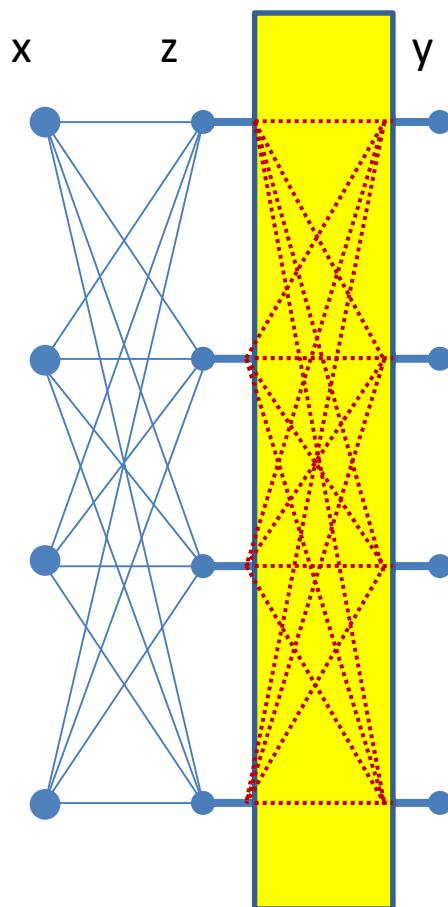
- Example: Softmax *vector* activation

$$z_i = \sum_j w_{ji} x_j + b_i$$

$$y = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Parameters are
weights w_{ji}
and bias b_i

Multiplicative combination: Can be viewed as a case of vector activations



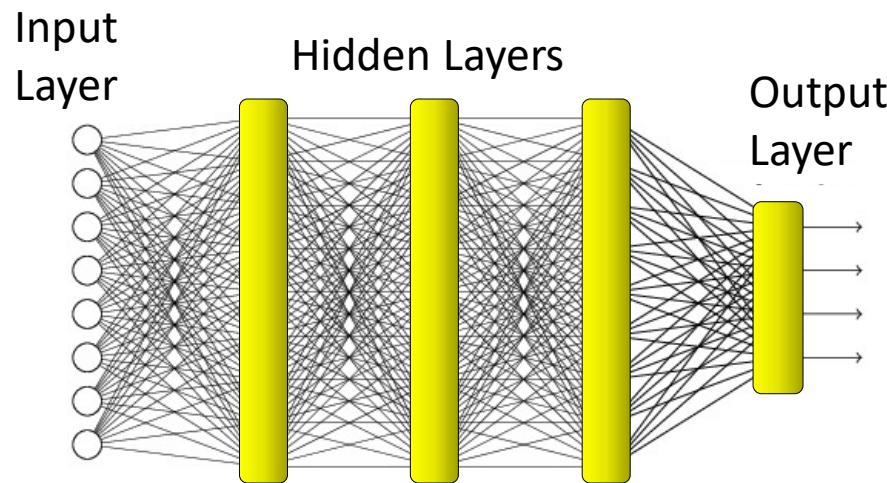
$$z_i = \sum_j w_{ji} x_j + b_i$$

$$y_i = \prod_l (z_l)^{\alpha_{li}}$$

Parameters are
weights w_{ji}
and bias b_i

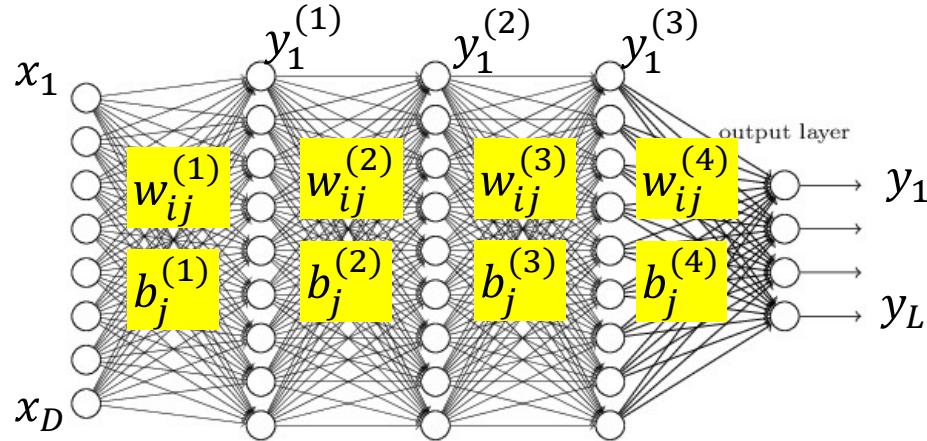
- A layer of multiplicative combination is a special case of vector activation

Typical network



- In a layered network, each layer of perceptrons can be viewed as a single vector activation

Notation



- The input layer is the 0^{th} layer
- We will represent the output of the i -th perceptron of the k^{th} layer as $y_i^{(k)}$
 - **Input to network:** $y_i^{(0)} = x_i$
 - **Output of network:** $y_i = y_i^{(N)}$
- We will represent the weight of the connection between the i -th unit of the $k-1$ th layer and the j th unit of the k -th layer as $w_{ij}^{(k)}$
 - The bias to the j th unit of the k -th layer is $b_j^{(k)}$

Problem Setup: Things to define

- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Minimize the following function

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



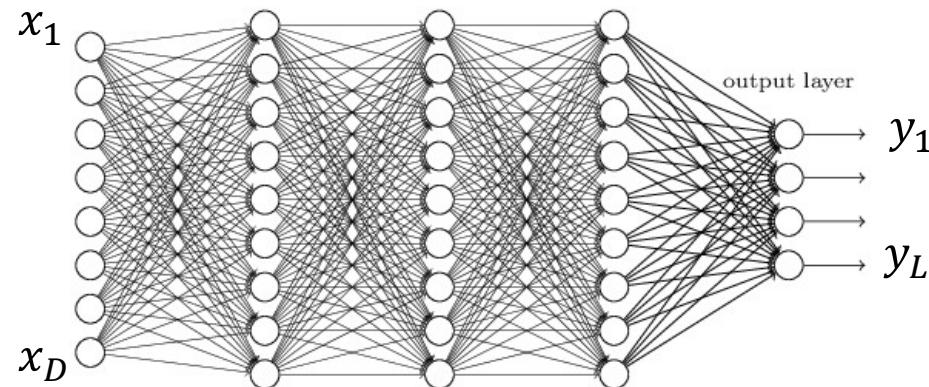
What is $f()$ and
what are its
parameters W ?

Problem Setup: Things to define

- Given a training set of input-output pairs
 $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- What are these input-output pairs?

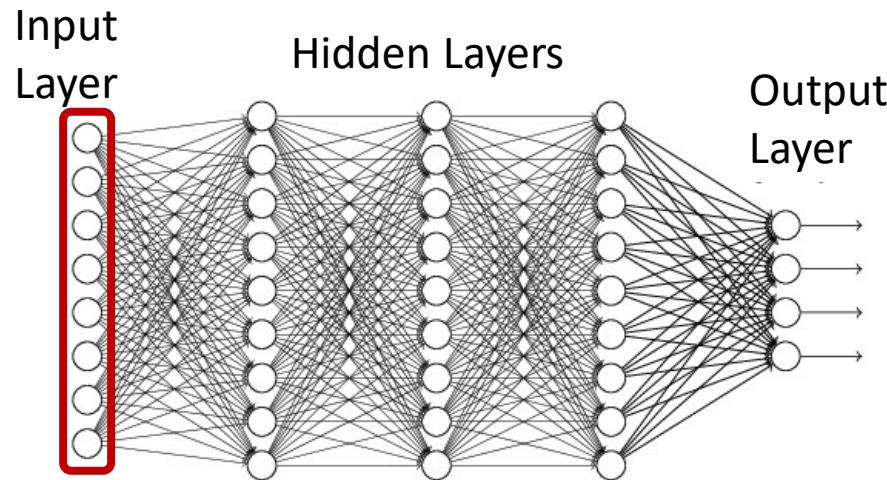
$$Loss(W) = \frac{1}{T} \sum_i div(f(X_i; W), d_i)$$

Input, target output, and actual output: Vector notation



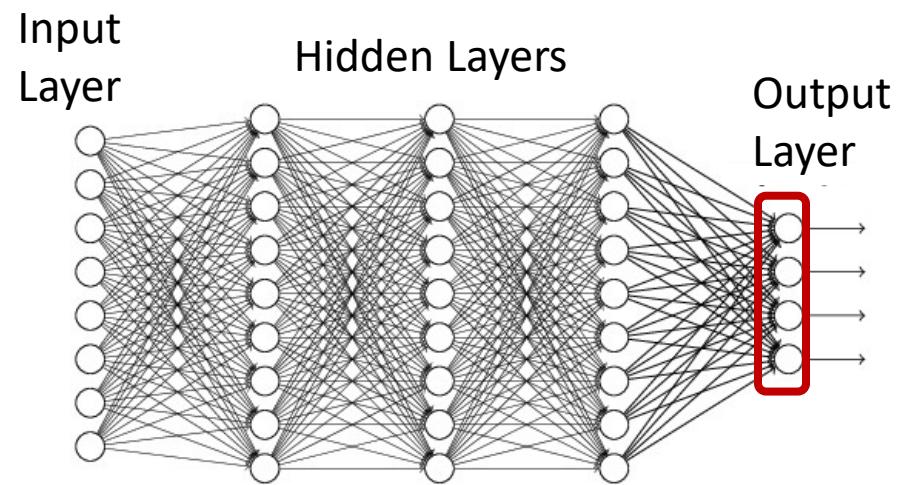
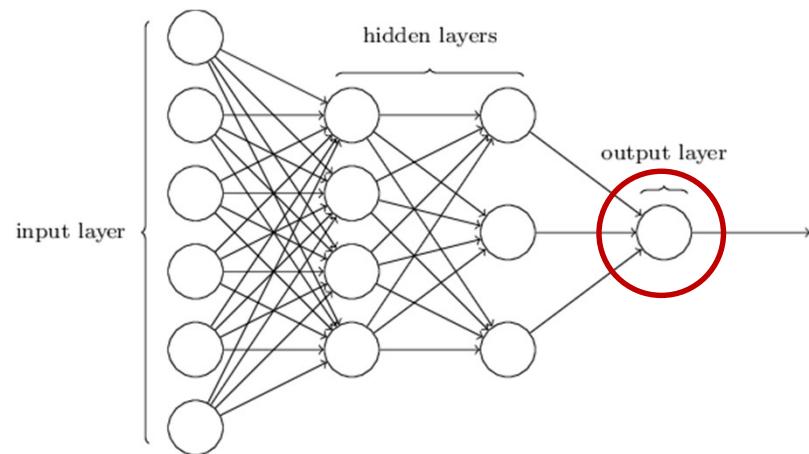
- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- $X_n = [x_{n1}, x_{n2}, \dots, x_{nD}]^T$ is the nth input vector
- $d_n = [d_{n1}, d_{n2}, \dots, d_{nL}]^T$ is the nth desired output
- $Y_n = [y_{n1}, y_{n2}, \dots, y_{nL}]^T$ is the nth vector of *actual* outputs of the network
 - Function of input X_n and network parameters
- We will sometimes drop the first subscript when referring to a *specific* instance

Representing the input



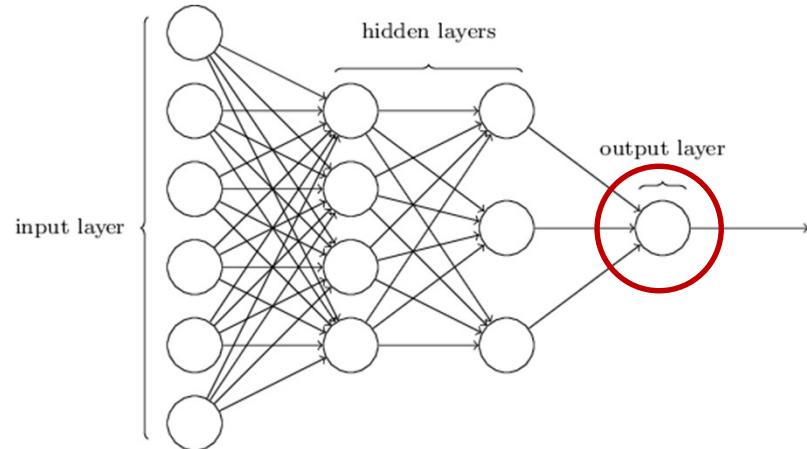
- Vectors of numbers
 - (or may even be just a scalar, if input layer is of size 1)
 - E.g. vector of pixel values
 - E.g. vector of speech features
 - E.g. real-valued vector representing text
 - We will see how this happens later in the course
 - Other real valued vectors

Representing the output



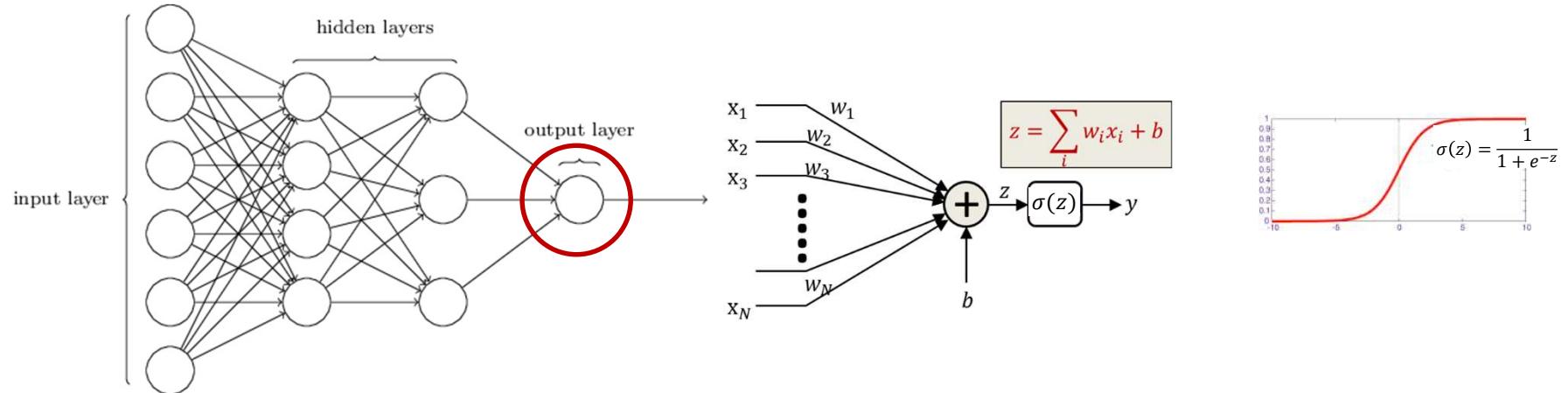
- If the desired *output* is real-valued, no special tricks are necessary
 - Scalar Output : single output neuron
 - $d = \text{scalar} (\text{real value})$
 - Vector Output : as many output neurons as the dimension of the desired output
 - $d = [d_1 \ d_2 \dots \ d_L]$ (vector of real values)

Representing the output



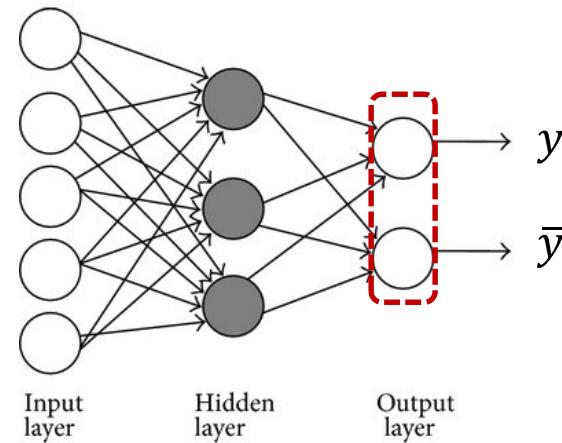
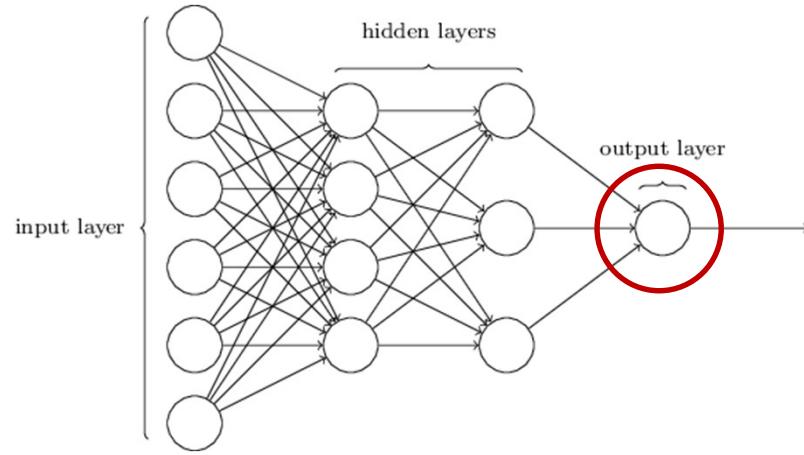
- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
 - 1 = Yes it's a cat
 - 0 = No it's not a cat.

Representing the output



- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
- Output activation: Typically a sigmoid
 - Viewed as the *probability* $P(Y = 1|X)$ of class value 1
 - Indicating the fact that for actual data, in general a feature value X may occur for both classes, but with different probabilities
 - Is differentiable

Representing the output

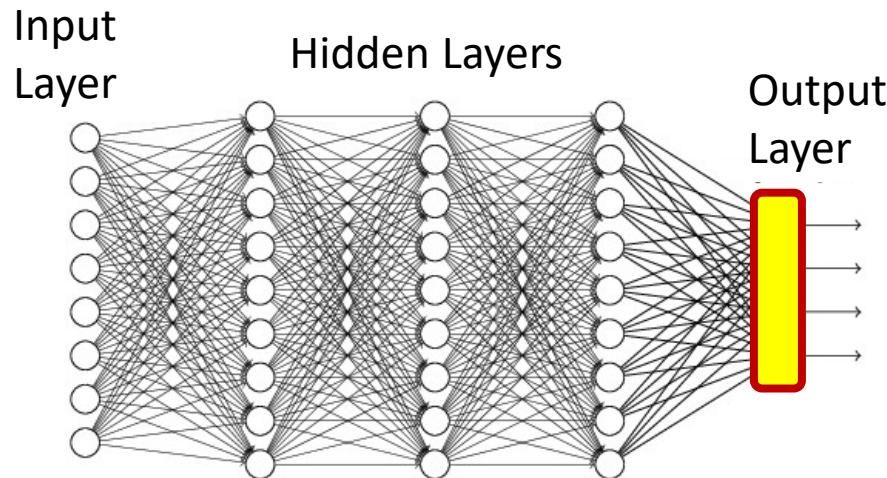


- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
 - 1 = Yes it's a cat
 - 0 = No it's not a cat.
- Sometimes represented by *two* outputs, one representing the desired output, the other representing the *negation* of the desired output
 - Yes: $\rightarrow [1 0]$
 - No: $\rightarrow [0 1]$
- The output explicitly becomes a 2-output softmax

Multi-class output: One-hot representations

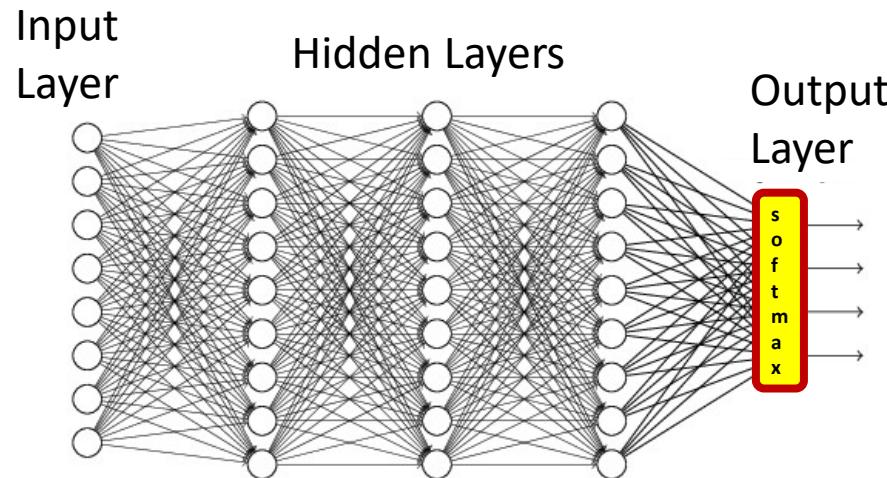
- Consider a network that must distinguish if an input is a cat, a dog, a camel, a hat, or a flower
- We can represent this set as the following vector, with the classes arranged in a chosen order:
$$[\text{cat } \text{dog } \text{camel } \text{hat } \text{flower}]^T$$
- For inputs of each of the five classes the desired output is:
 - cat: $[1 \ 0 \ 0 \ 0 \ 0]^T$
 - dog: $[0 \ 1 \ 0 \ 0 \ 0]^T$
 - camel: $[0 \ 0 \ 1 \ 0 \ 0]^T$
 - hat: $[0 \ 0 \ 0 \ 1 \ 0]^T$
 - flower: $[0 \ 0 \ 0 \ 0 \ 1]^T$
- For an input of any class, we will have a five-dimensional vector output with four zeros and a single 1 at the position of that class
- This is a *one hot vector*

Multi-class networks



- For a multi-class classifier with N classes, the one-hot representation will have N binary target outputs
 - The desired output d is an N -dimensional binary vector
- The neural network's output too must ideally be binary ($N-1$ zeros and a single 1 in the right place)
- More realistically, it will be a probability vector
 - N probability values that sum to 1.

Multi-class classification: Output



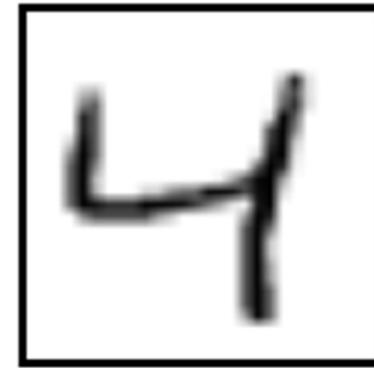
- Softmax *vector* activation is often used at the output of multi-class classifier nets

$$z_i = \sum_j w_{ji}^{(n)} y_j^{(n-1)}$$

$$y_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- This can be viewed as the probability $y_i = P(\text{class} = i | X)$

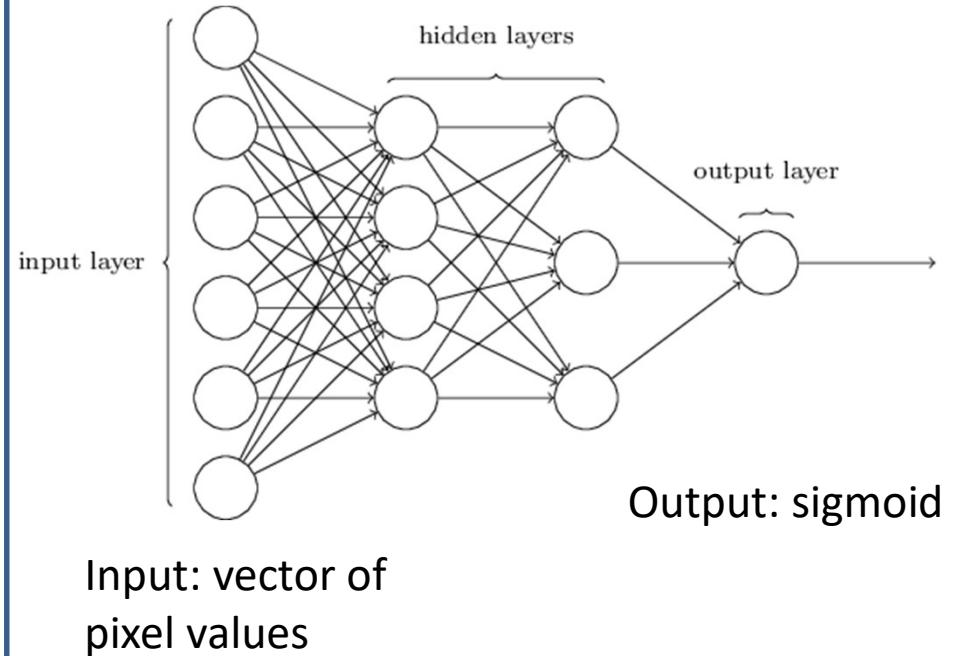
Inputs and outputs: Typical Problem Statement



- We are given a number of “training” data instances
- E.g. images of digits, along with information about which digit the image represents
- Tasks:
 - Binary recognition: Is this a “2” or not
 - Multi-class recognition: Which digit is this?

Typical Problem statement: binary classification

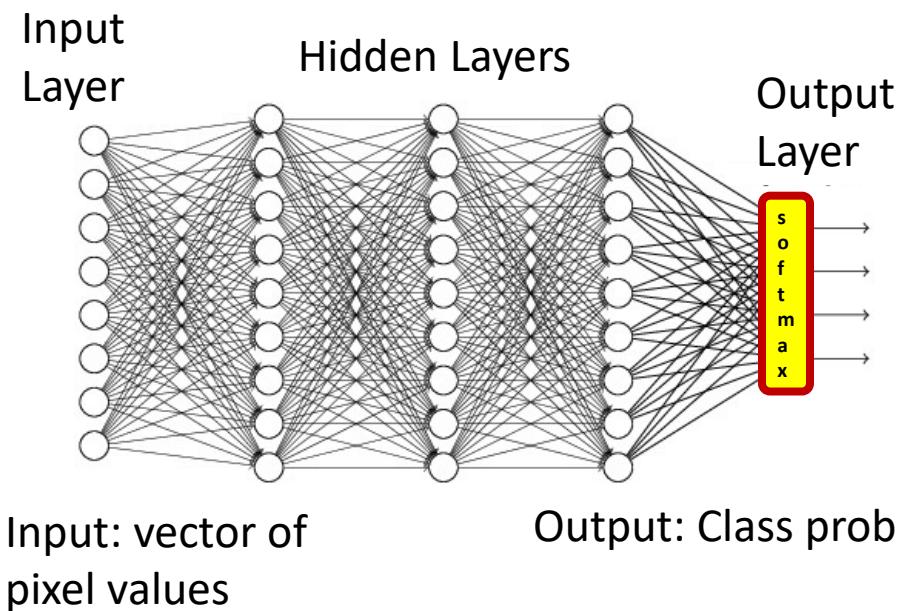
Training data	
(Σ, 0)	(2, 1)
(2, 1)	(4, 0)
(0, 0)	(2, 1)



- Given, many positive and negative examples (training data),
 - learn all weights such that the network does the desired job

Typical Problem statement: multiclass classification

Training data	
(Σ, 5)	(2, 2)
(2, 2)	(4, 4)
(0, 0)	(2, 2)



- Given, many positive and negative examples (training data),
 - learn all weights such that the network does the desired job

Problem Setup: Things to define

- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Minimize the following function

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

What is the
divergence $div()$?

Problem Setup: Things to define

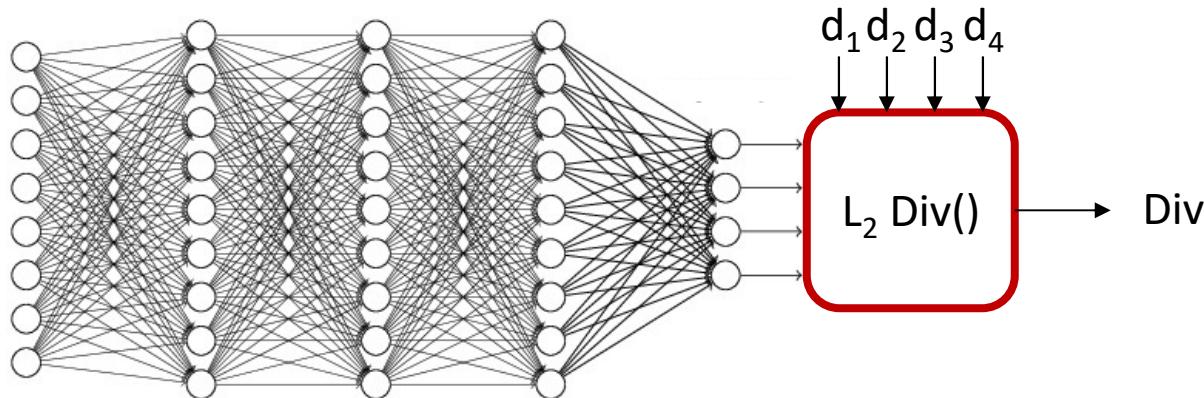
- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Minimize the following function

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

What is the divergence $div()$?

Note: For $Loss(W)$ to be differentiable w.r.t W , $div()$ must be differentiable

Examples of divergence functions



- For real-valued output vectors, the (scaled) L_2 divergence is popular

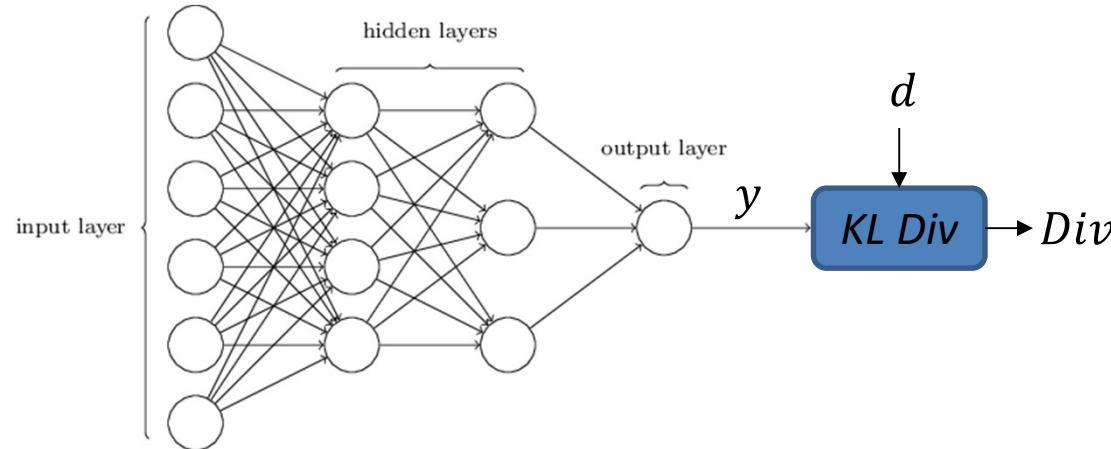
$$Div(Y, d) = \frac{1}{2} \|Y - d\|^2 = \frac{1}{2} \sum_i (y_i - d_i)^2$$

- Squared Euclidean distance between true and desired output
- Note: this is differentiable

$$\frac{dDiv(Y, d)}{dy_i} = (y_i - d_i)$$

$$\nabla_Y Div(Y, d) = [y_1 - d_1, y_2 - d_2, \dots]$$

For binary classifier



- For binary classifier with scalar output, $Y \in (0,1)$, d is 0/1, the Kullback Leibler (KL) divergence between the probability distribution $[Y, 1 - Y]$ and the ideal output probability $[d, 1 - d]$ is popular

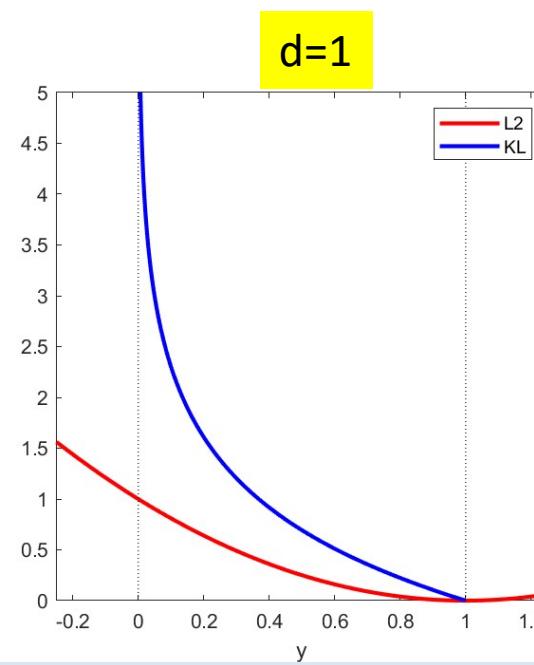
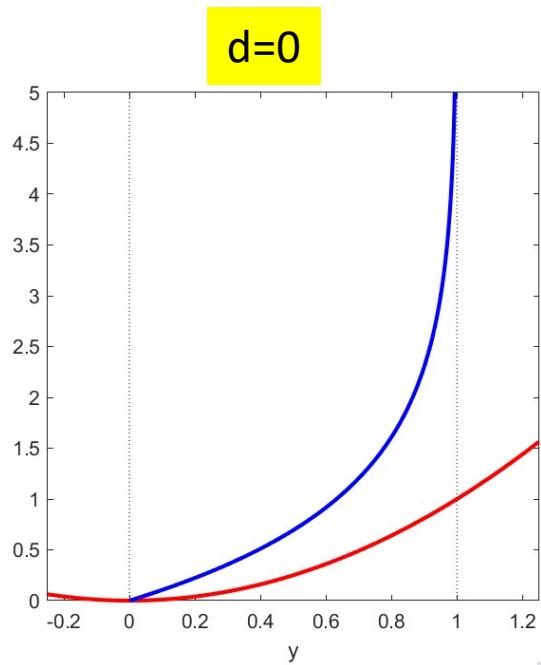
$$Div(Y, d) = -d\log Y - (1 - d)\log(1 - Y)$$

- Minimum when $d = Y$

- **Derivative**

$$\frac{dDiv(Y, d)}{dY} = \begin{cases} -\frac{1}{Y} & \text{if } d = 1 \\ \frac{1}{1 - Y} & \text{if } d = 0 \end{cases}$$

KL vs L2

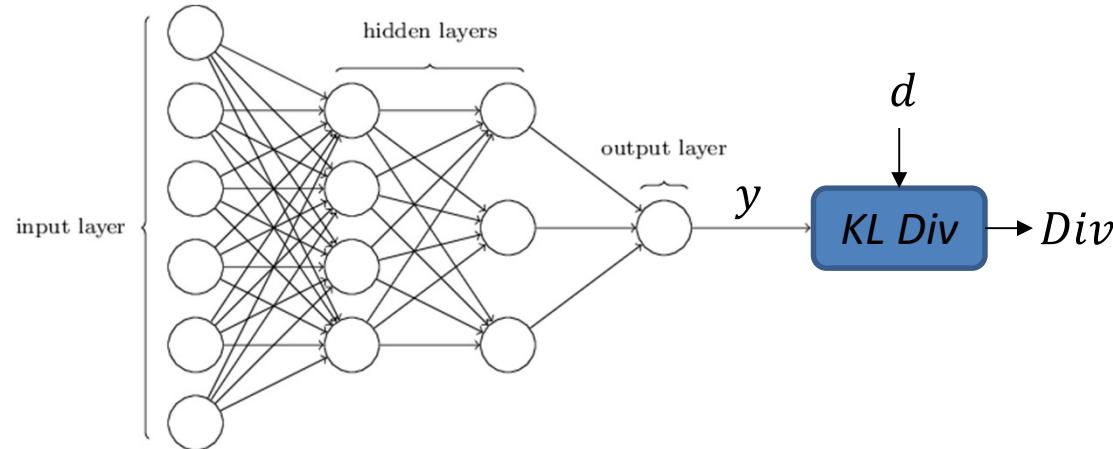


$$L2(Y, d) = (y - d)^2$$

$$KL(Y, d) = -d \log Y - (1 - d) \log(1 - Y)$$

- Both KL and L2 have a minimum when y is the target value of d
- KL rises much more steeply away from d
 - Encouraging faster convergence of gradient descent
- The derivative of KL is *not* equal to 0 at the minimum
 - It is 0 for L2, though

For binary classifier



- For binary classifier with scalar output, $Y \in (0,1)$, d is 0/1, the Kullback Leibler (KL) divergence between the probability distribution $[Y, 1 - Y]$ and the ideal output probability $[d, 1 - d]$ is popular

$$Div(Y, d) = -d\log Y - (1 - d)\log(1 - Y)$$

- Minimum when $d = Y$

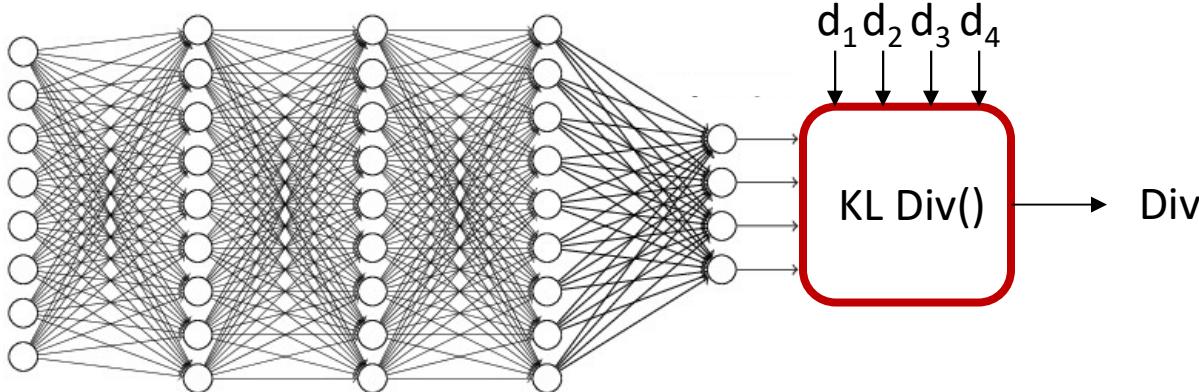
- Derivative

$$\frac{dDiv(Y, d)}{dY} = \begin{cases} -\frac{1}{Y} & \text{if } d = 1 \\ \frac{1}{1 - Y} & \text{if } d = 0 \end{cases}$$

Note: when $y = d$ the derivative is *not* 0

*Even though div() = 0
(minimum) when $y = d$*

For multi-class classification



- Desired output d is a one hot vector $[0 \ 0 \dots 1 \ \dots 0 \ 0 \ 0]$ with the 1 in the c -th position (for class c)
- Actual output will be probability distribution $[y_1, y_2, \dots]$
- The KL divergence between the desired one-hot output and actual output:

$$Div(Y, d) = \sum_i d_i \log d_i - \sum_i d_i \log y_i = -\log y_c$$

– Note $\sum_i d_i \log d_i = 0$ for one-hot $d \Rightarrow Div(Y, d) = -\sum_i d_i \log y_i$

- Derivative

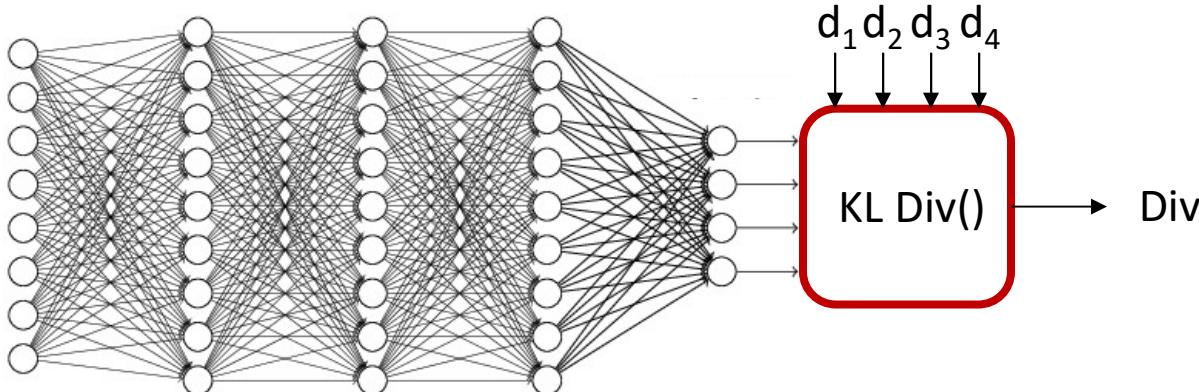
$$\frac{dDiv(Y, d)}{dY_i} = \begin{cases} -\frac{1}{y_c} & \text{for the } c-\text{th component} \\ 0 & \text{for remaining component} \end{cases}$$

$$\nabla_Y Div(Y, d) = \left[0 \ 0 \ \dots \frac{-1}{y_c} \dots 0 \ 0 \right]$$

The slope is negative
w.r.t. y_c

Indicates *increasing* y_c
will *reduce* divergence

For multi-class classification



- Desired output d is a one hot vector $[0 \ 0 \dots 1 \ \dots 0 \ 0 \ 0]$ with the 1 in the c -th position (for class c)
- Actual output will be probability distribution $[y_1, y_2, \dots]$
- The KL divergence between the desired one-hot output and actual output:

$$Div(Y, d) = - \sum_i d_i \log y_i = - \log y_c$$

- Derivative

$$\frac{dDiv(Y, d)}{dy_i} = \begin{cases} -\frac{1}{y_c} & \text{for the } c-\text{th component} \\ 0 & \text{for remaining component} \end{cases}$$

$$\nabla_Y Div(Y, d) = \left[0 \ 0 \ \dots \frac{-1}{y_c} \dots 0 \ 0 \right]$$

The slope is negative w.r.t. y_c

Indicates *increasing* y_c will *reduce* divergence

Note: when $y = d$ the derivative is *not* 0

Even though div() = 0 (minimum) when $y = d$

KL divergence vs cross entropy

- KL divergence between d and y :

$$KL(Y, d) = \sum_i d_i \log d_i - \sum_i d_i \log y_i$$

- *Cross-entropy* between d and y :

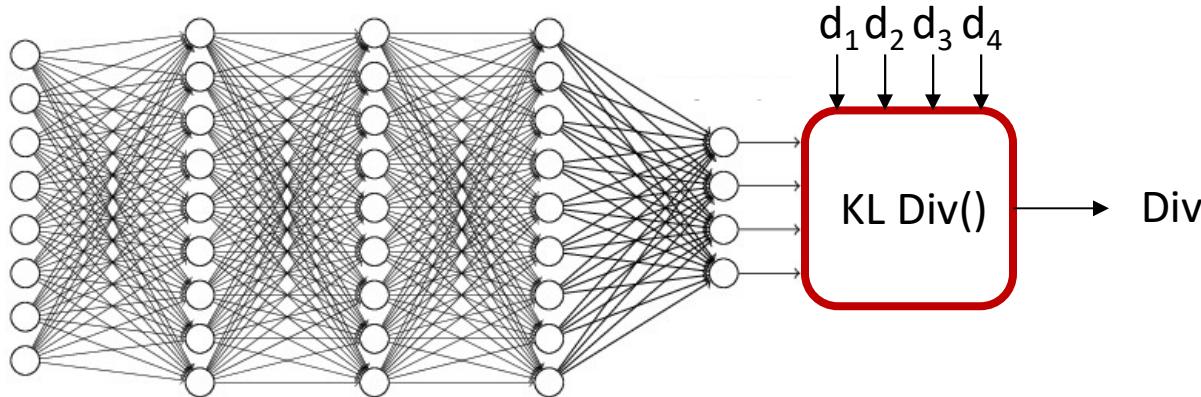
$$Xent(Y, d) = - \sum_i d_i \log y_i$$

- The cross entropy is merely the KL - entropy of d

$$Xent(Y, d) = KL(Y, d) - \sum_i d_i \log d_i = KL(Y, d) - H(d)$$

- The W that minimizes cross-entropy will minimize the KL divergence
 - since d is the desired output and does not depend on the network, $H(d)$ does not depend on the net
 - In fact, for one-hot d , $H(d) = 0$ (and $KL = Xent$)
- We will generally minimize to the *cross-entropy* loss rather than the KL divergence
 - The Xent is *not* a divergence, and although it attains its minimum when $y = d$, its minimum value is not 0

“Label smoothing”



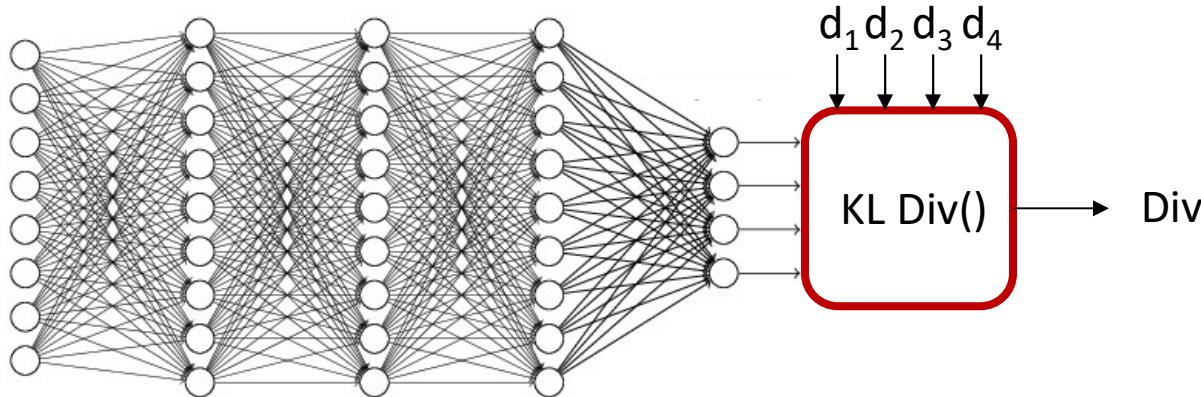
- It is sometimes useful to set the target output to $[\epsilon \ \epsilon \dots (1 - (K - 1)\epsilon) \dots \epsilon \ \epsilon \ \epsilon]$ with the value $1 - (K - 1)\epsilon$ in the c -th position (for class c) and ϵ elsewhere for some small ϵ
 - “Label smoothing” -- aids gradient descent
- The KL divergence remains:

$$Div(Y, d) = \sum_i d_i \log d_i - \sum_i d_i \log y_i$$

- Derivative

$$\frac{dDiv(Y, d)}{dY_i} = \begin{cases} -\frac{1 - (K - 1)\epsilon}{y_c} & \text{for the } c\text{-th component} \\ -\frac{\epsilon}{y_i} & \text{for remaining components} \end{cases}$$

“Label smoothing”



- It is sometimes useful to set the target output to $[\epsilon \ \epsilon \dots (1 - (K - 1)\epsilon) \dots \epsilon \ \epsilon \ \epsilon]$ with the value $1 - (K - 1)\epsilon$ in the c -th position (for class c) and ϵ elsewhere for some small ϵ
 - “Label smoothing” -- aids gradient descent
- The KL divergence remains:

$$Div(Y, d) = \sum_i d_i \log d_i - \sum_i d_i \log y_i$$

- Derivative

$$\frac{dDiv(Y, d)}{dY_i} = \begin{cases} -\frac{1 - (K - 1)\epsilon}{y_c} & \text{for the } c\text{-th component} \\ -\frac{\epsilon}{y_i} & \text{for remaining components} \end{cases}$$

Negative derivatives encourage increasing the probabilities of all classes, including incorrect classes! (Seems wrong, no?)

Problem Setup: Things to define

- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Minimize the following function

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

ALL TERMS HAVE BEEN DEFINED

Story so far

- Neural nets are universal approximators
- Neural networks are trained to approximate functions by adjusting their parameters to minimize the average divergence between their actual output and the desired output at a set of “training instances”
 - Input-output samples from the function to be learned
 - The average divergence is the “Loss” to be minimized
- To train them, several terms must be defined
 - The network itself
 - The manner in which inputs are represented as numbers
 - The manner in which outputs are represented as numbers
 - As numeric vectors for real predictions
 - As one-hot vectors for classification functions
 - The divergence function that computes the error between actual and desired outputs
 - L2 divergence for real-valued predictions
 - KL divergence for classifiers

Problem Setup

- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- The divergence on the i^{th} instance is $\text{div}(Y_i, d_i)$
 - $Y_i = f(X_i; W)$
- The loss

$$\text{Loss} = \frac{1}{T} \sum_i \text{div}(Y_i, d_i)$$

- Minimize Loss w.r.t $\{w_{ij}^{(k)}, b_j^{(k)}\}$

Recap: Gradient Descent Algorithm

- Initialize:
 - W^0
 - $k = 0$
- To minimize any function $L(W)$ w.r.t W
- do
 - $W^{k+1} = W^k - \eta^k \nabla L(W^k)^T$
 - $k = k + 1$
- while $|L(W^k) - L(W^{k-1})| > \varepsilon$

Recap: Gradient Descent Algorithm

- In order to minimize $L(W)$ w.r.t. W
- Initialize:
 - W^0
 - $k = 0$

- do
 - For every component i
 - $W_i^{k+1} = W_i^k - \eta^k \frac{\partial L}{\partial W_i}$ Explicitly stating it by component
 - $k = k + 1$
- while $|L(W^k) - L(W^{k-1})| > \varepsilon$

Training Neural Nets through Gradient Descent

Total training Loss:

$$Loss = \frac{1}{T} \sum_t Div(\mathbf{Y}_t, \mathbf{d}_t)$$

- 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

Assuming the bias is also represented as a weight

Training Neural Nets through Gradient Descent

Total training Loss:

$$Loss = \frac{1}{T} \sum_t Div(\mathbf{Y}_t, \mathbf{d}_t)$$

- Gradient descent algorithm:
- Initialize all weights $\{w_{ij}^{(k)}\}$
- 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 Err has converged

Assuming the bias is also represented as a weight

The derivative

Total training Loss:

$$Loss = \frac{1}{T} \sum_t Div(\mathbf{Y}_t, \mathbf{d}_t)$$

- Computing the derivative

Total derivative:

$$\frac{dLoss}{dw_{i,j}^{(k)}} = \frac{1}{T} \sum_t \frac{dDiv(\mathbf{Y}_t, \mathbf{d}_t)}{dw_{i,j}^{(k)}}$$

Training by gradient descent

- Initialize all weights $\{w_{ij}^{(k)}\}$
- Do:
 - For all i, j, k , initialize $\frac{dLoss}{dw_{i,j}^{(k)}} = 0$
 - For all $t = 1:T$
 - For every layer k for all i, j :
 - Compute $\frac{dDiv(\mathbf{Y}_t, \mathbf{d}_t)}{dw_{i,j}^{(k)}}$
 - $\frac{dLoss}{dw_{i,j}^{(k)}} += \frac{dDiv(\mathbf{Y}_t, \mathbf{d}_t)}{dw_{i,j}^{(k)}}$
 - For every layer k for all i, j :
$$w_{i,j}^{(k)} = w_{i,j}^{(k)} - \frac{\eta}{T} \frac{dLoss}{dw_{i,j}^{(k)}}$$
 - Until Err has converged

The derivative

Total training Loss:

$$Loss = \frac{1}{T} \sum_t Div(\mathbf{Y}_t, \mathbf{d}_t)$$

Total derivative:

$$\frac{dLoss}{dw_{i,j}^{(k)}} = \frac{1}{T} \sum_t \frac{dDiv(\mathbf{Y}_t, \mathbf{d}_t)}{dw_{i,j}^{(k)}}$$

- So we must first figure out how to compute the derivative of divergences of individual training inputs

Calculus Refresher: Basic rules of calculus

For any differentiable function

$$y = f(x)$$

with derivative

$$\frac{dy}{dx}$$

the following must hold for sufficiently small Δx  $\Delta y \approx \frac{dy}{dx} \Delta x$

For any differentiable function

$$y = f(x_1, x_2, \dots, x_M)$$

with partial derivatives

$$\frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, \dots, \frac{\partial y}{\partial x_M}$$

the following must hold for sufficiently small $\Delta x_1, \Delta x_2, \dots, \Delta x_M$

Both by the definition
 $\Delta y = \nabla f \Delta x$

$$y \approx \frac{\partial y}{\partial x_1} \Delta x_1 + \frac{\partial y}{\partial x_2} \Delta x_2 + \dots + \frac{\partial y}{\partial x_M} \Delta x_M$$

Calculus Refresher: Chain rule

For any nested function $y = f(g(x))$

$$\frac{dy}{dx} = \frac{df}{dg(x)} \frac{dg(x)}{dx}$$

Check - we can confirm that : $\Delta y = \frac{dy}{dx} \Delta x$

$$z = g(x) \rightarrow \Delta z = \frac{dg(x)}{dx} \Delta x$$

$$y = f(z) \rightarrow \Delta y = \frac{df}{dz} \Delta z = \frac{df}{dg(x)} \frac{dg(x)}{dx} \Delta x$$

Calculus Refresher: Distributed Chain rule

$$y = f(g_1(x), g_2(x), \dots, g_M(x))$$

$$\frac{dy}{dx} = \frac{\partial f}{\partial g_1(x)} \frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)} \frac{dg_2(x)}{dx} + \dots + \frac{\partial f}{\partial g_M(x)} \frac{dg_M(x)}{dx}$$

Check: $\Delta y = \frac{dy}{dx} \Delta x$ Let $z_i = g_i(x)$

$$\Delta y = \frac{\partial f}{\partial z_1} \Delta z_1 + \frac{\partial f}{\partial z_2} \Delta z_2 + \dots + \frac{\partial f}{\partial z_M} \Delta z_M$$

$$\Delta y = \frac{\partial f}{\partial z_1} \frac{dz_1}{dx} \Delta x + \frac{\partial f}{\partial z_2} \frac{dz_2}{dx} \Delta x + \dots + \frac{\partial f}{\partial z_M} \frac{dz_M}{dx} \Delta x$$

$$\Delta y = \left(\frac{\partial f}{\partial g_1(x)} \frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)} \frac{dg_2(x)}{dx} + \dots + \frac{\partial f}{\partial g_M(x)} \frac{dg_M(x)}{dx} \right) \Delta x$$



Calculus Refresher: Distributed Chain rule

$$y = f(g_1(x), g_2(x), \dots, g_M(x))$$

$$\frac{dy}{dx} = \frac{\partial f}{\partial g_1(x)} \frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)} \frac{dg_2(x)}{dx} + \dots + \frac{\partial f}{\partial g_M(x)} \frac{dg_M(x)}{dx}$$

Check: $\Delta y = \frac{dy}{dx} \Delta x$

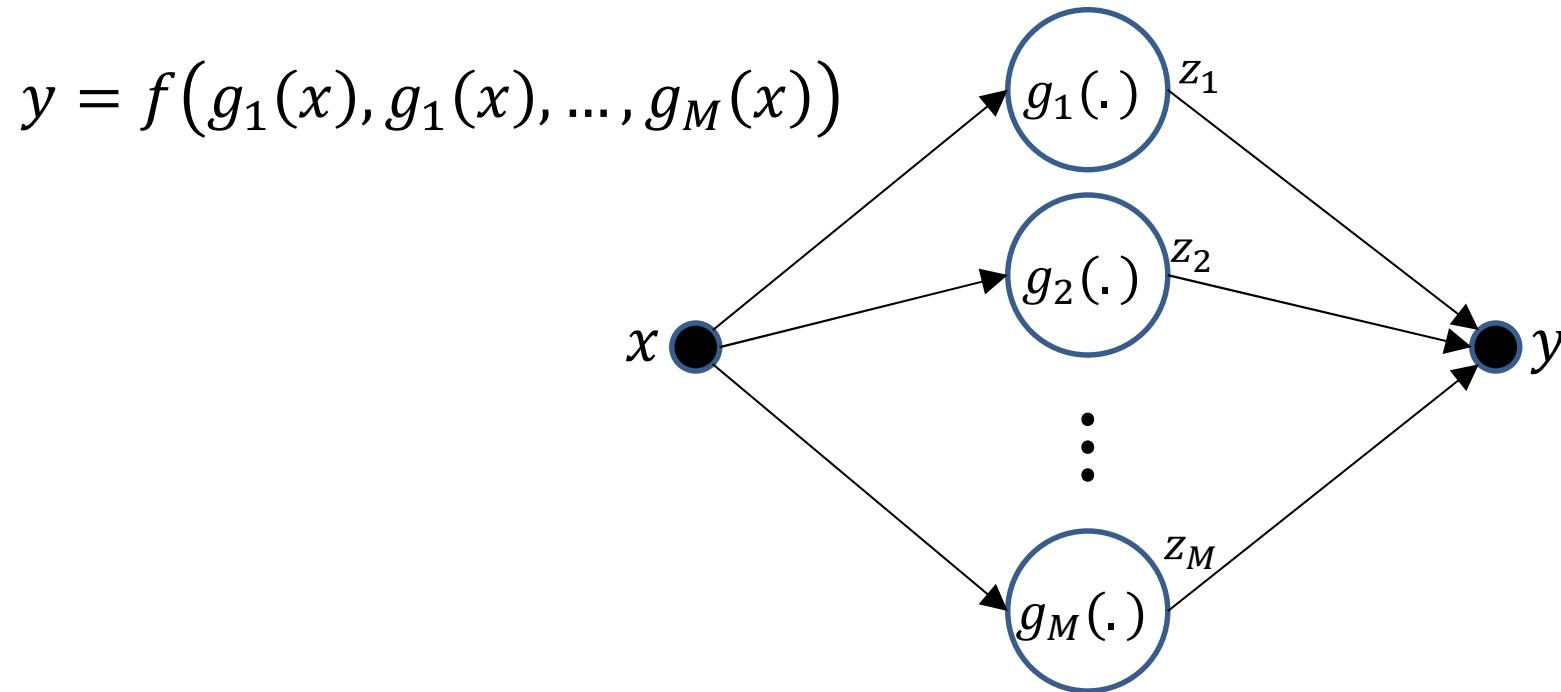
$$\Delta y = \frac{\partial f}{\partial g_1(x)} \Delta g_1(x) + \frac{\partial f}{\partial g_2(x)} \Delta g_2(x) + \dots + \frac{\partial f}{\partial g_M(x)} \Delta g_M(x)$$

$$\Delta y = \frac{\partial f}{\partial g_1(x)} \frac{dg_1(x)}{dx} \Delta x + \frac{\partial f}{\partial g_2(x)} \frac{dg_2(x)}{dx} \Delta x + \dots + \frac{\partial f}{\partial g_M(x)} \frac{dg_M(x)}{dx} \Delta x$$

$$\Delta y = \left(\frac{\partial f}{\partial g_1(x)} \frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)} \frac{dg_2(x)}{dx} + \dots + \frac{\partial f}{\partial g_M(x)} \frac{dg_M(x)}{dx} \right) \Delta x$$

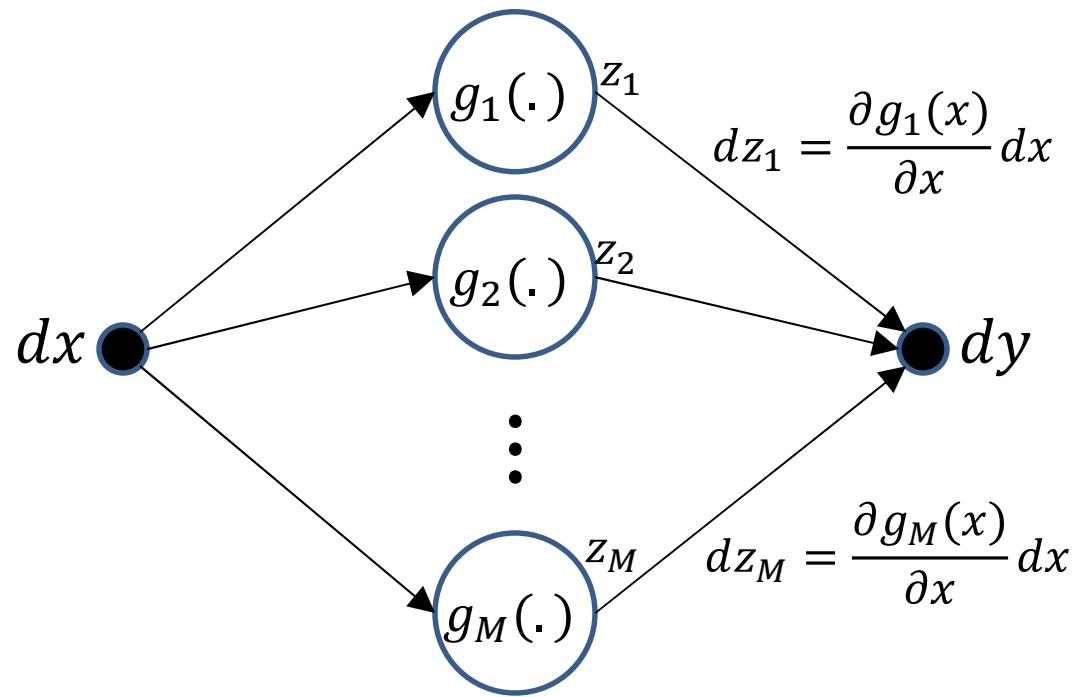
✓ 68

Distributed Chain Rule: Influence Diagram



- x affects y through each of $g_1 \dots g_M$

Distributed Chain Rule: Influence Diagram

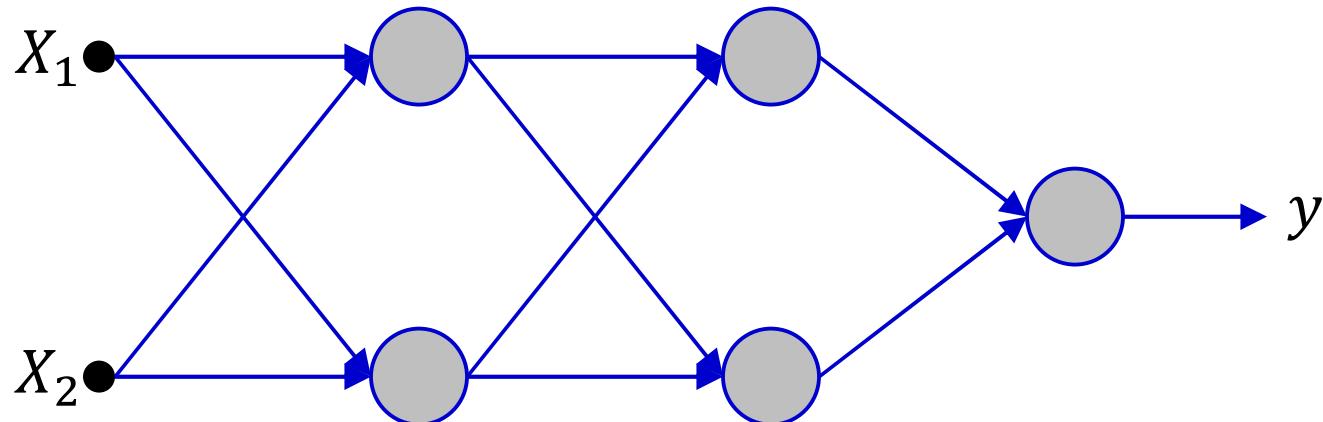


- Small perturbations in x cause small perturbations in each of $g_1 \dots g_M$, each of which individually additively perturbs y

Returning to our problem

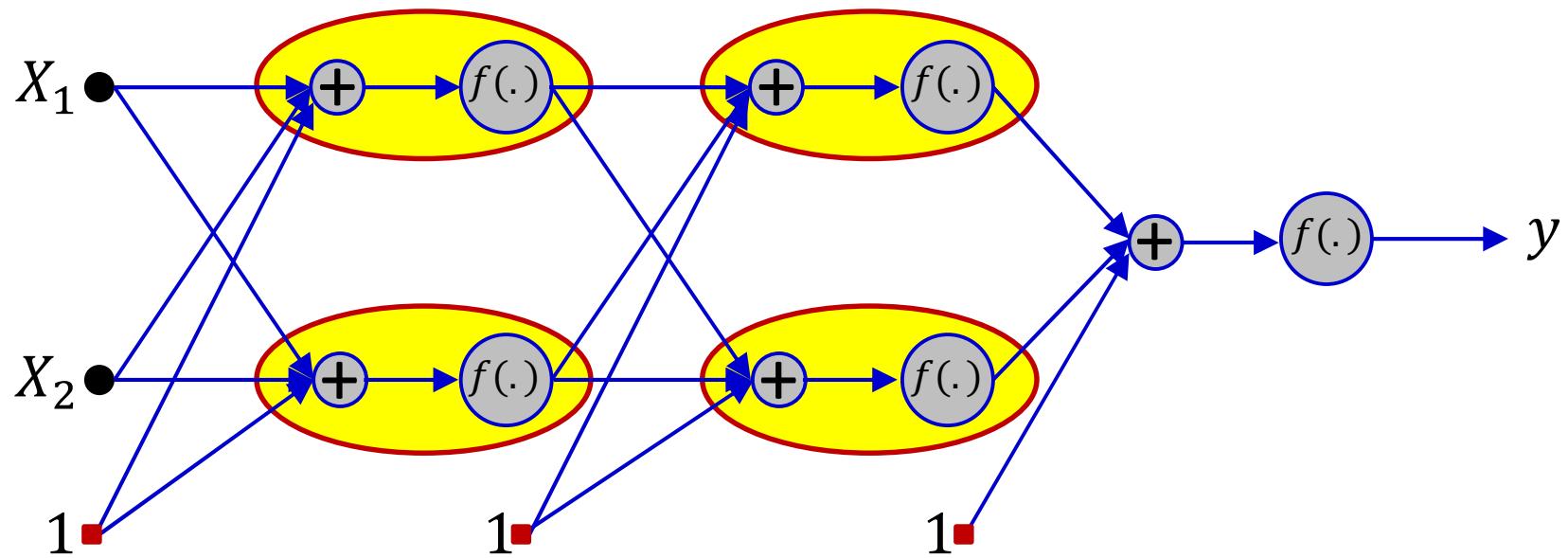
- How to compute $\frac{d\text{Div}(\mathbf{Y}, \mathbf{d})}{dw_{i,j}^{(k)}}$

A first closer look at the network



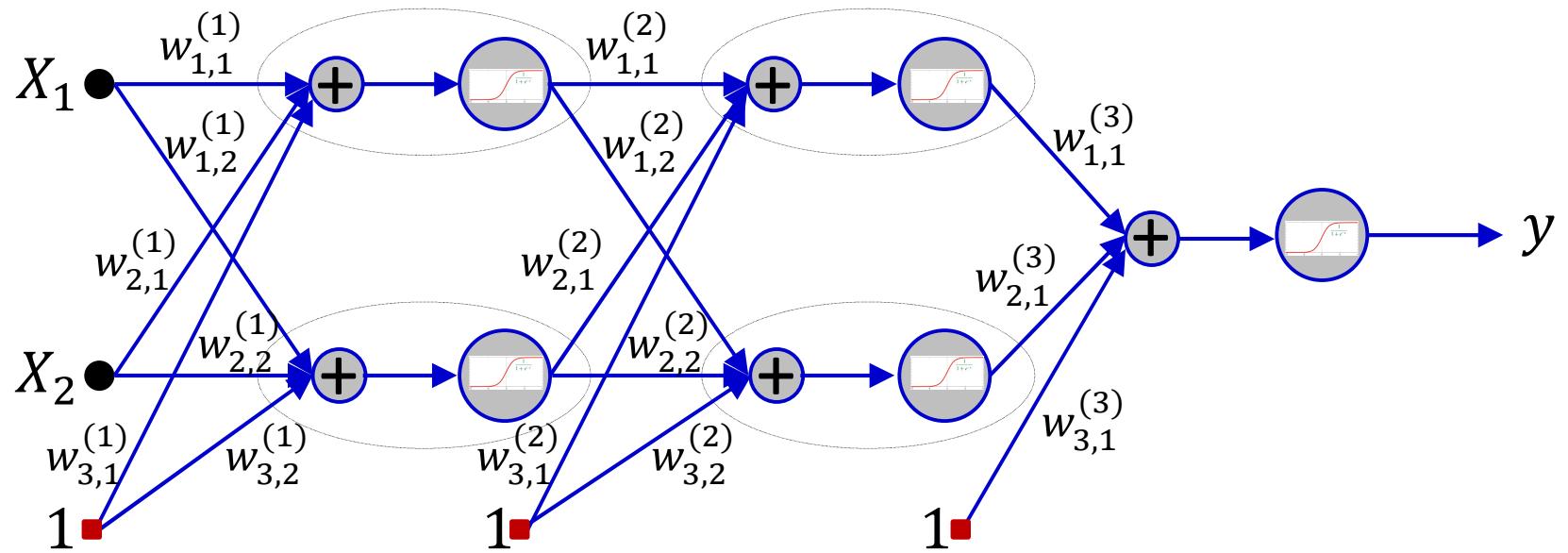
- Showing a tiny 2-input network for illustration
 - Actual network would have many more neurons and inputs

A first closer look at the network



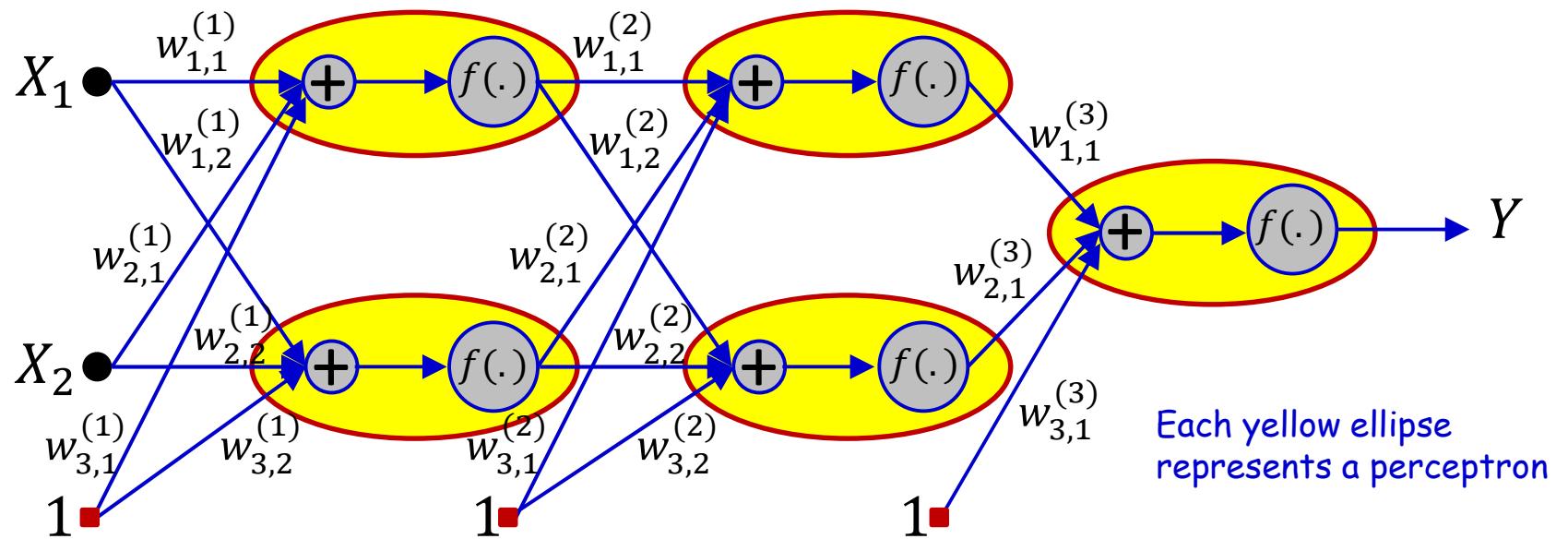
- Showing a tiny 2-input network for illustration
 - Actual network would have many more neurons and inputs
- Explicitly separating the weighted sum of inputs from the activation

A first closer look at the network



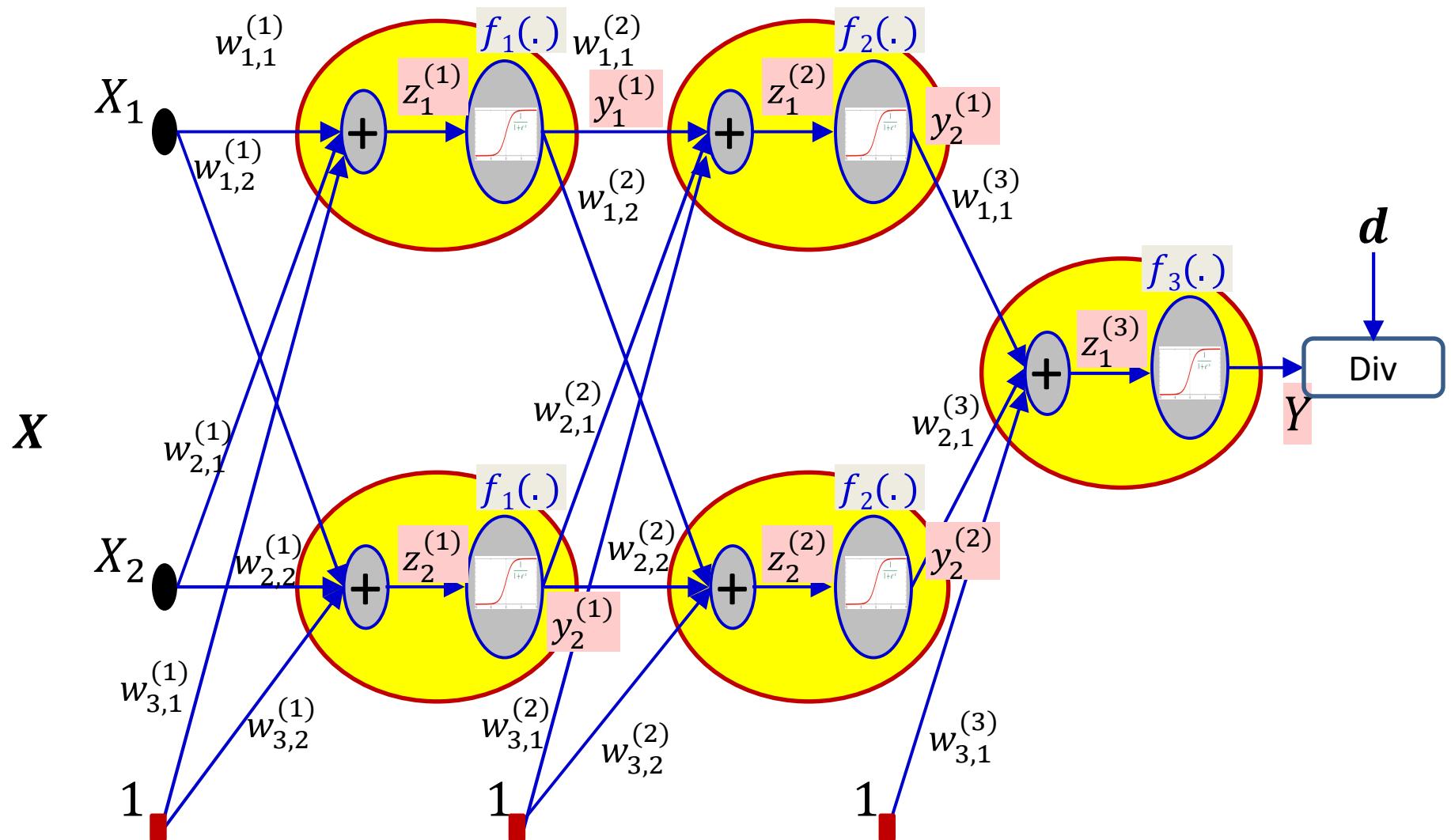
- Showing a tiny 2-input network for illustration
 - Actual network would have many more neurons and inputs
- Expanded **with all weights and activations shown**
- The overall function is differentiable w.r.t every weight, bias and input

Computing the derivative for a *single* input

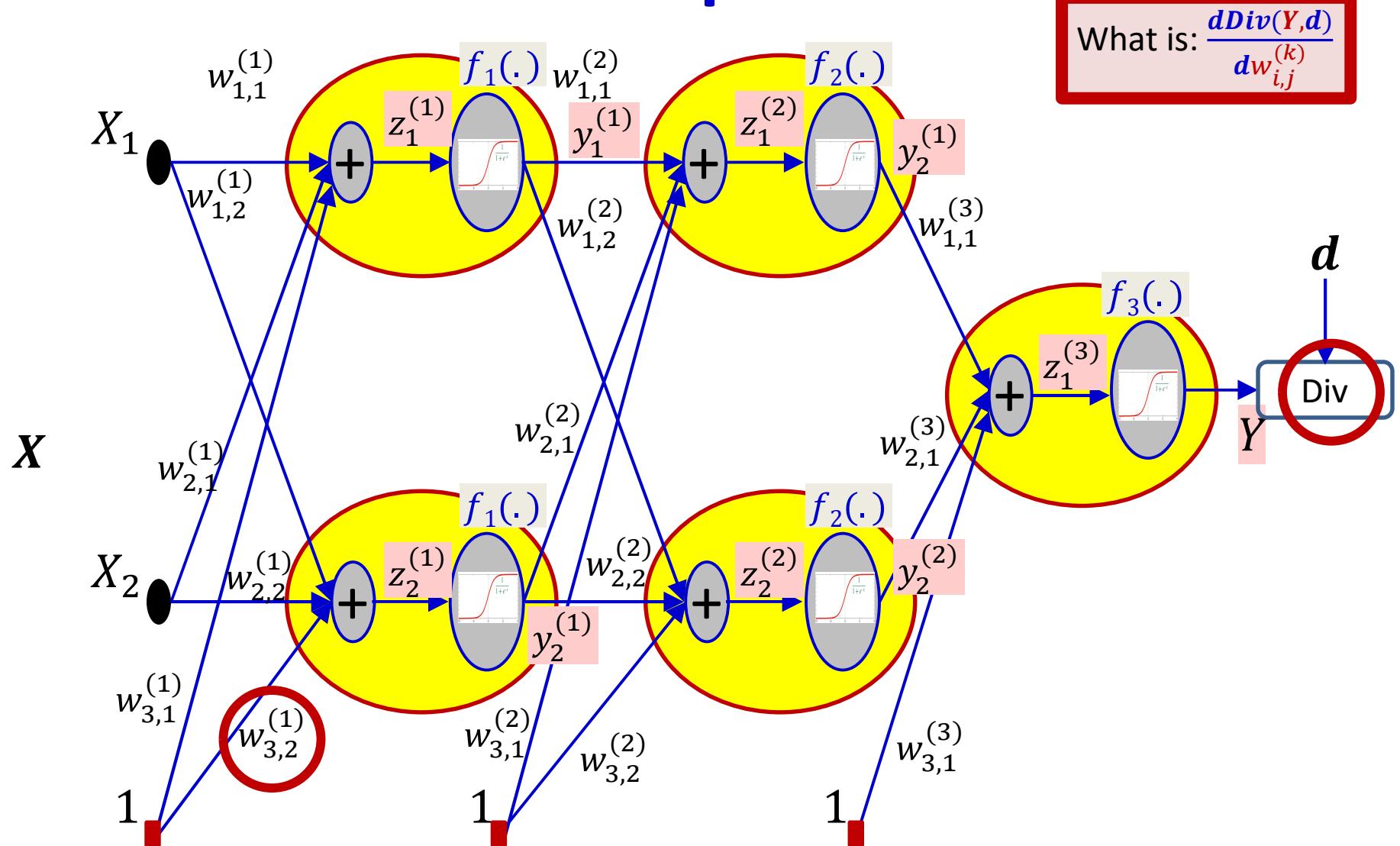


- Aim: compute derivative of $\text{Div}(Y, d)$ w.r.t. each of the weights
- But first, lets label *all* our variables and activation functions

Computing the derivative for a *single* input

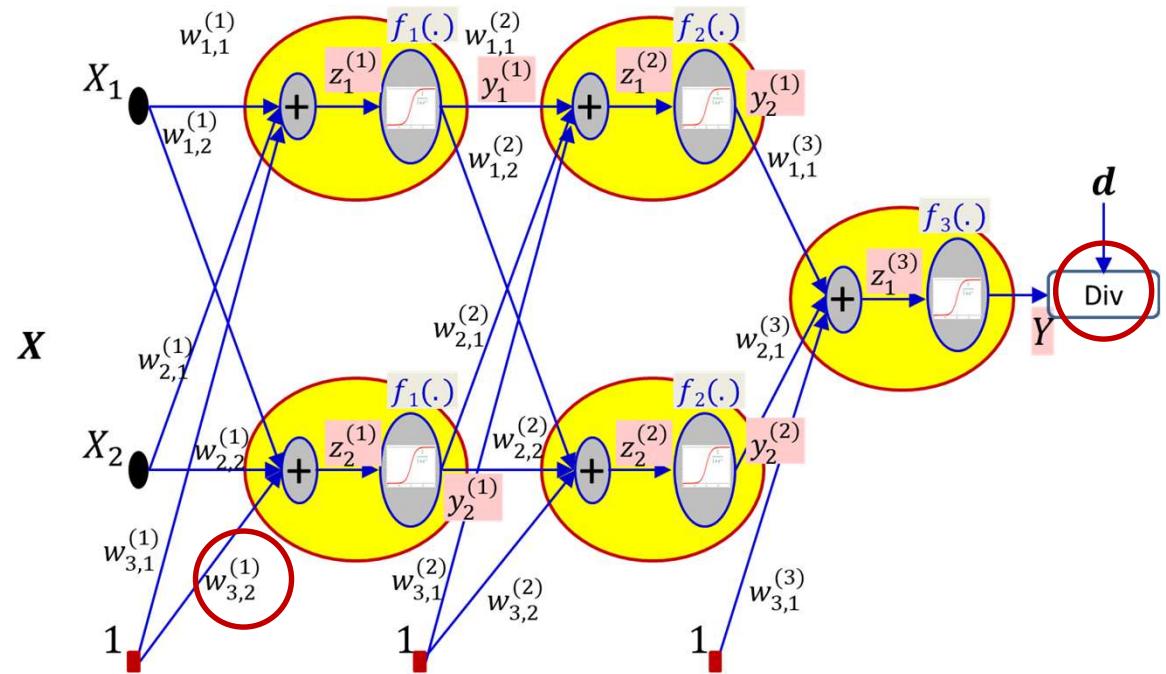


Computing the derivative for a *single* input



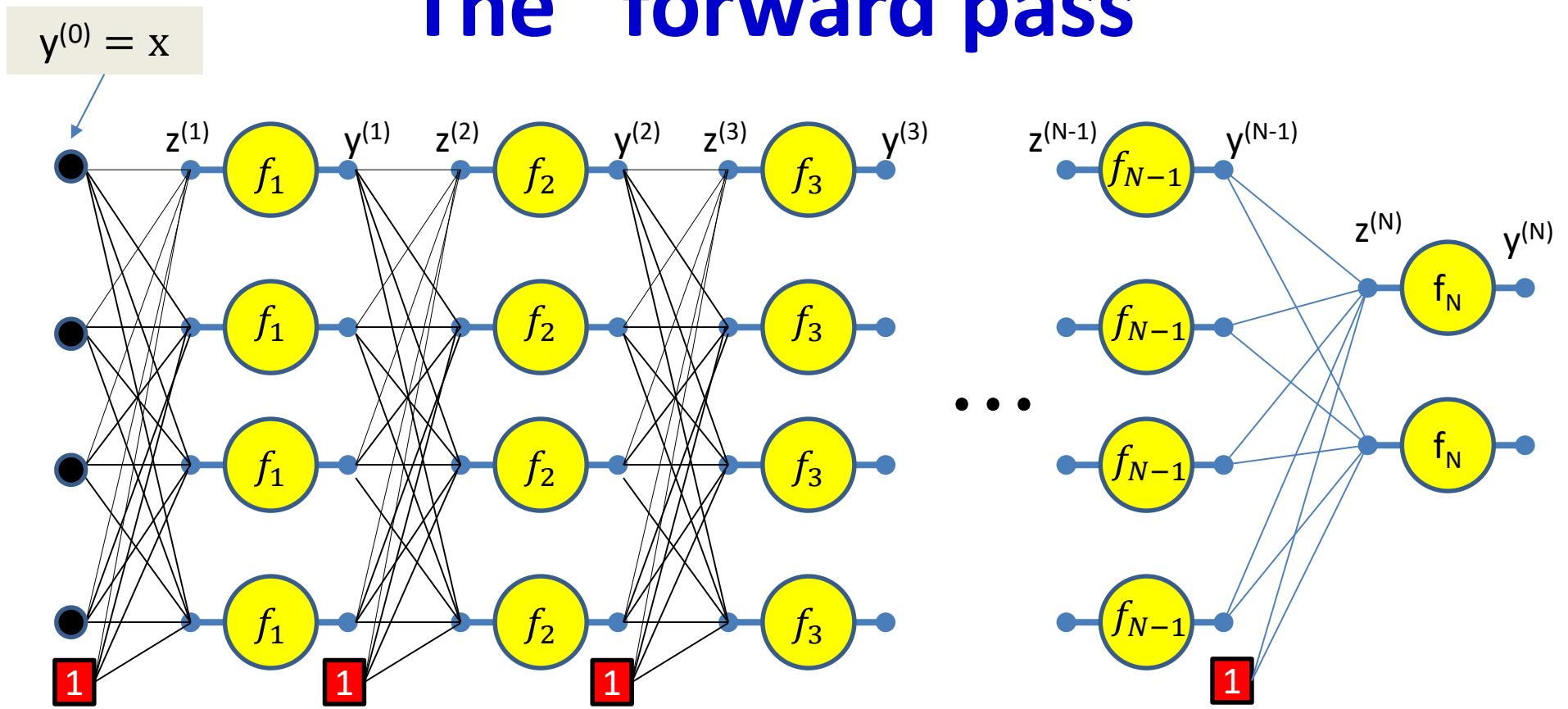
Computing the gradient

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



- Note: computation of the derivative requires intermediate and final output values of the network in response to the input

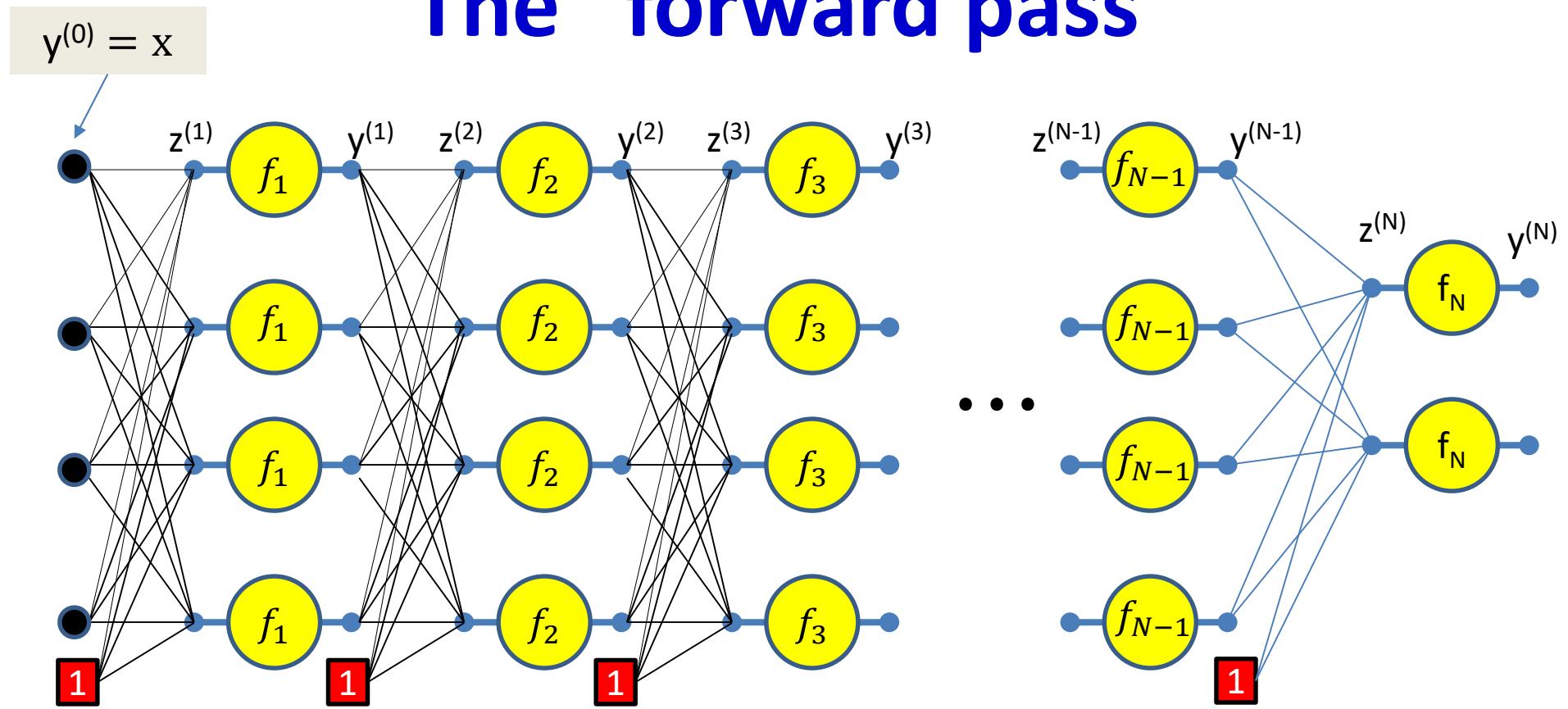
The “forward pass”



We will refer to the process of computing the output from an input as
the *forward pass*

We will illustrate the forward pass in the following slides

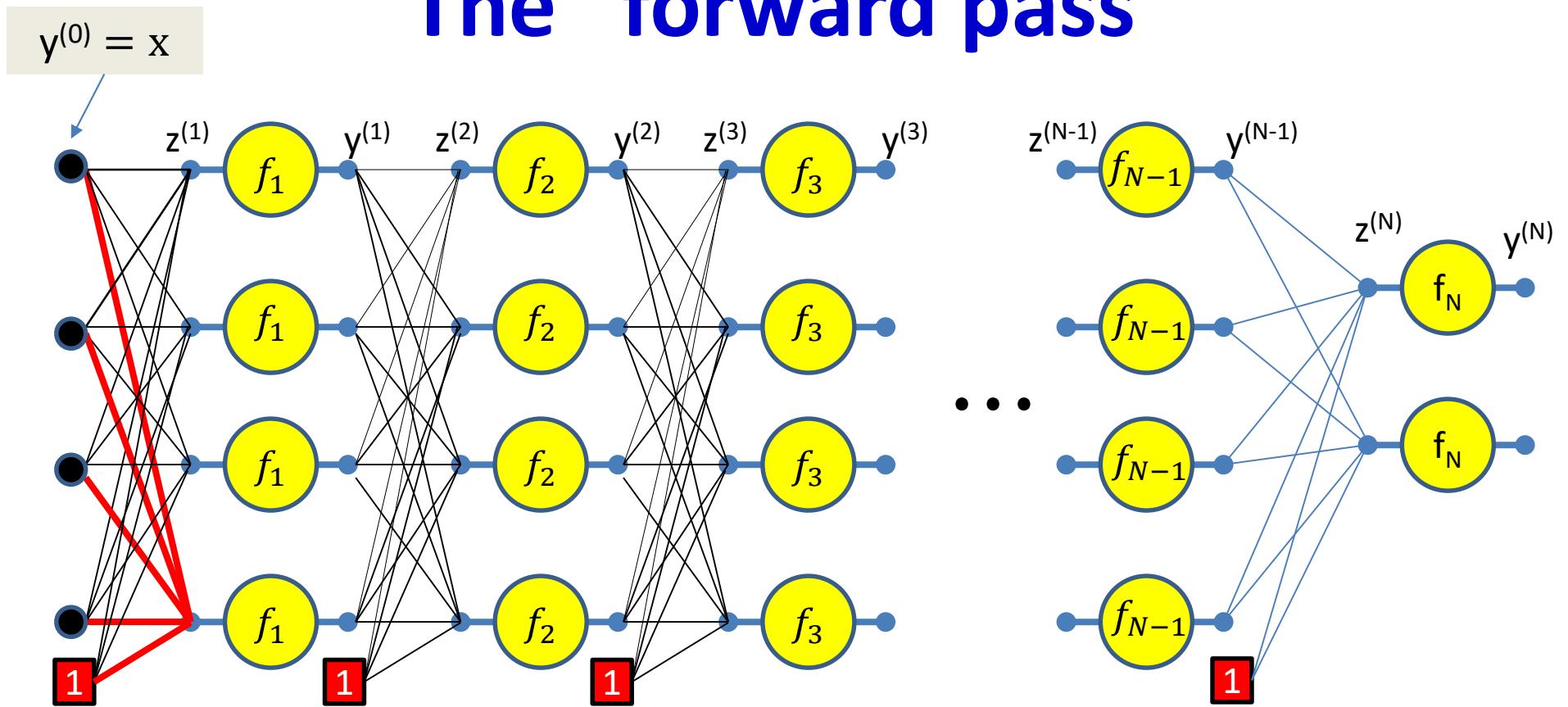
The “forward pass”



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

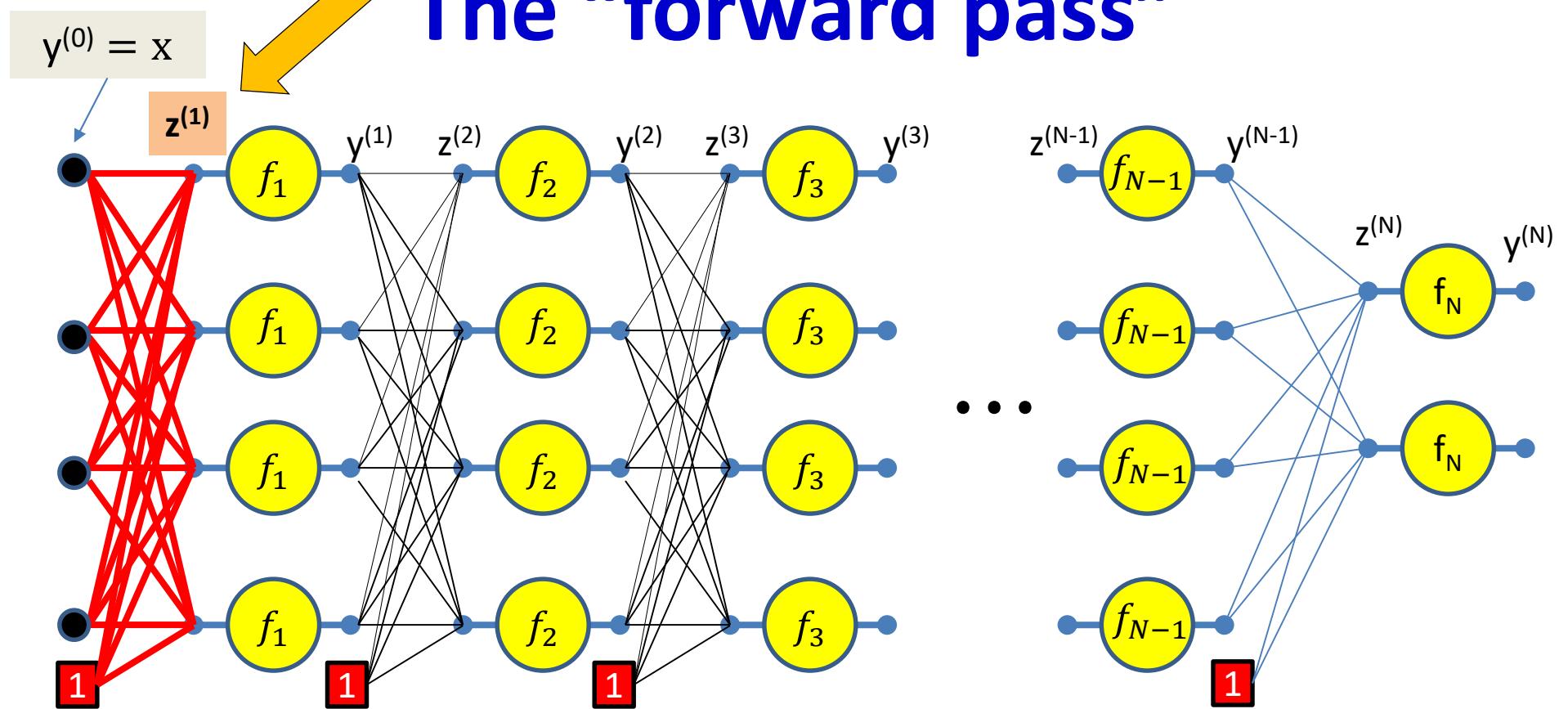
Assuming $w_{0j}^{(k)} = b_j^{(k)}$ and $y_0^{(k)} = 1$ -- assuming the bias is a weight and extending the output of every layer by a constant 1, to account for the biases

The “forward pass”

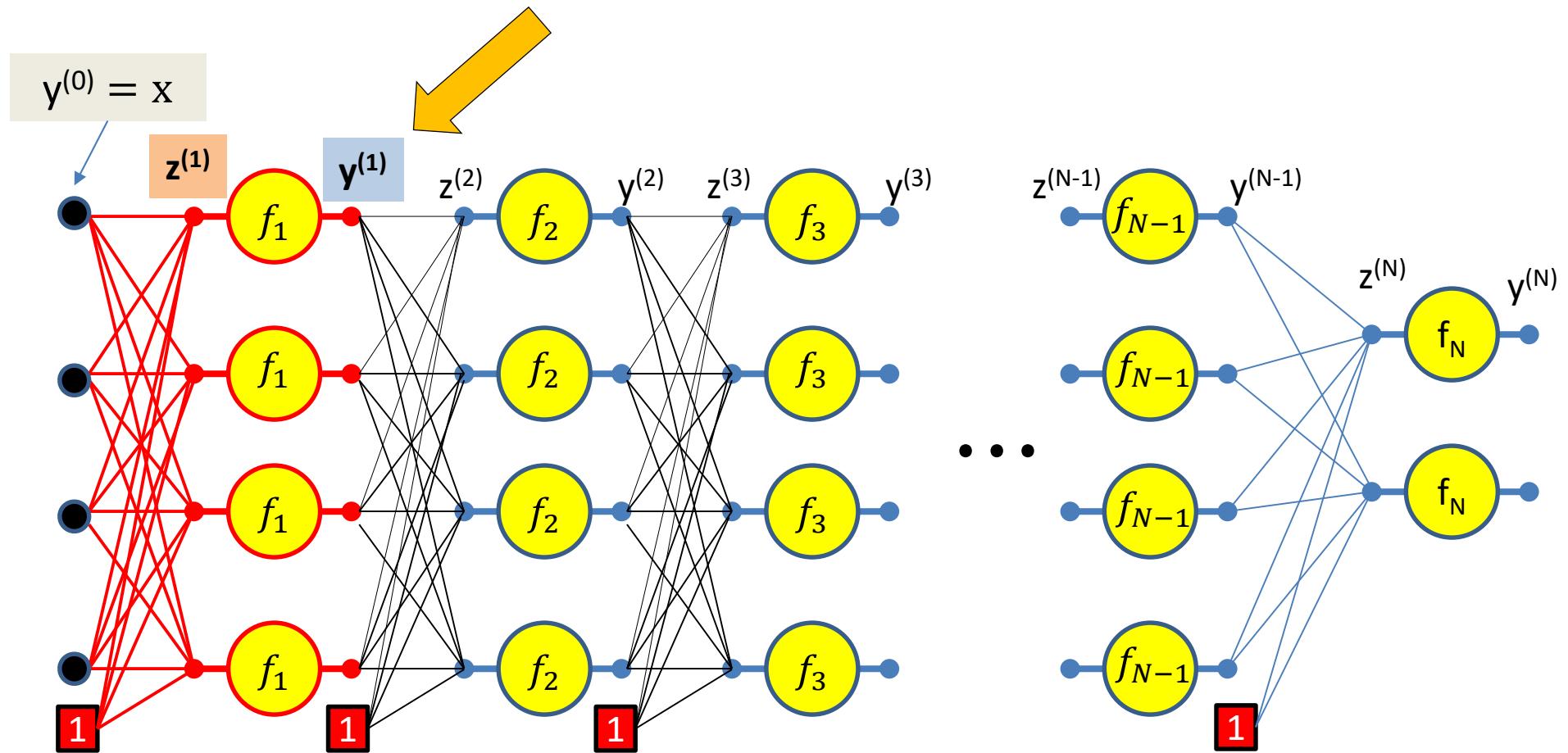


$$z_1^{(1)} = \sum_i w_{i1}^{(1)} y_i^{(0)}$$

The “forward pass”

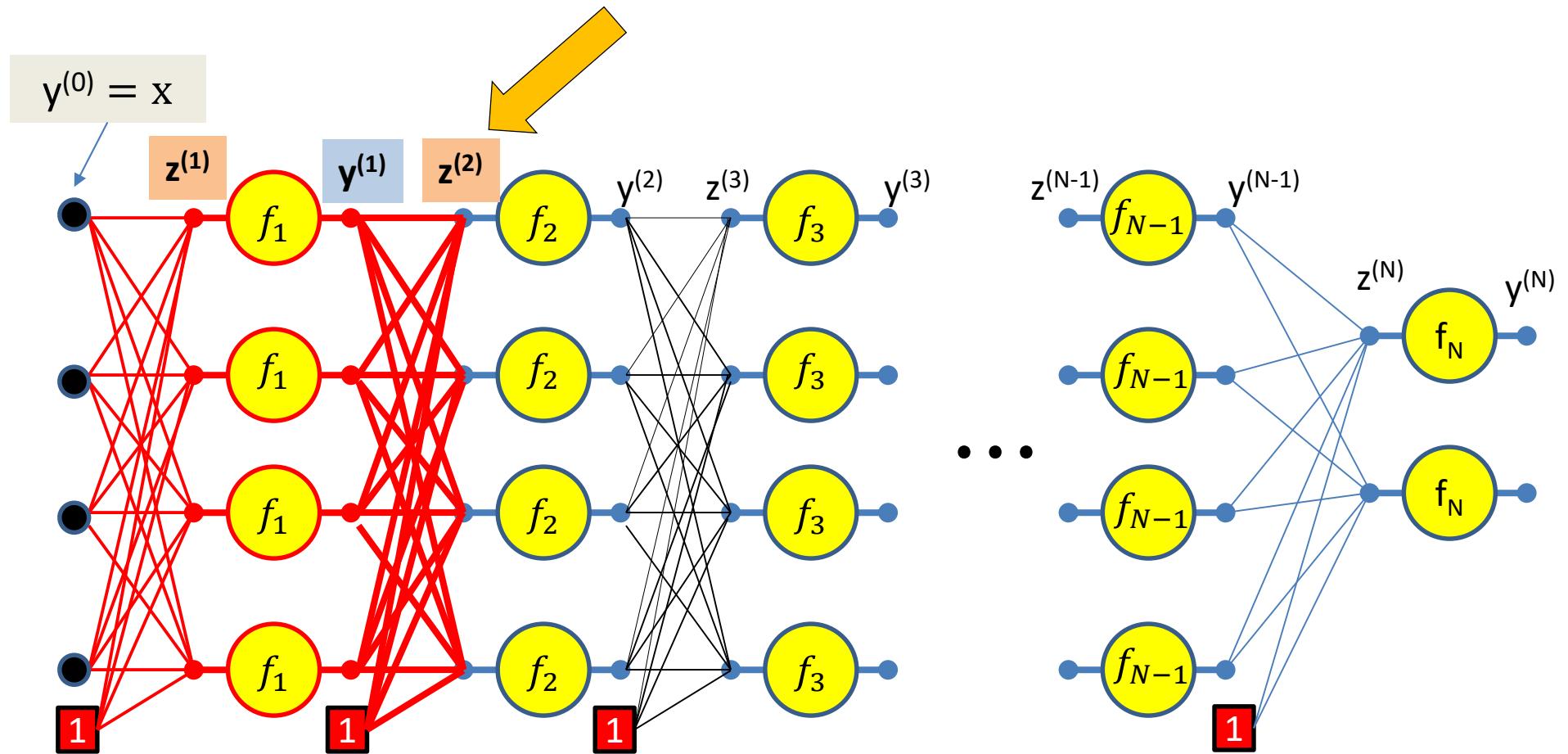


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$



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

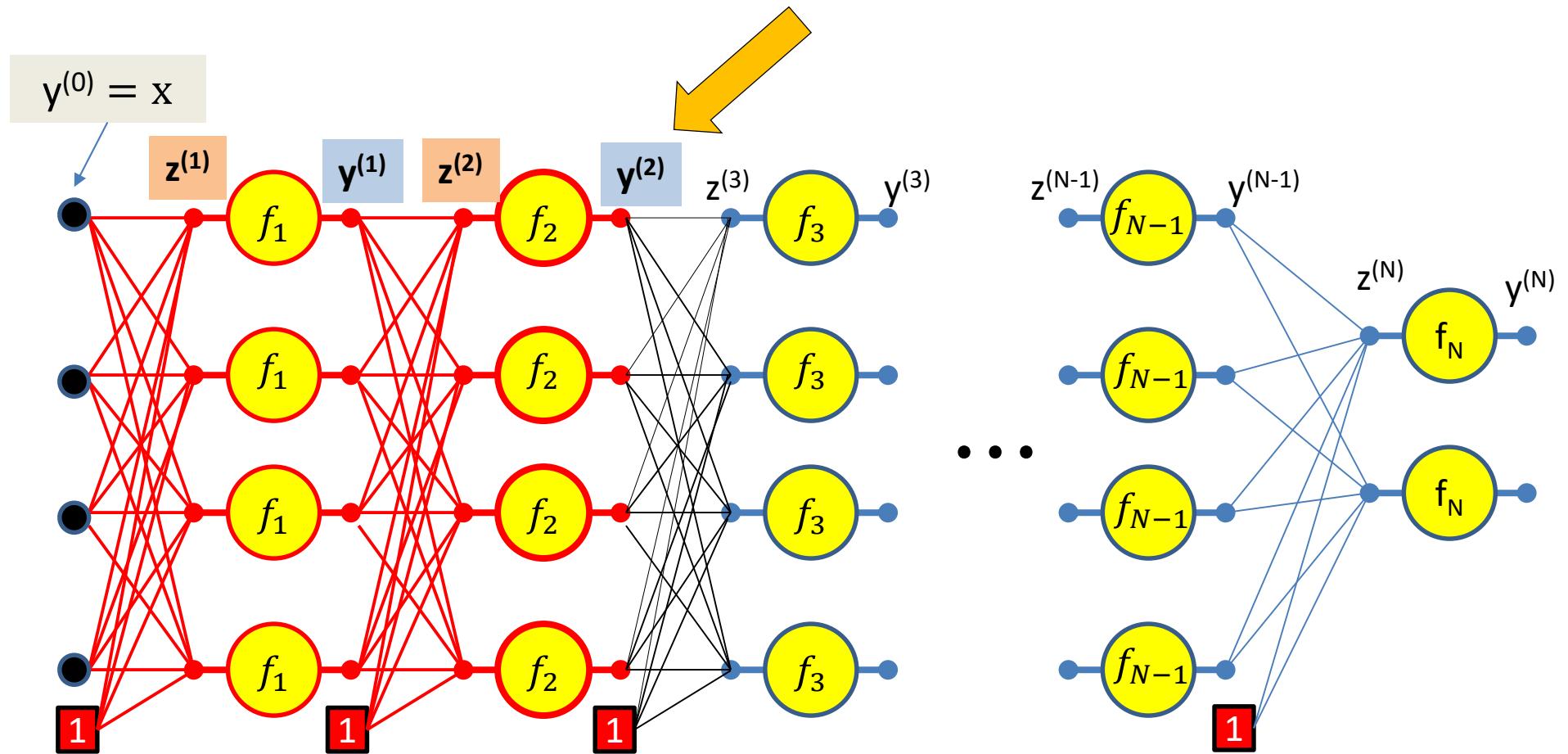
$$y_j^{(1)} = f_1(z_j^{(1)})$$



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

$$y_j^{(1)} = f_1(z_j^{(1)})$$

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

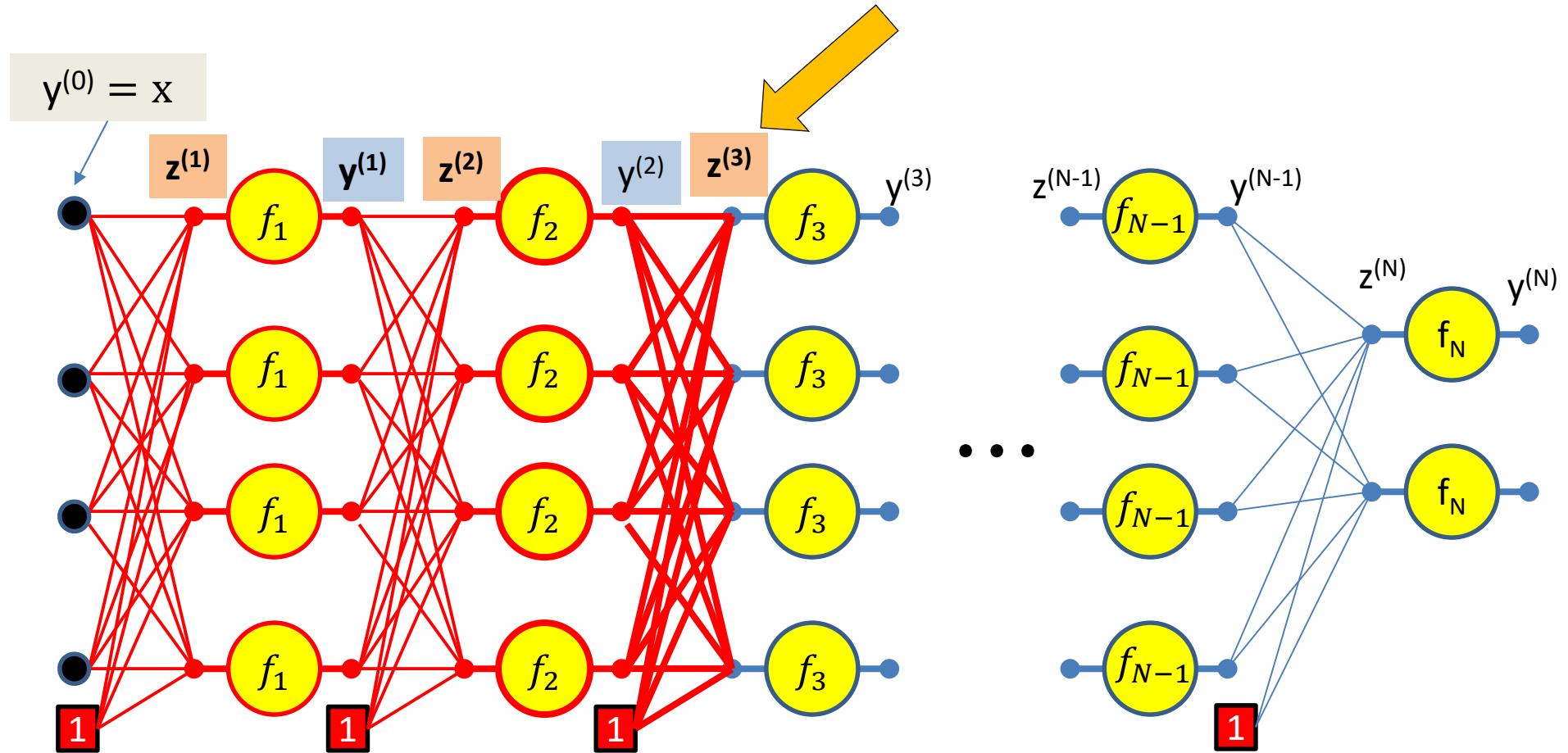


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

$$y_j^{(1)} = f_1(z_j^{(1)})$$

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

$$y_j^{(2)} = f_2(z_j^{(2)})$$



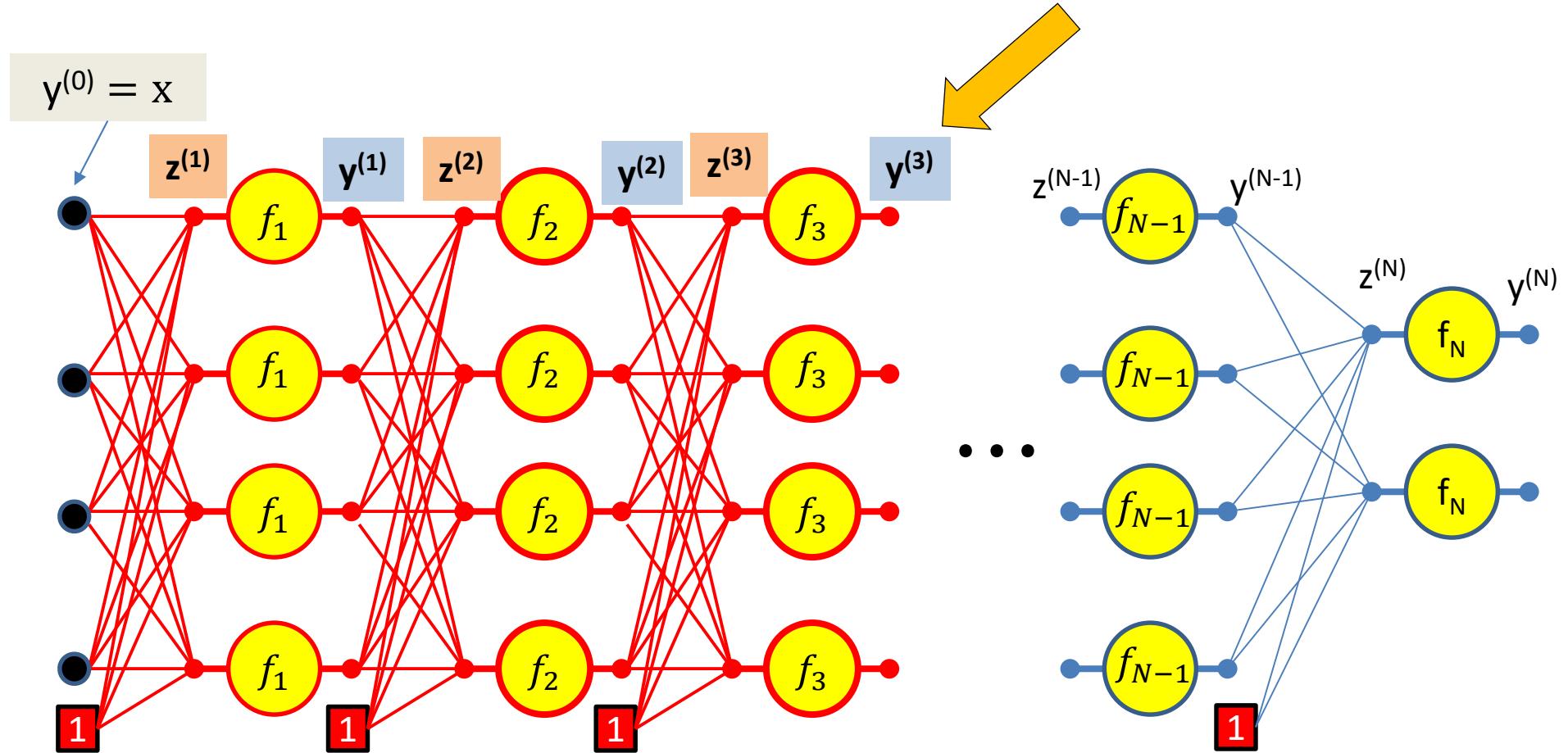
$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

$$y_j^{(1)} = f_1(z_j^{(1)})$$

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

$$y_j^{(2)} = f_2(z_j^{(2)})$$

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



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

$$y_j^{(1)} = f_1(z_j^{(1)})$$

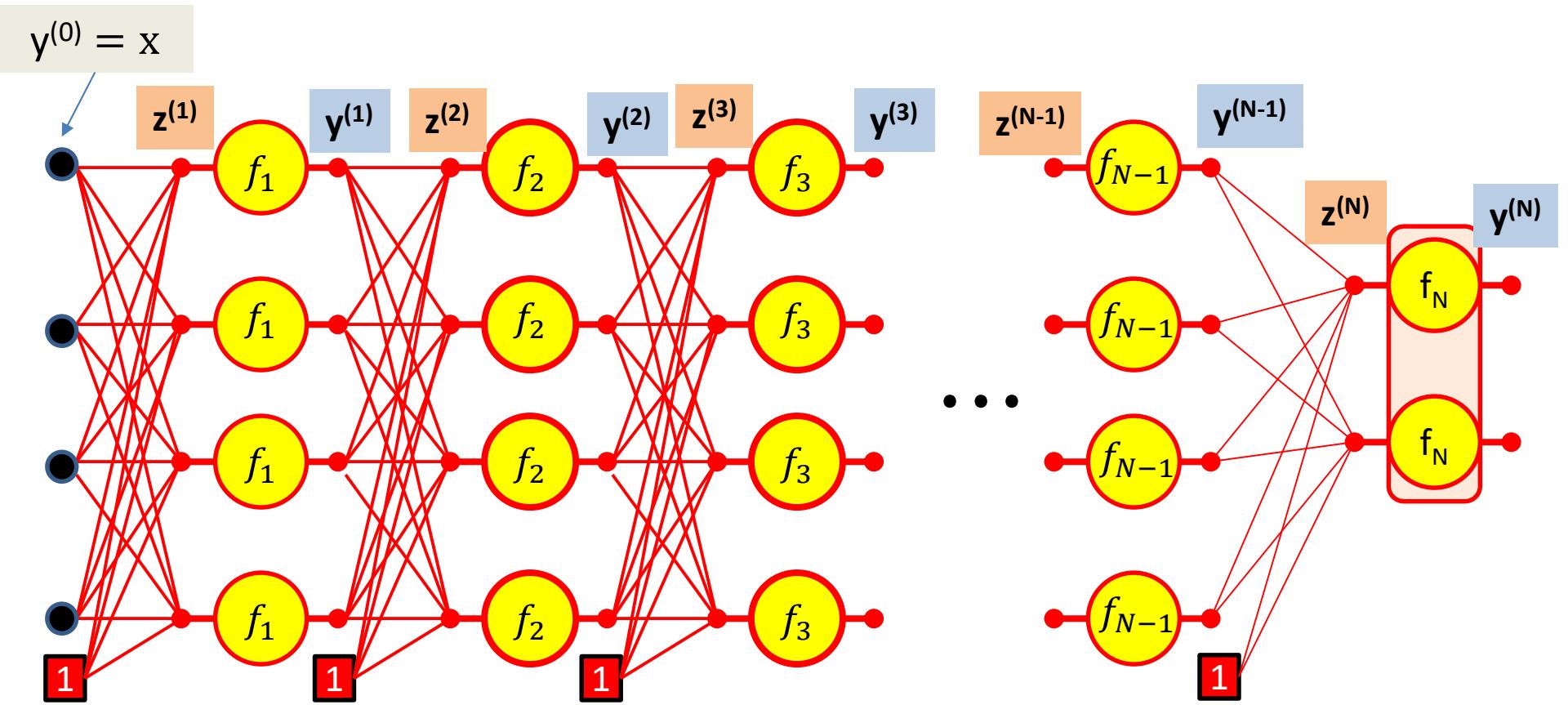
$$z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)}$$

$$y_j^{(2)} = f_2(z_j^{(2)})$$

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

$$y_j^{(3)} = f_3(z_j^{(3)})$$

\dots

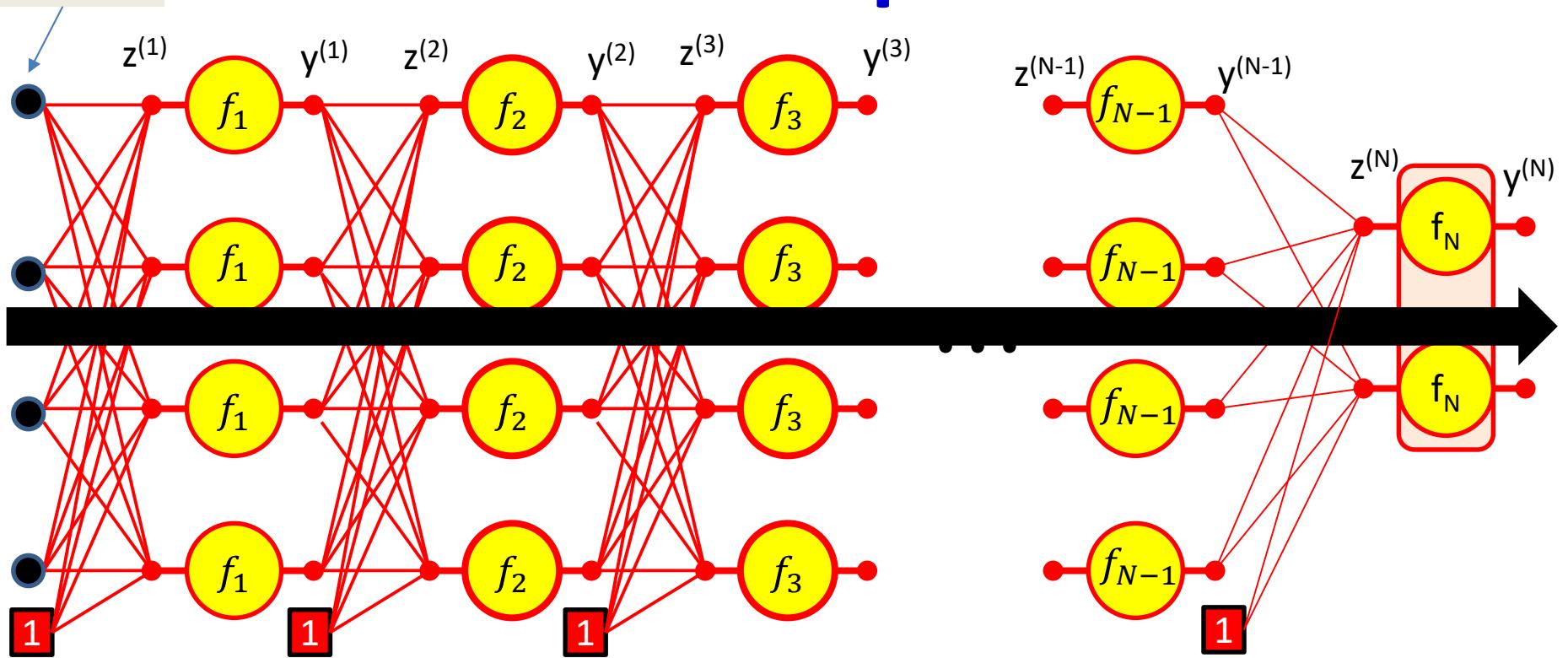


$$y_j^{(N-1)} = f_{N-1}(z_j^{(N-1)}) \quad z_j^{(N)} = \sum_i w_{ij}^{(N)} y_i^{(N-1)}$$

$$\mathbf{y}^{(N)} = f_N(\mathbf{z}^{(N)})$$

Forward Computation

$$y^{(0)} = x$$



ITERATE FOR $k = 1:N$

for $j = 1:\text{layer-width}$

$$y_i^{(0)} = x_i$$

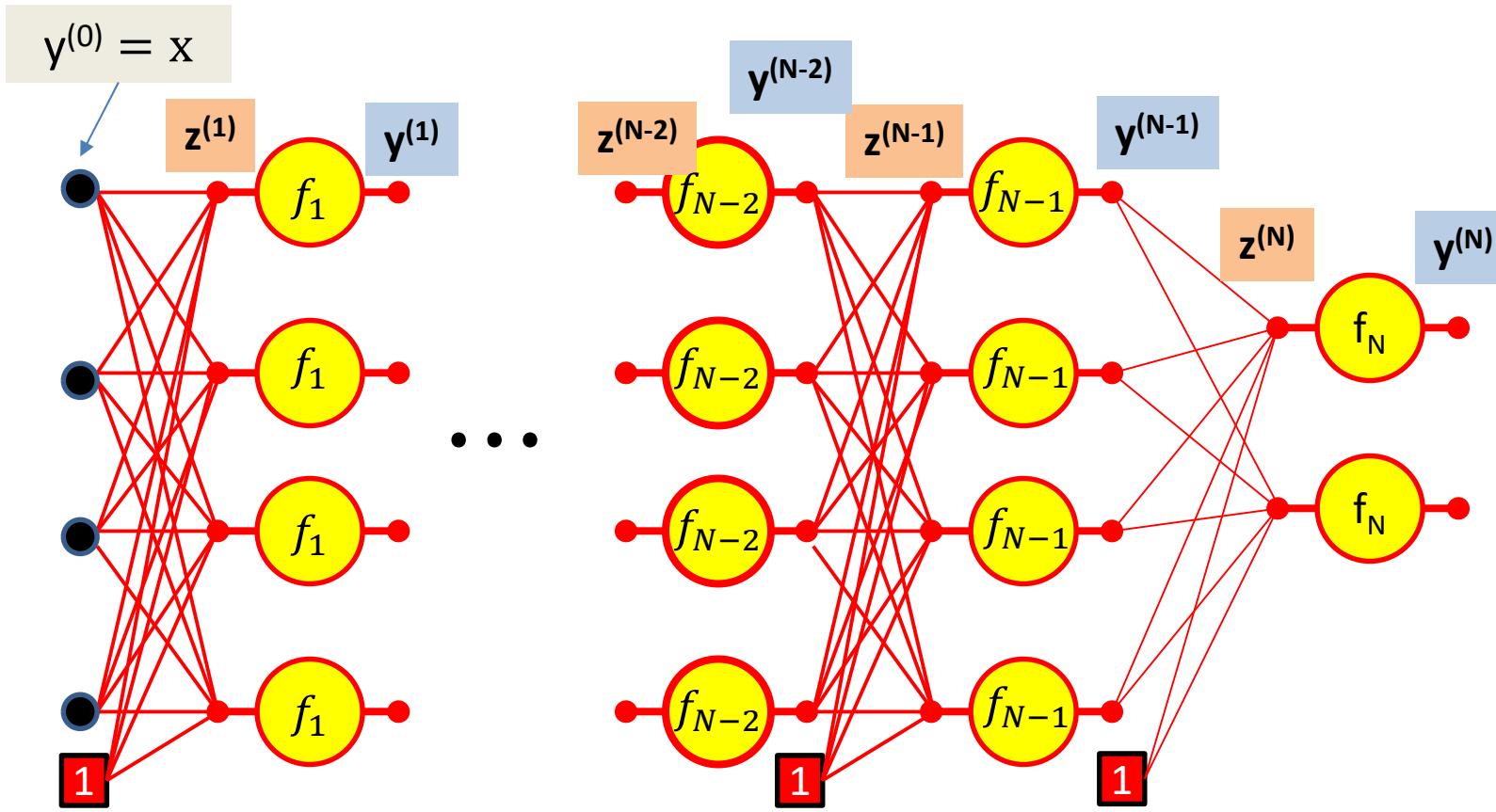
$$z_j^{(k)} = \sum_i w_{ij}^{(k)} y_i^{(k-1)}$$

$$y_j^{(k)} = f_k(z_j^{(k)})$$

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
 - $y_j^{(0)} = x_j, j = 1 \dots D; y_0^{(k=1\dots N)} = x_0 = 1$
- For layer $k = 1 \dots N$
 - For $j = 1 \dots D_k$ D_k is the size of the kth layer
 - $z_j^{(k)} = \sum_{i=0}^{D_{k-1}} w_{i,j}^{(k)} y_i^{(k-1)}$
 - $y_j^{(k)} = f_k(z_j^{(k)})$
- Output:
 - $Y = y_j^{(N)}, j = 1..D_N$

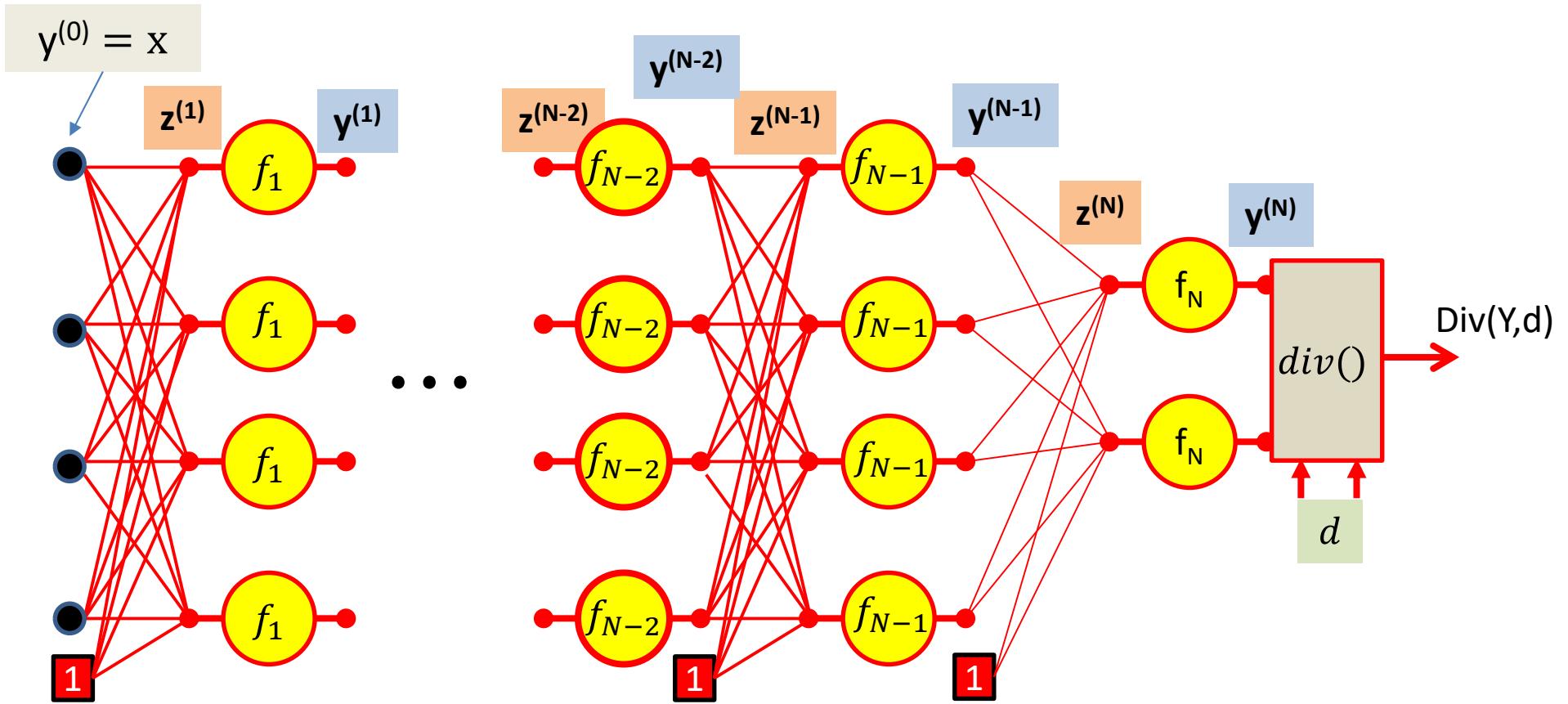
Computing derivatives



We have computed all these intermediate values in the forward computation

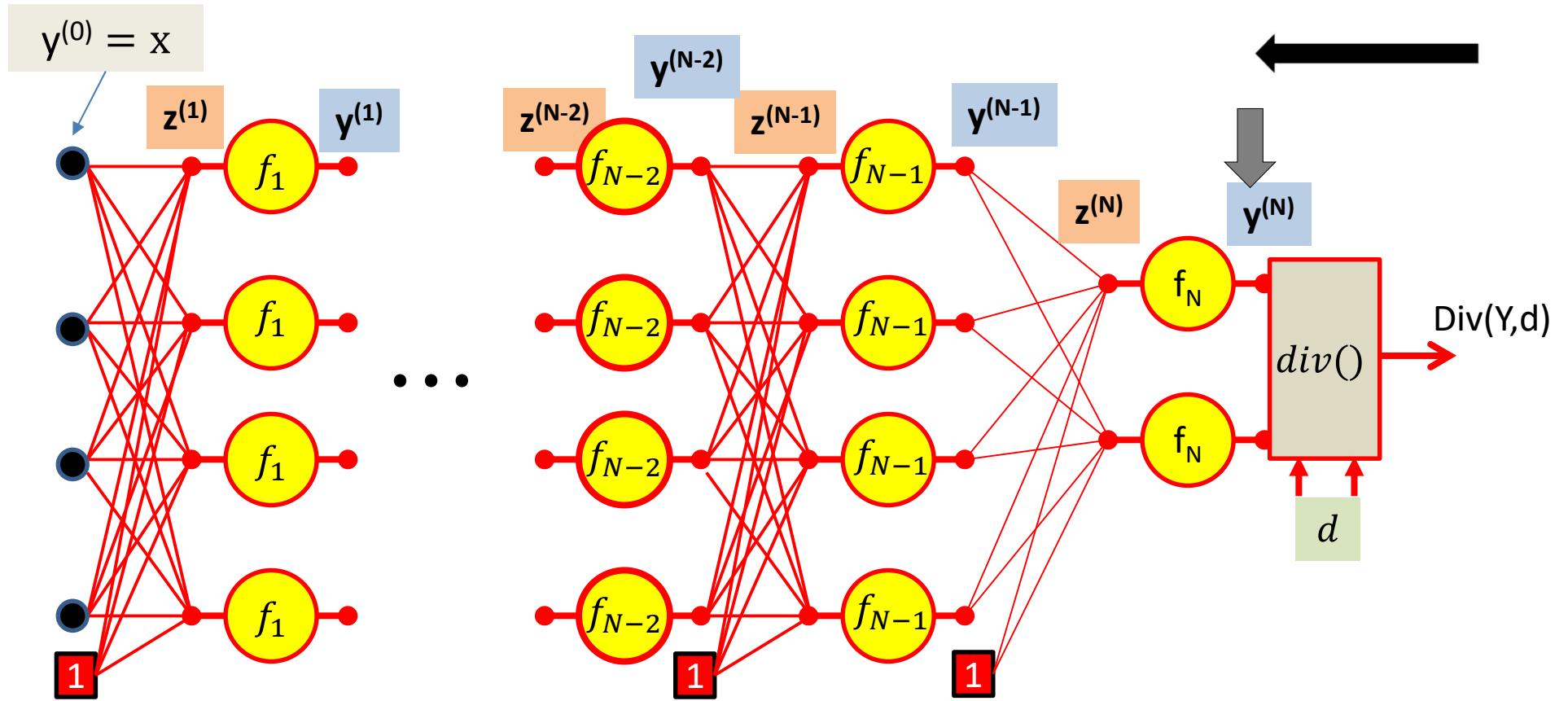
We must remember them - we will need them to compute the derivatives

Computing derivatives



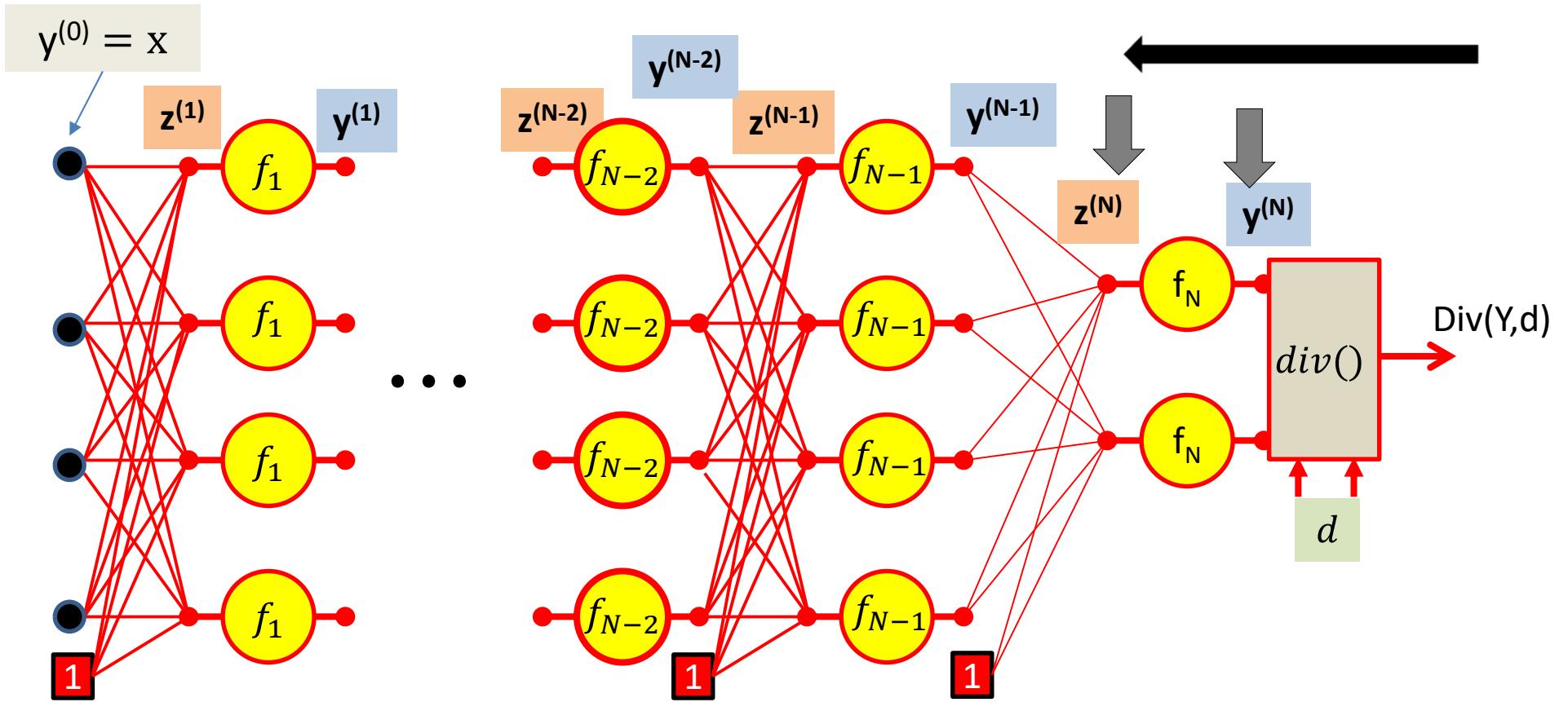
First, we compute the divergence between the output of the net $y = y^{(N)}$ and the desired output d

Computing derivatives



We then compute $\nabla_{Y^{(N)}} \text{div}(\cdot)$ the derivative of the divergence w.r.t. the final output of the network $y^{(N)}$

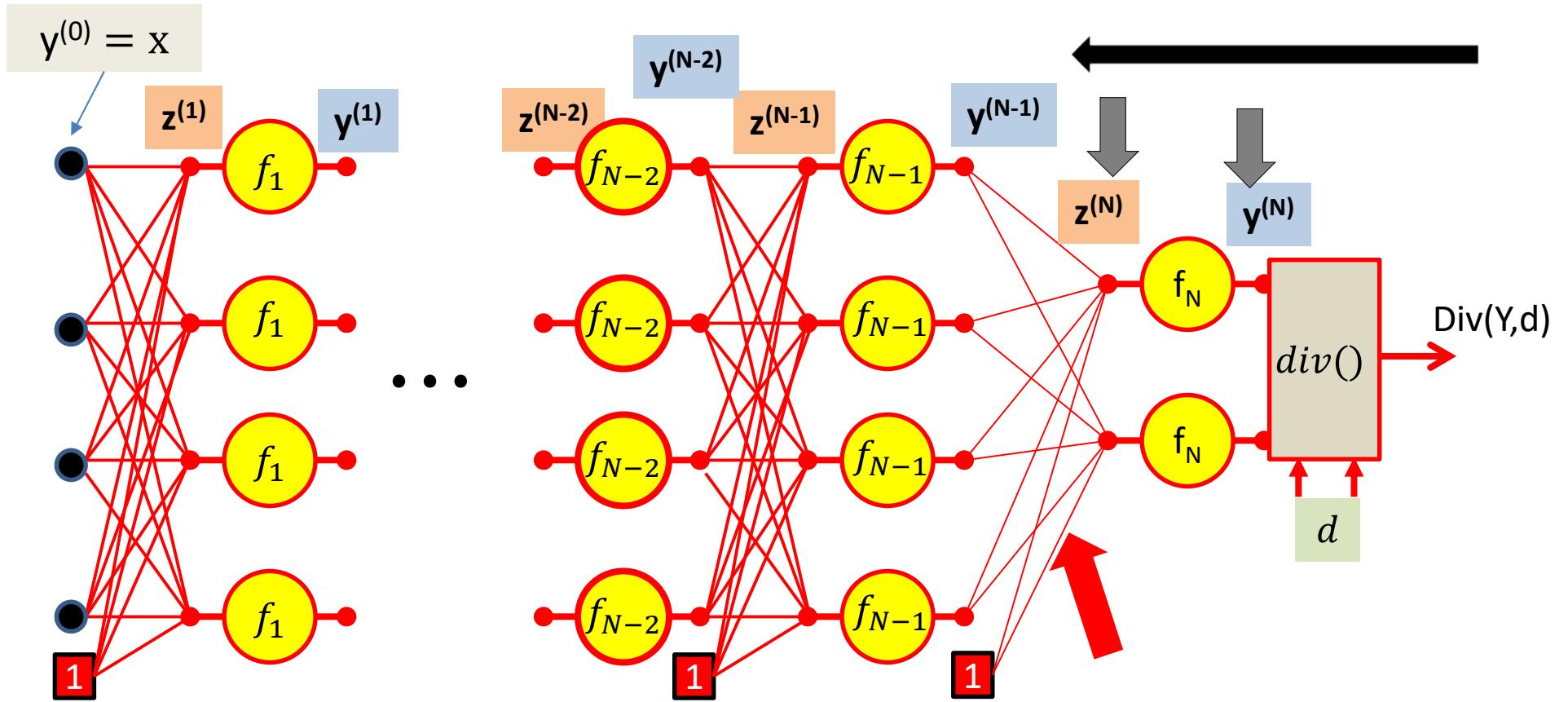
Computing derivatives



We then compute $\nabla_{Y^{(N)}} div(\cdot)$ the derivative of the divergence w.r.t. the final output of the network $y^{(N)}$

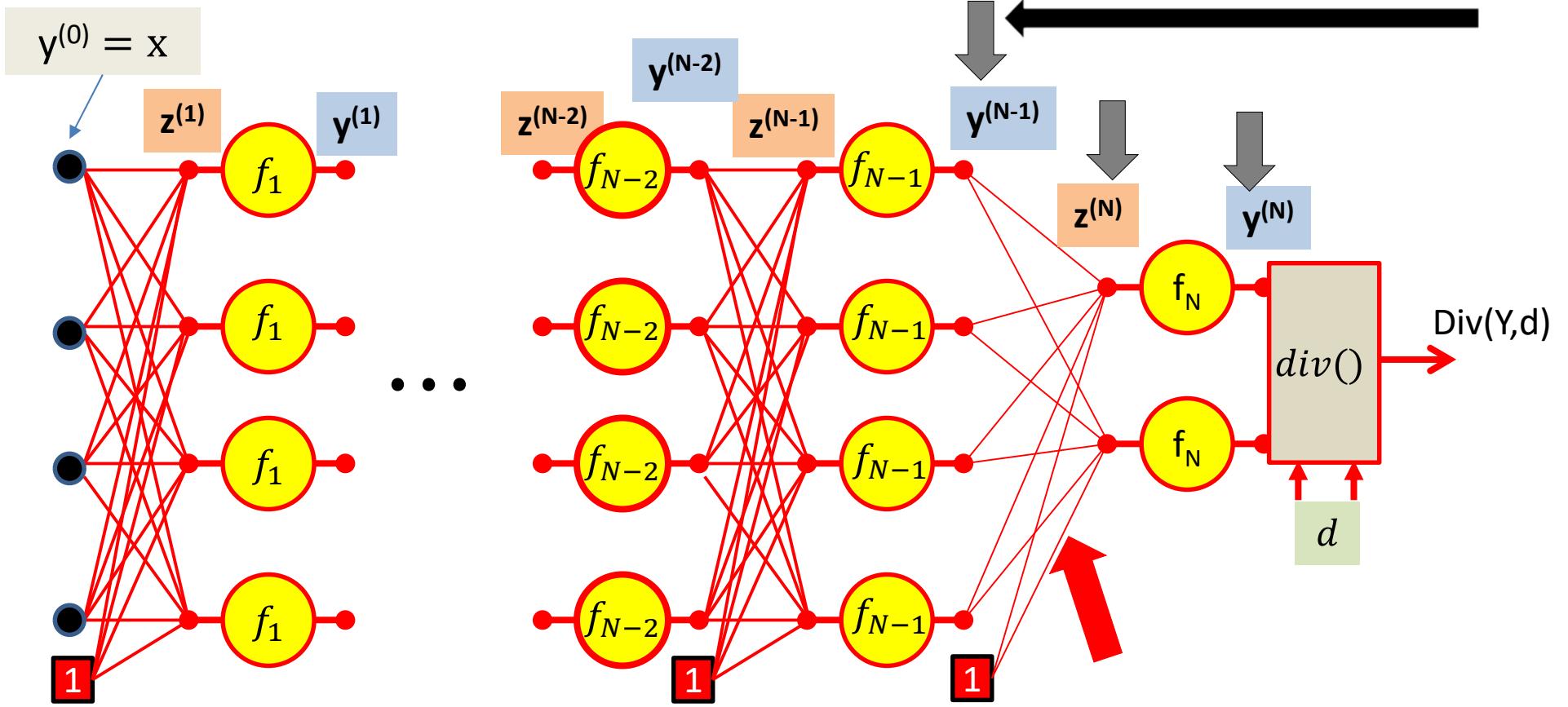
We then compute $\nabla_{z^{(N)}} div(\cdot)$ the derivative of the divergence w.r.t. the *pre-activation affine combination* $z^{(N)}$ using the chain rule

Computing derivatives



Continuing on, we will compute $\nabla_{W^{(N)}} \text{div}(\cdot)$ the derivative of the divergence with respect to the weights of the connections to the output layer

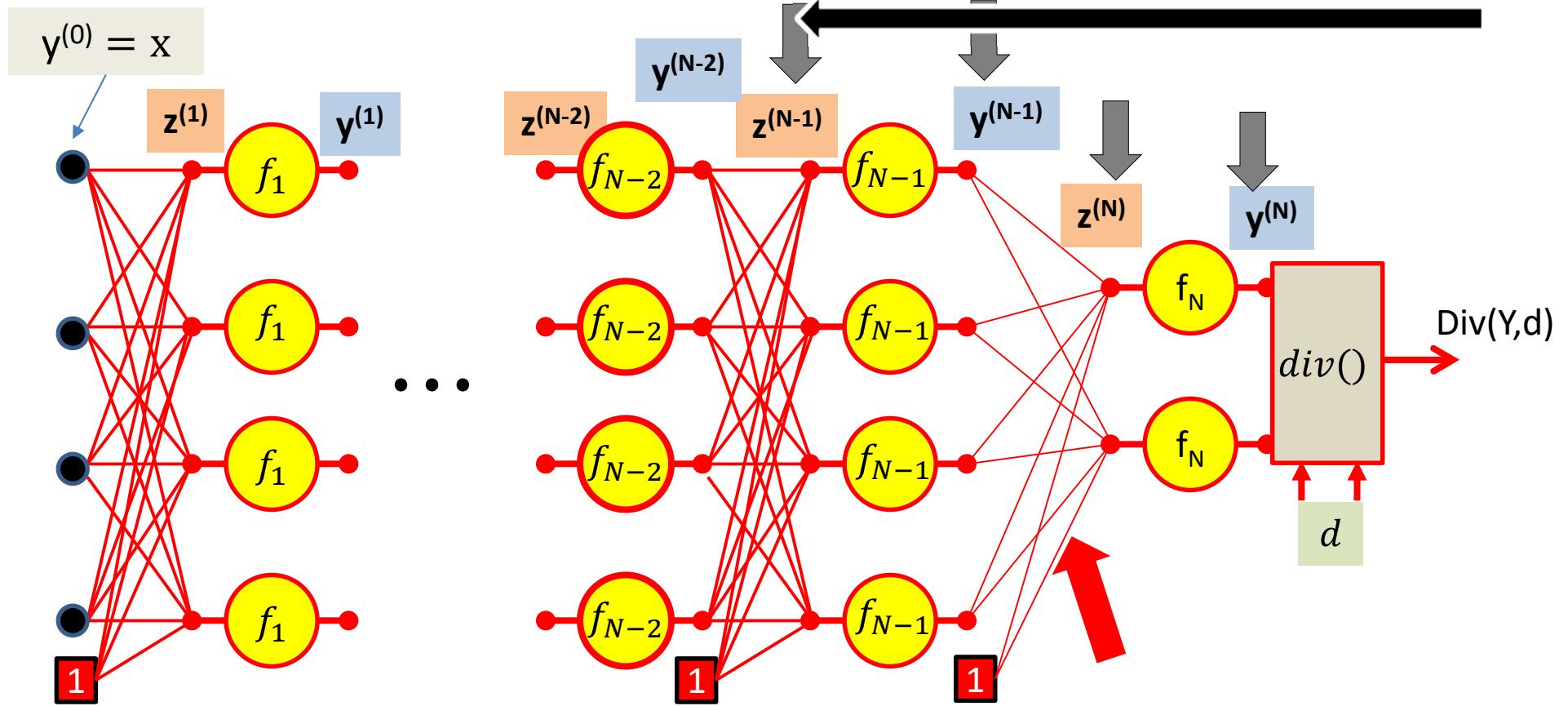
Computing derivatives



Continuing on, we will compute $\nabla_{W^{(N)}} \text{div}(\cdot)$ the derivative of the divergence with respect to the weights of the connections to the output layer

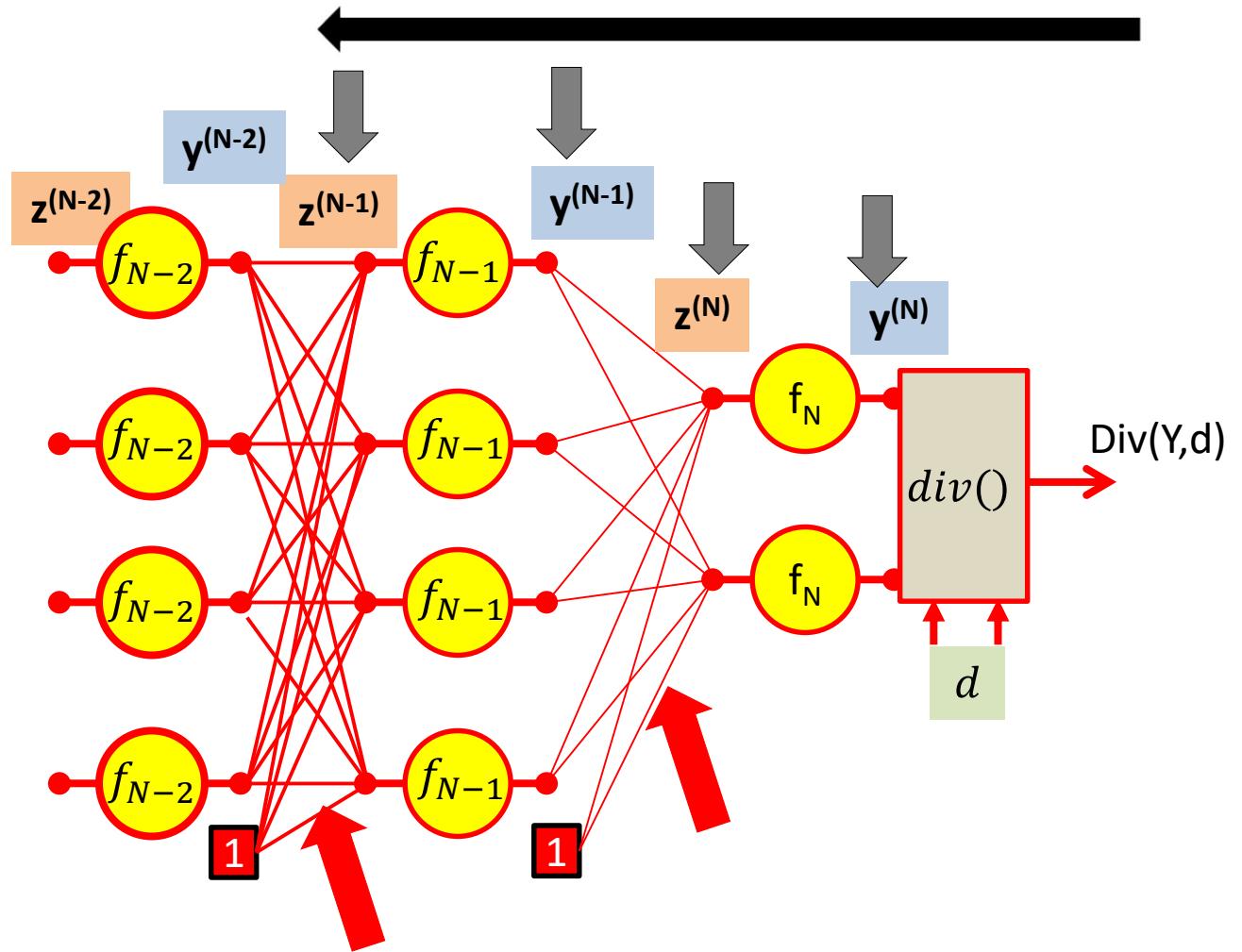
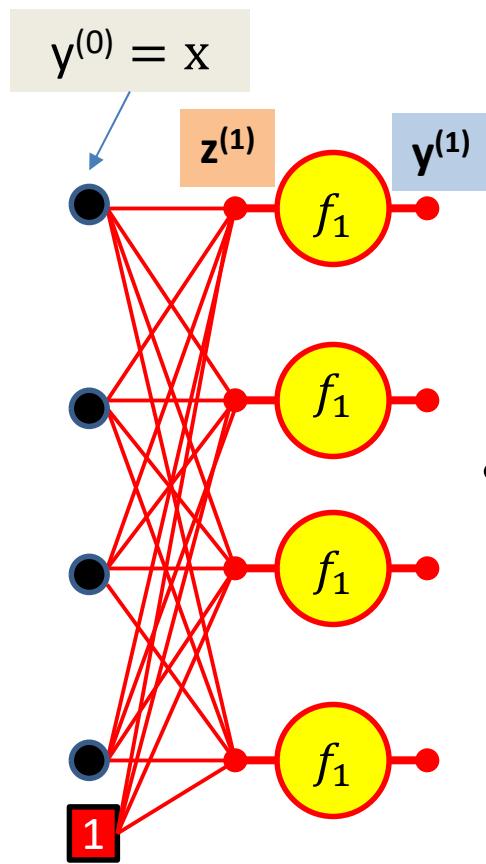
Then continue with the chain rule to compute $\nabla_{Y^{(N-1)}} \text{div}(\cdot)$ the derivative of the divergence w.r.t. the output of the N-1th layer

Computing derivatives



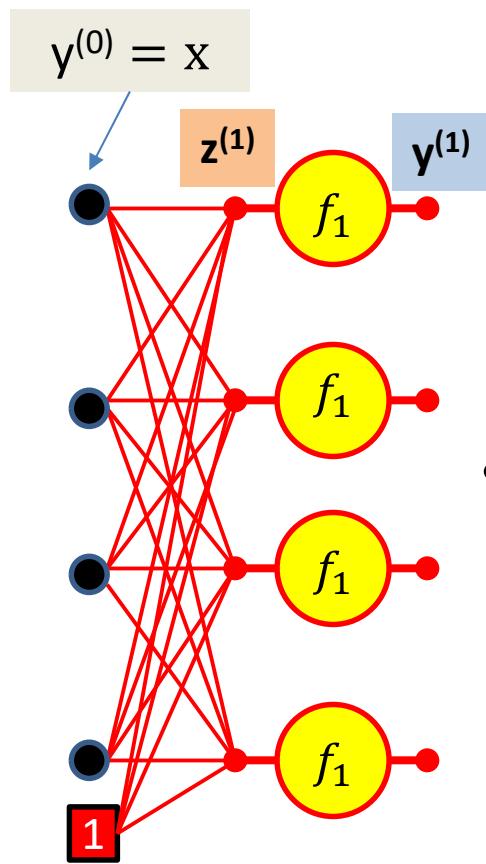
We continue our way backwards in the order shown

$$\nabla_{z^{(N-1)}} div(.)$$

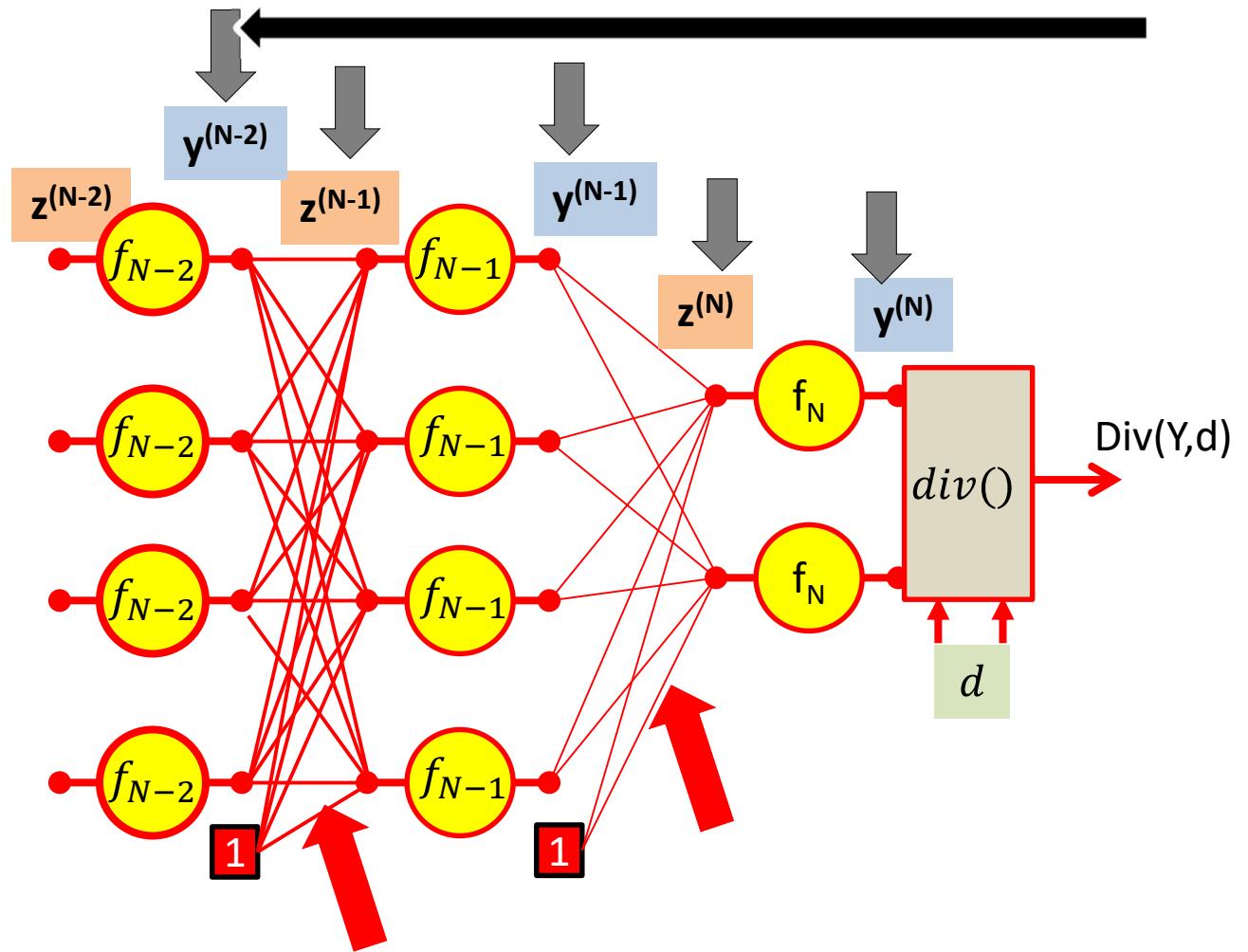


We continue our way backwards in the order shown

$$\nabla_{W^{(N-1)}} div(.)$$

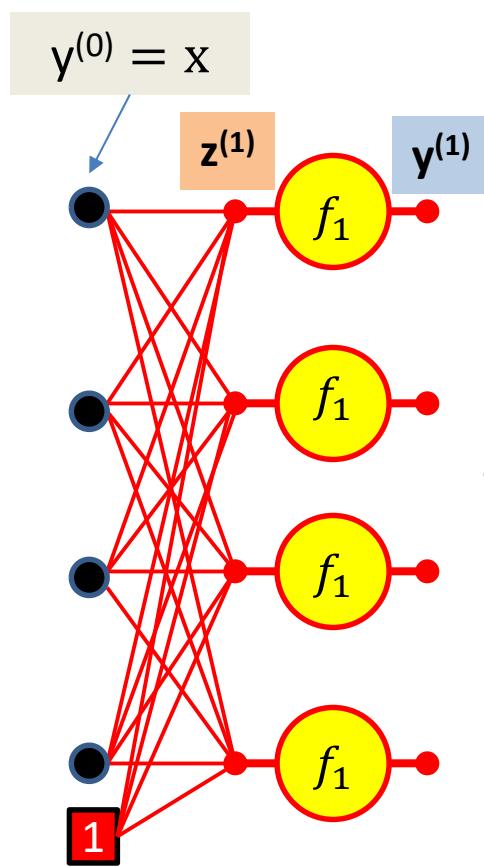


...

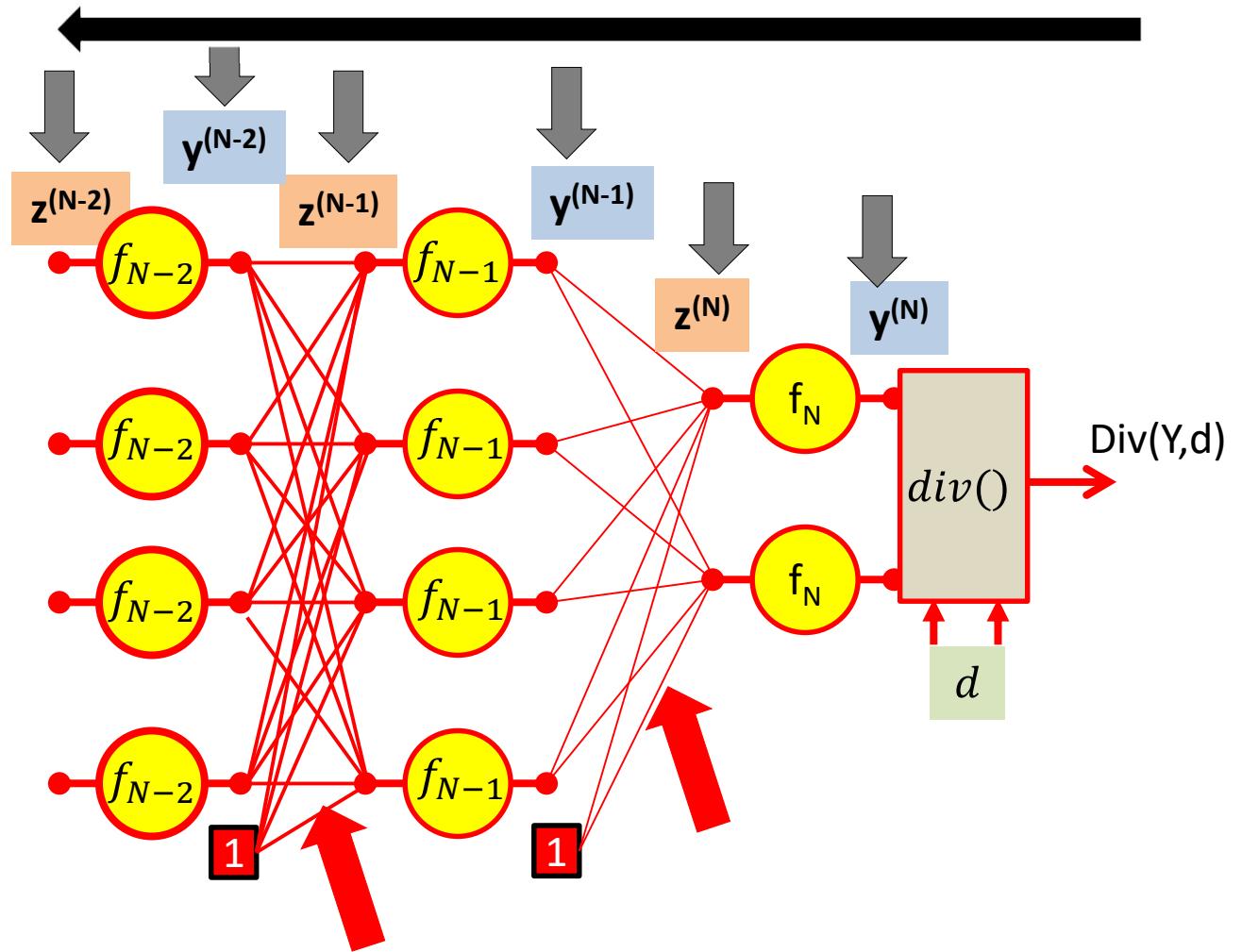


We continue our way backwards in the order shown

$$\nabla_{Y^{(N-2)}} \text{div}(\cdot)$$

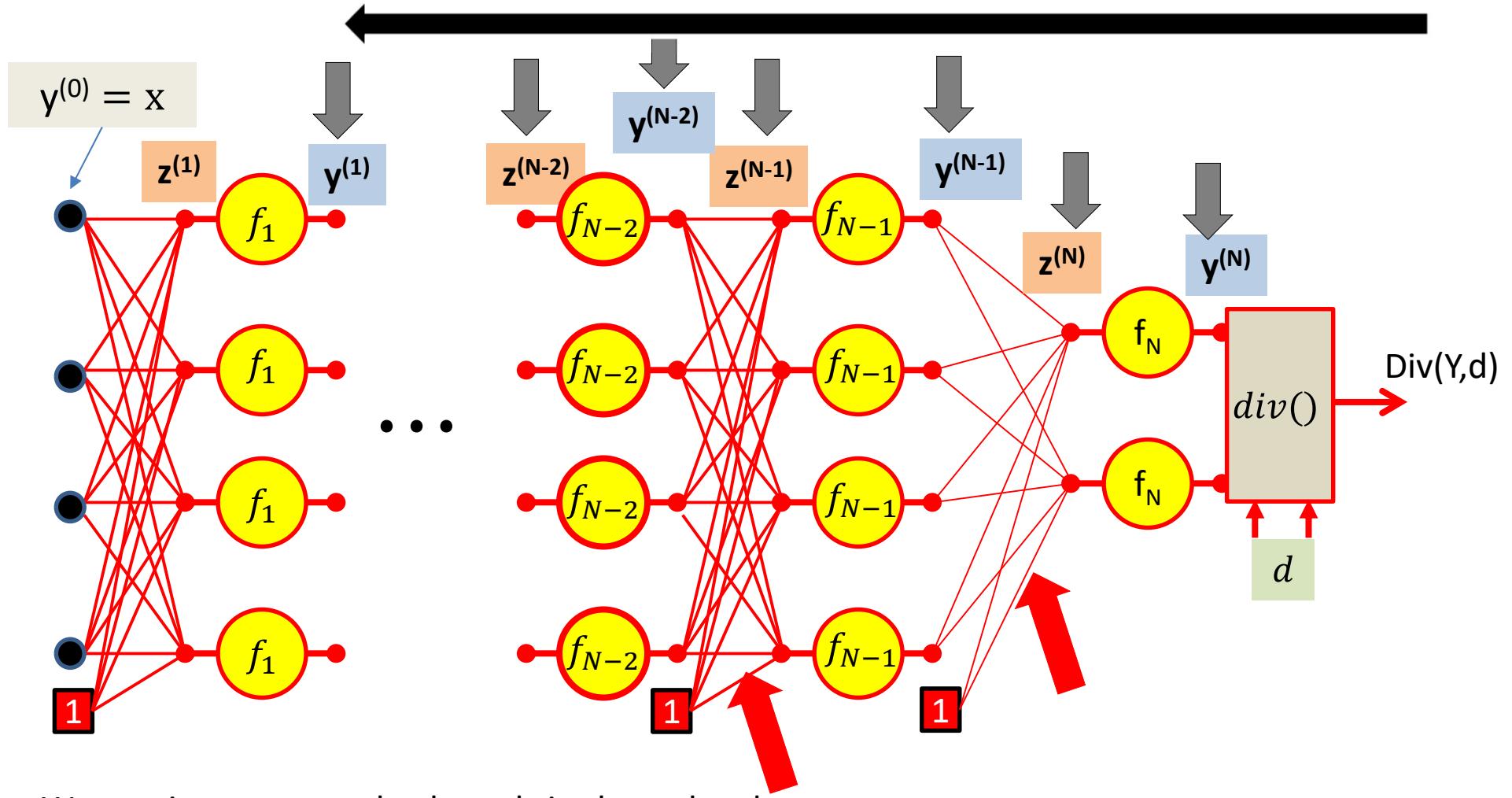


...



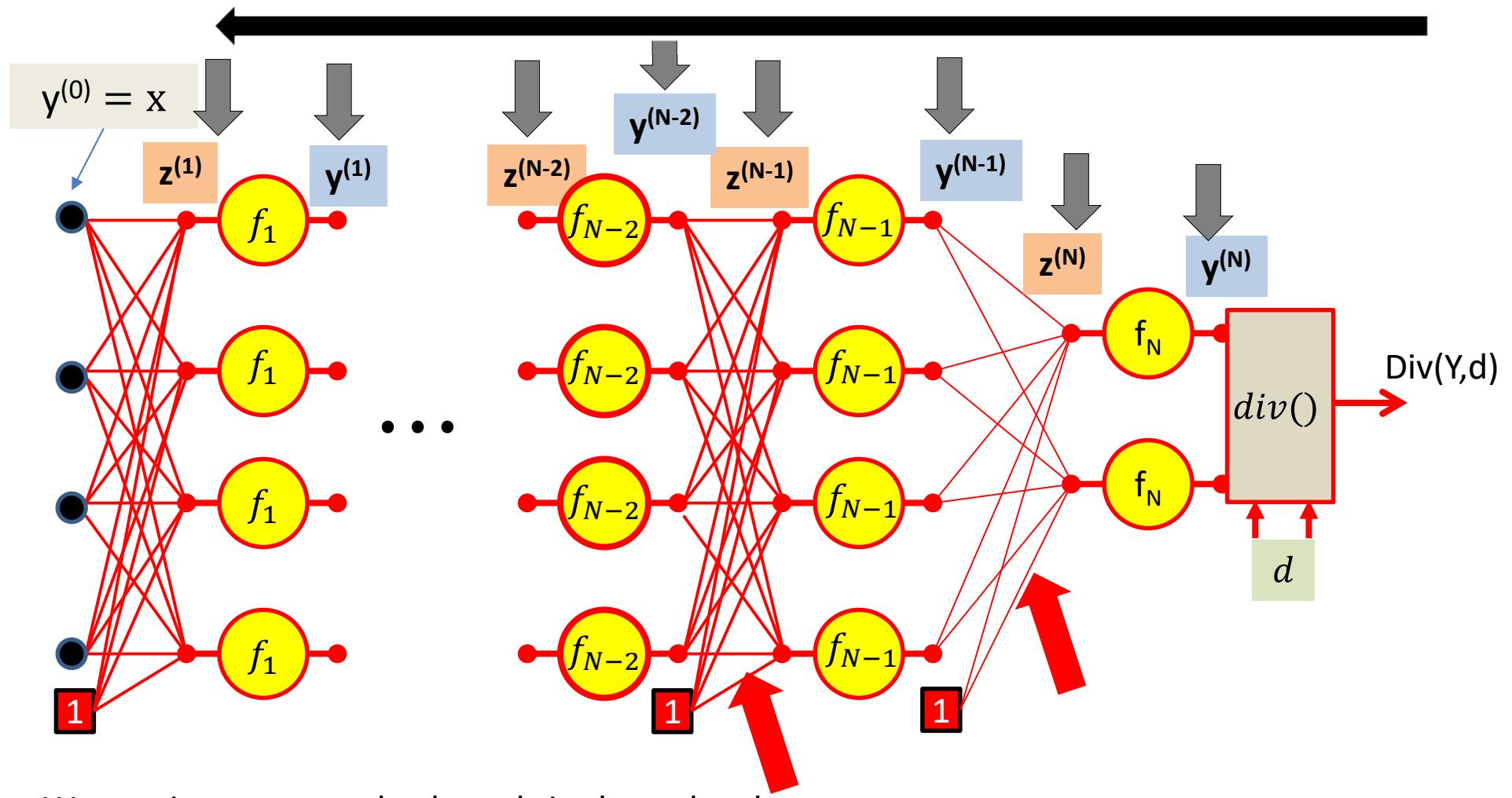
We continue our way backwards in the order shown

$$\nabla_{z^{(N-2)}} div(.)$$



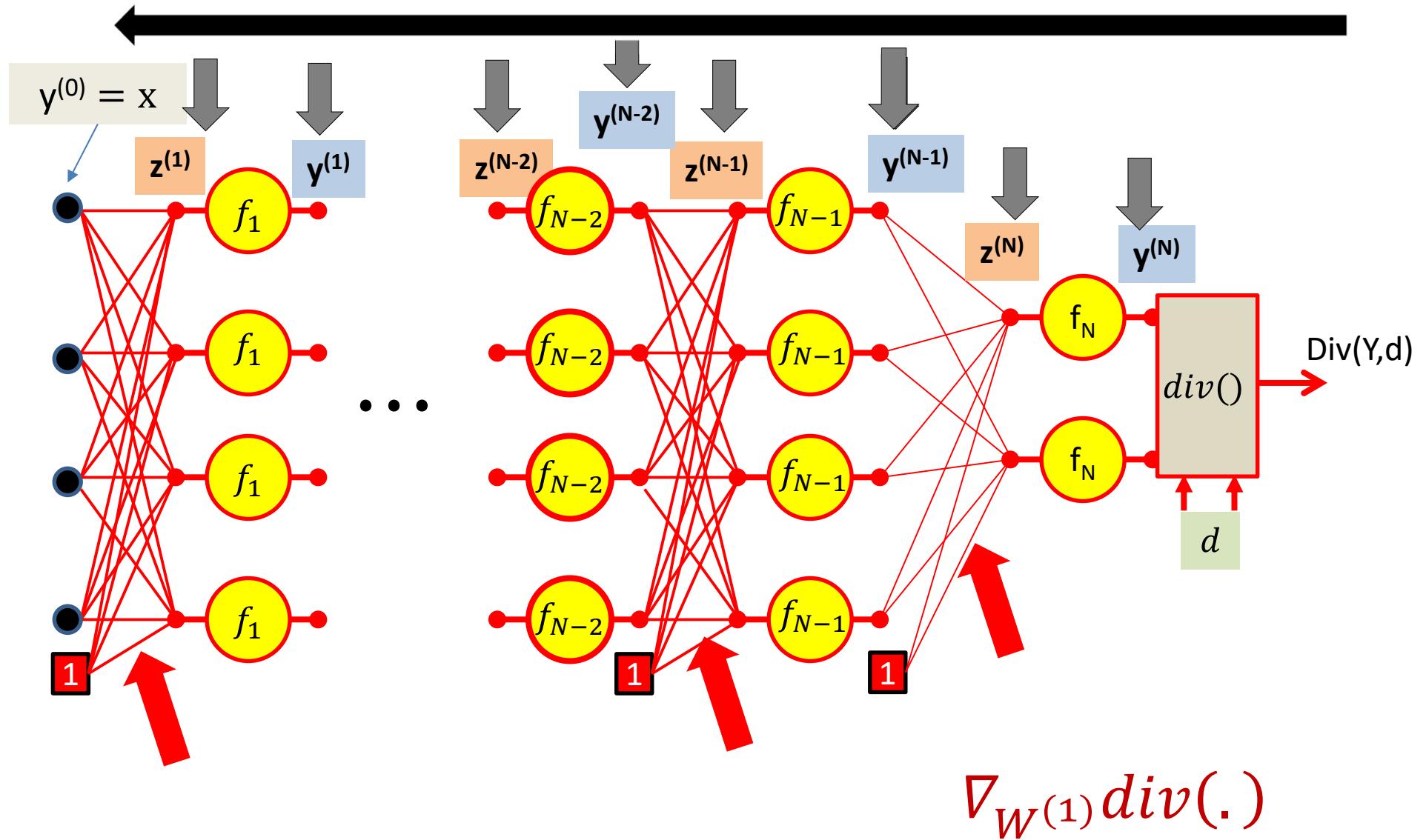
We continue our way backwards in the order shown

$$\nabla_{Y^{(1)}} \text{div}(\cdot)$$



We continue our way backwards in the order shown

$$\nabla_{z^{(1)}} div(.)$$

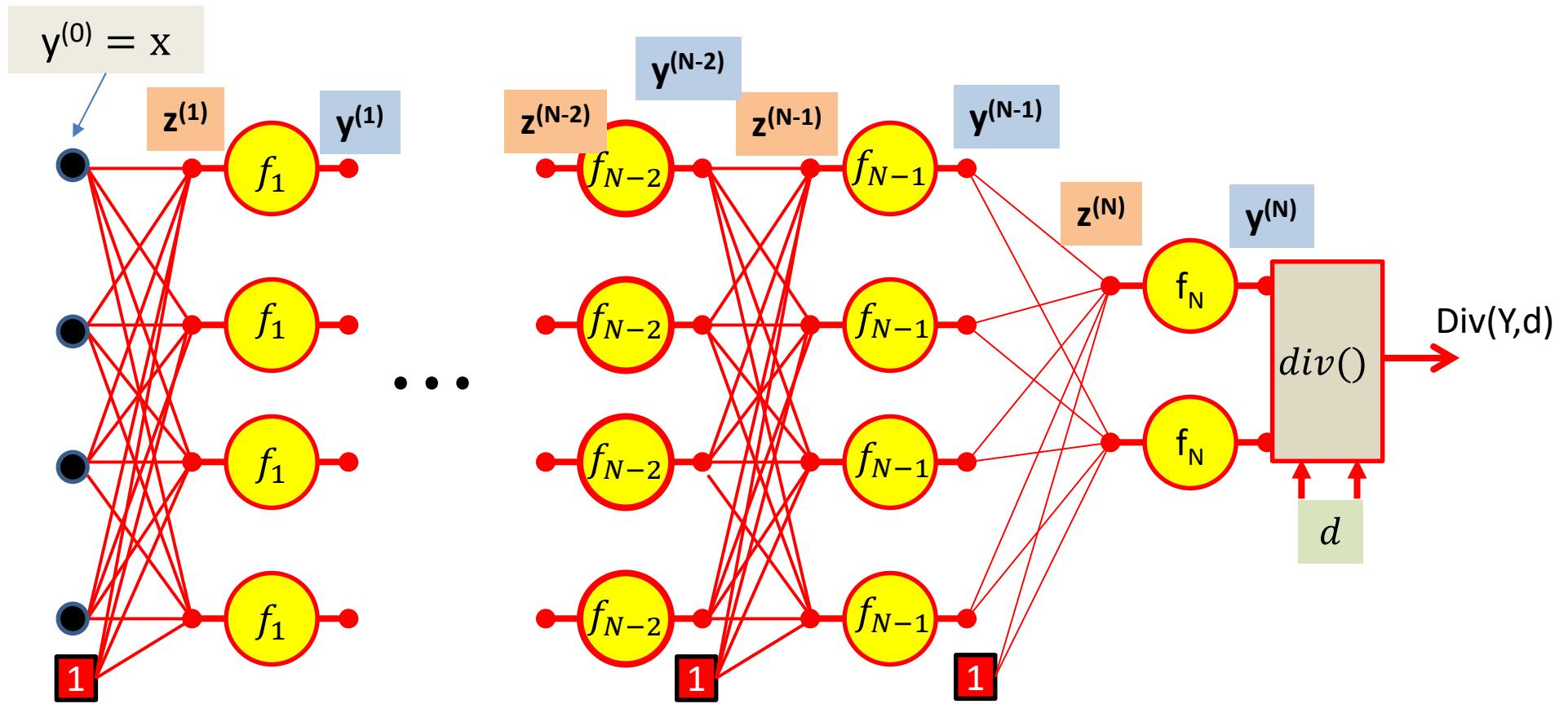


We continue our way backwards in the order shown

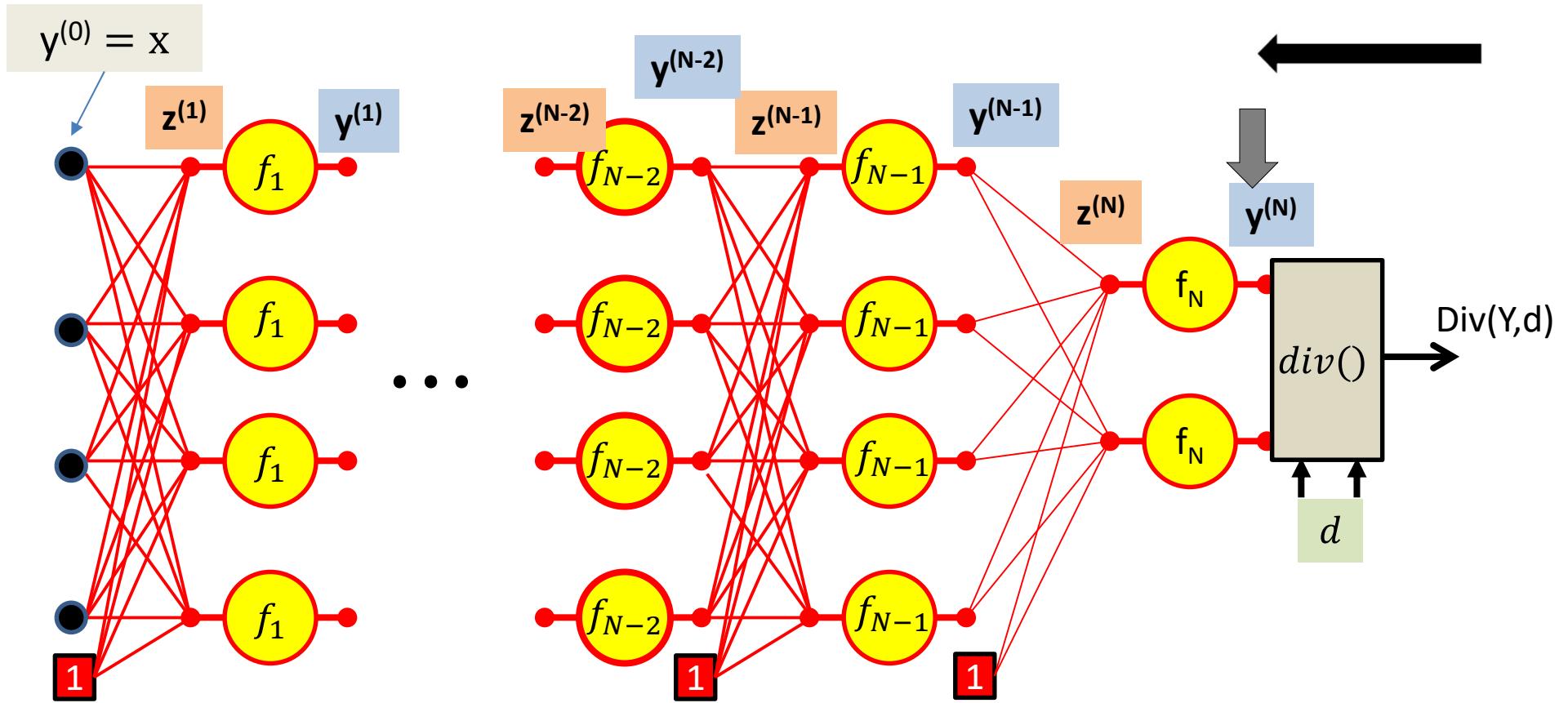
Backward Gradient Computation

- Lets actually see the math..

Computing derivatives



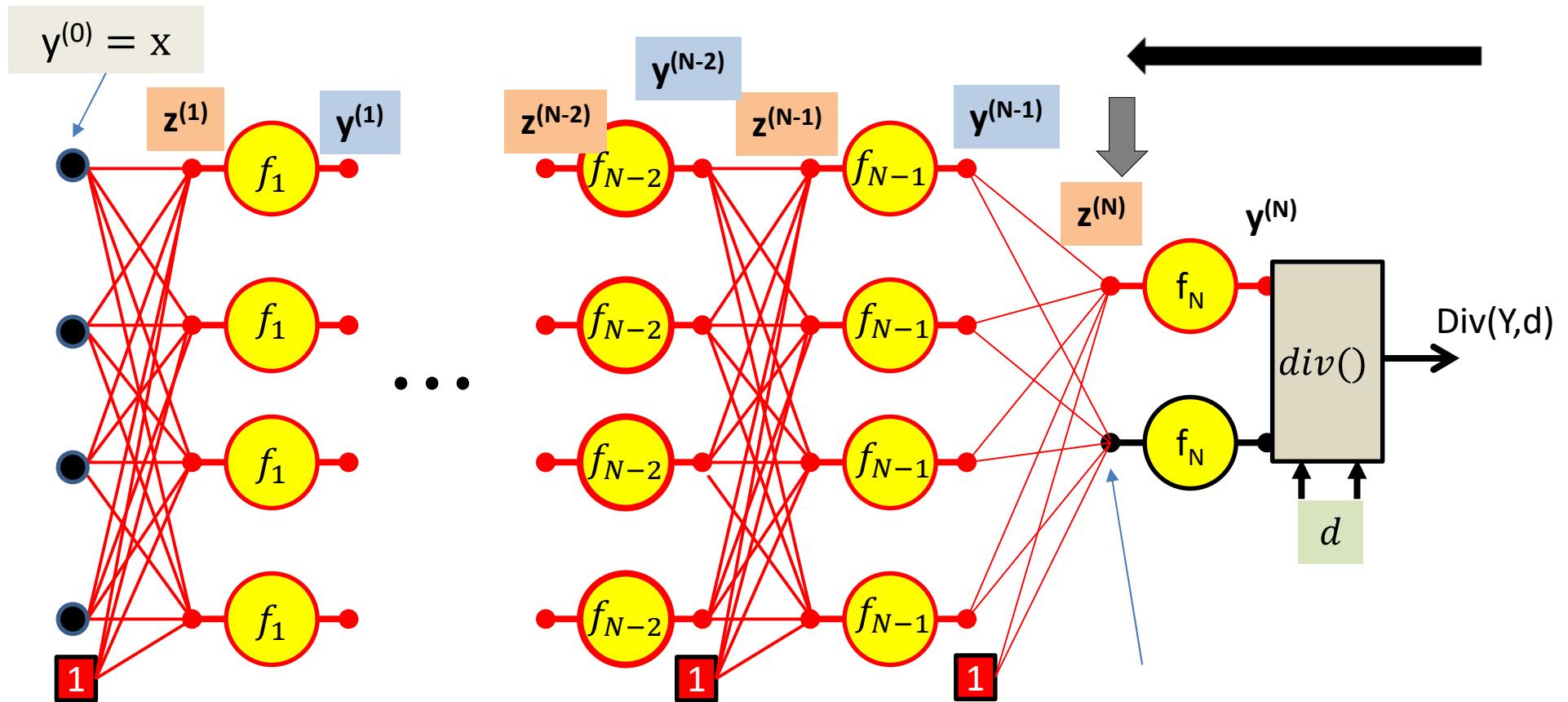
Computing derivatives



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

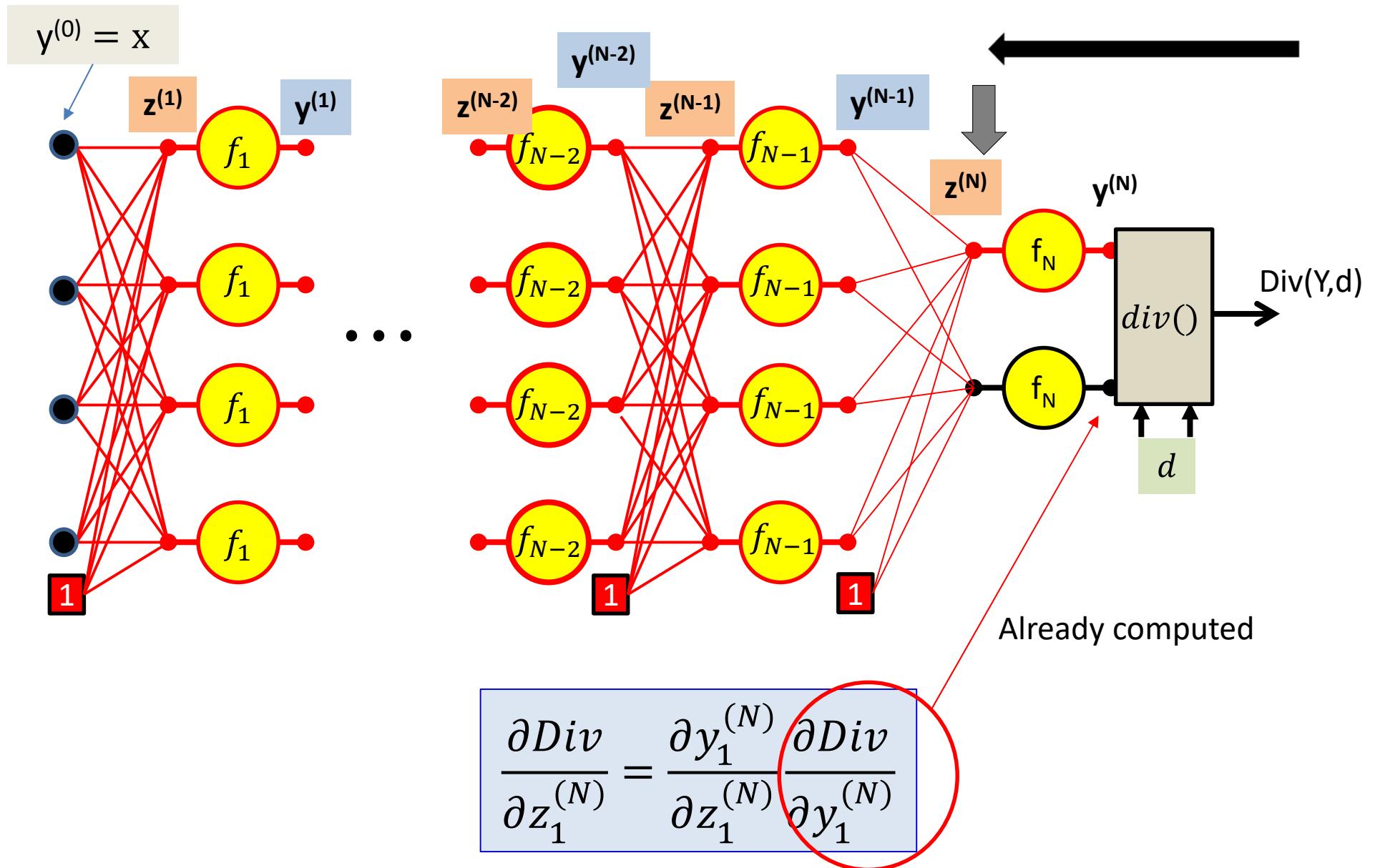
$$\frac{\partial Div(Y, d)}{\partial y_i^{(N)}} = \frac{\partial Div(Y, d)}{\partial y_i}$$

Computing derivatives

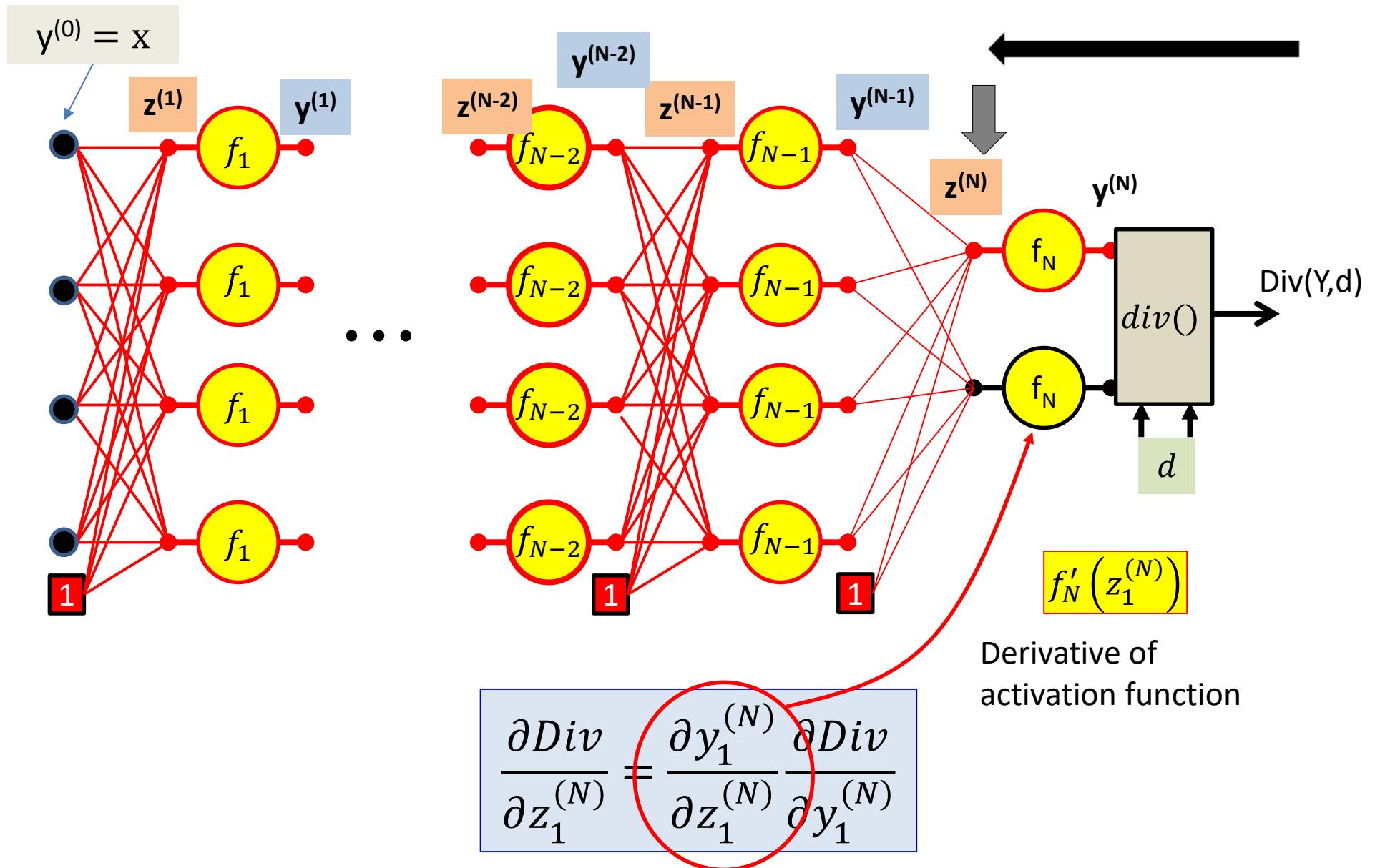


$$\frac{\partial Div}{\partial z_1^{(N)}} = \frac{\partial y_1^{(N)}}{\partial z_1^{(N)}} \frac{\partial Div}{\partial y_1^{(N)}}$$

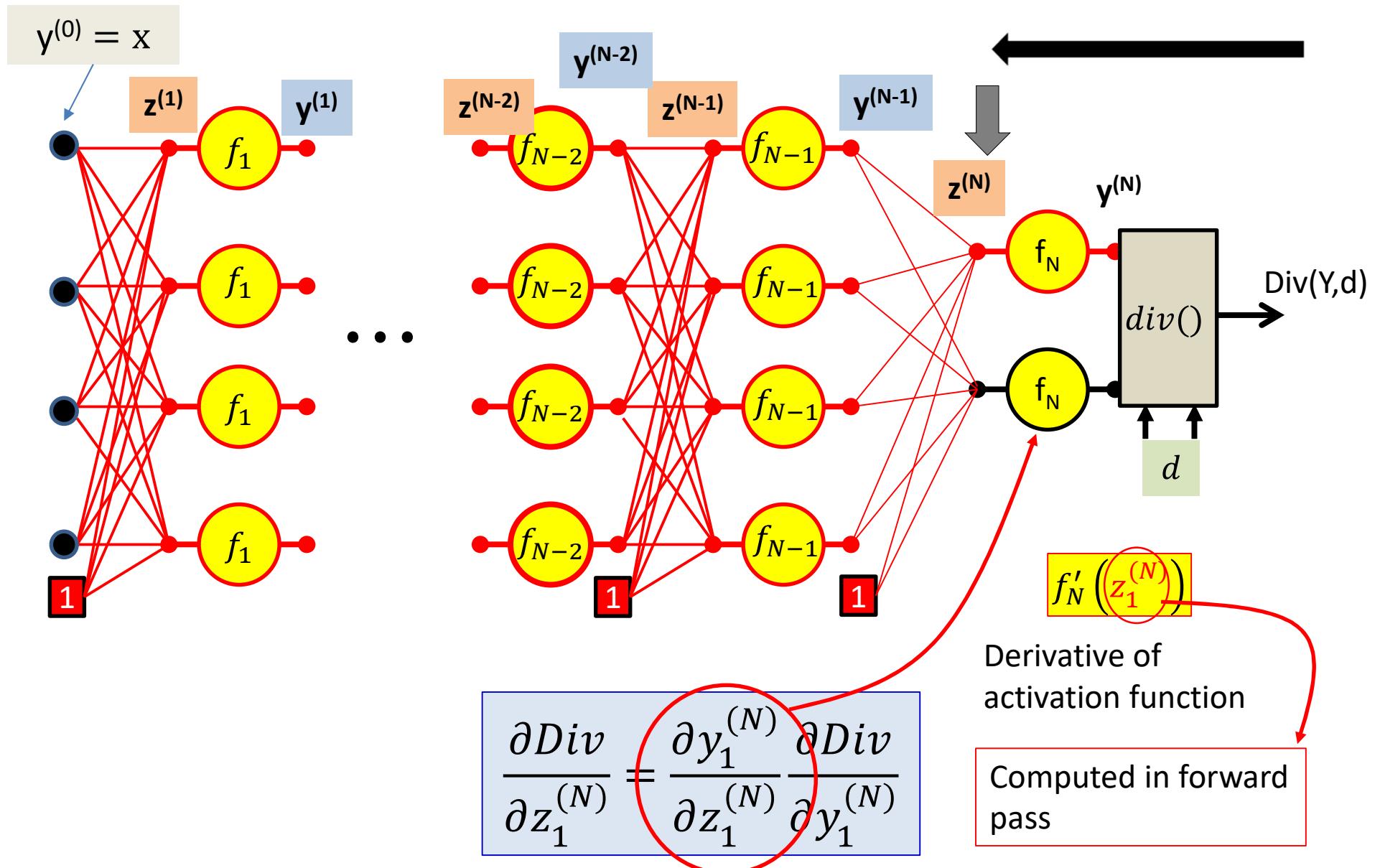
Computing derivatives



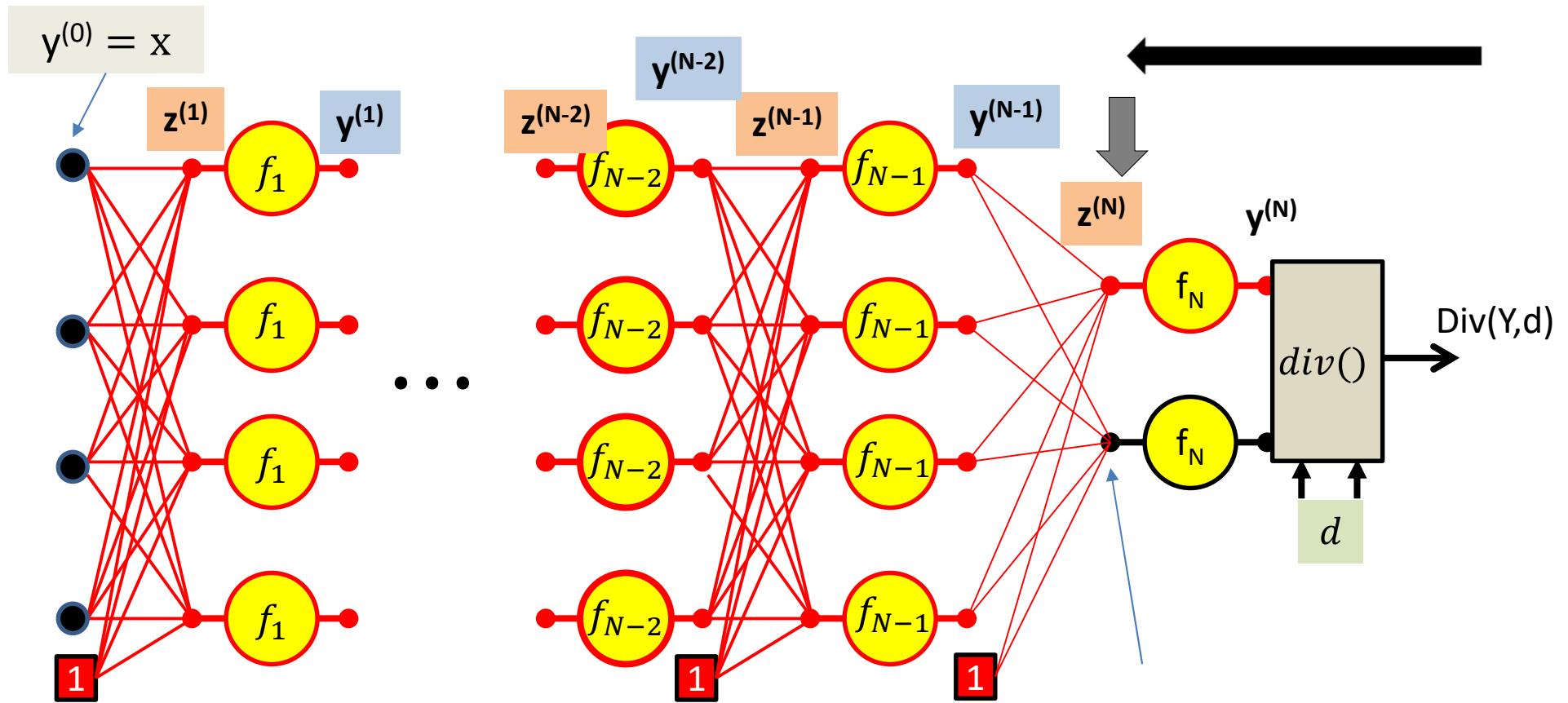
Computing derivatives



Computing derivatives

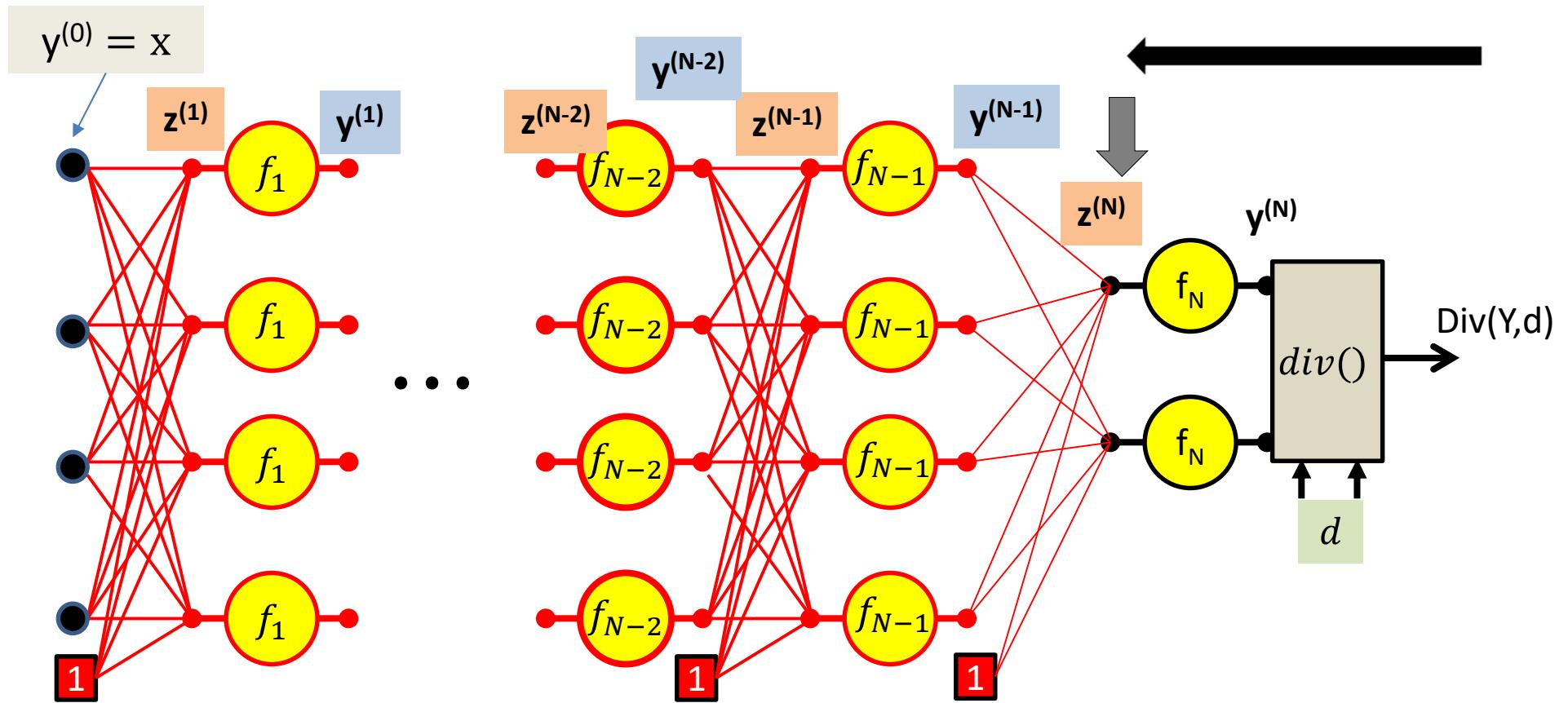


Computing derivatives



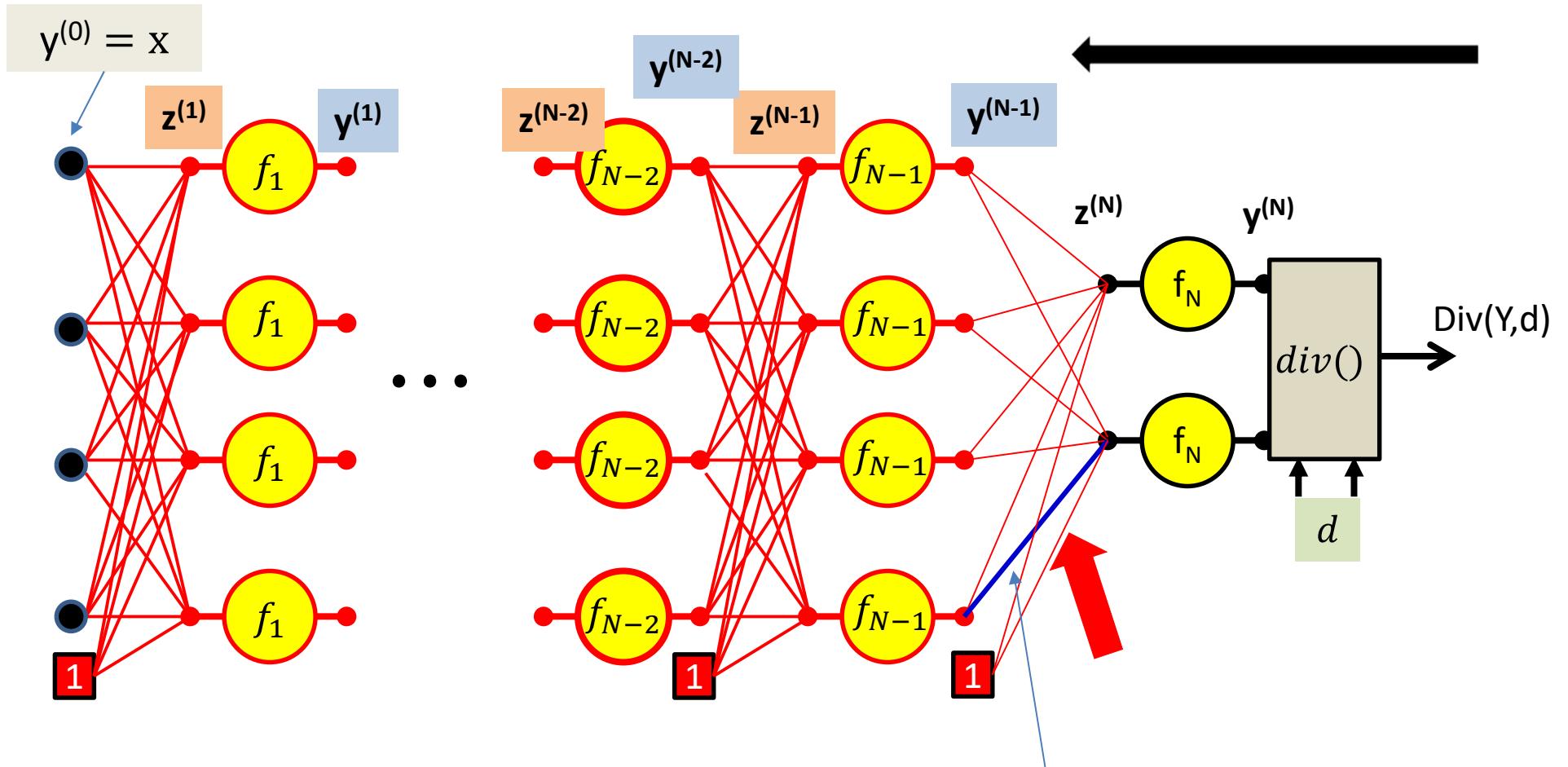
$$\frac{\partial \text{Div}}{\partial z_1^{(N)}} = f'_N(z_1^{(N)}) \frac{\partial \text{Div}}{\partial y_1^{(N)}}$$

Computing derivatives



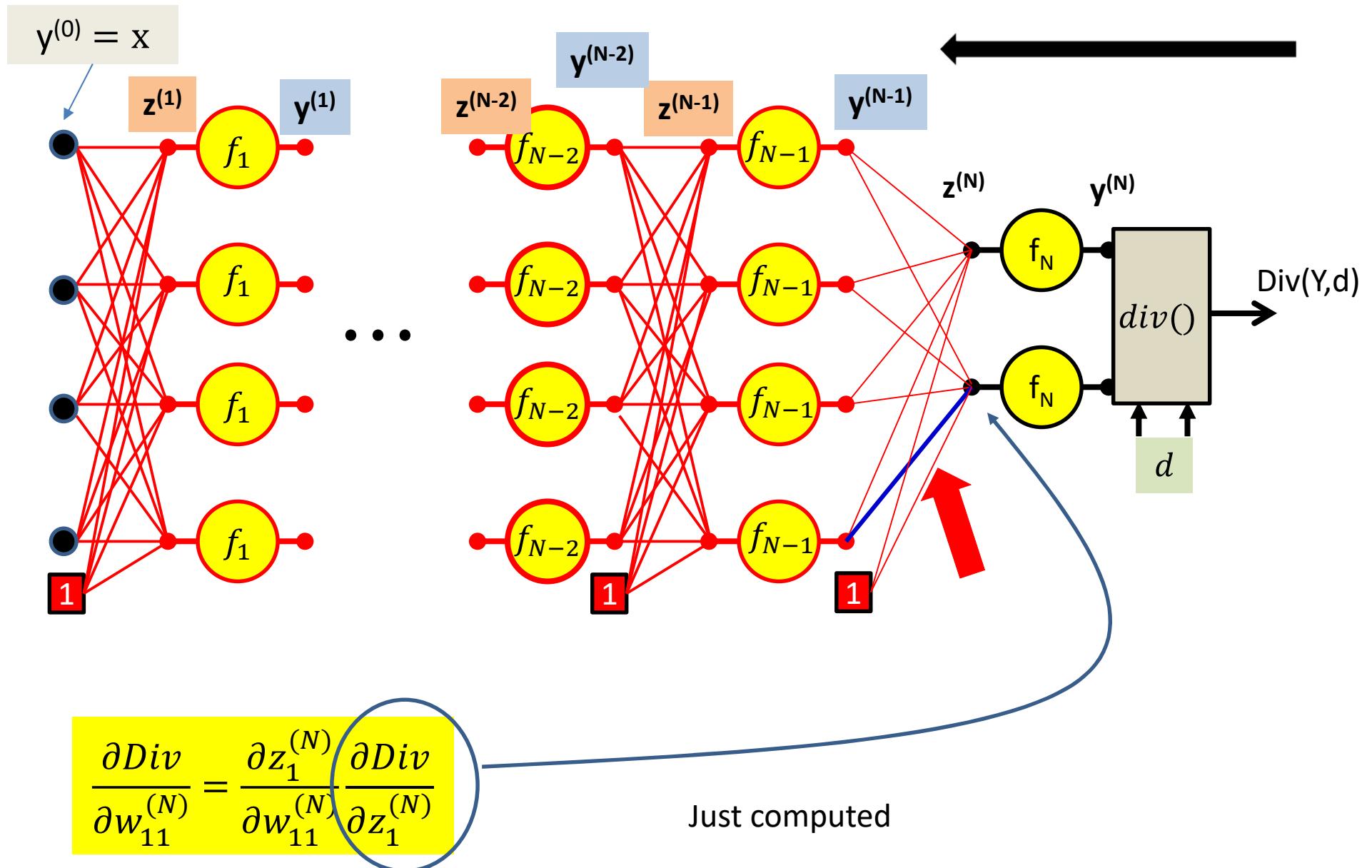
$$\frac{\partial \text{Div}}{\partial z_i^{(N)}} = f'_N(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y_i^{(N)}}$$

Computing derivatives

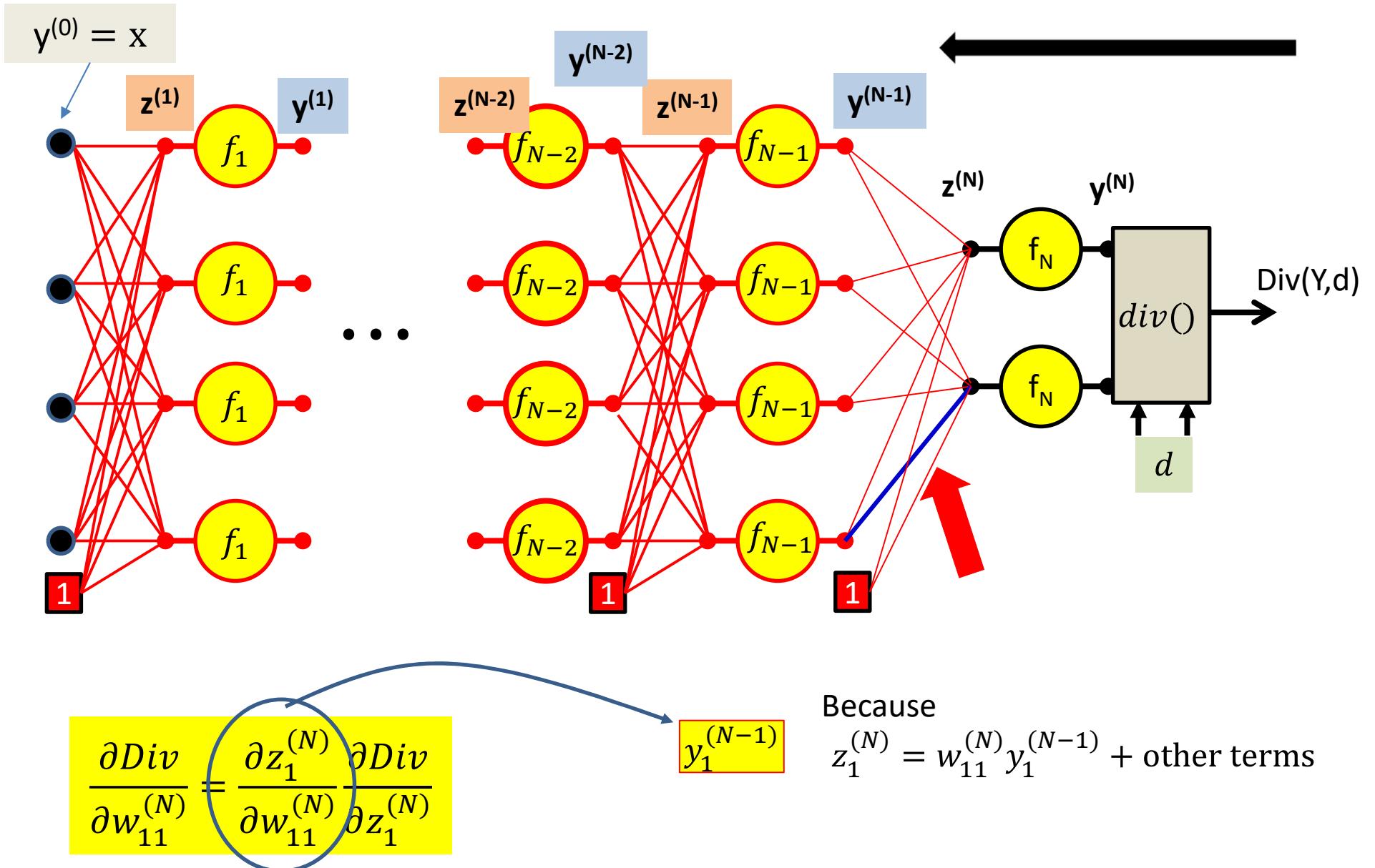


$$\frac{\partial \text{Div}}{\partial w_{11}^{(N)}} = \frac{\partial z_1^{(N)}}{\partial w_{11}^{(N)}} \frac{\partial \text{Div}}{\partial z_1^{(N)}}$$

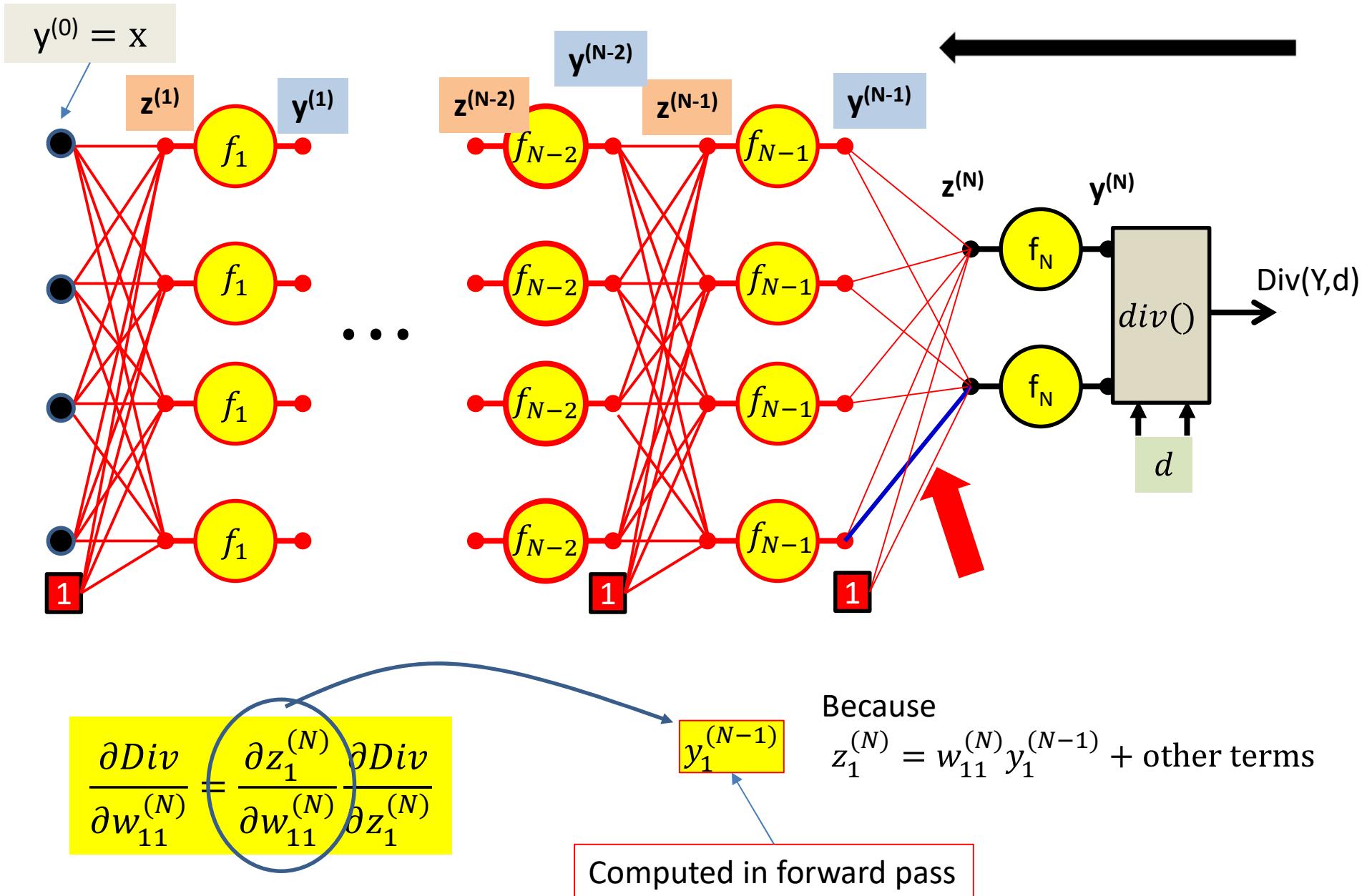
Computing derivatives



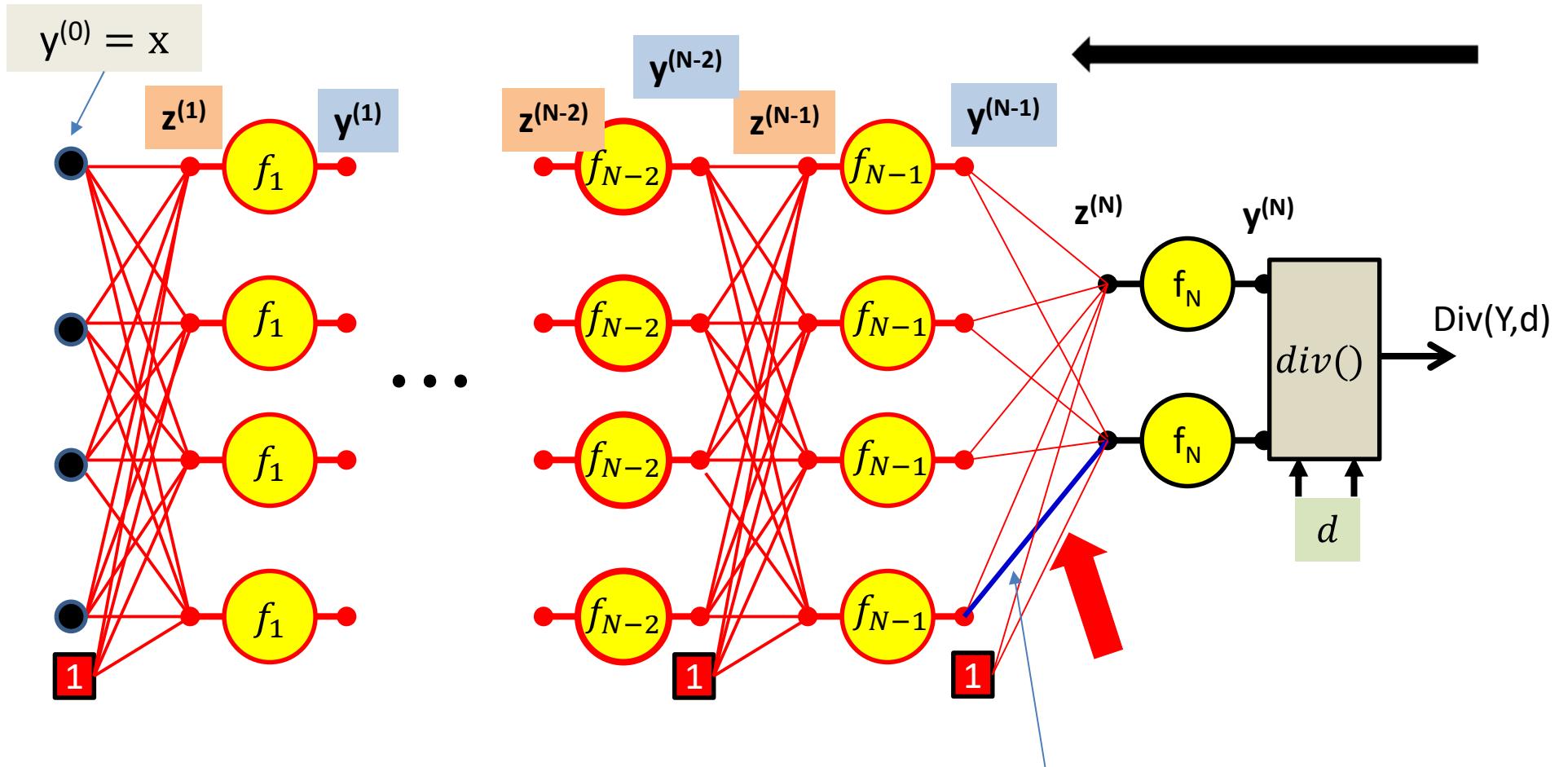
Computing derivatives



Computing derivatives

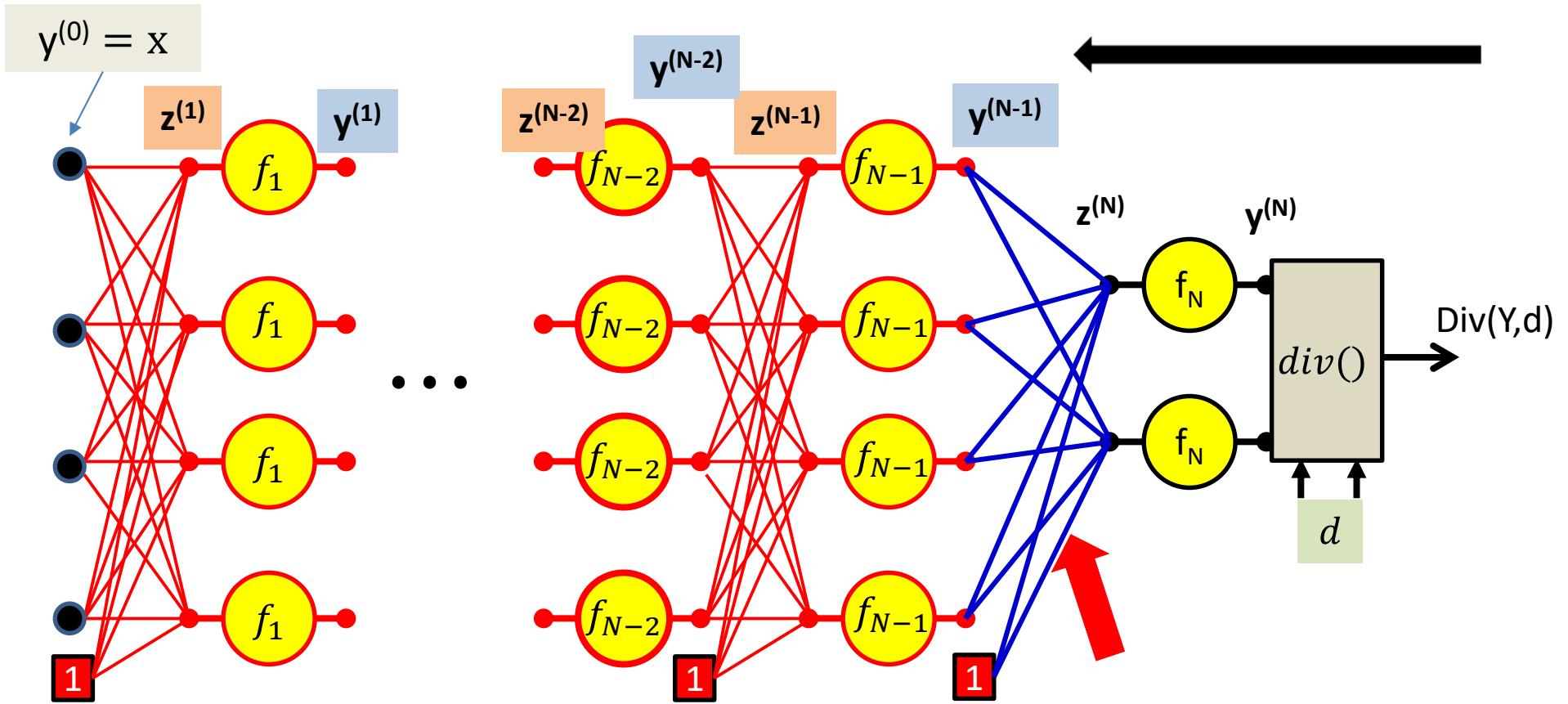


Computing derivatives



$$\frac{\partial Div}{\partial w_{11}^{(N)}} = y_1^{(N-1)} \frac{\partial Div}{\partial z_1^{(N)}}$$

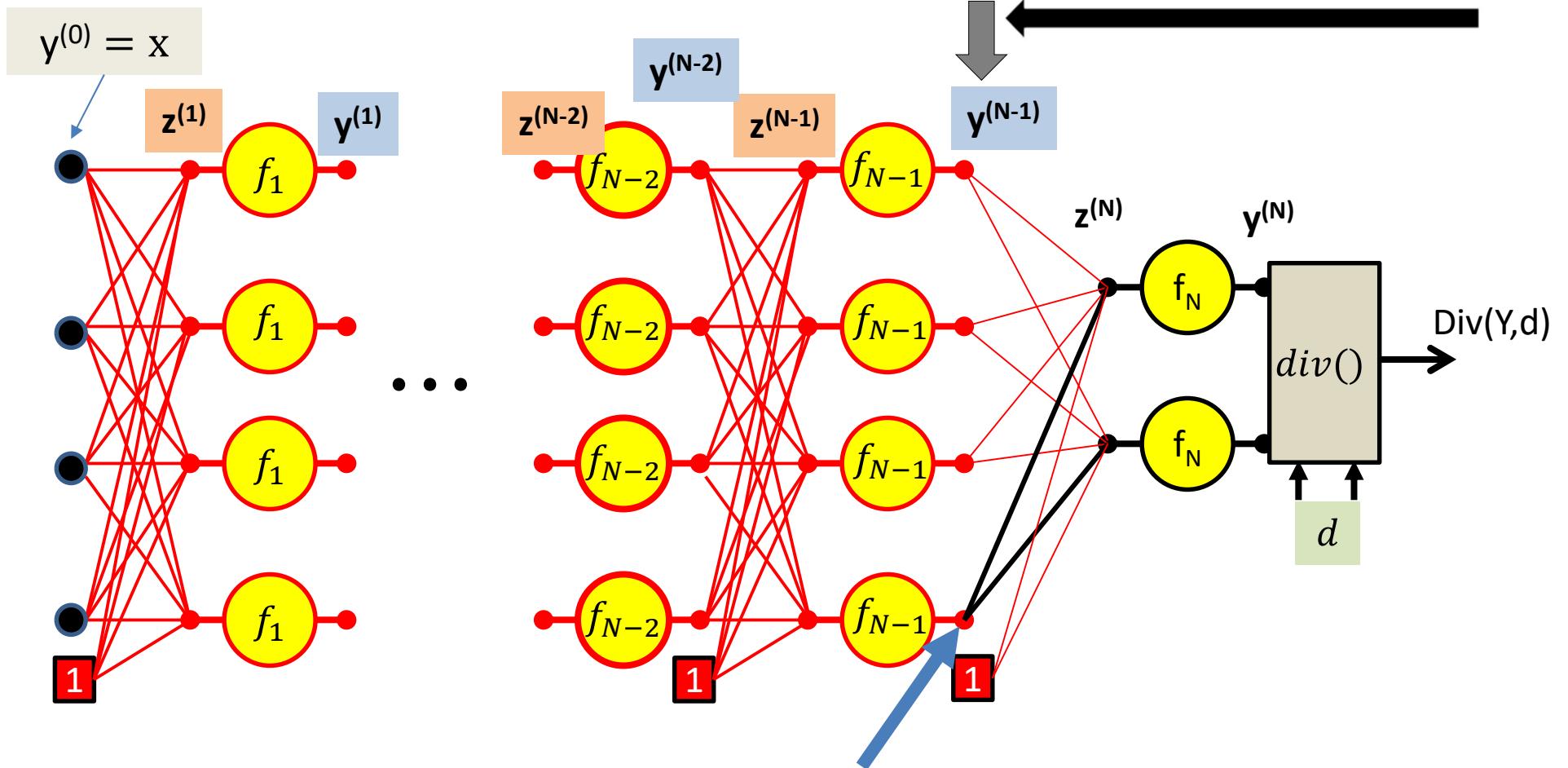
Computing derivatives



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

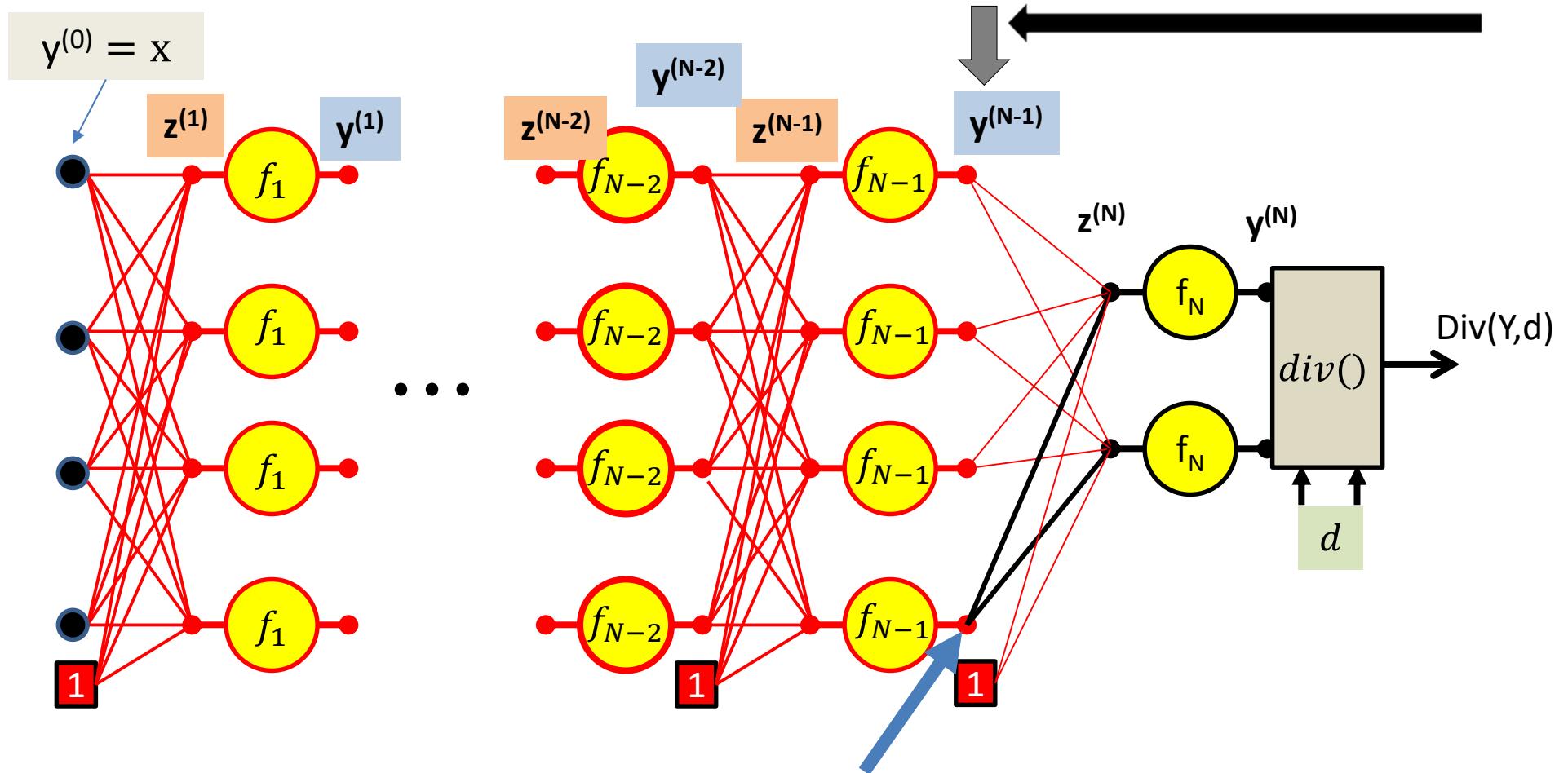
For the bias term $y_0^{(N-1)} = 1$

Computing derivatives



$$\frac{\partial Div}{\partial y_1^{(N-1)}} = \sum_j \frac{\partial z_j^{(N)}}{\partial y_1^{(N-1)}} \frac{\partial Div}{\partial z_j^{(N)}}$$

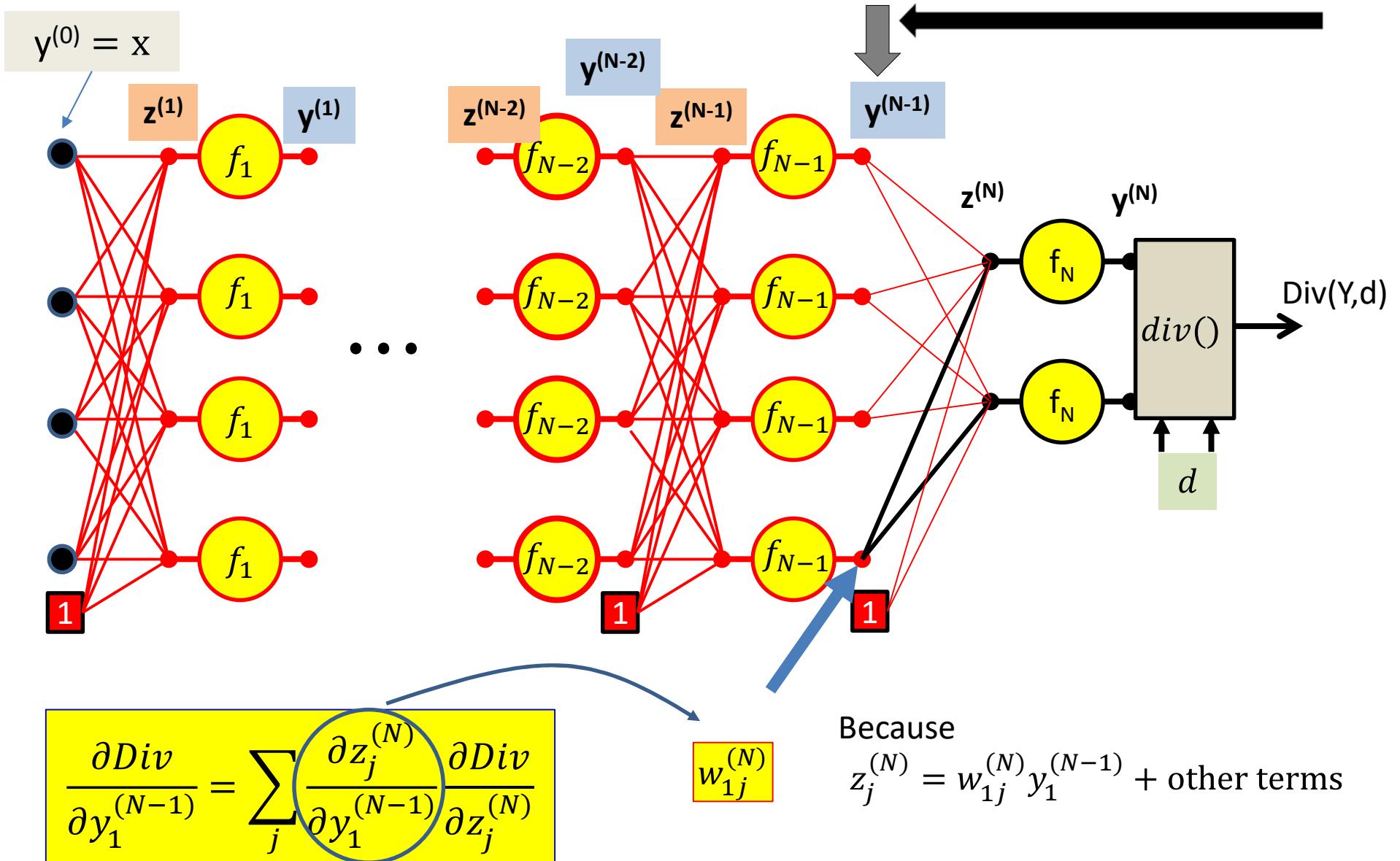
Computing derivatives



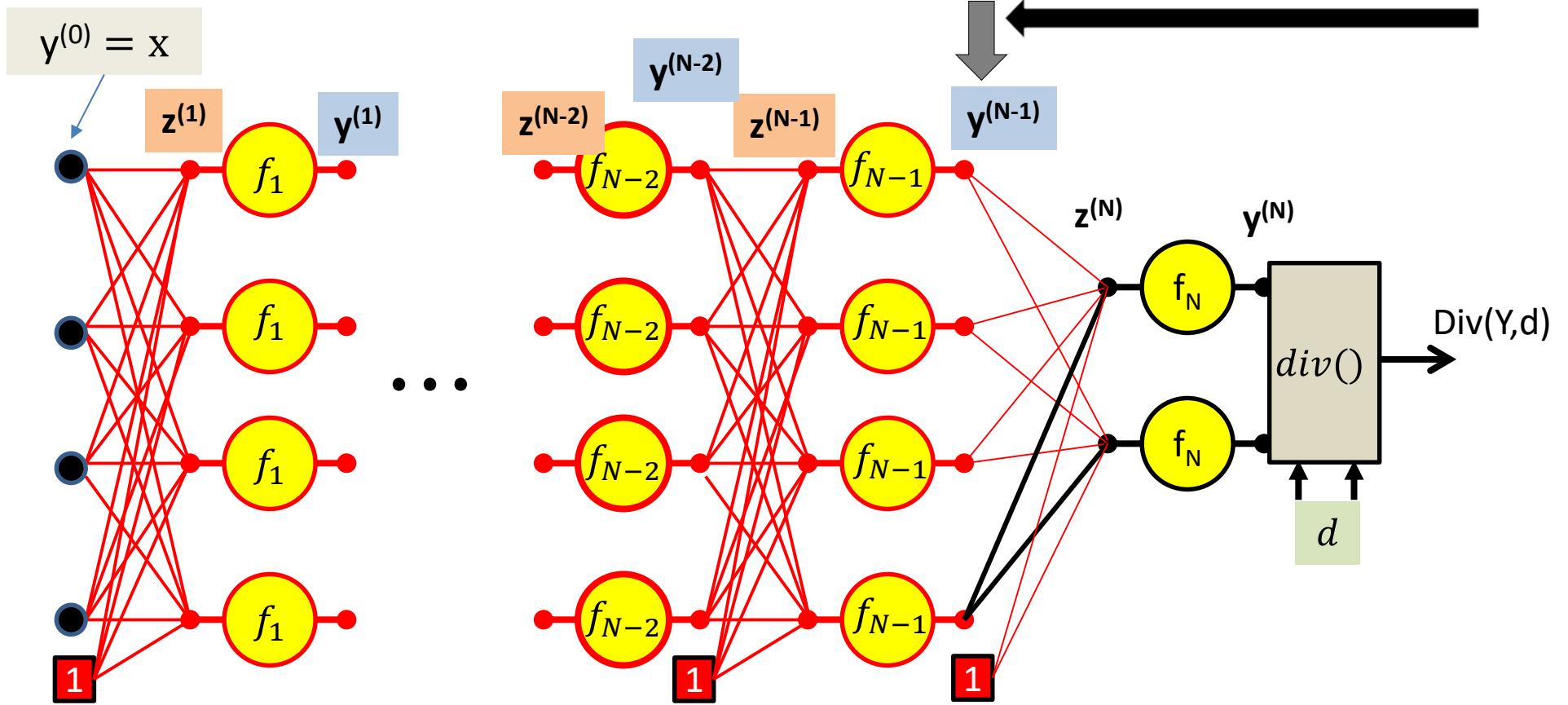
$$\frac{\partial Div}{\partial y_1^{(N-1)}} = \sum_j \frac{\partial z_j^{(N)}}{\partial y_1^{(N-1)}} \frac{\partial Div}{\partial z_j^{(N)}}$$

Already computed

Computing derivatives

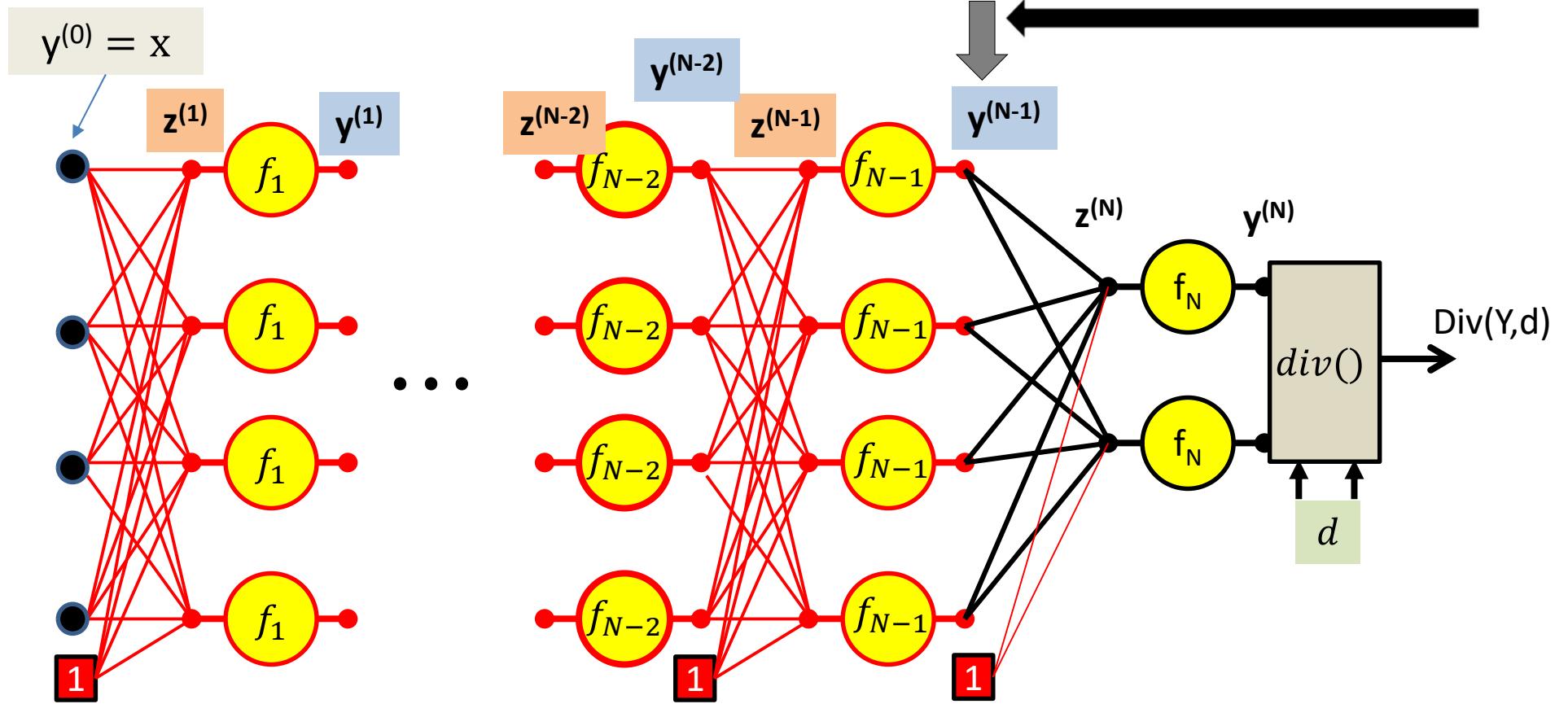


Computing derivatives



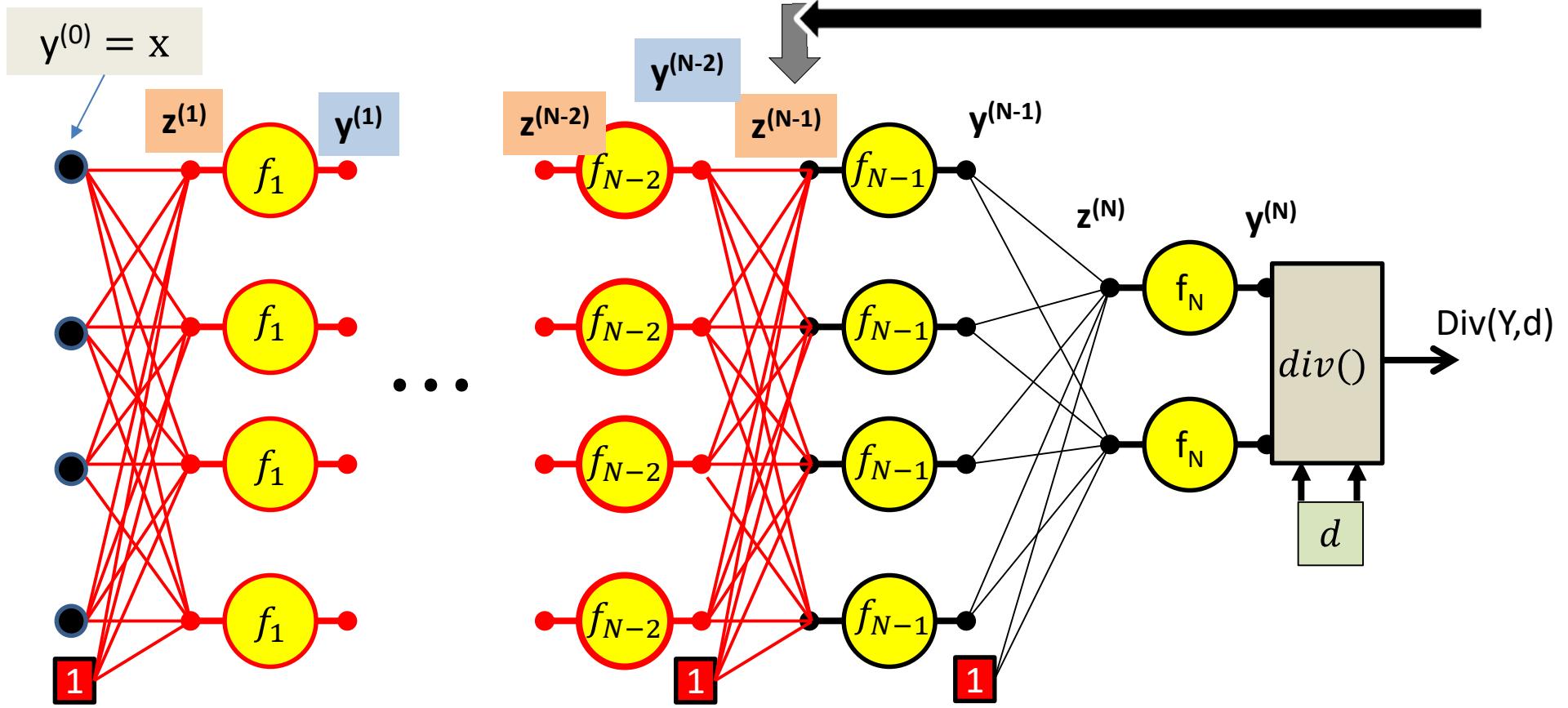
$$\frac{\partial \text{Div}}{\partial y_1^{(N-1)}} = \sum_j w_{1j}^{(N)} \frac{\partial \text{Div}}{\partial z_j^{(N)}}$$

Computing derivatives



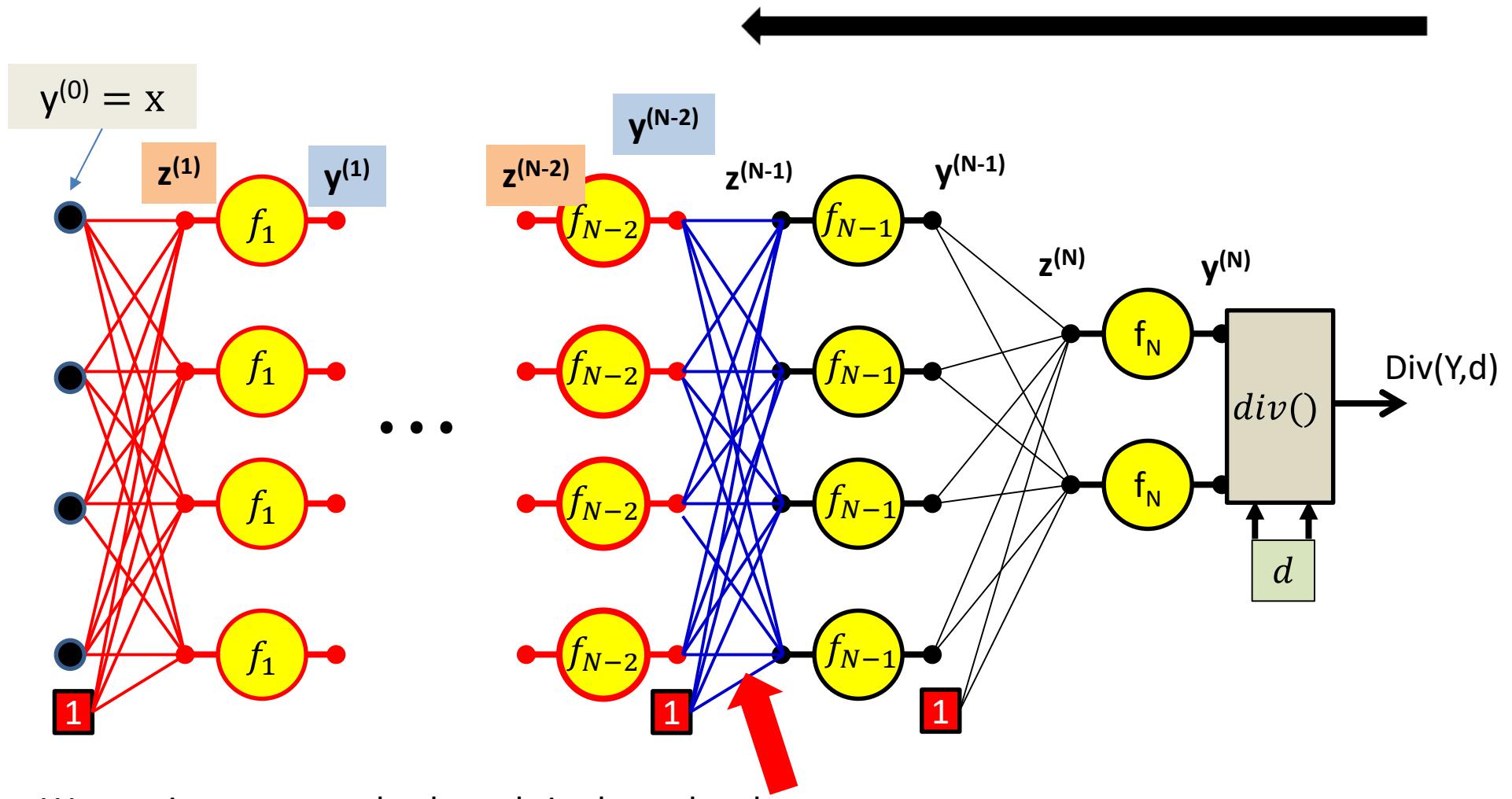
$$\frac{\partial Div}{\partial y_i^{(N-1)}} = \sum_j w_{ij}^{(N)} \frac{\partial Div}{\partial z_j^{(N)}}$$

Computing derivatives



We continue our way backwards in the order shown

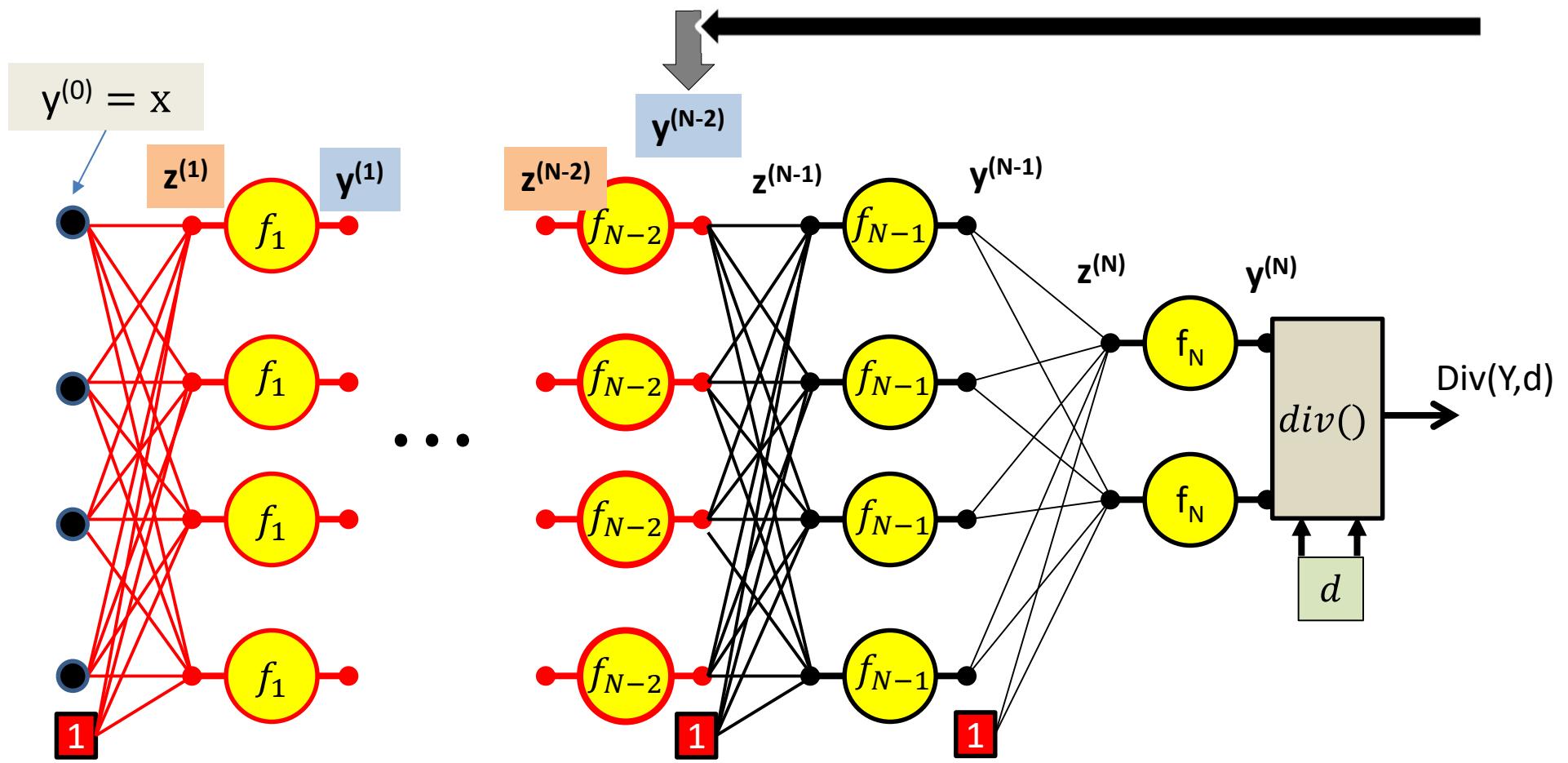
$$\frac{\partial Div}{\partial z_i^{(N-1)}} = f'_{N-1}(z_i^{(N-1)}) \frac{\partial Div}{\partial y_i^{(N-1)}}$$



We continue our way backwards in the order shown

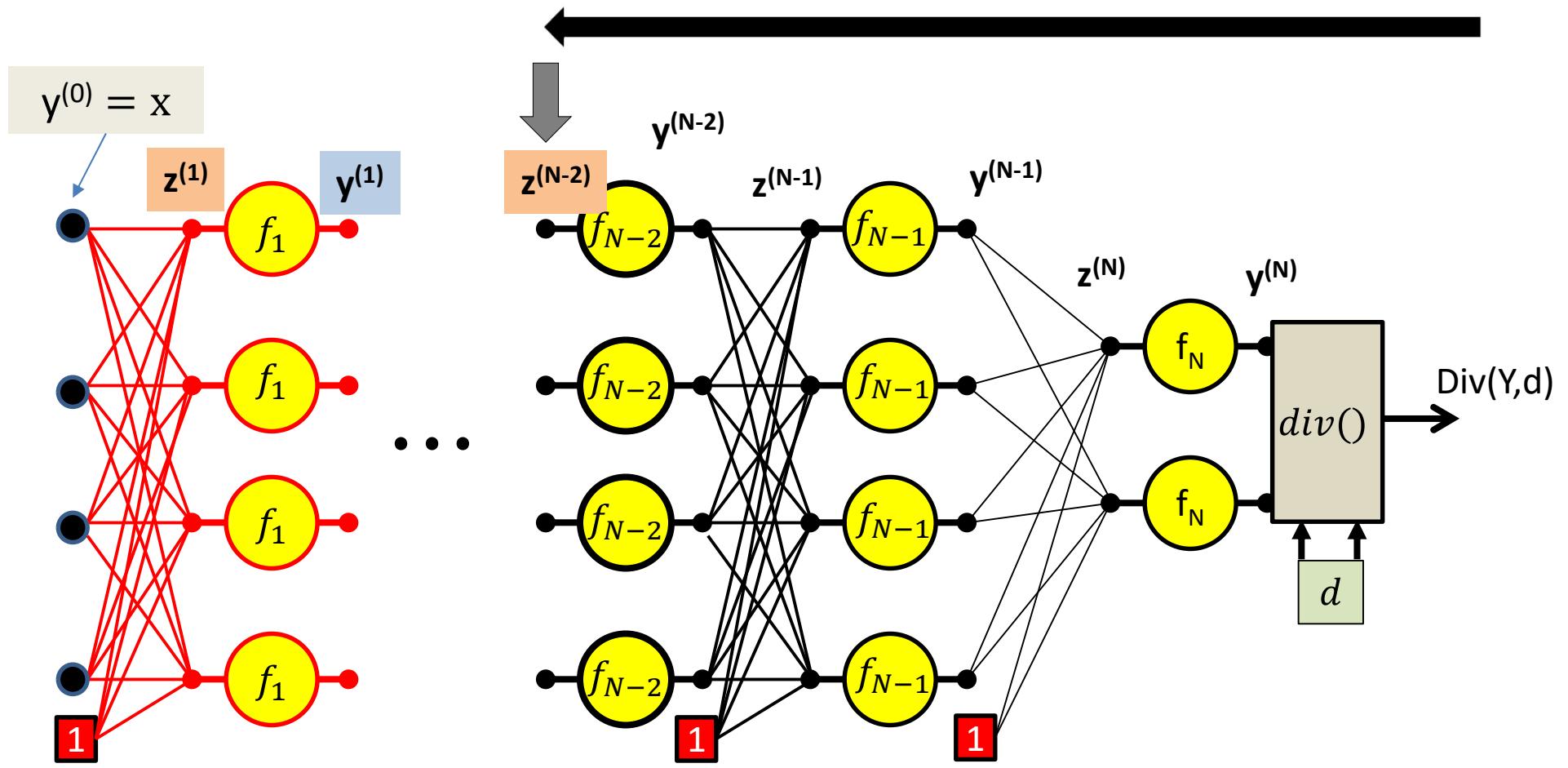
$$\frac{\partial \text{Div}}{\partial w_{ij}^{(N-1)}} = y_i^{(N-2)} \frac{\partial \text{Div}}{\partial z_j^{(N-1)}}$$

For the bias term $y_0^{(N-2)} = 1$



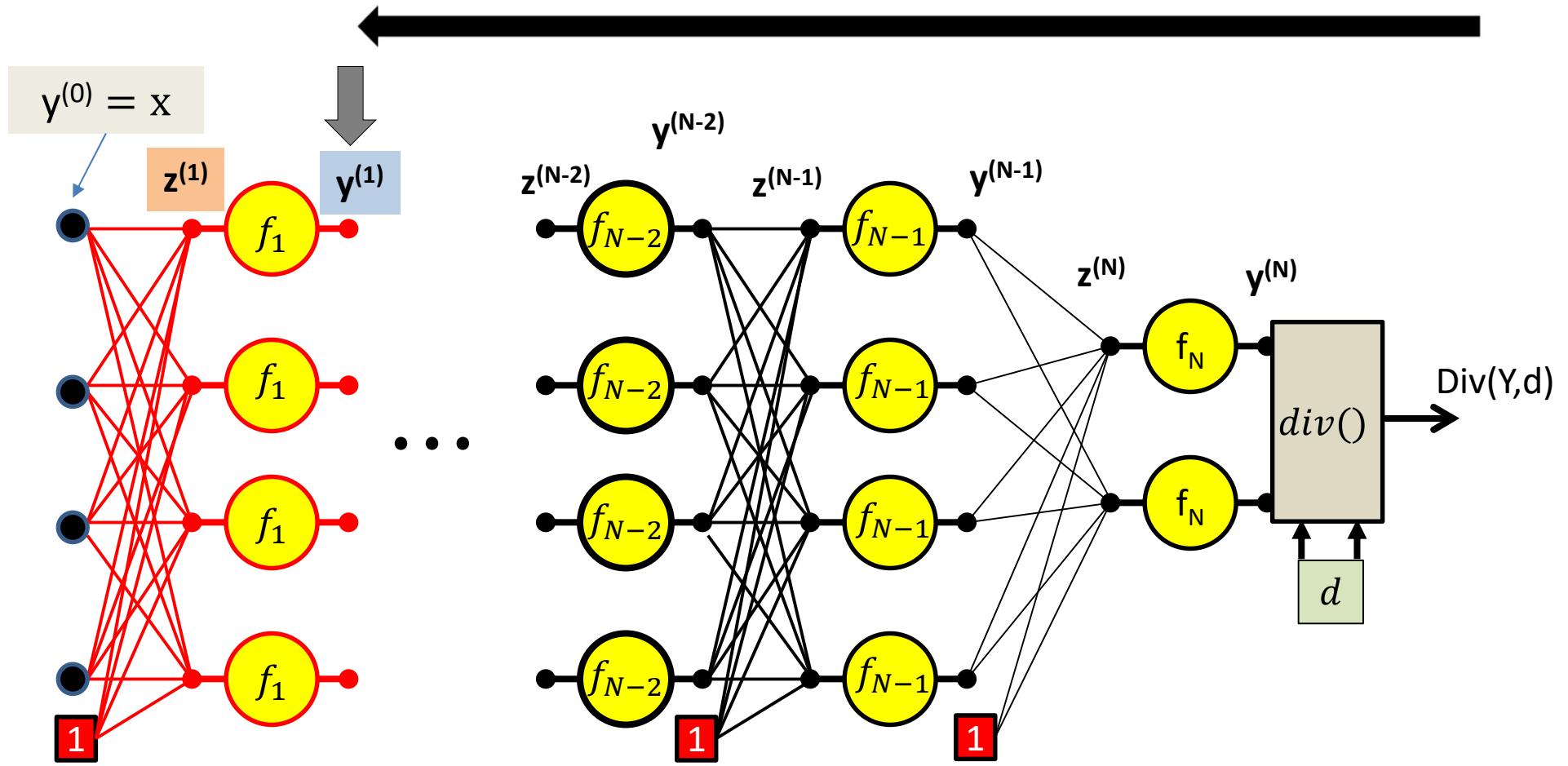
We continue our way backwards in the order shown

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



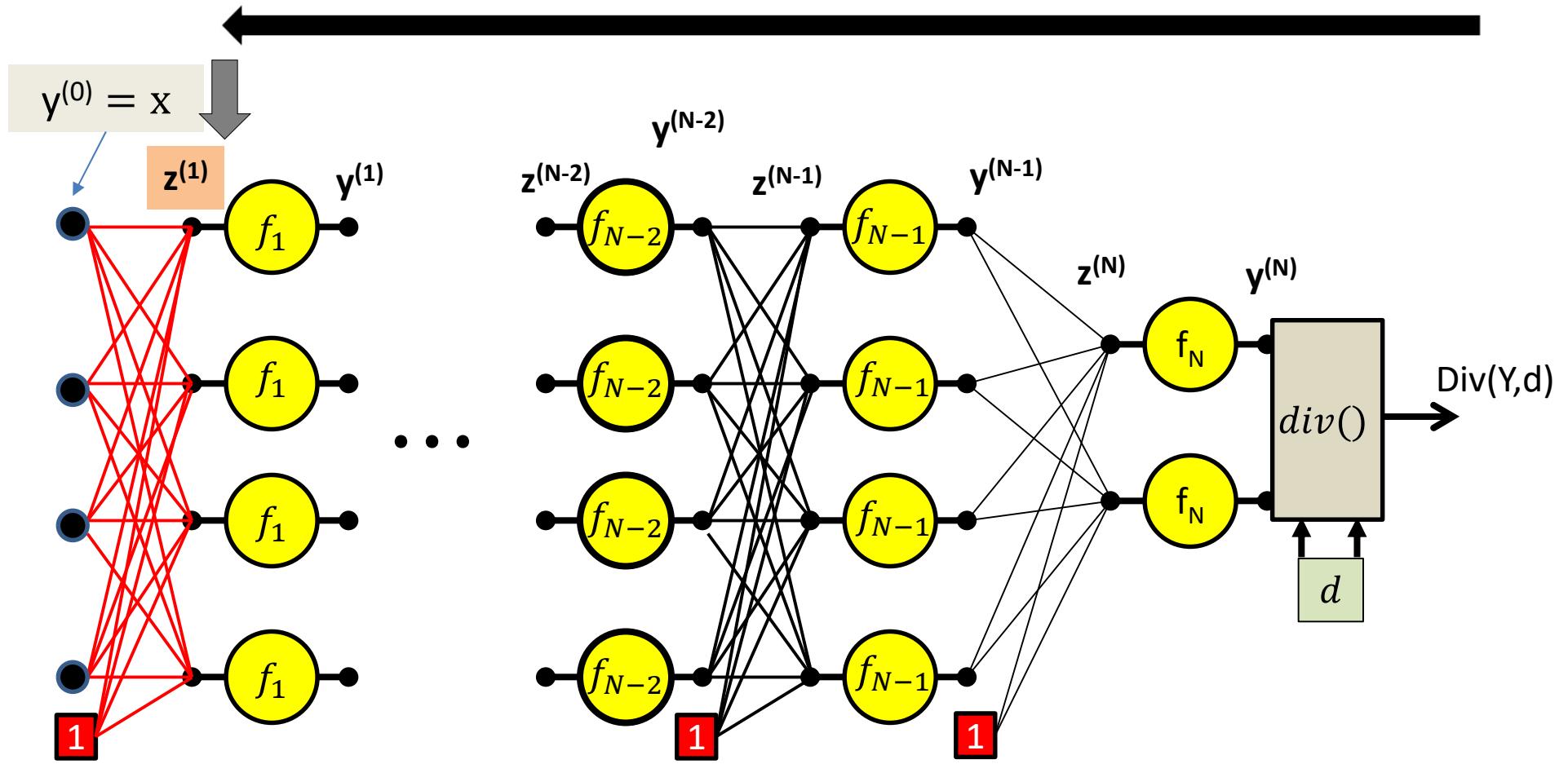
We continue our way backwards in the order shown

$$\frac{\partial Div}{\partial z_i^{(N-2)}} = f'_{N-2}(z_i^{(N-2)}) \frac{\partial Div}{\partial y_i^{(N-2)}}$$



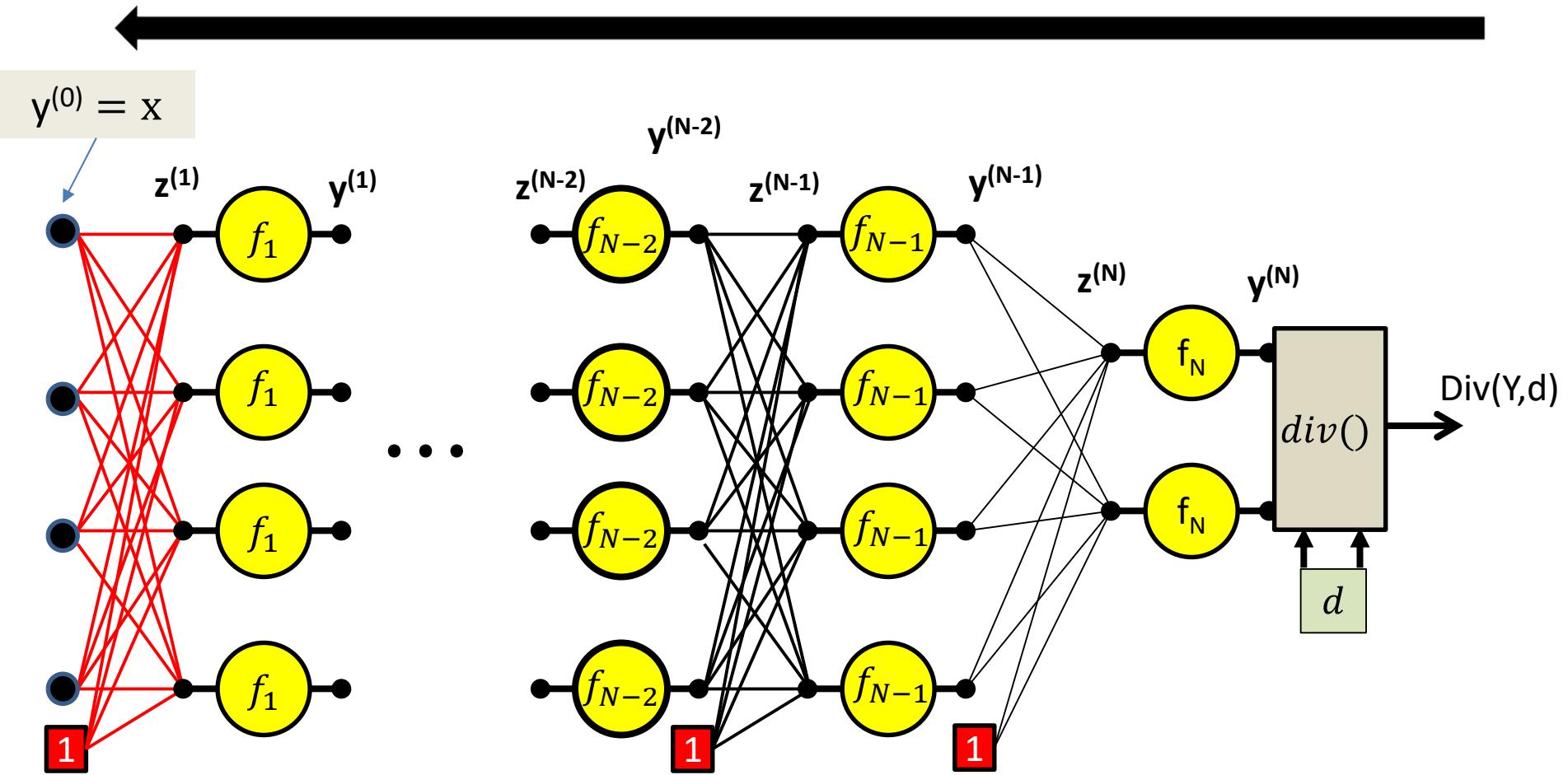
We continue our way backwards in the order shown

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



We continue our way backwards in the order shown

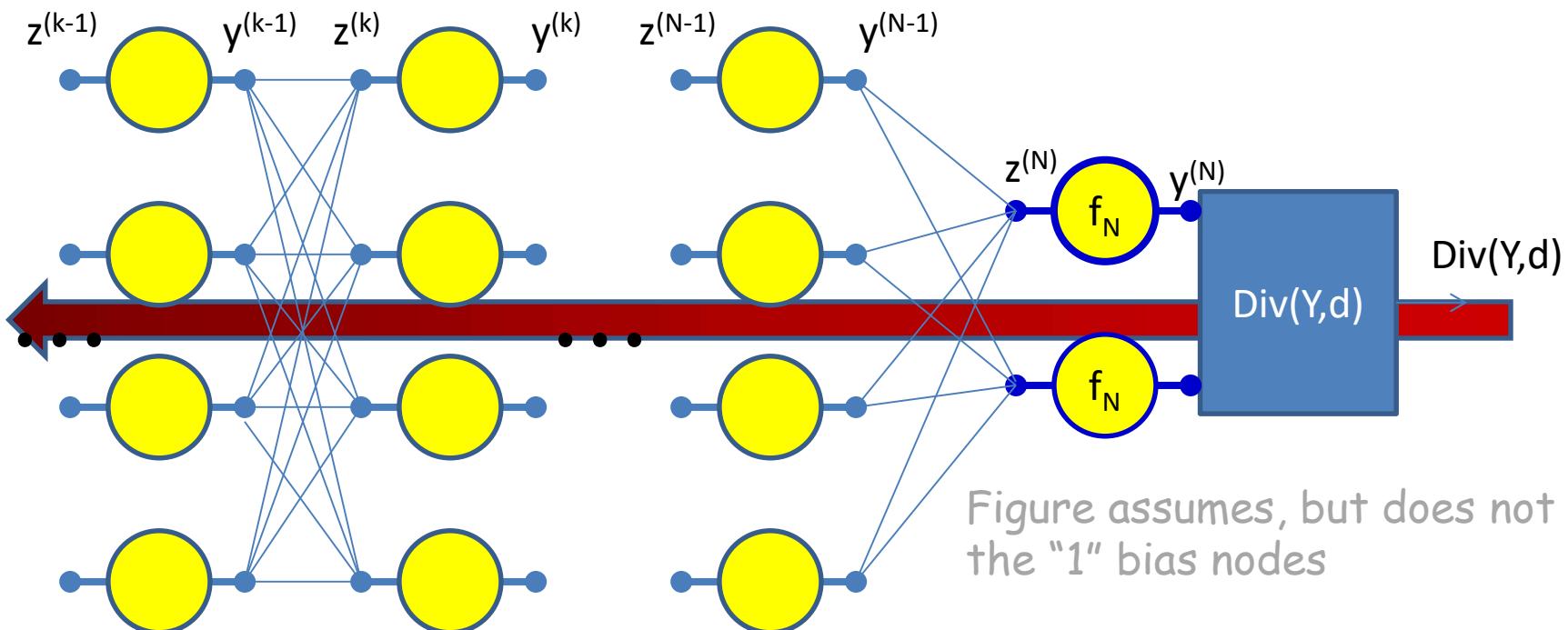
$$\frac{\partial Div}{\partial z_i^{(1)}} = f'_1(z_i^{(1)}) \frac{\partial Div}{\partial y_i^{(1)}}$$



We continue our way backwards in the order shown

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

Gradients: Backward Computation



Initialize: Gradient
w.r.t network output

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

$$\frac{\partial \text{Div}}{\partial z_i^{(N)}} = f'_k(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y_i^{(N)}}$$

For $k = N - 1..0$
For $i = 1: \text{layer width}$

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

$$\frac{\partial \text{Div}}{\partial z_i^{(k)}} = f'_k(z_i^{(k)}) \frac{\partial \text{Div}}{\partial y_i^{(k)}}$$

$$\forall j \frac{\partial \text{Div}}{\partial w_{ij}^{(k+1)}} = y_i^{(k)} \frac{\partial \text{Div}}{\partial z_j^{(k+1)}}$$

Backward Pass

- Output layer (N) :
 - For $i = 1 \dots D_N$
 - $\frac{\partial Div}{\partial y_i^{(N)}} = \frac{\partial Div(Y, d)}{\partial y_i}$
 - $\frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial Div}{\partial y_i^{(N)}} f'_N(z_i^{(N)})$
- For layer $k = N - 1$ down to 1
 - For $i = 1 \dots D_k$
 - $\frac{\partial Div}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial Div}{\partial z_j^{(k+1)}}$
 - $\frac{\partial Div}{\partial z_i^{(k)}} = \frac{\partial Div}{\partial y_i^{(k)}} f'_k(z_i^{(k)})$
 - $\frac{\partial Div}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \frac{\partial D}{\partial z_i^{(k+1)}} \text{ for } j = 1 \dots D_k$
 - $\frac{\partial Div}{\partial w_{ji}^{(1)}} = y_j^{(0)} \frac{\partial Div}{\partial z_i^{(1)}} \text{ for } j = 1 \dots D_0$

Backward Pass

- Output layer (N) :

- For $i = 1 \dots D_N$

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

- $\frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial Div}{\partial y_i^{(N)}} f'_N(z_i^{(N)})$

- For layer $k = N - 1$ down to 1

- For $i = 1 \dots D_k$

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

- $\frac{\partial D}{\partial z_i^{(k)}} = \frac{\partial Div}{\partial y_i^{(k)}} f'_k(z_i^{(k)})$

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

- $\frac{\partial Div}{\partial w_{ji}^{(1)}} = y_j^{(0)} \frac{\partial Div}{\partial z_i^{(1)}}$ for $j = 1 \dots D_0$

Called “**Backpropagation**” because the derivative of the loss is propagated “backwards” through the network

Very analogous to the forward pass:

Backward weighted combination of next layer

Backward equivalent of activation

Using notation $\dot{y} = \frac{\partial \text{Div}(Y, d)}{\partial y}$ etc (overdot represents derivative of Div w.r.t variable)

- Output layer (N) :

- For $i = 1 \dots D_N$

- $\dot{y}_i^{(N)} = \frac{\partial \text{Div}}{\partial y_i}$

- $\dot{z}_i^{(N)} = \dot{y}_i^{(N)} f'_N(z_i^{(N)})$

Called “**Backpropagation**” because the derivative of the loss is propagated “backwards” through the network

- For layer $k = N - 1$ down to 1

Very analogous to the forward pass:

- For $i = 1 \dots D_k$

- $\dot{y}_i^{(k)} = \sum_j w_{ij}^{(k+1)} \dot{z}_j^{(k+1)}$

Backward weighted combination of next layer

- $\dot{z}_i^{(k)} = \dot{y}_i^{(k)} f'_k(z_i^{(k)})$

Backward equivalent of activation

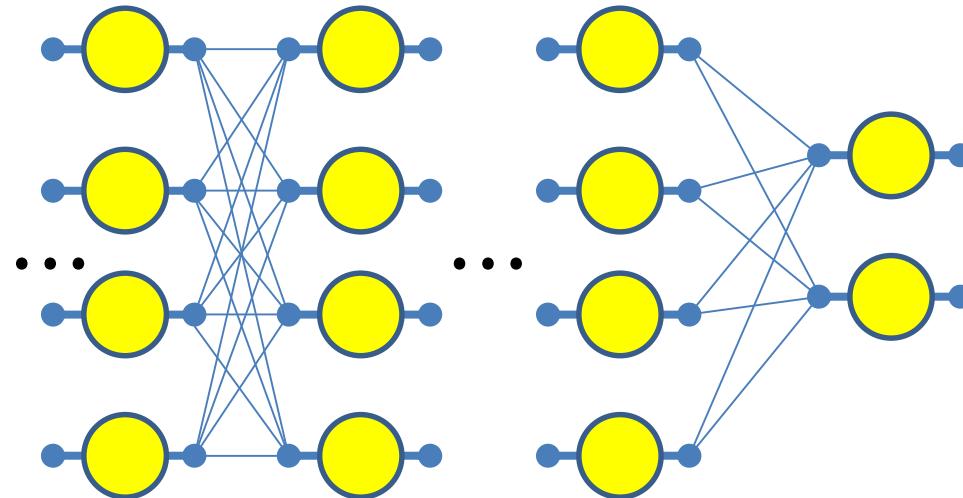
- $\frac{\partial \text{Div}}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \dot{z}_i^{(k+1)}$ for $j = 1 \dots D_k$

- $\frac{\partial \text{Div}}{\partial w_{ji}^{(1)}} = y_j^{(0)} \dot{z}_i^{(1)}$ for $j = 1 \dots D_0$

For comparison: the forward pass again

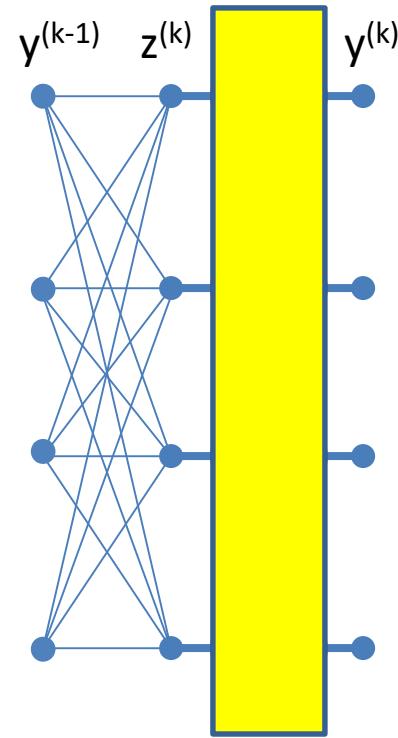
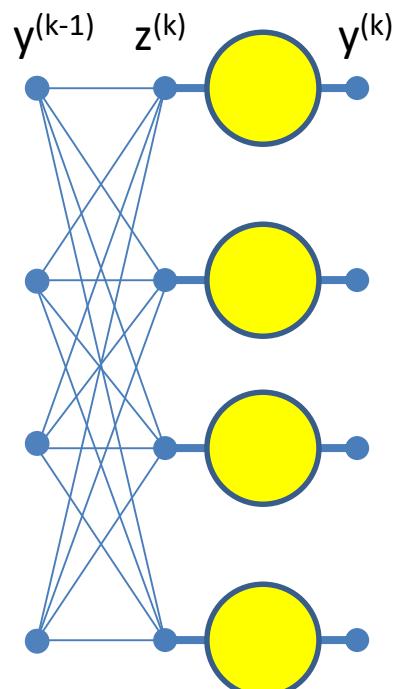
- Input: D dimensional vector $\mathbf{x} = [x_j, j = 1 \dots D]$
- Set:
 - $D_0 = D$, is the width of the 0th (input) layer
 - $y_j^{(0)} = x_j, j = 1 \dots D; y_0^{(k=1\dots N)} = x_0 = 1$
- For layer $k = 1 \dots N$
 - For $j = 1 \dots D_k$
 - $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)} y_i^{(k-1)}$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

Special cases



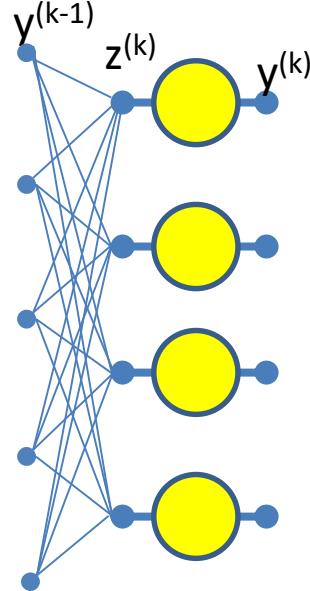
- Have assumed so far that
 1. The computation of the output of one neuron does not directly affect computation of other neurons in the same (or previous) layers
 2. Inputs to neurons only combine through weighted addition
 3. Activations are actually differentiable
 - All of these conditions are frequently not applicable
- Will not dwell on the topic in class, but explained in slides
 - Will appear in quiz. Please read the slides

Special Case 1. Vector activations



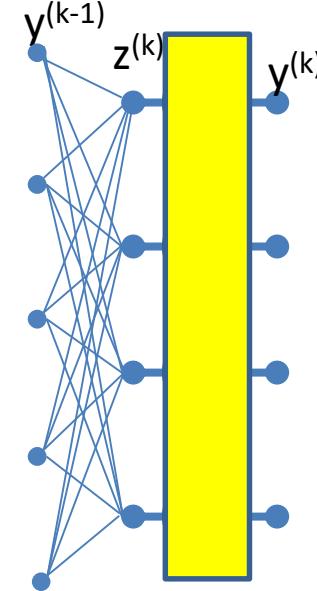
- Vector activations: all outputs are functions of all inputs

Special Case 1. Vector activations



Scalar activation: Modifying a z_i only changes corresponding y_i

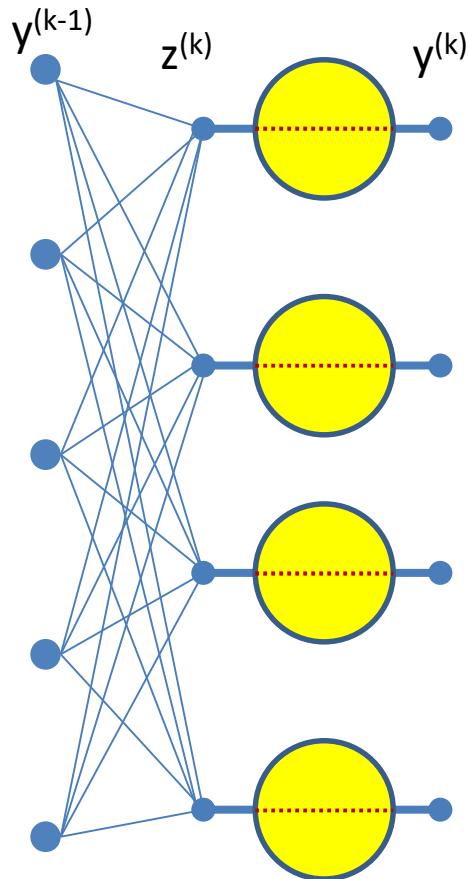
$$y_i^{(k)} = f(z_i^{(k)})$$



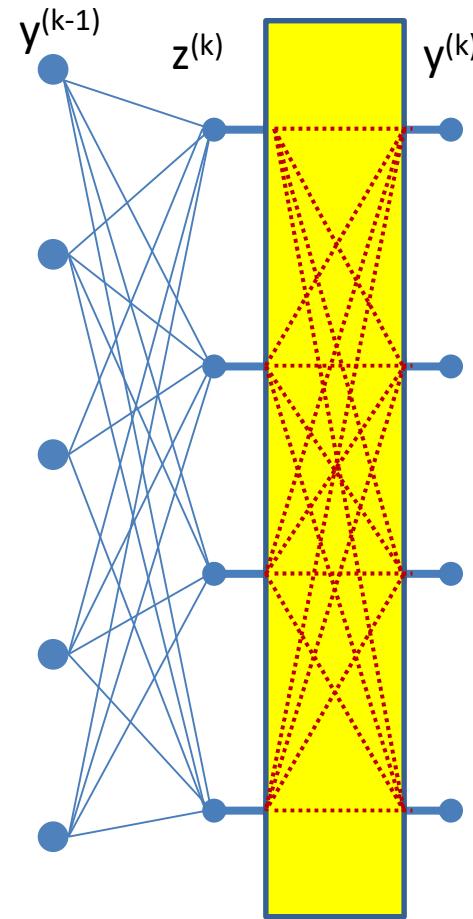
Vector activation: Modifying a z_i potentially changes all, $y_1 \dots y_M$

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

“Influence” diagram

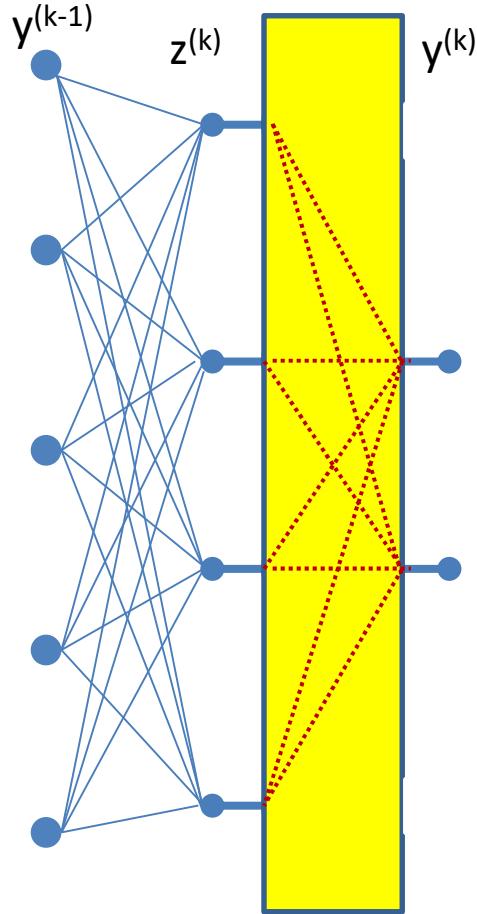
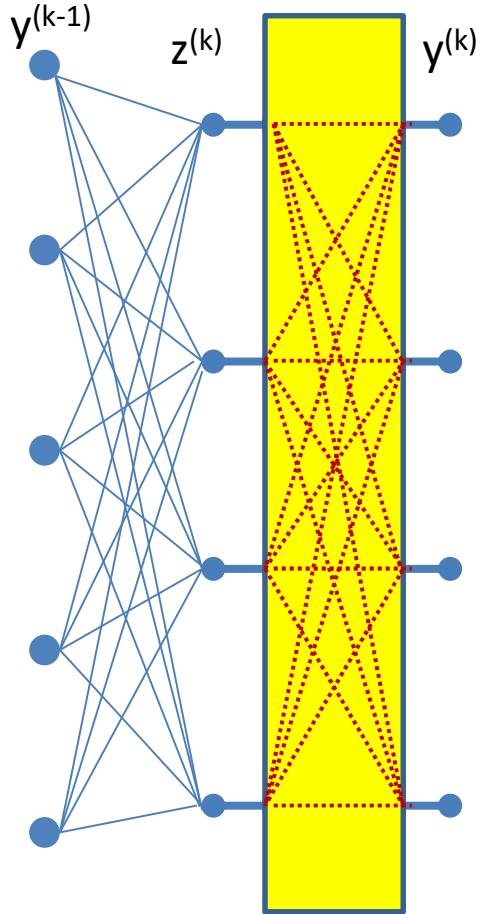


Scalar activation: Each z_i influences *one* y_i



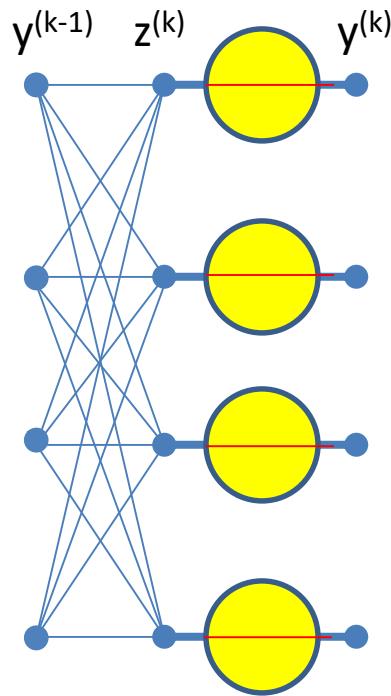
Vector activation: Each z_i influences all, $y_1 \dots y_M$

The number of outputs



- Note: The number of outputs ($y^{(k)}$) need not be the same as the number of inputs ($z^{(k)}$)
 - May be more or fewer

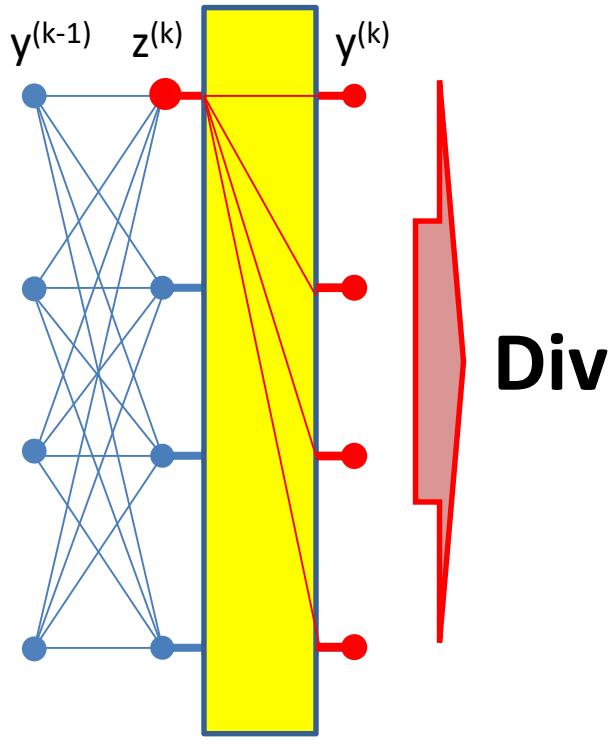
Scalar Activation: Derivative rule



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

- In the case of *scalar* activation functions, the derivative of the error w.r.t to the input to the unit is a simple product of derivatives

Derivatives of vector activation



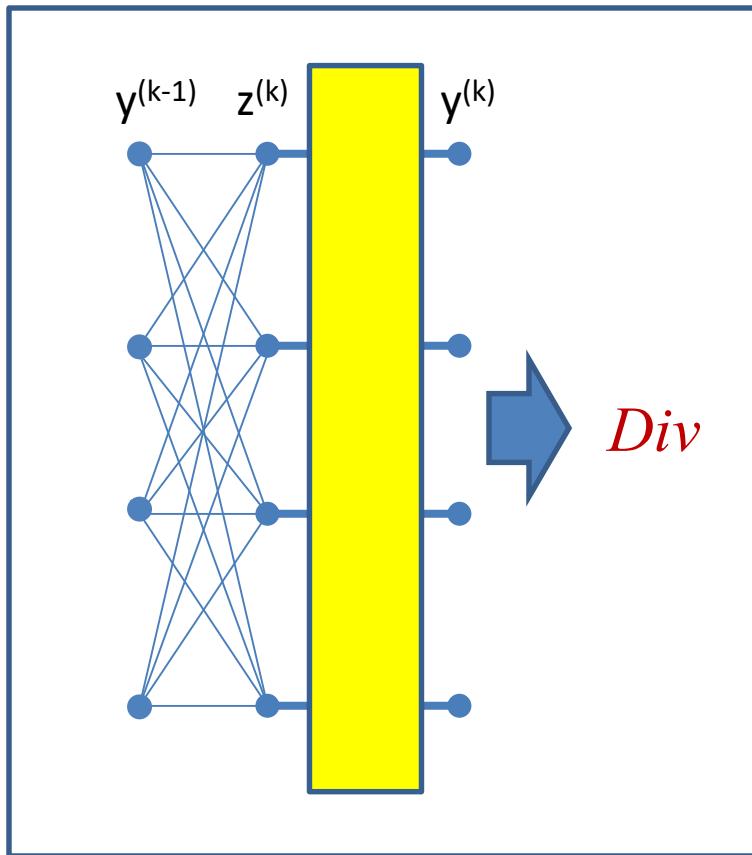
$$\frac{\partial \text{Div}}{\partial z_i^{(k)}} = \sum_j \frac{\partial \text{Div}}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}}$$

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

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

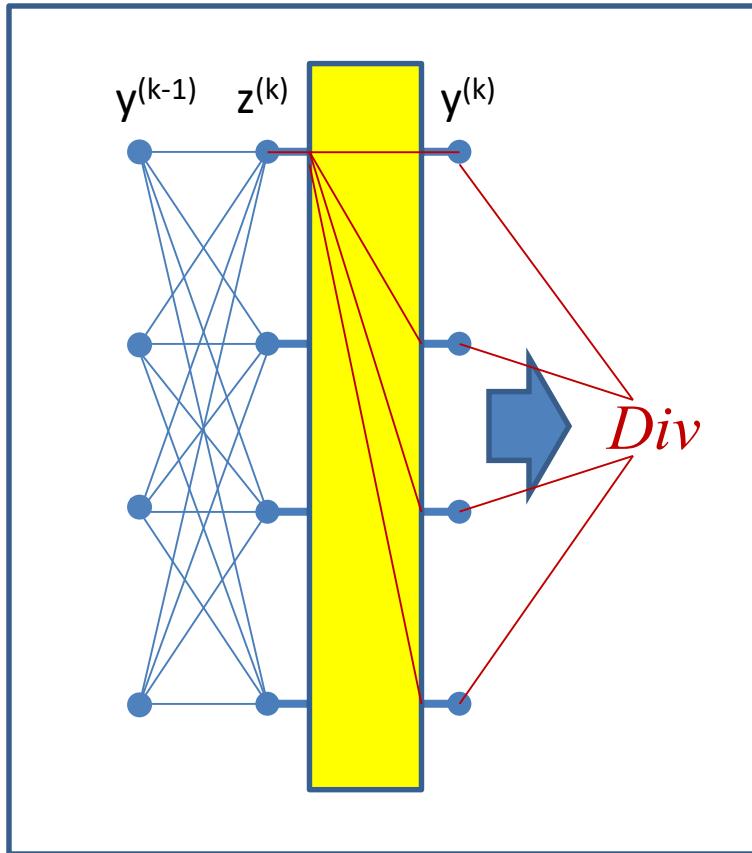
- For *vector* activations the derivative of the error w.r.t. to any input is a sum of partial derivatives
 - Regardless of the number of outputs $y_j^{(k)}$

Example Vector Activation: Softmax



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

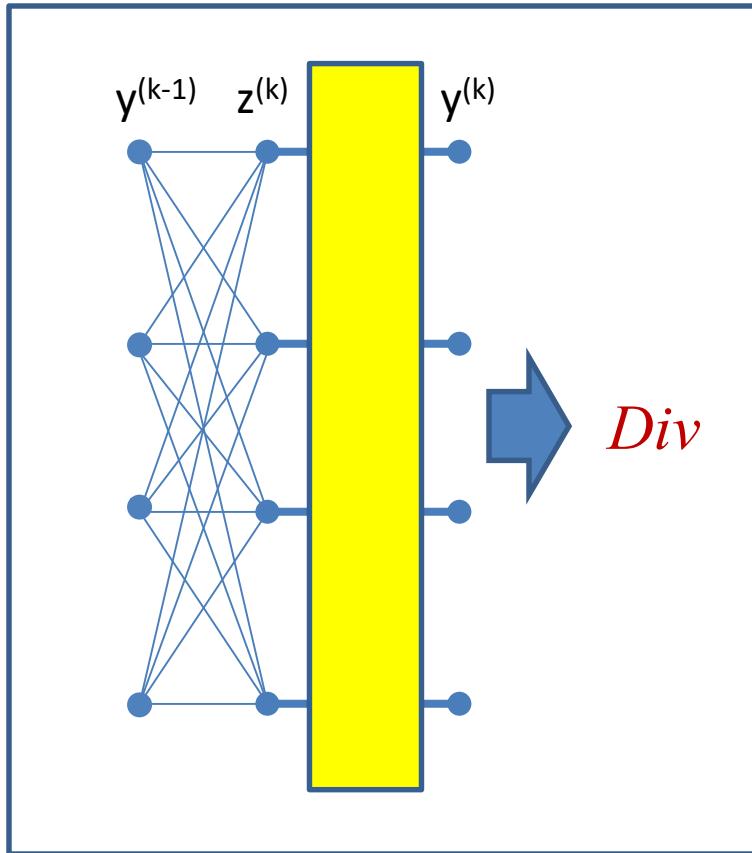
Example Vector Activation: Softmax



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

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

Example Vector Activation: Softmax

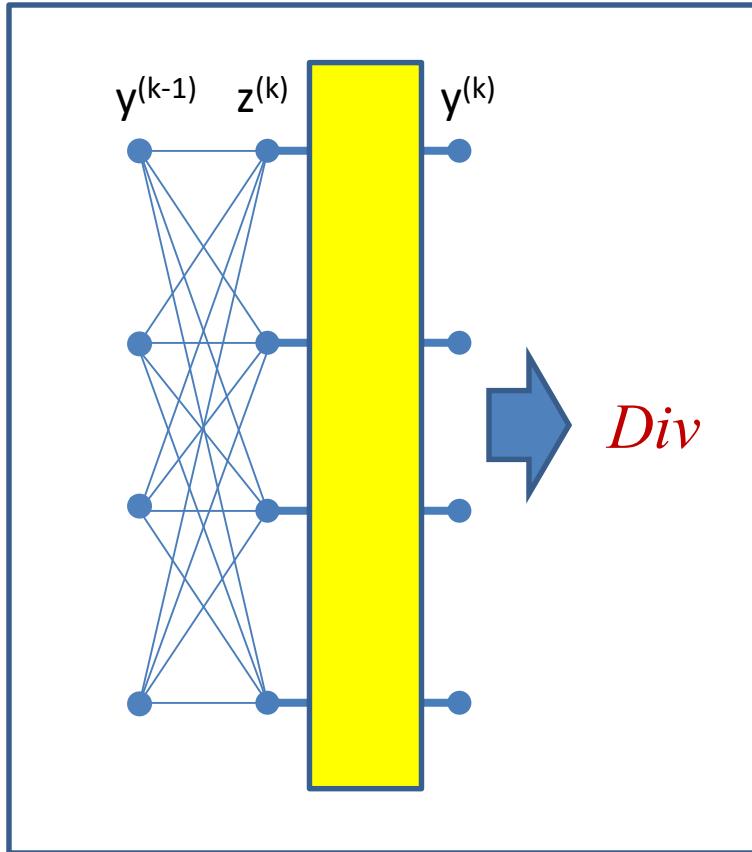


$$y_i^{(k)} = \frac{\exp(z_i^{(k)})}{\sum_j \exp(z_j^{(k)})}$$

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

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

Example Vector Activation: Softmax



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

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

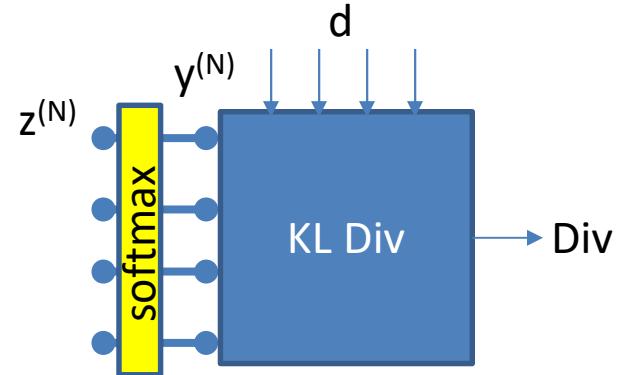
$$\frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} = \begin{cases} y_i^{(k)} (1 - y_i^{(k)}) & \text{if } i = j \\ -y_i^{(k)} y_j^{(k)} & \text{if } i \neq j \end{cases}$$

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

- For future reference
- δ_{ij} is the Kronecker delta: $\delta_{ij} = 1$ if $i = j$, 0 if $i \neq j$

Backward Pass for softmax output layer

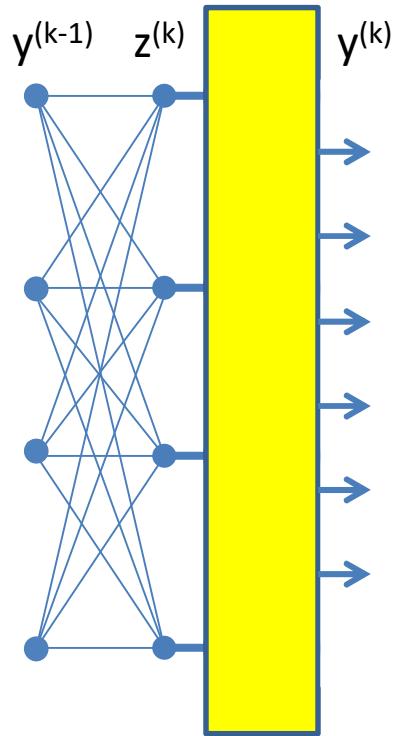
- Output layer (N) :
 - For $i = 1 \dots D_N$
 - $\frac{\partial \text{Div}}{\partial y_i^{(N)}} = \frac{\partial \text{Div}(Y, d)}{\partial y_i}$
 - $\frac{\partial \text{Div}}{\partial z_i^{(N)}} = \sum_j \frac{\partial \text{Div}(Y, d)}{\partial y_j^{(N)}} y_i^{(N)} (\delta_{ij} - y_j^{(N)})$
- For layer $k = N - 1$ down to 1
 - For $i = 1 \dots D_k$
 - $\frac{\partial \text{Div}}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial \text{Div}}{\partial z_j^{(k+1)}}$
 - $\frac{\partial \text{Div}}{\partial z_i^{(k)}} = f'_k(z_i^{(k)}) \frac{\partial \text{Div}}{\partial y_i^{(k)}}$
 - $\frac{\partial \text{Div}}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \frac{\partial \text{Div}}{\partial z_i^{(k+1)}} \text{ for } j = 1 \dots D_k$
 - $\frac{\partial \text{Div}}{\partial w_{ji}^{(1)}} = y_j^{(0)} \frac{\partial \text{Div}}{\partial z_i^{(1)}} \text{ for } j = 1 \dots D_0$



Special cases

- Examples of vector activations and other special cases on slides
 - Please look up
 - Will appear in quiz!

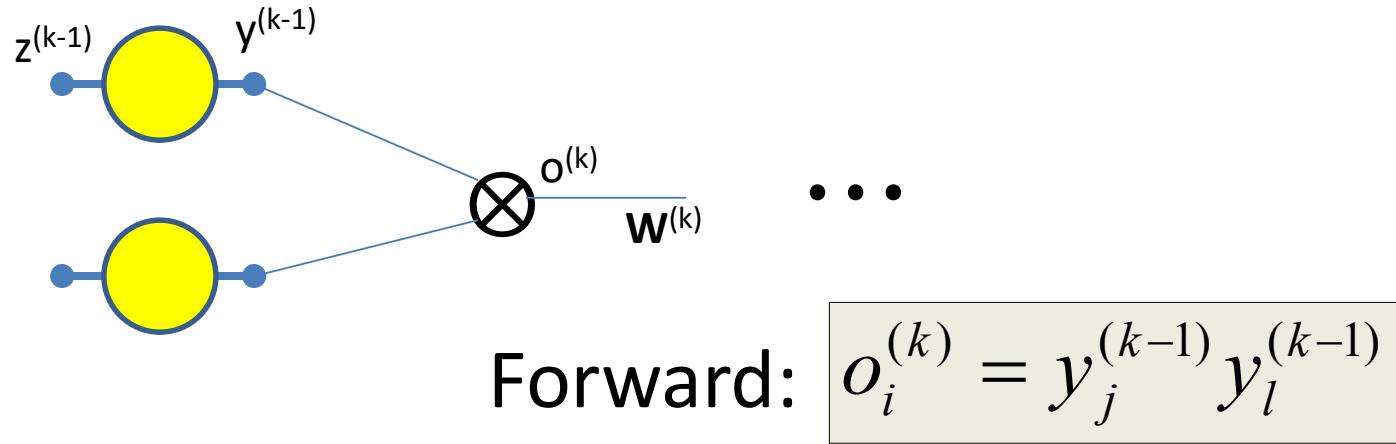
Vector Activations



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

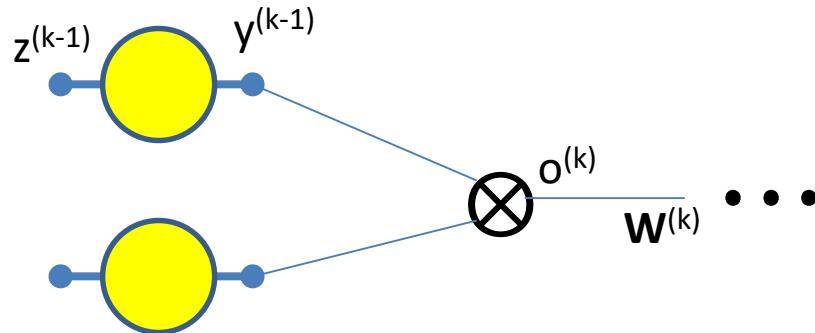
- In reality the vector combinations can be anything
 - E.g. linear combinations, polynomials, logistic (softmax), etc.

Special Case 2: Multiplicative networks



- 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.

Backpropagation: Multiplicative Networks



Forward:

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

Backward:

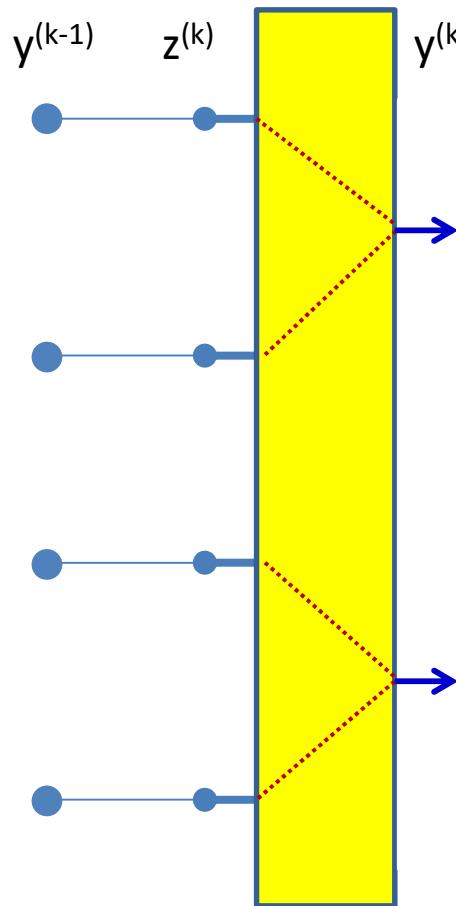
$$\frac{\partial \text{Div}}{\partial o_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial \text{Div}}{\partial z_j^{(k+1)}}$$

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

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

- Some types of networks have *multiplicative* combination

Multiplicative combination as a case of vector activations

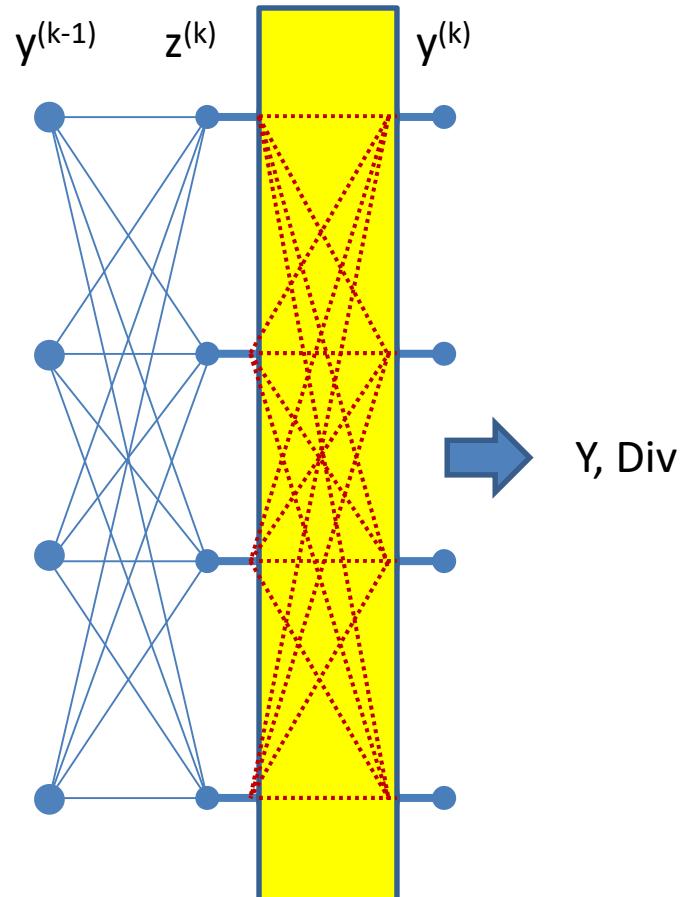


$$z_i^{(k)} = y_i^{(k-1)}$$

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

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

Multiplicative combination: Can be viewed as a case of vector activations



$$z_i^{(k)} = \sum_j w_{ji}^{(k)} y_j^{(k-1)}$$

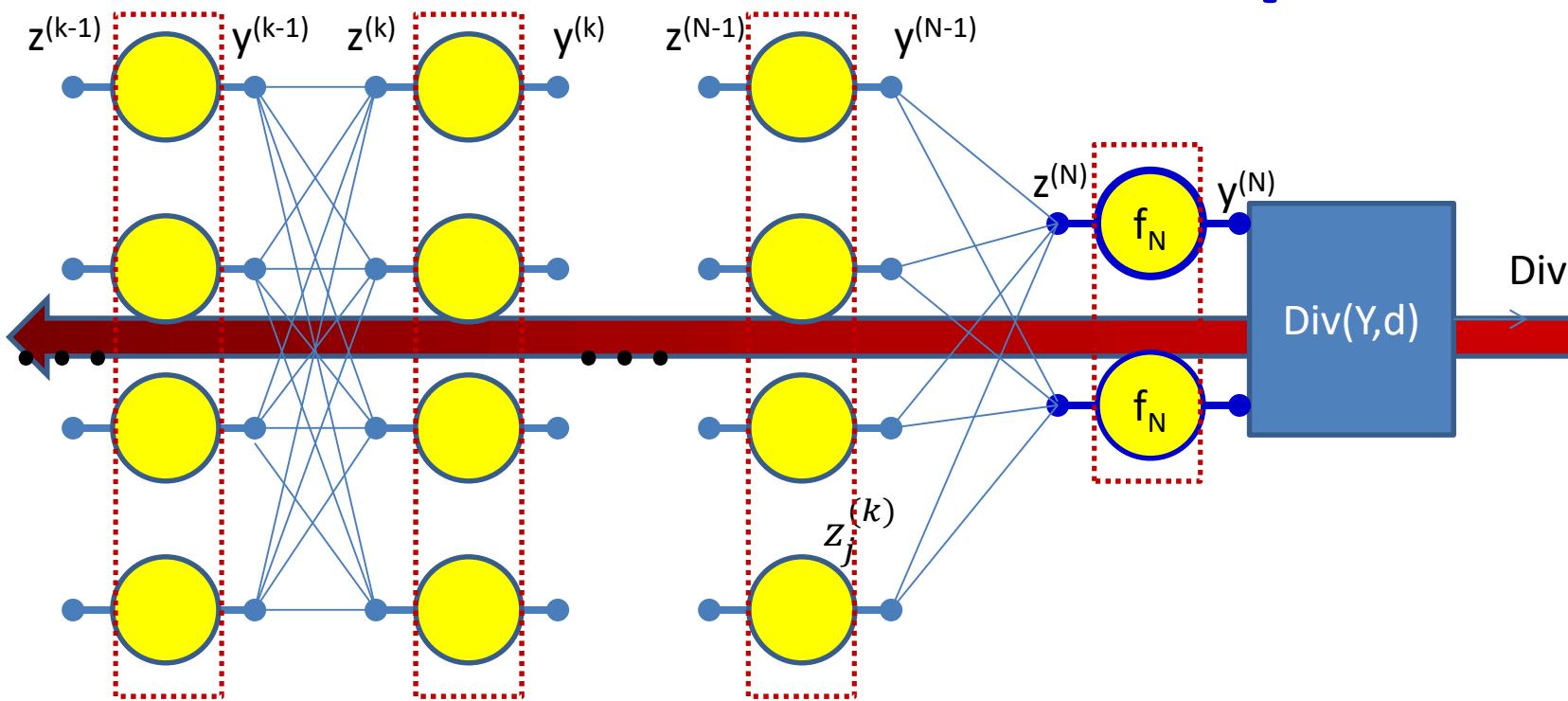
$$y_i^{(k)} = \prod_l (z_l^{(k)})^{\alpha_{li}^{(k)}}$$

$$\frac{\partial y_i^{(k)}}{\partial z_j^{(k)}} = \alpha_{ji}^{(k)} (z_j^{(k)})^{\alpha_{ji}^{(k)} - 1} \prod_{l \neq j} (z_l^{(k)})^{\alpha_{li}^{(k)}}$$

$$\frac{\partial Div}{\partial z_j^{(k)}} = \sum_i \frac{\partial Div}{\partial y_i^{(k)}} \frac{\partial y_i^{(k)}}{\partial z_j^{(k)}}$$

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

Gradients: Backward Computation



For $k = N \dots 1$

For $i = 1 : \text{layer width}$

If layer has vector activation

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

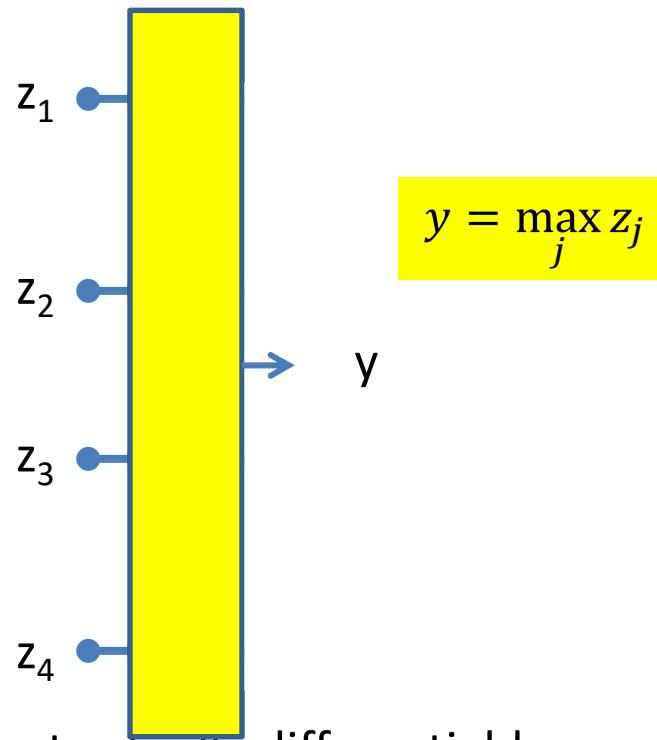
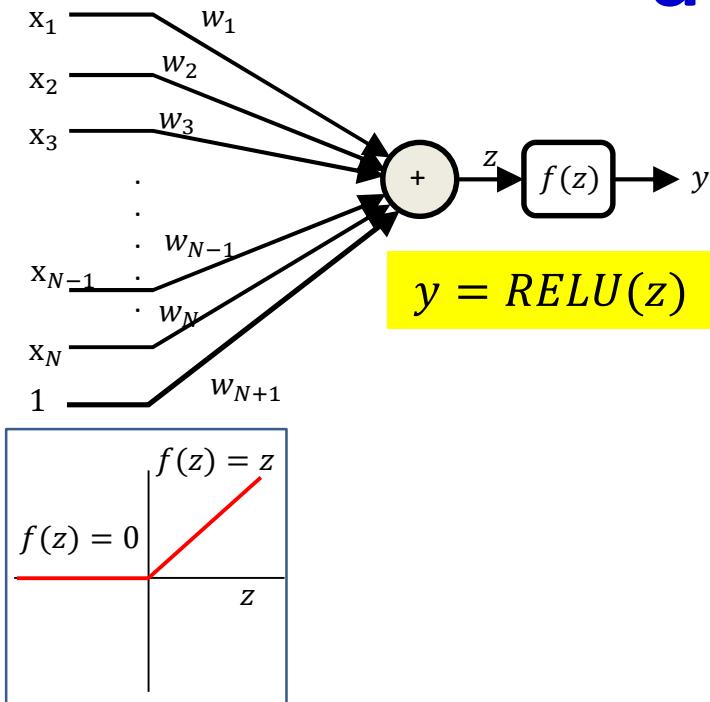
Else if activation is scalar

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

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

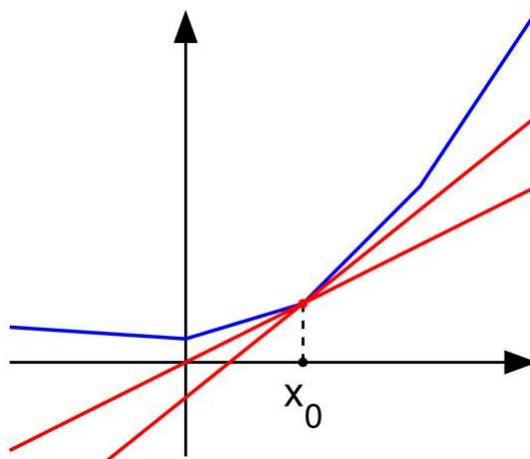
$$\frac{\partial \text{Div}}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial \text{Div}}{\partial z_j^{(k)}}$$

Special Case : Non-differentiable activations



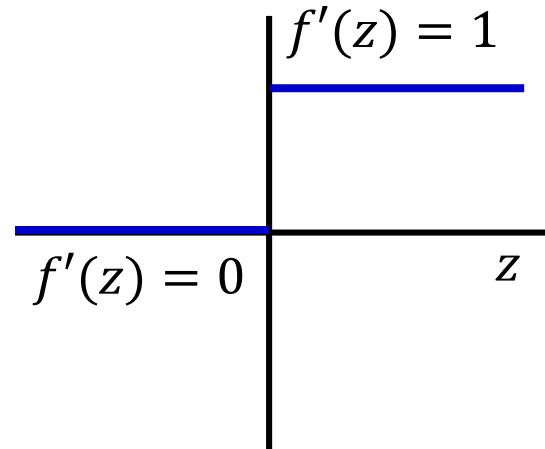
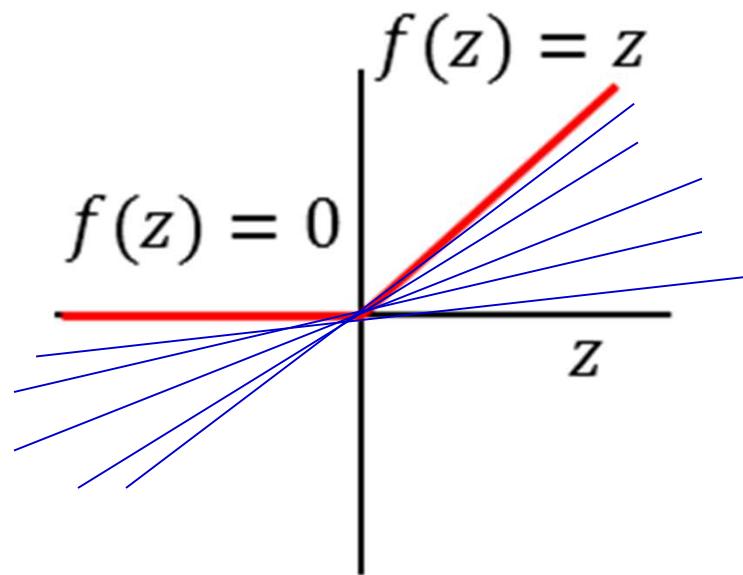
- Activation functions are sometimes not actually differentiable
 - E.g. The RELU (Rectified Linear Unit)
 - And its variants: leaky RELU, randomized leaky RELU
 - E.g. The “max” function
- Must use “subgradients” where available
 - Or “secants”

The subgradient



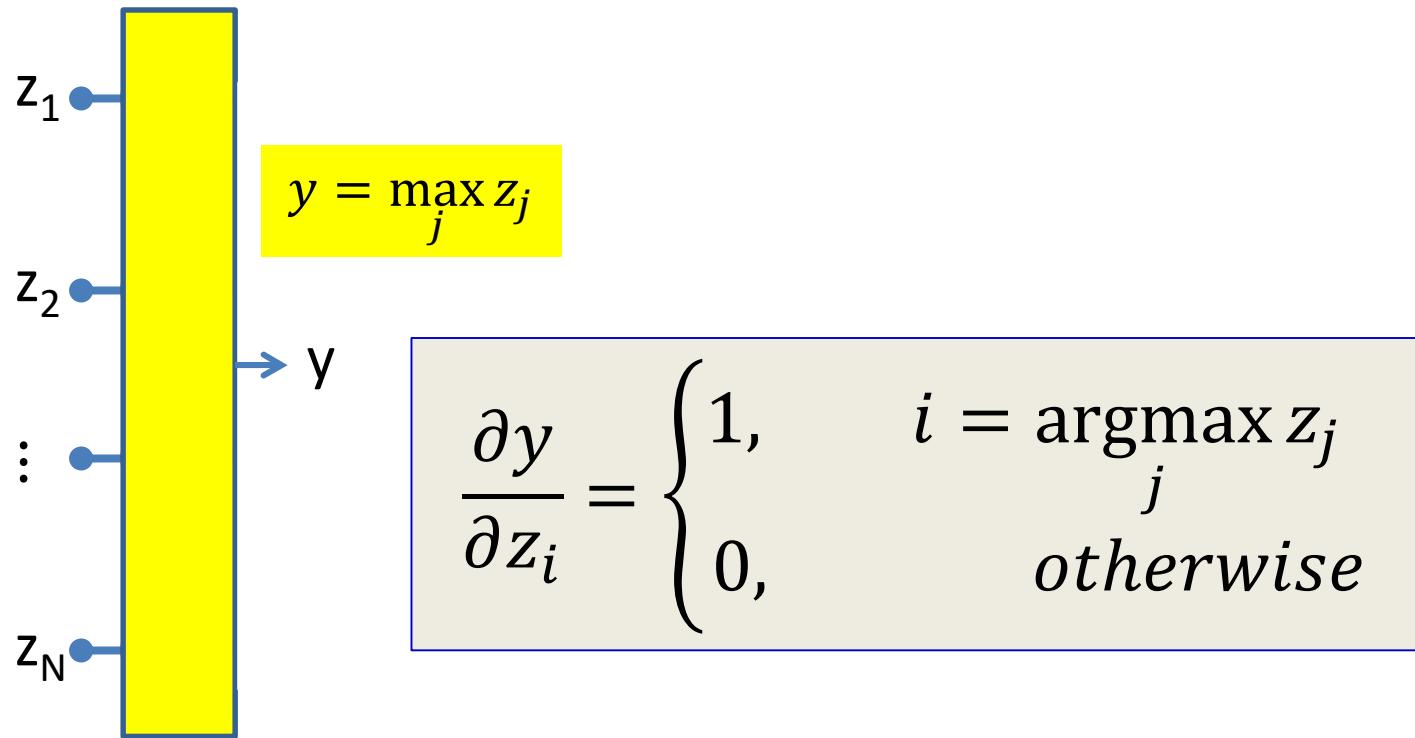
- A subgradient of a function $f(x)$ at a point x_0 is any vector v such that
$$(f(x) - f(x_0)) \geq v^T(x - x_0)$$
 - Any direction such that moving in that direction increases the function
- Guaranteed to exist only for convex functions
 - “bowl” shaped functions
 - For non-convex functions, the equivalent 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



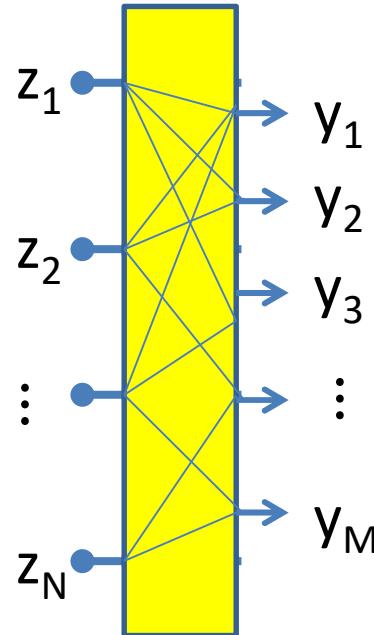
- Can use any subgradient
 - At the differentiable points on the curve, this is the same as the gradient
 - Typically, will use the equation given

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 $i = 1 \dots D_N$
 - $\frac{\partial \text{Div}}{\partial y_i^{(N)}} = \frac{\partial \text{Div}(Y, d)}{\partial y_i}$
 - $\frac{\partial \text{Div}}{\partial z_i^{(N)}} = \frac{\partial \text{Div}}{\partial y_i^{(N)}} \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \quad OR \quad \sum_j \frac{\partial \text{Div}}{\partial y_j^{(N)}} \frac{\partial y_j^{(N)}}{\partial z_i^{(N)}} \quad (\text{vector activation})$
 - For layer $k = N - 1$ down to 1
 - For $i = 1 \dots D_k$
 - $\frac{\partial \text{Div}}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial \text{Div}}{\partial z_j^{(k+1)}}$
 - $\frac{\partial \text{Div}}{\partial z_i^{(k)}} = \frac{\partial \text{Div}}{\partial y_i^{(k)}} \frac{\partial y_i^{(k)}}{\partial z_i^{(k)}} \quad OR \quad \sum_j \frac{\partial \text{Div}}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} \quad (\text{vector activation})$
 - $\frac{\partial \text{Div}}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \frac{\partial \text{Div}}{\partial z_i^{(k+1)}} \quad \text{for } j = 1 \dots D_k$
 - $\frac{\partial \text{Div}}{\partial w_{ji}^{(1)}} = y_j^{(0)} \frac{\partial \text{Div}}{\partial z_i^{(1)}} \quad \text{for } j = 1 \dots D_0$
- 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

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

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

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

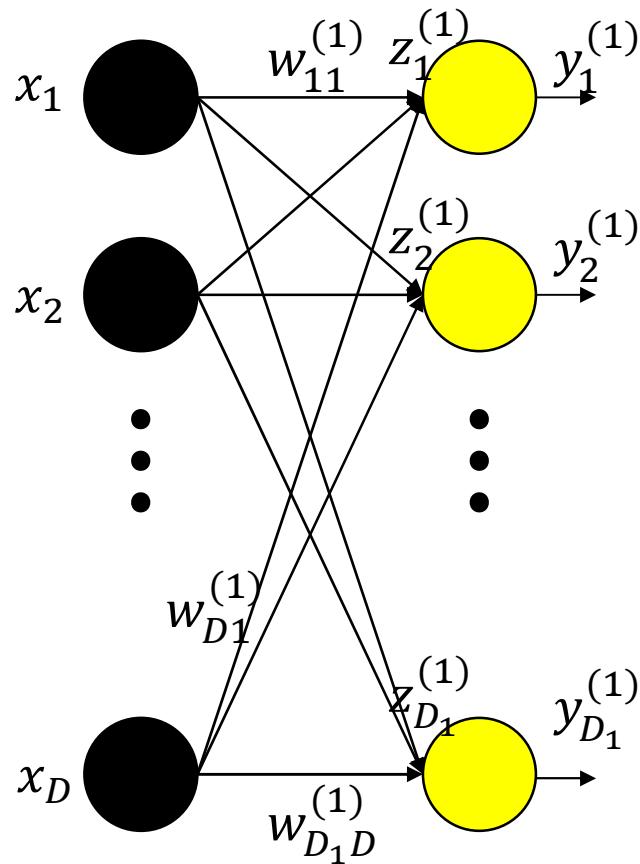
Training by BackProp

- Initialize weights $\mathbf{W}^{(k)}$ for all layers $k = 1 \dots K$
- Do: (*Gradient descent iterations*)
 - Initialize $\text{Loss} = 0$; For all i, j, k , initialize $\frac{d\text{Loss}}{d\mathbf{w}_{i,j}^{(k)}} = 0$
 - For all $t = 1:T$ (*Iterate over training instances*)
 - **Forward pass:** Compute
 - Output \mathbf{Y}_t
 - $\text{Loss} += \text{Div}(\mathbf{Y}_t, \mathbf{d}_t)$
 - **Backward pass:** For all i, j, k :
 - Compute $\frac{d\text{Div}(\mathbf{Y}_t, \mathbf{d}_t)}{d\mathbf{w}_{i,j}^{(k)}}$
 - Compute $\frac{d\text{Loss}}{d\mathbf{w}_{i,j}^{(k)}} += \frac{d\text{Div}(\mathbf{Y}_t, \mathbf{d}_t)}{d\mathbf{w}_{i,j}^{(k)}}$
 - For all i, j, k , update:
$$\mathbf{w}_{i,j}^{(k)} = \mathbf{w}_{i,j}^{(k)} - \frac{\eta}{T} \frac{d\text{Loss}}{d\mathbf{w}_{i,j}^{(k)}}$$
 - 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



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

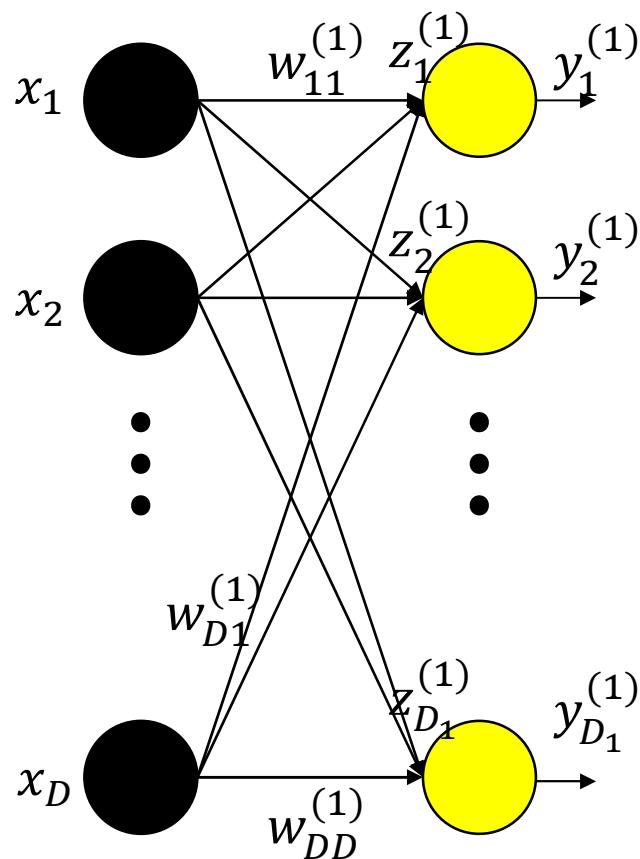
$$\mathbf{z}_k = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{D_k}^{(k)} \end{bmatrix}$$

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

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

- 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



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

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

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

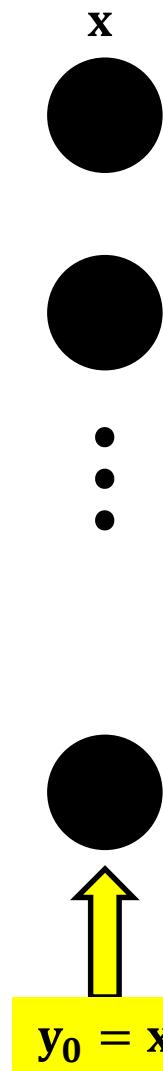
$$\mathbf{W}_k = \begin{bmatrix} w_{11}^{(k)} & w_{21}^{(k)} & \vdots & w_{D_{k-1}1}^{(k)} \\ w_{12}^{(k)} & w_{22}^{(k)} & \vdots & w_{D_{k-1}2}^{(k)} \\ \dots & \dots & \ddots & \vdots \\ w_{1D_k}^{(k)} & w_{2D_k}^{(k)} & \dots & w_{D_{k-1}D_k}^{(k)} \end{bmatrix} \quad \mathbf{b}_k = \begin{bmatrix}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

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

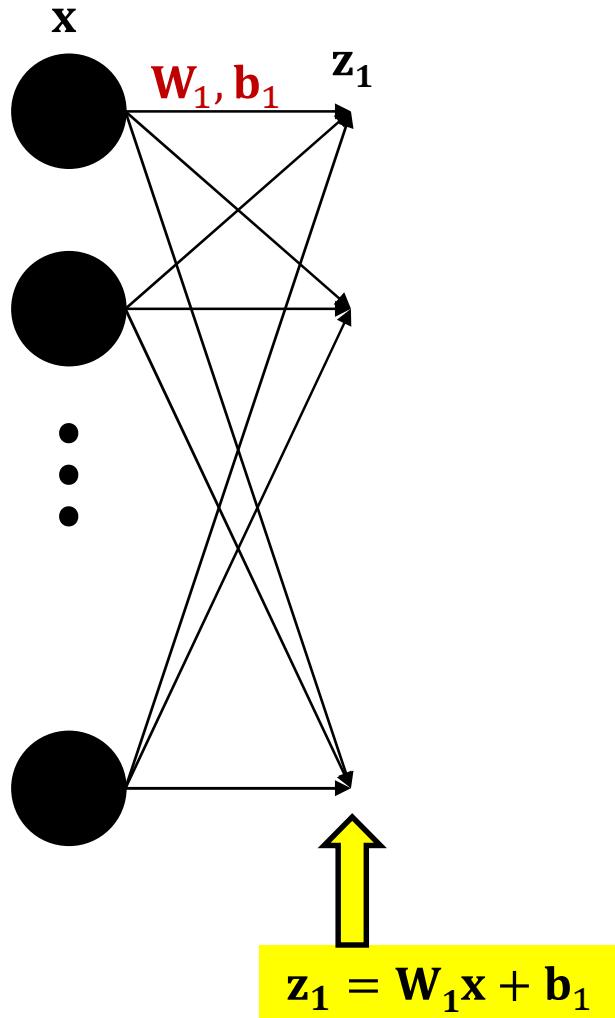
$$\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k$$

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

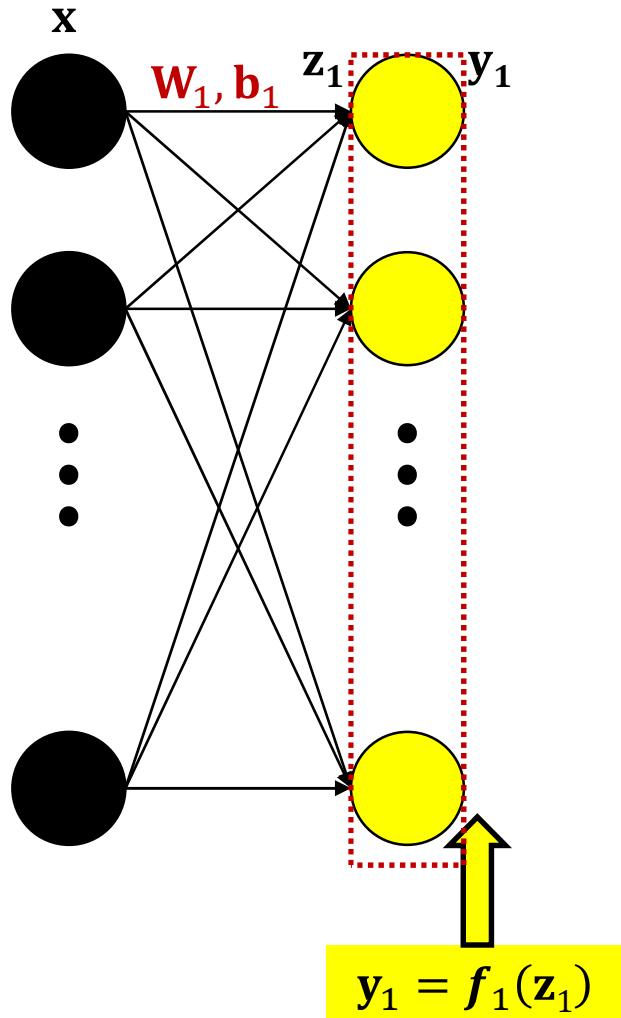
The forward pass: Evaluating the network



The forward pass



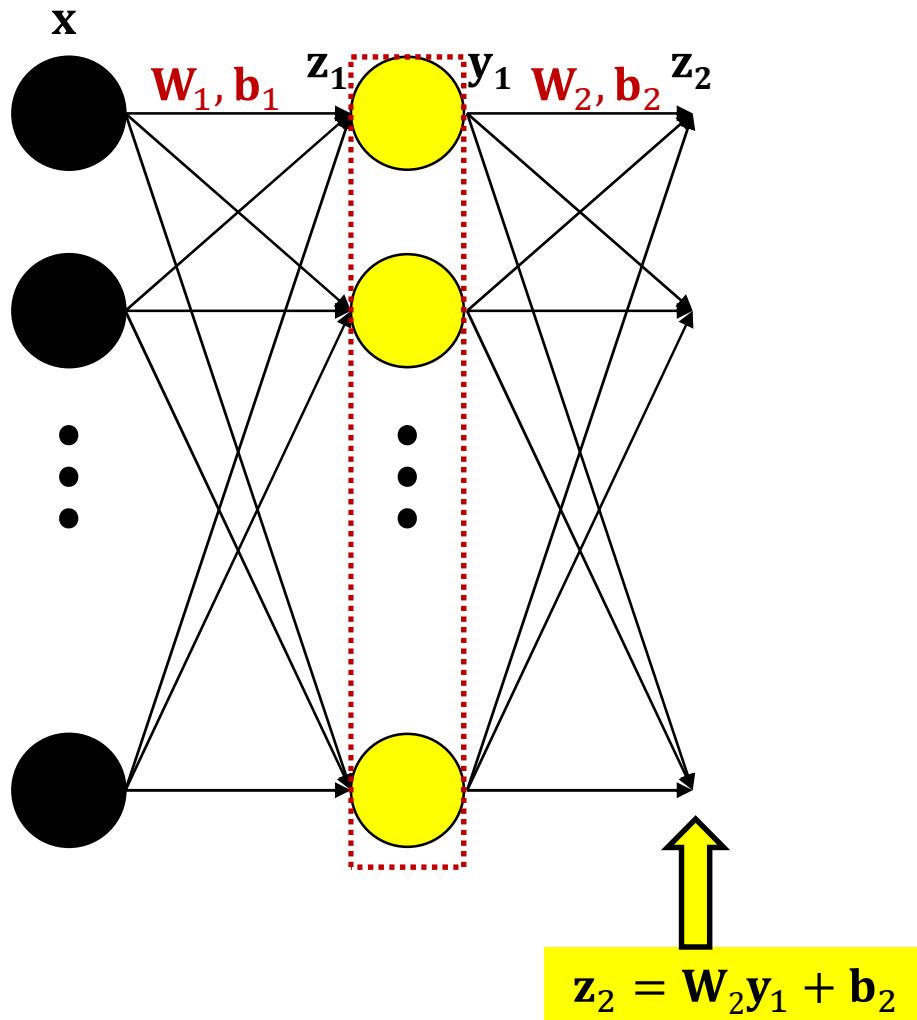
The forward pass



The Complete computation

$$y_1 = f_1(W_1 x + b_1)$$

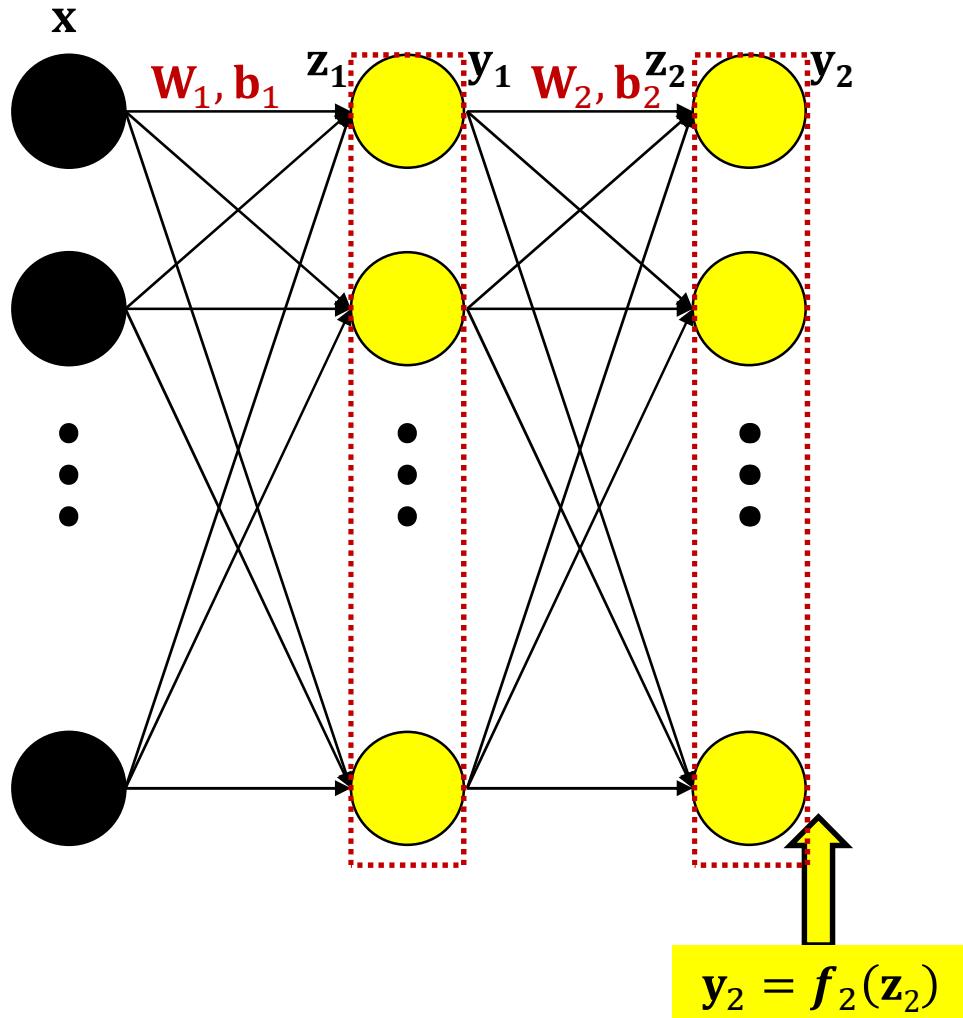
The forward pass



The Complete computation

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

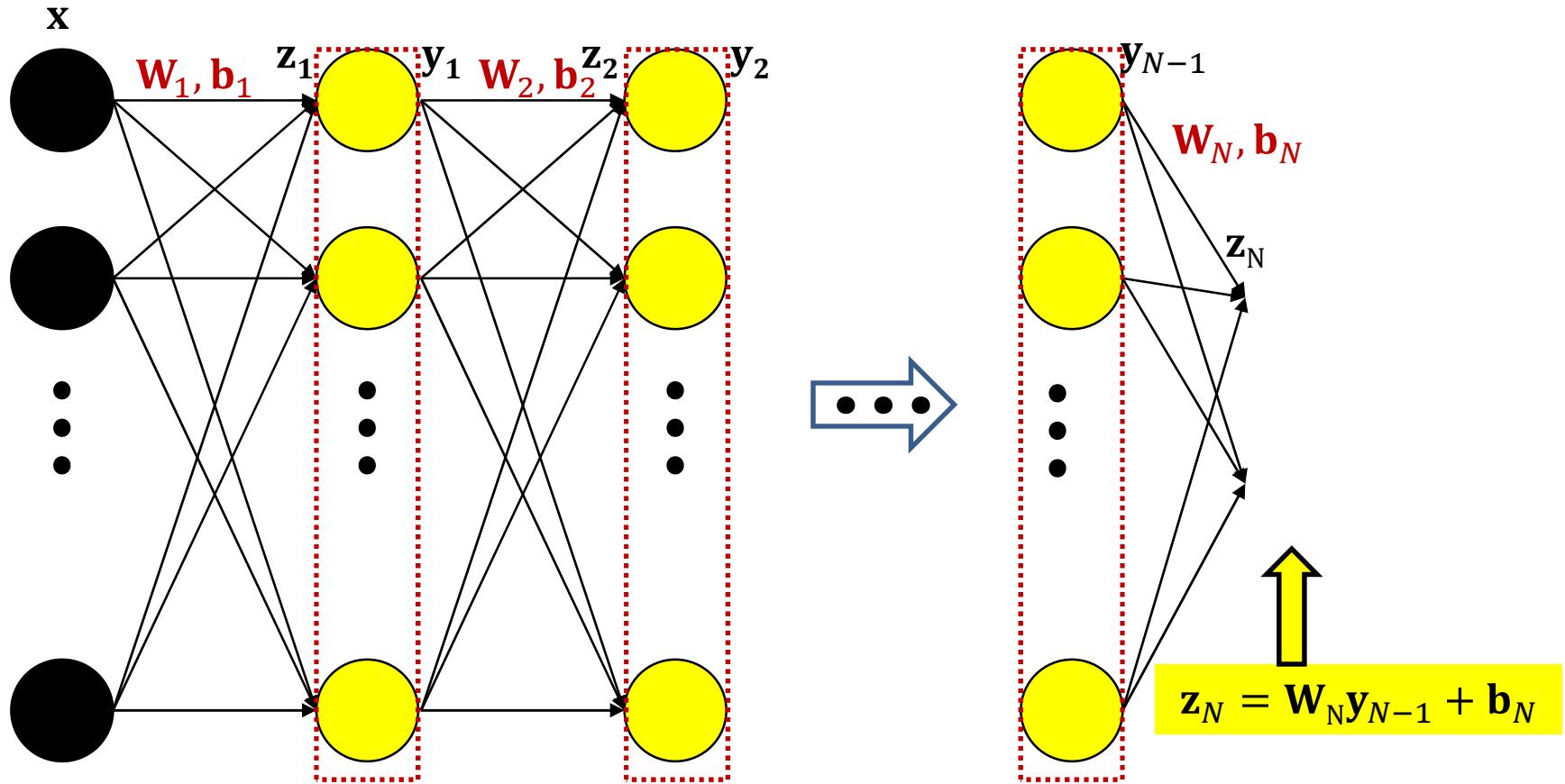
The forward pass



The Complete computation

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

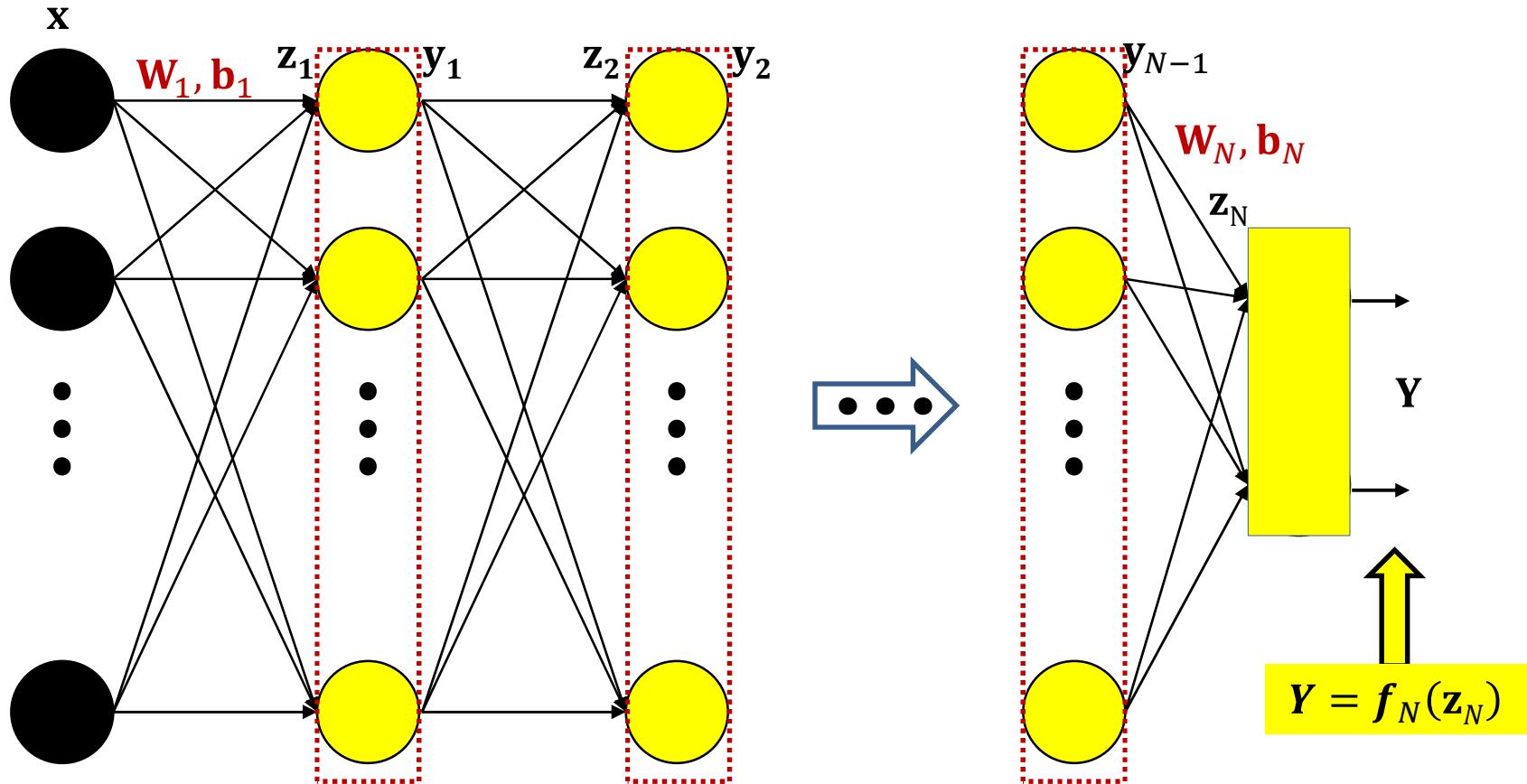
The forward pass



The Complete computation

$$z_N = W_N f_{N-1}(\dots f_2(W_2 f_1(W_1 x + b_1) + b_2) \dots) + b_N$$

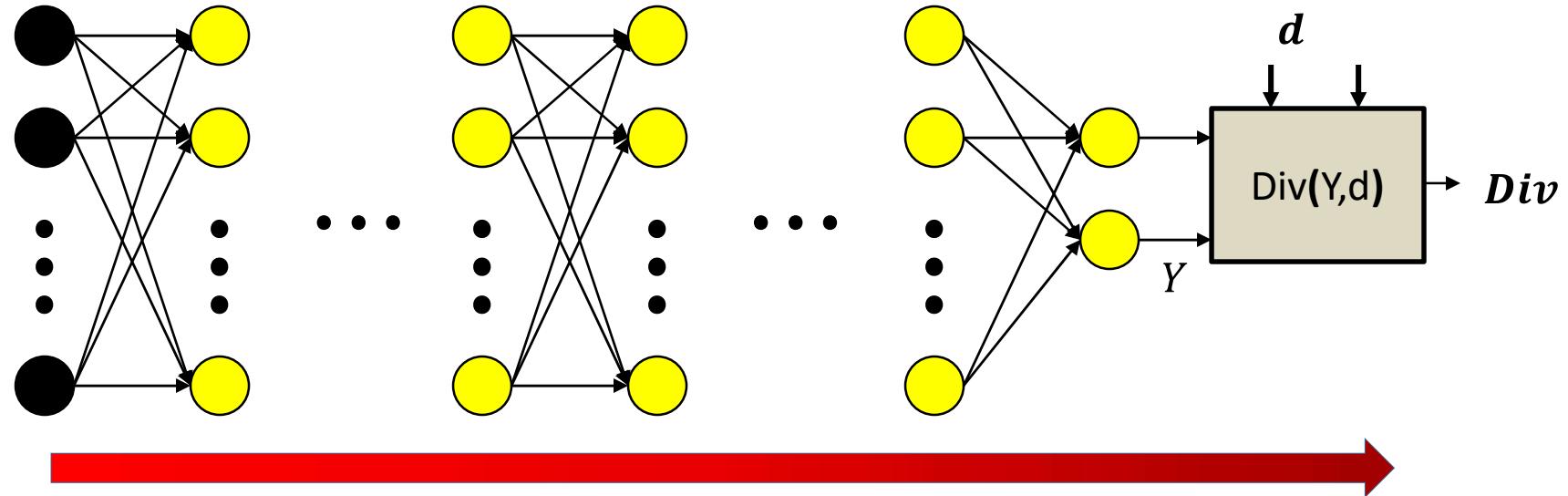
The forward pass



The Complete computation

$$Y = f_N(W_N f_{N-1}(\dots f_2(W_2 f_1(W_1 x + b_1) + b_2) \dots) + b_N)$$

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

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

The Forward Pass

- Set $\mathbf{y}_0 = \mathbf{x}$
- Recursion through layers:

- For layer $k = 1$ to N :

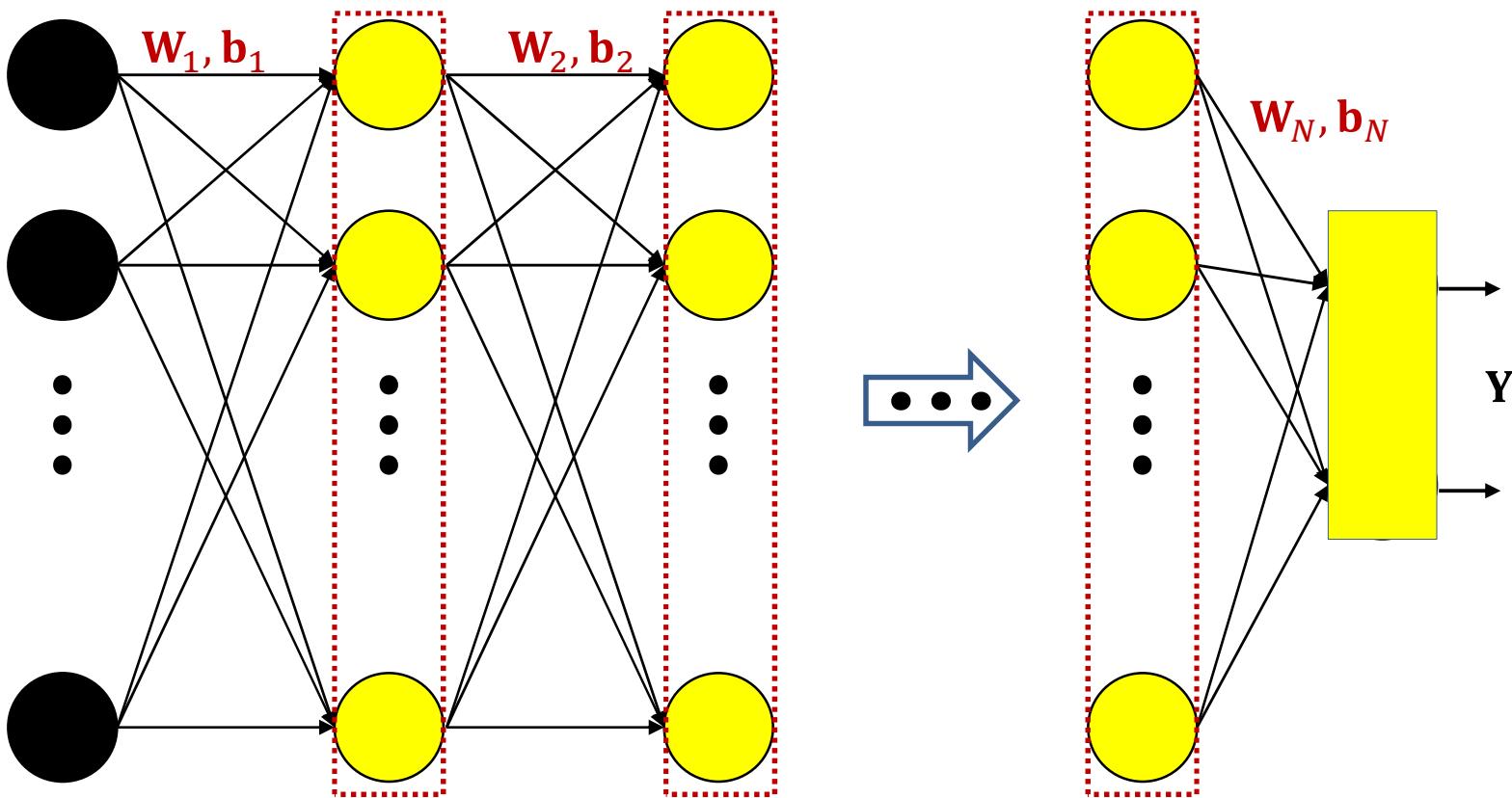
$$\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k$$

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

- Output:

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

The backward pass



- The network is a nested function

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

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

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

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

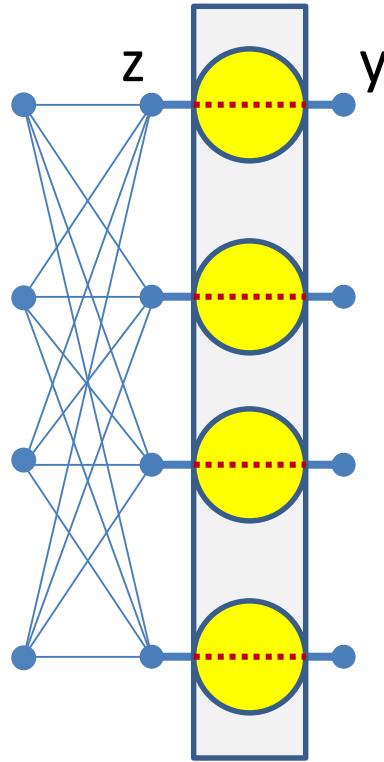
$$\mathbf{y} = f(\mathbf{z})$$

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

Check:

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

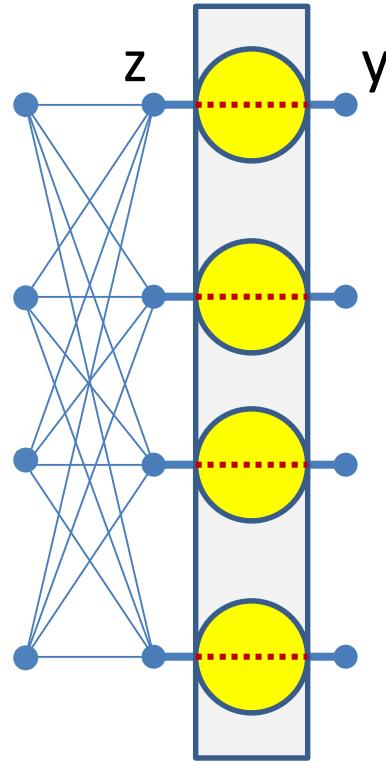
Jacobians can describe the derivatives of neural activations w.r.t their input



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

- **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

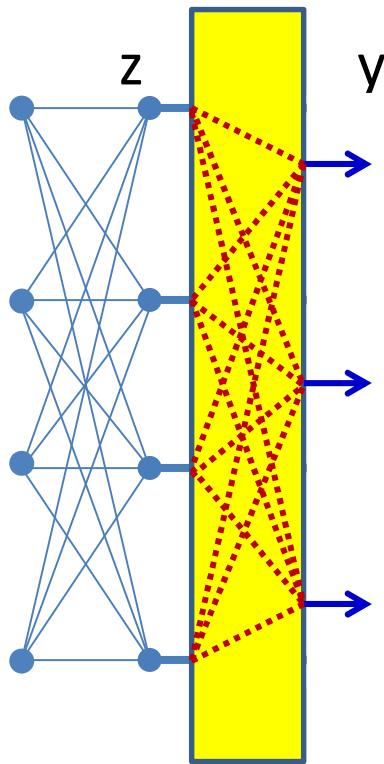


$$y_i = f(z_i)$$

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

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

For *Vector* activations

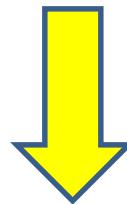


$$J_y(\mathbf{z}) = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \dots & \frac{\partial y_1}{\partial z_D} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \dots & \frac{\partial y_2}{\partial z_D} \\ \dots & \dots & \ddots & \dots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

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

Special case: Affine functions

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

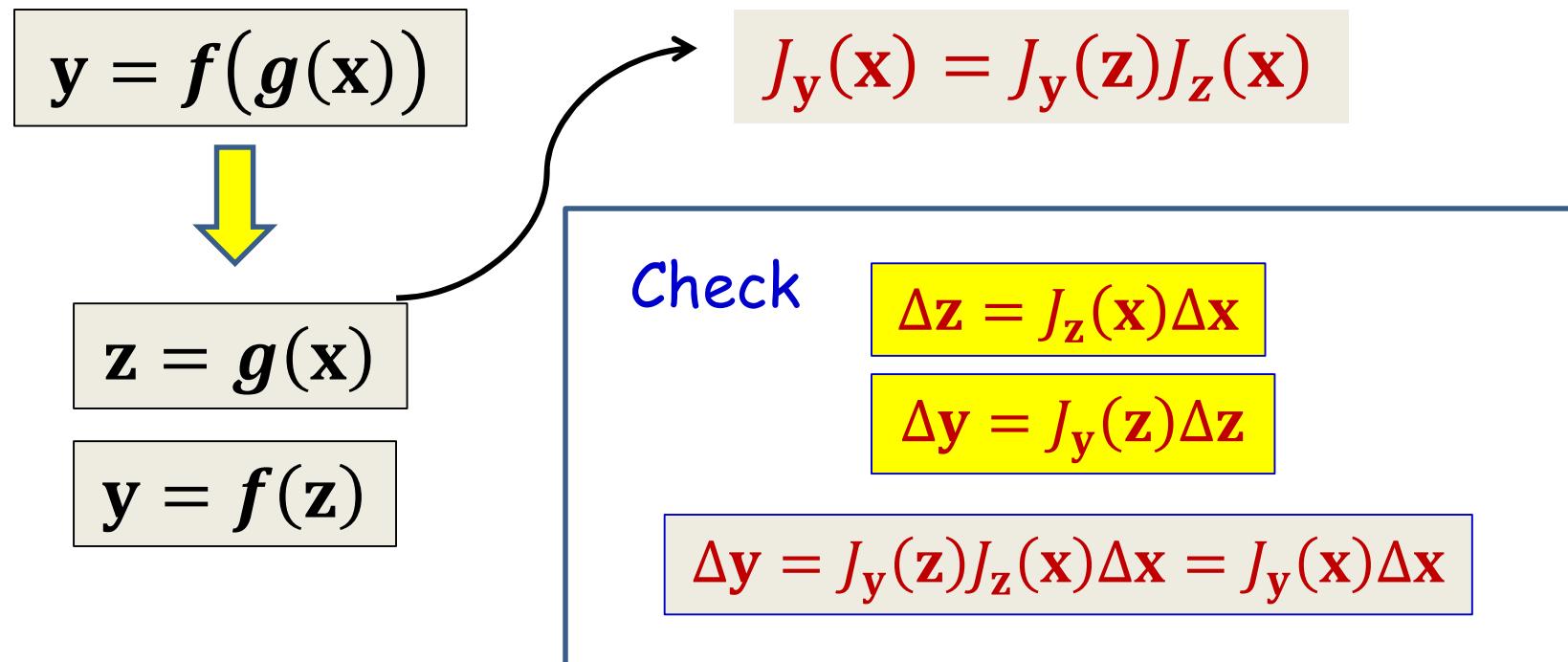


$$J_{\mathbf{z}}(\mathbf{y}) = \mathbf{W}$$

- 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

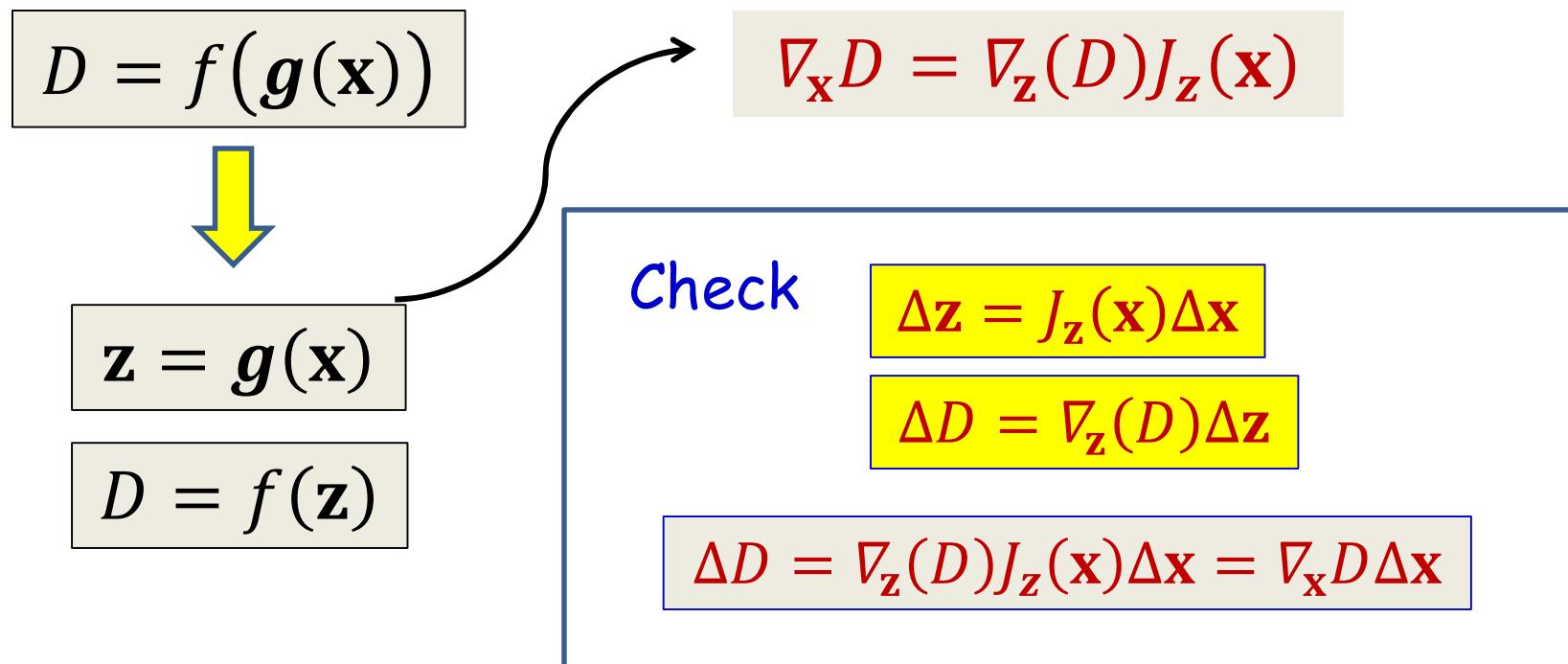
- We can define a chain rule for Jacobians
- **For vector functions of vector inputs:**



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

Vector derivatives: Chain rule

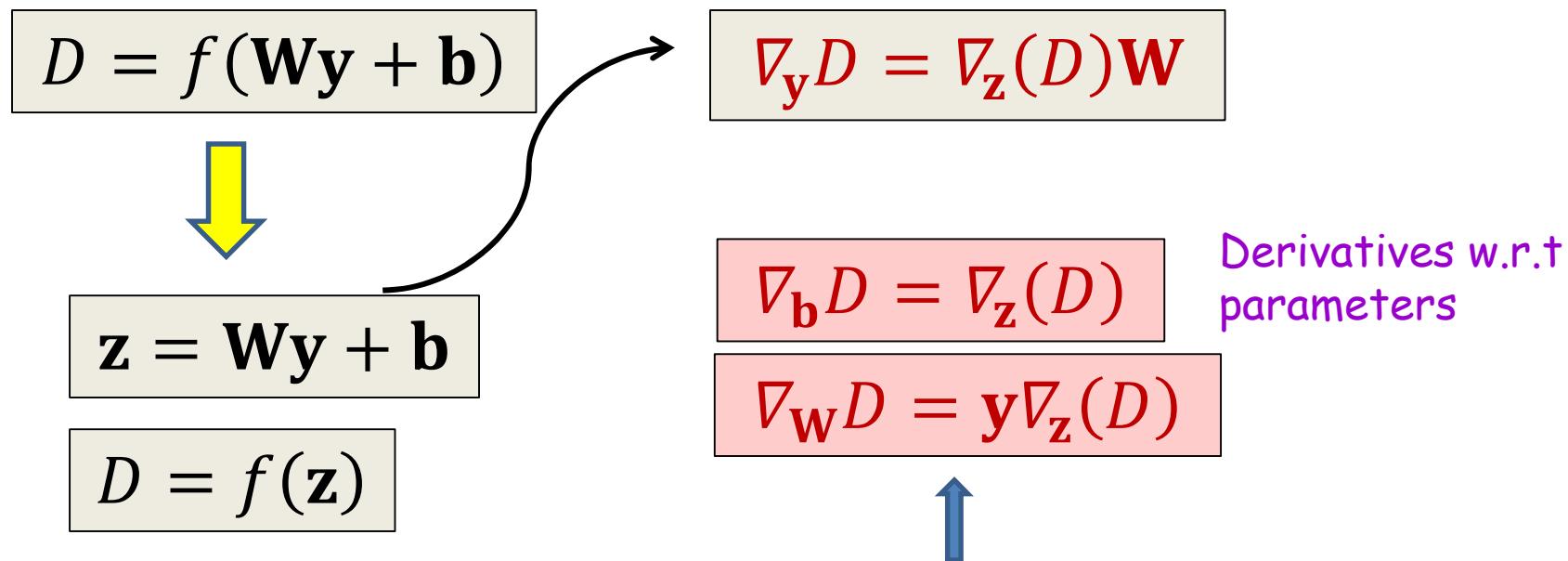
- *The chain rule can combine Jacobians and Gradients*
- **For scalar functions of vector inputs ($g()$ is vector):**



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

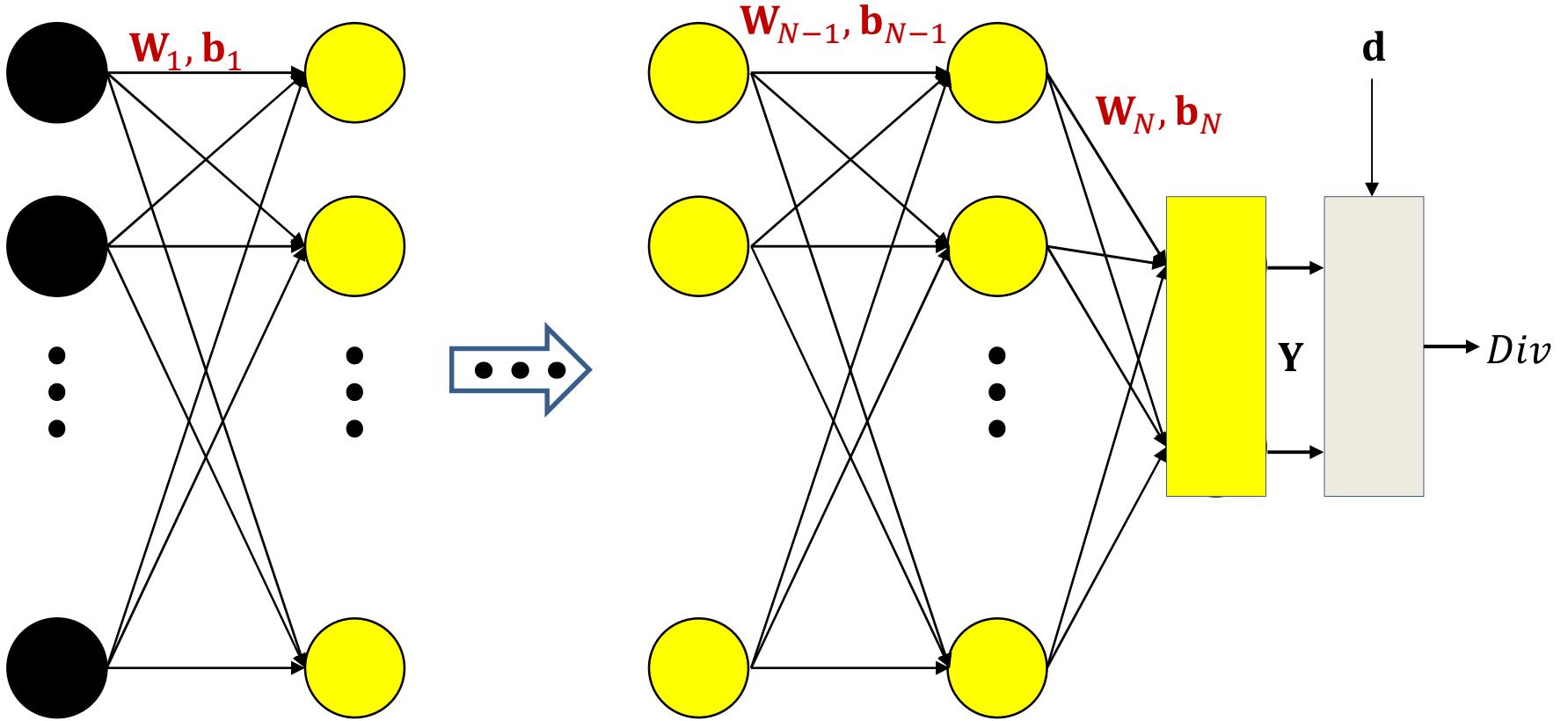
Special Case

- 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

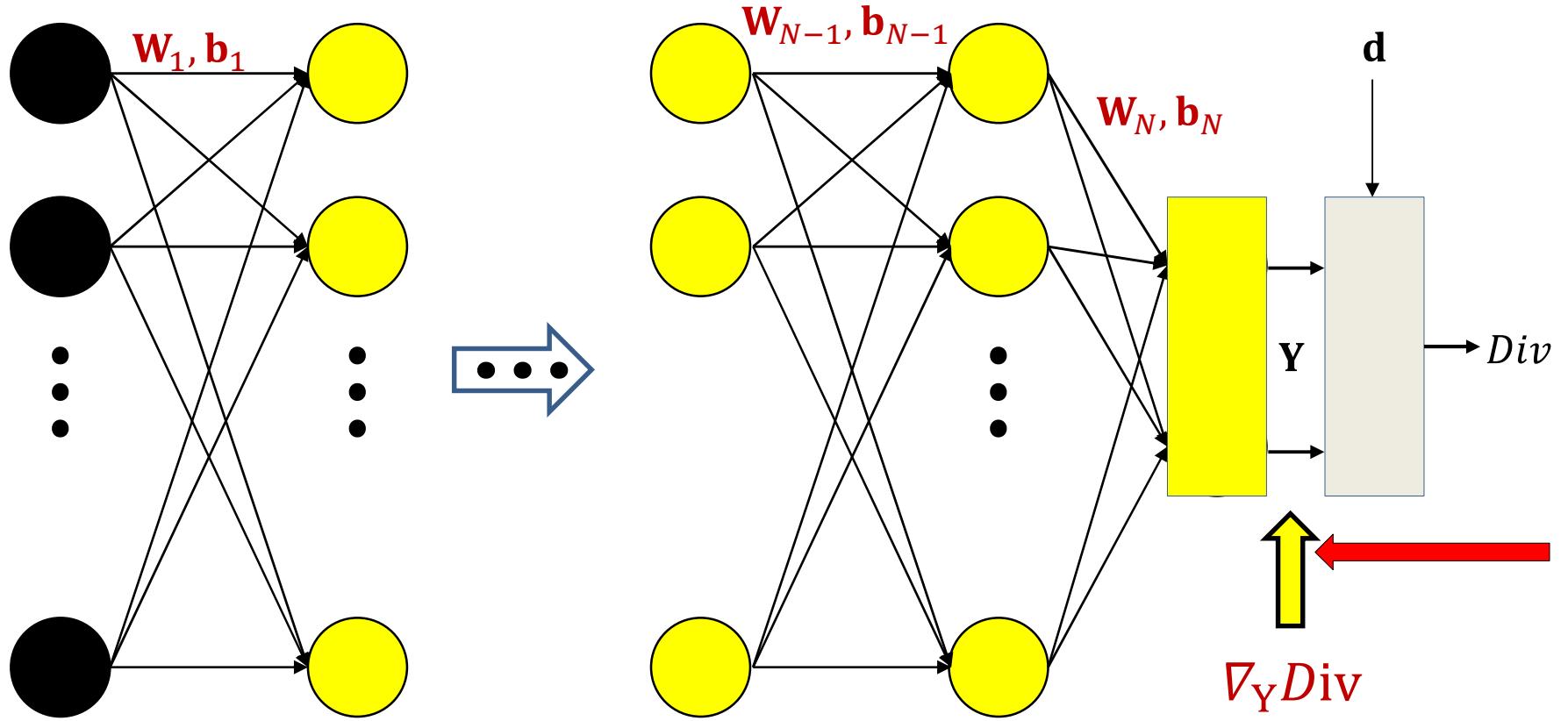
The backward pass



In the following slides we will also be using the notation $\nabla_{\mathbf{z}} \mathbf{Y}$ to represent the Jacobian $J_{\mathbf{Y}}(\mathbf{z})$ to explicitly illustrate the chain rule

In general $\nabla_{\mathbf{a}} \mathbf{b}$ 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})

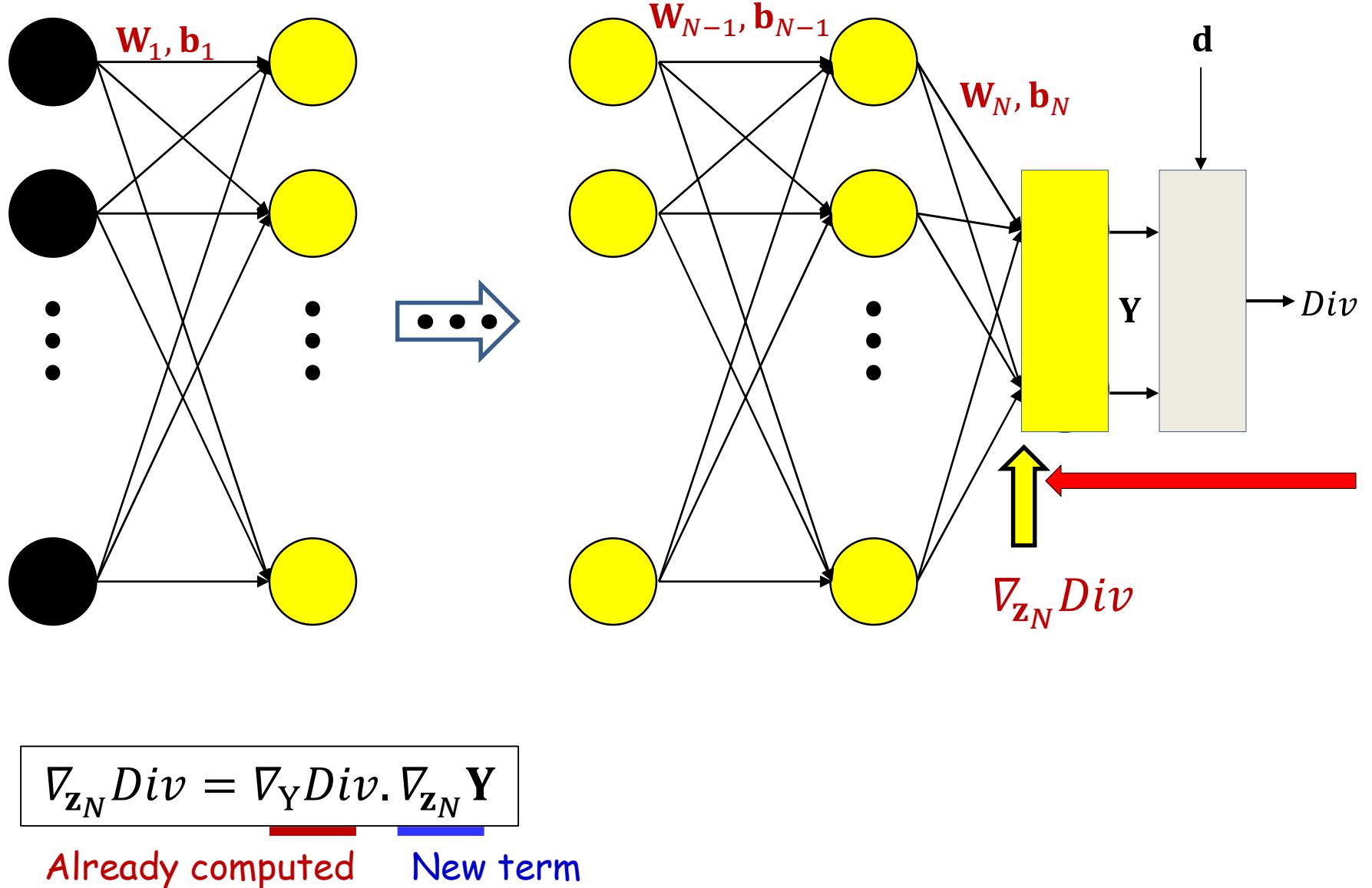
The backward pass



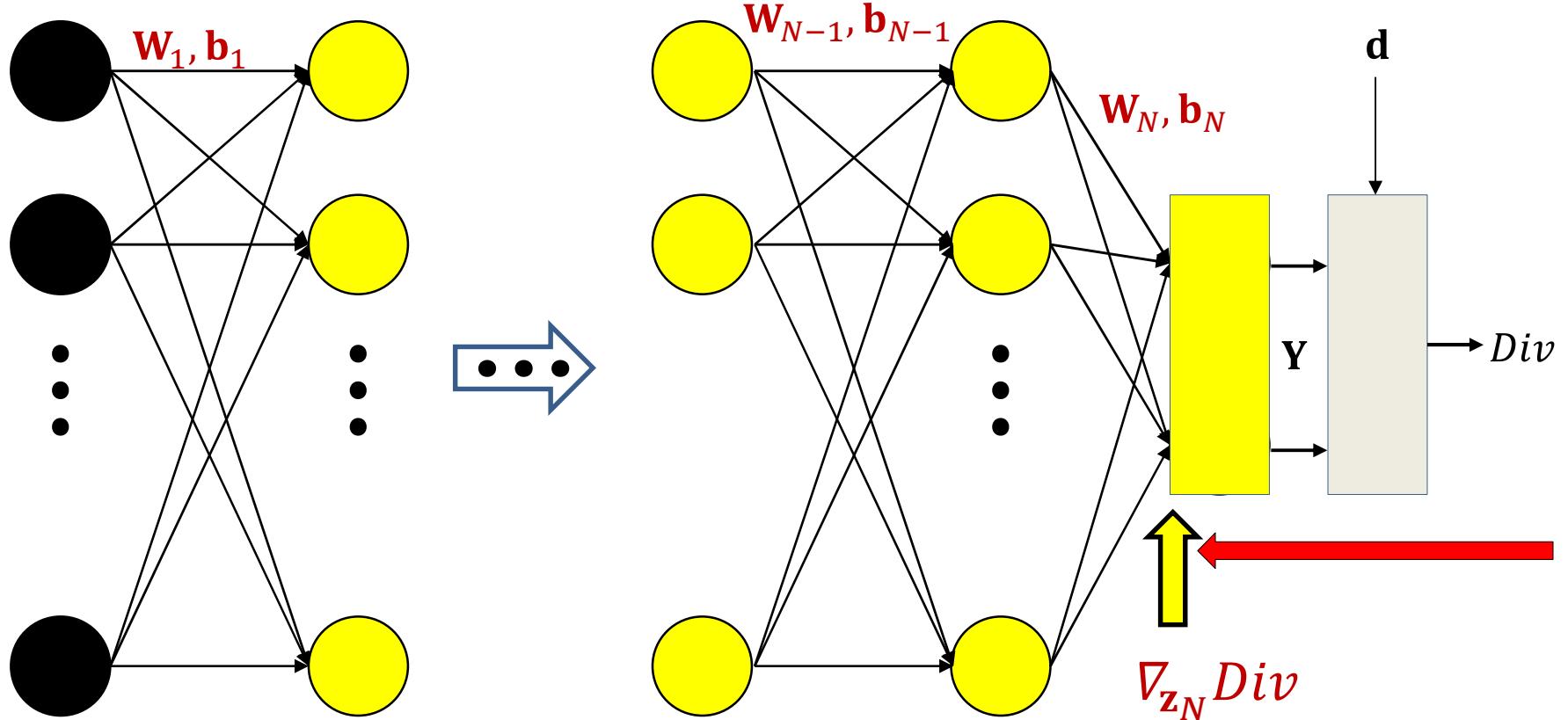
First compute the derivative of the divergence w.r.t. Y .
The actual derivative depends on the divergence function.

N.B: The gradient is the transpose of the derivative

The backward pass

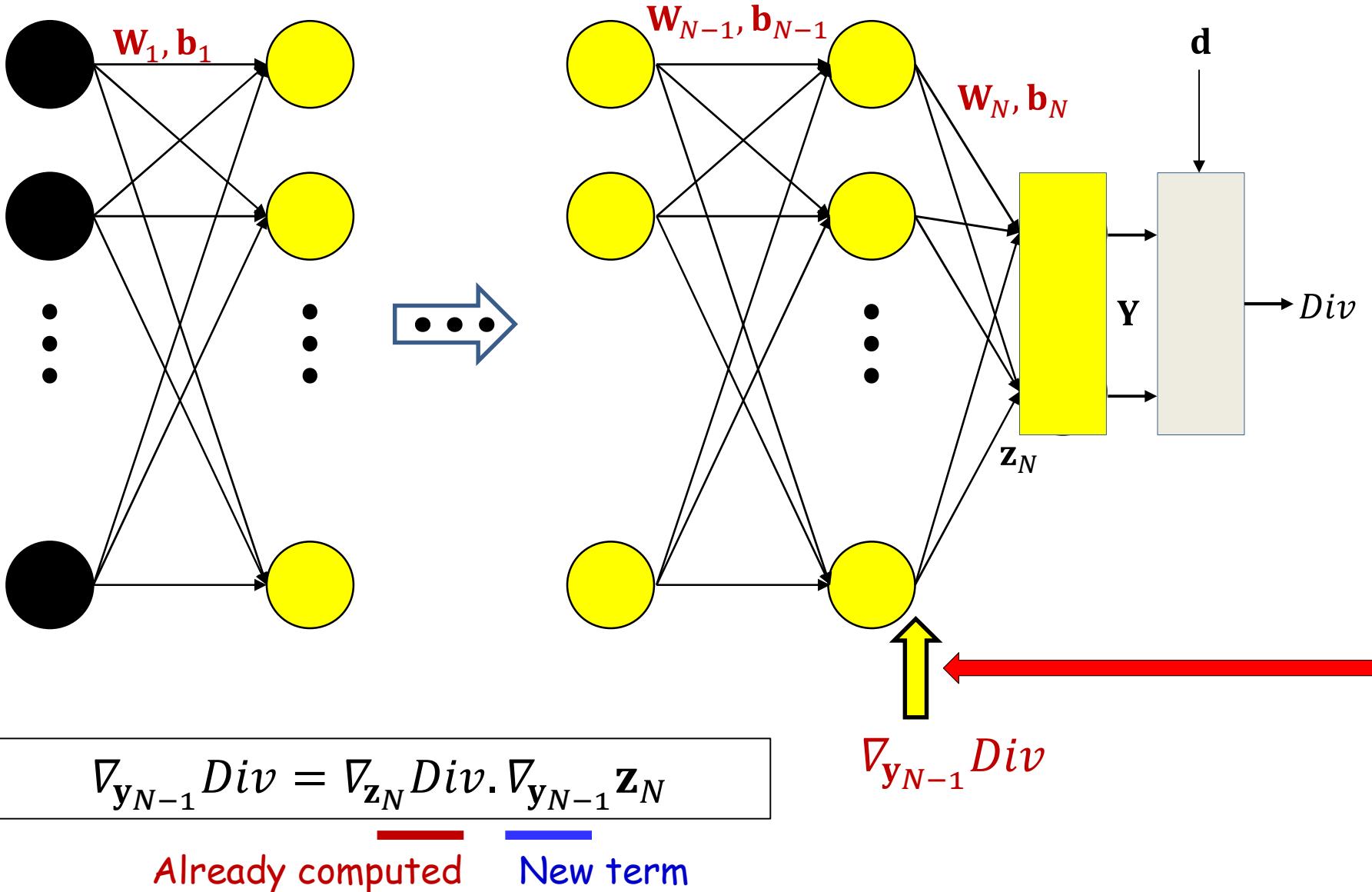


The backward pass

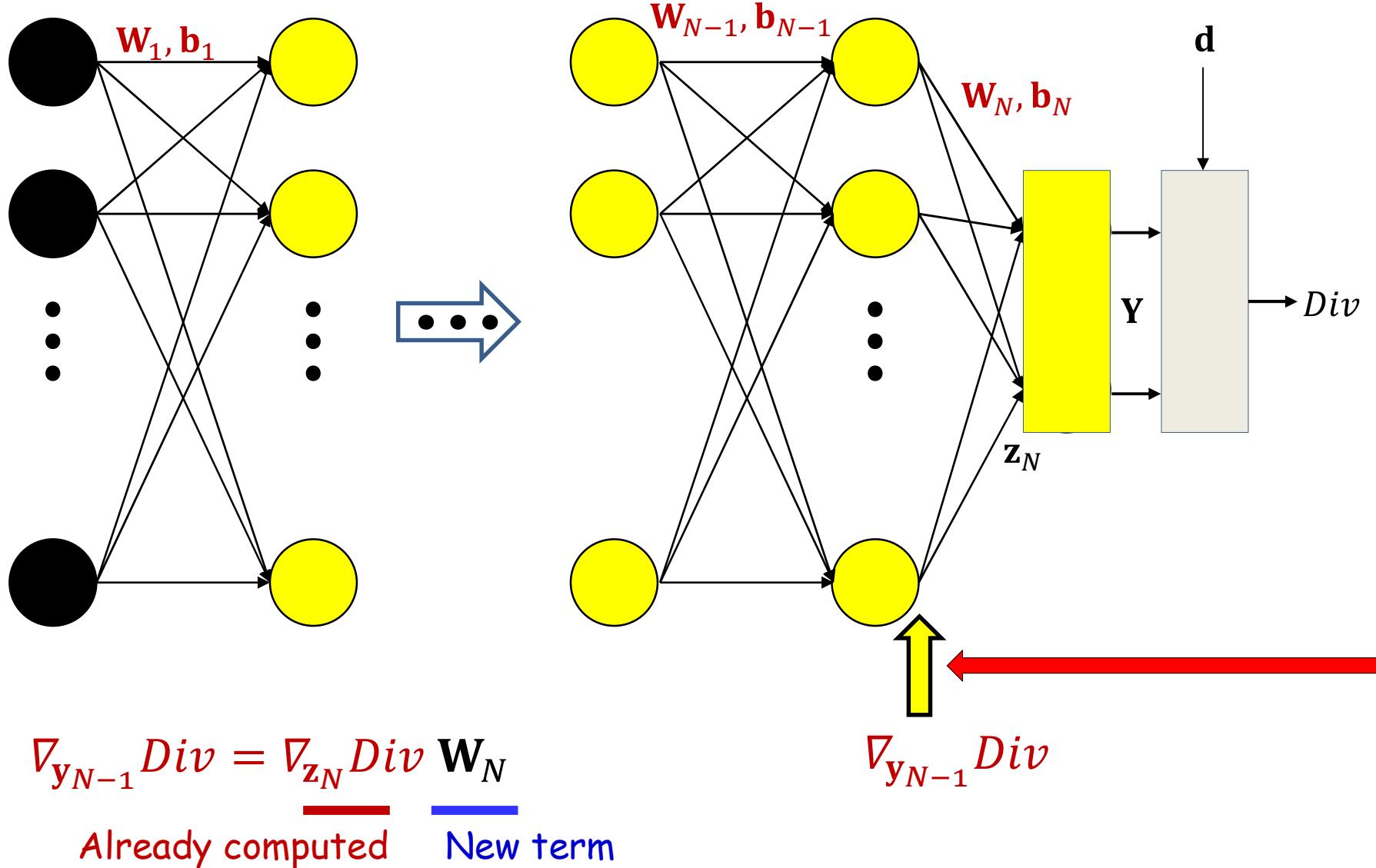


$$\nabla_{\mathbf{z}_N} Div = \underbrace{\nabla_{\mathbf{Y}} Div}_{\text{Already computed}} \underbrace{J_{\mathbf{Y}}(\mathbf{z}_N)}_{\text{New term}}$$

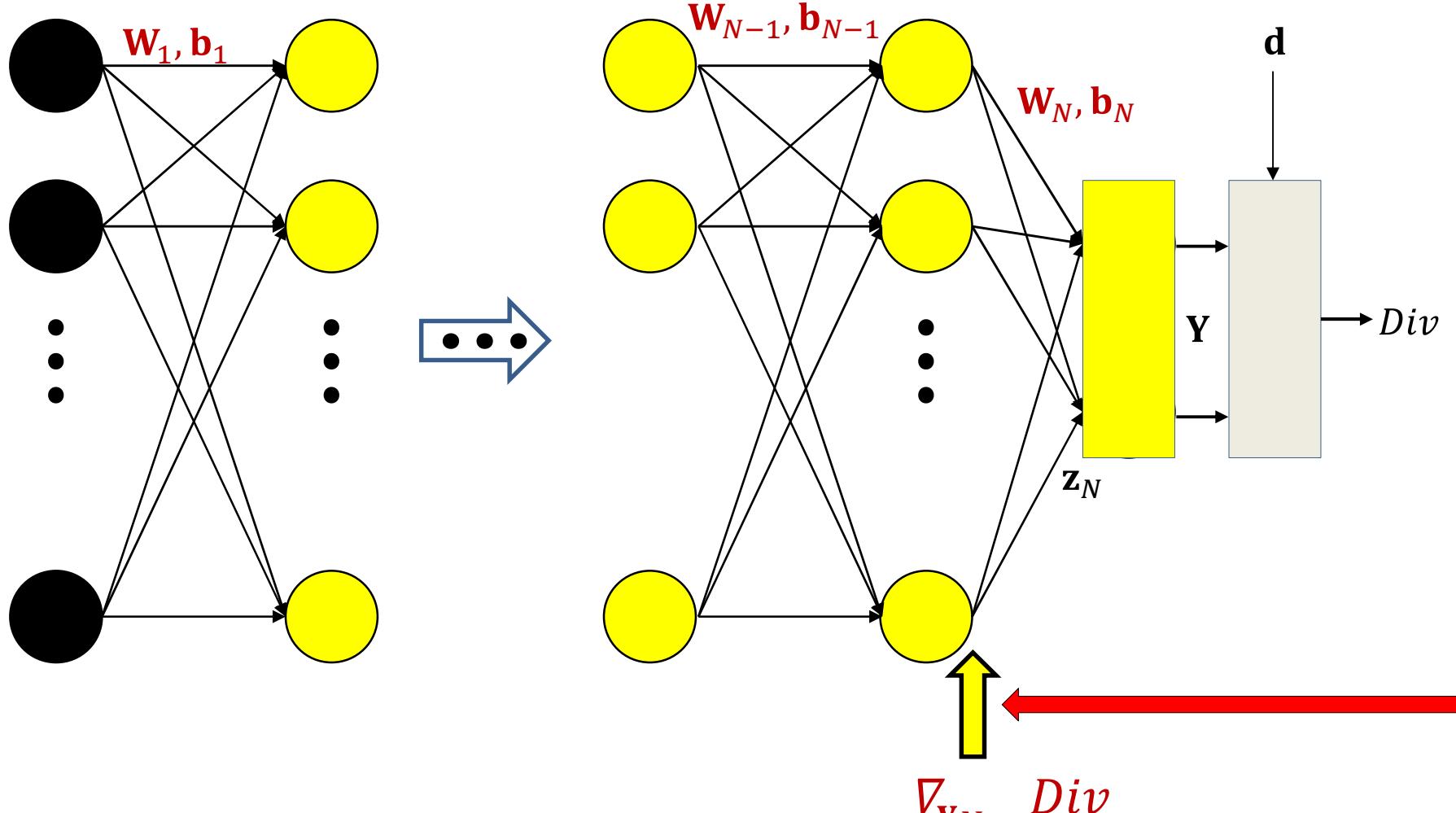
The backward pass



The backward pass



The backward pass

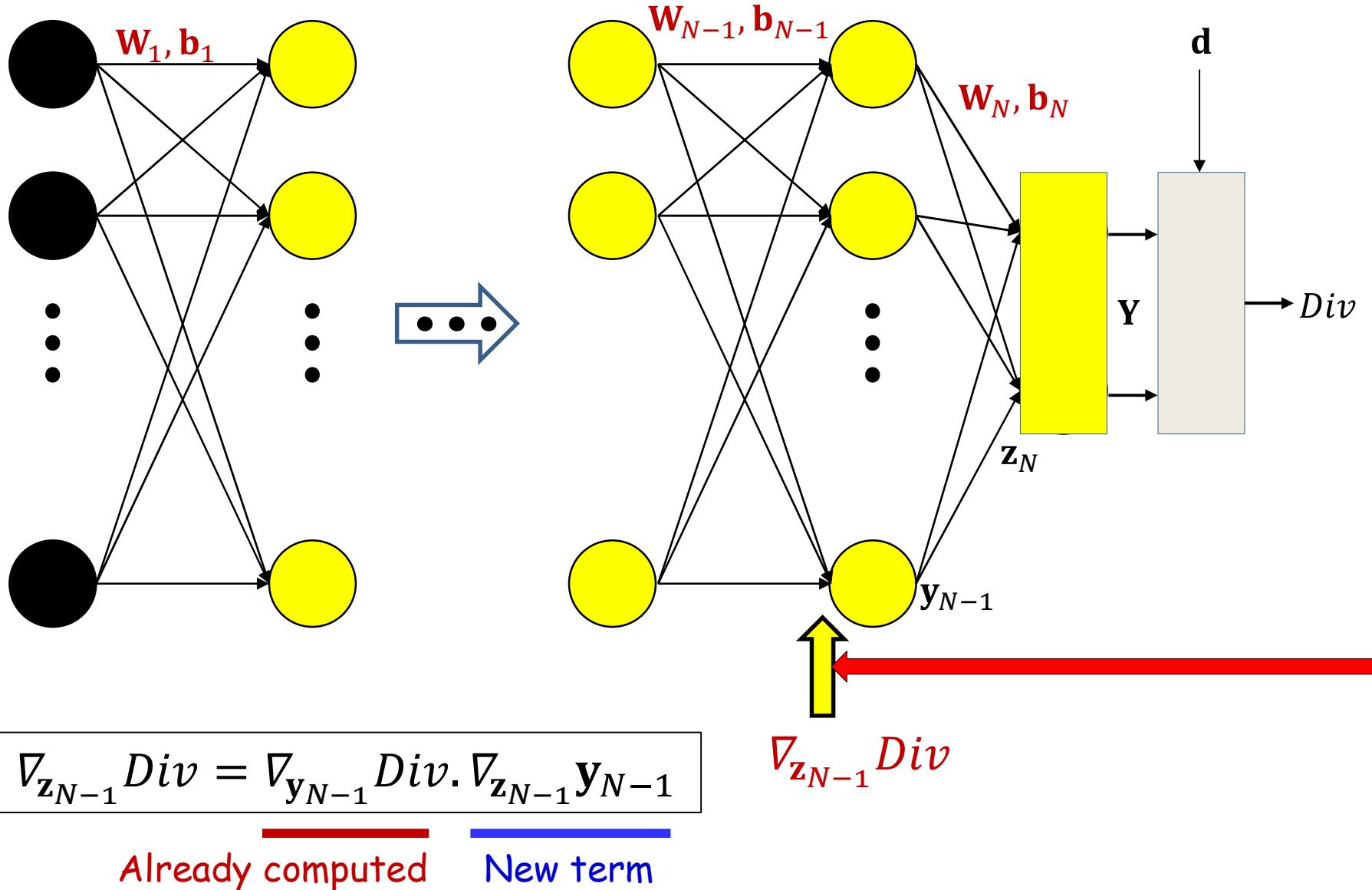


$$\nabla_{y_{N-1}} \text{Div} = \nabla_{z_N} \text{Div} \mathbf{W}_N$$

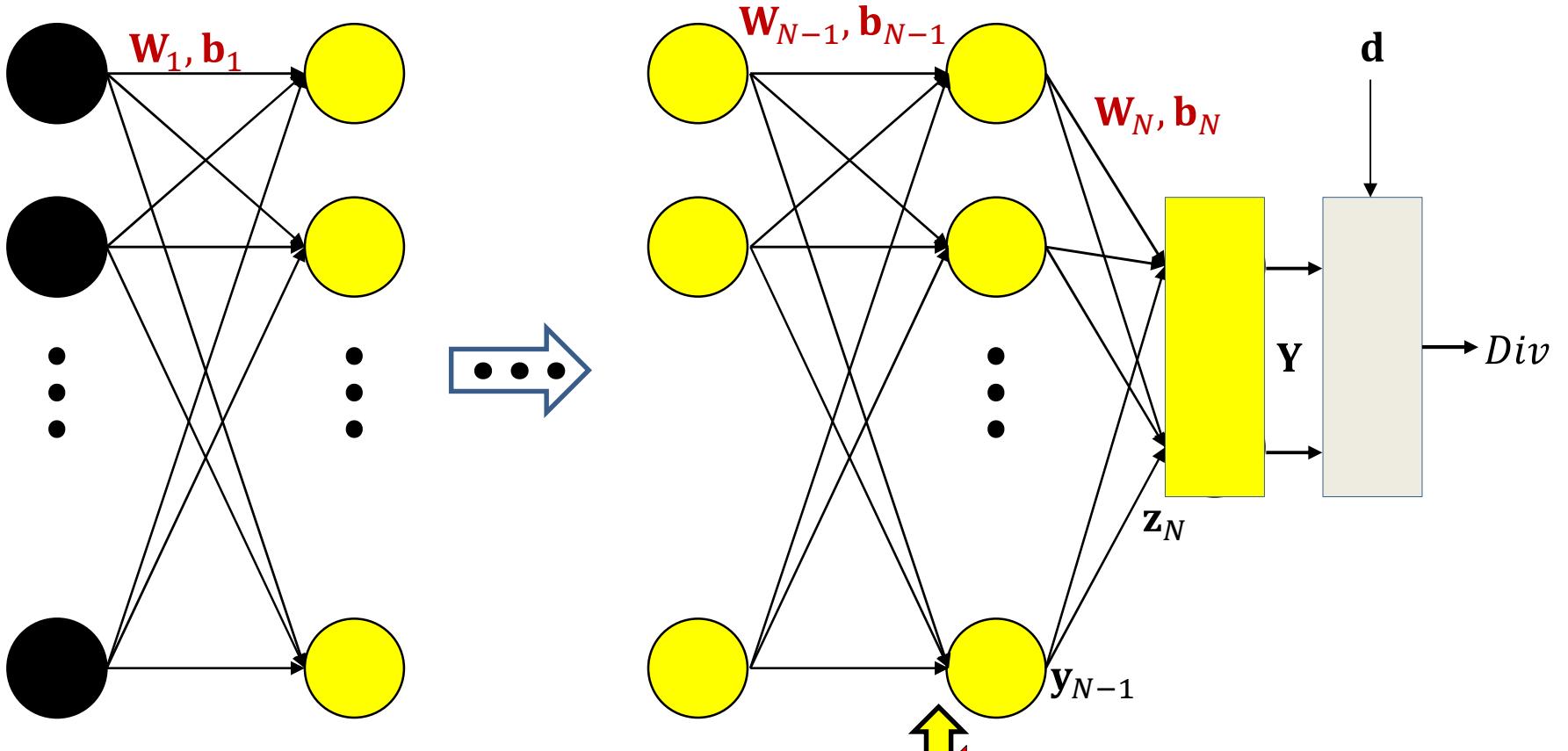
$$\nabla_{\mathbf{W}_N} \text{Div} = \mathbf{y}_{N-1} \nabla_{z_N} \text{Div}$$

$$\nabla_{\mathbf{b}_N} \text{Div} = \nabla_{z_N} \text{Div}$$

The backward pass



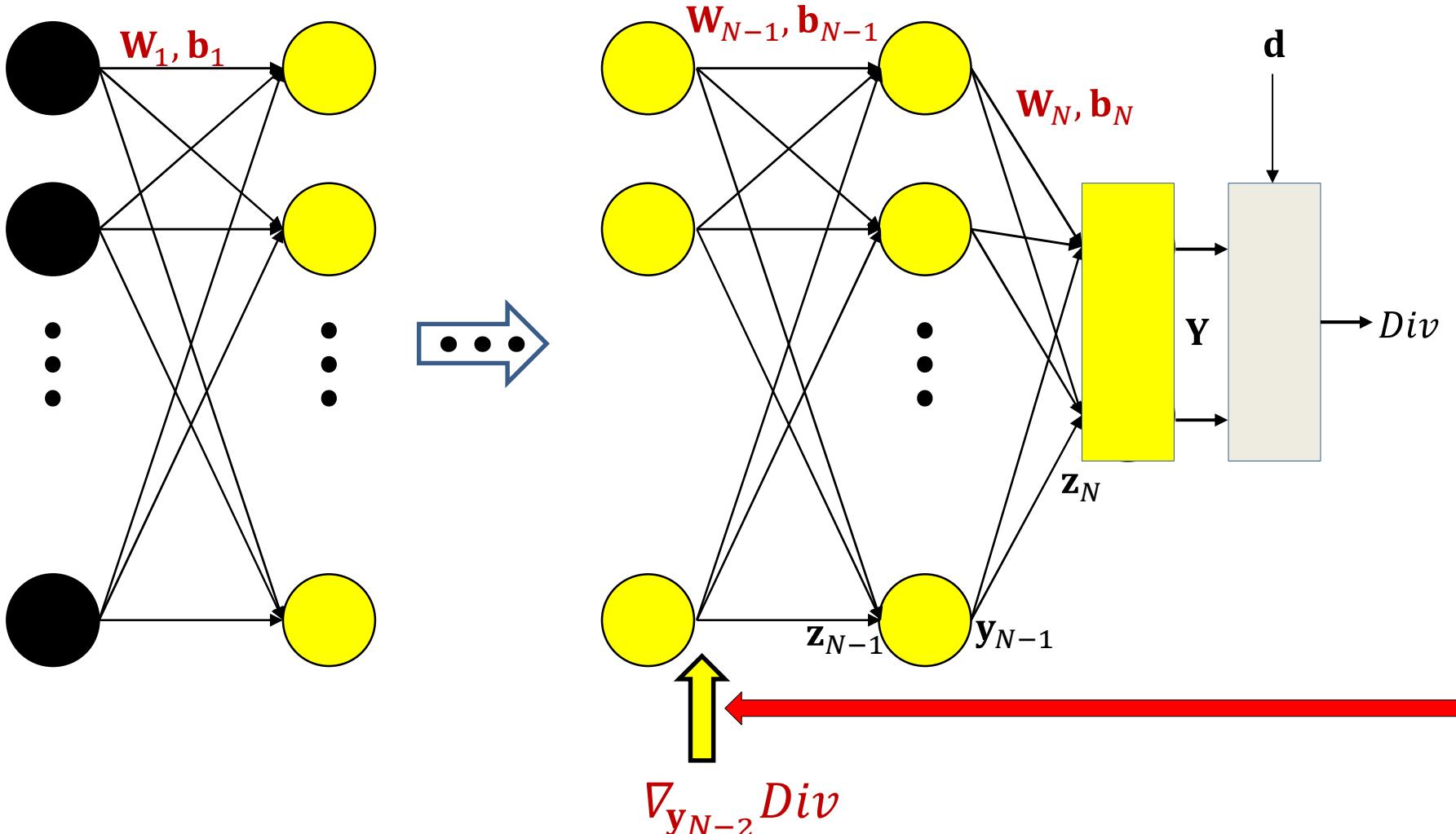
The backward pass



$$\nabla_{z_{N-1}} \text{Div} = \nabla_{y_{N-1}} \text{Div} J_{y_{N-1}}(z_{N-1})$$

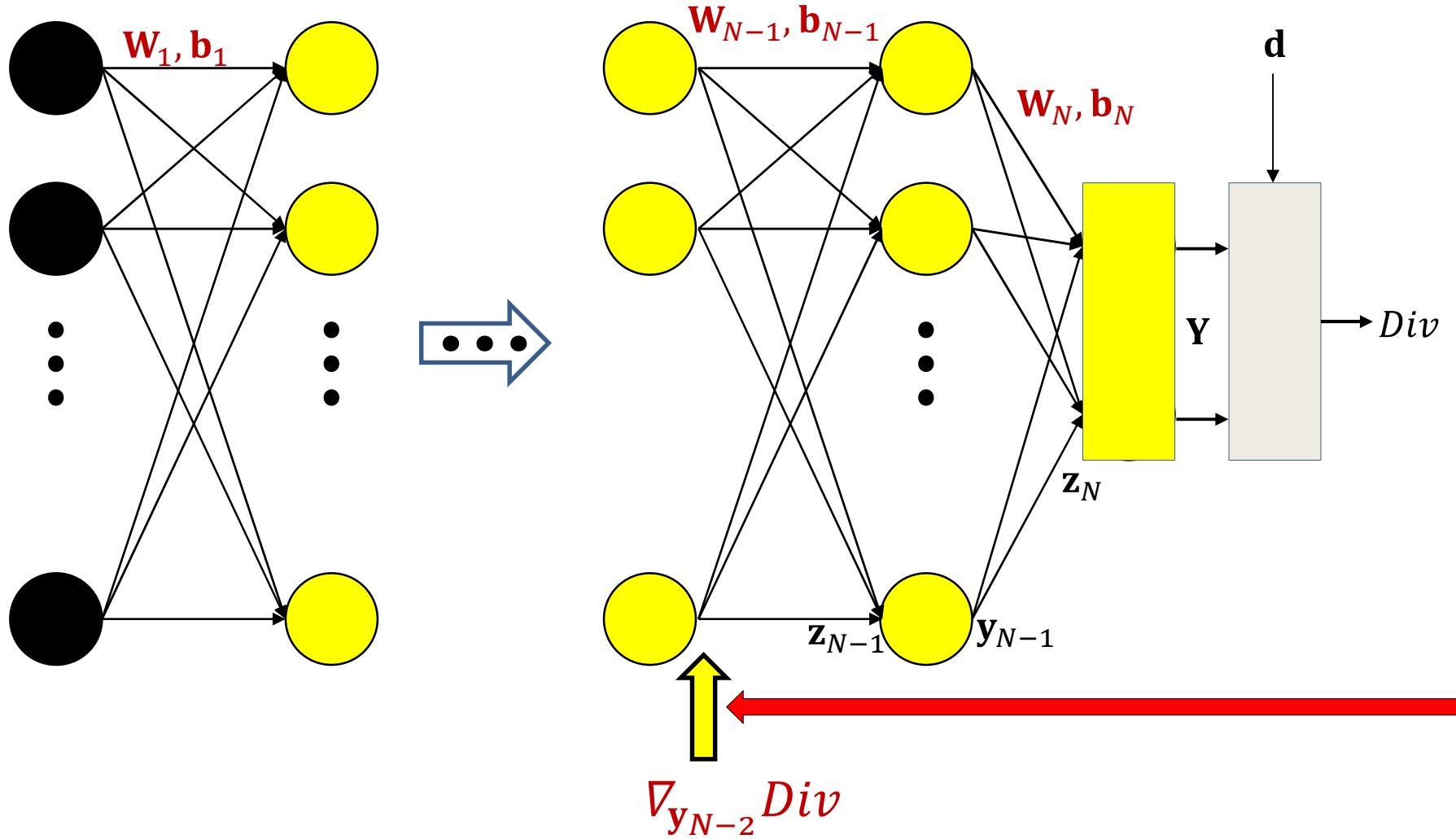
The Jacobian will be a diagonal matrix for scalar activations

The backward pass



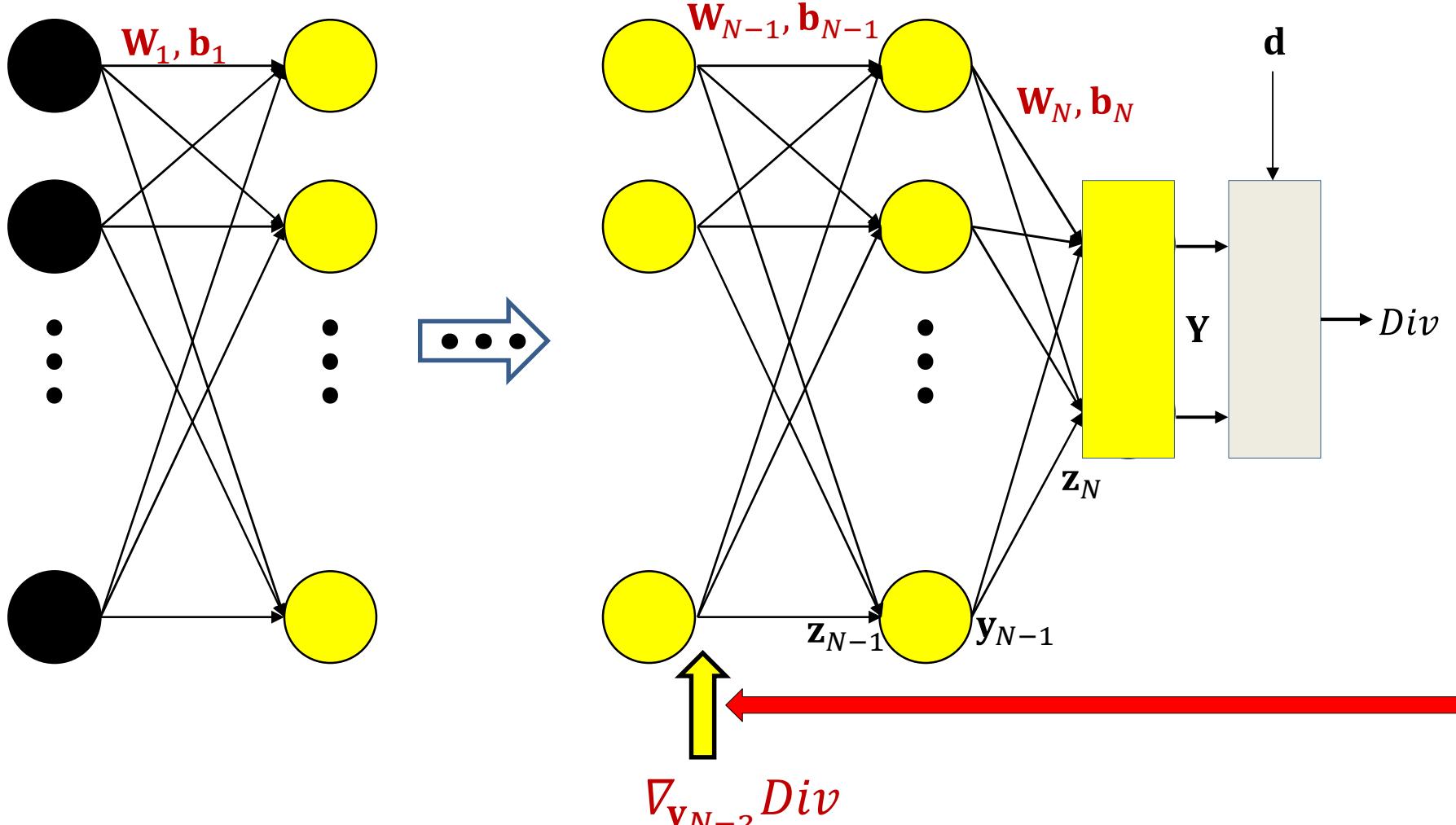
$$\nabla_{\mathbf{y}_{N-2}} Div = \nabla_{\mathbf{z}_{N-1}} Div \cdot \nabla_{\mathbf{y}_{N-2}} \mathbf{z}_{N-1}$$

The backward pass



$$\nabla_{y_{N-2}} Div = \nabla_{z_{N-1}} Div \mathbf{W}_{N-1}$$

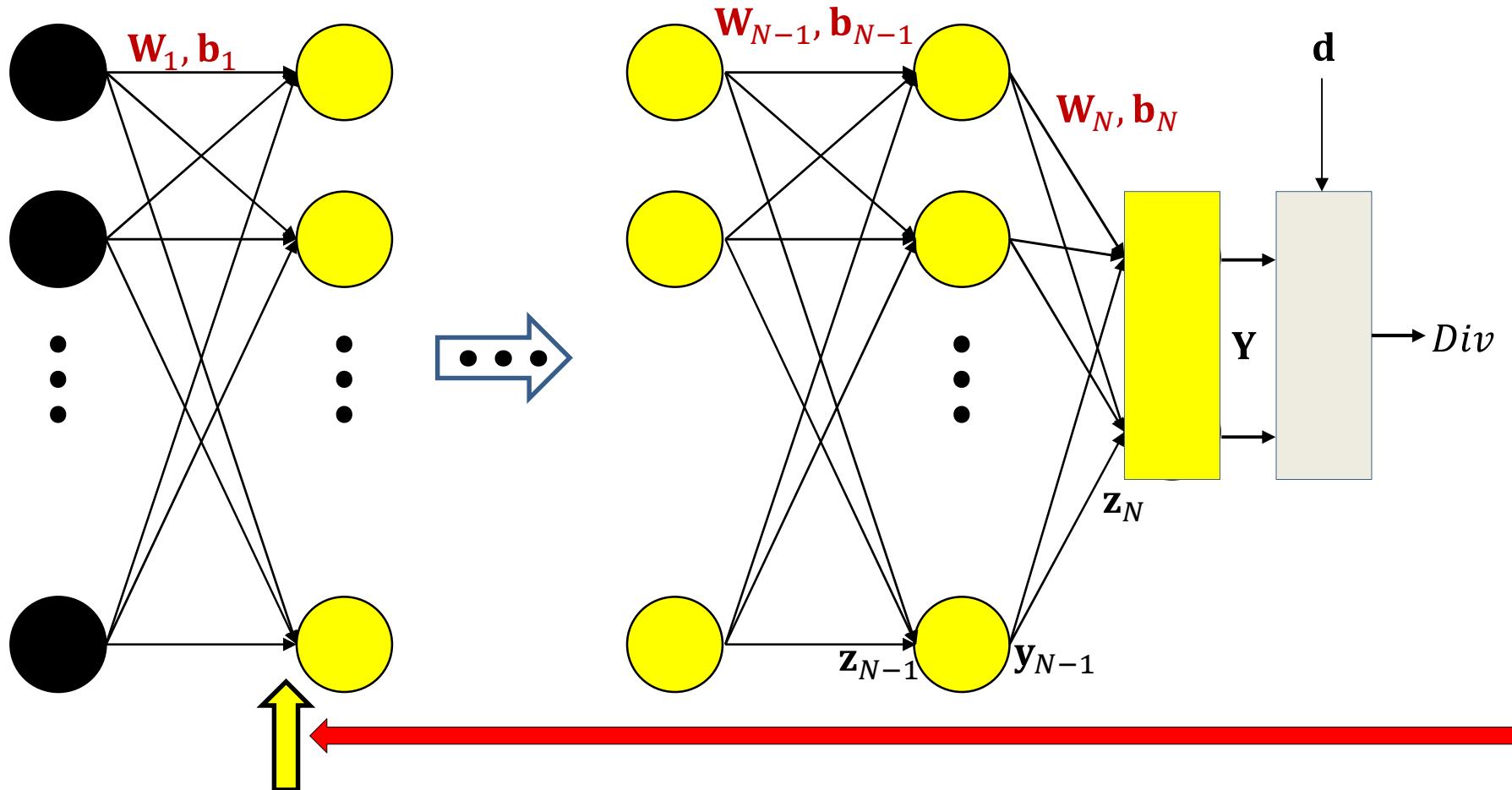
The backward pass



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

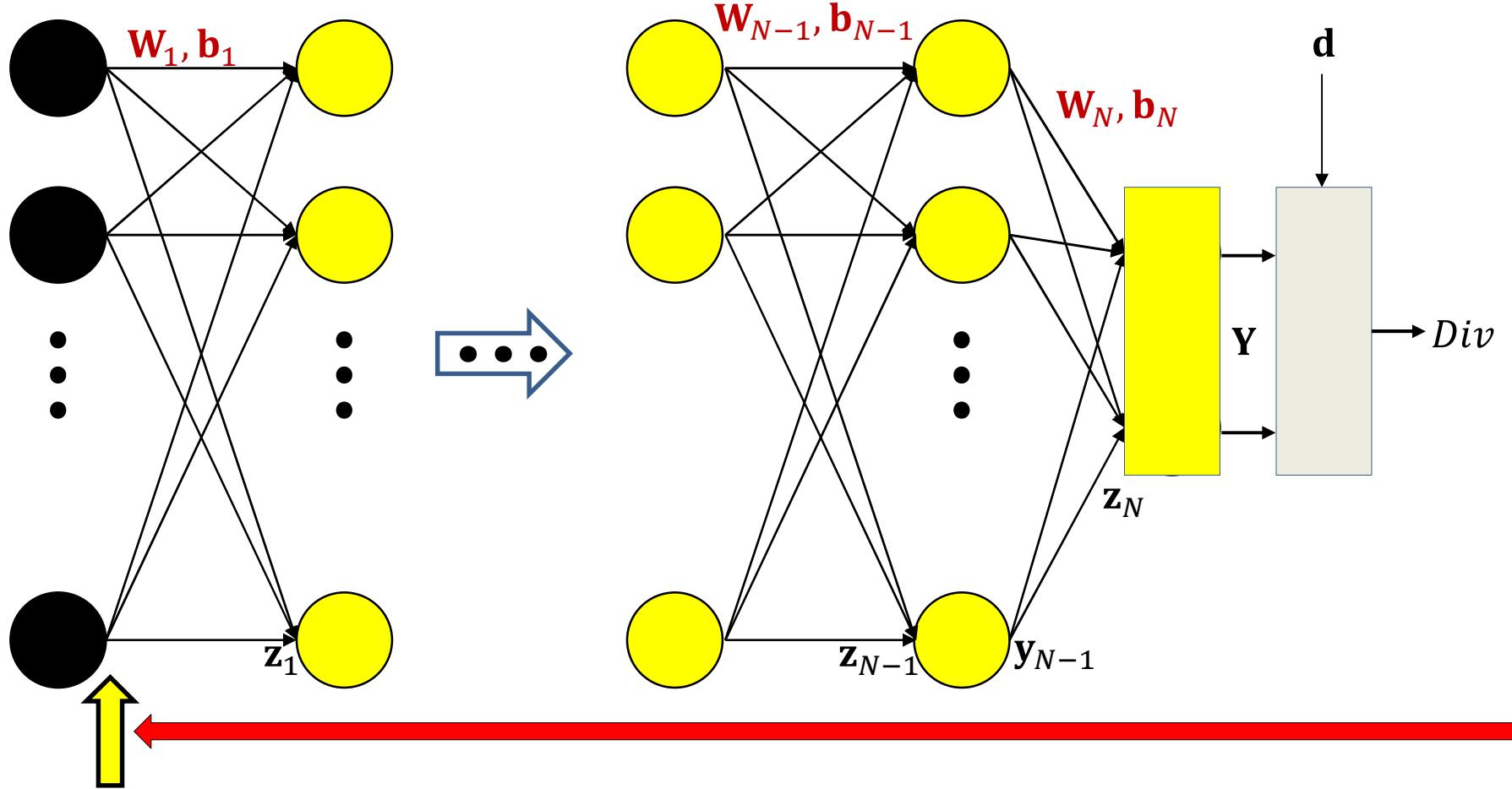
$\nabla_{W_{N-1}} Div = y_{N-2} \nabla_{z_{N-1}} Div$
$\nabla_{b_{N-1}} Div = \nabla_{z_{N-1}} Div$

The backward pass



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

The backward pass



$$\nabla_{W_1} Div = \mathbf{x} \nabla_{z_1} Div$$

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

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 $k = N$ down to 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:

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

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

- Output:

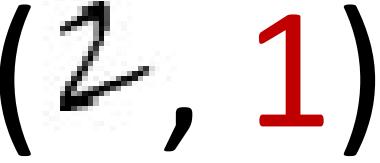
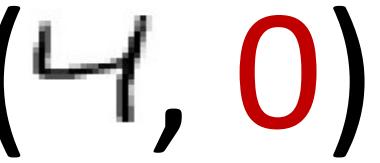
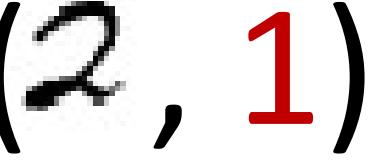
$$\mathbf{Y} = \mathbf{y}_N$$

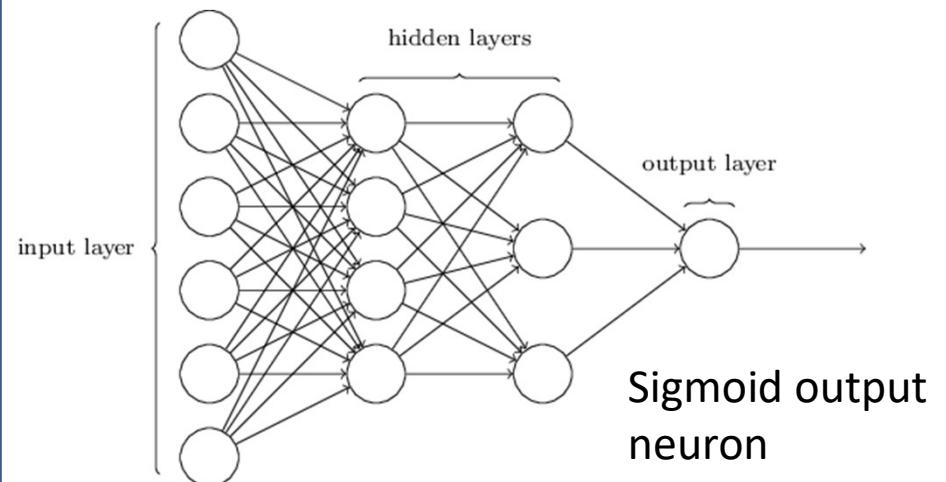
Neural network training algorithm

- 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 $\mathbf{Y}(X_t)$
 - 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

( , 0)	( , 1)
( , 1)	( , 0)
( , 0)	( , 1)

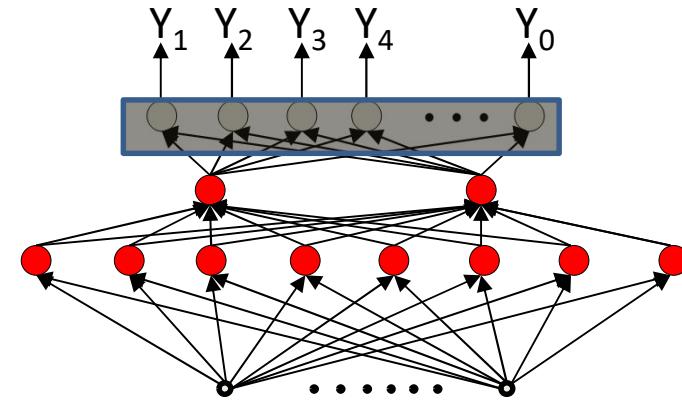


- Simple Problem: Recognizing “2” or “not 2”
- Single output with sigmoid activation
 - $Y \in (0,1)$
 - d is either 0 or 1
- Use KL divergence
- Backpropagation to learn network parameters

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

Story so far

- Neural networks must be trained to minimize the average divergence between the output of the network and the desired output over a set of training instances, with respect to network parameters.
- Minimization is performed using gradient descent
- Gradients (derivatives) of the divergence (for any individual instance) w.r.t. network parameters can be computed using backpropagation
 - Which requires a “forward” pass of inference followed by a “backward” pass of gradient computation
- The computed gradients can be incorporated into gradient descent

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..*

Next up

- Convergence and generalization