

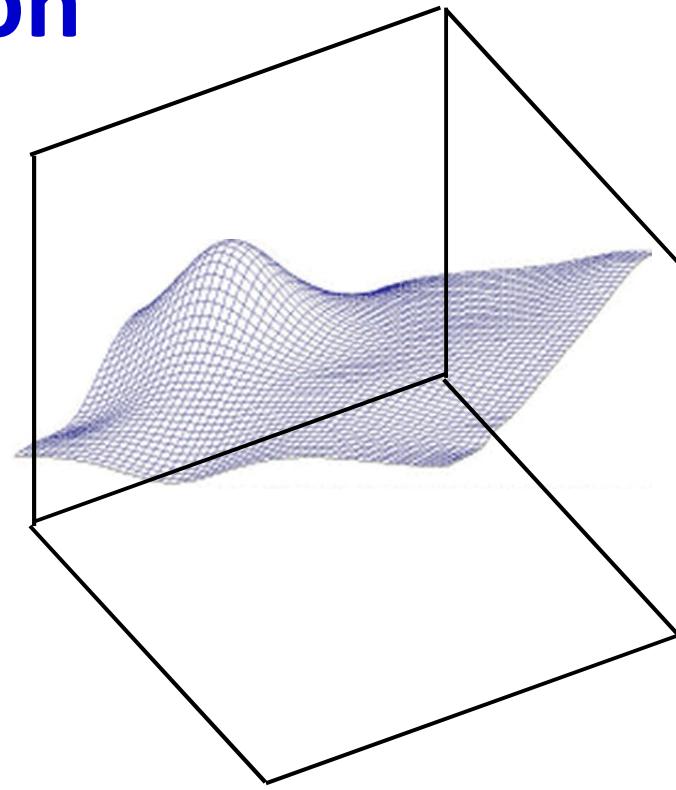
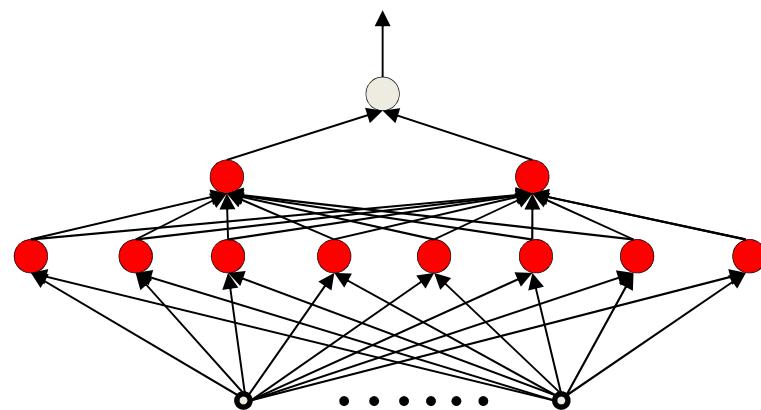
# **Neural Networks**

## **Learning the network: Backprop**

11-785, Fall 2018

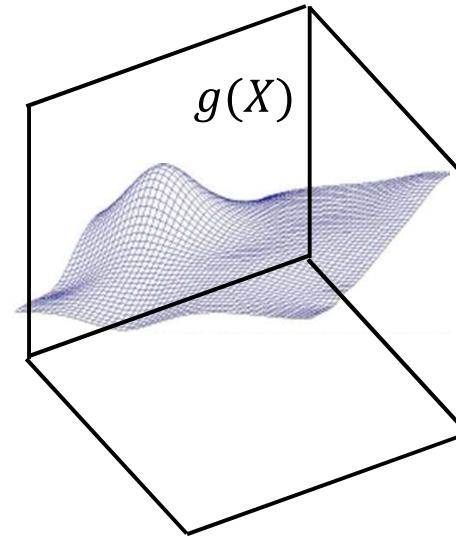
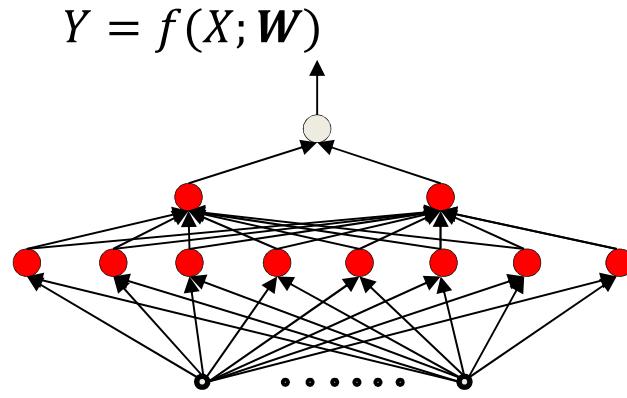
Lecture 4

# Recap: The MLP *can* represent any function



- The MLP *can be constructed* to represent anything
- But *how* do we construct it?

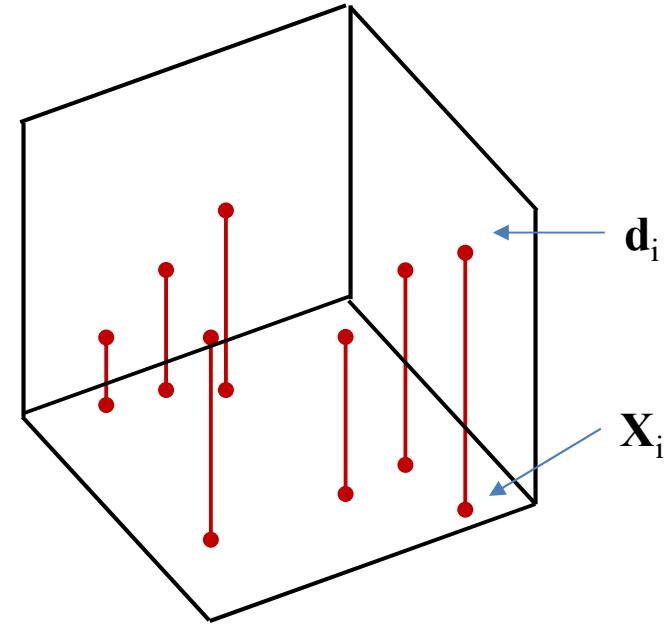
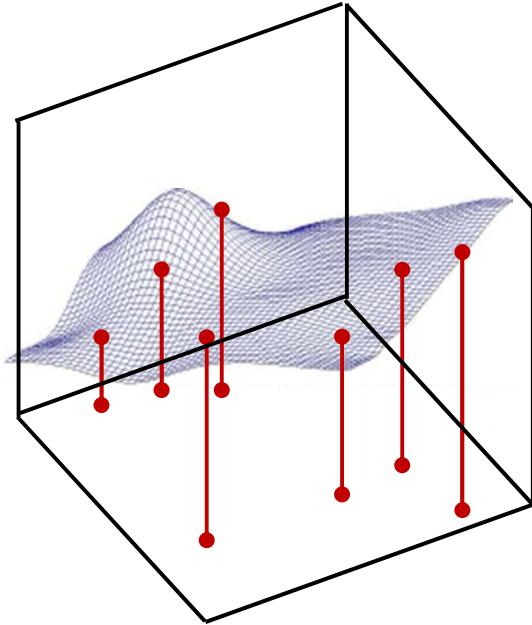
# Recap: How to learn the function



- By minimizing expected error

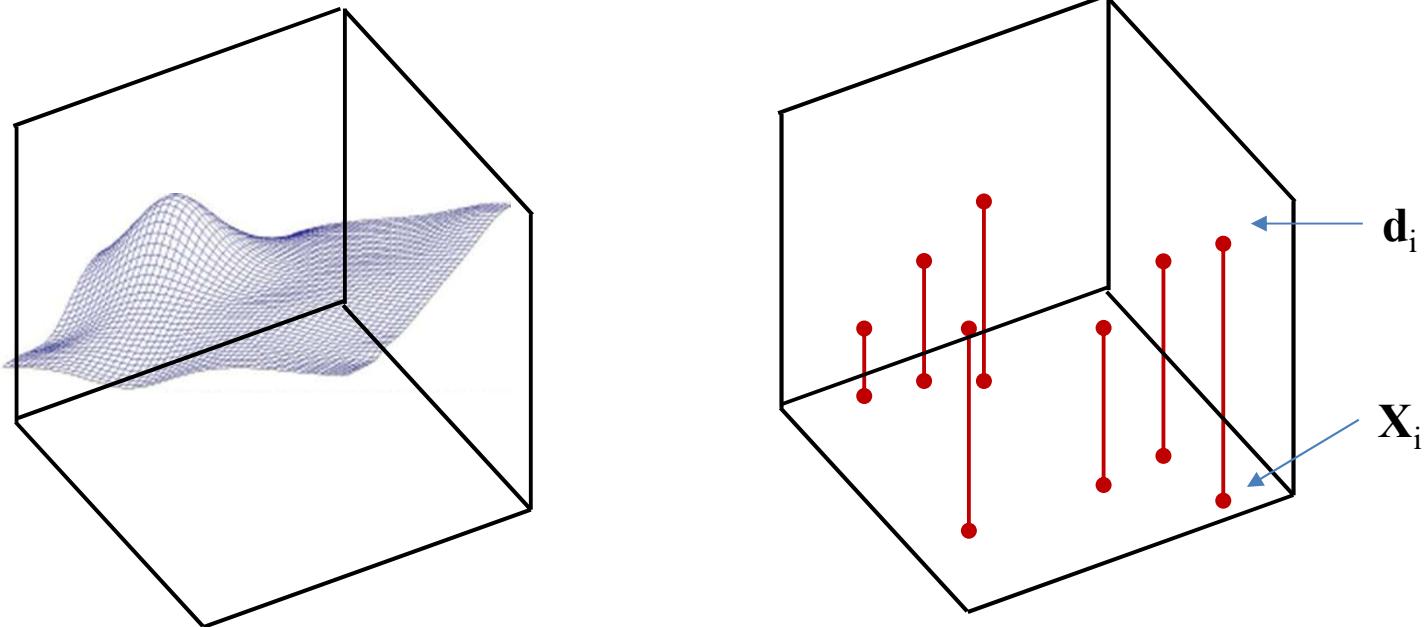
$$\begin{aligned}\widehat{W} &= \operatorname{argmin}_W \int_X \operatorname{div}(f(X; W), g(X)) P(X) dX \\ &= \operatorname{argmin}_W E[\operatorname{div}(f(X; W), g(X))]\end{aligned}$$

# Recap: Sampling the function



- $g(X)$  is unknown, so sample it
  - Basically, get input-output pairs for a number of samples of input  $X_i$ 
    - Many samples  $(X_i, d_i)$ , where  $d_i = g(X_i) + \text{noise}$
    - Good sampling: the samples of  $X$  will be drawn from  $P(X)$
  - Estimate function from the samples

# The *Empirical* risk



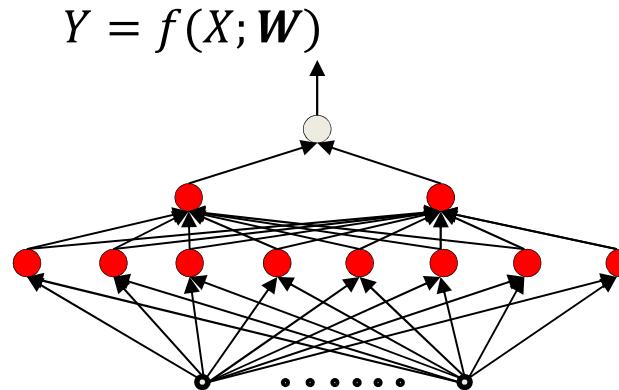
- The *expected* error is the average error over the entire input space

$$E[div(f(X; W), g(X))] = \int_X div(f(X; W), g(X))P(X)dX$$

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

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

# Empirical Risk Minimization



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

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

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

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

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

# 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

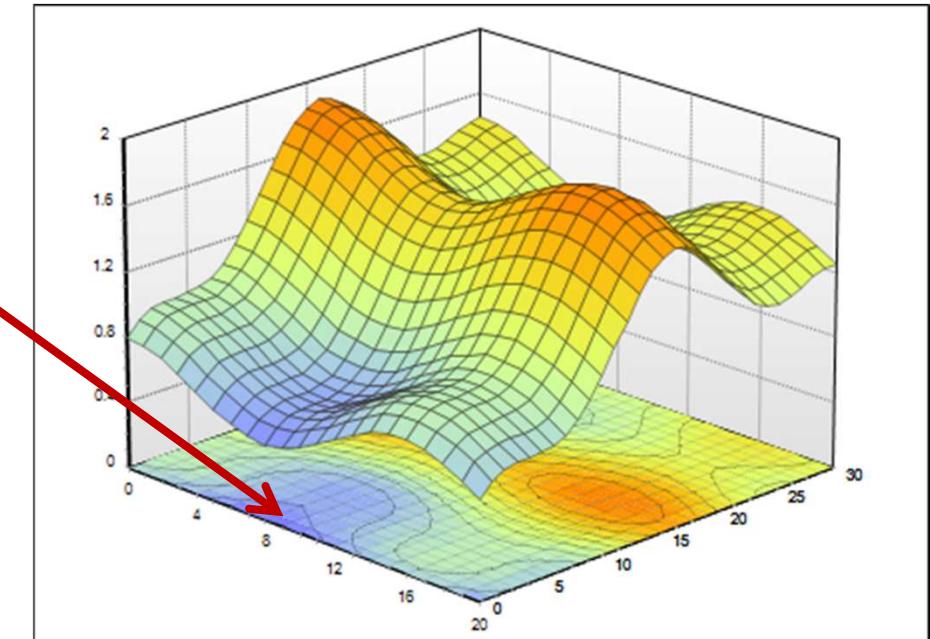
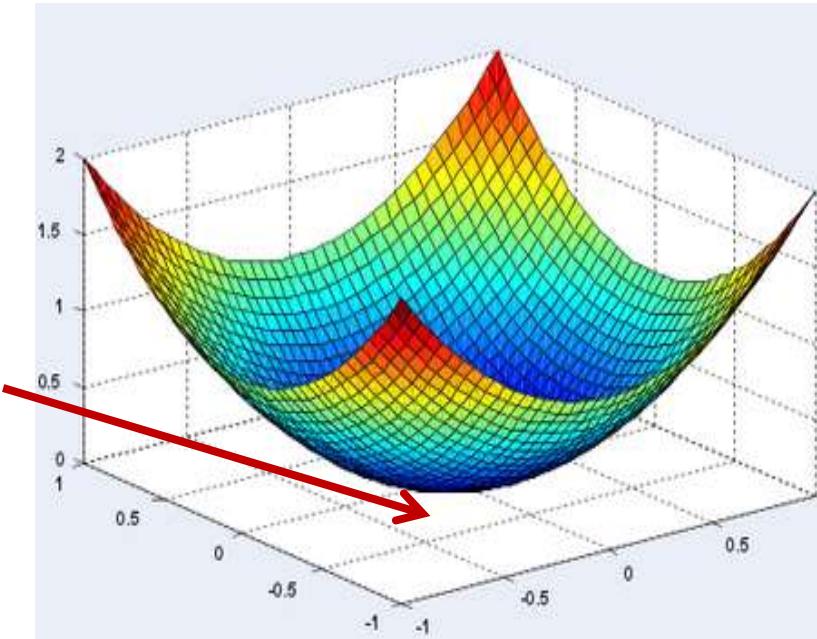
$$Err(W) = \frac{1}{T} \sum_i \text{div}(f(X_i; W), d_i)$$

w.r.t  $W$

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

- A CRASH COURSE ON FUNCTION  
OPTIMIZATION

# Finding the minimum of a scalar function of a multi-variate input



- The optimum point is a turning point – the gradient will be 0

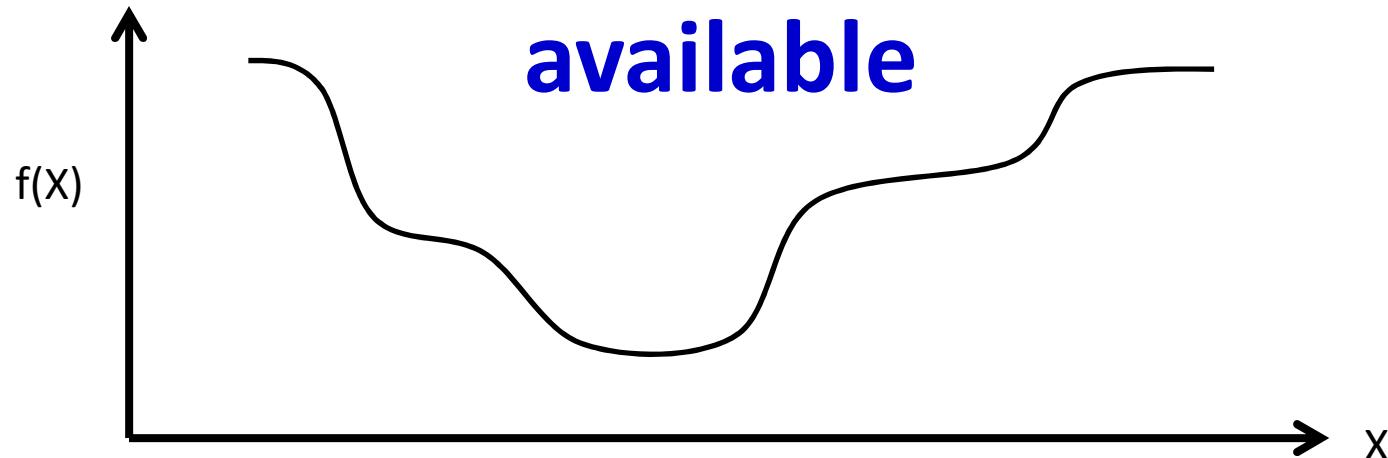
# Unconstrained Minimization of function (Multivariate)

1. Solve for the  $X$  where the gradient equation equals to zero

$$\nabla f(X) = 0$$

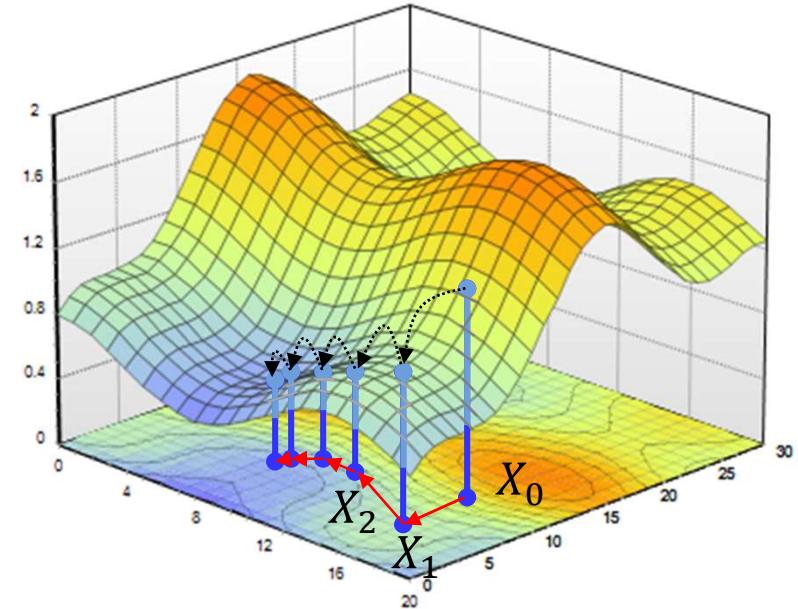
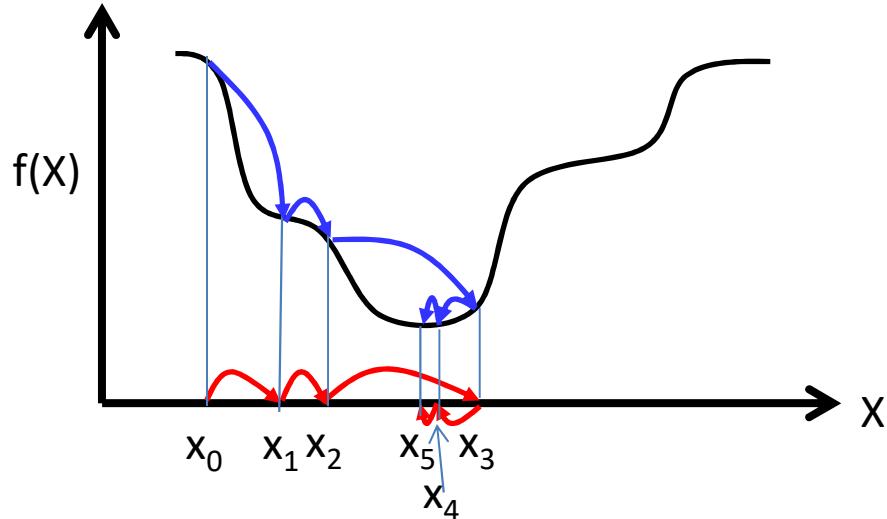
2. Compute the Hessian Matrix  $\nabla^2 f(X)$  at the candidate solution and verify that
  - Hessian is positive definite (eigenvalues positive) -> to identify local minima
  - Hessian is negative definite (eigenvalues negative) -> to identify local maxima

# Closed Form Solutions are not always available



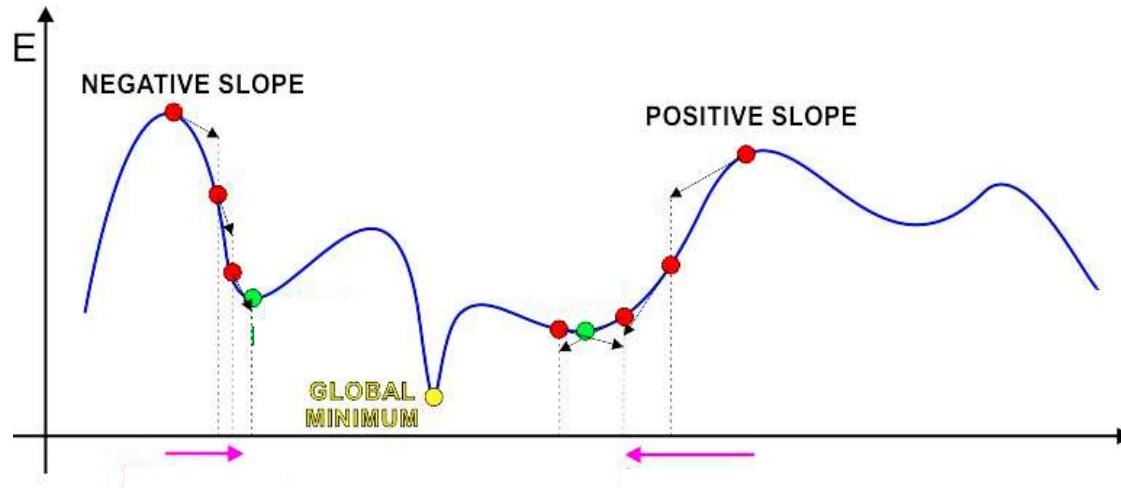
- Often it is not possible to simply solve  $\nabla f(X) = 0$ 
  - The function to minimize/maximize may have an intractable form
- In these situations, iterative solutions are used
  - Begin with a “guess” for the optimal  $X$  and refine it iteratively until the correct value is obtained

# Iterative solutions



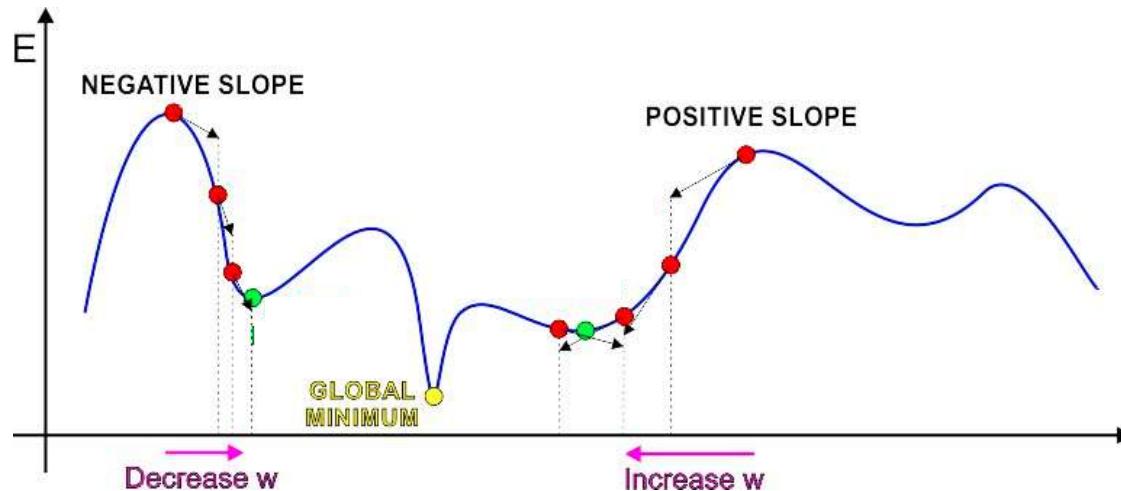
- Iterative solutions
  - Start from an initial guess  $X_0$  for the optimal  $X$
  - Update the guess towards a (hopefully) “better” value of  $f(X)$
  - Stop when  $f(X)$  no longer decreases
- Problems:
  - Which direction to step in
  - How big must the steps be

# The Approach of Gradient Descent



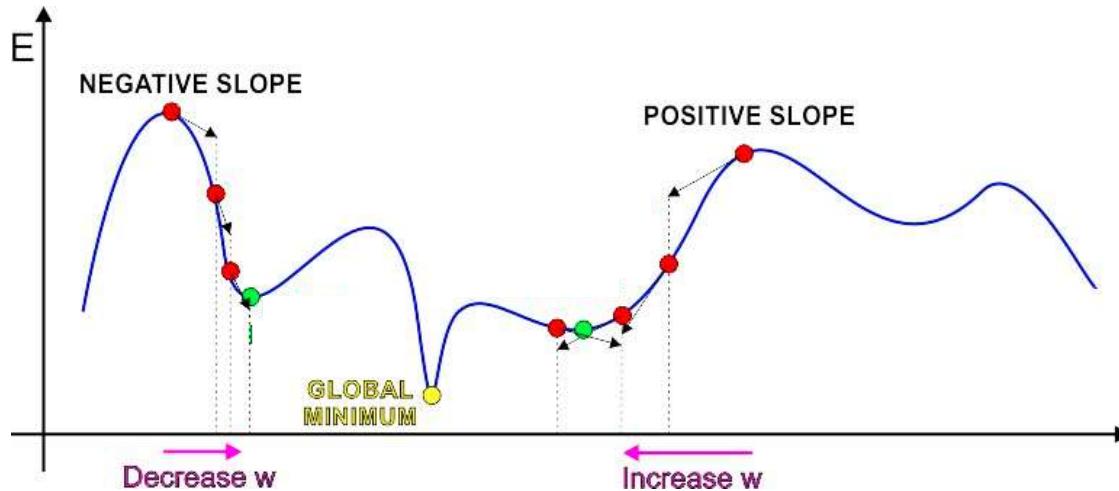
- Iterative solution:
  - Start at some point
  - Find direction in which to shift this point to decrease error
    - This can be found from the derivative of the function
      - A positive derivative → moving left decreases error
      - A negative derivative → moving right decreases error
  - Shift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

# The Approach of Gradient Descent



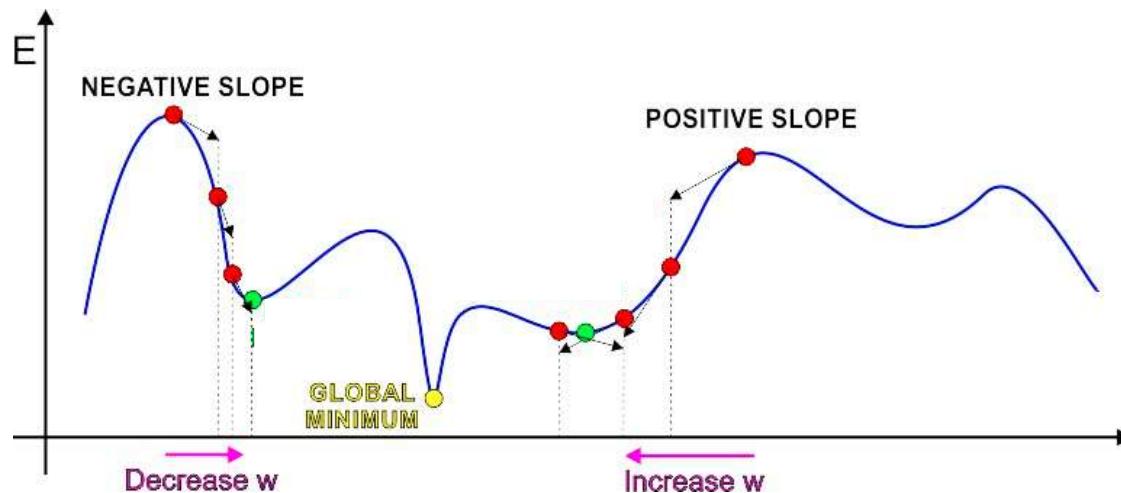
- Iterative solution: Trivial algorithm
  - Initialize  $x^0$
  - While  $f'(x^k) \neq 0$ 
    - If  $\text{sign}(f'(x^k))$  is positive:
      - $x^{k+1} = x^k - \text{step}$
    - Else
      - $x^{k+1} = x^k + \text{step}$
  - What must step be to ensure we actually get to the optimum?

# The Approach of Gradient Descent



- Iterative solution: Trivial algorithm
  - Initialize  $x^0$
  - While  $f'(x^k) \neq 0$ 
    - $x^{k+1} = x^k - sign(f'(x^k)).step$
  - Identical to previous algorithm

# The Approach of Gradient Descent



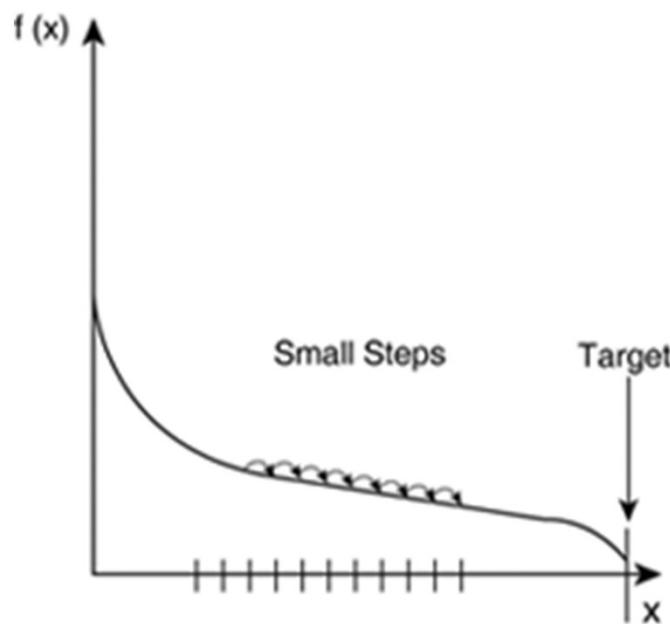
- Iterative solution: Trivial algorithm
  - Initialize  $x_0$
  - While  $f'(x^k) \neq 0$ 
    - $x^{k+1} = x^k - \eta^k f'(x^k)$
  - $\eta^k$  is the “step size”

# Gradient descent/ascent (multivariate)

- The gradient descent/ascent method to find the minimum or maximum of a function  $f$  iteratively
  - To find a *maximum* move *in the direction of the gradient*
$$x^{k+1} = x^k + \eta^k \nabla f(x^k)^T$$
  - To find a *minimum* move *exactly opposite the direction of the gradient*
$$x^{k+1} = x^k - \eta^k \nabla f(x^k)^T$$
- Many solutions to choosing step size  $\eta^k$ 
  - Later lecture

# 1. Fixed step size

- Fixed step size
  - Use fixed value for  $\eta^k$

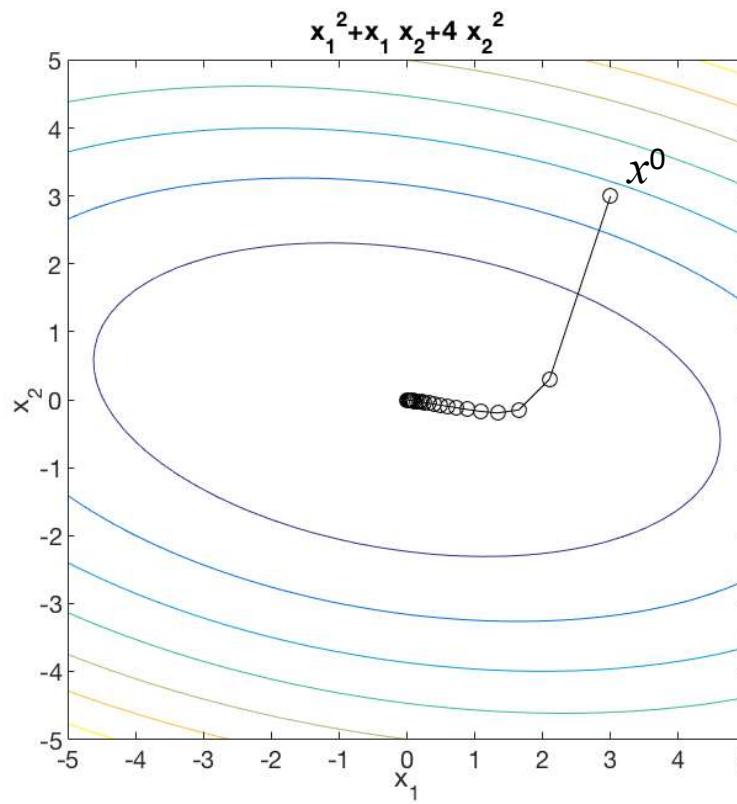


# Influence of step size example (constant step size)

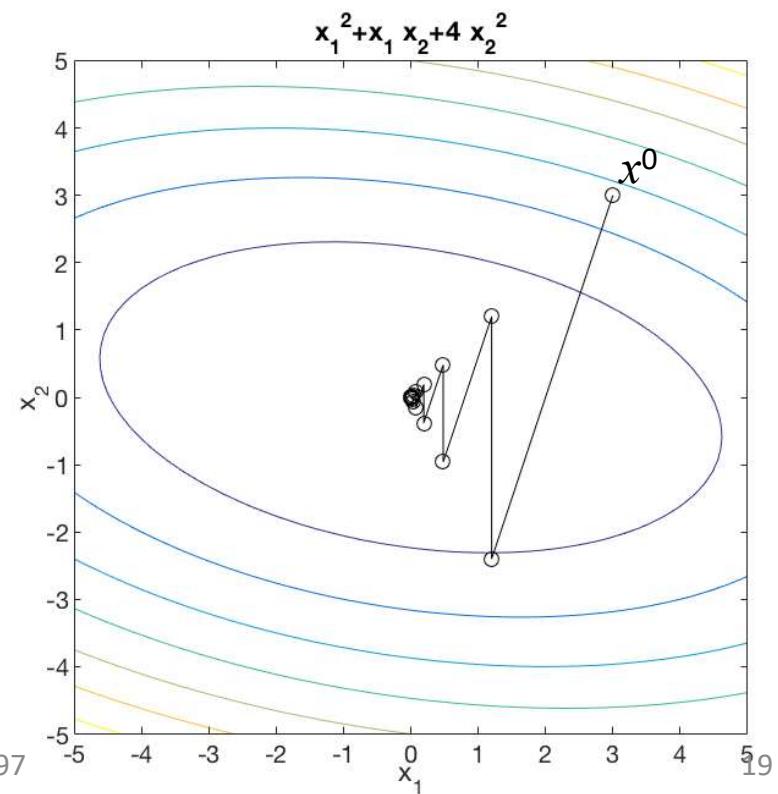
$$f(x_1, x_2) = (x_1)^2 + x_1 x_2 + 4(x_2)^2$$

$$x^{initial} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

$$\eta = 0.1$$



$$\eta = 0.2$$



# What is the optimal step size?

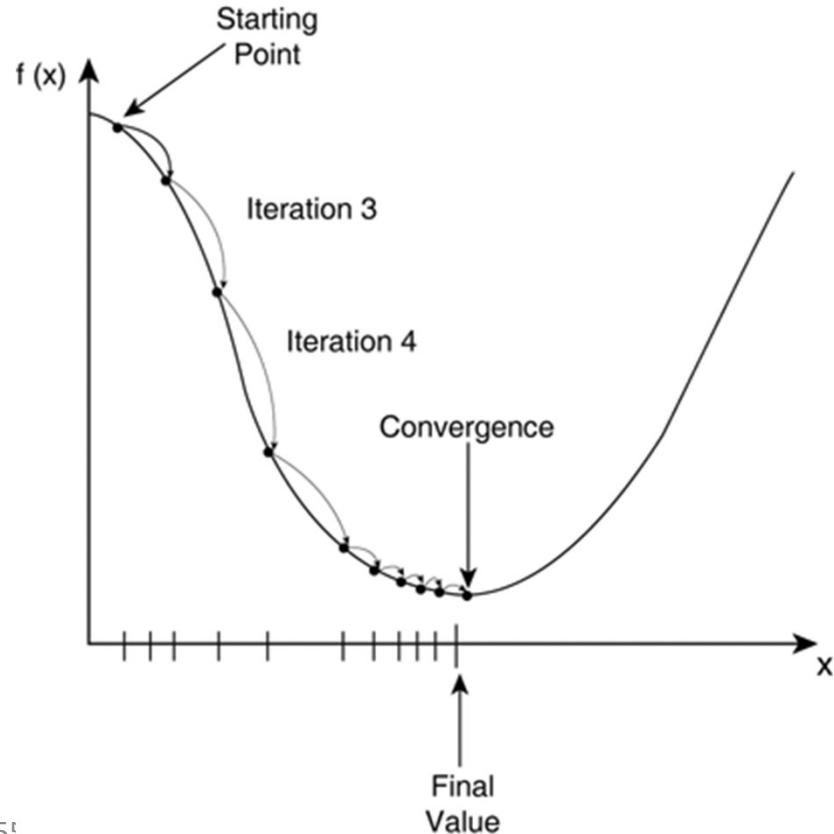
- Step size is critical for fast optimization
- Will revisit this topic later
- For now, simply assume a potentially-iteration-dependent step size

# Gradient descent convergence criteria

- The gradient descent algorithm converges when one of the following criteria is satisfied

$$|f(x^{k+1}) - f(x^k)| < \varepsilon_1$$

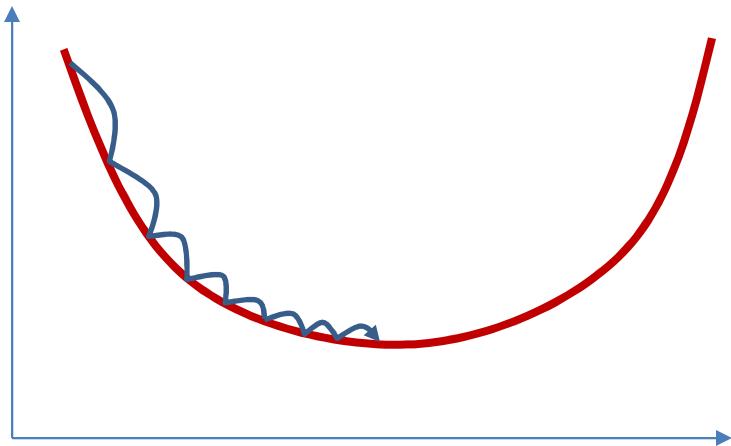
- Or  $\|\nabla f(x^k)\| < \varepsilon_2$



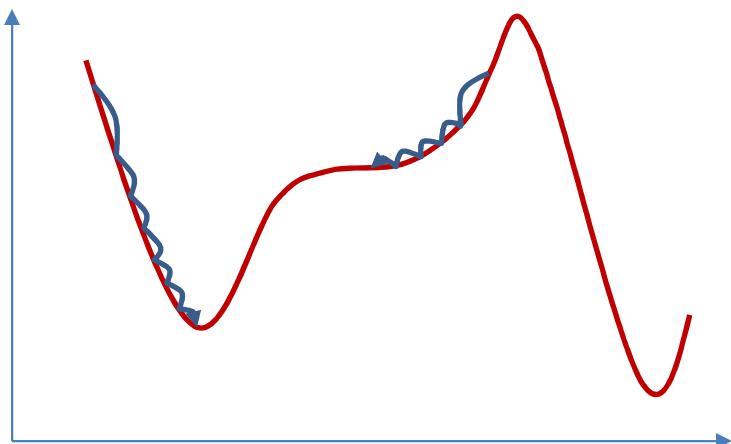
# Overall Gradient Descent Algorithm

- Initialize:
  - $x^0$
  - $k = 0$
- While  $|f(x^{k+1}) - f(x^k)| > \varepsilon$ 
  - $x^{k+1} = x^k - \eta^k \nabla f(x^k)^T$
  - $k = k + 1$

# 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

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

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?

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

# 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?

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

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

# 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?

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

What is the divergence  $\text{div}()$ ?

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

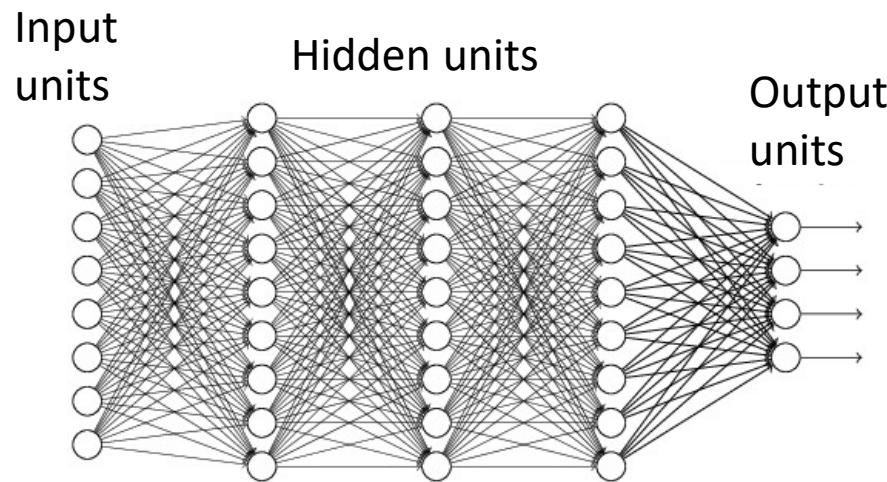
# Problem Setup: Things to define

- 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

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

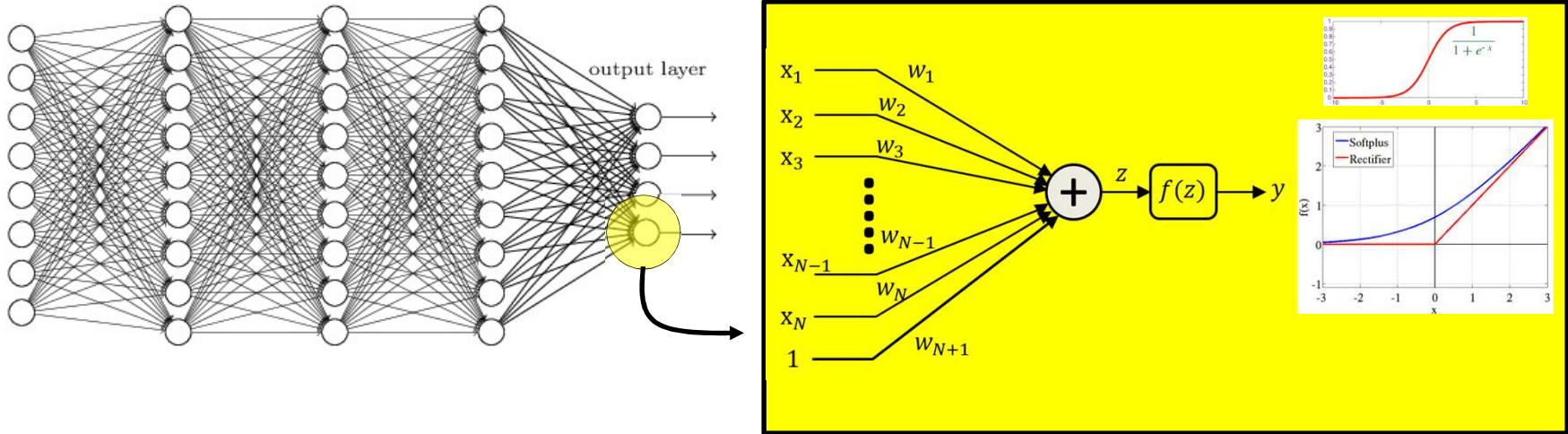
What is  $f()$  and  
what are its  
parameters  $W$ ?

# What is $f()$ ? Typical network



- Multi-layer perceptron
- A *directed* network with a set of inputs and outputs
  - No loops
- Generic terminology
  - We will refer to the inputs as the *input units*
    - **No neurons here – the “input units” are just the inputs**
  - We refer to the outputs as the output units
  - Intermediate units are “hidden” units

# The individual neurons



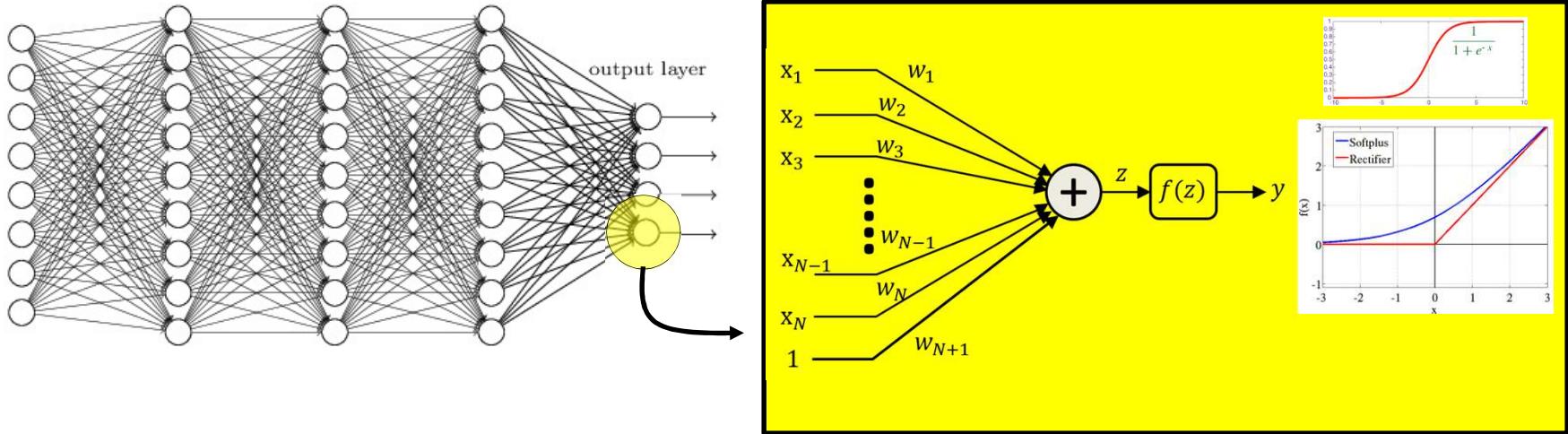
- Individual neurons operate on a set of inputs and produce a single output
  - **Standard setup:** A differentiable activation function applied the sum of weighted inputs and a bias

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

- 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 differentiable activation function applied the sum of weighted inputs and a bias

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

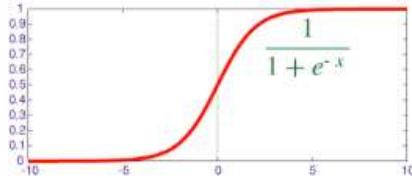
- More generally: *any* differentiable function

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

We will assume this unless otherwise specified

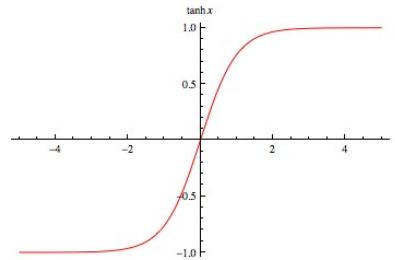
Parameters are weights  $w_i$  and bias  $b$

# Activations and their derivatives



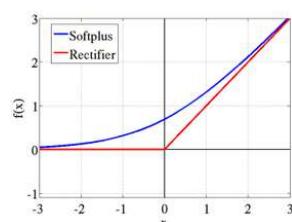
$$f(z) = \frac{1}{1 + \exp(-z)}$$

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



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

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



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

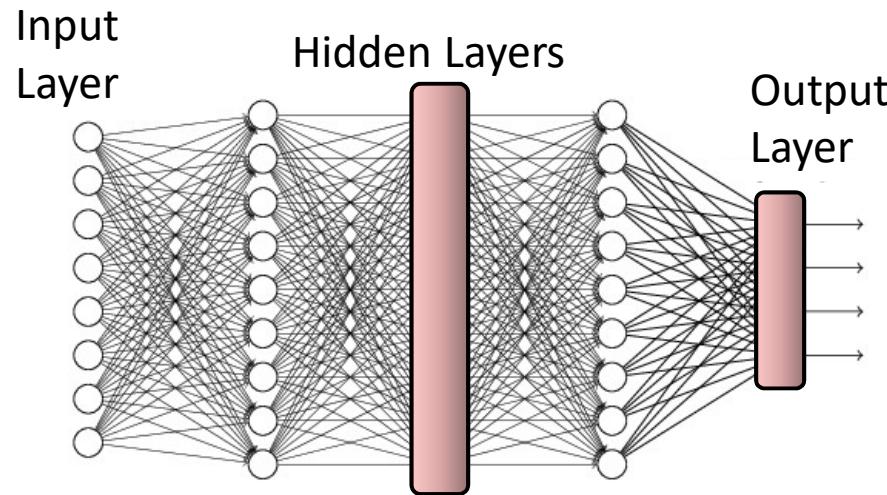
[\*]  $f'(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$

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

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

- Some popular activation functions and their derivatives

# Vector Activations

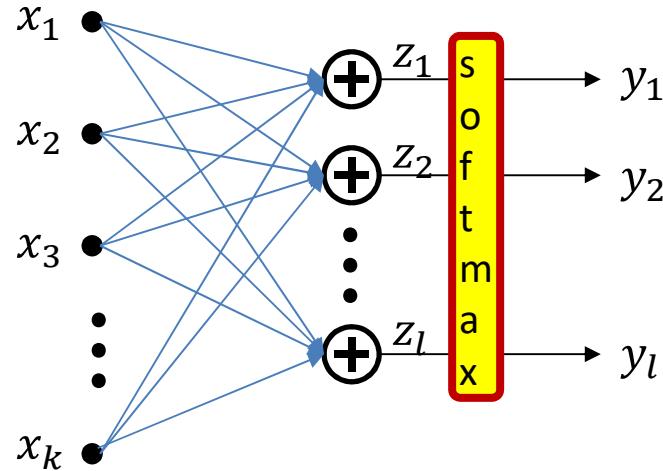


- We can also have neurons that have *multiple coupled* outputs

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

- 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



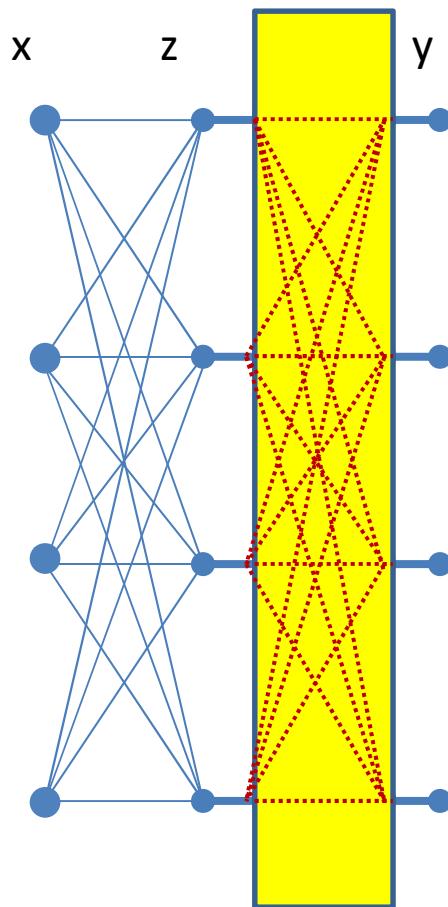
- Example: Softmax *vector* activation

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

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

Parameters are  
weights  $w_{ji}$   
and bias  $b_i$

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



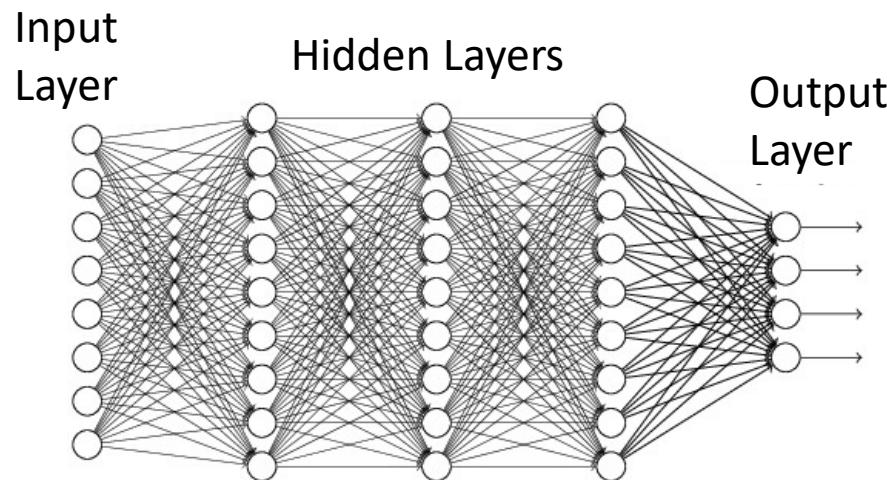
$$z_i = \sum_j w_{ji} x_j + b_i$$

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

Parameters are  
weights  $w_{ji}$   
and bias  $b_i$

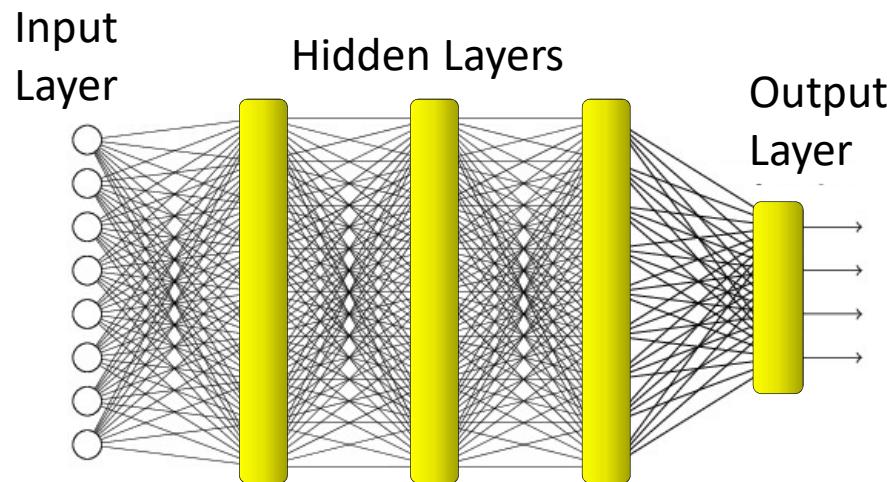
- A layer of multiplicative combination is a special case of vector activation

# Typical network



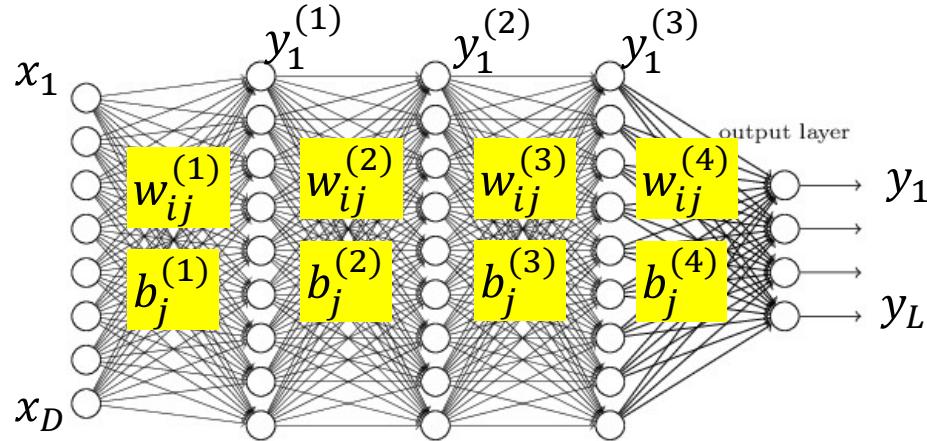
- We assume a “layered” network for simplicity
  - We will refer to the inputs as the *input layer*
    - No neurons here – the “layer” simply refers to inputs
  - We refer to the outputs as the *output layer*
  - Intermediate layers are “hidden” layers

# Typical network



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

# Notation



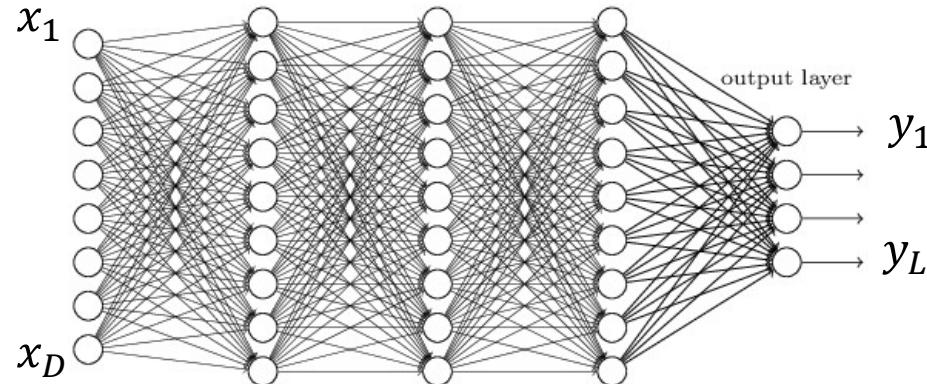
- The input layer is the  $0^{\text{th}}$  layer
- We will represent the output of the  $i$ -th perceptron of the  $k^{\text{th}}$  layer as  $y_i^{(k)}$ 
  - **Input to network:**  $y_i^{(0)} = x_i$
  - **Output of network:**  $y_i = y_i^{(N)}$
- We will represent the weight of the connection between 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  
 $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- What are these input-output pairs?

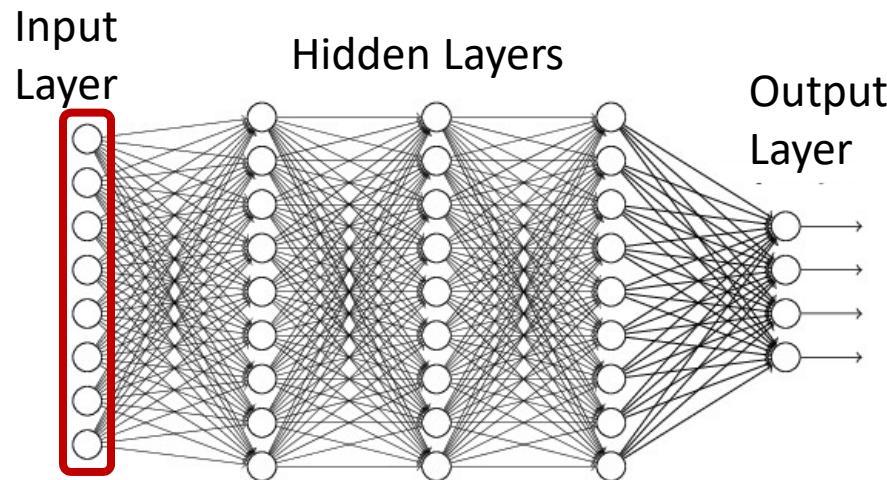
$$Err(W) = \frac{1}{T} \sum_i div(f(X_i; W), d_i)$$

# Vector notation



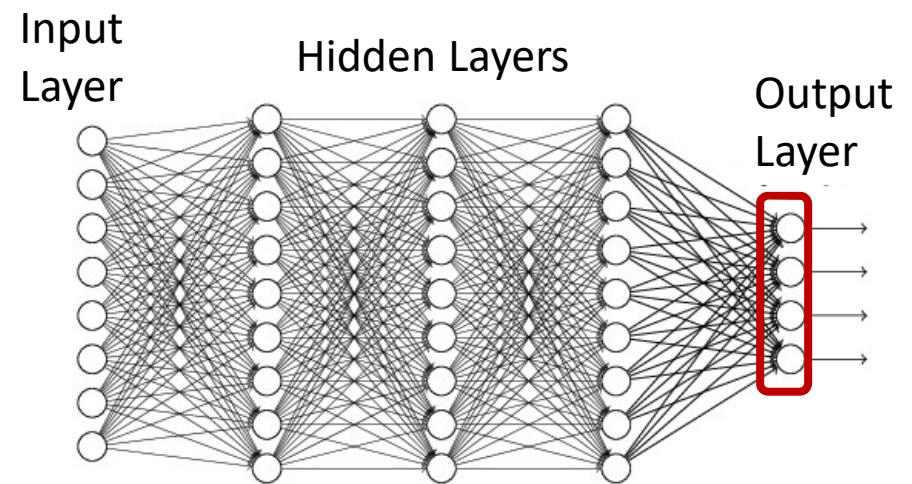
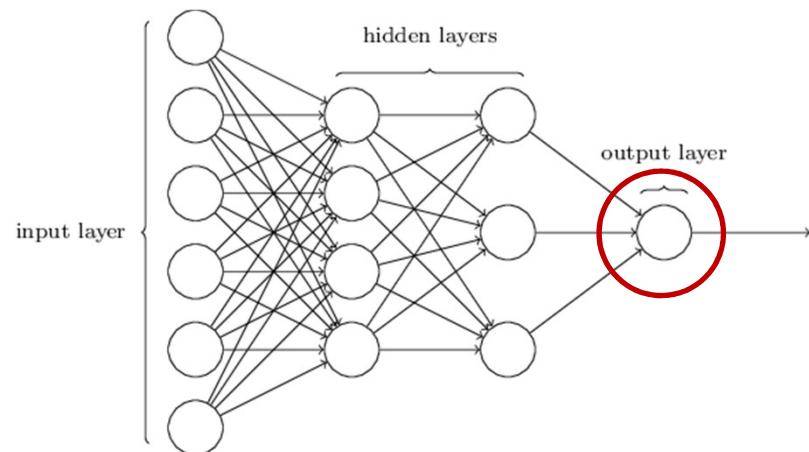
- Given a training set of input-output pairs  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- $X_n = [x_{n1}, x_{n2}, \dots, x_{nD}]$  is the nth input vector
- $d_n = [d_{n1}, d_{n2}, \dots, d_{nL}]$  is the nth desired output
- $Y_n = [y_{n1}, y_{n2}, \dots, y_{nL}]$  is the nth vector of *actual* outputs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

# Representing the input



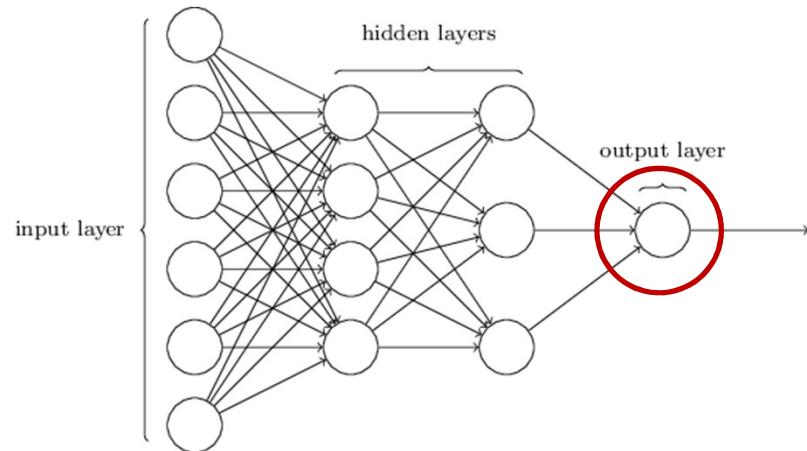
- Vectors of numbers
  - (or may even be just a scalar, if input layer is of size 1)
  - E.g. vector of pixel values
  - E.g. vector of speech features
  - E.g. real-valued vector representing text
    - We will see how this happens later in the course
  - Other real valued vectors

# Representing the output



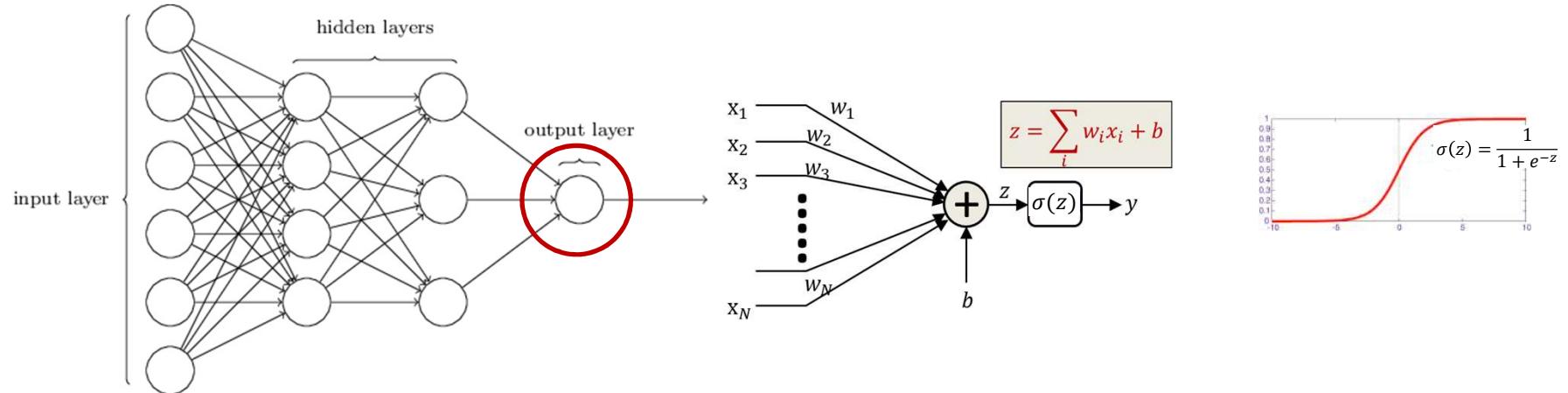
- If the desired *output* is real-valued, no special tricks are necessary
  - Scalar Output : single output neuron
    - $d = \text{scalar} (\text{real value})$
  - Vector Output : as many output neurons as the dimension of the desired output
    - $d = [d_1 \ d_2 \dots \ d_L]$  (vector of real values)

# Representing the output



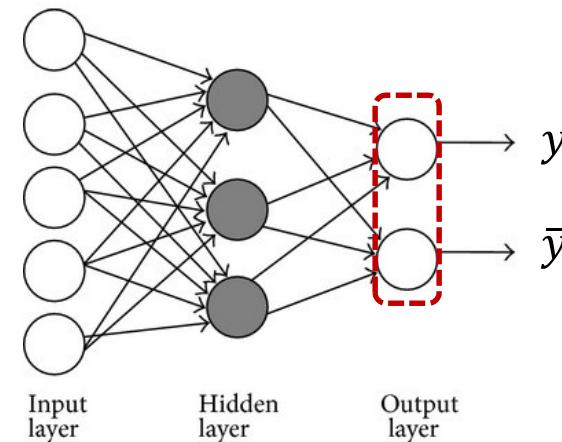
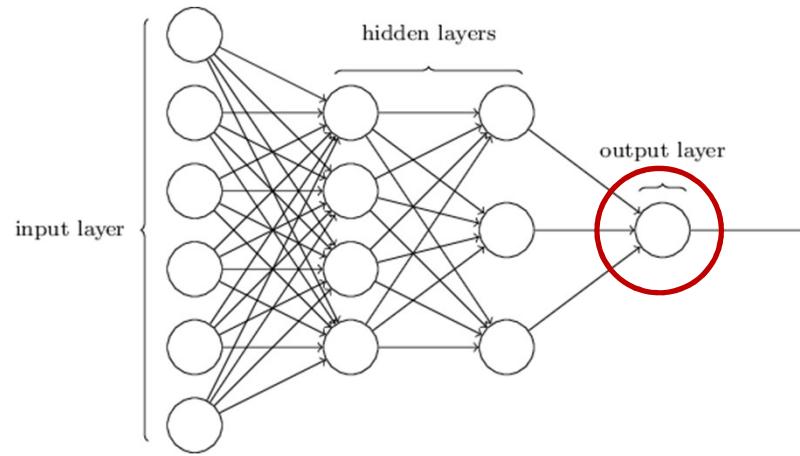
- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
  - 1 = Yes it's a cat
  - 0 = No it's not a cat.

# 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

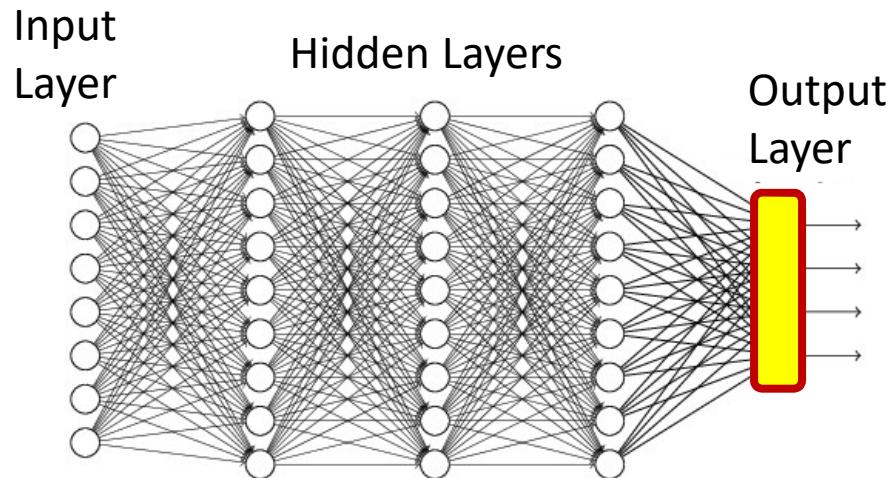


- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
  - 1 = Yes it's a cat
  - 0 = No it's not a cat.
- Sometimes represented by *two independent* outputs, one representing the desired output, the other representing the *negation* of the desired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

# Multi-class output: One-hot representations

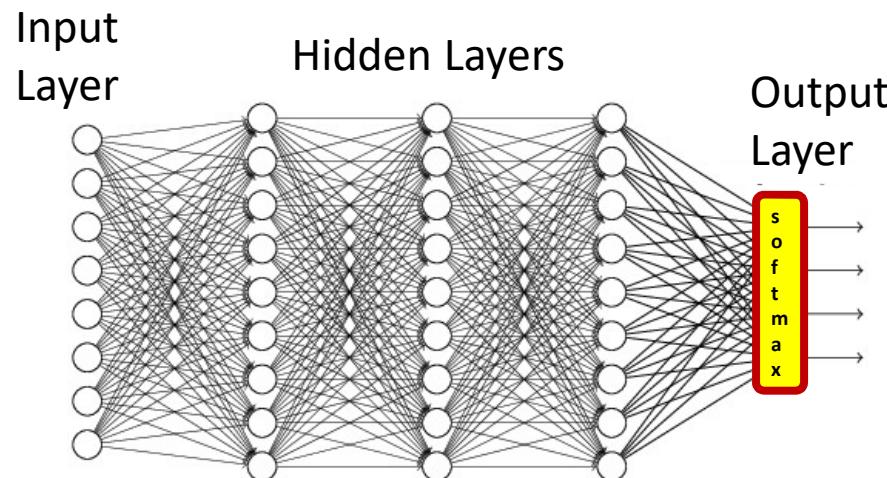
- Consider a network that must distinguish if an input is a cat, a dog, a camel, a hat, or a flower
- We can represent this set as the following vector:
$$[\text{cat } \text{dog } \text{camel } \text{hat } \text{flower}]^T$$
- For inputs of each of the five classes the desired output is:
  - cat:  $[1 \ 0 \ 0 \ 0 \ 0]^T$
  - 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 outputs
  - 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



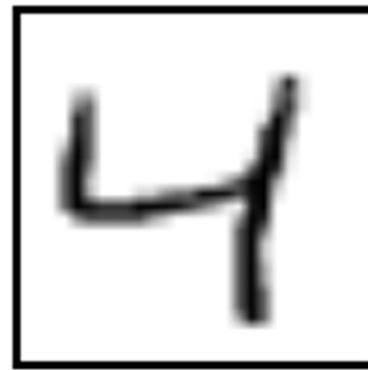
- Softmax *vector* activation is often used at the output of multi-class classifier nets

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

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

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

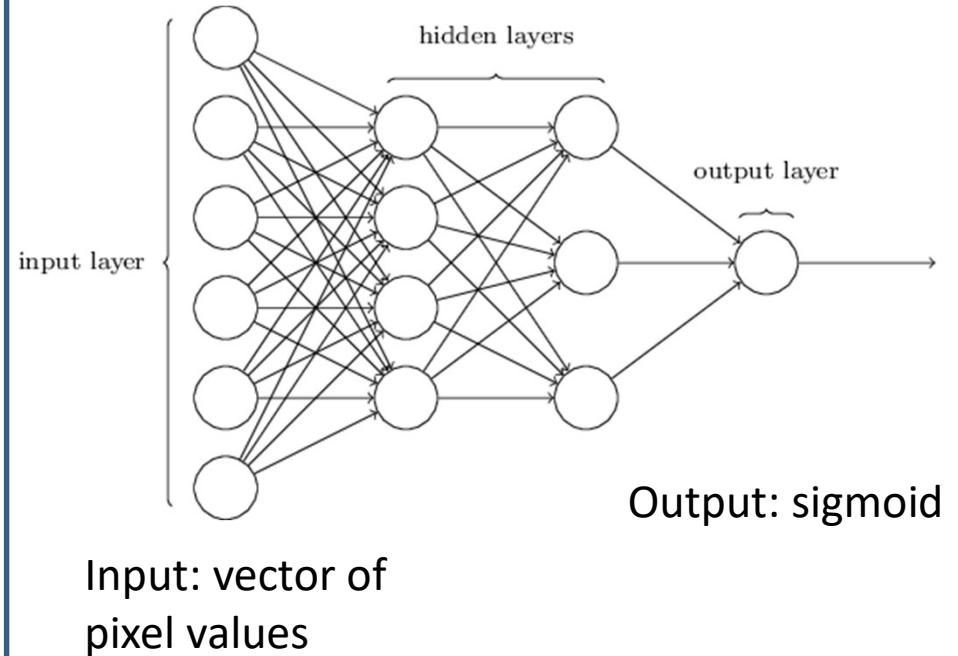
# Typical Problem Statement



- 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? Is this a digit in the first place?

# Typical Problem statement: binary classification

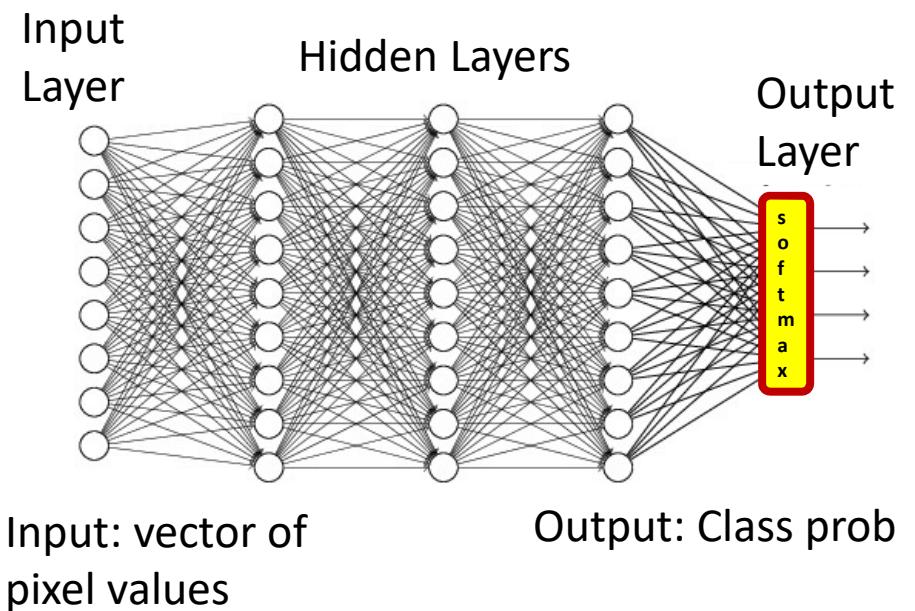
Training data	
(Σ, 0)	(2, 1)
(2, 1)	(4, 0)
(0, 0)	(2, 1)



- 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

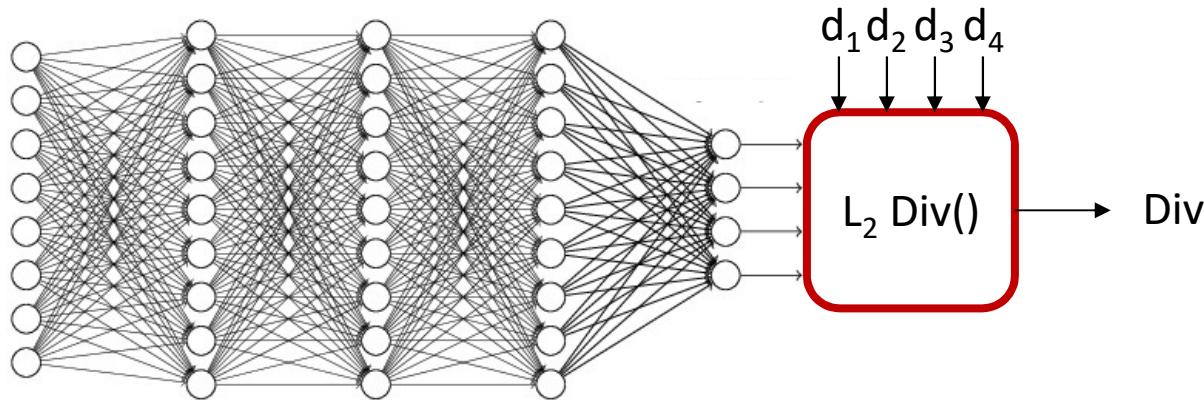
# Problem Setup: Things to define

- 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

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

What is the  
divergence  $\text{div}()$ ?

# Examples of divergence functions



- For real-valued output vectors, the (scaled) L<sub>2</sub> divergence is popular

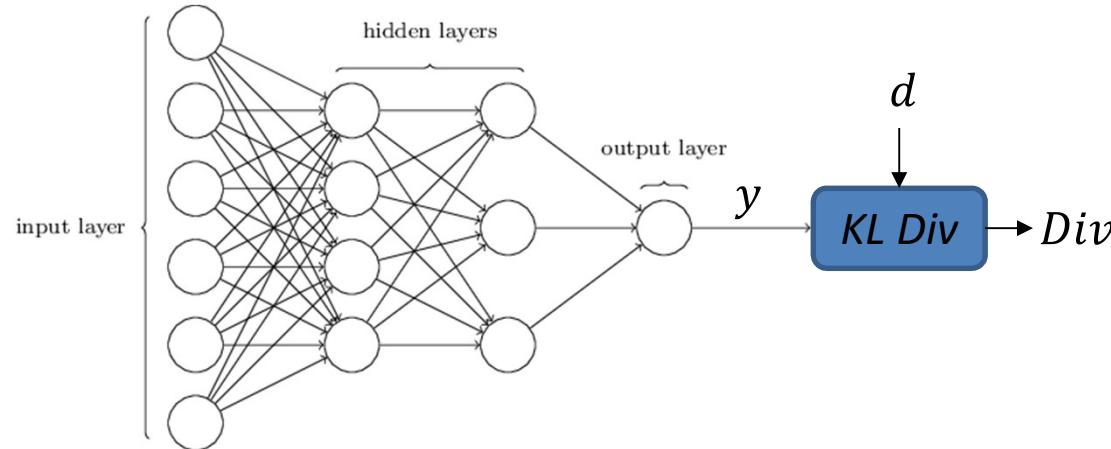
$$Div(Y, d) = \frac{1}{2} \|Y - d\|^2 = \frac{1}{2} \sum_i (y_i - d_i)^2$$

- 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 cross entropy between the probability distribution  $[Y, 1 - Y]$  and the ideal output probability  $[d, 1 - d]$  is popular

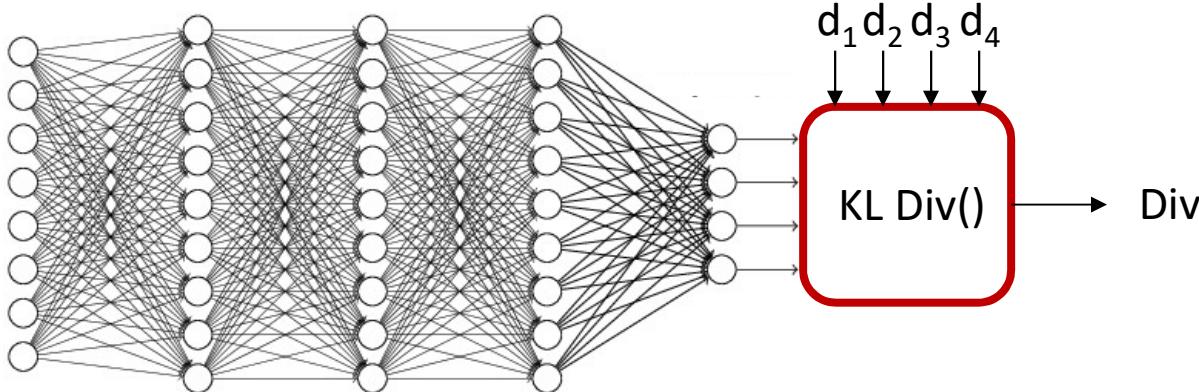
$$Div(Y, d) = -d\log Y - (1 - d)\log(1 - Y)$$

- Minimum when  $d = Y$

- Derivative

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

# 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 cross-entropy between the desired one-hot output and actual output:

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

- Derivative

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

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

# Problem Setup

- Given a training set of input-output pairs  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- The error on the  $i^{\text{th}}$  instance is  $\text{div}(Y_i, d_i)$
- The total error

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

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

# Recap: Gradient Descent Algorithm

- In order to minimize any function  $f(x)$  w.r.t.  $x$
- Initialize:
  - $x^0$
  - $k = 0$
- While  $|f(x^{k+1}) - f(x^k)| > \varepsilon$ 
  - $x^{k+1} = x^k - \eta^k \nabla f(x^k)^T$
  - $k = k + 1$

# Recap: Gradient Descent Algorithm

- In order to minimize any function  $f(x)$  w.r.t.  $x$
- Initialize:
  - $x^0$
  - $k = 0$
- While  $|f(x^{k+1}) - f(x^k)| > \varepsilon$ 
  - For every component  $i$ 
    - $x_i^{k+1} = x_i^k - \eta^k \frac{df}{dx_i}$
  - $k = k + 1$

Explicitly stating it by component

# Training Neural Nets through Gradient Descent

Total training error:

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

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

# Training Neural Nets through Gradient Descent

Total training error:

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

- Gradient descent algorithm:
- Initialize all weights  $\{w_{ij}^{(k)}\}$
- Do:
  - For every layer  $k$  for all  $i, j$ , update:
    - $w_{i,j}^{(k)} = w_{i,j}^{(k)} - \eta \frac{dErr}{dw_{i,j}^{(k)}}$
- Until  $Err$  has converged

# The derivative

Total training error:

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

- Computing the derivative

Total derivative:

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

# Training by gradient descent

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

# The derivative

Total training error:

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

Total derivative:

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

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

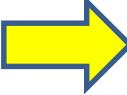
# Calculus Refresher: Basic rules of calculus

For any differentiable function

$$y = f(x)$$

with derivative

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

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

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  $\Delta x_1, \Delta x_2, \dots, \Delta x_M$

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

# Calculus Refresher: Chain rule

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

$$\frac{dy}{dx} = \frac{\partial y}{\partial g(x)} \frac{dg(x)}{dx}$$

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

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

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



# Calculus Refresher: Distributed Chain rule

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

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

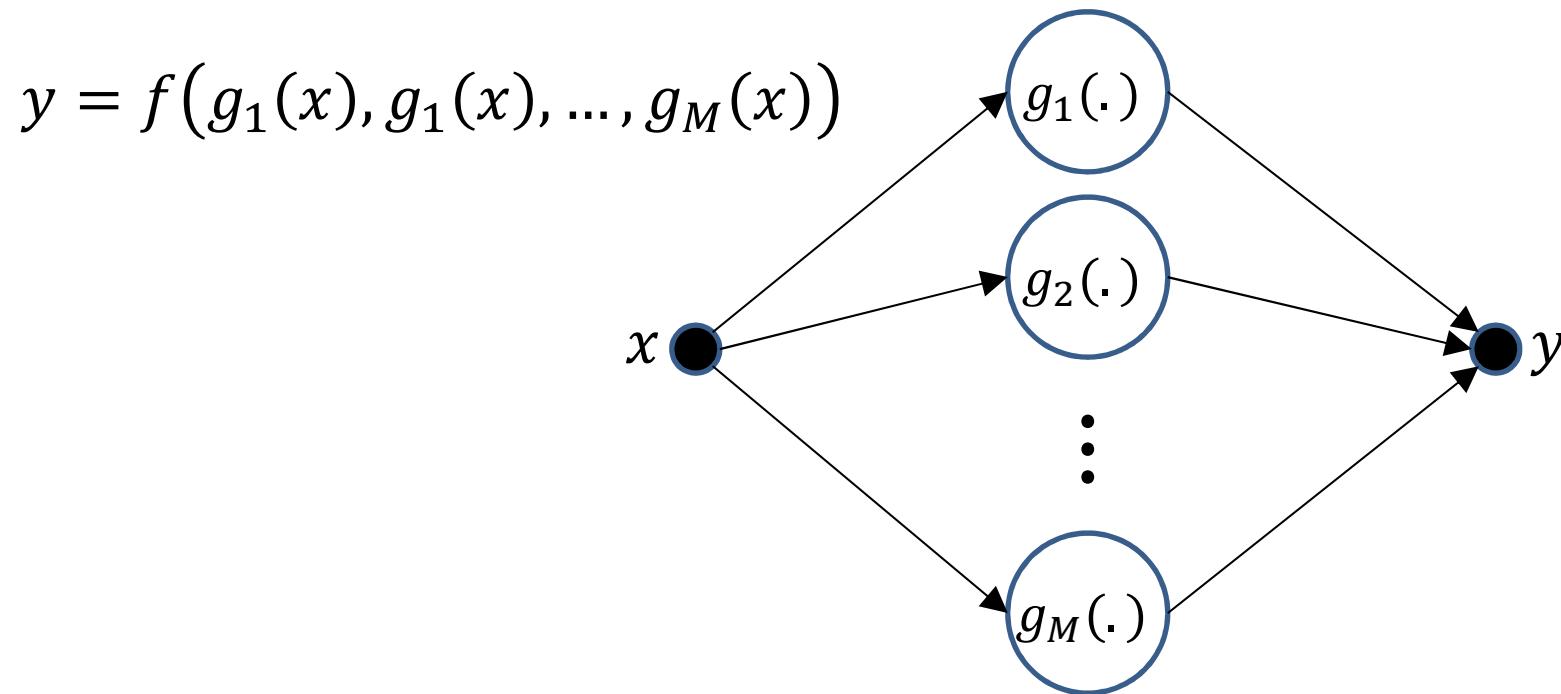
Check:  $\Delta y = \frac{dy}{dx} \Delta x$

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

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

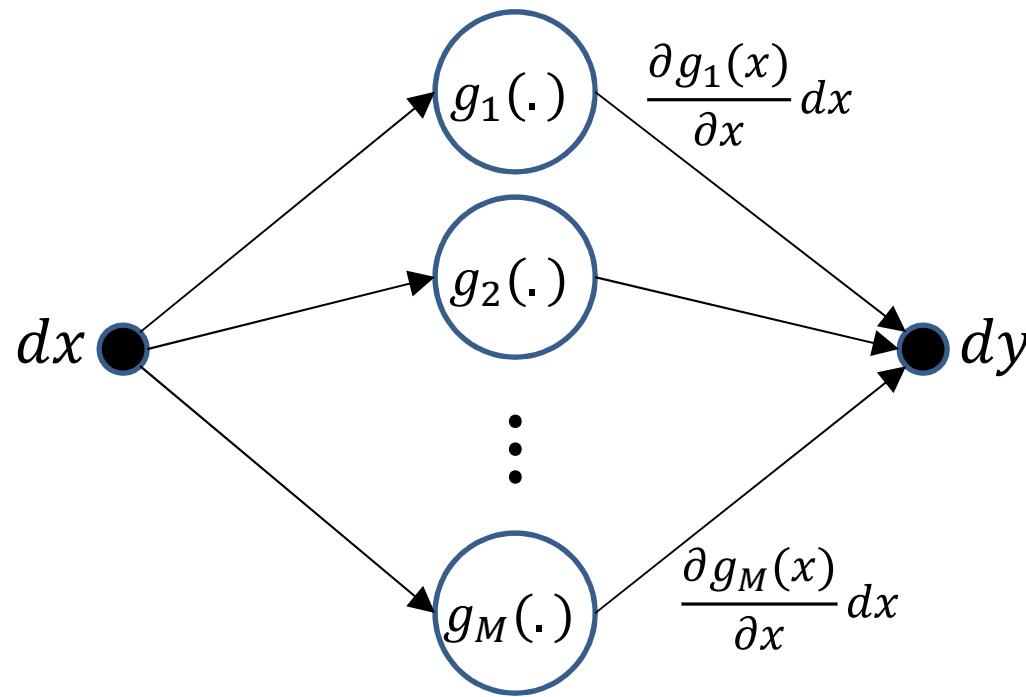
$$\Delta y = \left( \frac{\partial y}{\partial g_1(x)} \frac{dg_1(x)}{dx} + \frac{\partial y}{\partial g_2(x)} \frac{dg_2(x)}{dx} + \dots + \frac{\partial y}{\partial g_M(x)} \frac{dg_M(x)}{dx} \right) \Delta x$$

# Distributed Chain Rule: Influence Diagram



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

# Distributed Chain Rule: Influence Diagram

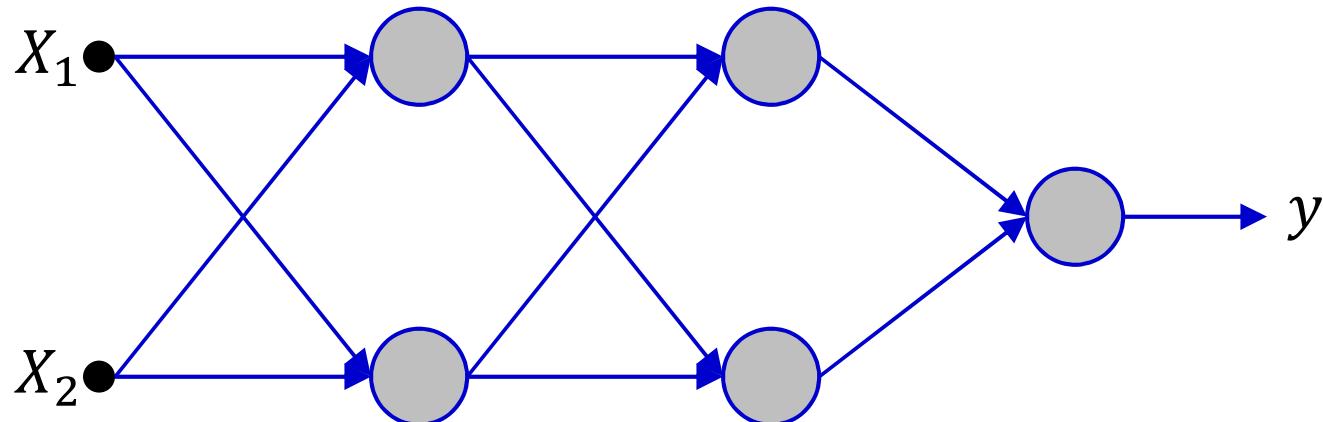


- Small perturbations in  $x$  cause small perturbations in each of  $g_1 \dots g_M$ , each of which individually additively perturbs  $y$

# Returning to our problem

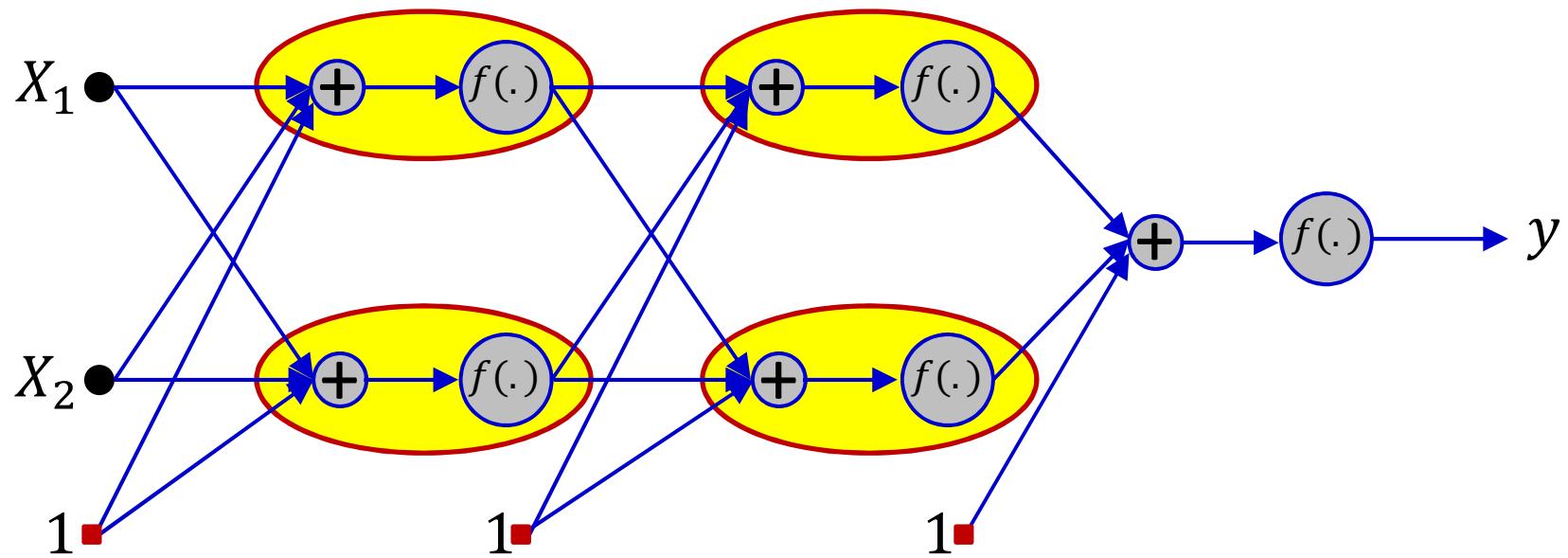
- How to compute  $\frac{d\text{Div}(\mathbf{Y}, \mathbf{d})}{dw_{i,j}^{(k)}}$

# A first closer look at the network



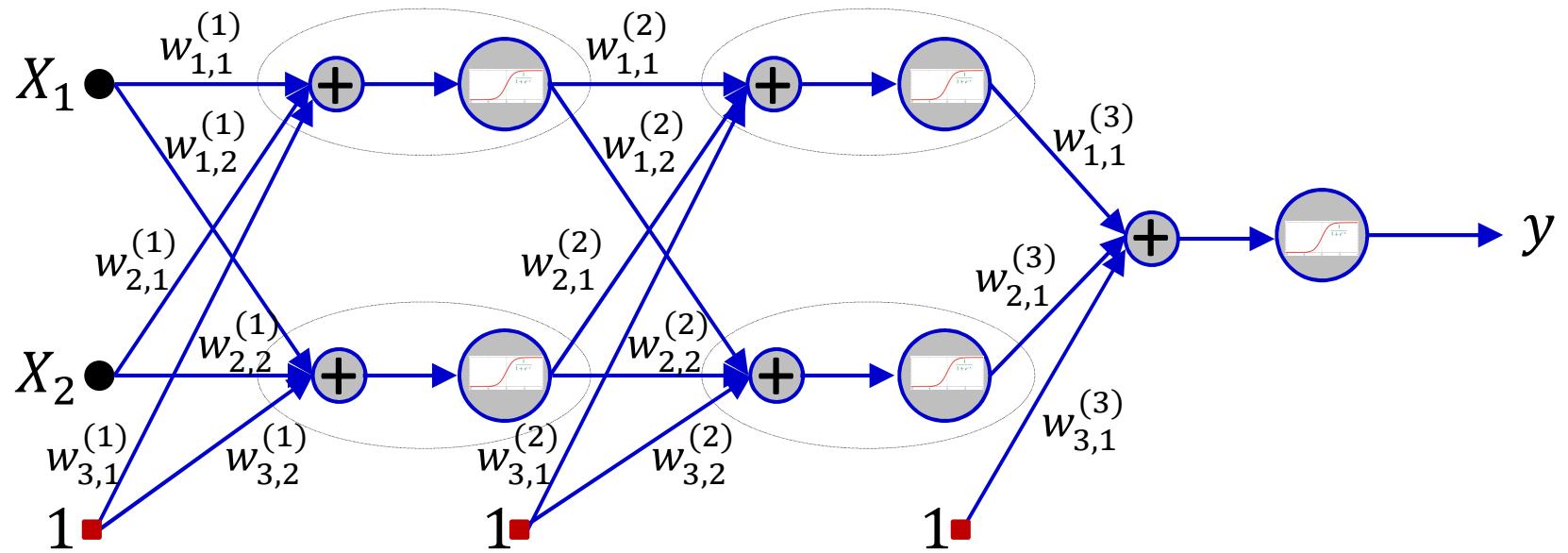
- Showing a tiny 2-input network for illustration
  - Actual network would have many more neurons and inputs

# A first closer look at the network



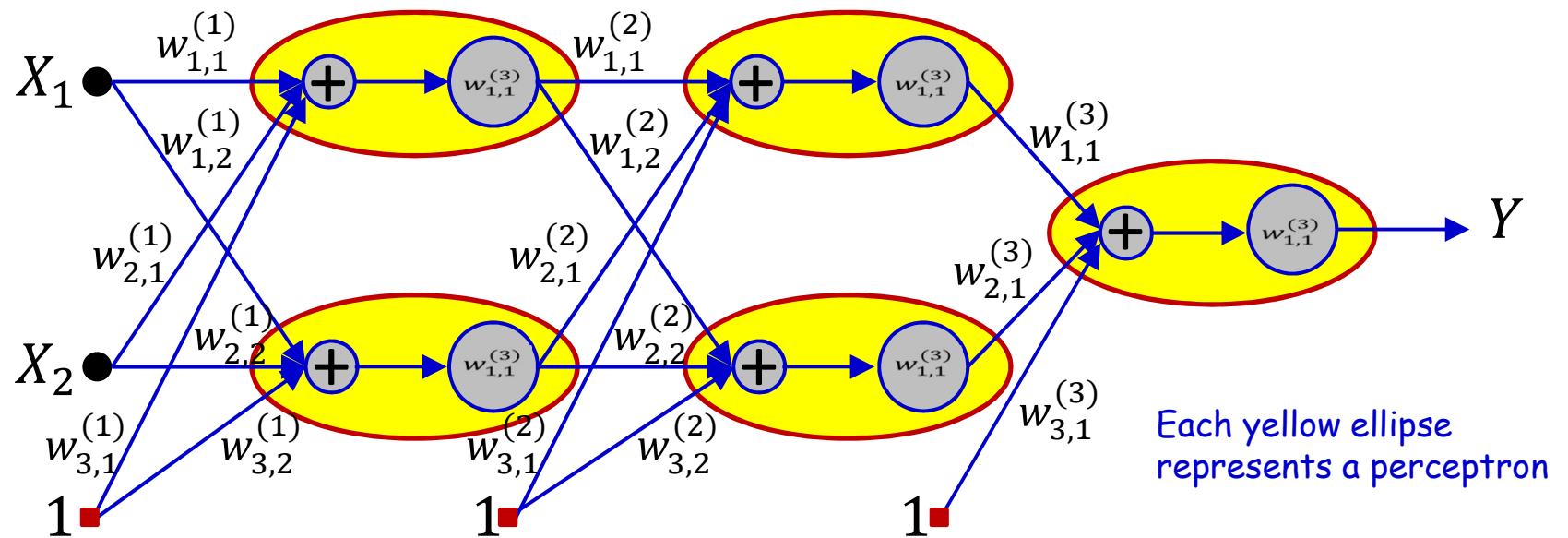
- Showing a tiny 2-input network for illustration
  - Actual network would have many more neurons and inputs
- Explicitly separating the weighted sum of inputs from the activation

# A first closer look at the network



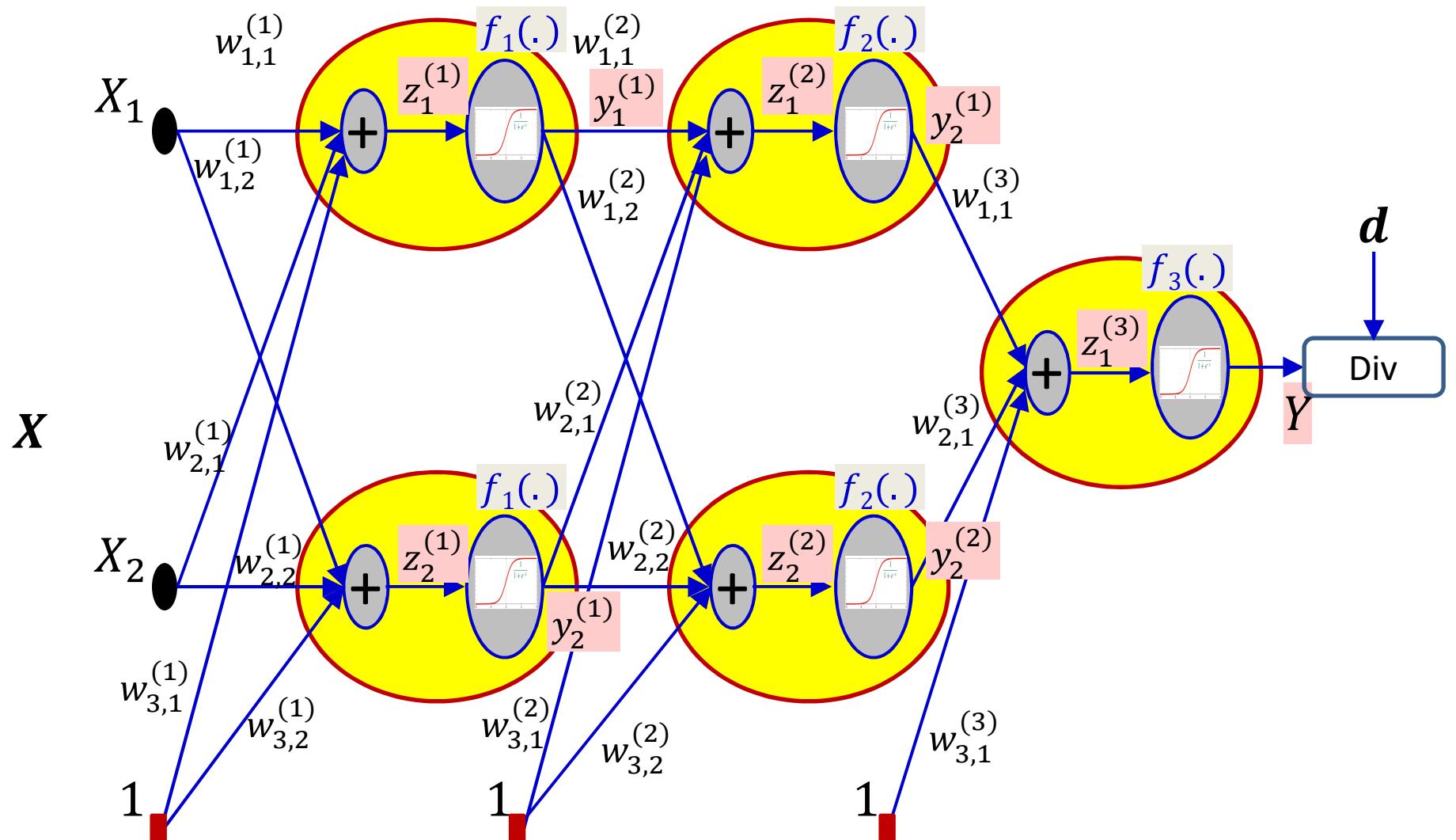
- Showing a tiny 2-input network for illustration
  - Actual network would have many more neurons and inputs
- Expanded **with all weights and activations shown**
- The overall function is differentiable w.r.t every weight, bias and input

# 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 gradient

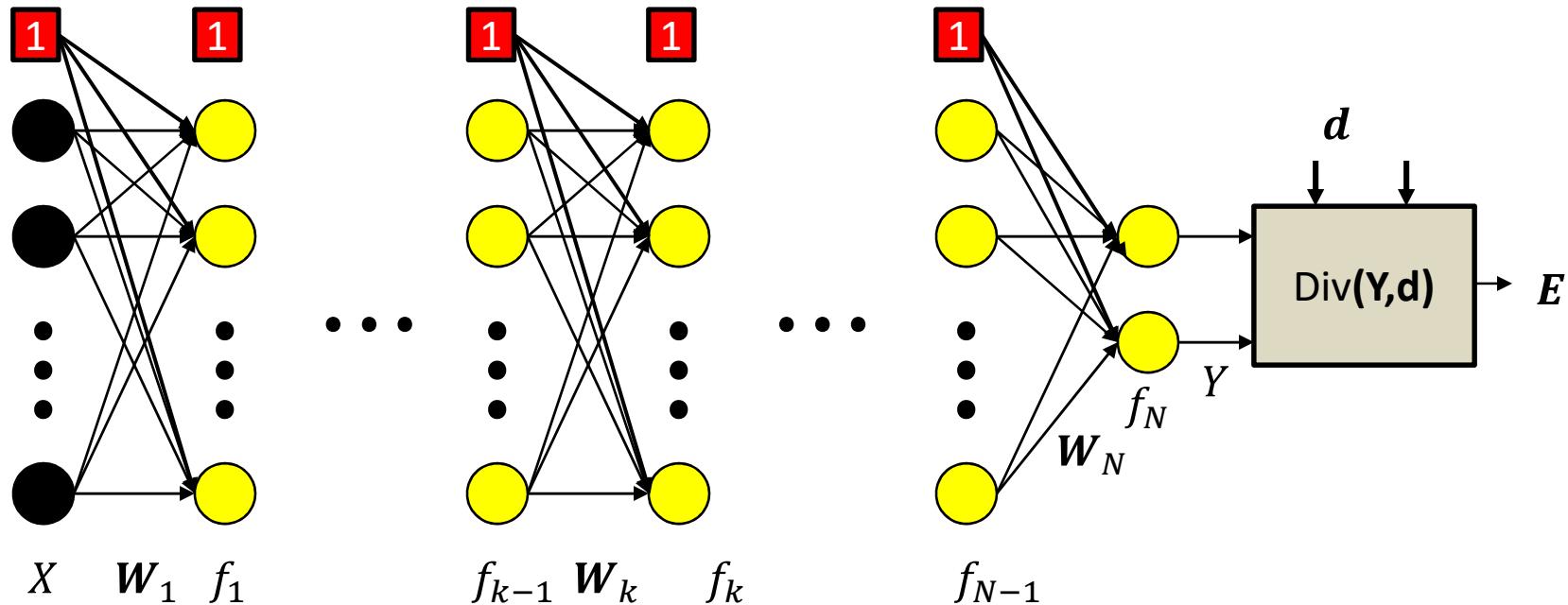
- What is:  $\frac{d\text{Div}(\mathbf{Y}, \mathbf{d})}{dw_{i,j}^{(k)}}$

– Derive on board?

# Computing the gradient

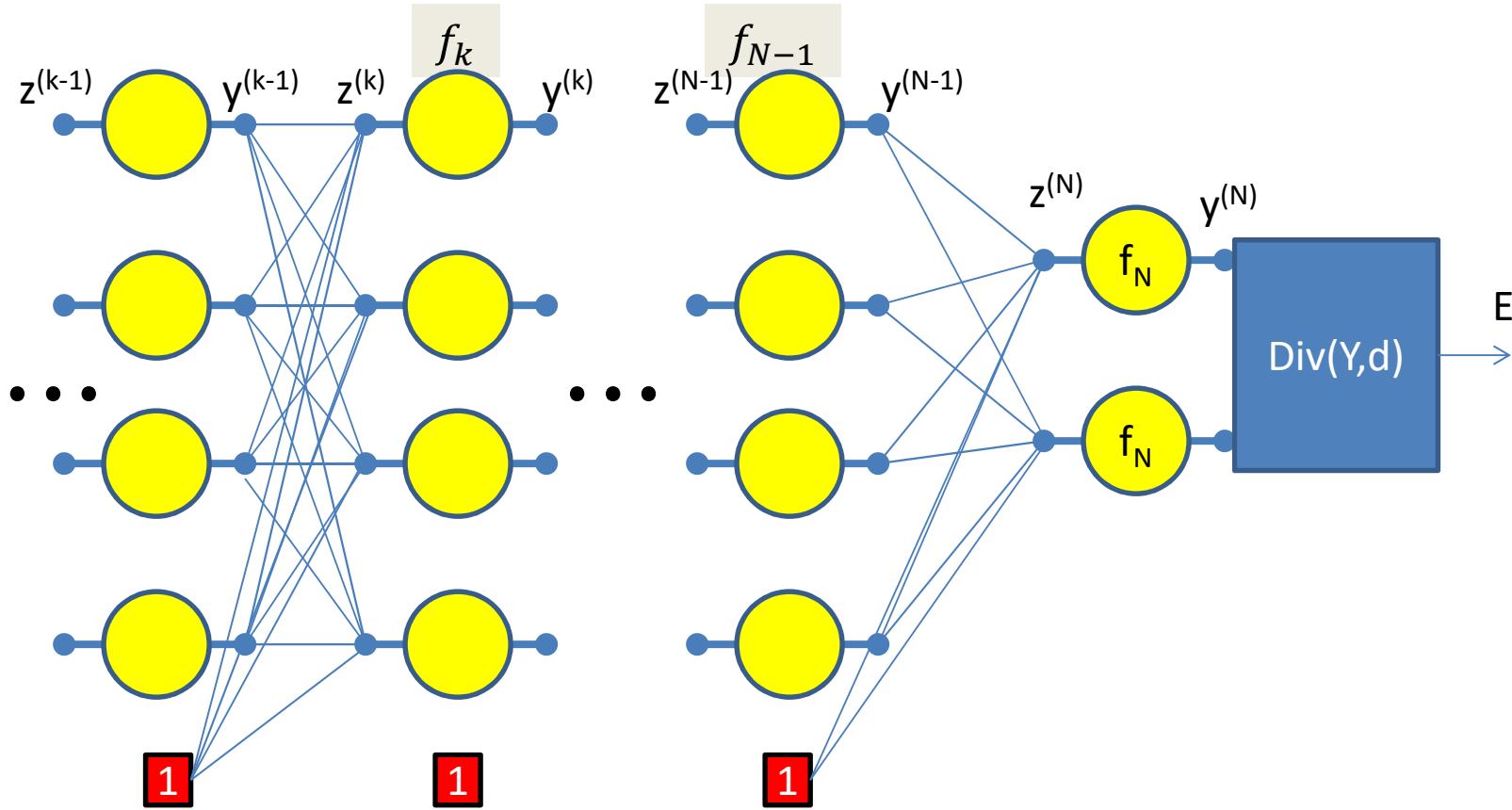
- What is:  $\frac{d\text{Div}(\mathbf{Y}, \mathbf{d})}{dw_{i,j}^{(k)}}$
- Derive on board?
- Note: computation of the derivative requires intermediate and final output values of the network in response to the input

# BP: Scalar Formulation



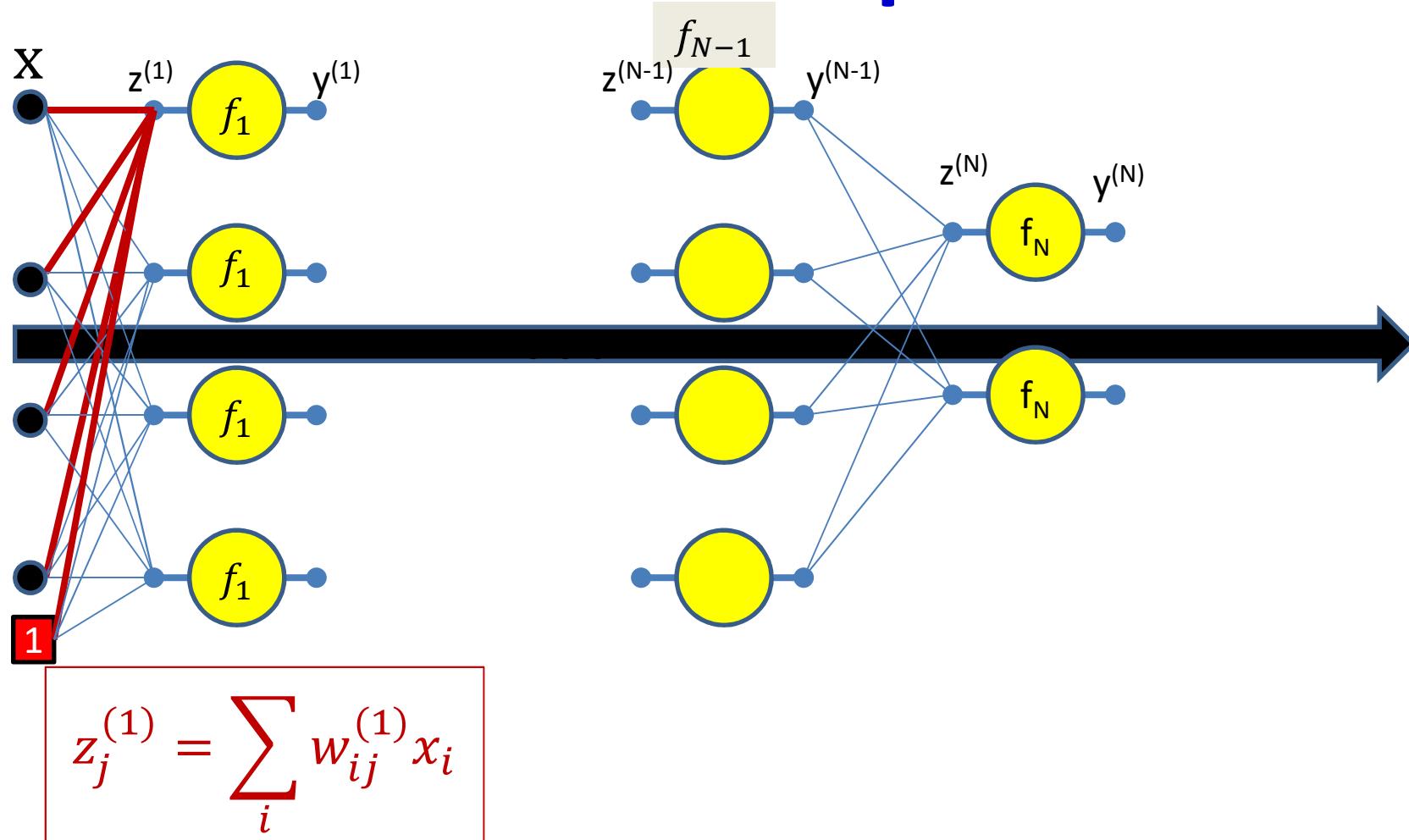
- The network again

# Gradients: Local Computation



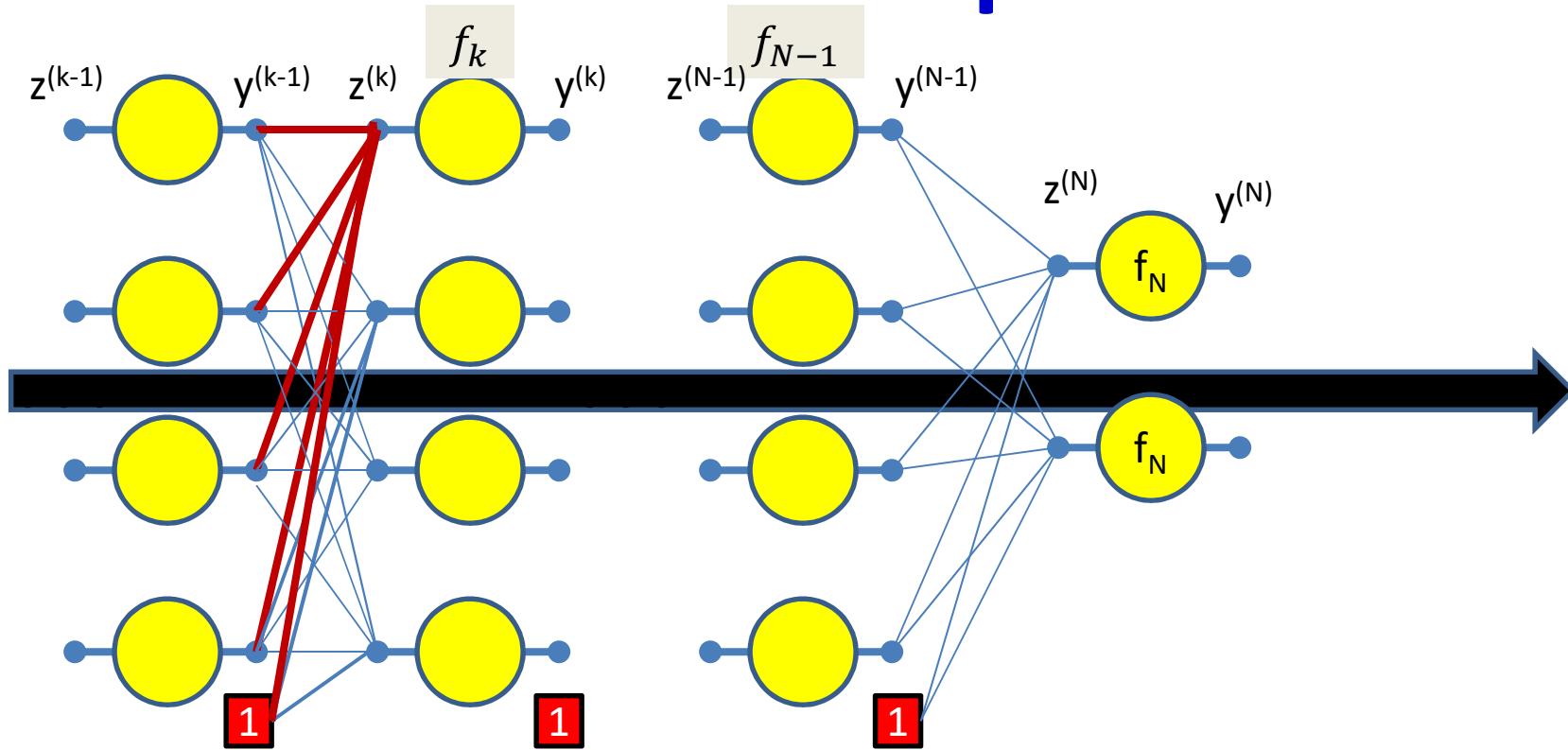
- Redrawn
- Separately label input and output of each node

# Forward Computation



Assuming  $w_{0j}^{(1)} = b_j^{(1)}$  and  $x_0 = 1$

# Forward Computation

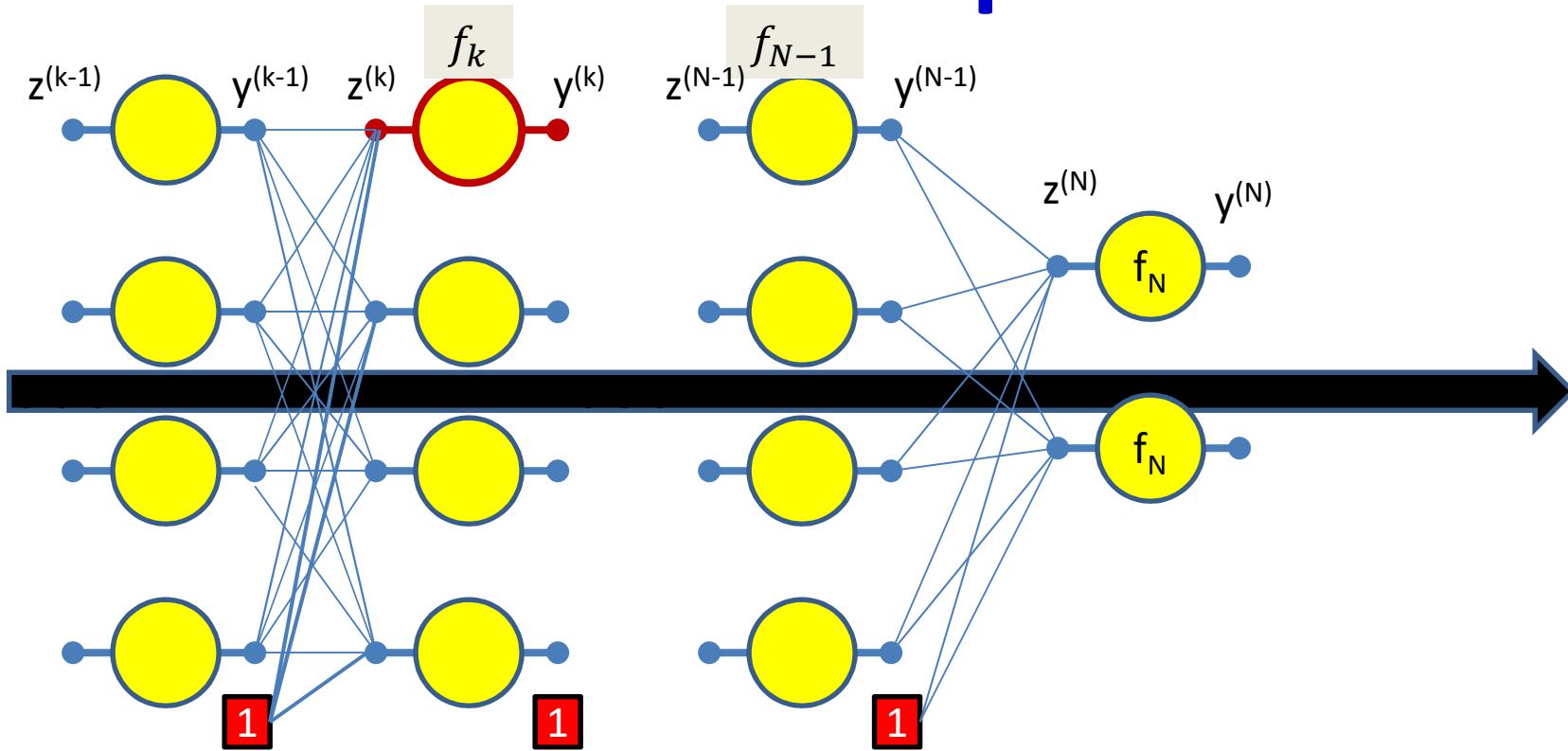


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i$$

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

Assuming  $w_{0j}^{(k)} = b_j^{(k)}$  and  $y_0^{(k-1)} = 1$

# Forward Computation

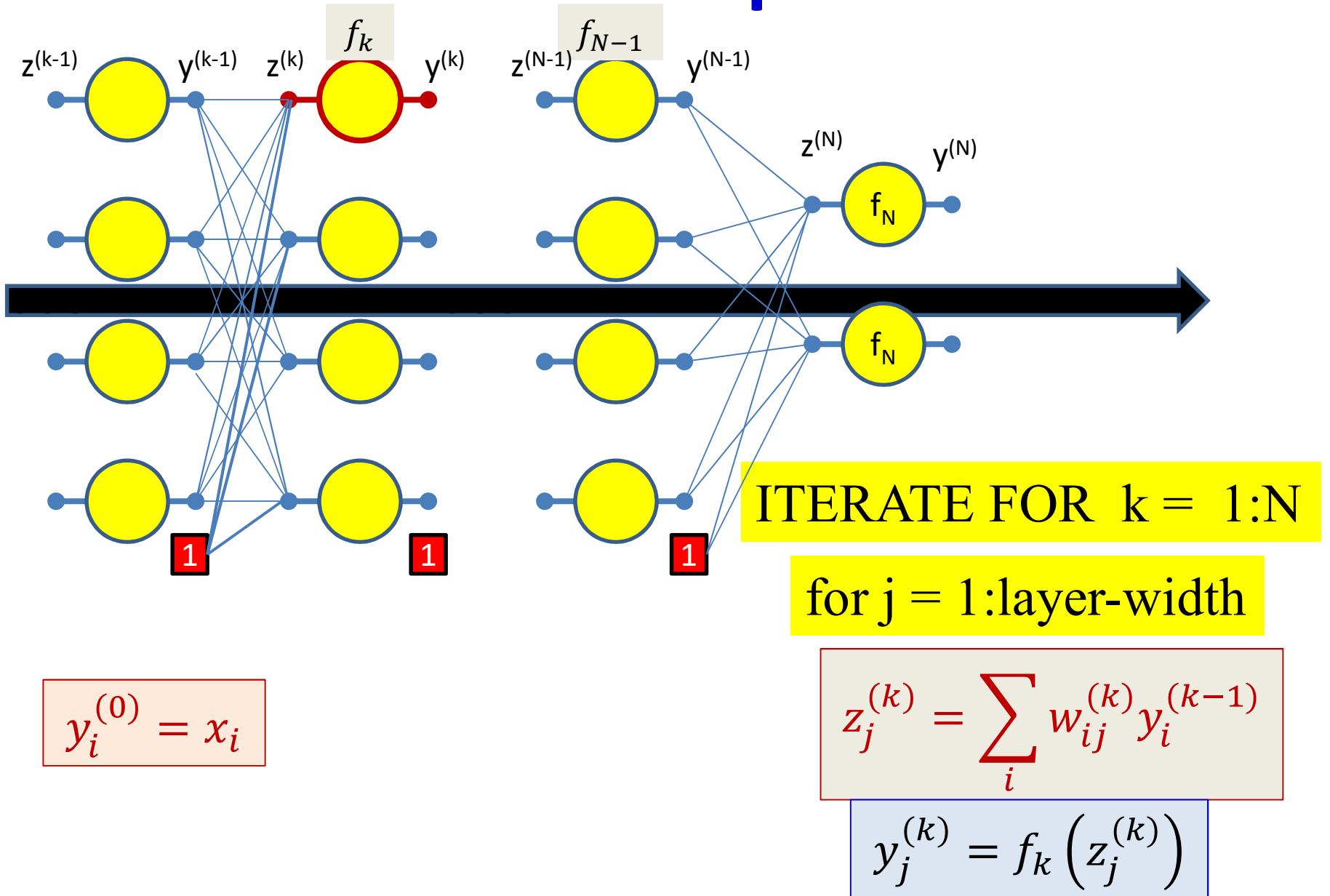


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i$$

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

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

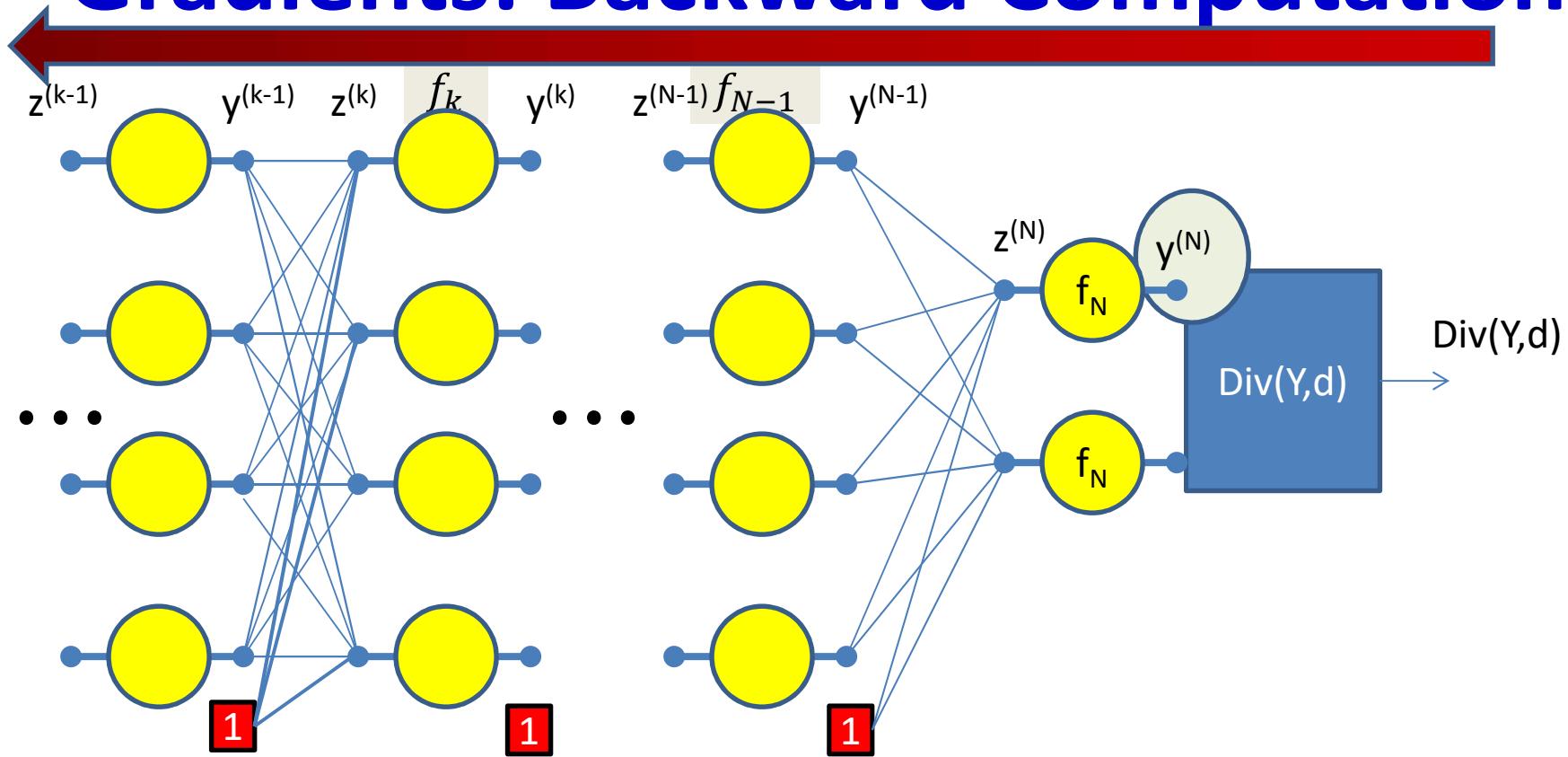
# Forward Computation



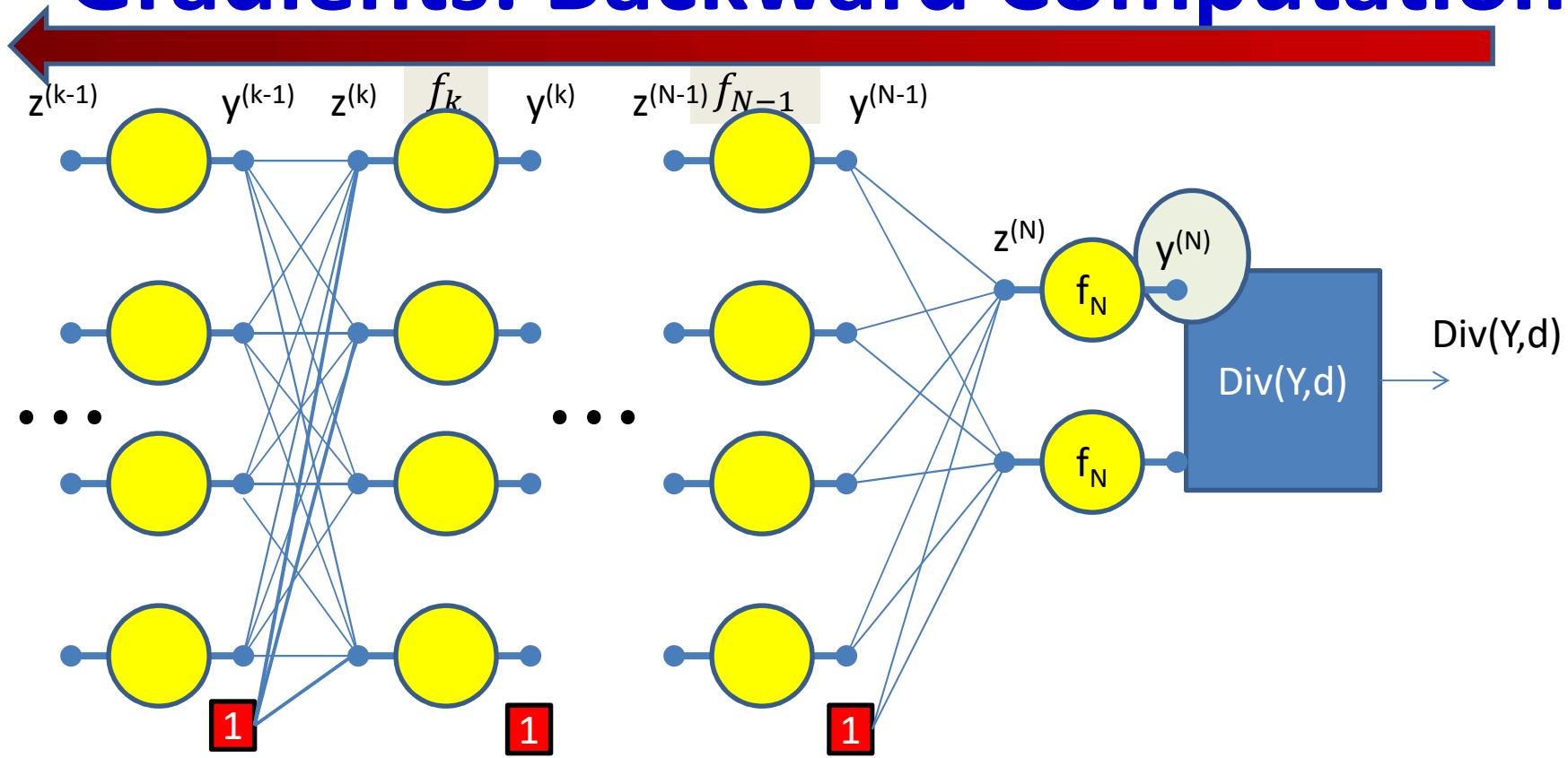
# Forward “Pass”

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

# Gradients: Backward Computation

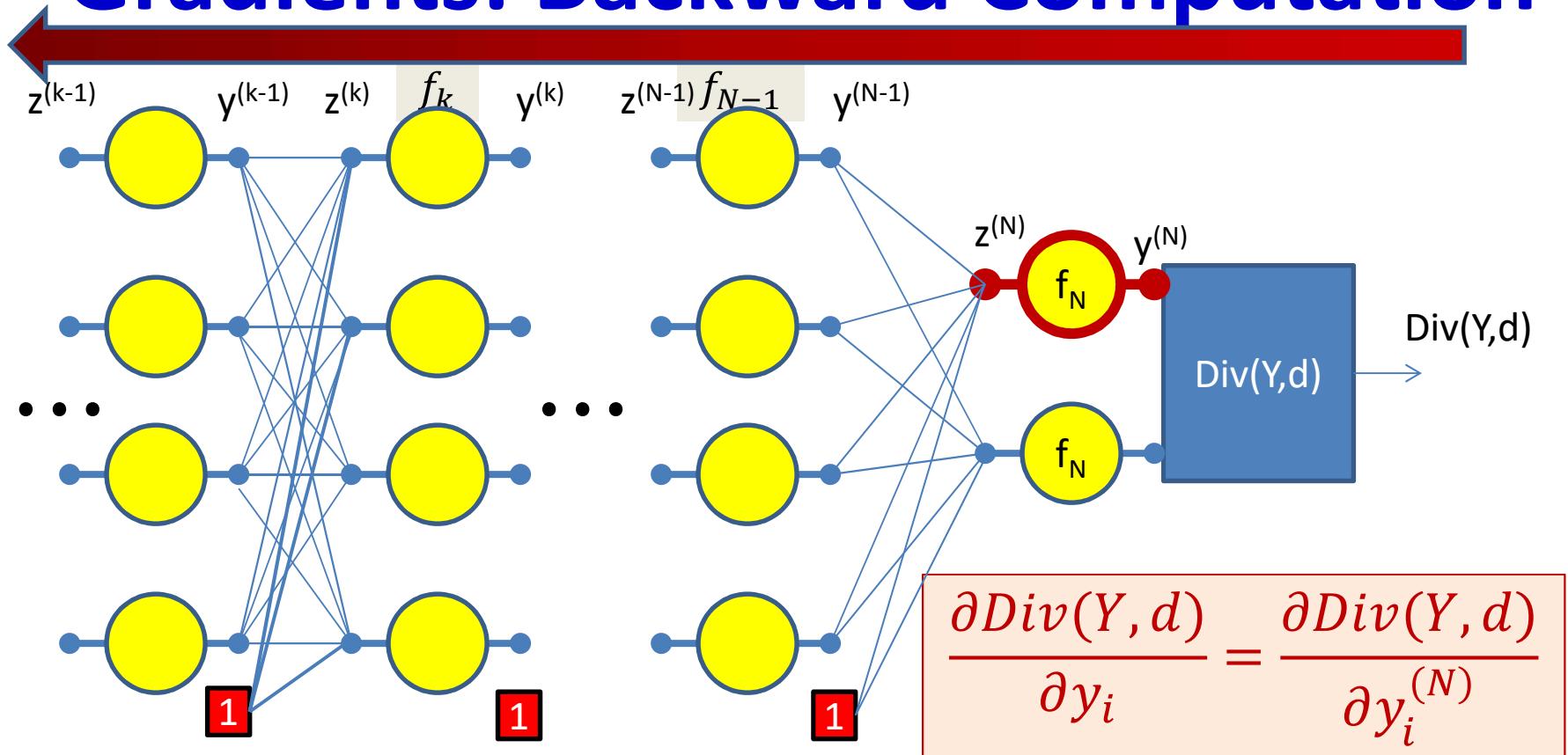


# Gradients: Backward Computation



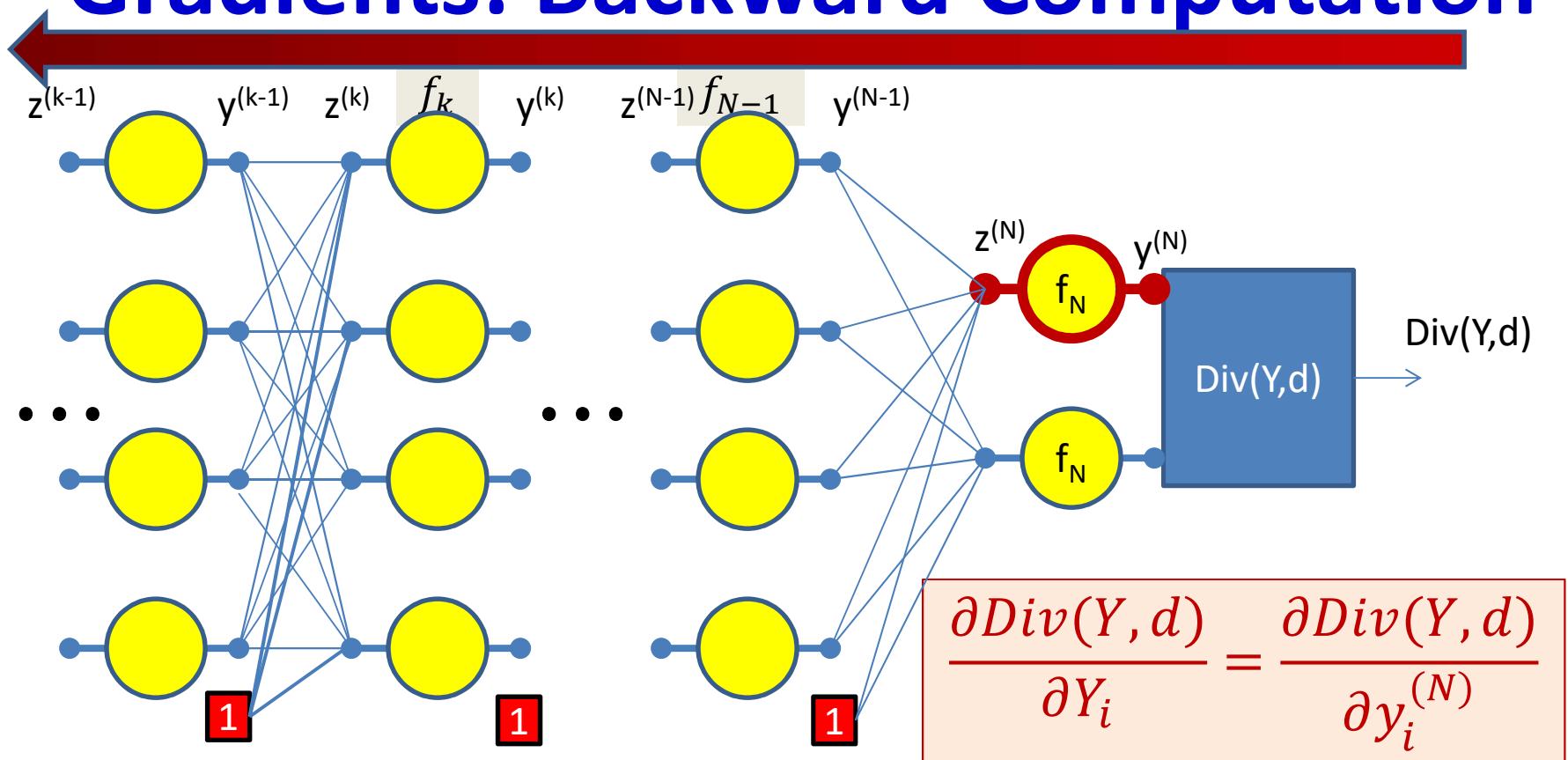
$$\frac{\partial \text{Div}(Y, d)}{\partial y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}}$$

# Gradients: Backward Computation



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

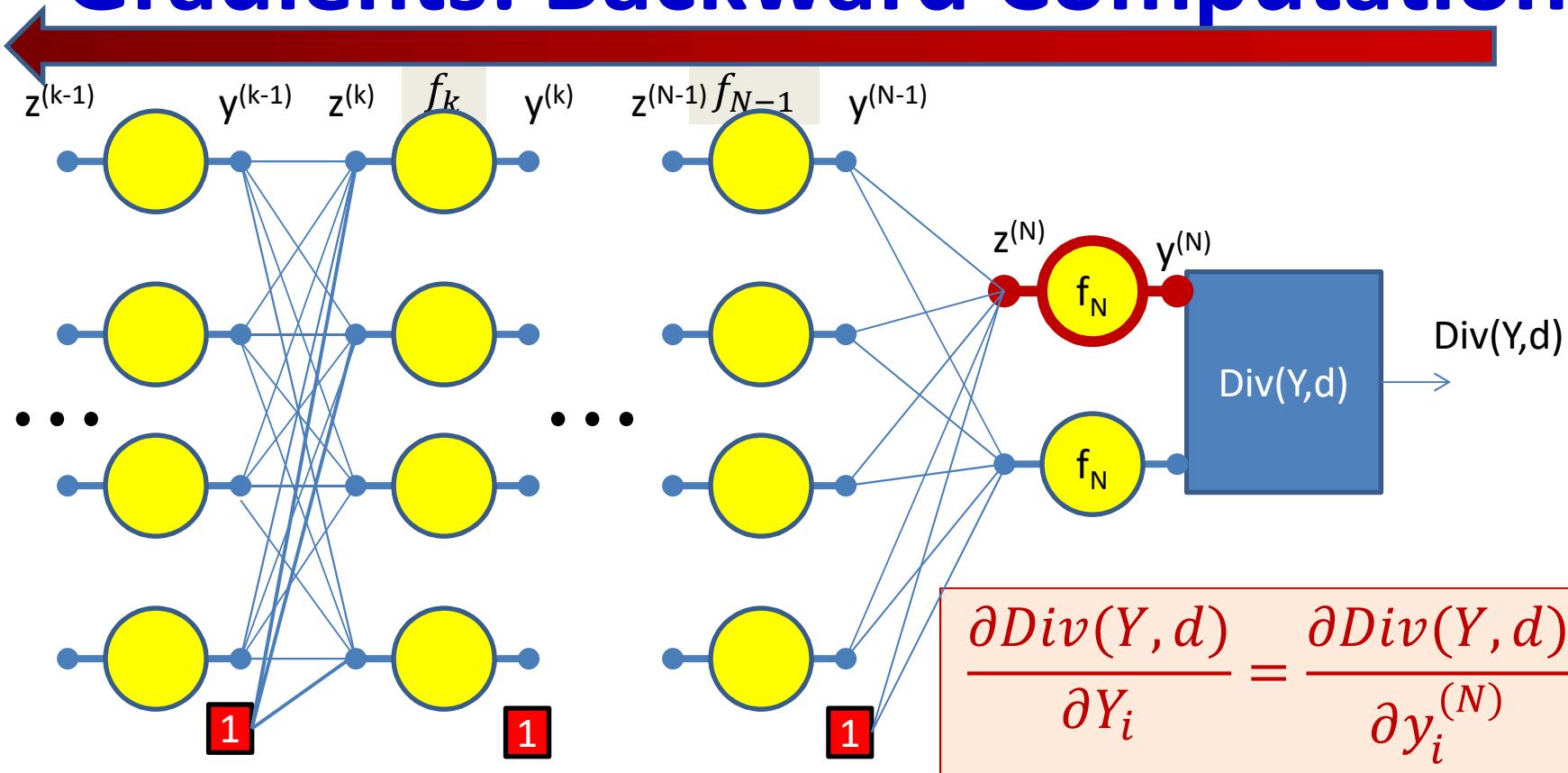
# Gradients: Backward Computation



$z_i^{(N)}$  computed during the forward pass

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

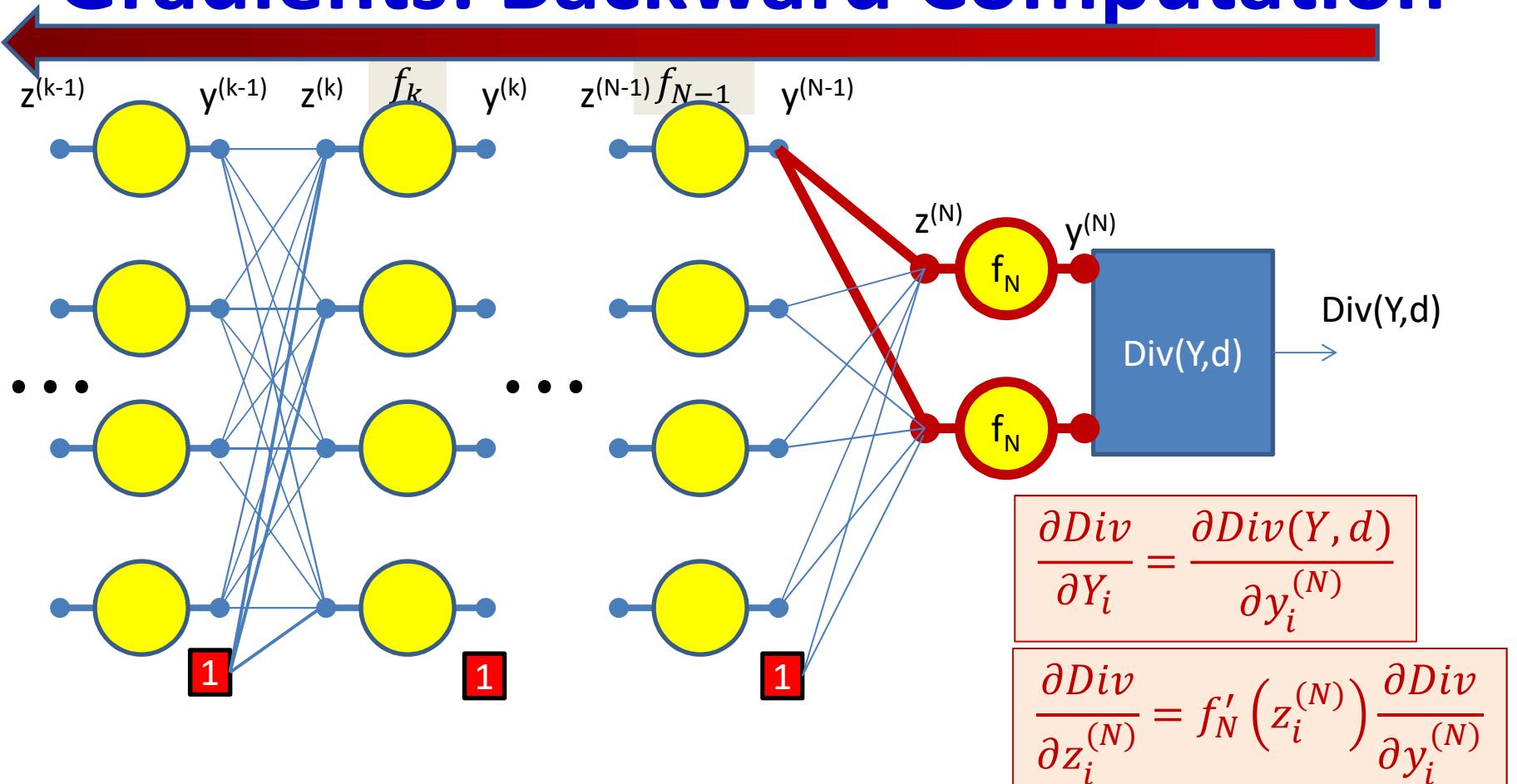
# Gradients: Backward Computation



Derivative of the activation function of Nth layer

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

# Gradients: Backward Computation

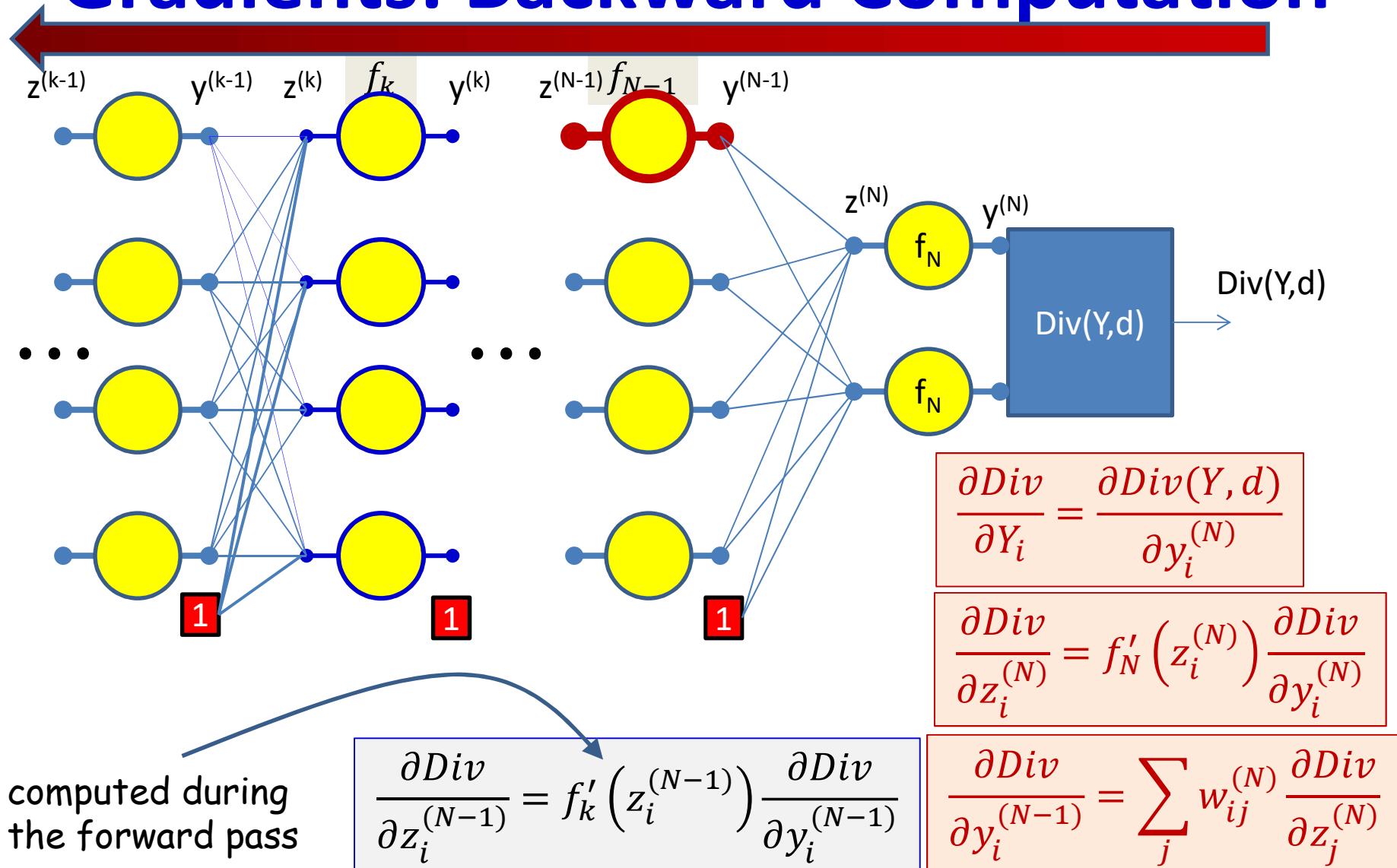


$$\frac{\partial Div}{\partial y_i^{(N-1)}} = \sum_j \frac{\partial z_j^{(N)}}{\partial y_i^{(N-1)}} \frac{\partial Div}{\partial z_j^{(N)}} = \sum_j w_{ij}^{(N)} \frac{\partial Div}{\partial z_j^{(N)}}$$

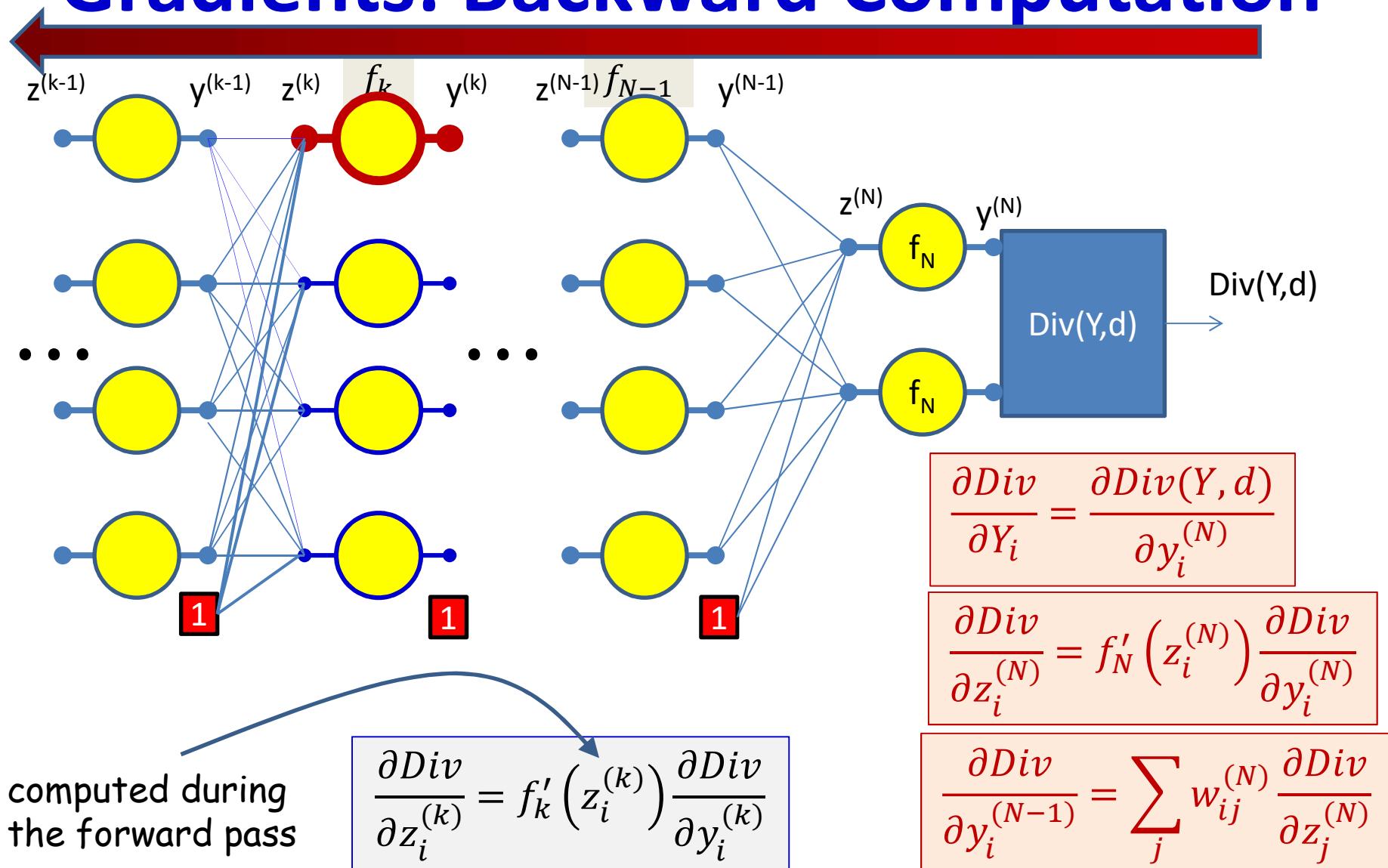
Because :

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

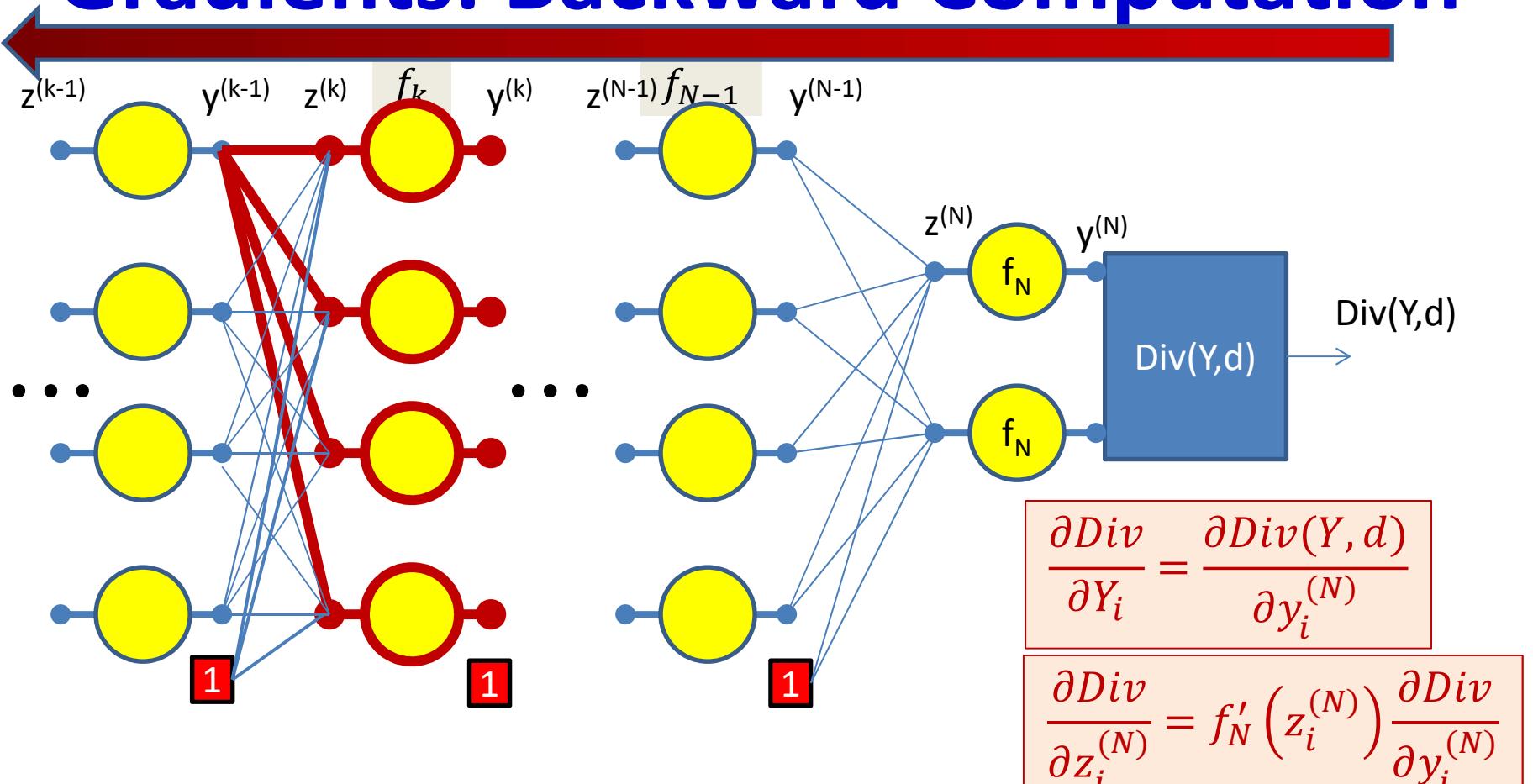
# Gradients: Backward Computation



# Gradients: Backward Computation

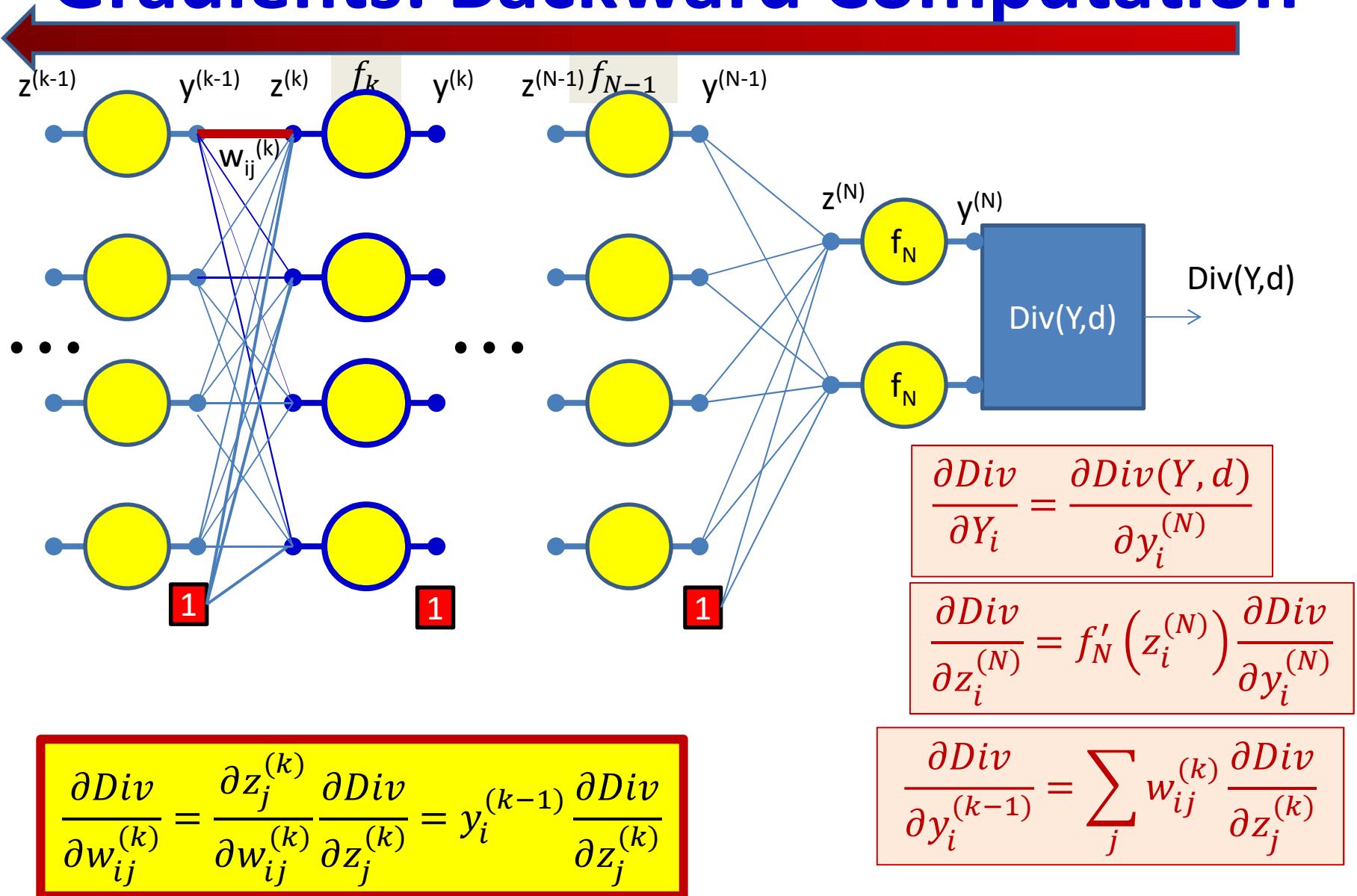


# Gradients: Backward Computation

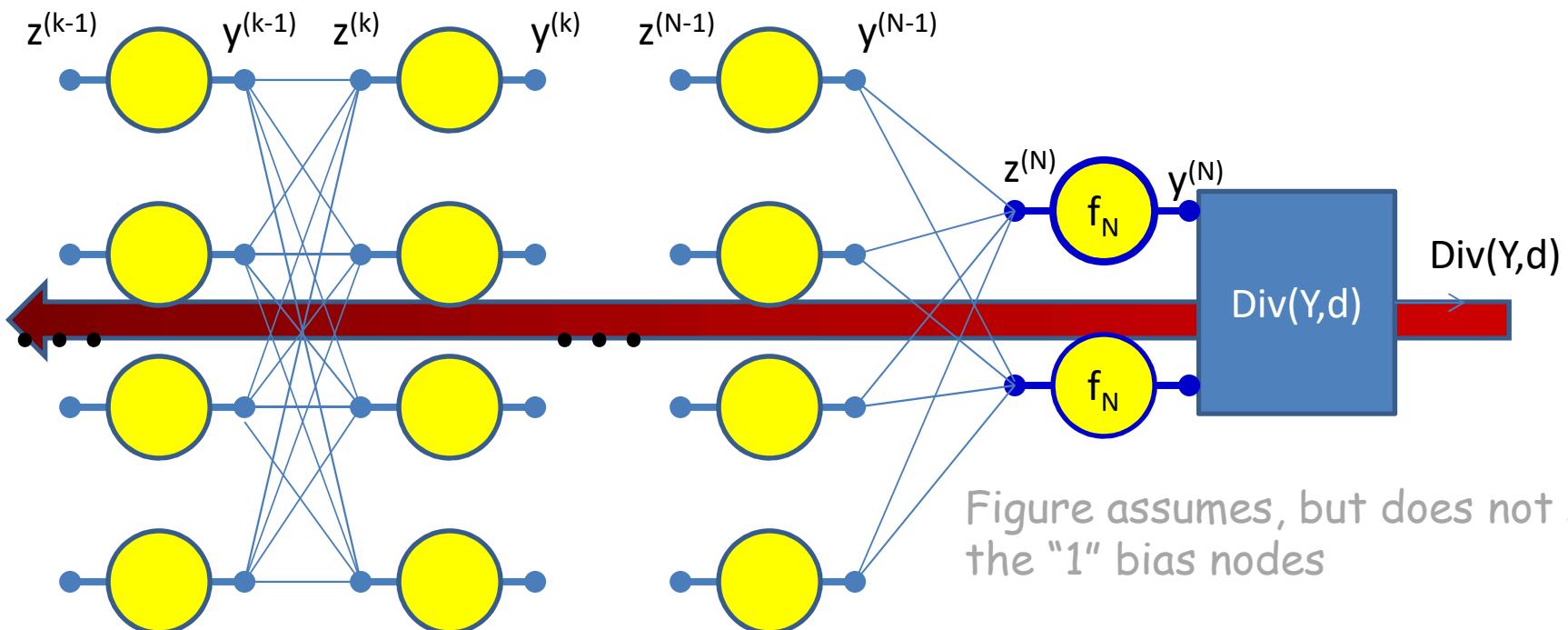


$$\frac{\partial \text{Div}}{\partial y_i^{(k-1)}} = \sum_j \frac{\partial z_j^{(k)}}{\partial y_i^{(k-1)}} \frac{\partial \text{Div}}{\partial z_j^{(k)}} = \sum_j w_{ij}^{(k)} \frac{\partial \text{Div}}{\partial z_j^{(k)}}$$

# Gradients: Backward Computation



# Gradients: Backward Computation



Initialize: Gradient  
w.r.t network output

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

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

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

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

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

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

# Backward Pass

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

# Backward Pass

- Output layer ( $N$ ) :

– For  $i = 1 \dots D_N$

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

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

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

Very analogous to the forward pass:

- For layer  $k = N - 1$  down to 0

– For  $i = 1 \dots D_k$

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

Backward weighted combination of next layer

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

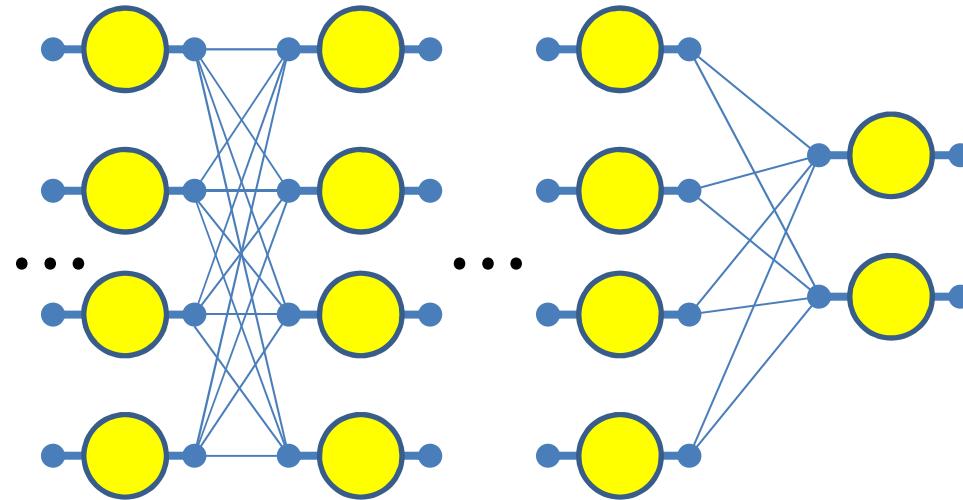
Backward equivalent of activation

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

# For comparison: the forward pass again

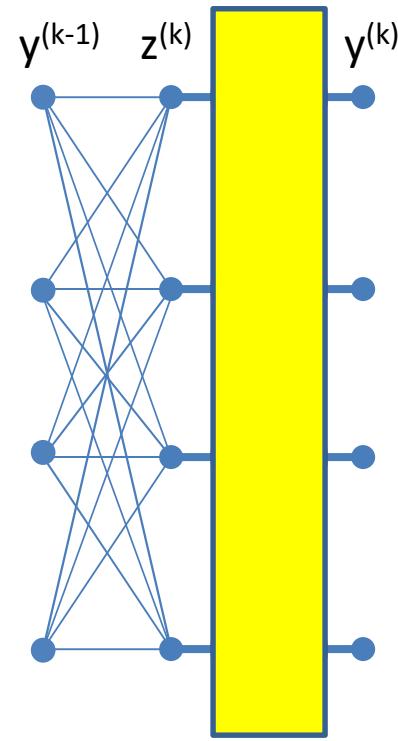
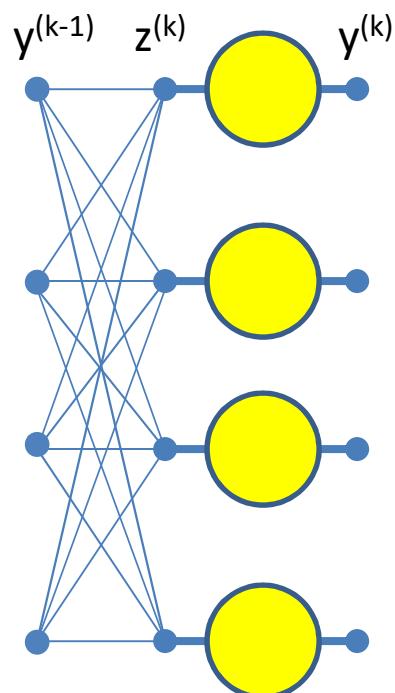
- Input:  $D$  dimensional vector  $\mathbf{x} = [x_j, j = 1 \dots D]$
- Set:
  - $D_0 = D$ , is the width of the 0<sup>th</sup> (input) layer
  - $y_j^{(0)} = x_j, j = 1 \dots D; y_0^{(k=1\dots N)} = x_0 = 1$
- For layer  $k = 1 \dots N$ 
  - For  $j = 1 \dots D_k$ 
    - $z_j^{(k)} = \sum_{i=0}^{N_k}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

# Special cases



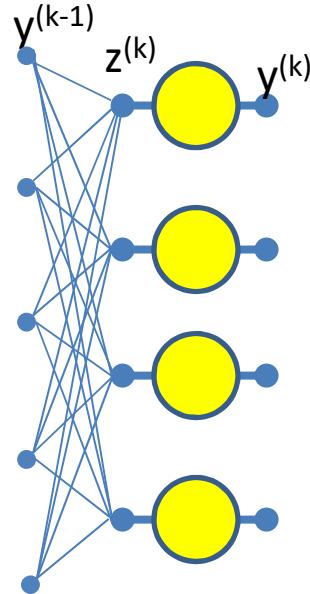
- Have assumed so far that
  1. The computation of the output of one neuron does not directly affect computation of other neurons in the same (or previous) layers
  2. Outputs of neurons only combine through weighted addition
  3. Activations are actually differentiable
    - All of these conditions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

# Special Case 1. Vector activations



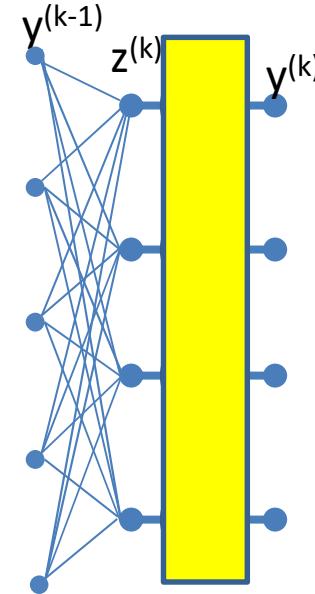
- Vector activations: all outputs are functions of all inputs

# Special Case 1. Vector activations



Scalar activation: Modifying a  $z_i$  only changes corresponding  $y_i$

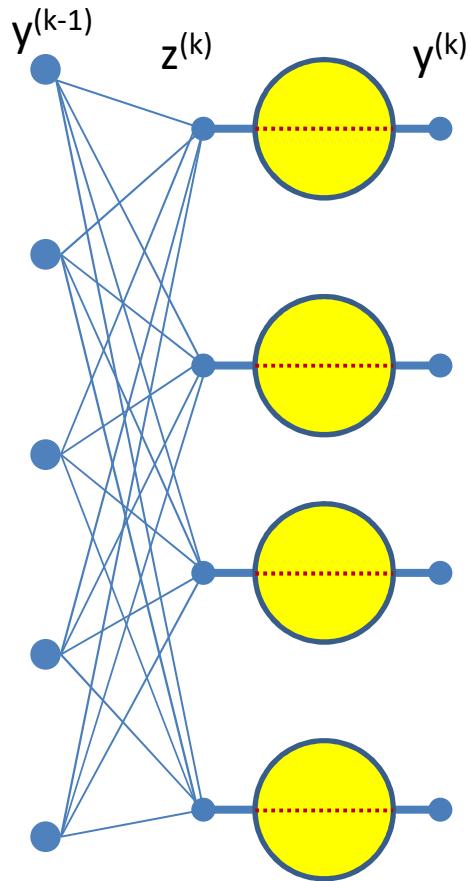
$$y_i^{(k)} = f(z_i^{(k)})$$



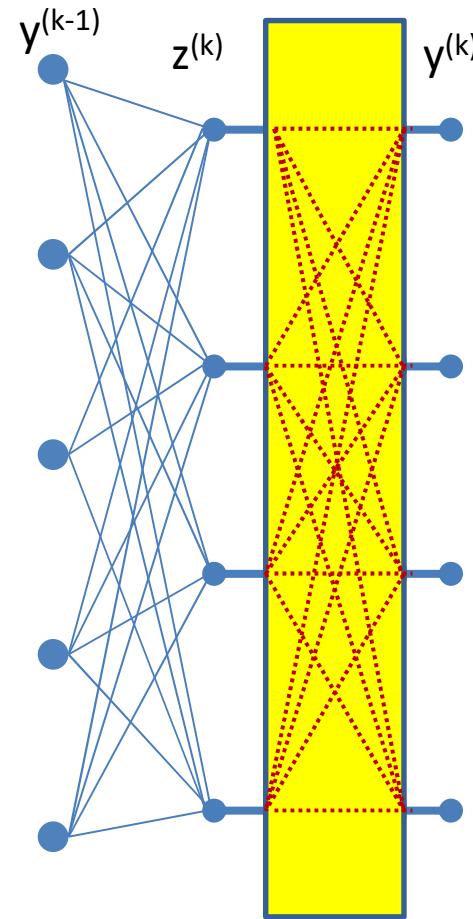
Vector activation: Modifying a  $z_i$  potentially changes all,  $y_1 \dots y_M$

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

# “Influence” diagram

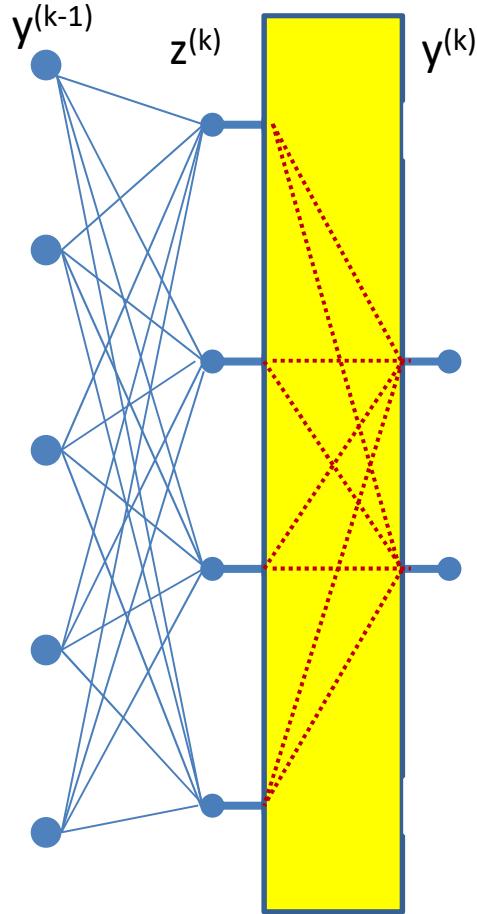
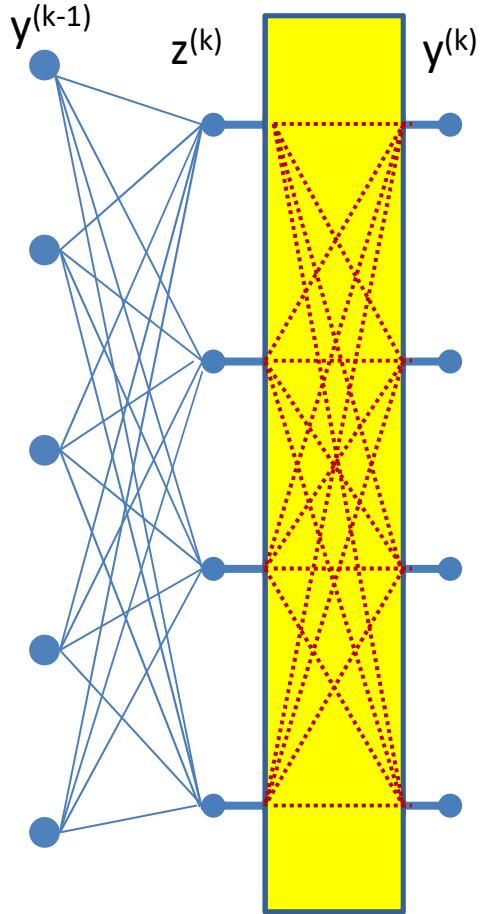


Scalar activation: Each  $z_i$  influences *one*  $y_i$



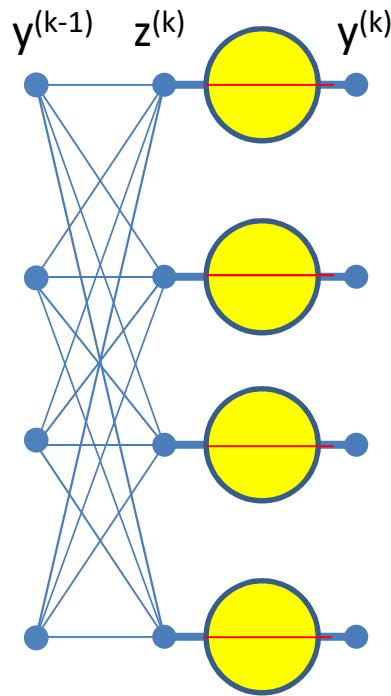
Vector activation: Each  $z_i$  influences all,  $y_1 \dots y_M$

# 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

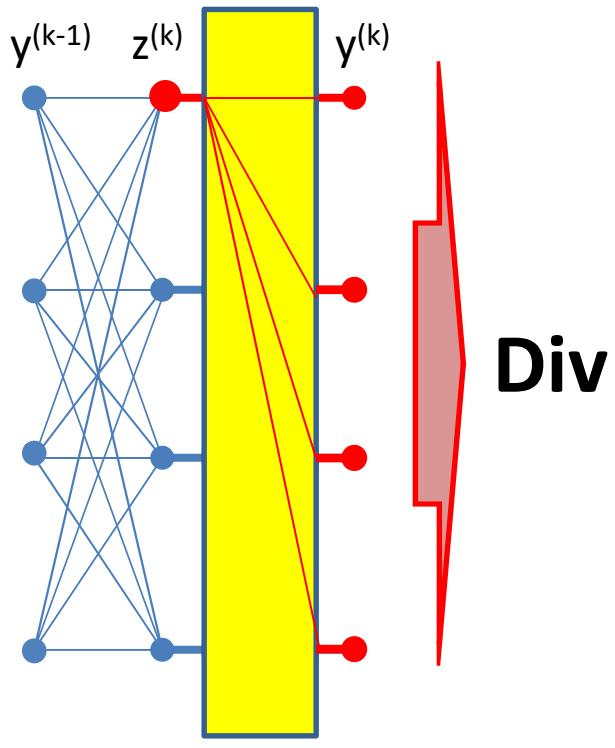
# Scalar Activation: Derivative rule



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

- 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:

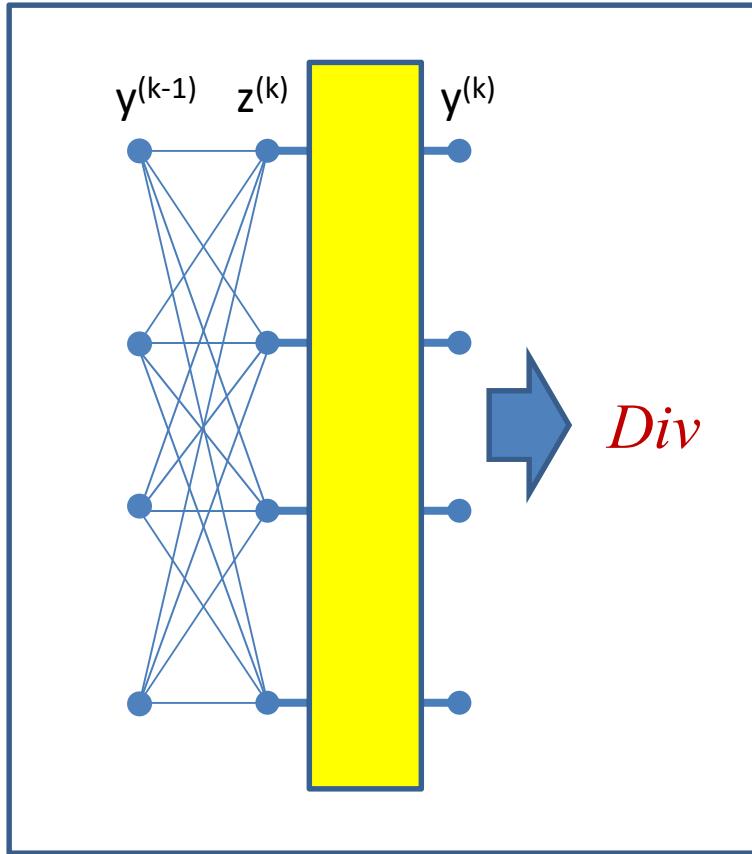
$$\frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} = 0 \text{ for } i \neq j$$

- 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  $y_j^{(k)}$

# Special cases

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

# Example Vector Activation: Softmax



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

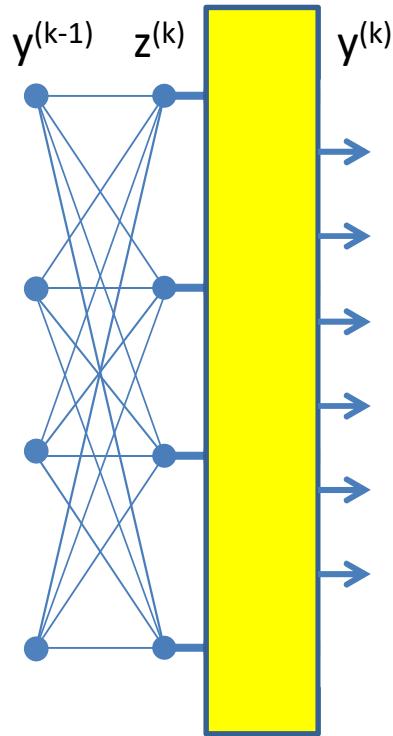
$$\frac{\partial \text{Div}}{\partial z_i^{(k)}} = \sum_j \frac{\partial \text{Div}}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}}$$

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

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

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

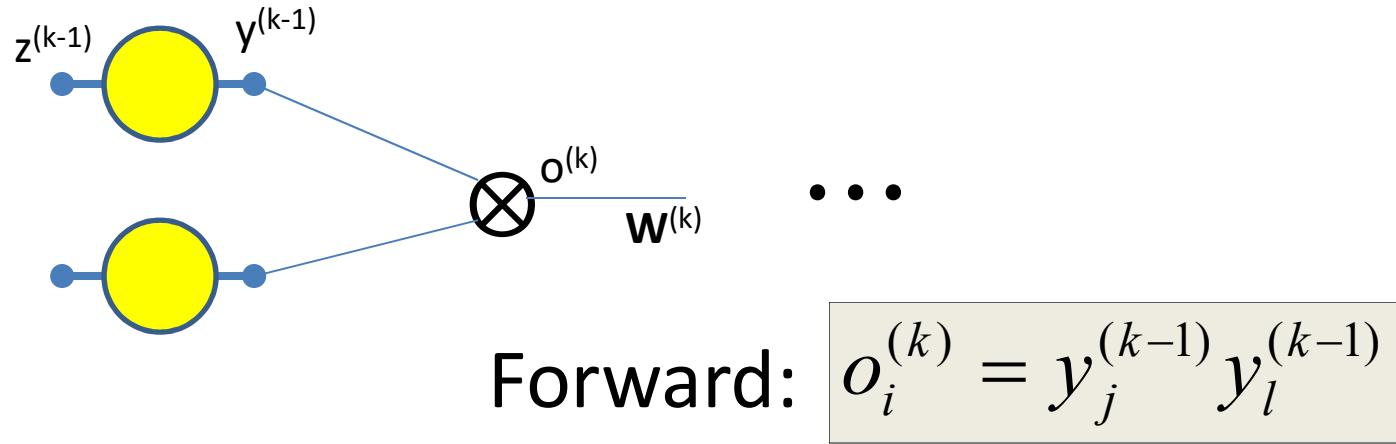
# Vector Activations



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

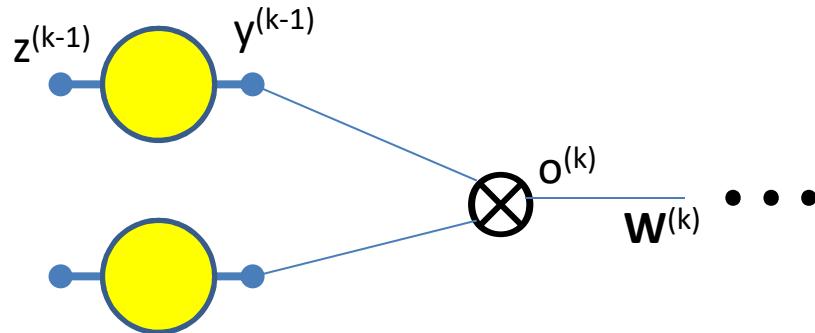
- In reality the vector combinations can be anything
  - E.g. linear combinations, polynomials, logistic (softmax), etc.

# 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:

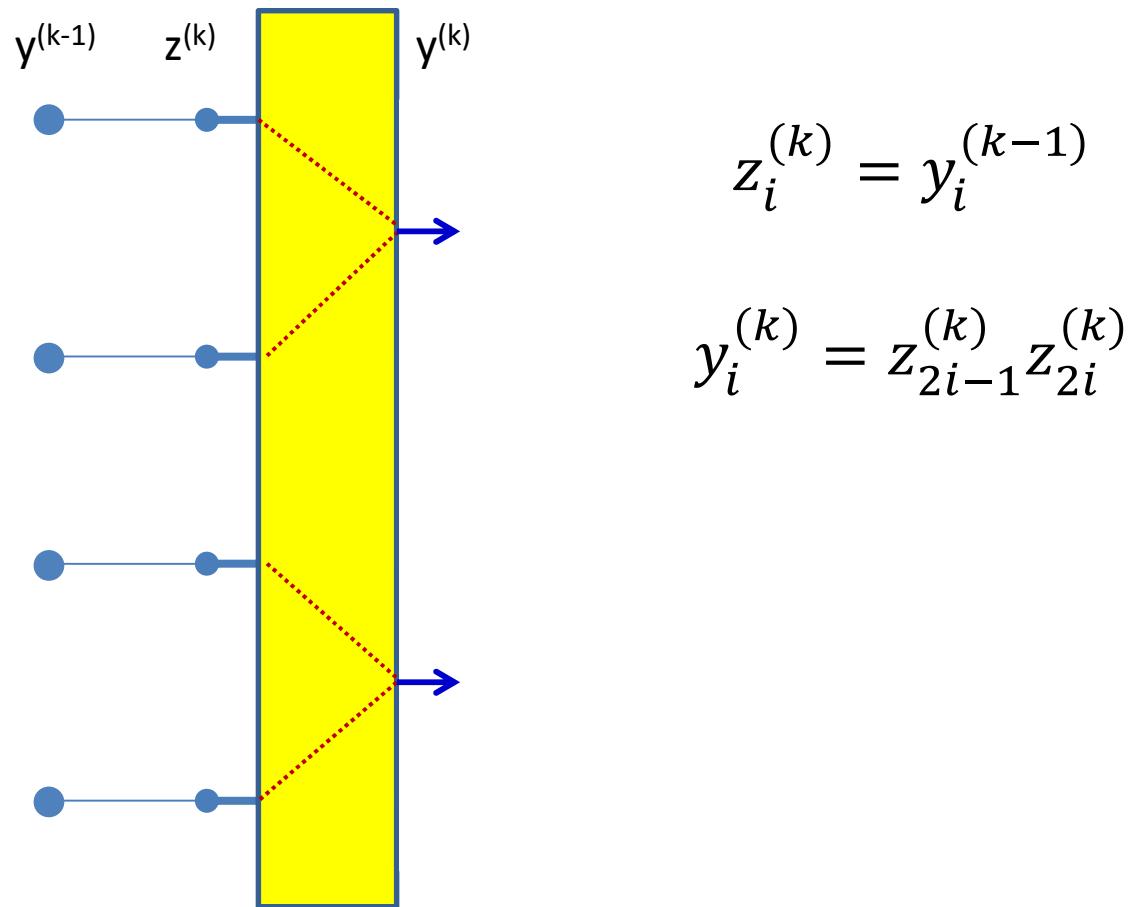
$$\frac{\partial \text{Div}}{\partial o_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial \text{Div}}{\partial z_j^{(k+1)}}$$

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

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

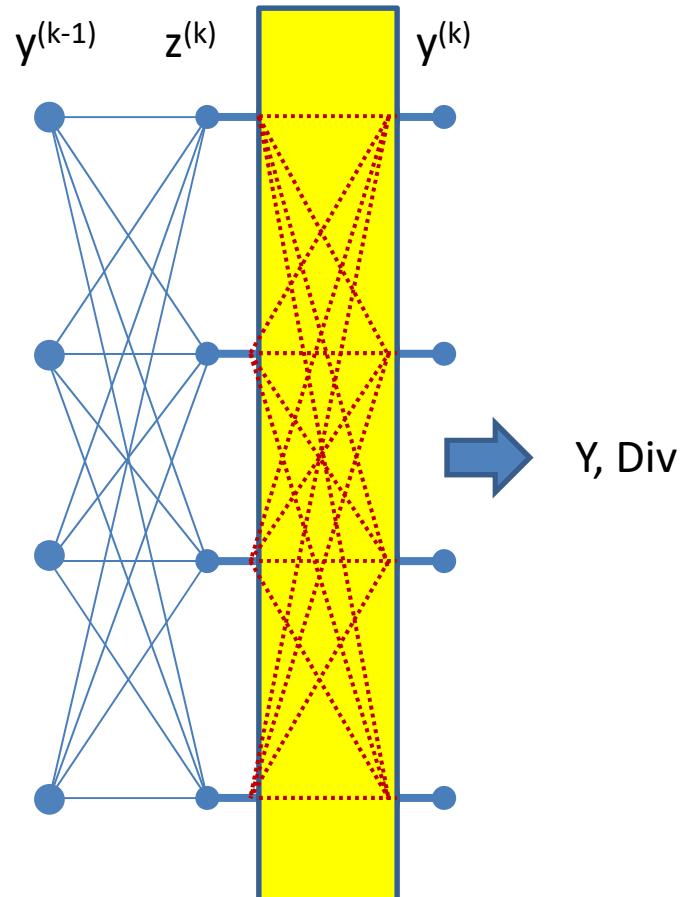
- Some types of networks have *multiplicative* combination

# Multiplicative combination as a case of vector activations



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

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



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

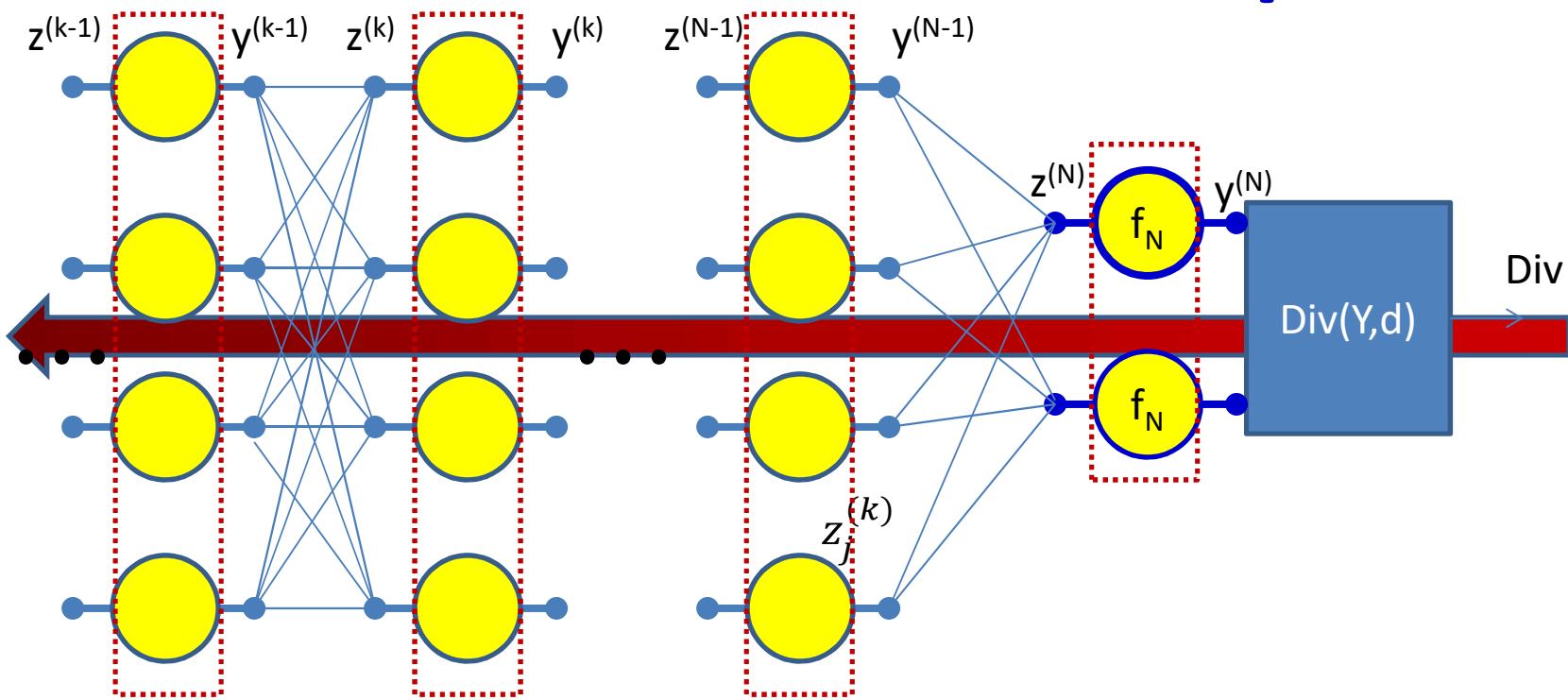
$$y_i^{(k)} = \prod_l (z_l^{(k)})^{\alpha_{li}^{(k)}}$$

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

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

- 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

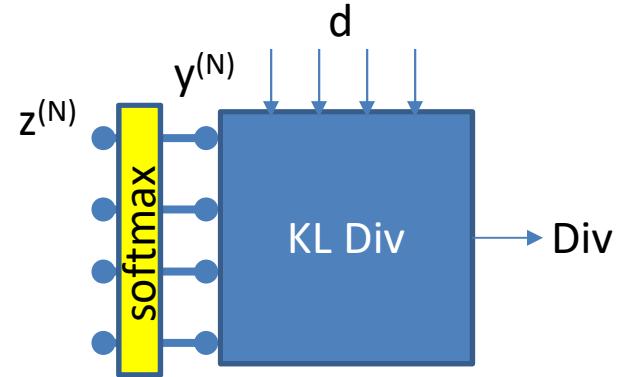
$$\frac{\partial \text{Div}}{\partial z_i^{(k)}} = \frac{\partial \text{Div}}{\partial y_i^{(k)}} \frac{\partial y_i^{(k)}}{\partial z_i^{(k)}}$$

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

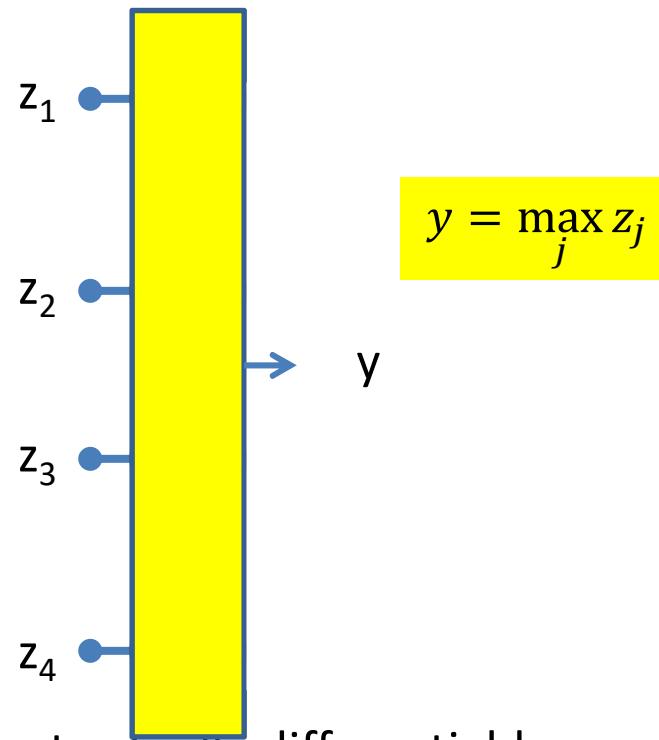
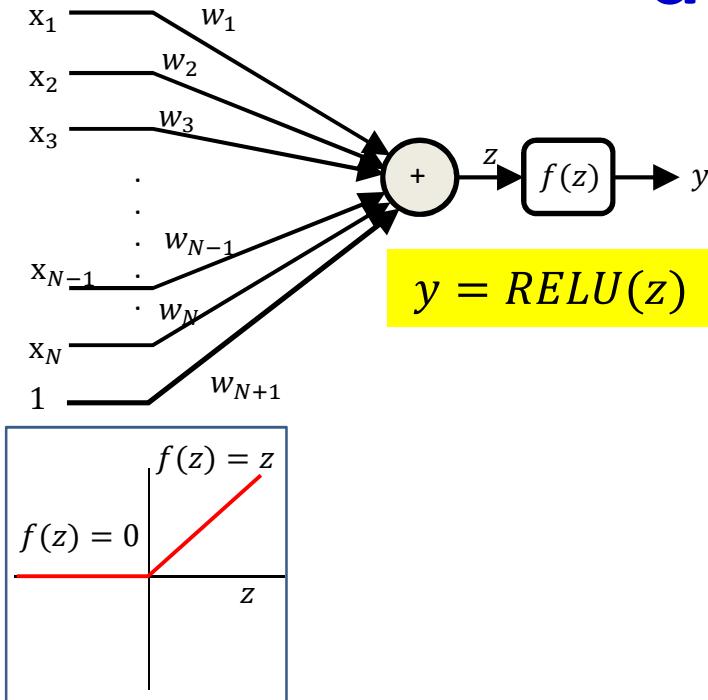
$$\frac{\partial \text{Div}}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial \text{Div}}{\partial z_j^{(k)}}$$

# Backward Pass for softmax output layer

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

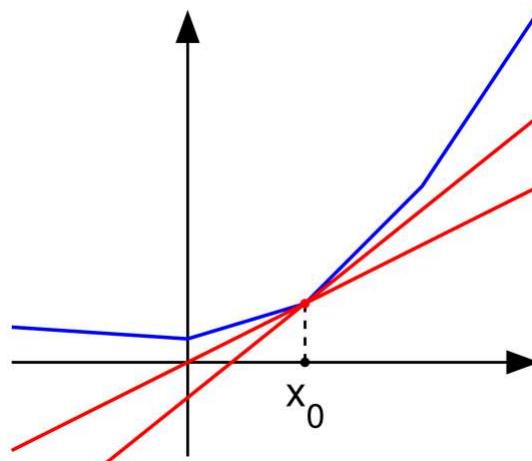


# Special Case 3: Non-differentiable activations



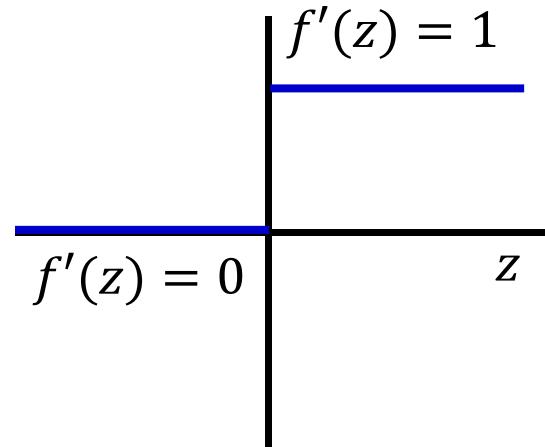
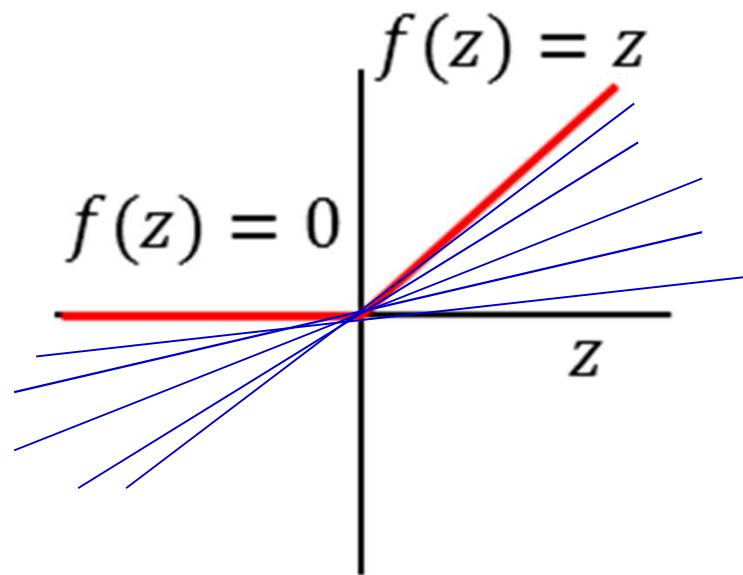
- Activation functions are sometimes not actually differentiable
  - E.g. The RELU (Rectified Linear Unit)
    - And its variants: leaky RELU, randomized leaky RELU
  - E.g. The “max” function
- Must use “subgradients” where available
  - Or “secants”

# The subgradient



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

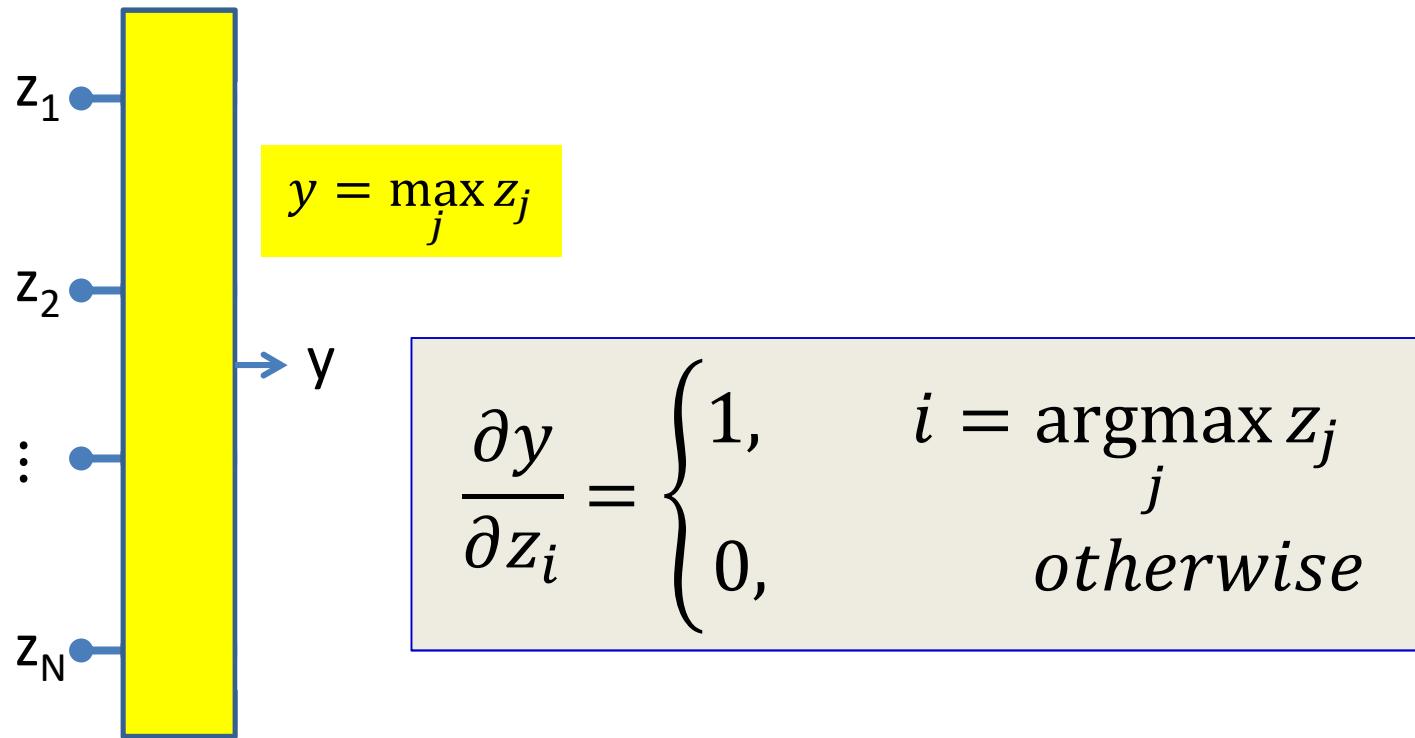
# Subgradients and the RELU



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

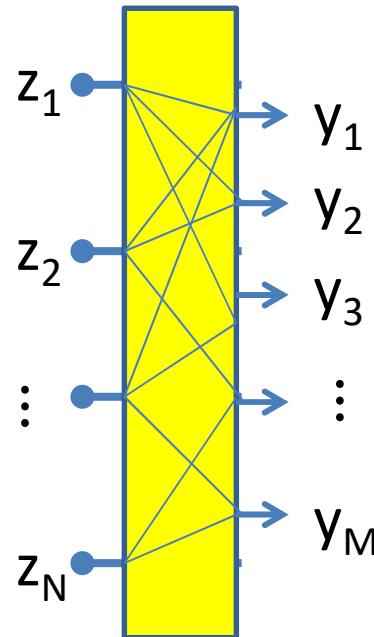
- Can use any subgradient
  - At the differentiable points on the curve, this is the same as the gradient
  - Typically, will use the equation given

# Subgradients and the Max



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

# Subgradients and the Max



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

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

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

# Backward Pass: Recap

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

# 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 Error is the sum of the error over all training instances

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

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

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

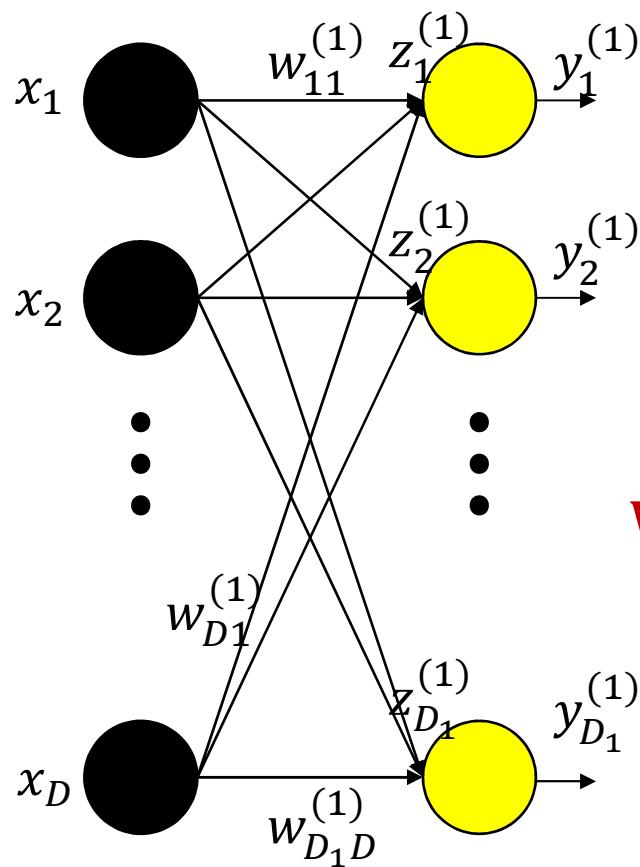
# Training by BackProp

- Initialize all weights  $(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \dots, \mathbf{W}^{(K)})$
- Do:
  - Initialize  $Err = 0$ ; For all  $i, j, k$ , initialize  $\frac{dErr}{dw_{i,j}^{(k)}} = 0$
  - For all  $t = 1:T$  (Loop over training instances)
    - Forward pass: Compute
      - Output  $\mathbf{Y}_t$
      - $Err += Div(\mathbf{Y}_t, \mathbf{d}_t)$
    - Backward pass: For all  $i, j, k$ :
      - Compute  $\frac{dDiv(\mathbf{Y}_t, \mathbf{d}_t)}{dw_{i,j}^{(k)}}$
      - Compute  $\frac{dErr}{dw_{i,j}^{(k)}} += \frac{dDiv(\mathbf{Y}_t, \mathbf{d}_t)}{dw_{i,j}^{(k)}}$
    - For all  $i, j, k$ , update:
$$w_{i,j}^{(k)} = w_{i,j}^{(k)} - \frac{\eta}{T} \frac{dErr}{dw_{i,j}^{(k)}}$$
  - Until  $Err$  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



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

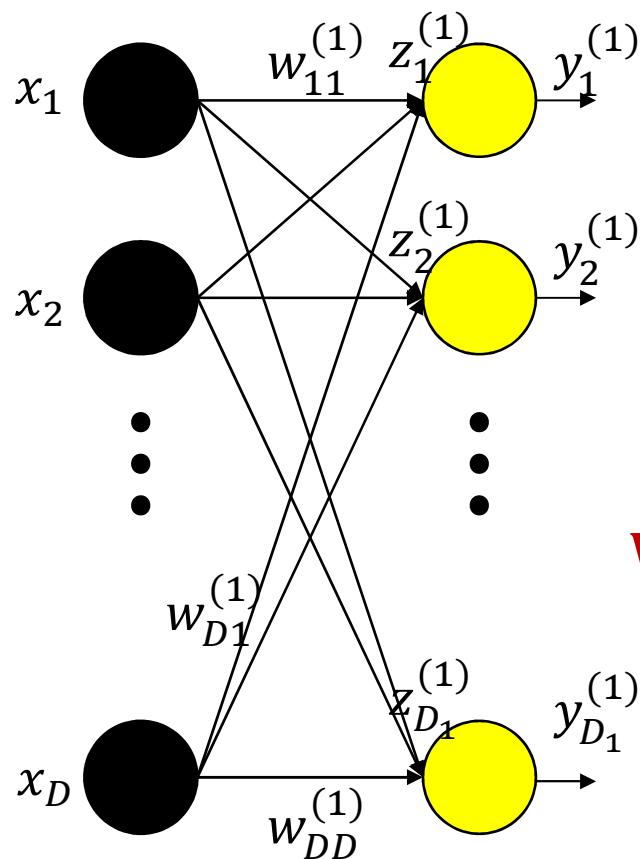
$$\mathbf{z}_k = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{D_k}^{(k)} \end{bmatrix}$$

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

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

- 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



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

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

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

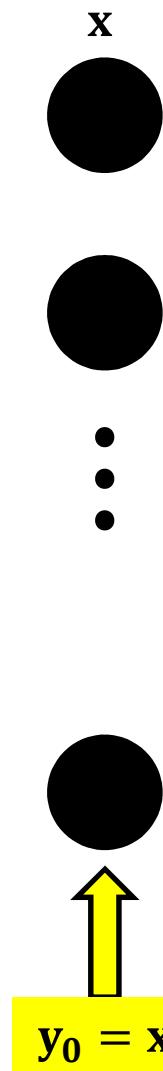
$$\mathbf{W}_k = \begin{bmatrix} w_{11}^{(k)} & w_{21}^{(k)} & \vdots & w_{D_k 1}^{(k)} \\ w_{12}^{(k)} & w_{22}^{(k)} & \vdots & w_{D_k 2}^{(k)} \\ \dots & \dots & \ddots & \vdots \\ w_{1D_{k+1}}^{(k)} & w_{2D_{k+1}}^{(k)} & \dots & w_{D_k D_{k+1}}^{(k)} \end{bmatrix} \quad \mathbf{b}_k = \begin{bmatrix}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

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

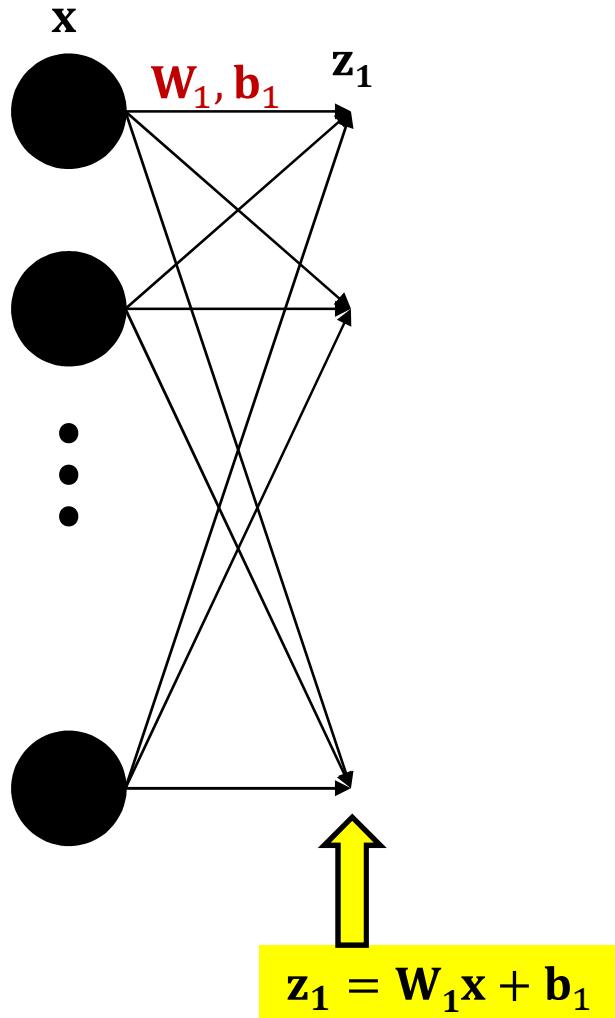
$$\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k$$

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

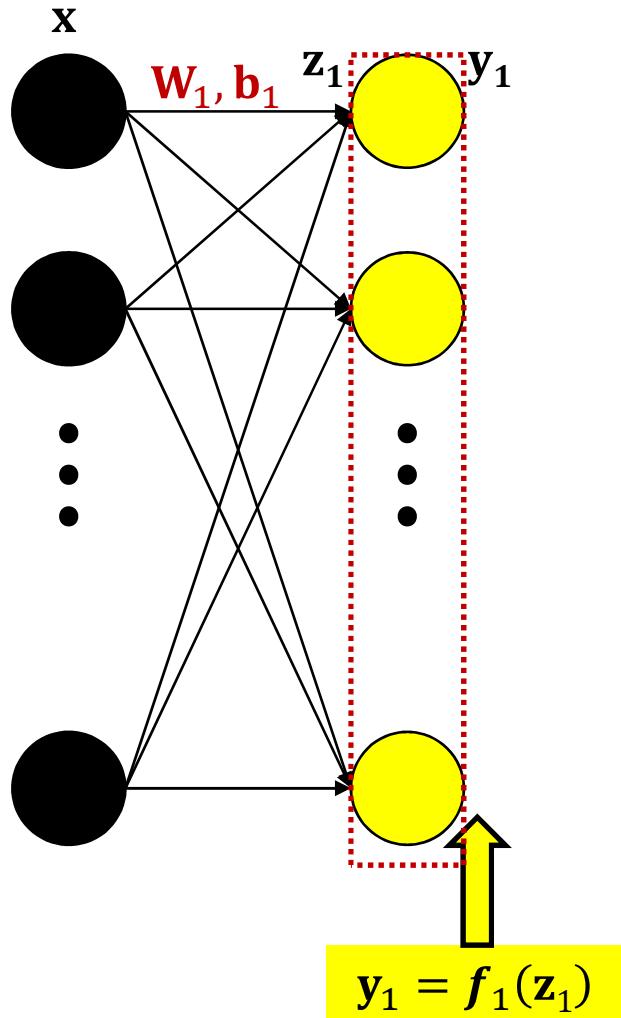
# The forward pass: Evaluating the network



# The forward pass



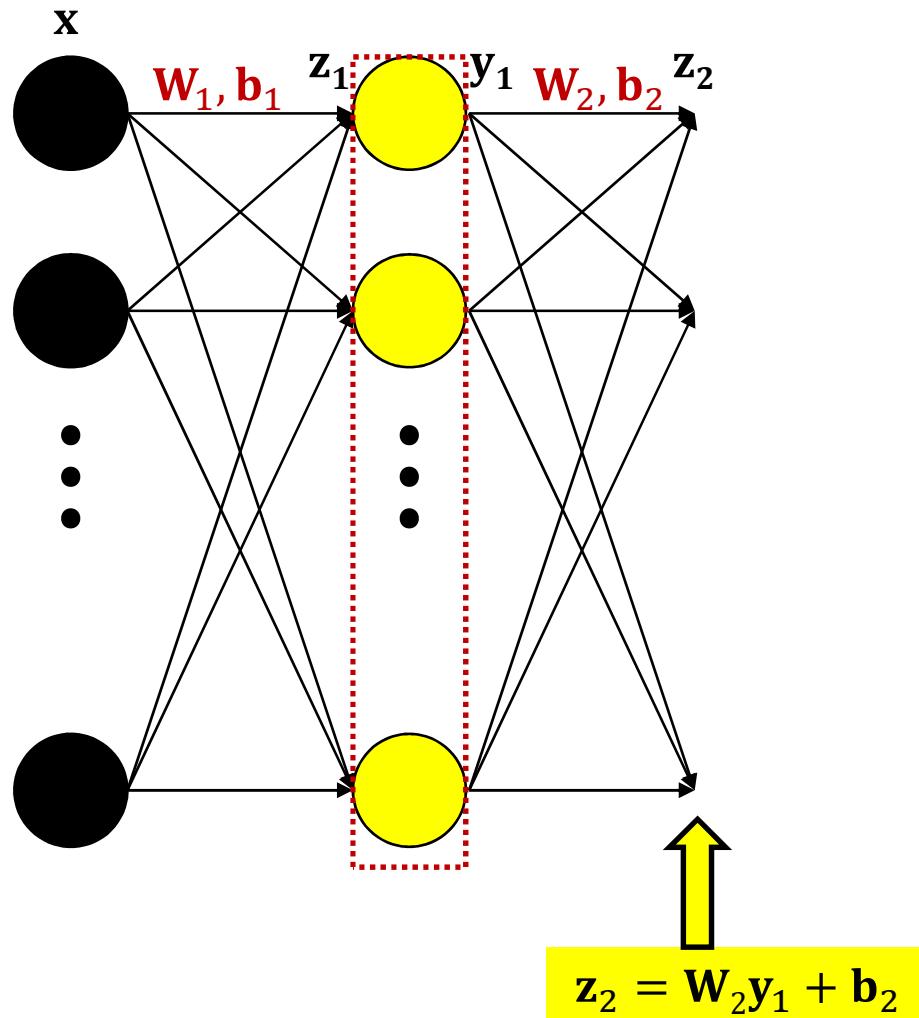
# The forward pass



The Complete computation

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

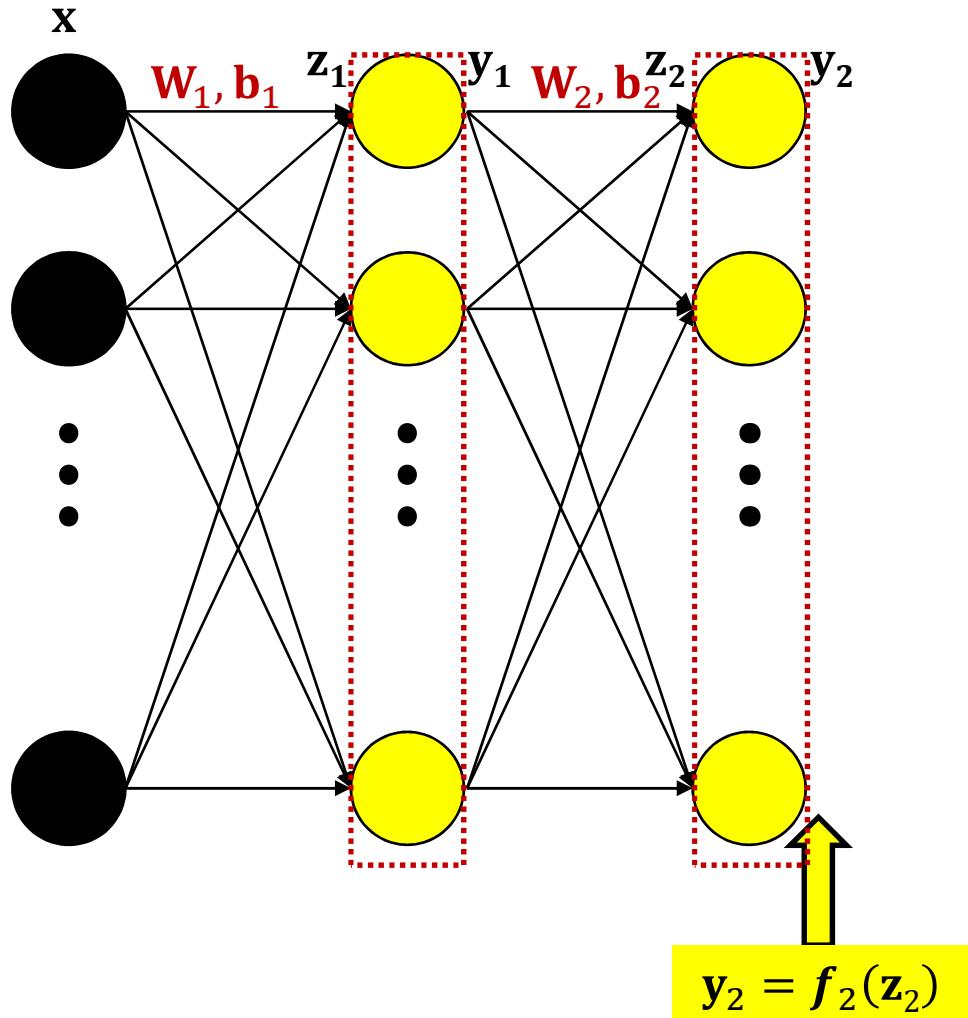
# The forward pass



The Complete computation

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

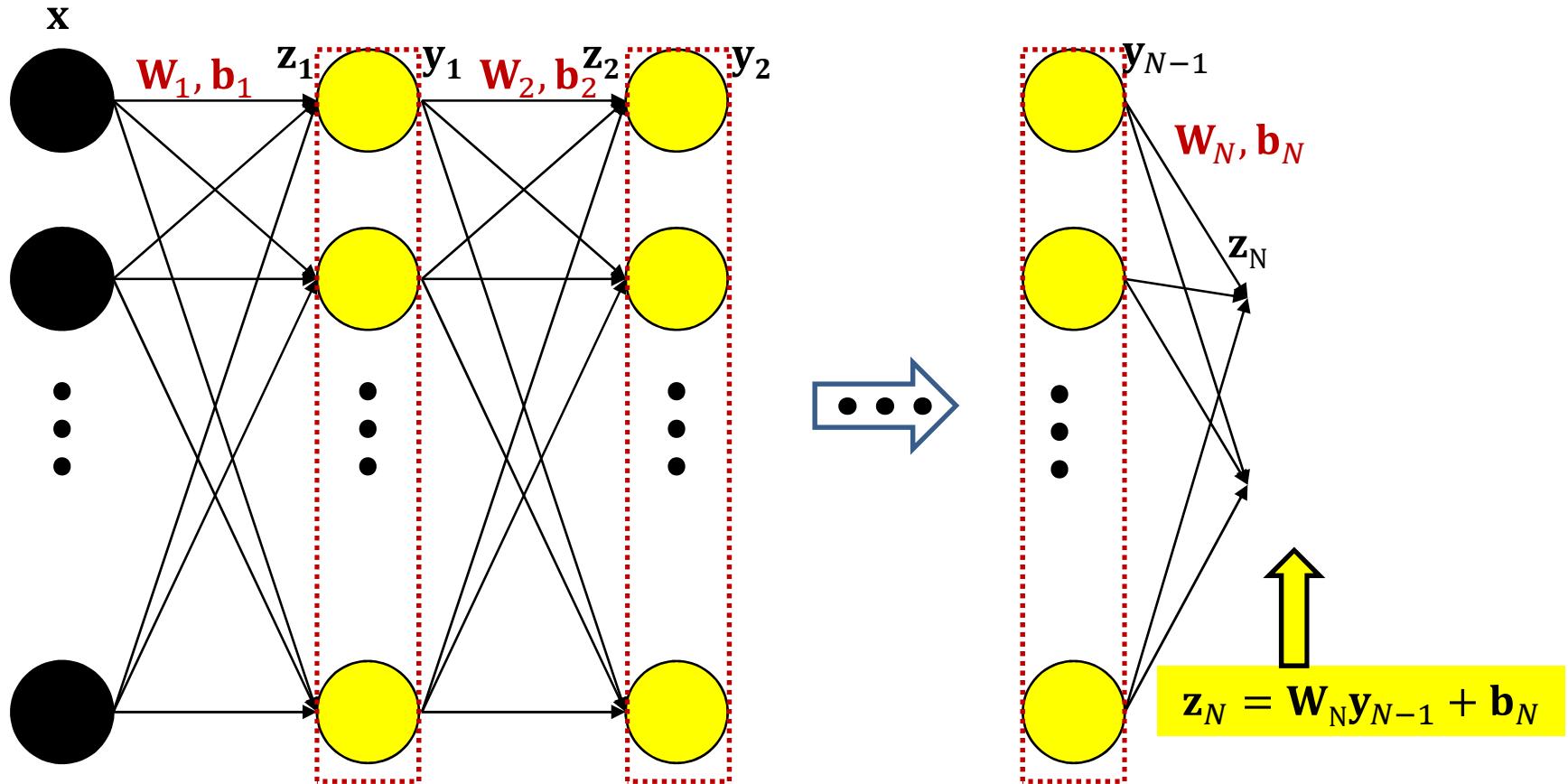
# The forward pass



The Complete computation

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

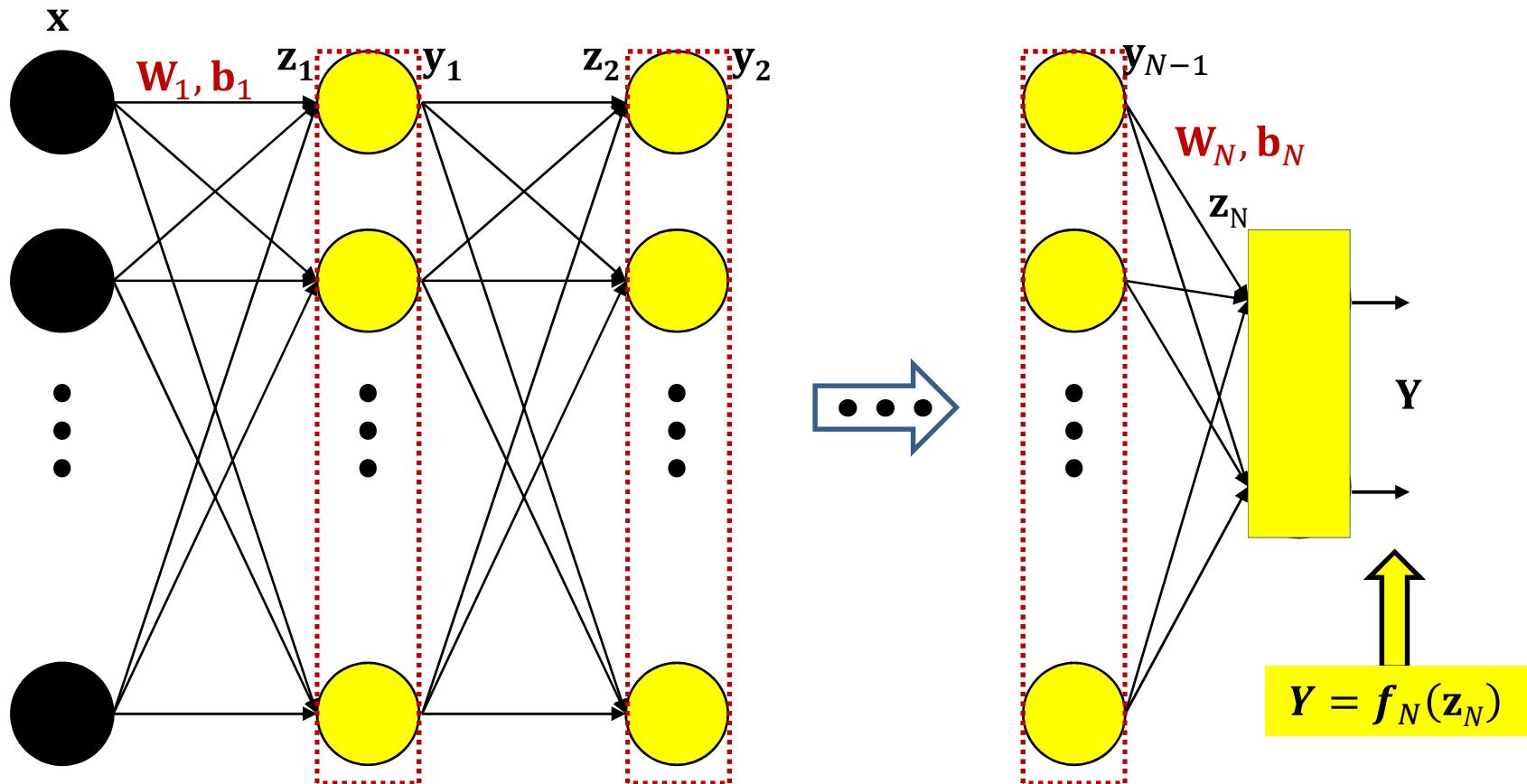
# The forward pass



The Complete computation

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

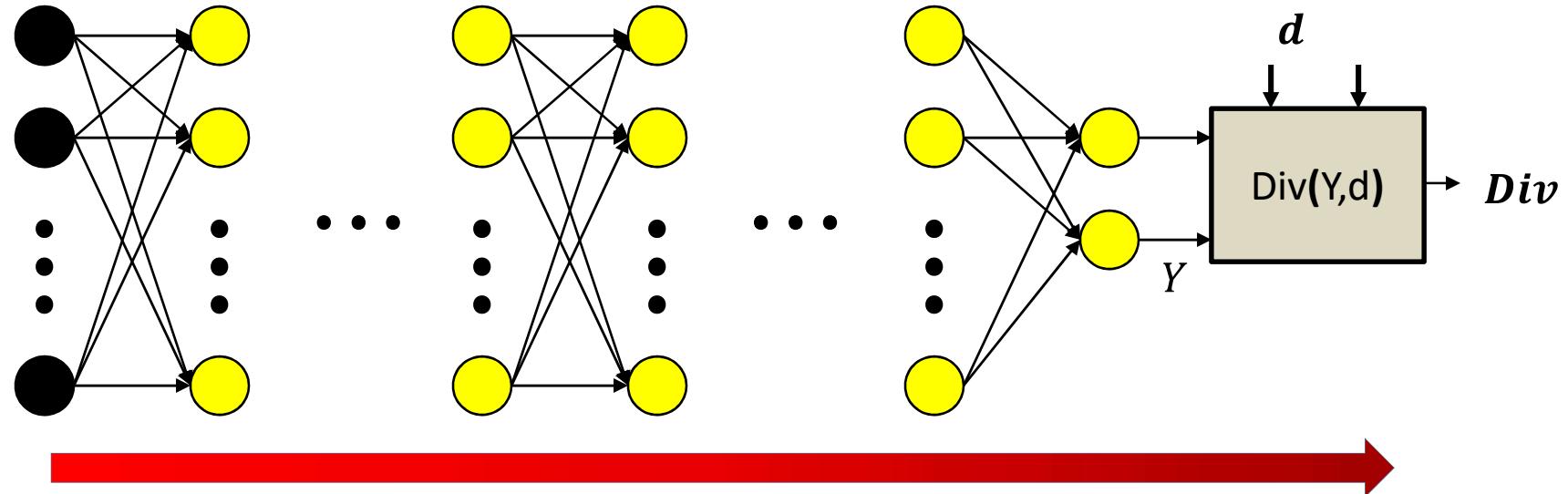
# The forward pass



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

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

# The Forward Pass

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

– Recursion:

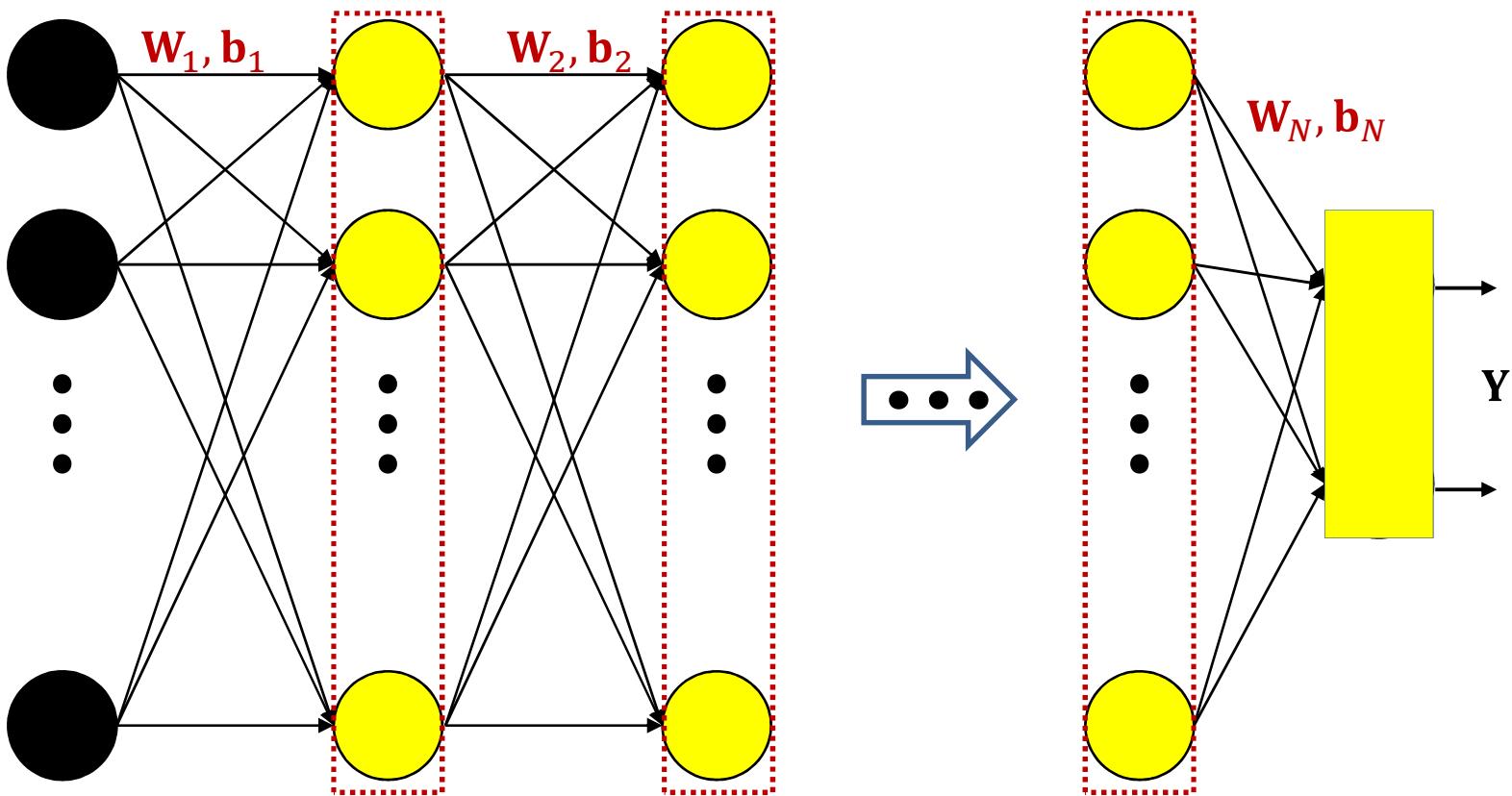
$$\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k$$

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

- Output:

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

# The backward pass



- The network is a nested function

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

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

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

# Calculus recap 2: The Jacobian

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

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

Using vector notation

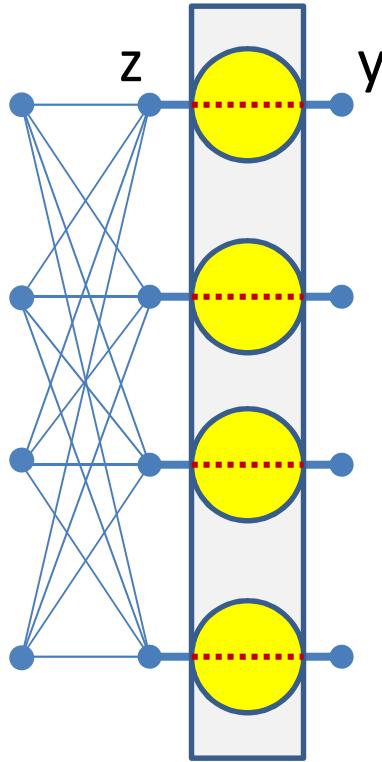
$$\mathbf{y} = f(\mathbf{z})$$

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

Check:

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

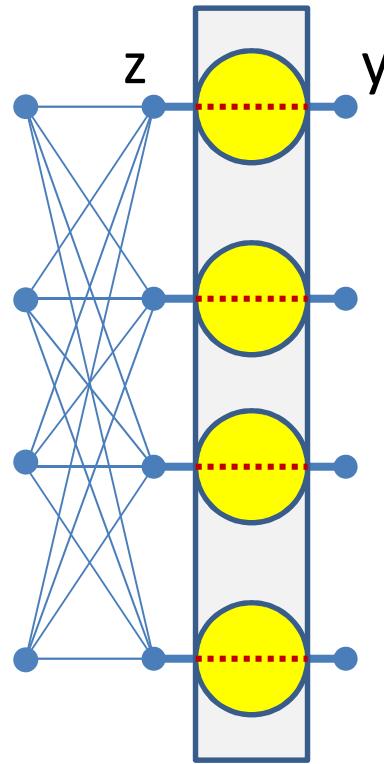
# Jacobians can describe the derivatives of neural activations w.r.t their input



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

- **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

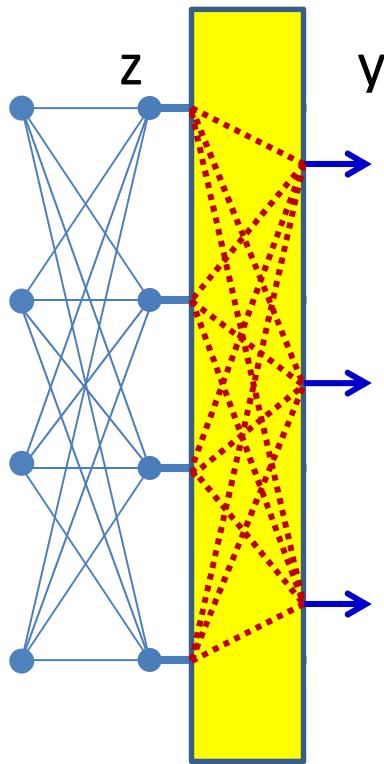


$$y_i = f(z_i)$$

$$J_y(\mathbf{z}) = \begin{bmatrix} f'(y_1) & 0 & \cdots & 0 \\ 0 & f'(y_2) & \cdots & 0 \\ \cdots & \cdots & \ddots & \cdots \\ 0 & 0 & \cdots & f'(y_M) \end{bmatrix}$$

- **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

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

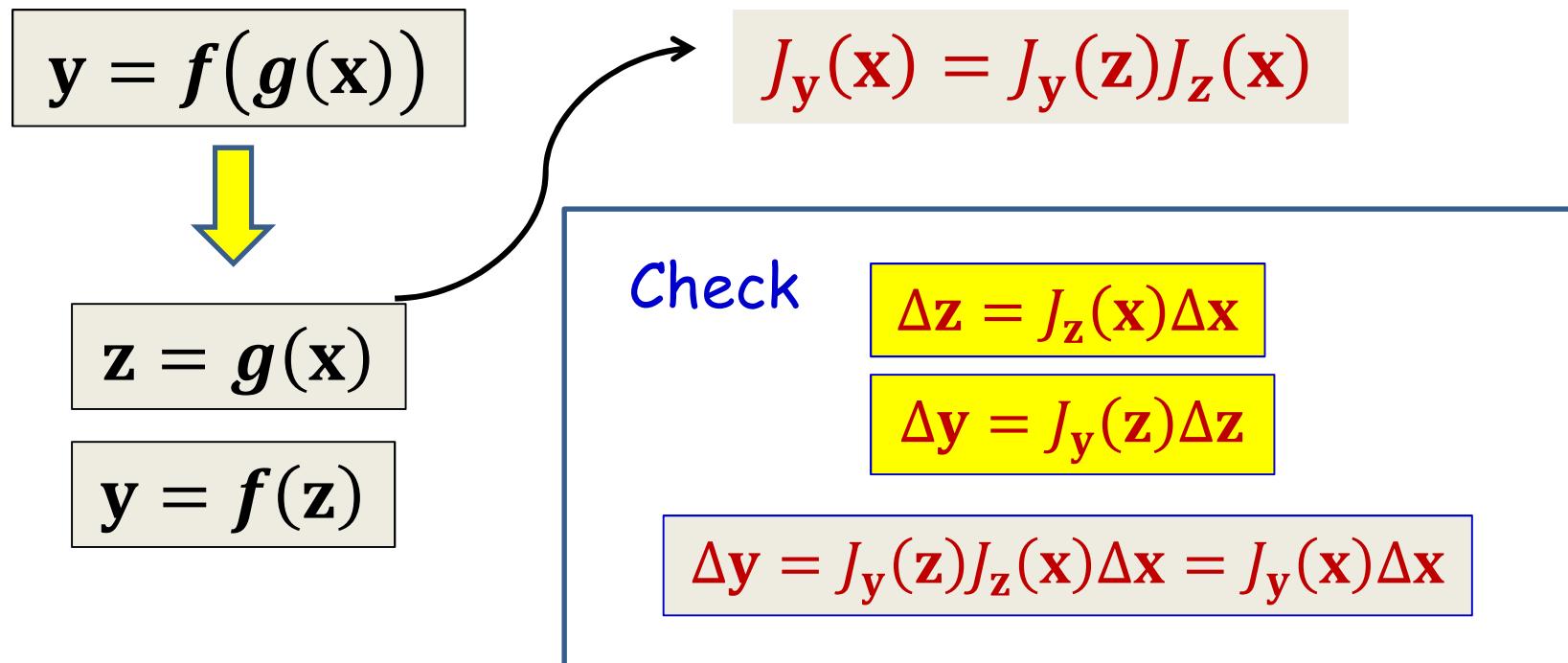


$$J_{\mathbf{z}}(\mathbf{y}) = \mathbf{W}$$

- 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

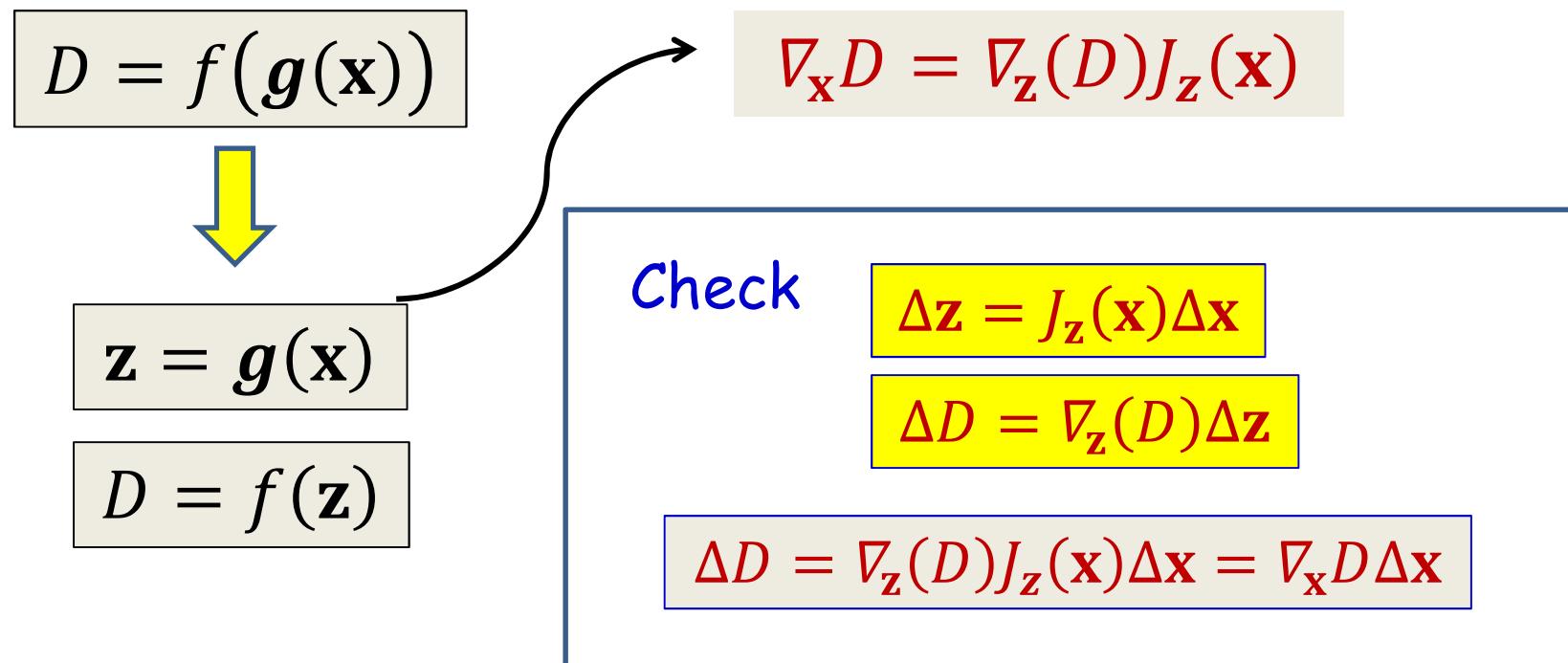
- We can define a chain rule for Jacobians
- **For vector functions of vector inputs:**



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

# Vector derivatives: Chain rule

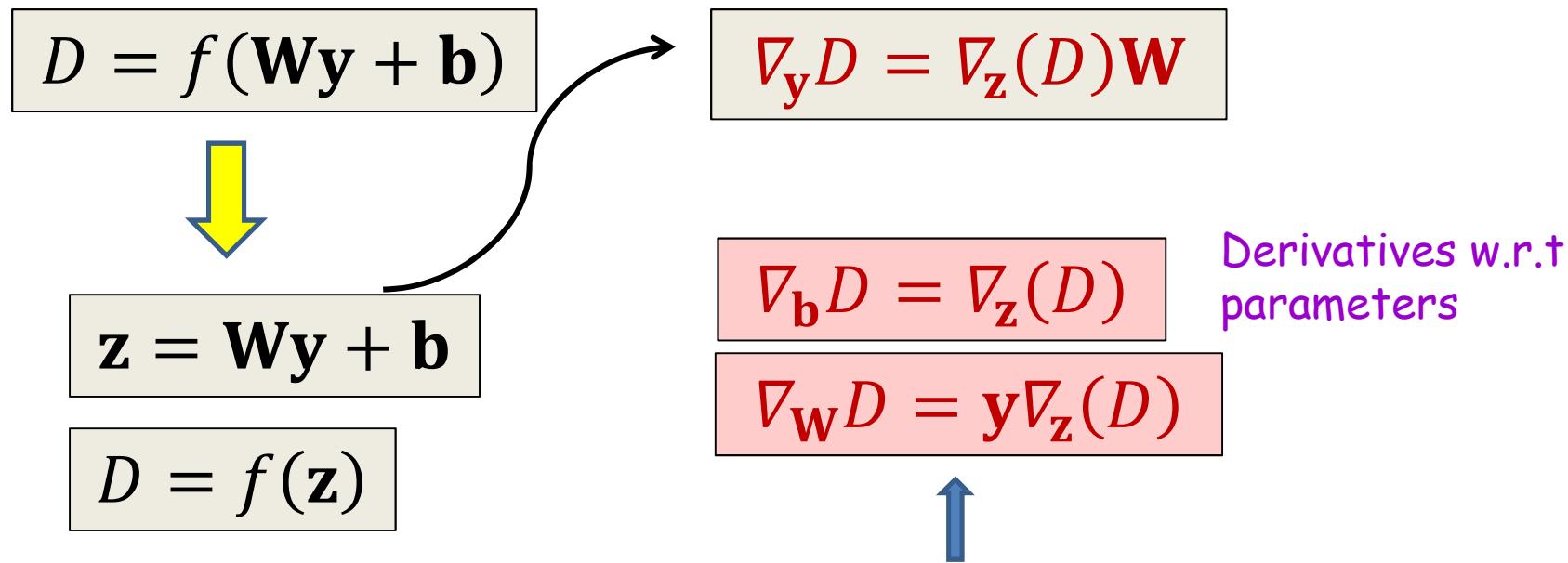
- *The chain rule can combine Jacobians and Gradients*
- **For scalar functions of vector inputs ( $g()$  is vector):**



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

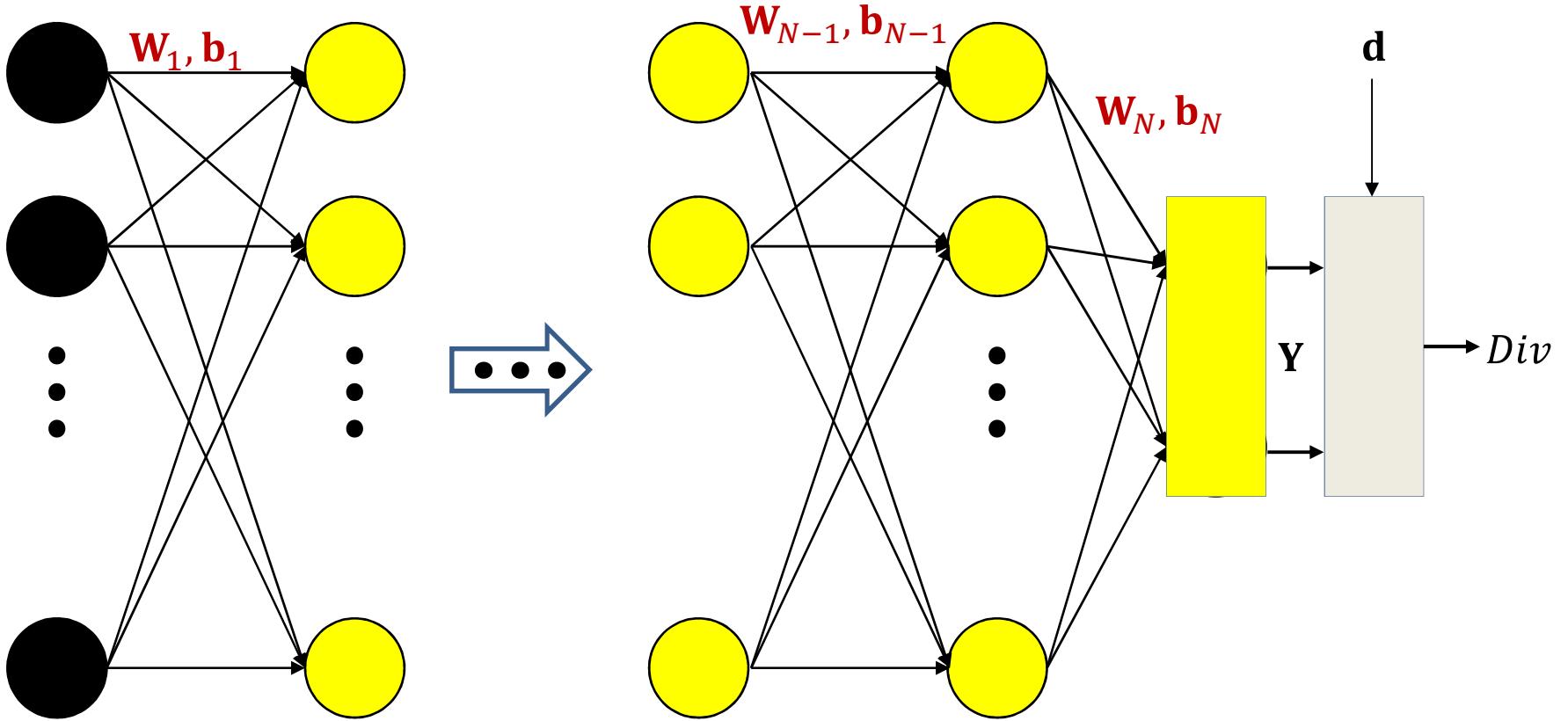
# Special Case

- 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

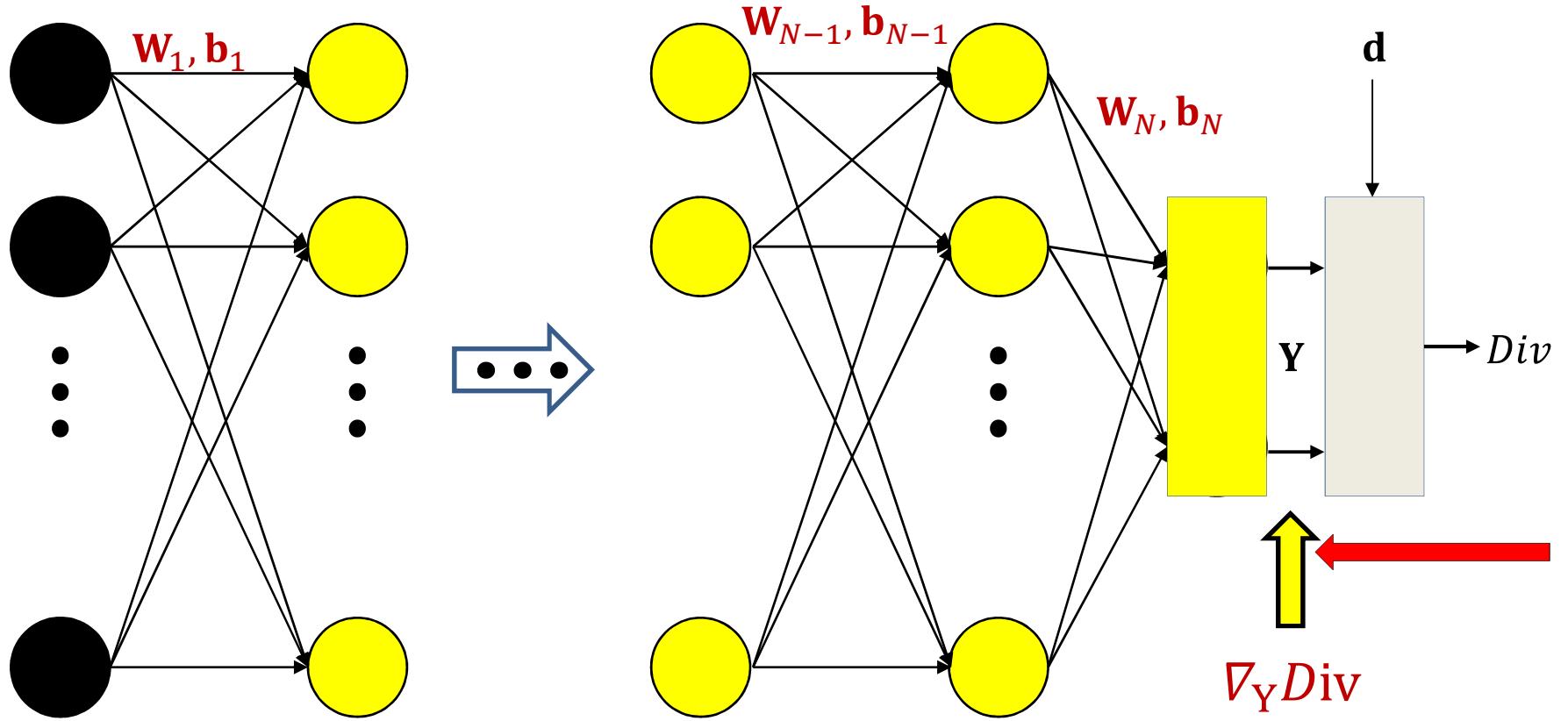
# The backward pass



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

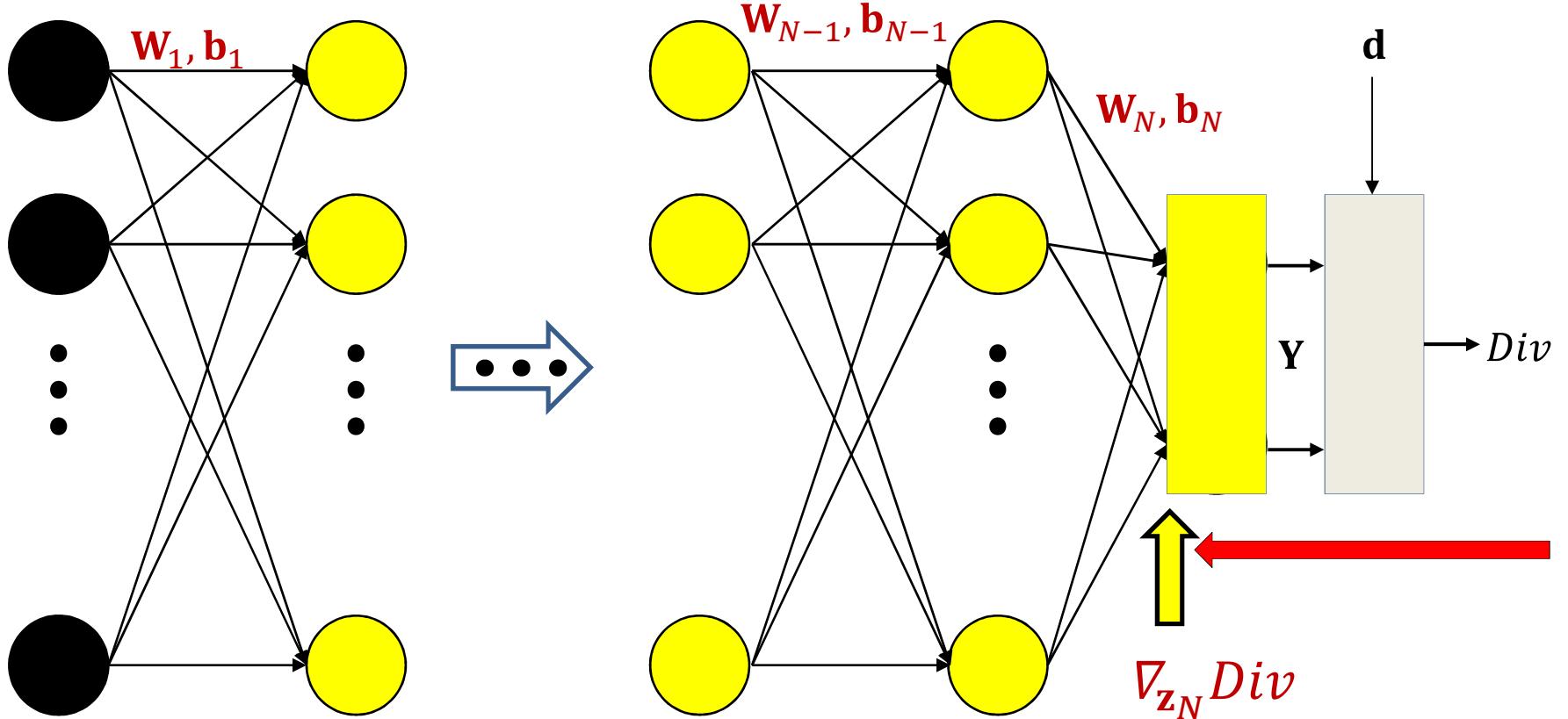
In general  $\nabla_a \mathbf{b}$  represents a derivative of  $\mathbf{b}$  w.r.t.  $a$  and could be a gradient (for scalar  $\mathbf{b}$ ) Or a Jacobian (for vector  $\mathbf{b}$ )

# The backward pass



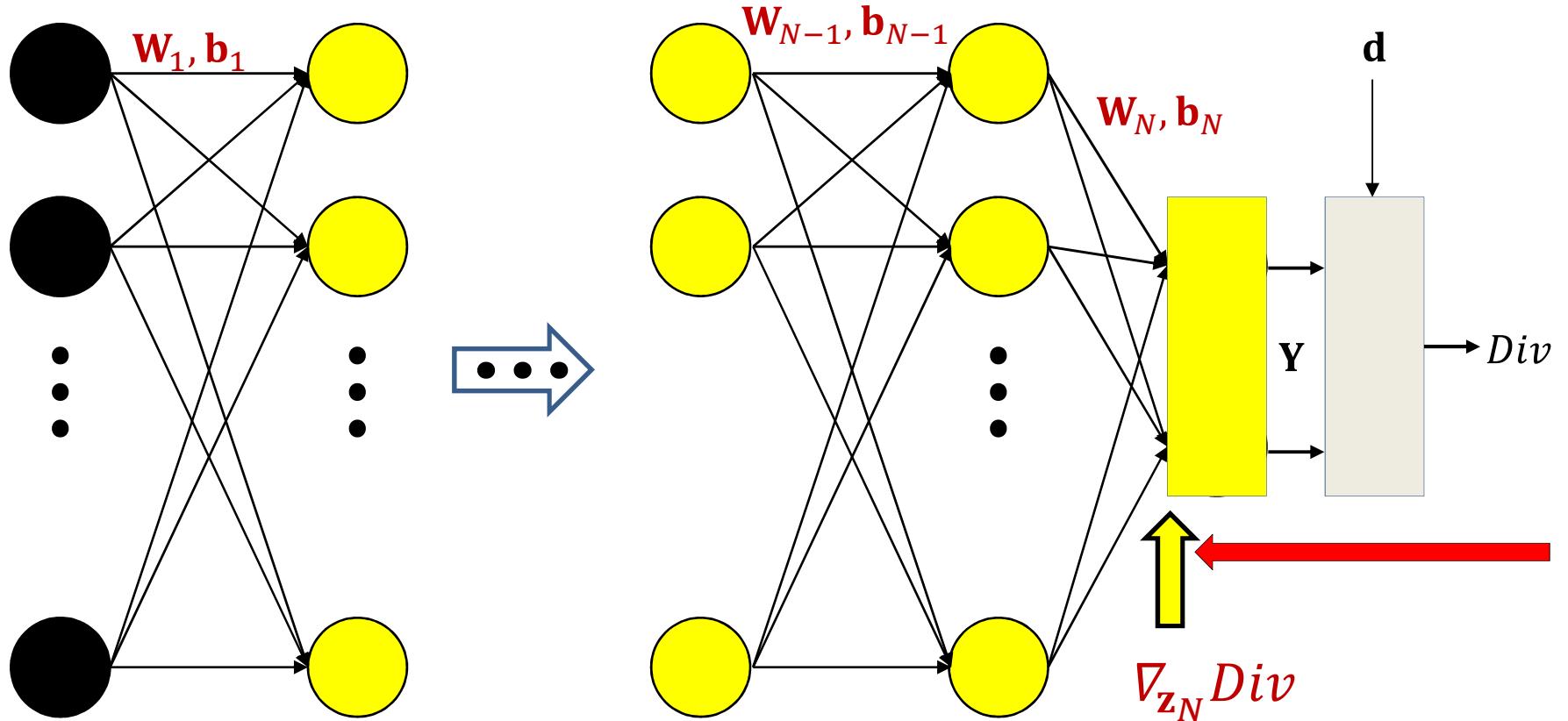
First compute the gradient of the divergence w.r.t.  $Y$ .  
The actual gradient depends on the divergence function.

# The backward pass



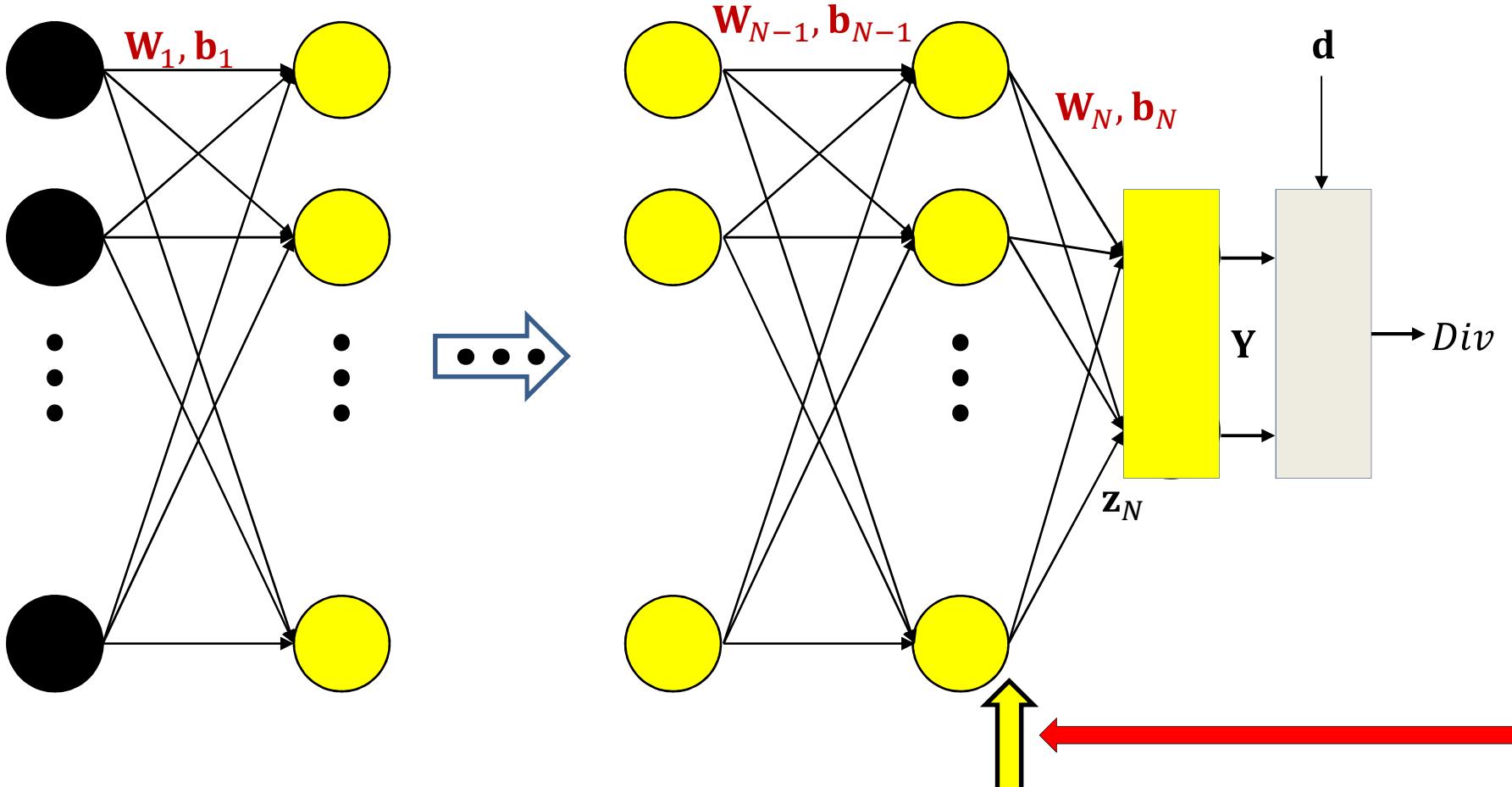
$$\nabla_{\mathbf{z}_N} \text{Div} = \nabla_{\mathbf{Y}} \text{Div} \cdot \nabla_{\mathbf{z}_N} \mathbf{Y}$$

# The backward pass



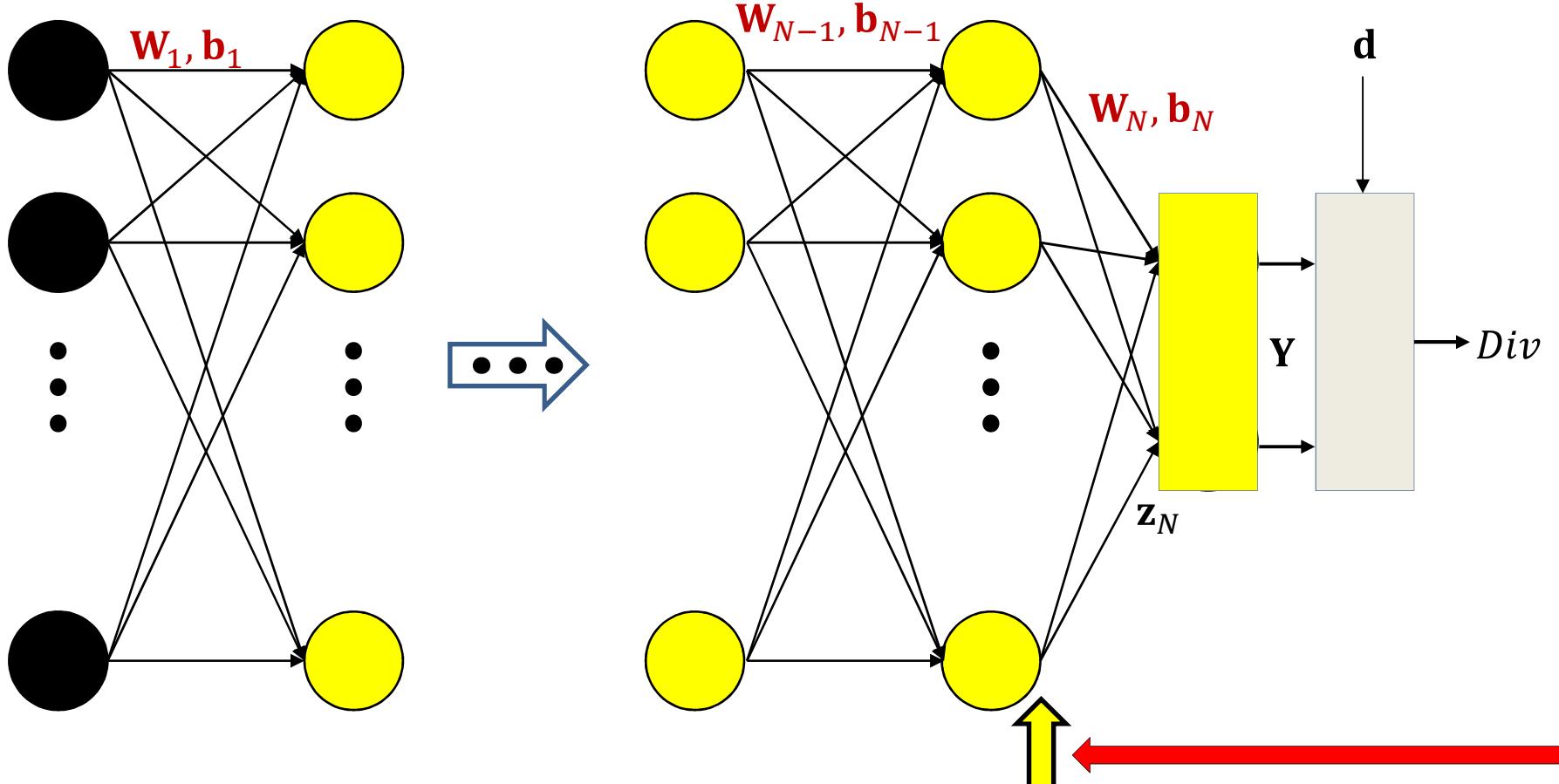
$$\nabla_{\mathbf{z}_N} Div = \nabla_{\mathbf{Y}} Div J_{\mathbf{Y}}(\mathbf{z}_N)$$

# The backward pass



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

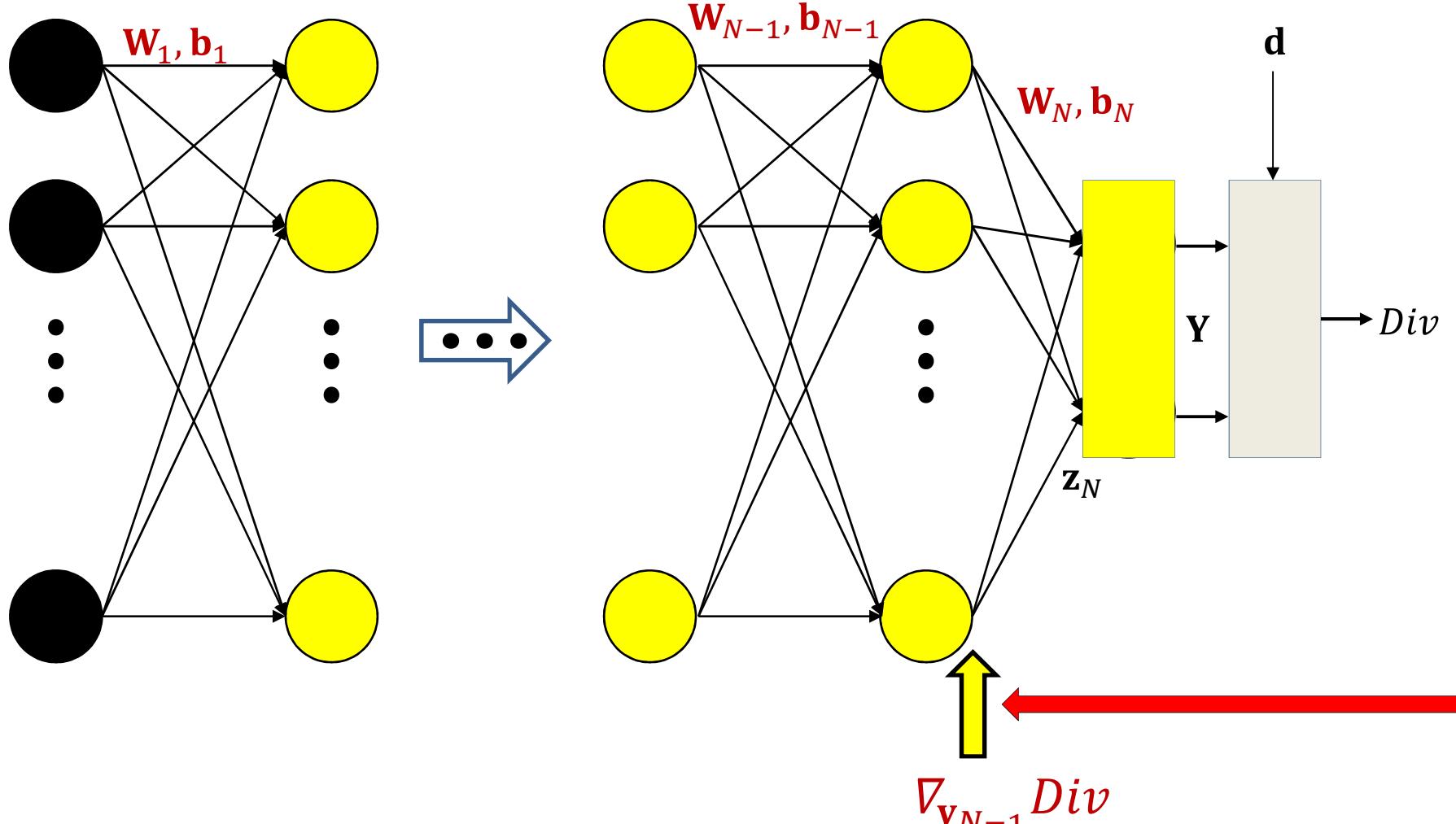
# The backward pass



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

$$\nabla_{y_{N-1}} Div$$

# The backward pass

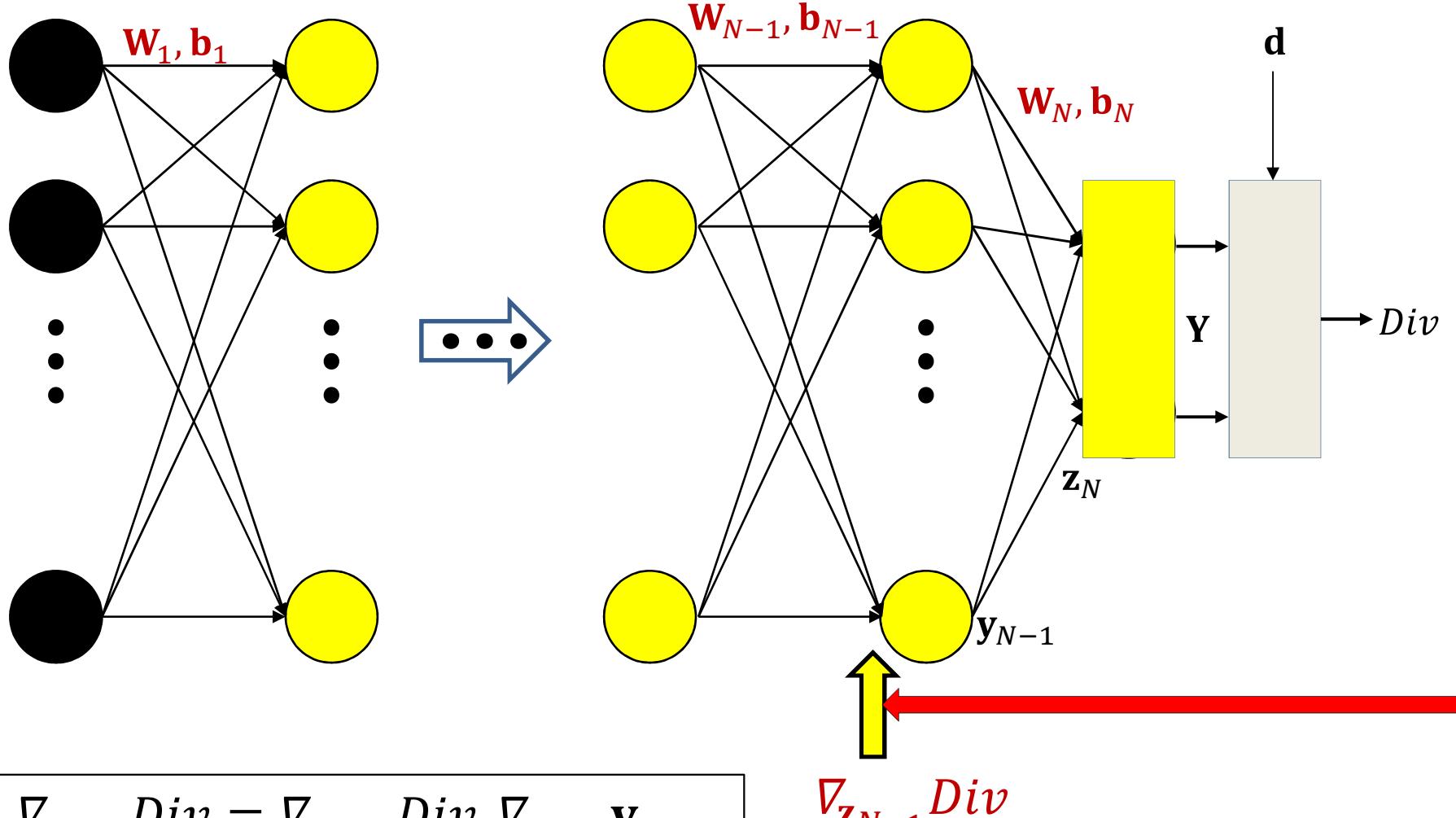


$$\nabla_{y_{N-1}} Div = \nabla_{z_N} Div \mathbf{W}_N$$

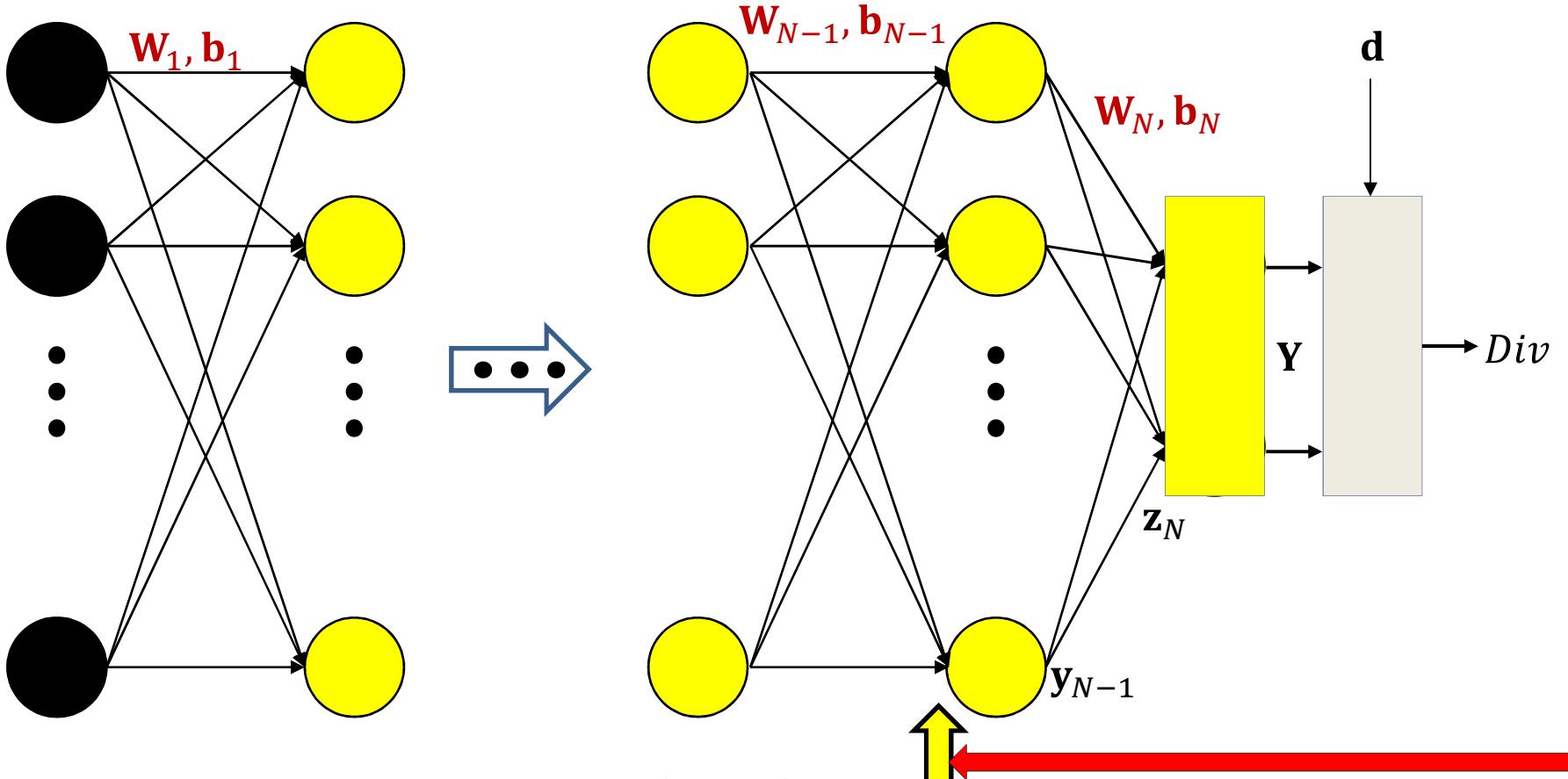
$$\nabla_{\mathbf{W}_N} Div = \mathbf{y}_{N-1} \nabla_{z_N} Div$$

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

# The backward pass



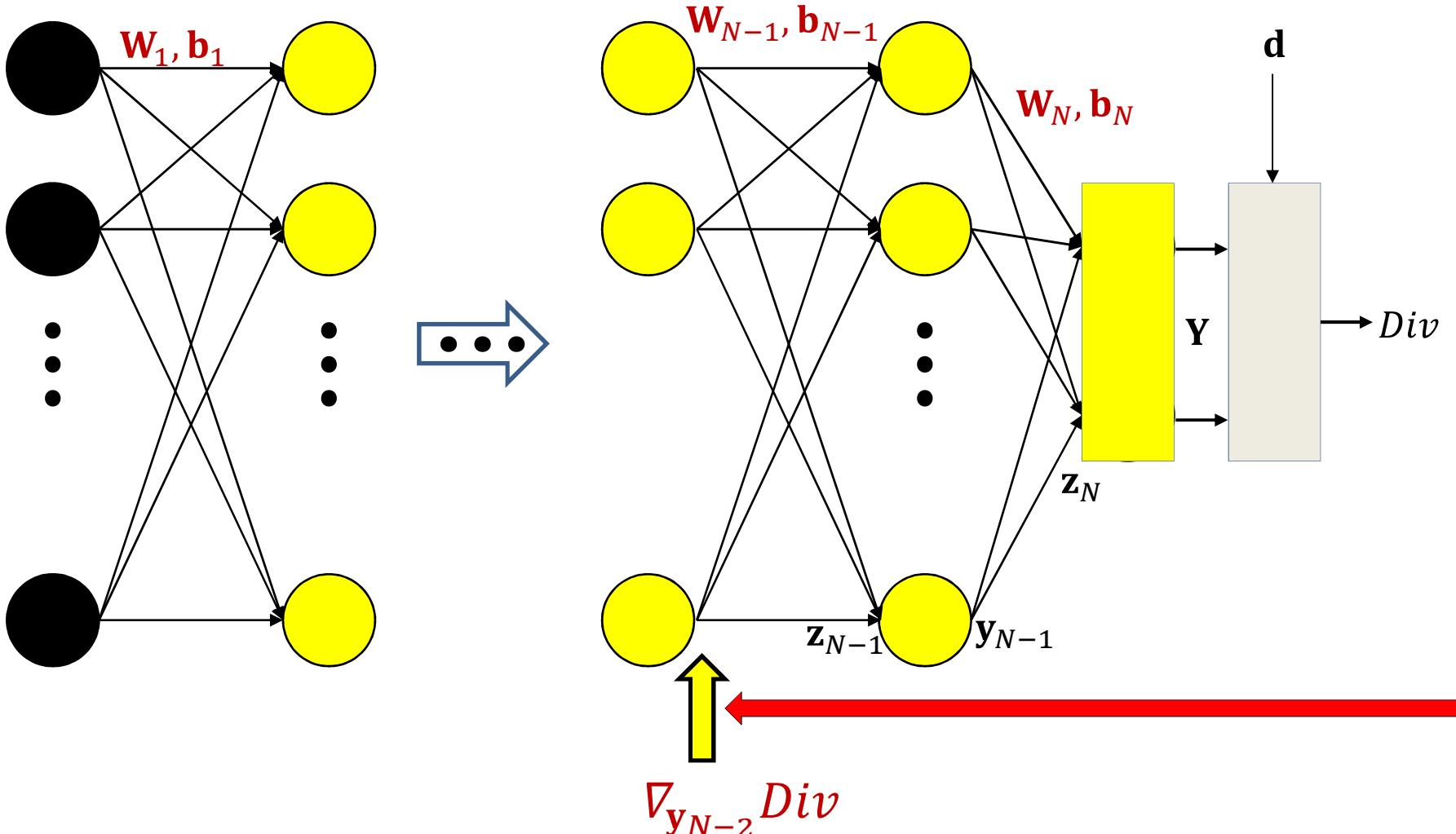
# The backward pass



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

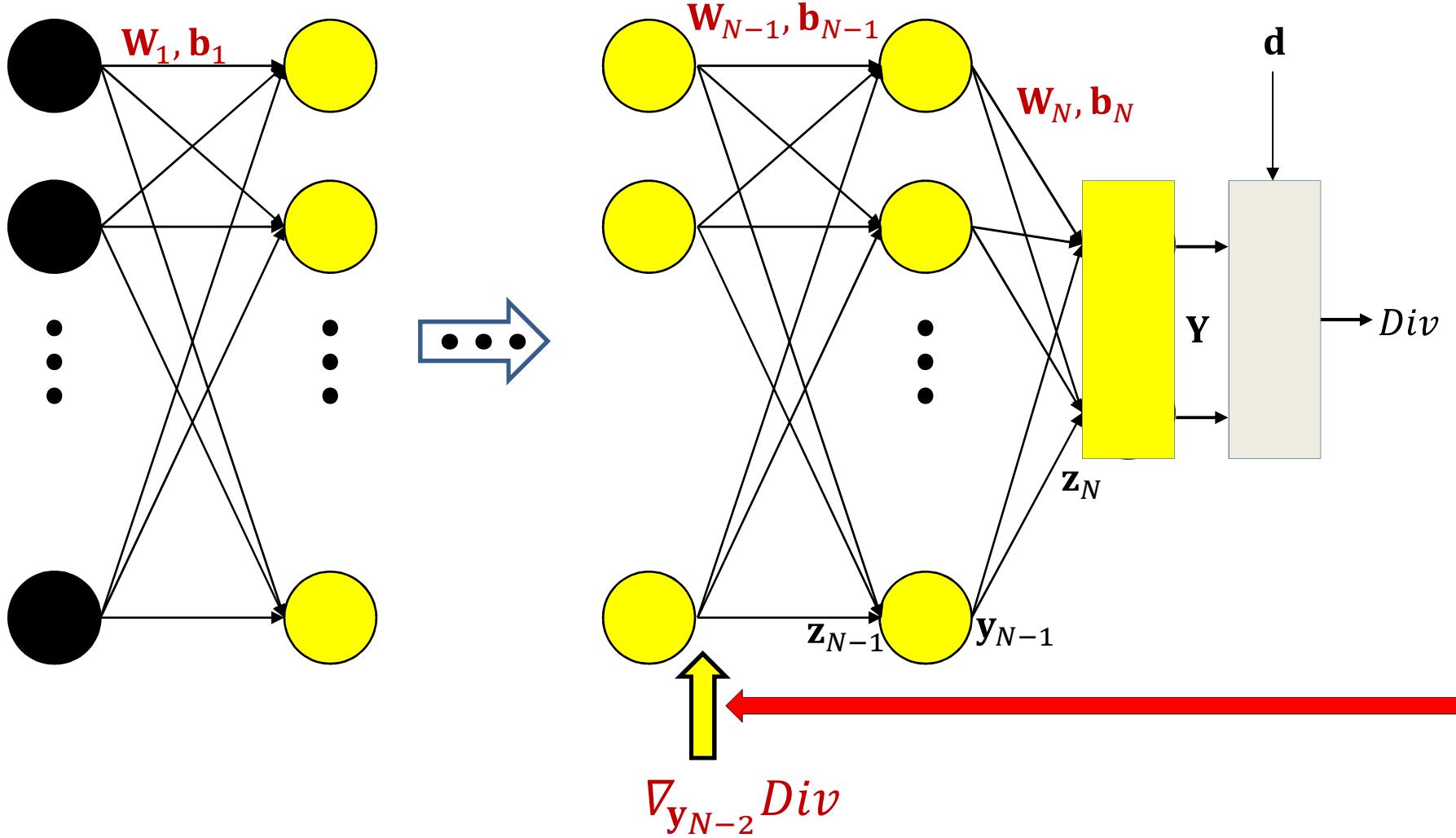
The Jacobian will be a diagonal matrix for scalar activations

# The backward pass



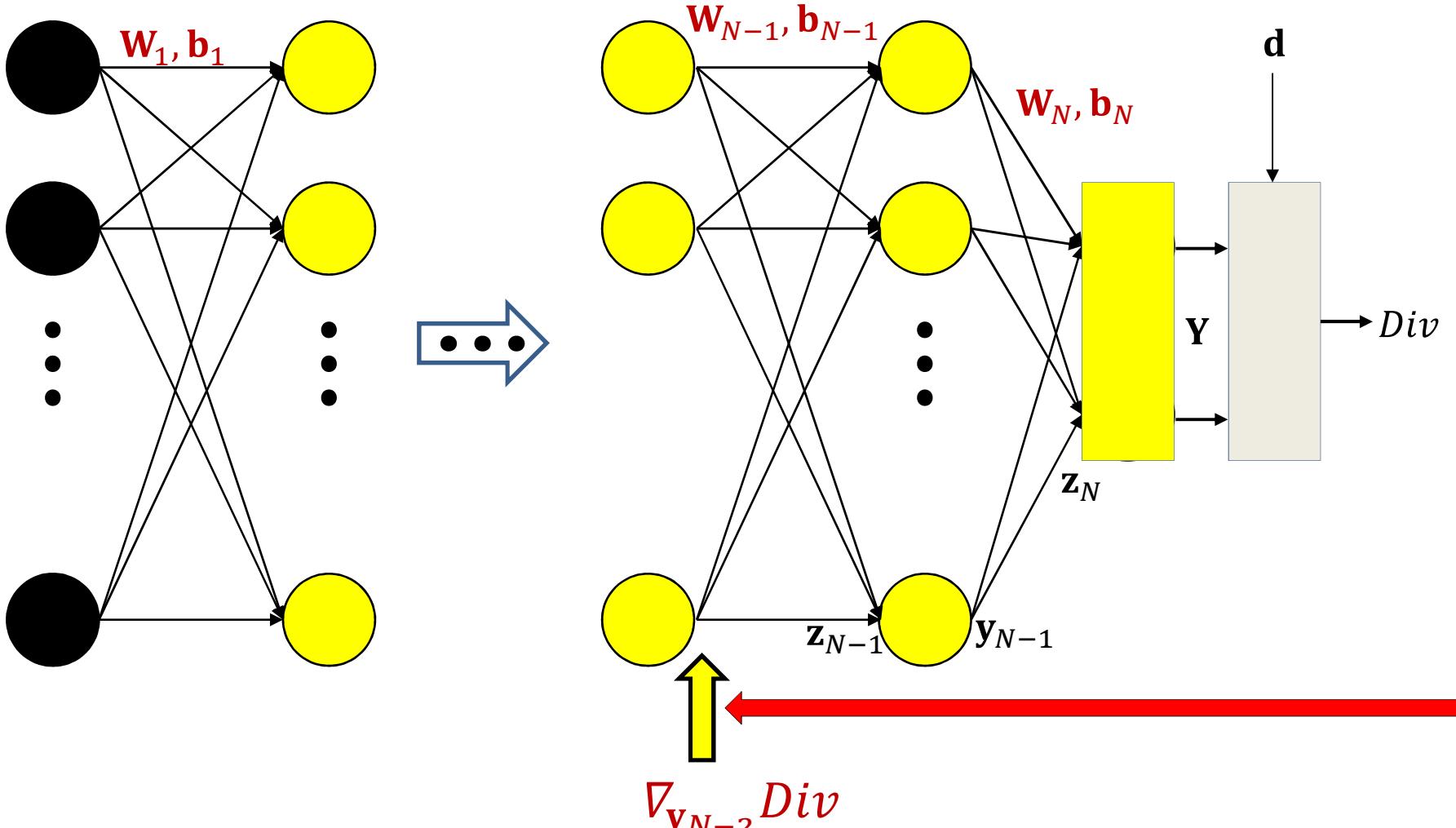
$$\nabla_{\mathbf{y}_{N-2}} Div = \nabla_{\mathbf{z}_{N-1}} Div \cdot \nabla_{\mathbf{y}_{N-2}} \mathbf{z}_{N-1}$$

# The backward pass



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

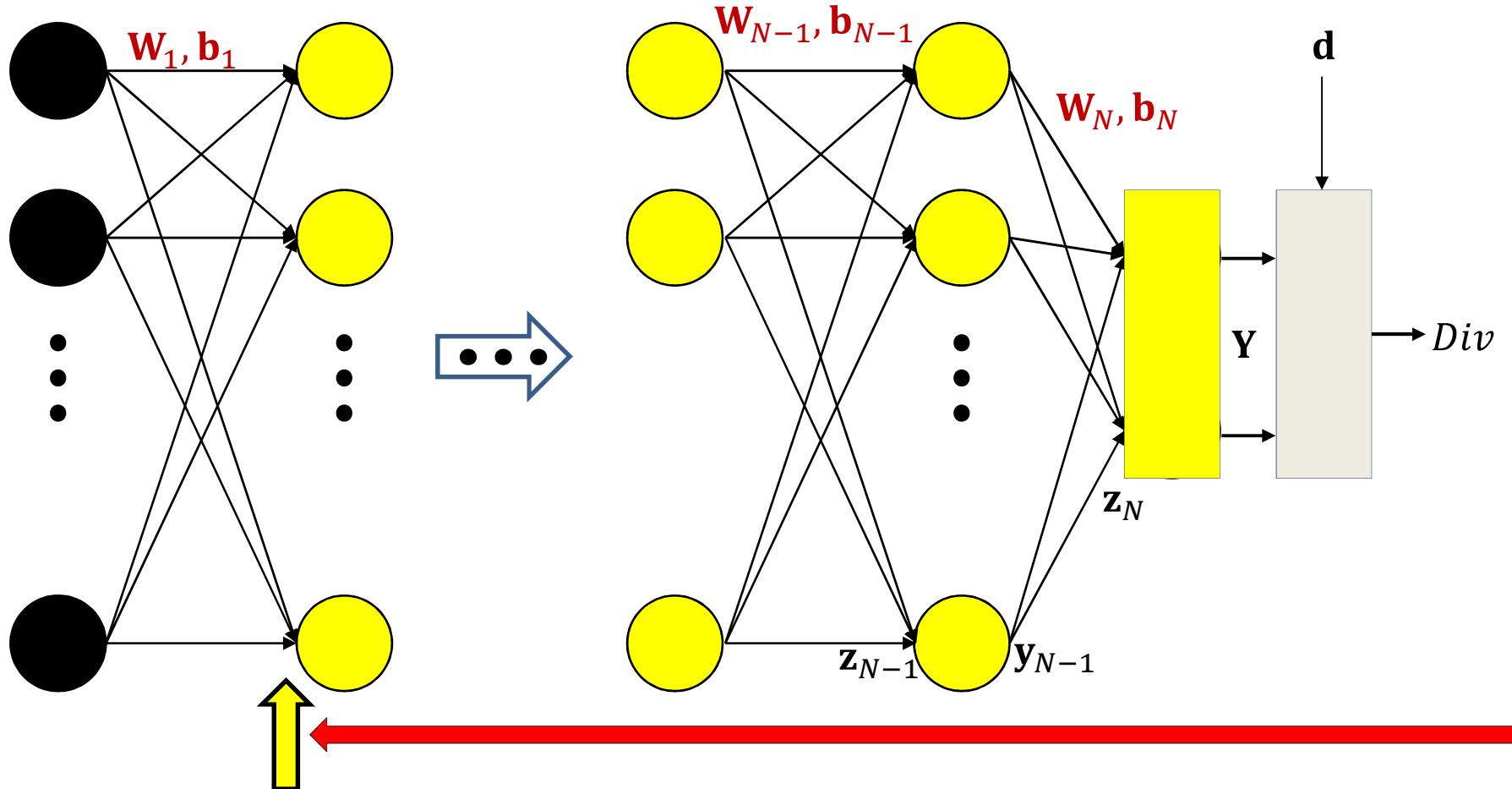
# The backward pass



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

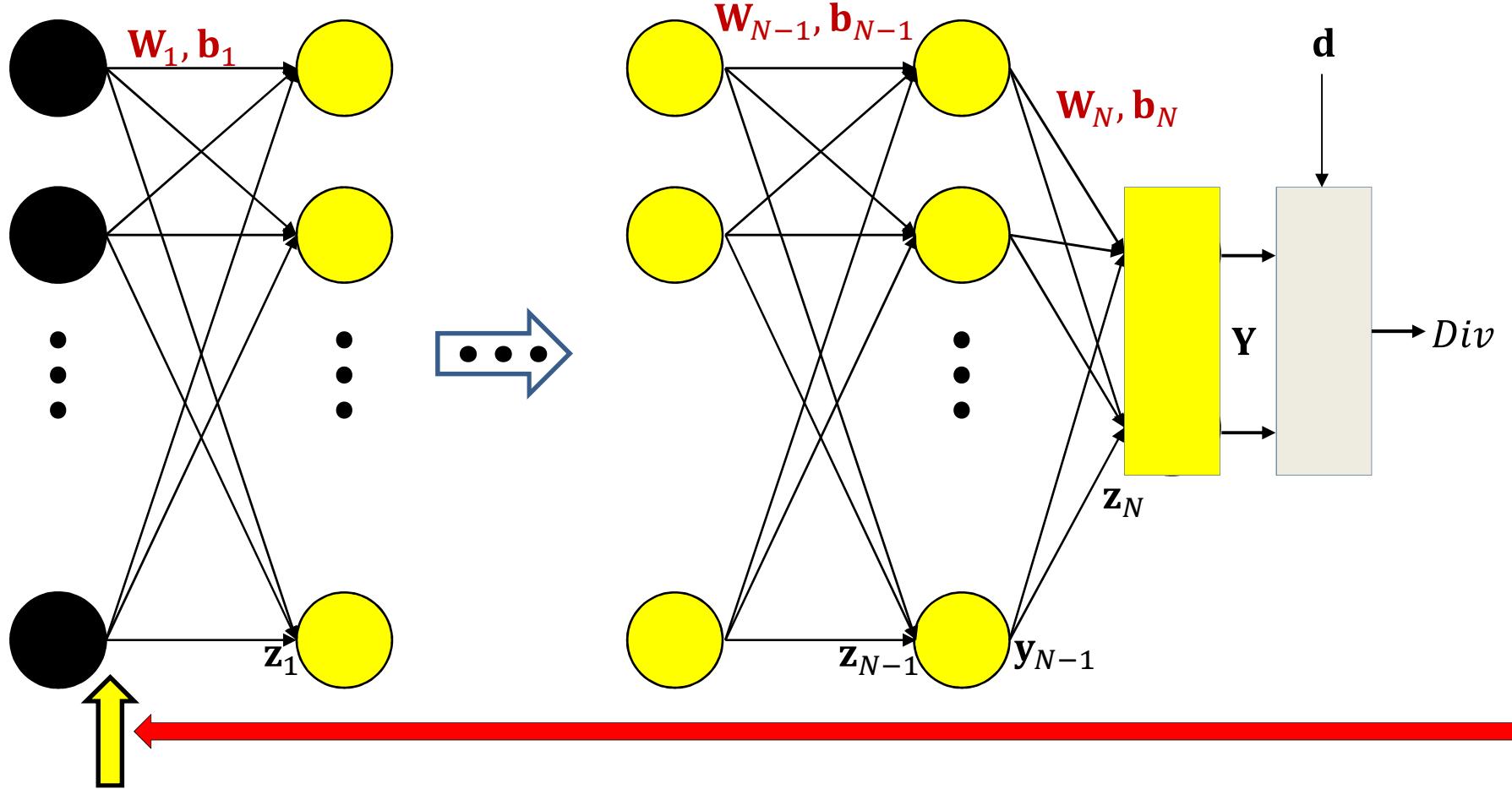
$\nabla_{W_{N-1}} Div = y_{N-2} \nabla_{z_{N-1}} Div$
$\nabla_{b_{N-1}} Div = \nabla_{z_{N-1}} Div$

# 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
  - Recursion:
$$\nabla_{\mathbf{z}_k} Div = \nabla_{\mathbf{y}_k} Div J_{\mathbf{y}_k}(\mathbf{z}_k)$$
$$\nabla_{\mathbf{y}_{k-1}} Div = \nabla_{\mathbf{z}_k} Div \mathbf{W}_k$$
  - 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  $k = N$  down to 1:
  - Compute  $J_{\mathbf{y}_k}(\mathbf{z}_k)$ 
    - Will require intermediate values computed in the forward pass
  - Recursion:

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  $N$ :

– Recursion:

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

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

- Output:

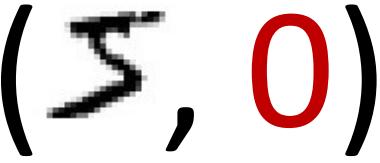
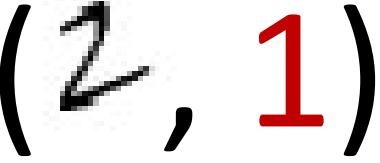
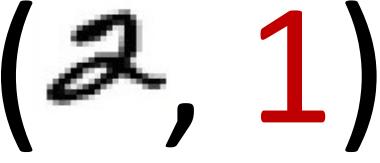
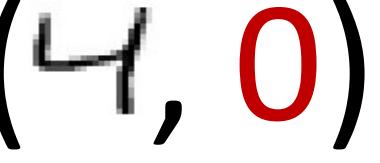
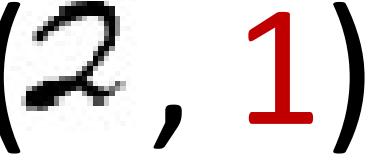
$$\mathbf{Y} = \mathbf{y}_N$$

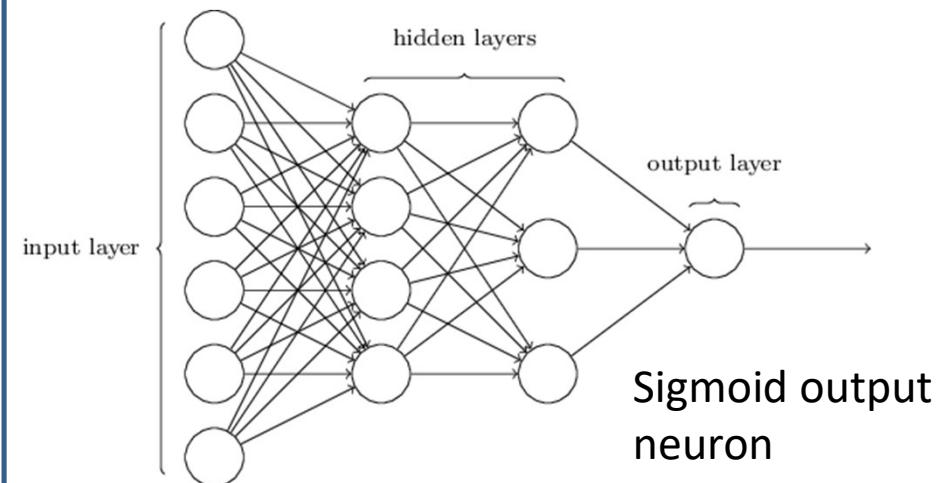
# Neural network training algorithm

- 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:
  - $Err = 0$
  - For all  $k$ , initialize  $\nabla_{\mathbf{W}_k} Err = 0, \nabla_{\mathbf{b}_k} Err = 0$
  - For all  $t = 1:T$ 
    - Forward pass : Compute
      - Output  $\mathbf{Y}(X_t)$
      - Divergence  $\mathbf{Div}(\mathbf{Y}_t, \mathbf{d}_t)$
      - $Err += \mathbf{Div}(\mathbf{Y}_t, \mathbf{d}_t)$
    - Backward pass: For all  $k$  compute:
      - $\nabla_{\mathbf{y}_k} \mathbf{Div} = \nabla_{\mathbf{z}_{k+1}} \mathbf{Div} \mathbf{W}_k$
      - $\nabla_{\mathbf{z}_k} \mathbf{Div} = \nabla_{\mathbf{y}_k} \mathbf{Div} J_{\mathbf{y}_k}(\mathbf{z}_k)$
      - $\nabla_{\mathbf{W}_k} \mathbf{Div}(\mathbf{Y}_t, \mathbf{d}_t); \nabla_{\mathbf{b}_k} \mathbf{Div}(\mathbf{Y}_t, \mathbf{d}_t)$
      - $\nabla_{\mathbf{W}_k} Err += \nabla_{\mathbf{W}_k} \mathbf{Div}(\mathbf{Y}_t, \mathbf{d}_t); \nabla_{\mathbf{b}_k} Err += \nabla_{\mathbf{b}_k} \mathbf{Div}(\mathbf{Y}_t, \mathbf{d}_t)$
  - For all  $k$ , update:
$$\mathbf{W}_k = \mathbf{W}_k - \frac{\eta}{T} (\nabla_{\mathbf{W}_k} Err)^T; \quad \mathbf{b}_k = \mathbf{b}_k - \frac{\eta}{T} (\nabla_{\mathbf{b}_k} Err)^T$$
- Until  $Err$  has converged

# Setting up for digit recognition

Training data

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

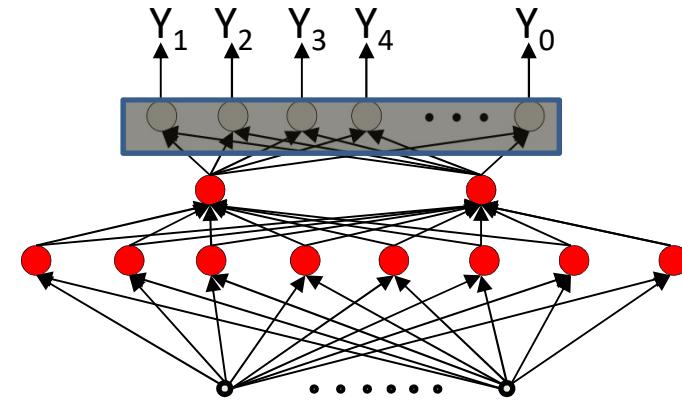


- Simple Problem: Recognizing “2” or “not 2”
- Single output with sigmoid activation
  - $Y \in (0,1)$
  - $d$  is either 0 or 1
- Use KL divergence
- Backpropagation to learn network parameters

# Recognizing the digit

Training data

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



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

# Issues

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

# Next up

- Convergence and generalization