Neural Networks
Learning the network: Part 2

11-785, Fall 2017

Lecture 4

Quiz 2

- A multi-class classifier can use log(C) output neurons, each predicting either 0 or 1 to encode C classes. For which of the following reasons is this setup is not typical:
  - It is possible for the classifier to produce a code that does not correspond to any class in this scheme.
  - It is an inefficient representation of the classes in terms of the number of output neurons.
  - It implicitly assumes that some classes are more similar to each other than other classes.
  - It is more computationally expensive to compute the gradient in this scheme.
- In the empirical risk minimization framework, the function which measures the error (divergence function) should always be non-negative (T/F)?

Quiz 2

- If the perceptron rule is used to train a multi-layer perceptron network, the training computation scales _____ with the number of data points.
- The perceptron learning rule will find the separating hyperplane with the largest margin(T/F)?

Quiz 2

- Which of the following is true of the MADALINE learning algorithm (select all that apply):
  - It computes the gradient of the network with respect to all of the weights in the network.
  - It greedily assigns the desired output label to a hidden node in the network during training.
  - It updates the weights for every training example.
  - To update the weights for a neuron the weighted sum of the inputs, rather than the output of the activation function, is compared to the desired label.
- For a single perceptron with a threshold activation function, the ADALINE learning rule _____ (select all that apply)
  - moves the weights in the direction of the negative gradient of the mean squared error.
  - is equivalent to the perceptron learning rule.
  - is equivalent to the generalized delta rule.
  - enables learning in a network with multiple layers.

Quiz 2

- Which of the following activation functions will have the largest magnitude gradient as the input to the activation function increases from 0 in the positive direction:
  - Threshold / Sigmoid / Softplus
- If the empirical risk of a neural network is 0 then (select all that apply):
  - The weights of the network will not change for any of the learning algorithms we have discussed.
  - The network has learned the target function.
  - The network will predict the correct class for *all* data points that it has not seen during training.
  - The network will predict the correct class for *all* data points that is has seen during training.

Quiz 2

- Which of the following are advantages of using a sigmoid activation function for all nodes in a neural network?
  - The output of each node has a probabilistic interpretation.
  - The gradient of the function computed by the network with respect to the weights of a neuron is smaller when the input to the neuron is near the mean input to the neuron.
  - By scaling the weights, a learning algorithm can change the output of a neuron to be more linear/less linear with respect to the input.
  - The error signal from the output layer can be used to greedily adjust weights throughout the network.

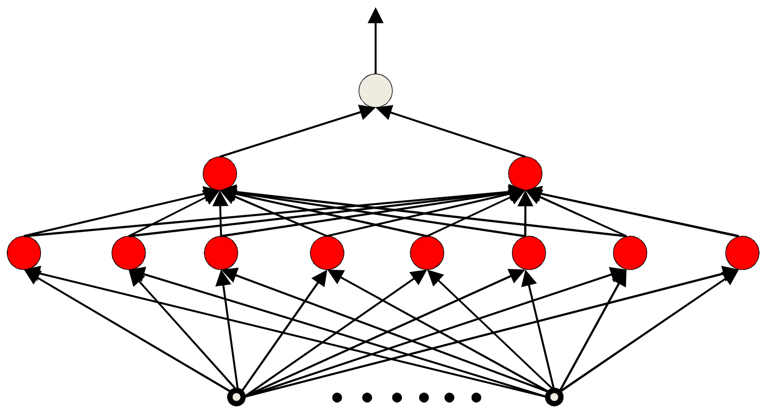
- If we use the generalized delta rule to update the weights of an output neuron, then the sigmoid activation function is less sensitive to outliers than the identity activation function (T/F)?

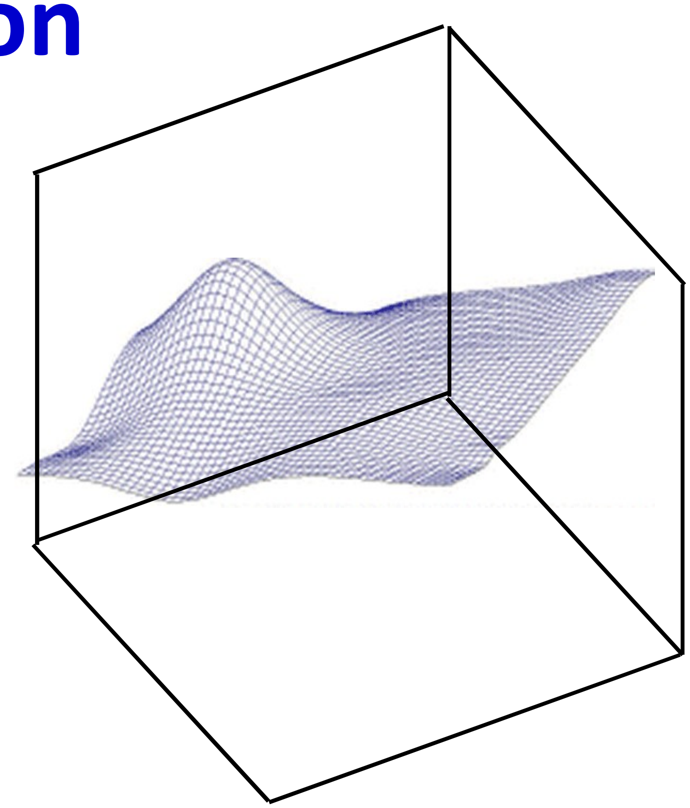
Design exercise

- Input: Binary coded number
- Output: One-hot vector

- Input units?
- Output units?
- Architecture?
- Activations?

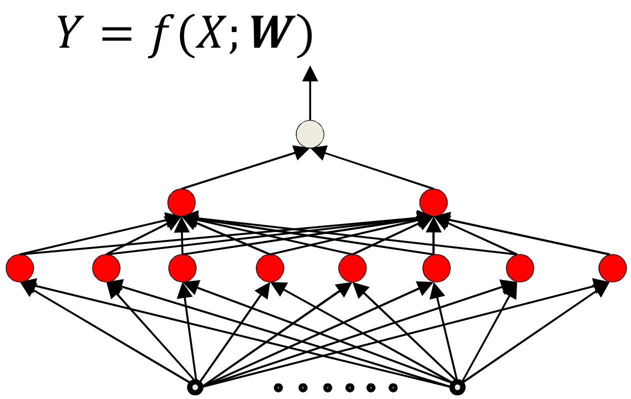
Recap: The MLP *can* represent any function

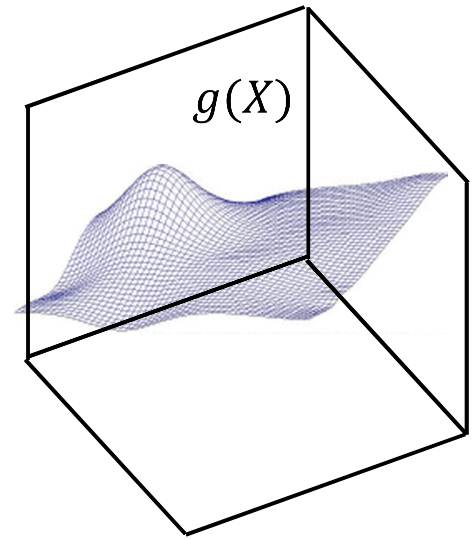




- The MLP *can be constructed* to represent anything
- But *how* do we construct it?

Recap: How to learn the function



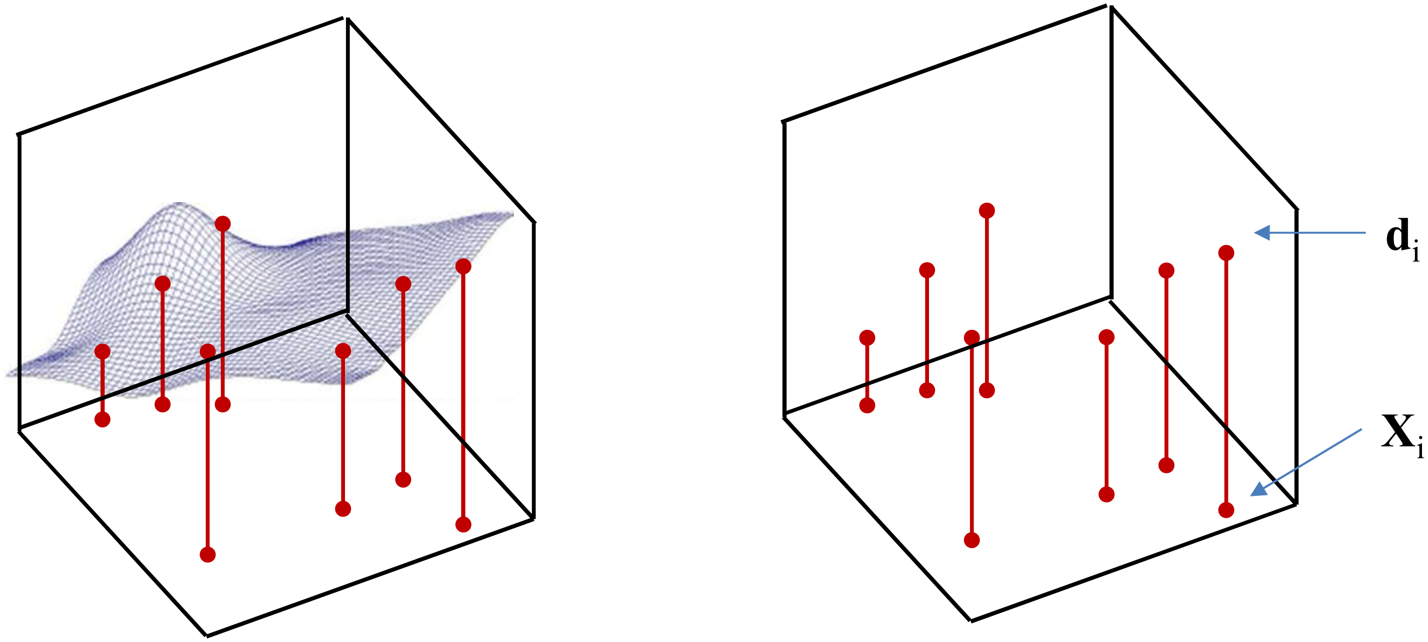


- By minimizing expected error

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}} \int_X div\big(f(X;W), g(X)\big)P(X)dX$$

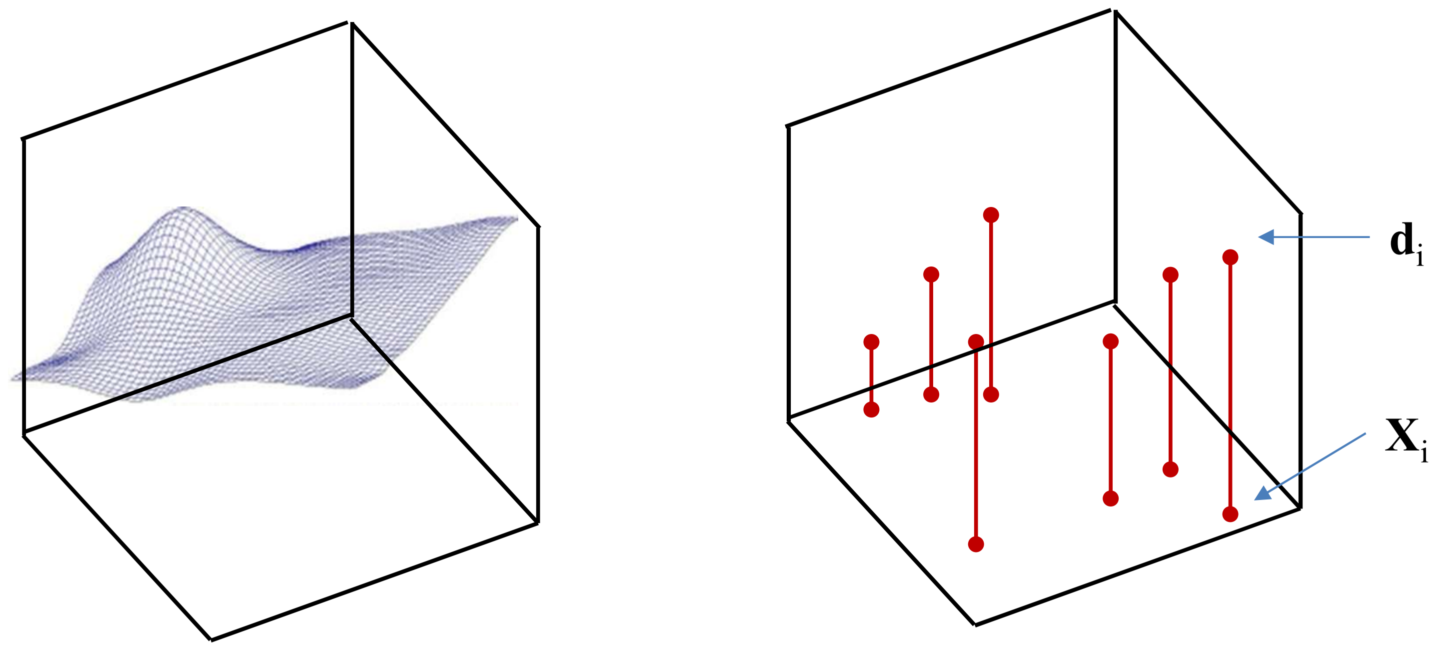
$$= \underset{W}{\operatorname{argmin}}\, E\big[div\big(f(X;W), g(X)\big)\big]$$

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 $d_i = g(X_i) + noise$
  - 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

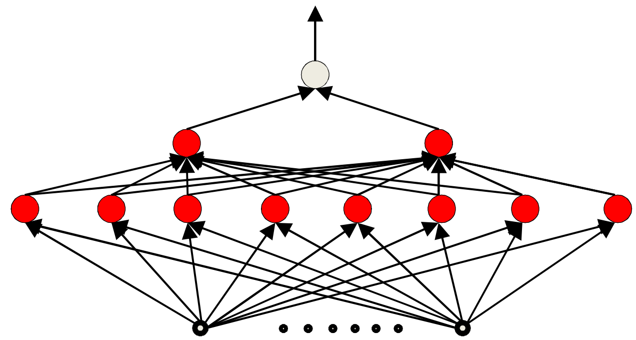
$$E\left[div\left(f(X;W),g(X)\right)\right] = \int_X div\left(f(X;W),g(X)\right)P(X)dX$$

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

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

Empirical Risk Minimization

$Y = f(X; \boldsymbol{W})$



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

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

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

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}}\, Err(W)$$

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

Problem Statement

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$

- Minimize the following function

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

  w.r.t W

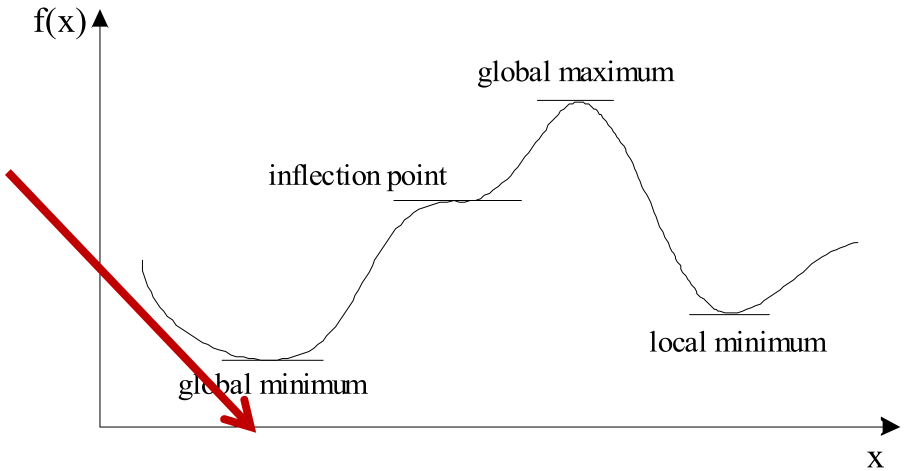
- This is problem of function minimization
  - An instance of optimization

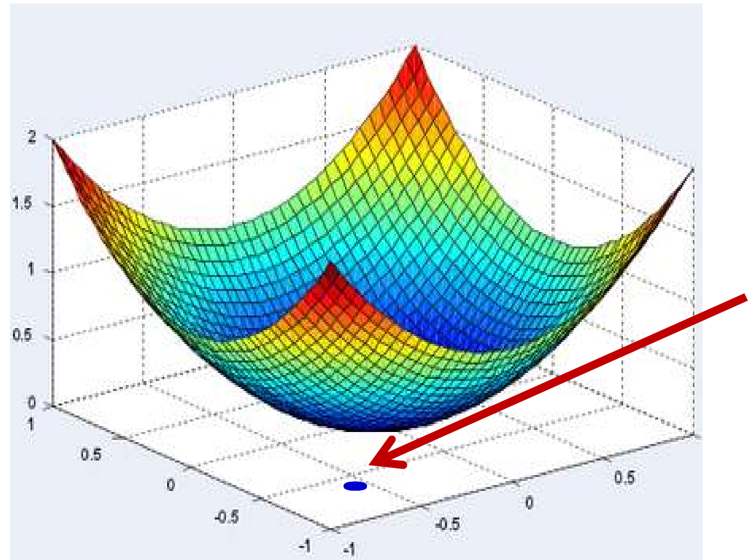
- **A CRASH COURSE ON FUNCTION OPTIMIZATION**

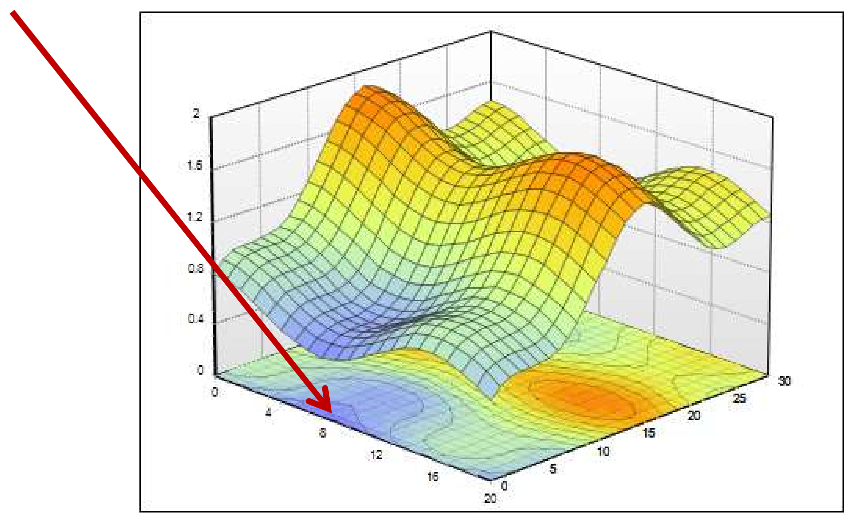
Caveat about following slides

- The following slides speak of optimizing a function w.r.t a variable “x”
- This is only mathematical notation. In our actual network optimization problem we would be optimizing w.r.t. network weights “w”
- To reiterate – “x” in the slides represents the variable that we’re optimizing a function over and not the input to a neural network
- **Do not get confused!**

The problem of optimization

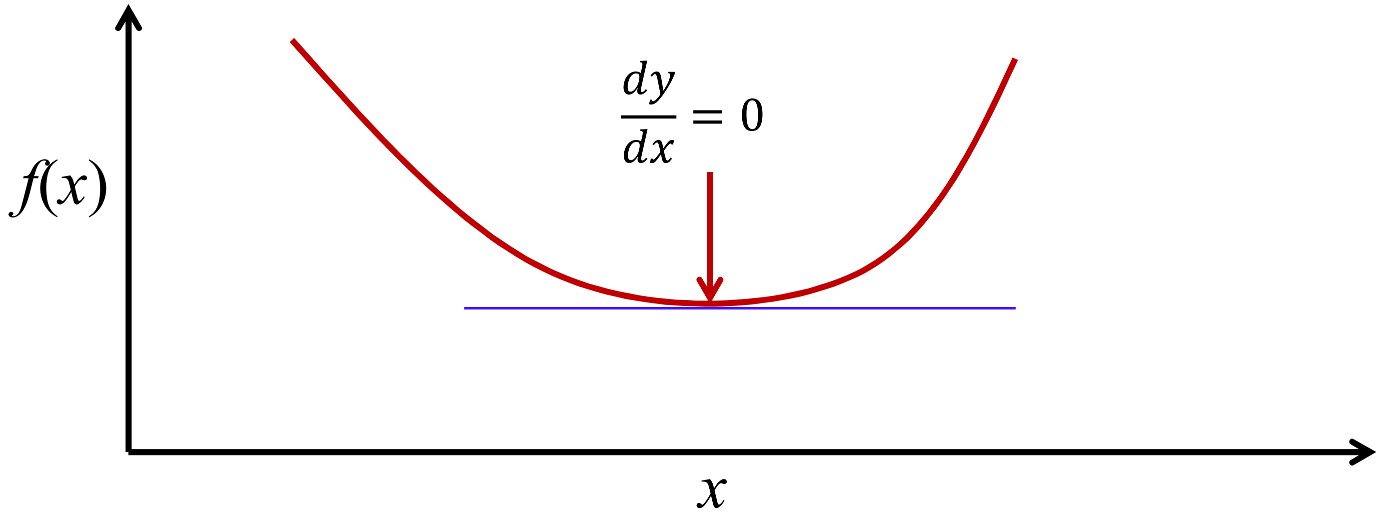






- General problem of optimization: find the value of $\boldsymbol{x}$ where $\mathbf{f}(\boldsymbol{x})$ is minimum

Finding the minimum of a function

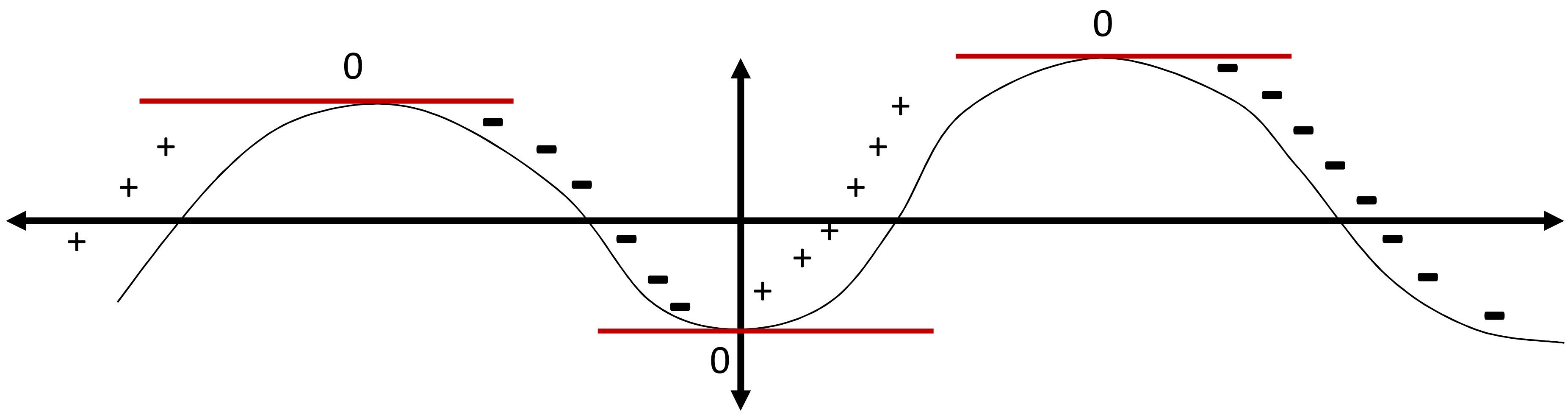


- Find the value x at which $f'(x) = 0$
  - Solve

$$\frac{df(x)}{dx} = 0$$

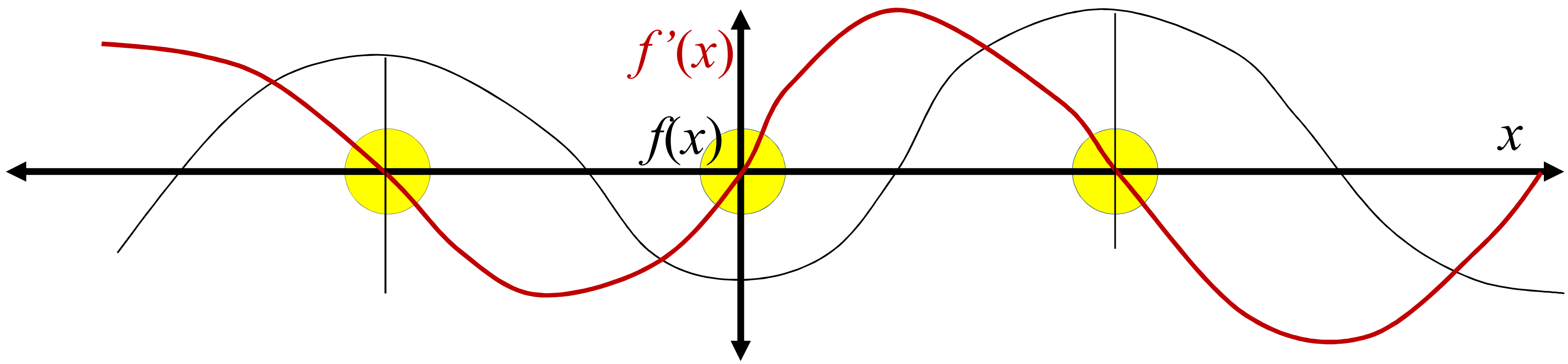
- The solution is a “turning point”
  - Derivatives go from positive to negative or vice versa at this point
- But is it a minimum?

Turning Points



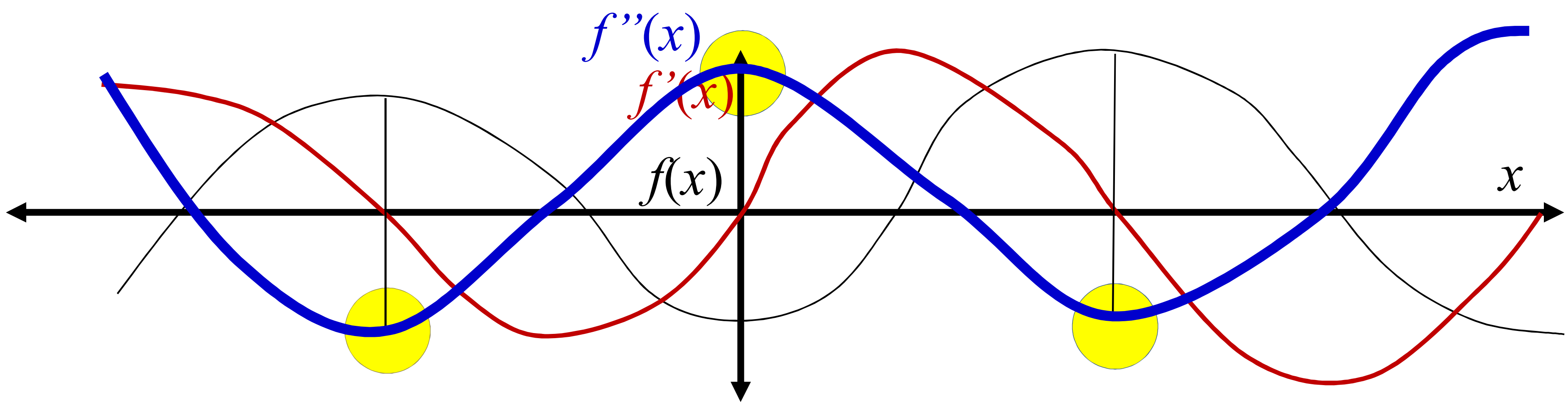
- Both *maxima* and *minima* have zero derivative
- Both are turning points

Derivatives of a curve



- Both *maxima* and *minima* are turning points
- Both *maxima* and *minima* have zero derivative

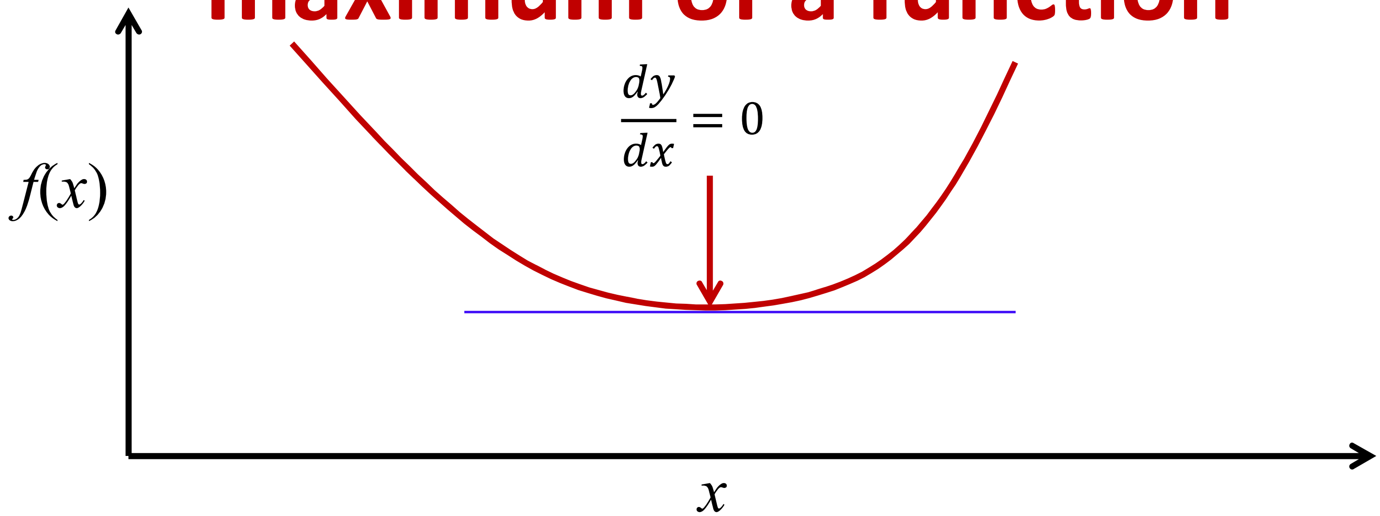
Derivative of the derivative of the curve



- Both *maxima* and *minima* are turning points
- Both *maxima* and *minima* have zero derivative

- The *second derivative* $f''(x)$ is –ve at maxima and +ve at minima!

Soln: Finding the minimum or maximum of a function



- Find the value x at which $f'(x) = 0$: Solve

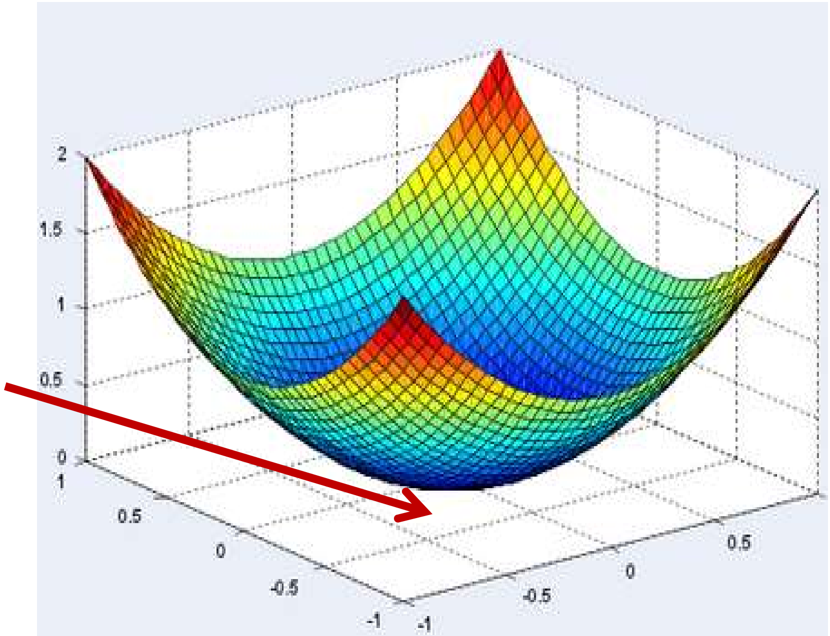
$$\frac{df(x)}{dx} = 0$$

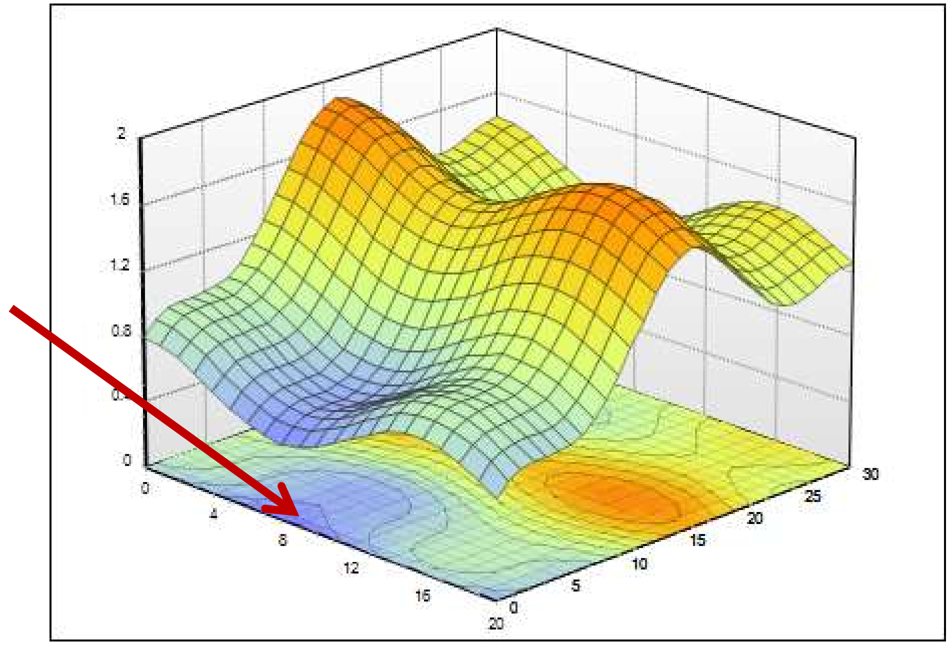
- The solution x_{soln} is a turning point
- Check the double derivative at x_{soln} : compute

$$f''(x_{soln}) = \frac{df'(x_{soln})}{dx}$$

- If $f''(x_{soln})$ is positive x_{soln} is a minimum, otherwise it is a maximum

What about functions of multiple variables?

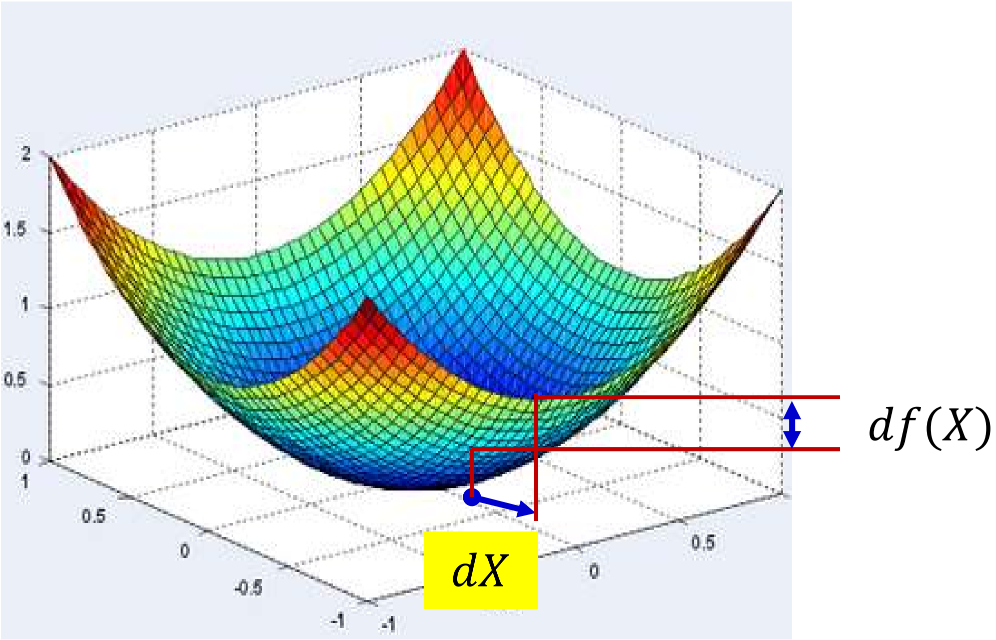




- The optimum point is still "turning" point
  - Shifting in any direction will increase the value
  - For smooth functions, miniscule shifts will not result in any change at all
- We must find a point where shifting in any direction by a microscopic amount will not change the value of the function

A brief note on derivatives of multivariate functions

The *Gradient* of a scalar function

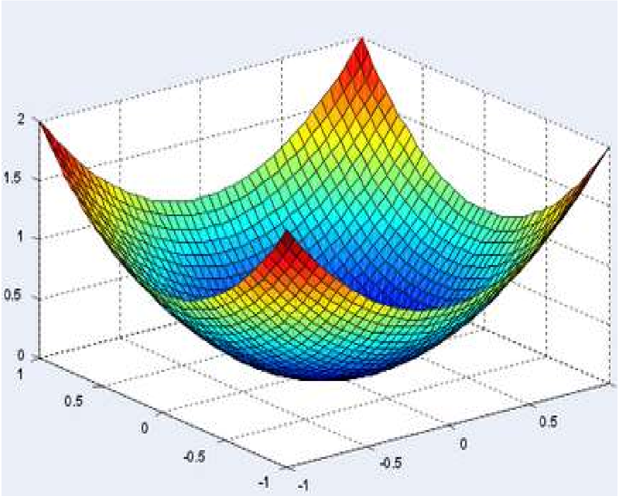


- The *Gradient* $\nabla f(X)$ of a scalar function $f(X)$ of a multi-variate input X is a multiplicative factor that gives us the change in $f(X)$ for tiny variations in X

$$df(X) = \nabla f(X)dX$$

Gradients of scalar functions with multi-variate inputs

- Consider $f(X) = f(x_1, x_2, \ldots, x_n)$

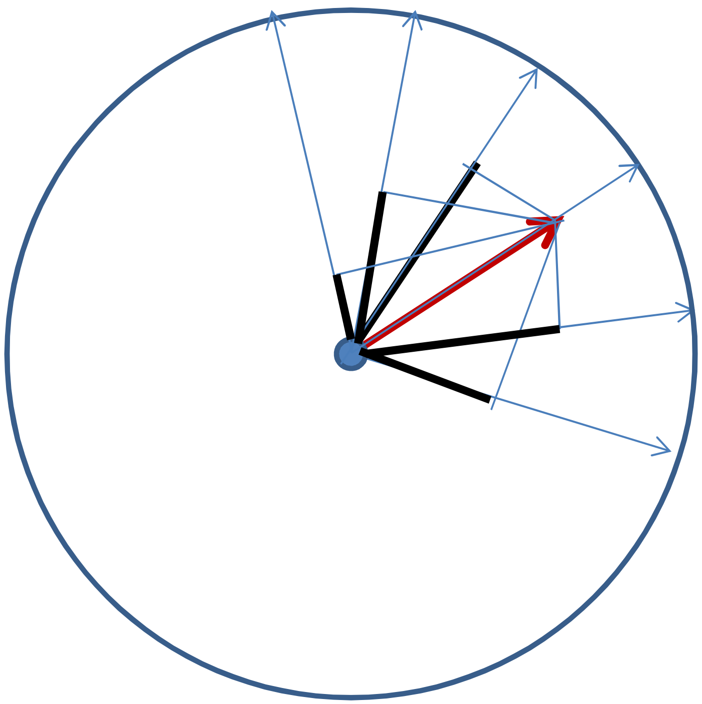


$$\nabla f(X) = \begin{bmatrix} \frac{\partial f(X)}{\partial x_1} & \frac{\partial f(X)}{\partial x_2} & \cdots & \frac{\partial f(X)}{\partial x_n} \end{bmatrix}$$

- Check:

$$df(X) = \nabla f(X) dX$$
$$= \frac{\partial f(X)}{\partial x_1} dx_1 + \frac{\partial f(X)}{\partial x_2} dx_2 + \cdots + \frac{\partial f(X)}{\partial x_n} dx_n$$

A well-known vector property



$$\mathbf{u}^{\mathrm{T}}\mathbf{v} = |\mathbf{u}||\mathbf{v}|cos\theta$$

- The inner product between two vectors of fixed lengths is maximum when the two vectors are aligned
  - i.e. when $\theta = 0$

Properties of Gradient

- $df(X) = \nabla f(X) dX$
  - The inner product between $\nabla f(X)$ and dX

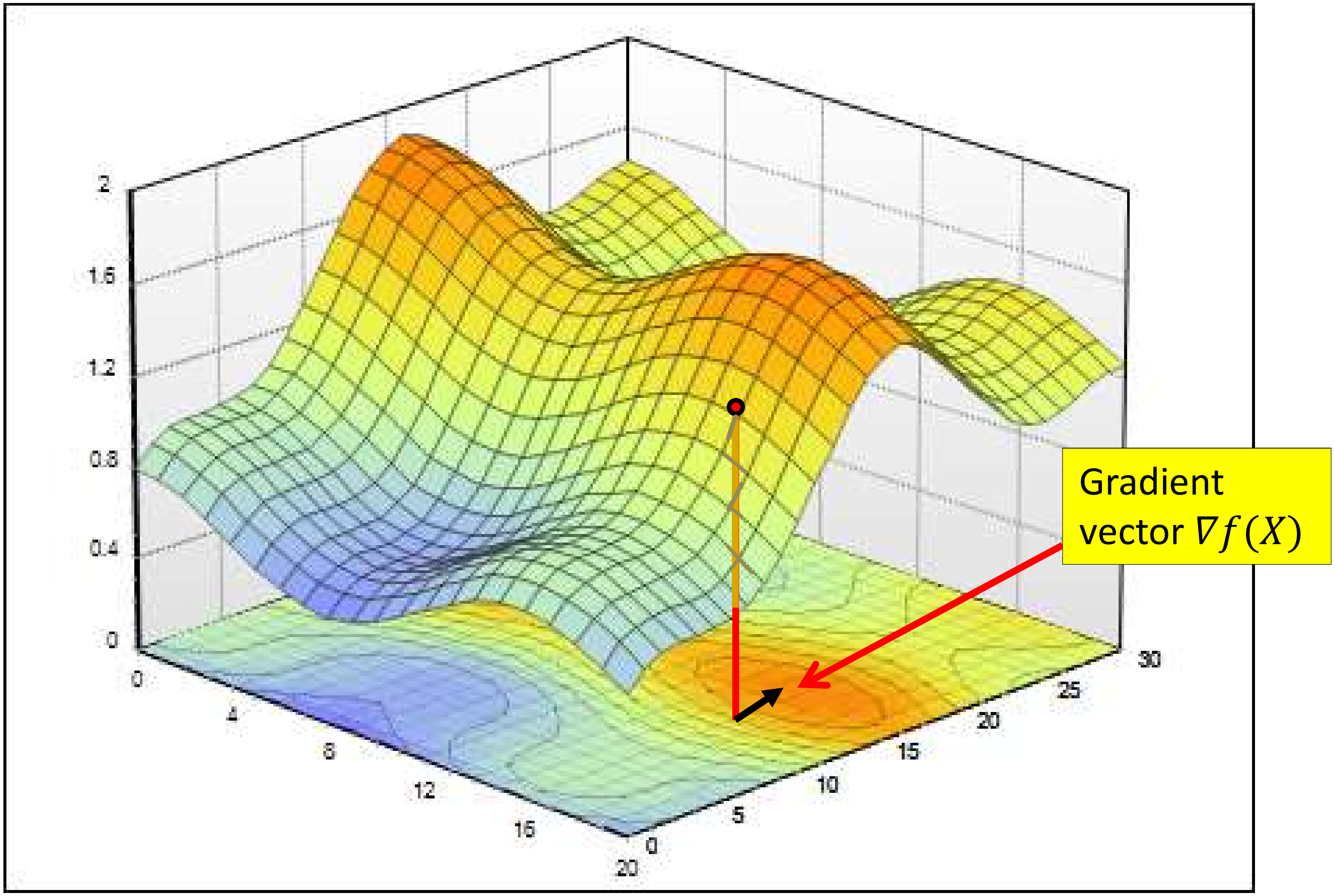
- Fixing the length of dX
  - E.g. $|dX| = 1$

- $df(X)$ is max if dX is aligned with $\nabla f(X)$
  - $\angle \nabla f(X), dX = 0$
  - The function $f(X)$ increases most rapidly if the input increment dX is perfectly aligned to $\nabla f(X)$

- The gradient is the direction of fastest increase in $f(X)$

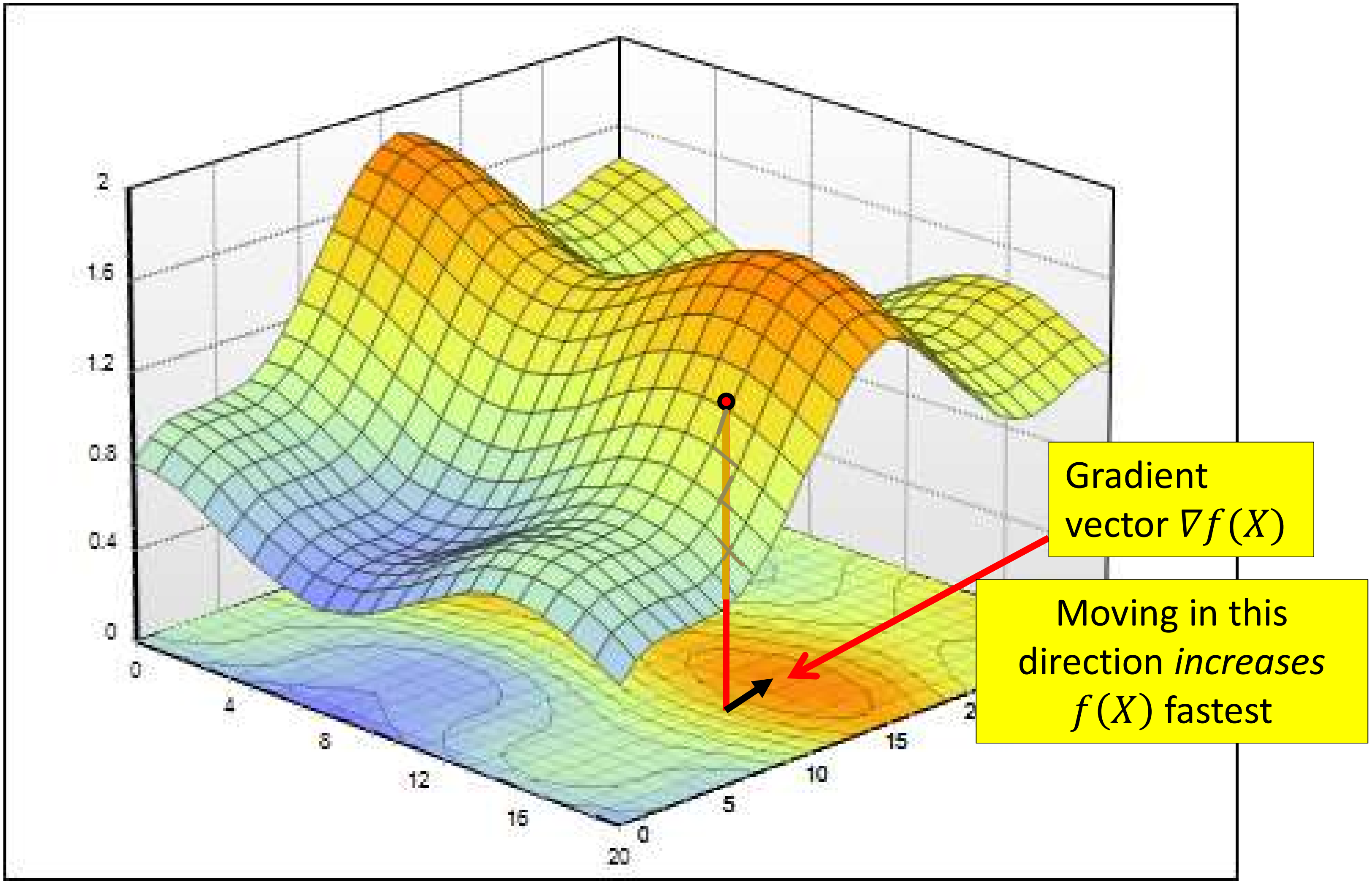
Some sloppy maths here, with apology – comparing row and column vectors

Gradient



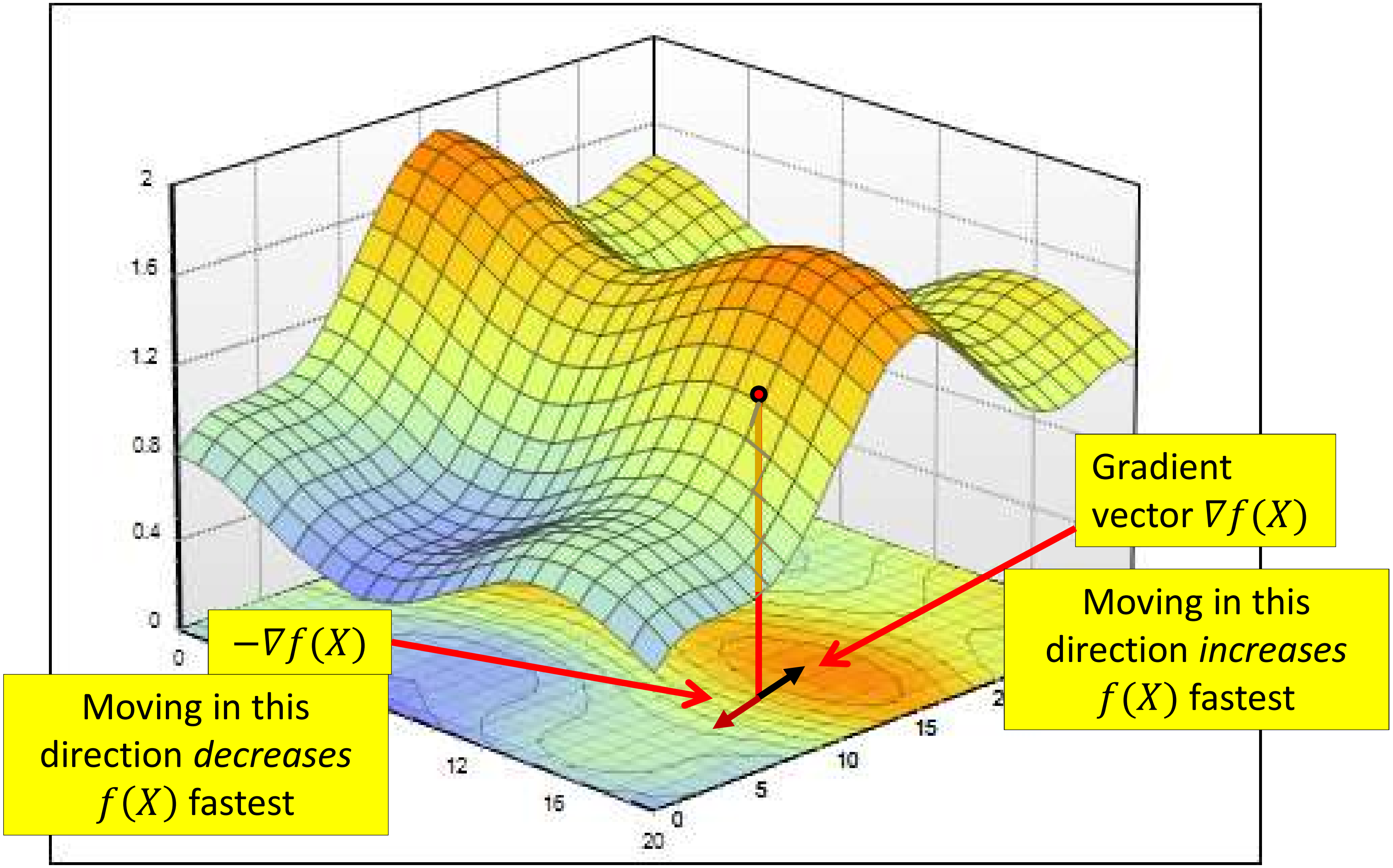
2
1.6
1.2
0.8
0.4
0
0
4
8
12
16
20
0
5
10
15
20
25
30
Gradient vector $\nabla f(X)$

Gradient



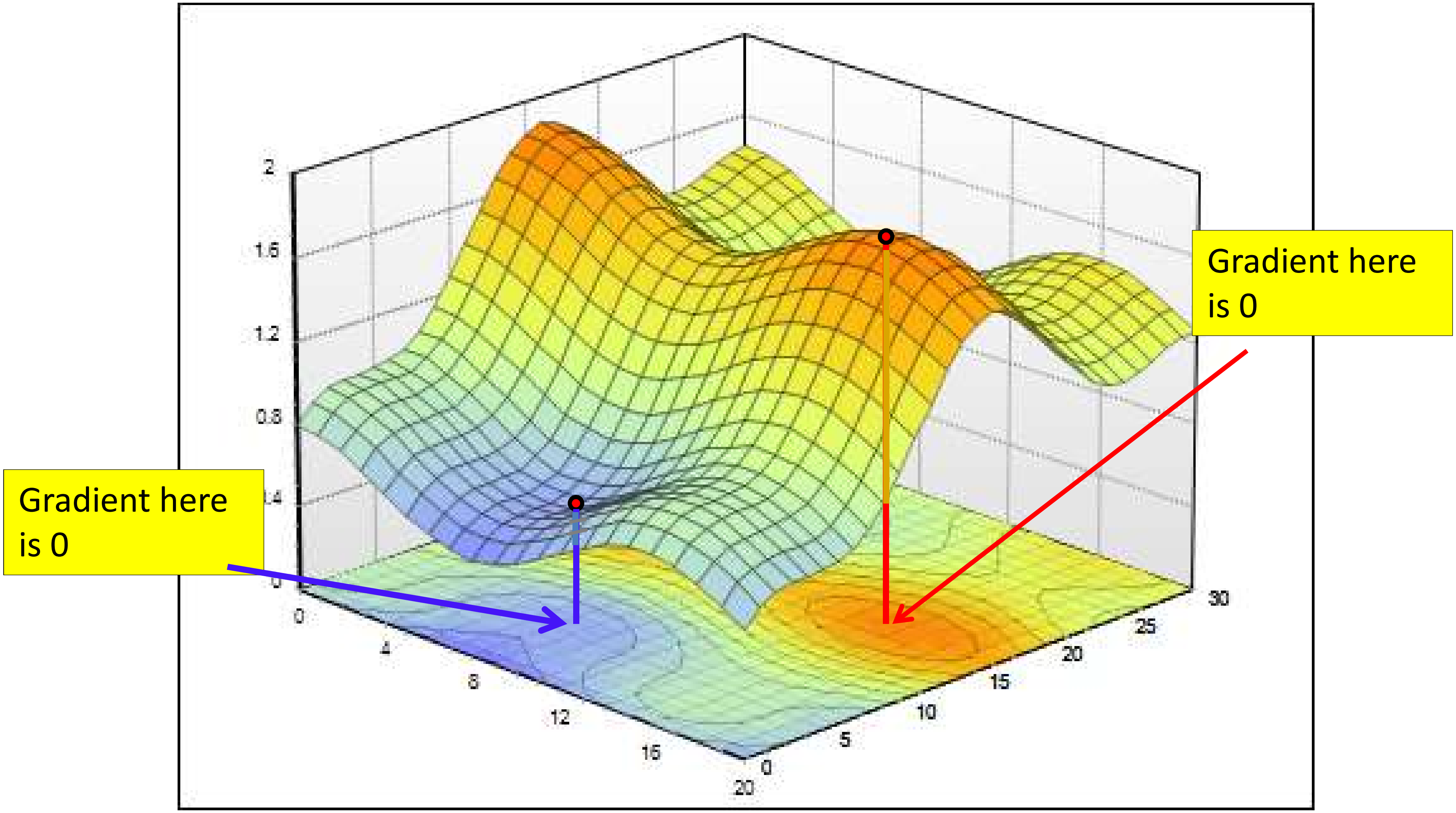
2
1.6
1.2
0.8
0.4
0
0
4
8
12
16
20
0
5
10
15
Gradient vector $\nabla f(X)$
Moving in this direction *increases* $f(X)$ fastest

Gradient



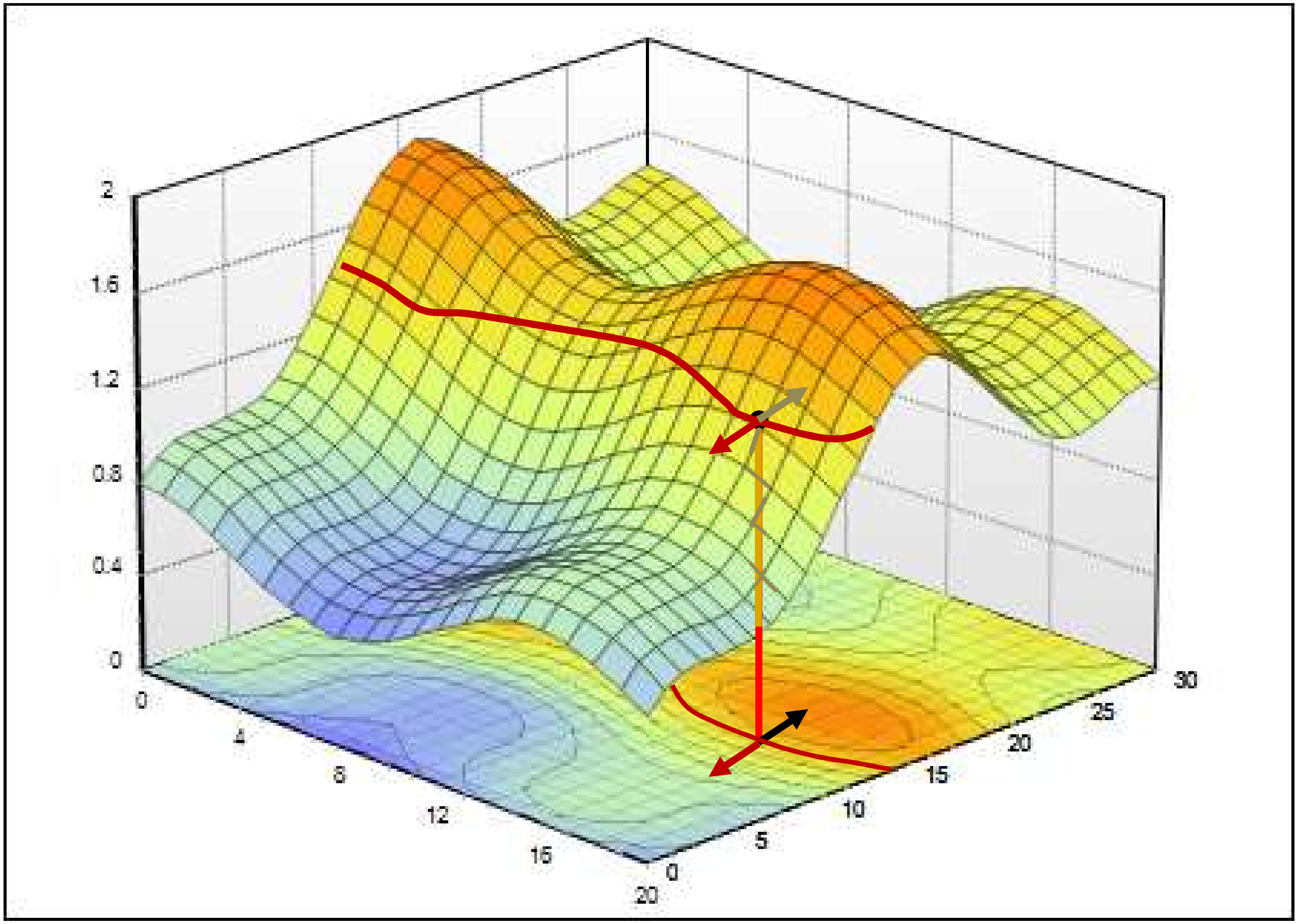
Gradient vector $\nabla f(X)$
Moving in this direction *increases* $f(X)$ fastest
$-\nabla f(X)$
Moving in this direction *decreases* $f(X)$ fastest

Gradient



Gradient here is 0
Gradient here is 0
2
1.6
1.2
0.8
0
0
4
8
12
16
20
0
5
10
15
20
25
30

Properties of Gradient: 2



- The gradient vector $\nabla f(X)$ is perpendicular to the level curve

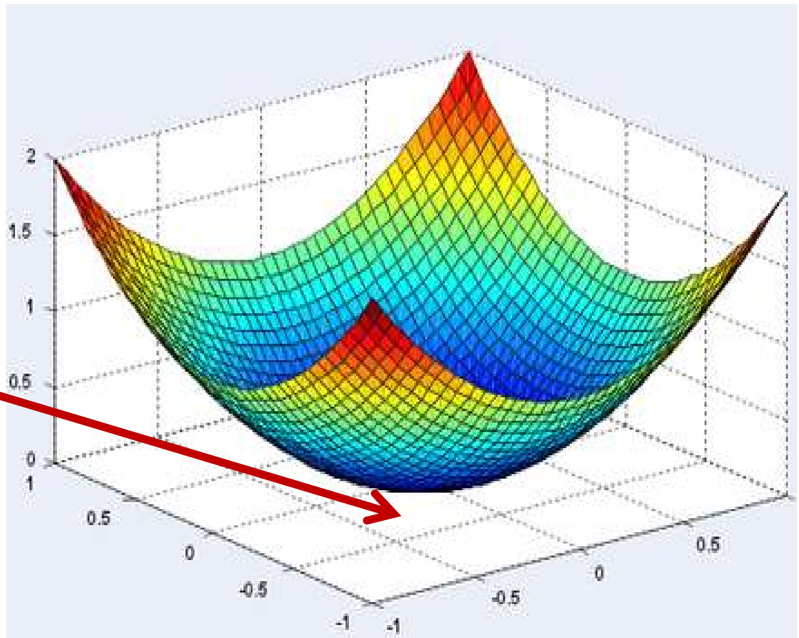
The Hessian

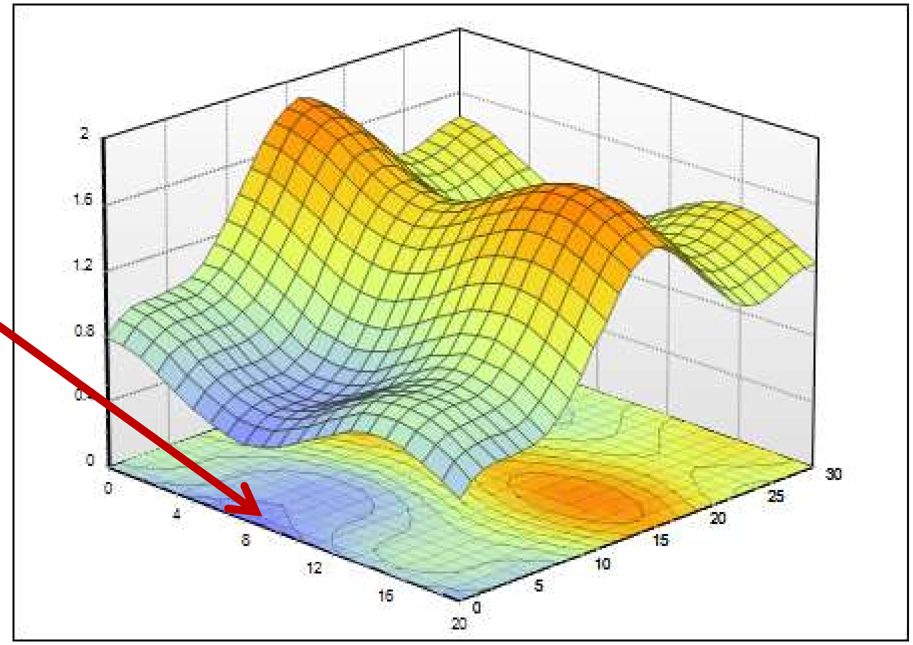
- The Hessian of a function $f(x_1, x_2, \ldots, x_n)$ is given by the second derivative

$$\nabla^2 f(x_1, \ldots, x_n) := \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1 \partial x_2} & . & . & \dfrac{\partial^2 f}{\partial x_1 \partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2 \partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & . & . & \dfrac{\partial^2 f}{\partial x_2 \partial x_n} \\ . & . & . & . & . \\ . & . & . & . & . \\ \dfrac{\partial^2 f}{\partial x_n \partial x_1} & \dfrac{\partial^2 f}{\partial x_n \partial x_2} & . & . & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Returning to direct 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

Unconstrained Minimization of function (Example)

- Minimize

$$f(x_1, x_2, x_3) = (x_1)^2 + x_1(1 - x_2) - (x_2)^2 - x_2 x_3 + (x_3)^2 + x_3$$

- Gradient

$$\nabla f = \begin{bmatrix} 2x_1 + 1 - x_2 \\ -x_1 + 2x_2 - x_3 \\ -x_2 + 2x_3 + 1 \end{bmatrix}^T$$

Unconstrained Minimization of function (Example)

- Set the gradient to null

$$\nabla f = 0 \Rightarrow \begin{bmatrix} 2x_1 + 1 - x_2 \\ -x_1 + 2x_2 - x_3 \\ -x_2 + 2x_3 + 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

- Solving the 3 equations system with 3 unknowns

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$$

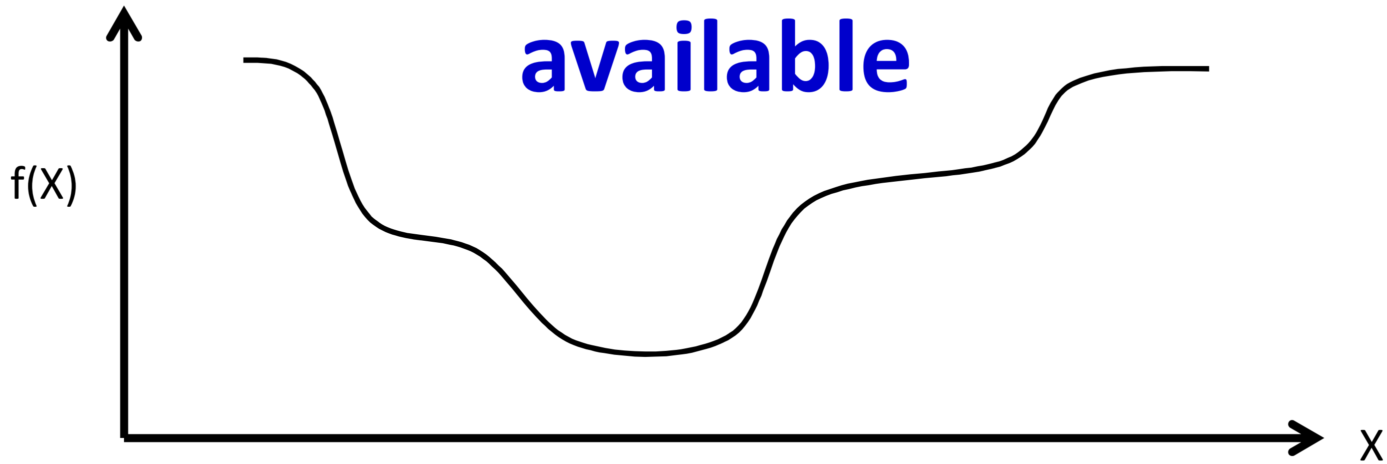
Unconstrained Minimization of function (Example)

- Compute the Hessian matrix $\nabla^2 f = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$
- Evaluate the eigenvalues of the Hessian matrix

$$\lambda_1 = 3.414, \;\; \lambda_2 = 0.586, \;\; \lambda_3 = 2$$

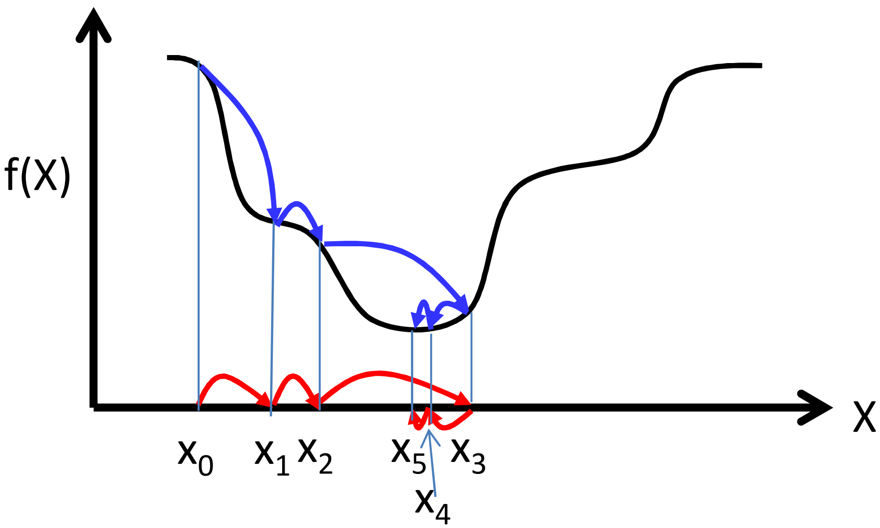
- All the eigenvalues are positives => the Hessian matrix is positive definite
- The point $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \end{bmatrix}$ is a minimum

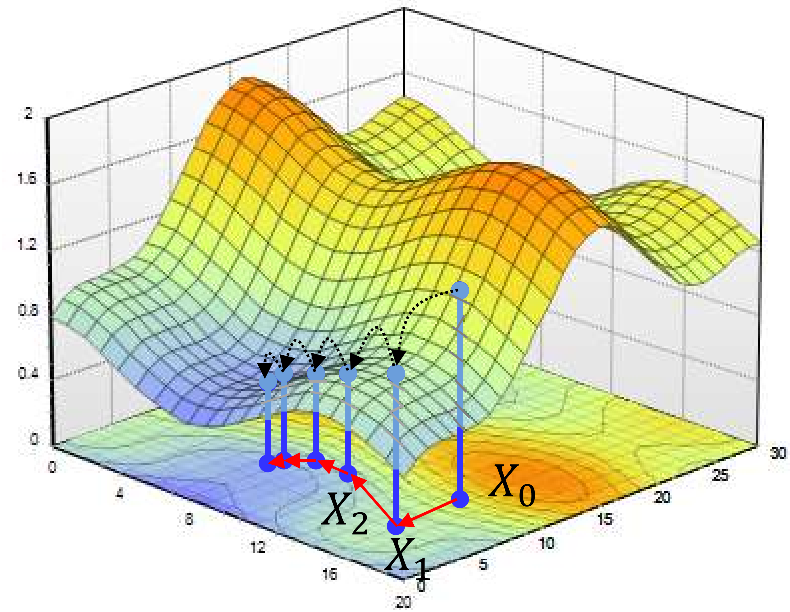
Closed Form Solutions are not always available



- 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

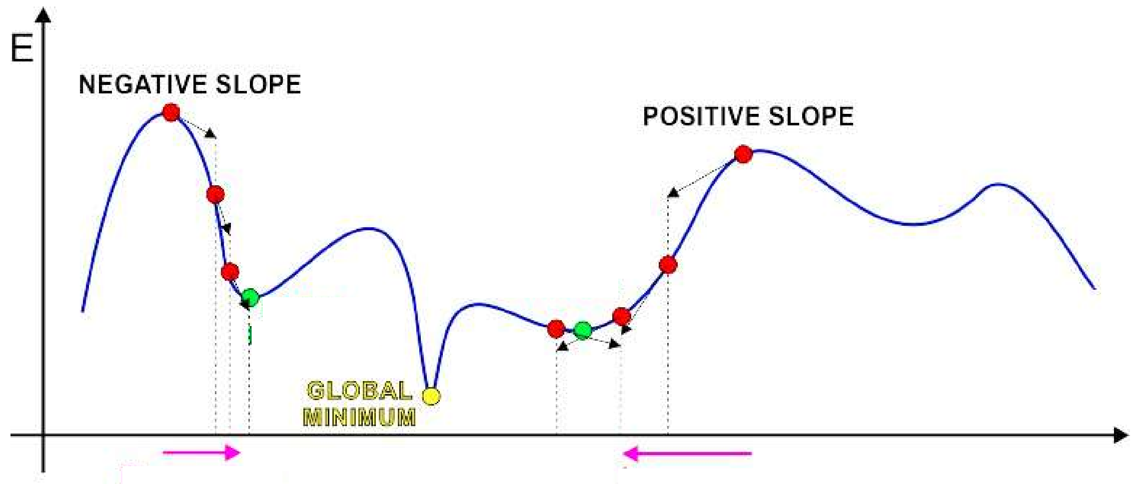
Iterative solutions





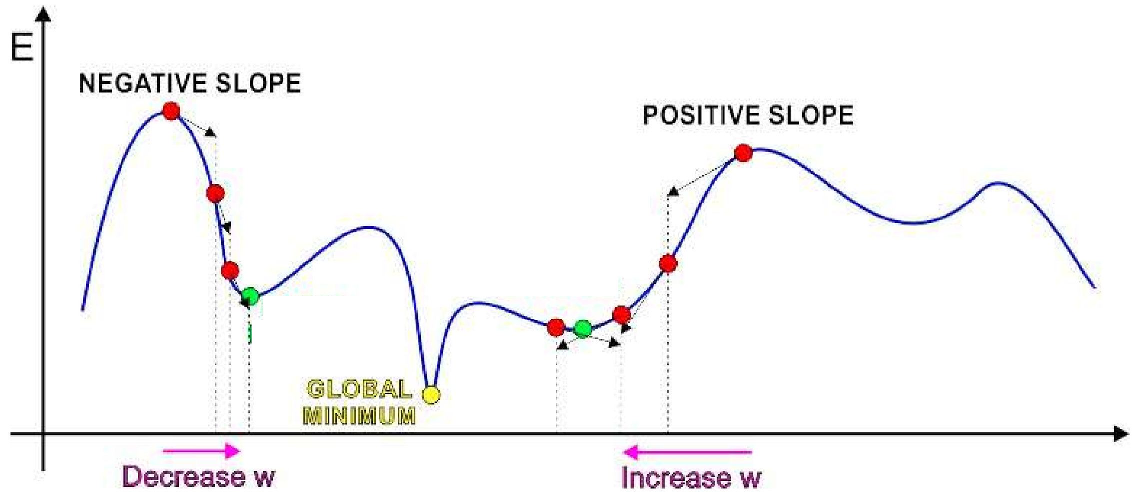
- Iterative solutions
  - Start from an initial guess X_0 for the optimal X
  - Update the guess towards a (hopefully) “better” value of $\mathrm{f}(X)$
  - Stop when $\mathrm{f}(X)$ no longer decreases
- Problems:
  - Which direction to step in
  - How big must the steps be

The Approach of Gradient Descent



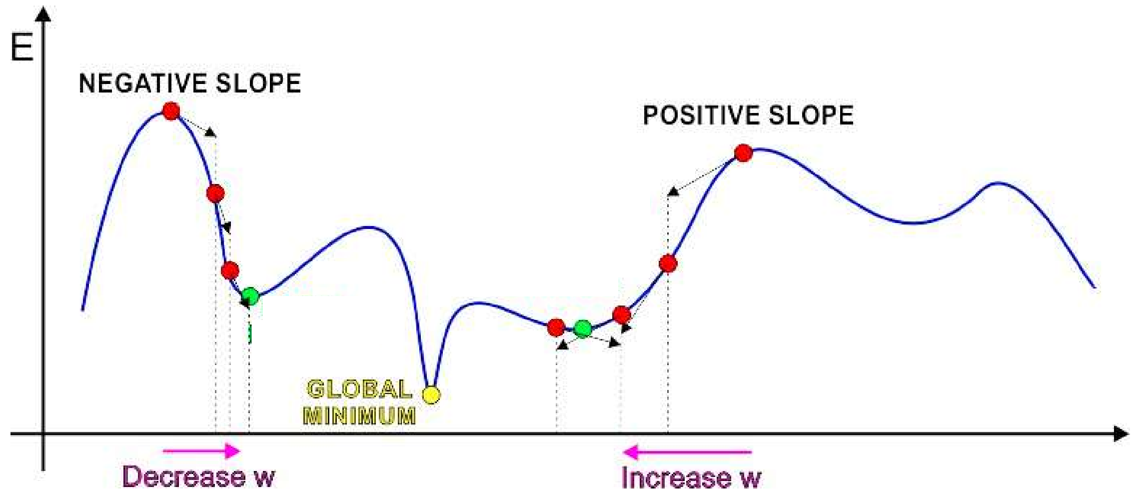
- Iterative solution:
  - Start at some point
  - Find direction in which to shift this point to decrease error
    - This can be found from the derivative of the function
      - A positive derivative → moving left decreases error
      - A negative derivative → moving right decreases error
  - Shift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

The Approach of Gradient Descent



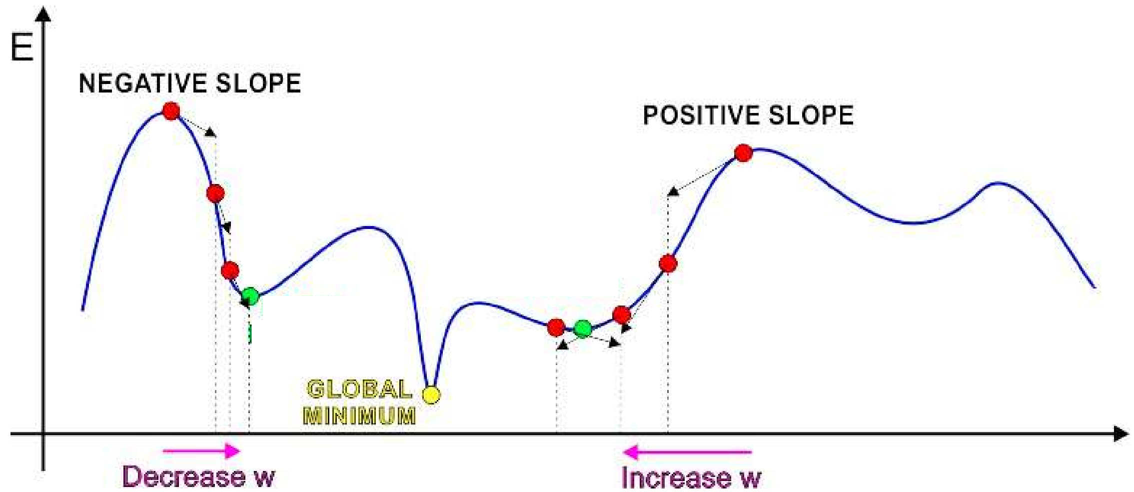
- Iterative solution: Trivial algorithm
  - Initialize x^0
  - While $f'\left(x^k\right) \neq 0$
    - If $sign\left(f'\left(x^k\right)\right)$ is positive:
      - $x^{k+1} = x^k - step$
    - Else
      - $x^{k+1} = x^k + step$
  - What must step be to ensure we actually get to the optimum?

The Approach of Gradient Descent



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

The Approach of Gradient Descent



- Iterative solution: Trivial algorithm
  - Initialize x_0
  - While $f'(x^k) \neq 0$
    - $x^{k+1} = x^k - \eta^k f'(x^k)$
  - η^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*

$$x^{k+1} = x^k + \eta^k \nabla f\left(x^k\right)^T$$

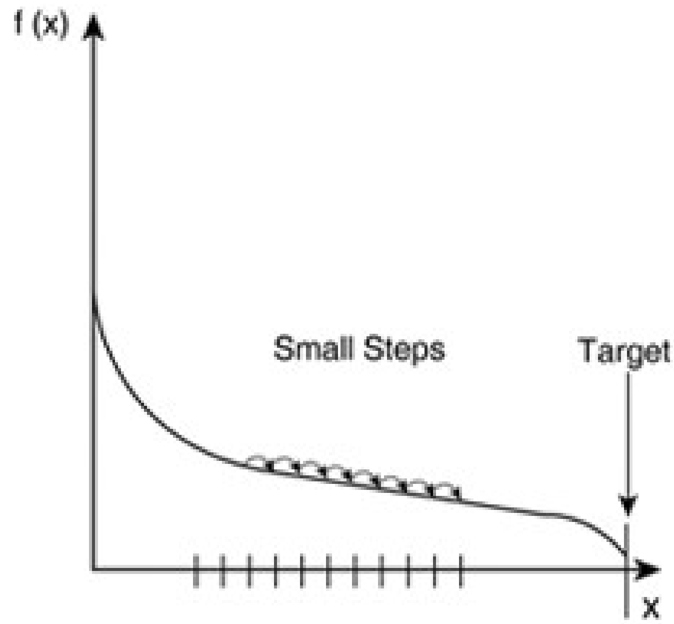
  - To find a *minimum* move *exactly opposite the direction of the gradient*

$$x^{k+1} = x^k - \eta^k \nabla f\left(x^k\right)^T$$

- Many solutions to choosing step size η^k
  - Later lecture

1. Fixed step size

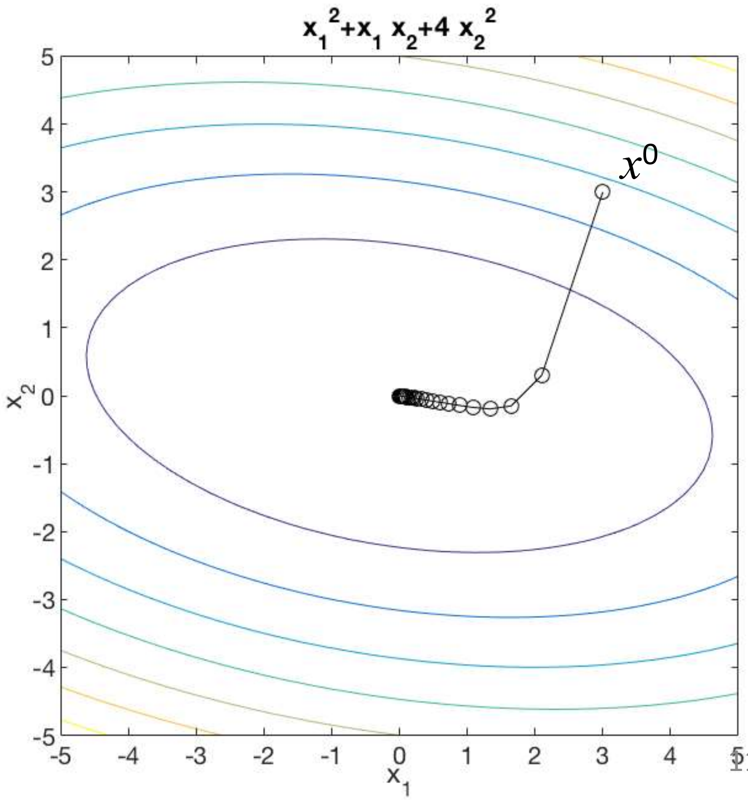
- Fixed step size
  - Use fixed value for η^k



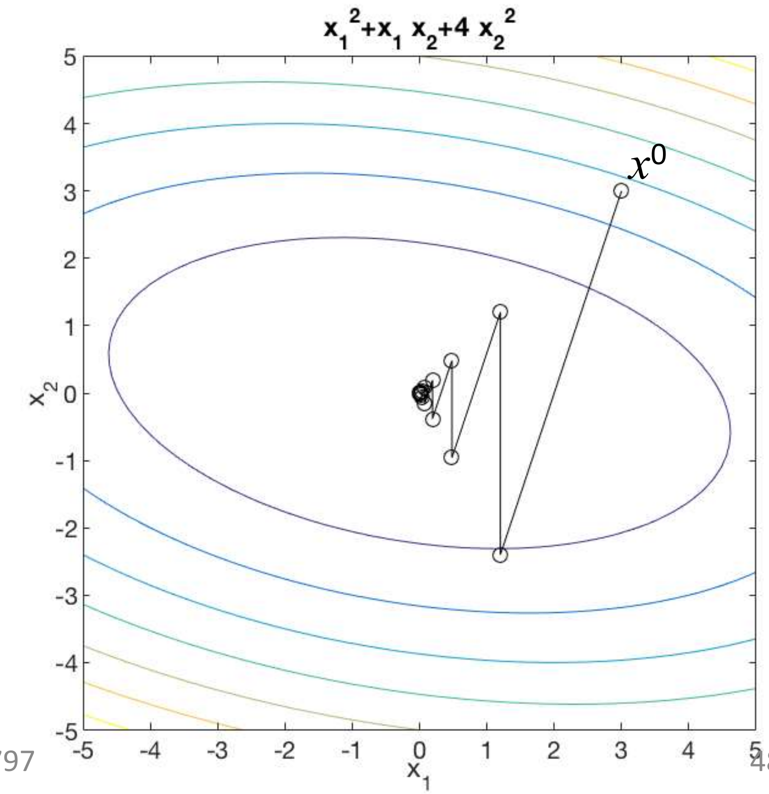
Influence of step size example (constant step size)

$$f(x_1, x_2) = (x_1)^2 + x_1 x_2 + 4(x_2)^2 \qquad x^{initial} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

$\eta = 0.1$



$\eta = 0.2$



What is the optimal step size?

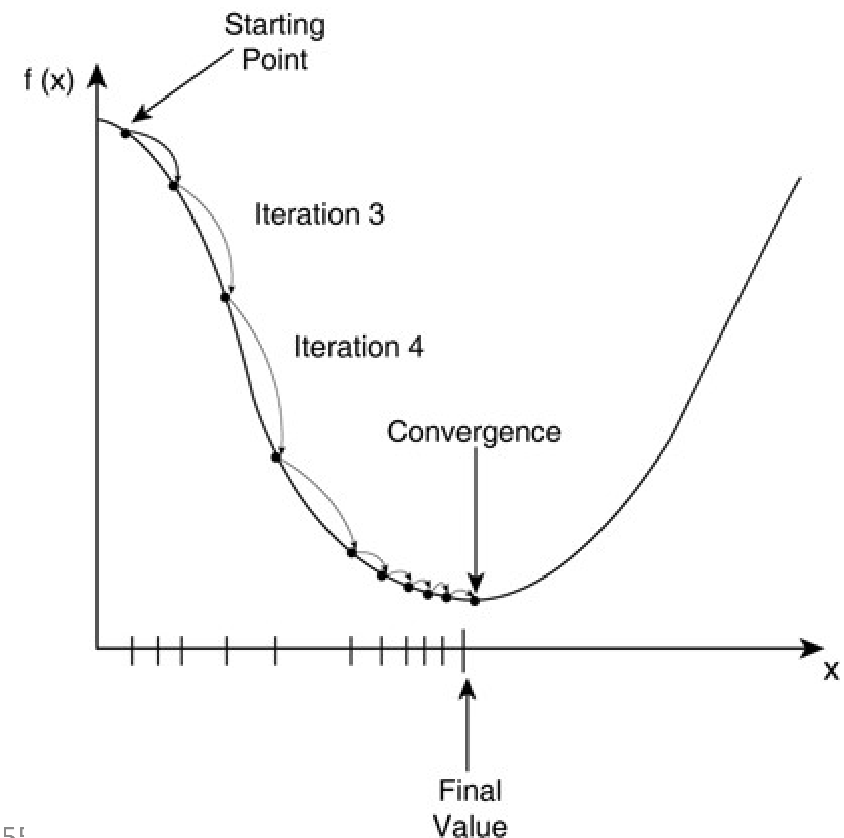
- Step size is critical for fast optimization
- Will revisit this topic later
- For now, simply assume a potentially-iteration-dependent step size

Gradient descent convergence criteria

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

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

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



Overall Gradient Descent Algorithm

- Initialize:
  - x^0
  - $k = 0$

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

- Returning to our problem..

Problem Statement

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$

- Minimize the following function

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

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 $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \dots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$
- What are these input-output pairs?

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

w.r.t W

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

Problem Setup: Things to define

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$
- What are these input-output pairs?

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

w.r.t W

What is f() and what are its parameters?

- This is problem of functio

  – An instance of optimization

Problem Setup: Things to define

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$

What are these input-output pairs?

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

What is the divergence div()?

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

- This is problem of functio
  - An instance of optimization

Problem Setup: Things to define

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \dots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$

- Minimize the following function

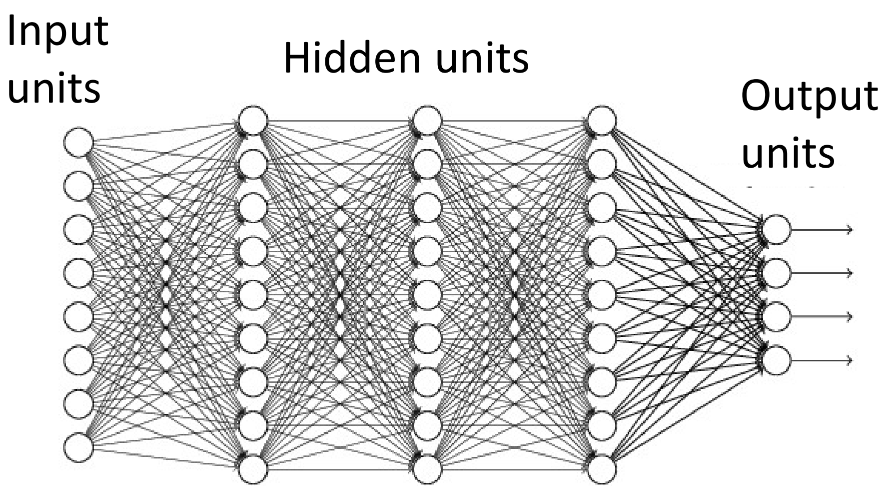
$$Err(W) = \frac{1}{T}\sum_i div(f(X_i; W), d_i)$$

w.r.t W

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

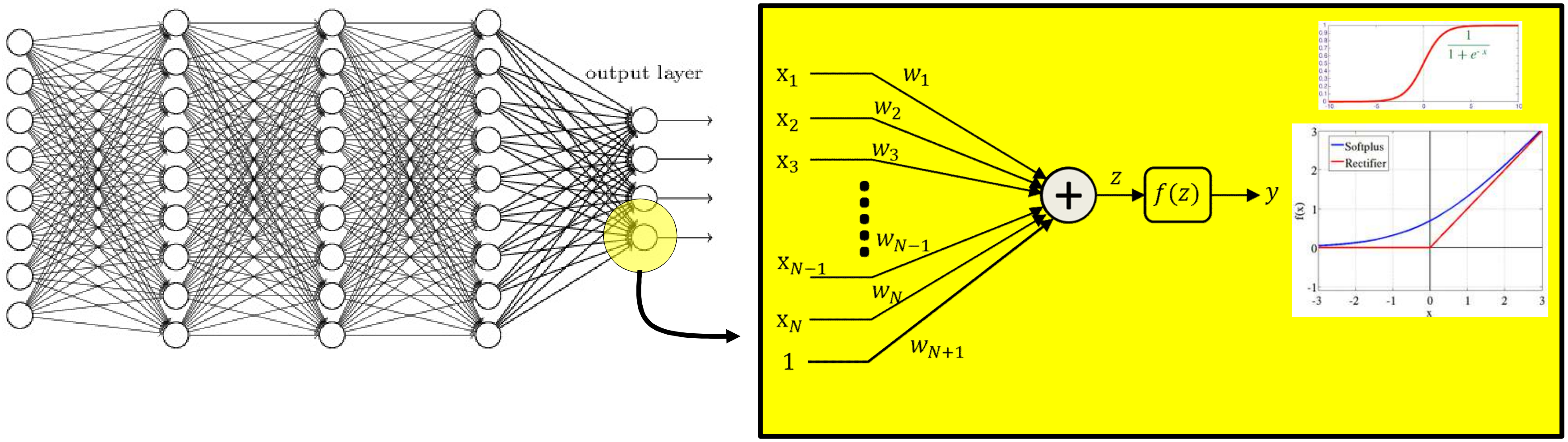
- This is problem of functio
  – An instance of optimization

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



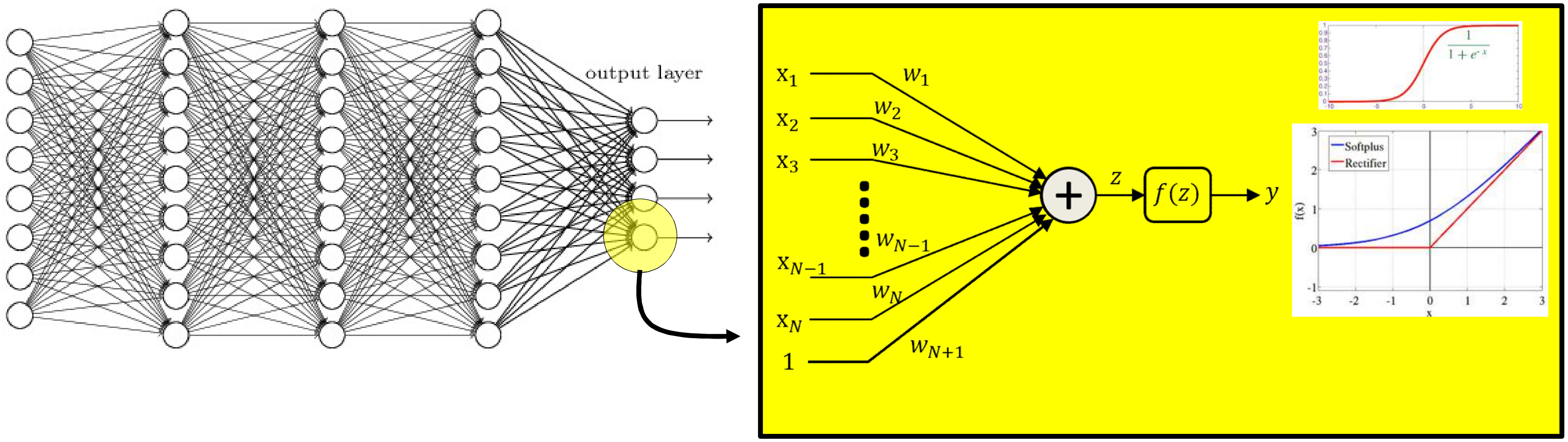
- Individual neurons operate on a set of inputs and produce a single output
  - **Standard setup:** A differentiable activation function applied the sum of weighted inputs and a bias

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

  - More generally: *any* differentiable function

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

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

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

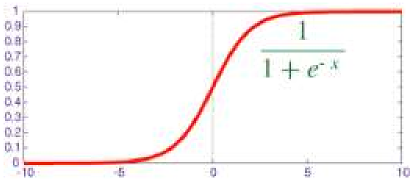
  - More generally: *any* differentiable function

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

We will assume this unless otherwise specified

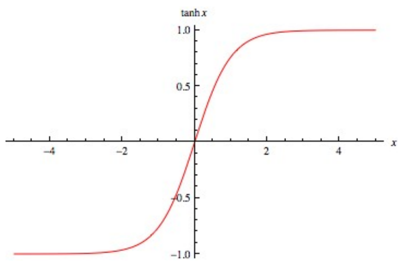
Parameters are weights w_i and bias b

Activations and their derivatives



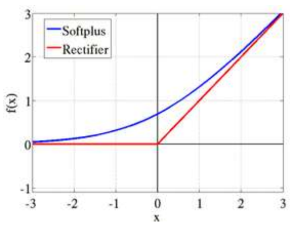
$$f(z) = \frac{1}{1 + \exp(-z)}$$

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



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

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



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

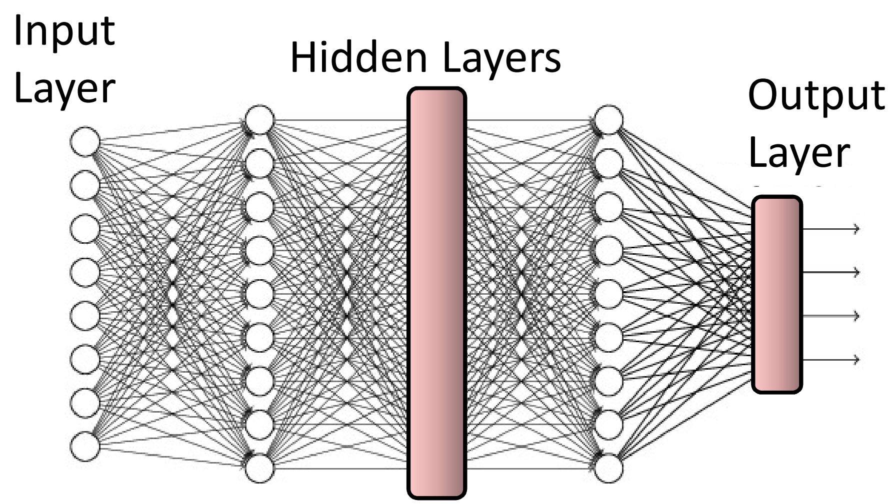
This space left intentionally (kind of) blank

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

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

- Some popular activation functions and their derivatives

Vector Activations

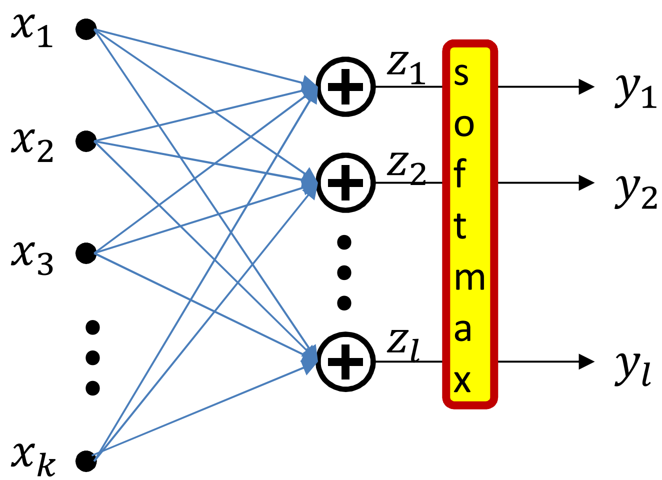


- We can also have neurons that have *multiple coupled* outputs

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

  – Function $f()$ operates on set of inputs to produce set of outputs
  – Modifying the parameters W will affect *all* outputs

Vector activation example: Softmax



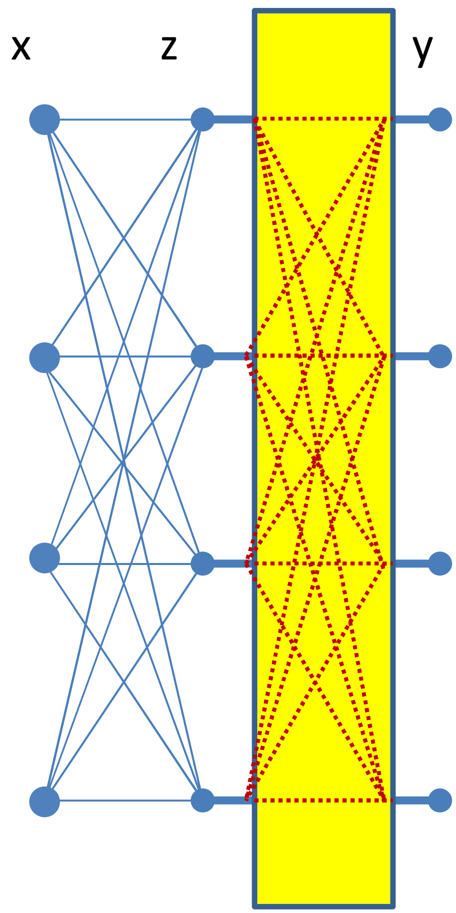
- Example: Softmax *vector* activation

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

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

Parameters are weights w_{ji} and bias b_i

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



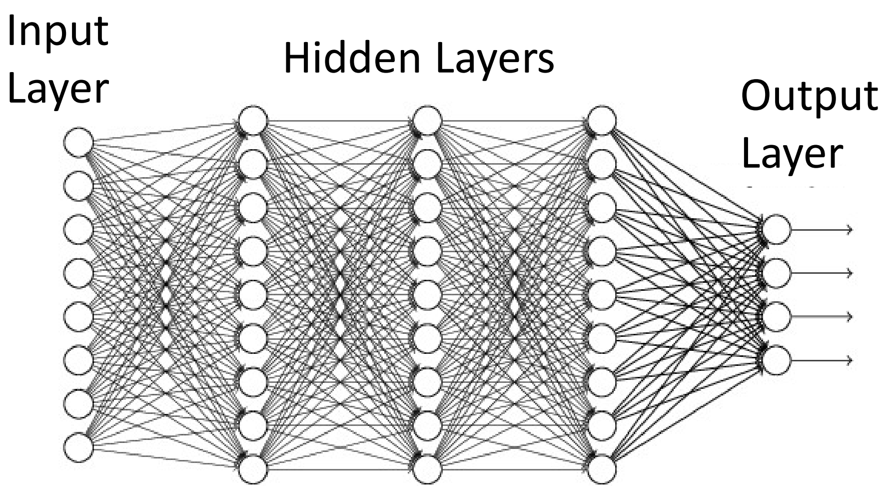
$$z_i = \sum_j w_{ji} x_j + b_i$$

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

Parameters are weights w_{ji} and bias b_i

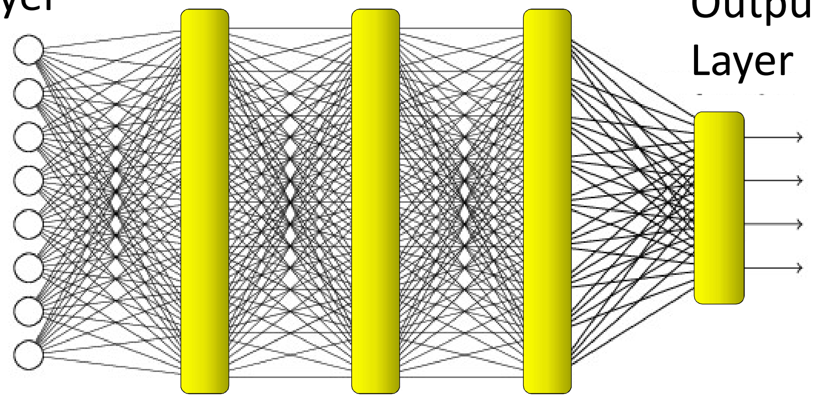
- A layer of multiplicative combination is a special case of vector activation

Typical network



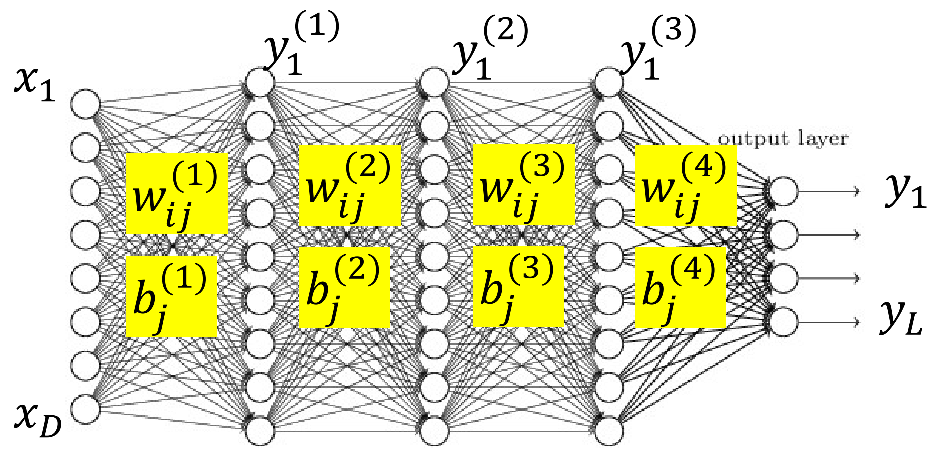
- We assume a "layered" network for simplicity
  - We will refer to the inputs as the *input layer*
    - No neurons here – the "layer" simply refers to inputs
  - We refer to the outputs as the output layer
  - Intermediate layers are "hidden" layers

Typical network



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

Notation



- The input layer is the 0th layer
- We will represent the output of the i-th perceptron of the kth 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-1th layer and the jth unit of the k-th layer as $w_{ij}^{(k)}$
  - The bias to the jth unit of the k-th layer is $b_j^{(k)}$

Problem Setup: Things to define

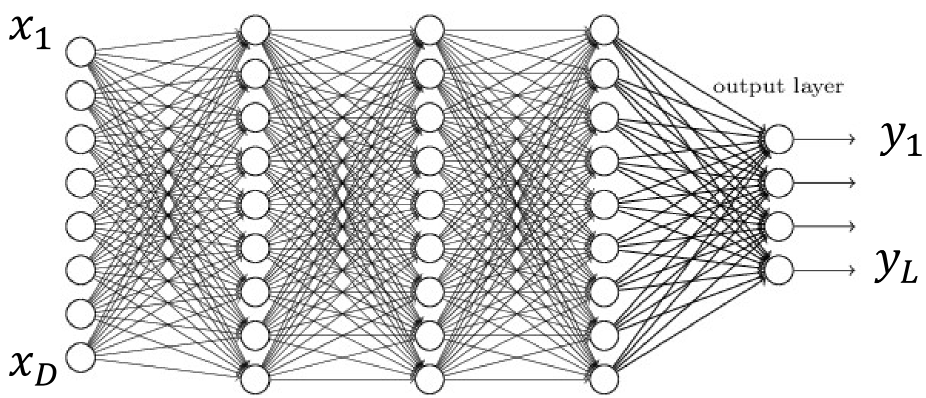
- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$
- What are these input-output pairs?

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

w.r.t W

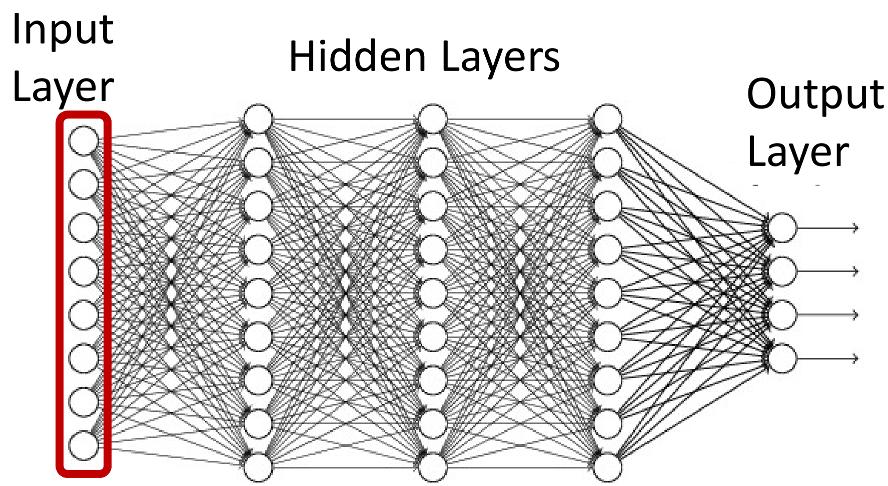
- This is problem of function minimization
  - An instance of optimization

Vector notation



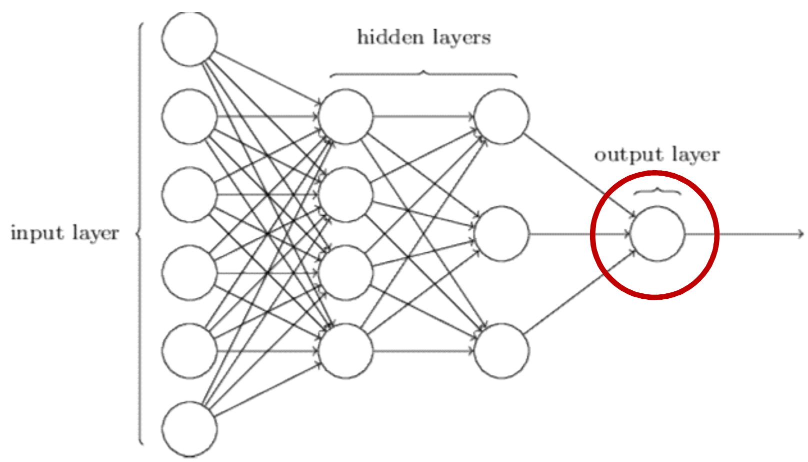
- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
- $X_n = [x_{n1}, x_{n2}, \ldots, x_{nD}]$ is the nth input vector
- $d_n = [d_{n1}, d_{n2}, \ldots, d_{nL}]$ is the nth desired output
- $Y_n = [y_{n1}, y_{n2}, \ldots, y_{nL}]$ is the nth vector of *actual* outputs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

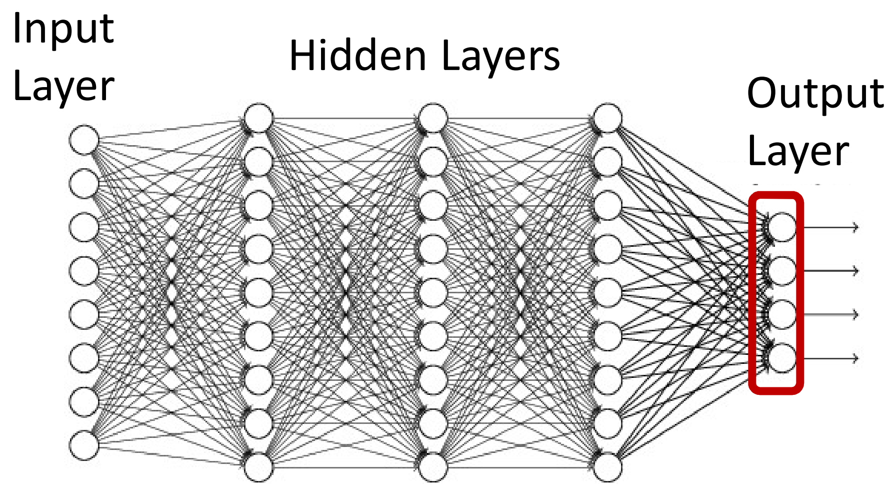
Representing the input



- 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

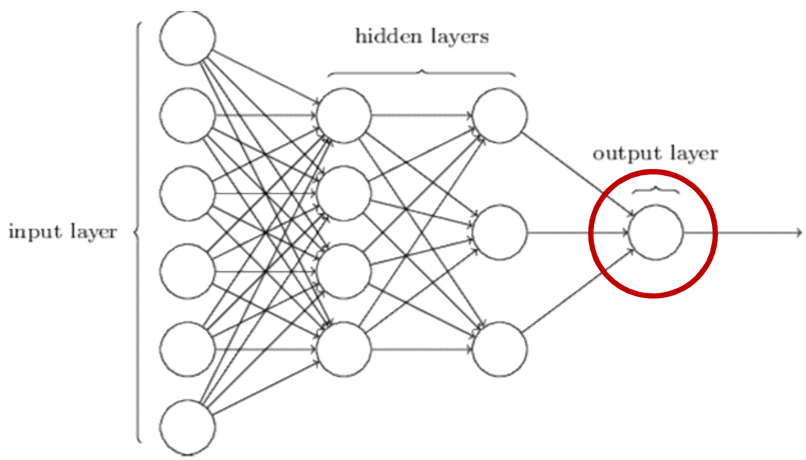
Representing the output





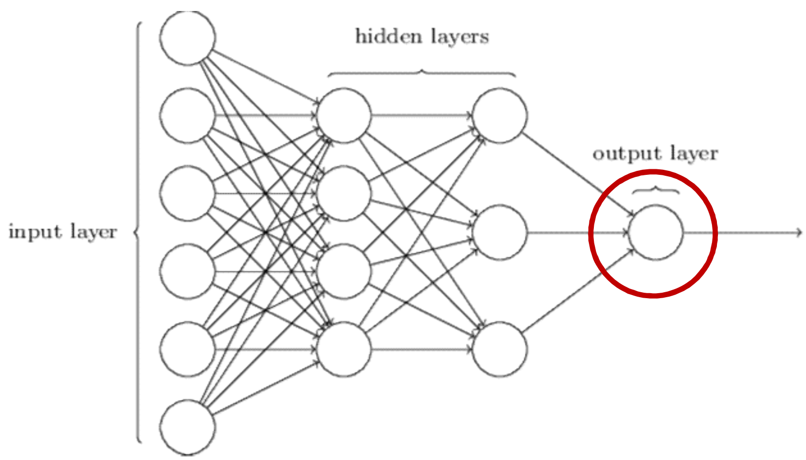
- If the desired *output* is real-valued, no special tricks are necessary
  - Scalar Output : single output neuron
    - d = scalar (real value)
  - Vector Output : as many output neurons as the dimension of the desired output
    - $d = [d_1\ d_2\ ..\ d_L]$ (vector of real values)

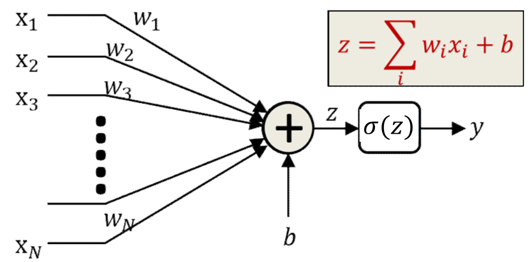
Representing the output

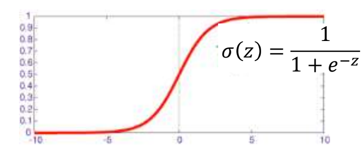


- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
  - 1 = Yes it's a cat
  - 0 = No it's not a cat.

Representing the output

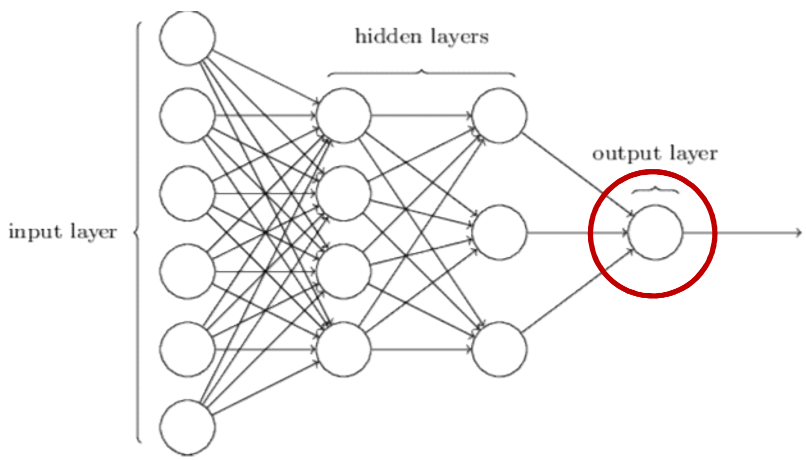


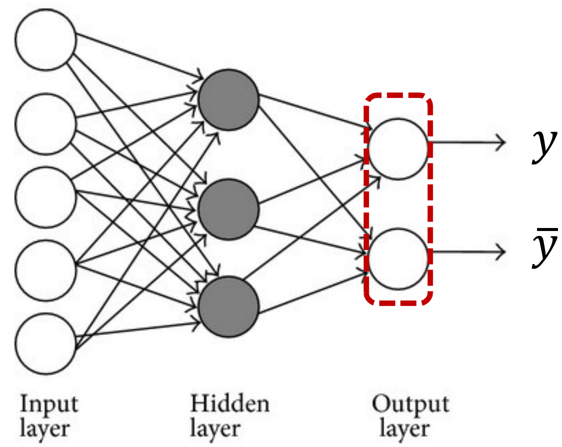




- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
- Output activation: Typically a sigmoid
  - Viewed as the *probability* $P(Y = 1|X)$ of class value 1
    - Indicating the fact that for actual data, in general an feature value X may occur 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: → [1 0]
  - No: → [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:

  [cat dog camel hat flower]^T

- For inputs of each of the five classes the desired output is:

  cat: $[1\ 0\ 0\ 0\ 0]^T$

  dog: $[0\ 1\ 0\ 0\ 0]^T$

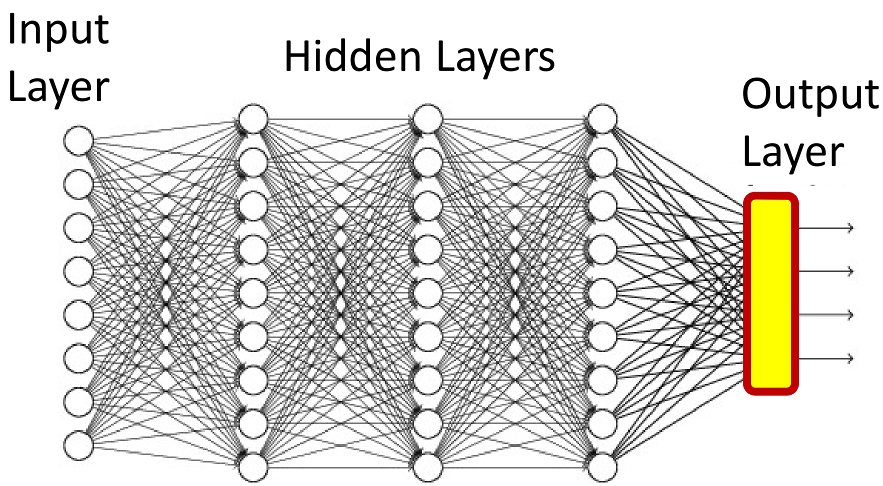
  camel: $[0\ 0\ 1\ 0\ 0]^T$

  hat: $[0\ 0\ 0\ 1\ 0]^T$

  flower: $[0\ 0\ 0\ 0\ 1]^T$

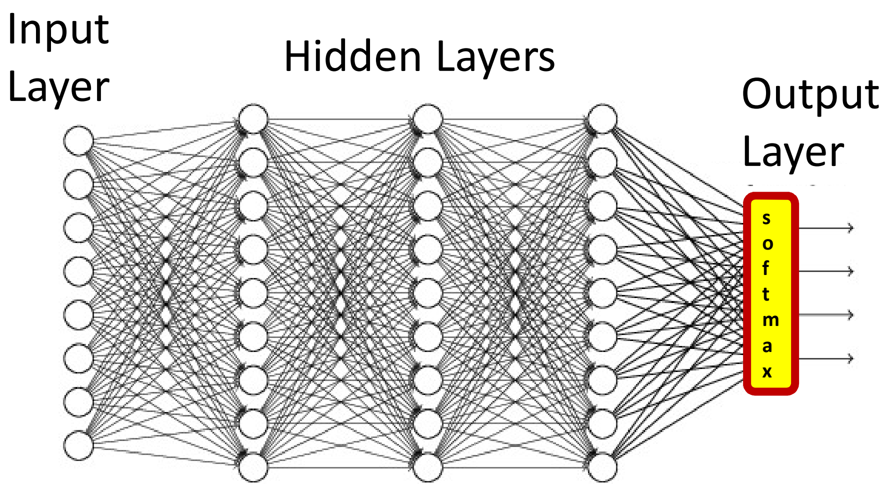
- For an input of any class, we will have a five-dimensional vector output with four zeros and a single 1 at the position of that class
- This is a *one hot vector*

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



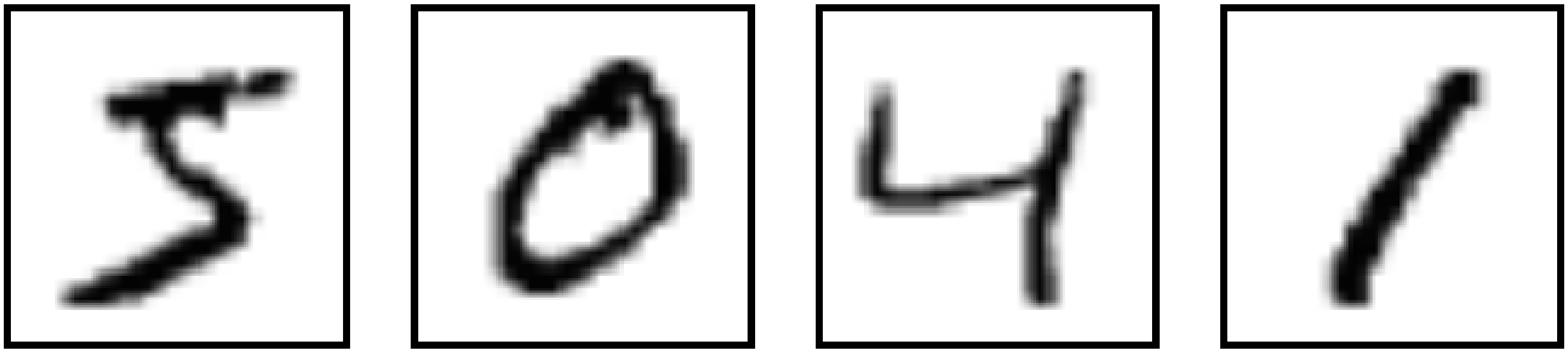
- Softmax *vector* activation is often used at the output of multi-class classifier nets

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

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

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

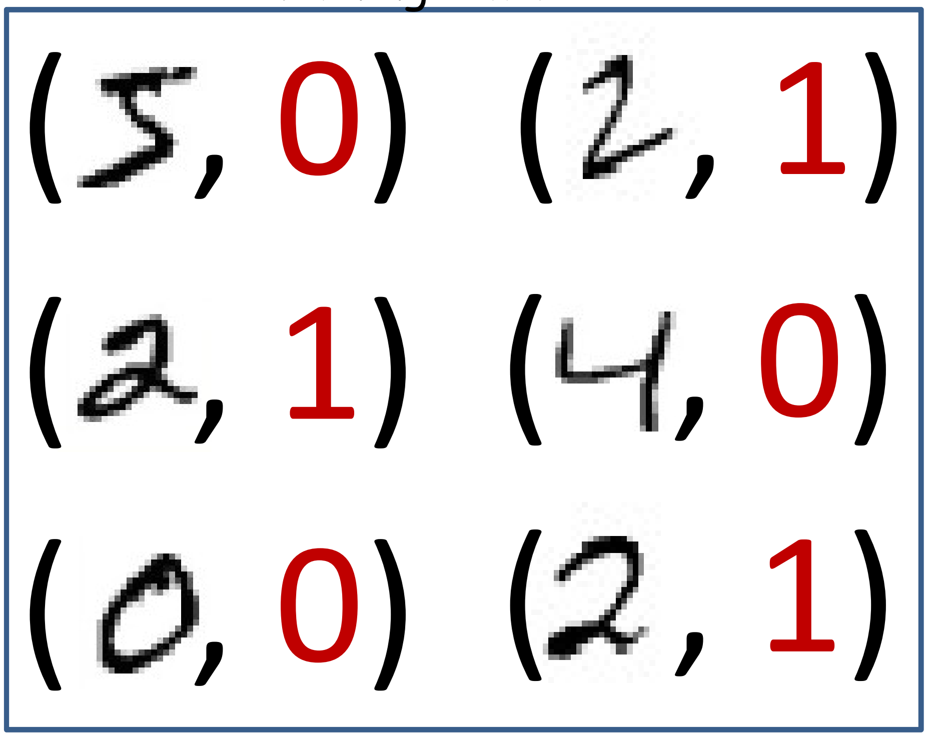
Typical Problem Statement

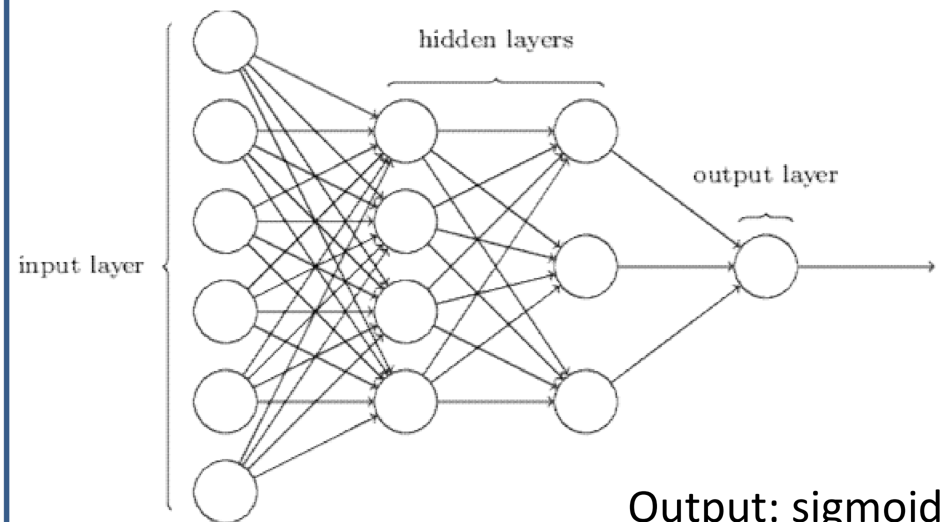


- We are given a number of “training” data instances
- E.g. images of digits, along with information about which digit the image represents
- Tasks:
  - Binary recognition: Is this a “2” or not
  - Multi-class recognition: Which digit is this? Is this a digit in the first place?

Typical Problem statement: binary classification

Training data





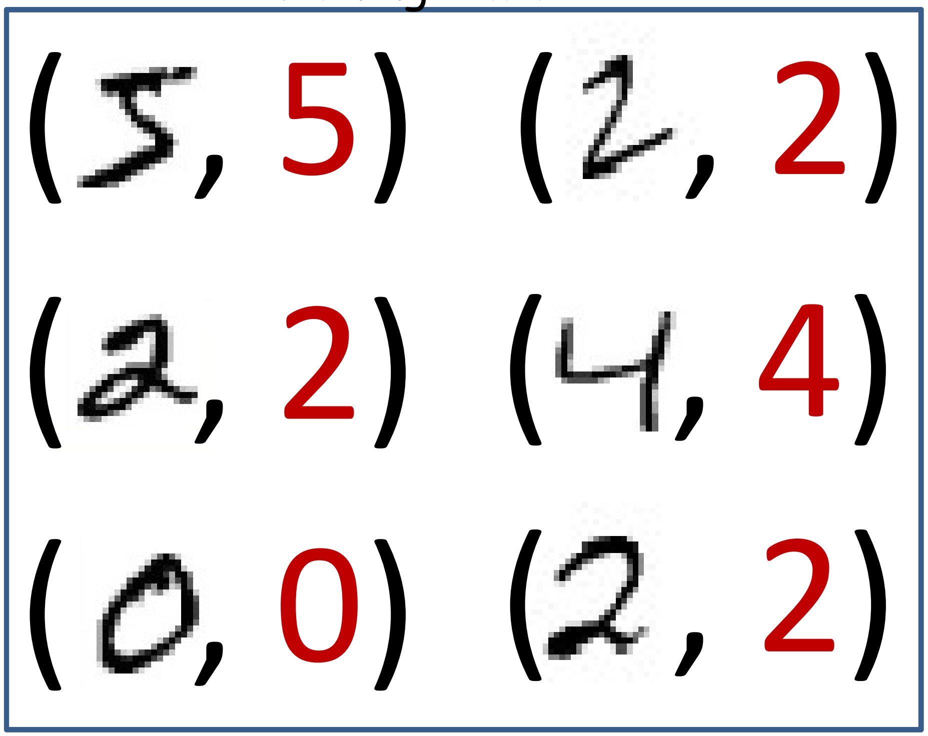
Input: vector of pixel values

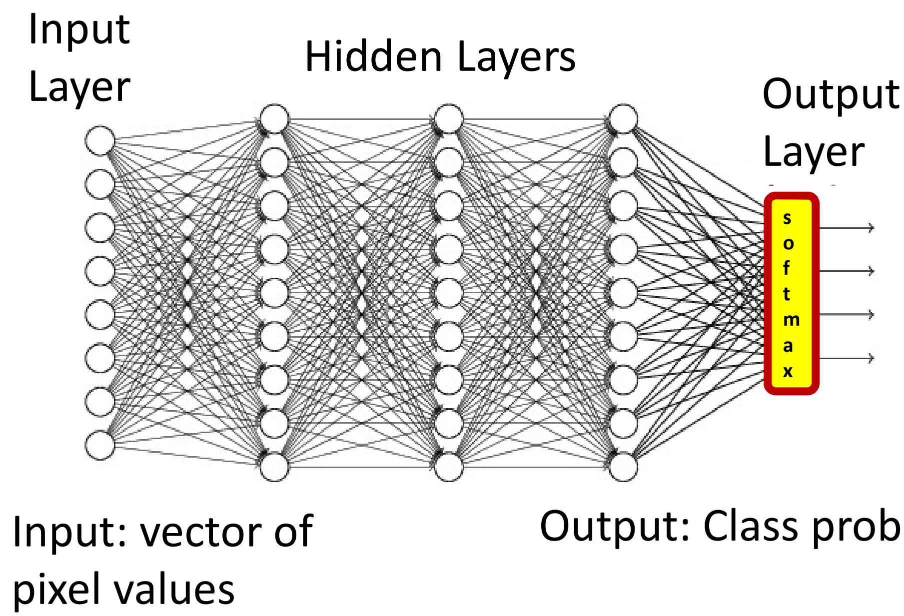
Output: sigmoid

- 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





- 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 $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$

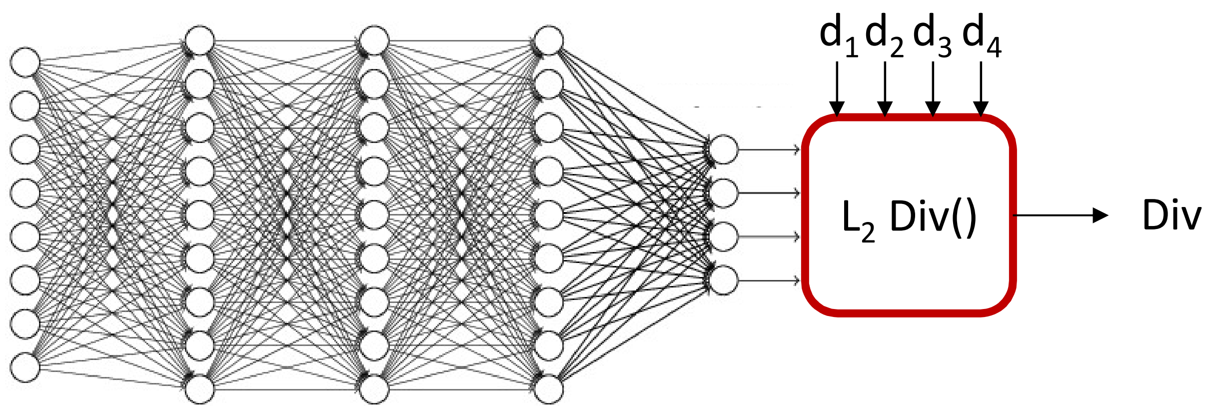
- Minimize the following function

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

What is the divergence div()?

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

Examples of divergence functions



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

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

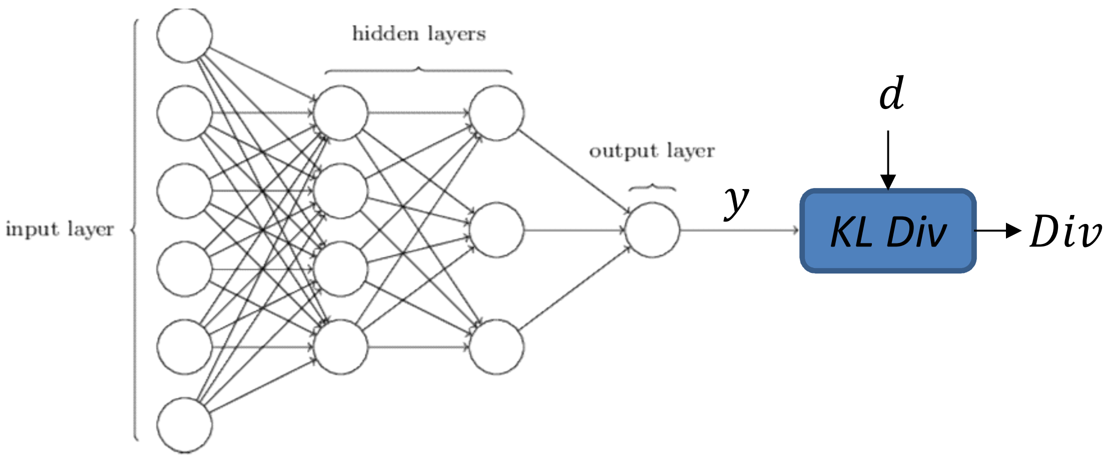
  – Squared Euclidean distance between true and desired output

  – Note: this is differentiable

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

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

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

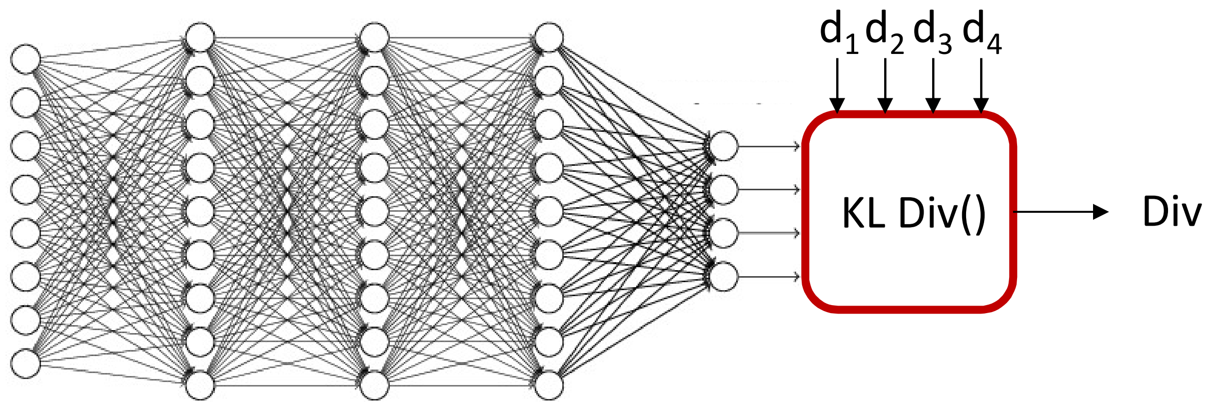
$$Div(Y,d) = -d\log Y - (1-d)\log(1-Y)$$

  – Minimum when $d = Y$

- Derivative

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

For multi-class classification



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

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

- Derivative

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

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

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 ith instance is $div(Y_i, d_i)$
- The total error

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

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

Recap: Gradient Descent Algorithm

- In order to minimize any function $f(x)$ w.r.t. x
- Initialize:
  - x^0
  - $k = 0$

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

Recap: Gradient Descent Algorithm

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

Explicitly stating it by component

  - $k = k + 1$

Training Neural Nets through Gradient Descent

Total training error:

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

- Gradient descent algorithm:
- Initialize all weights and biases $\left\{w_{ij}^{(k)}\right\}$
  - 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

Assuming the bias is also represented as a weight

Training Neural Nets through Gradient Descent

Total training error:

$$Err = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} Div(\boldsymbol{Y_t}, \boldsymbol{d_t})$$

- Gradient descent algorithm:
- Initialize all weights $\left\{w_{ij}^{(k)}\right\}$
- 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:

$$Err = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} Div(\boldsymbol{Y_t}, \boldsymbol{d_t})$$

- Computing the derivative

Total derivative:

$$\frac{dErr}{dw_{i,j}^{(k)}} = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} \frac{dDiv(\boldsymbol{Y_t}, \boldsymbol{d_t})}{dw_{i,j}^{(k)}}$$

Training by gradient descent

- Initialize all weights $\left\{w_{ij}^{(k)}\right\}$
- Do:
  - For all $i, j, k,$ initialize $\frac{dEr}{dw_{i,j}^{(k)}} = 0$
  - For all $t = 1 : T$
    - For every layer k for all i, j:
      - Compute $\frac{d\boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{dw_{i,j}^{(k)}}$
      - Compute $\frac{dErr}{dw_{i,j}^{(k)}} += \frac{d\boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{dw_{i,j}^{(k)}}$
  - For every layer k for all i, j:

$$w_{i,j}^{(k)} = w_{i,j}^{(k)} - \frac{\eta}{T}\frac{dErr}{dw_{i,j}^{(k)}}$$

- Until Err has converged

The derivative

Total training error:

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

Total derivative:

$$\frac{dErr}{dw_{i,j}^{(k)}} = \frac{1}{T}\sum_t \frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}}$$

- 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 $\Rightarrow$ $\Delta y \approx \frac{dy}{dx}\Delta x$

For any differentiable function

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

with partial derivatives

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

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

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

Calculus Refresher: Chain rule

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

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

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

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

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

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

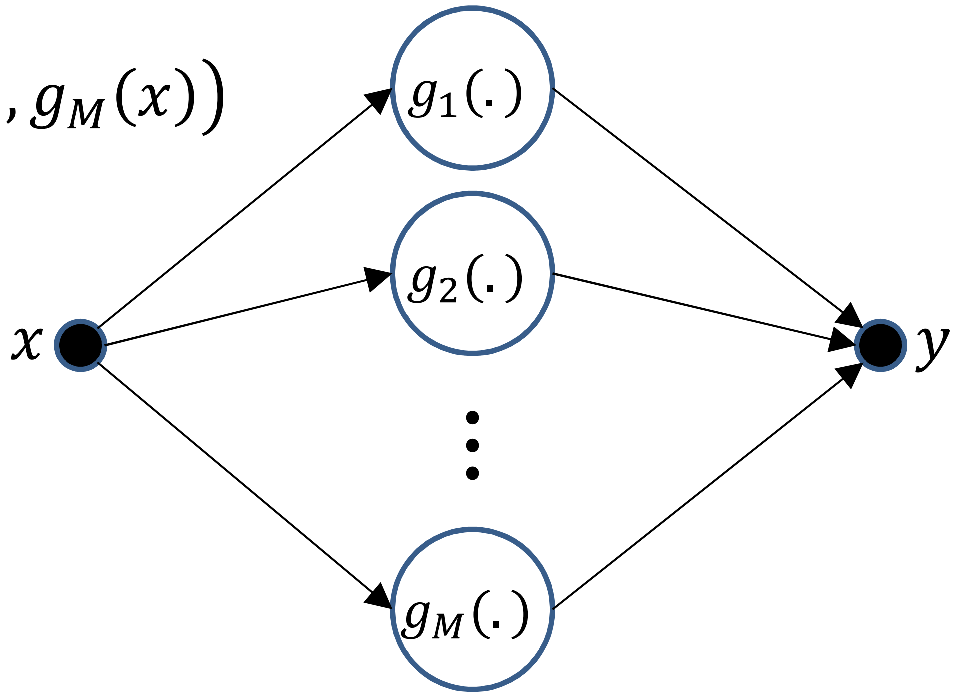
$$\Delta y = \frac{\partial y}{\partial g_1(x)}\Delta g_1(x) + \frac{\partial y}{\partial g_2(x)}\Delta g_2(x) + \cdots + \frac{\partial y}{\partial g_M(x)}\Delta g_M(x)$$

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

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

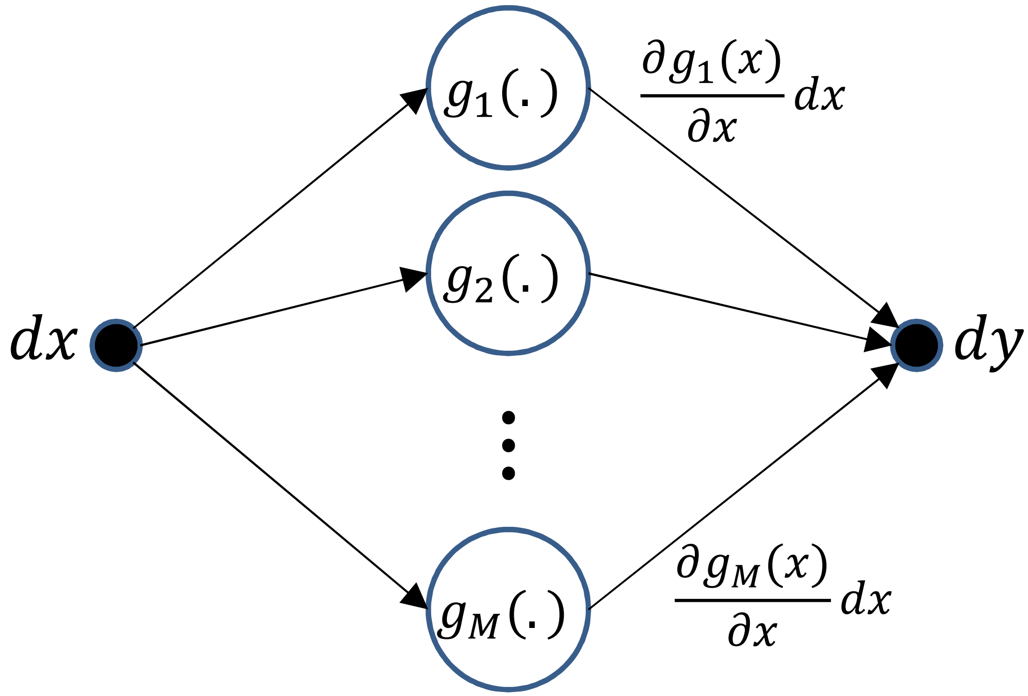
Distributed Chain Rule: Influence Diagram

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



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

Distributed Chain Rule: Influence Diagram

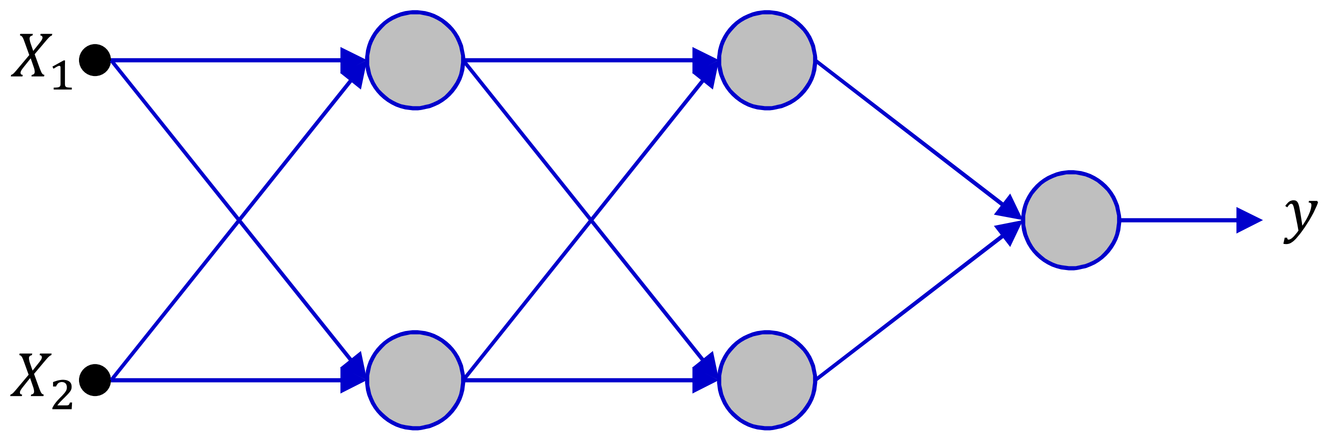


- Small perturbations in x cause small perturbations in each of $g_1 \ldots g_M$, each of which individually additively perturbs y

Returning to our problem

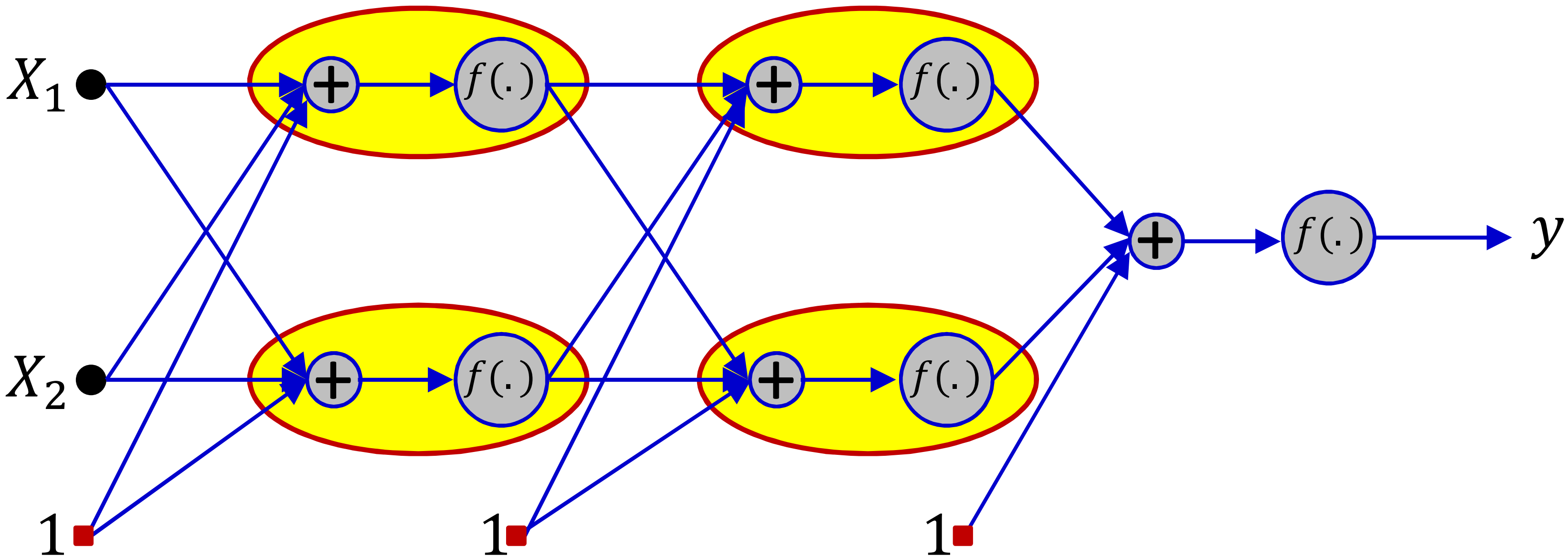
- How to compute $\frac{d\boldsymbol{Div}(\boldsymbol{Y},\boldsymbol{d})}{dw_{i,j}^{(k)}}$

A first closer look at the network



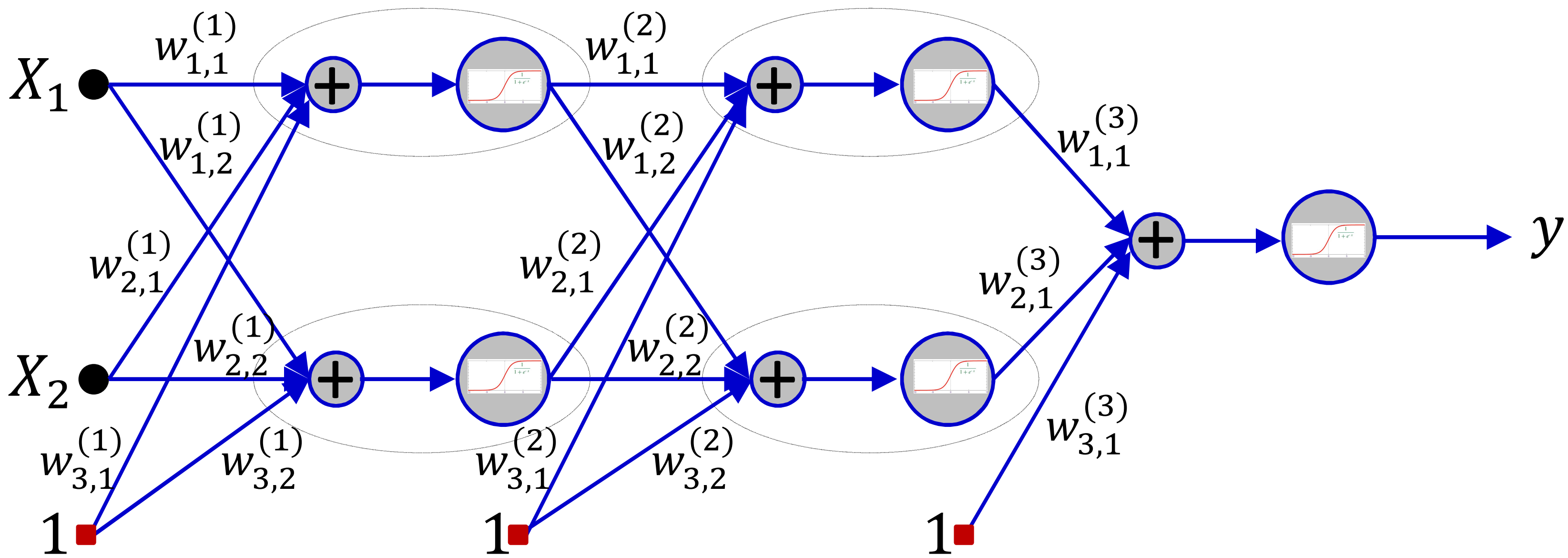
- Showing a tiny 2-input network for illustration
  - Actual network would have many more neurons and inputs

A first closer look at the network



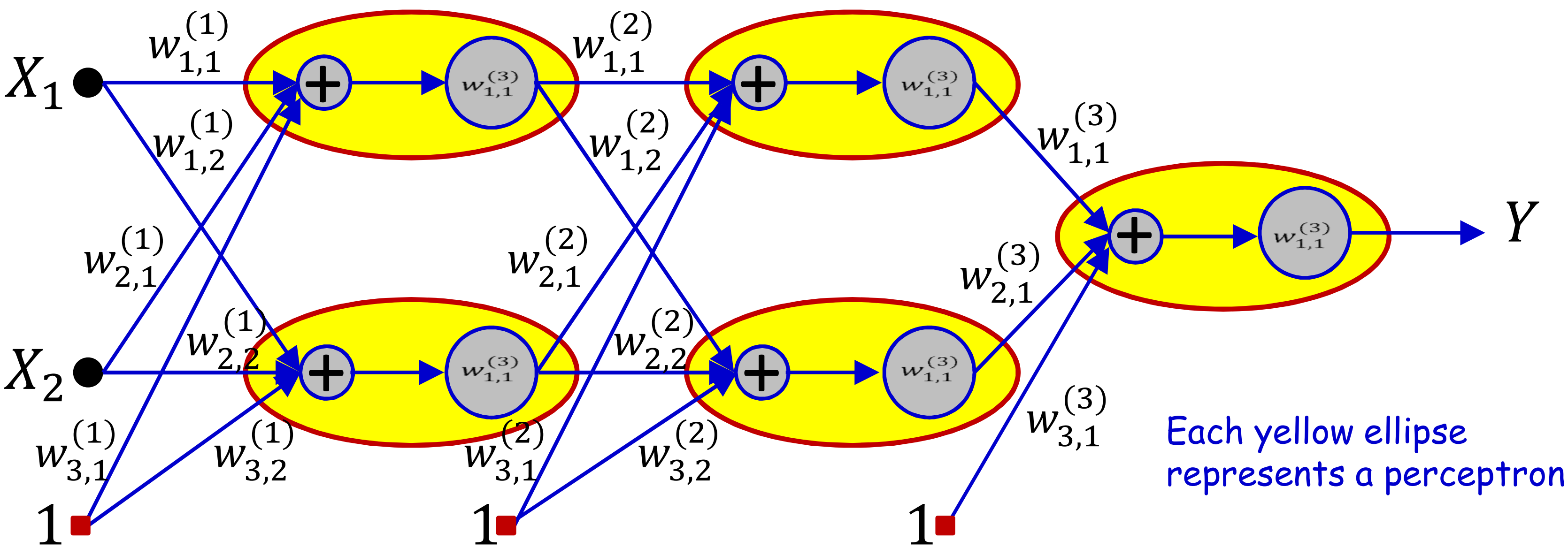
- Showing a tiny 2-input network for illustration
  - Actual network would have many more neurons and inputs
- Explicitly separating the weighted sum of inputs from the activation

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

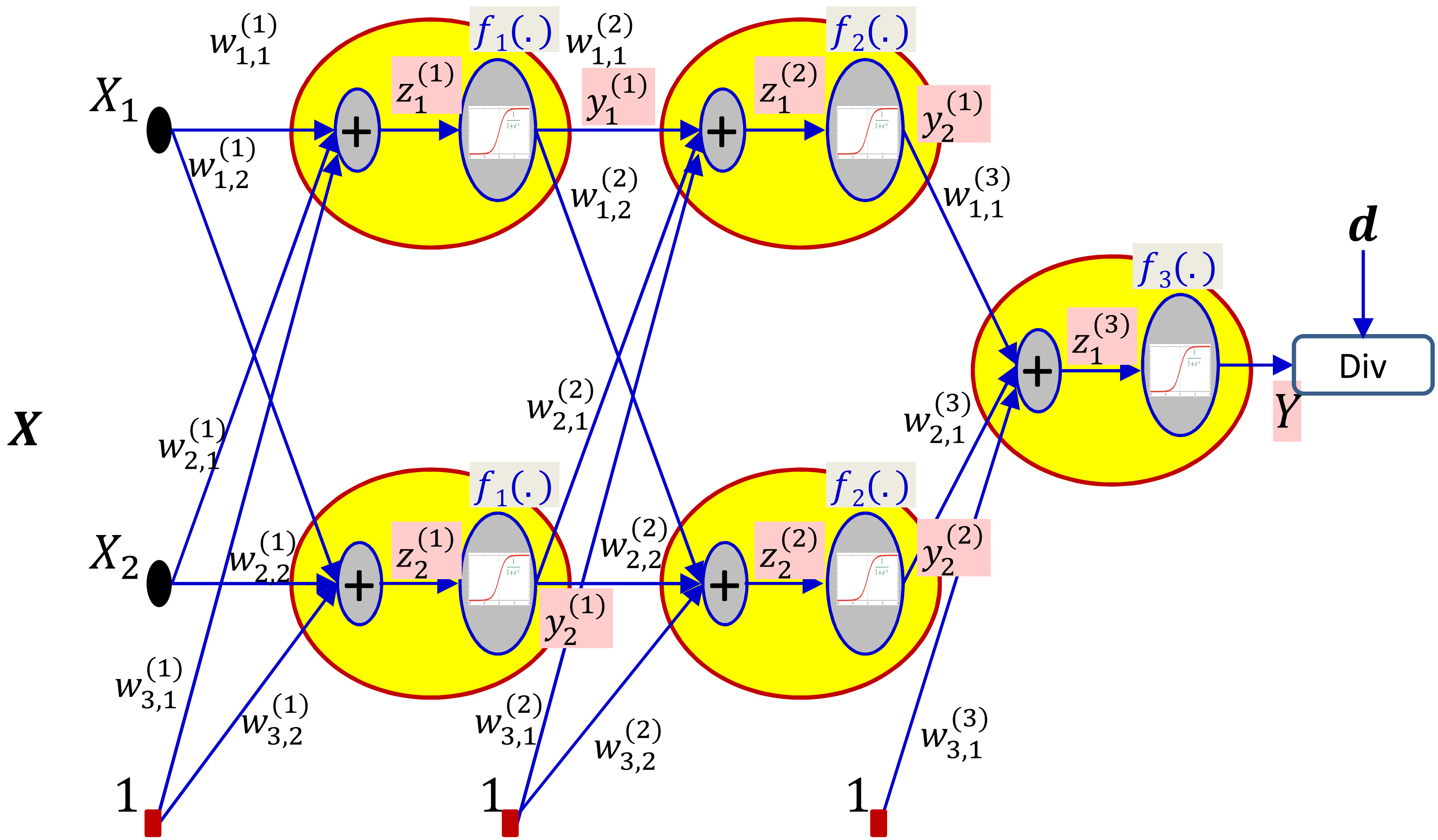
Computing the derivative for a *single* input



- Aim: compute derivative of $Div(Y, d)$ 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: $\frac{dDiv(Y,d)}{dw_{i,j}^{(k)}}$
  - Derive on board?

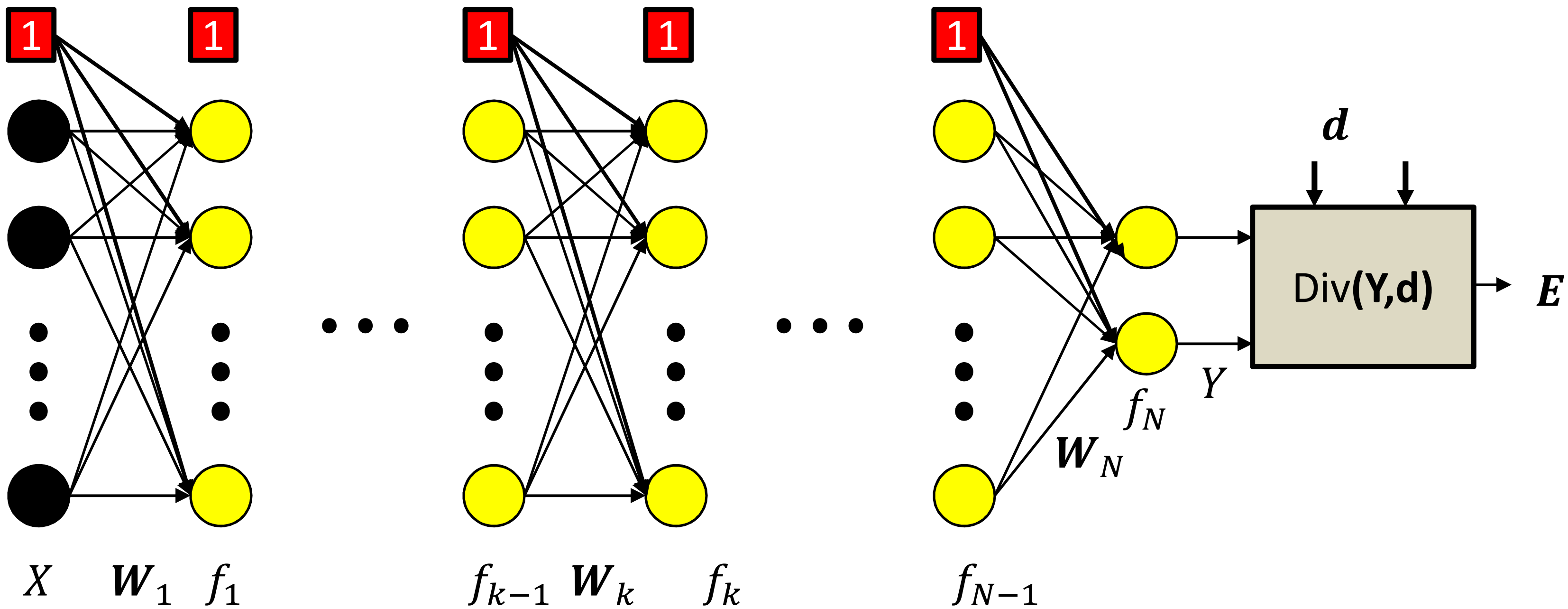
Computing the gradient

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

- Derive on board?

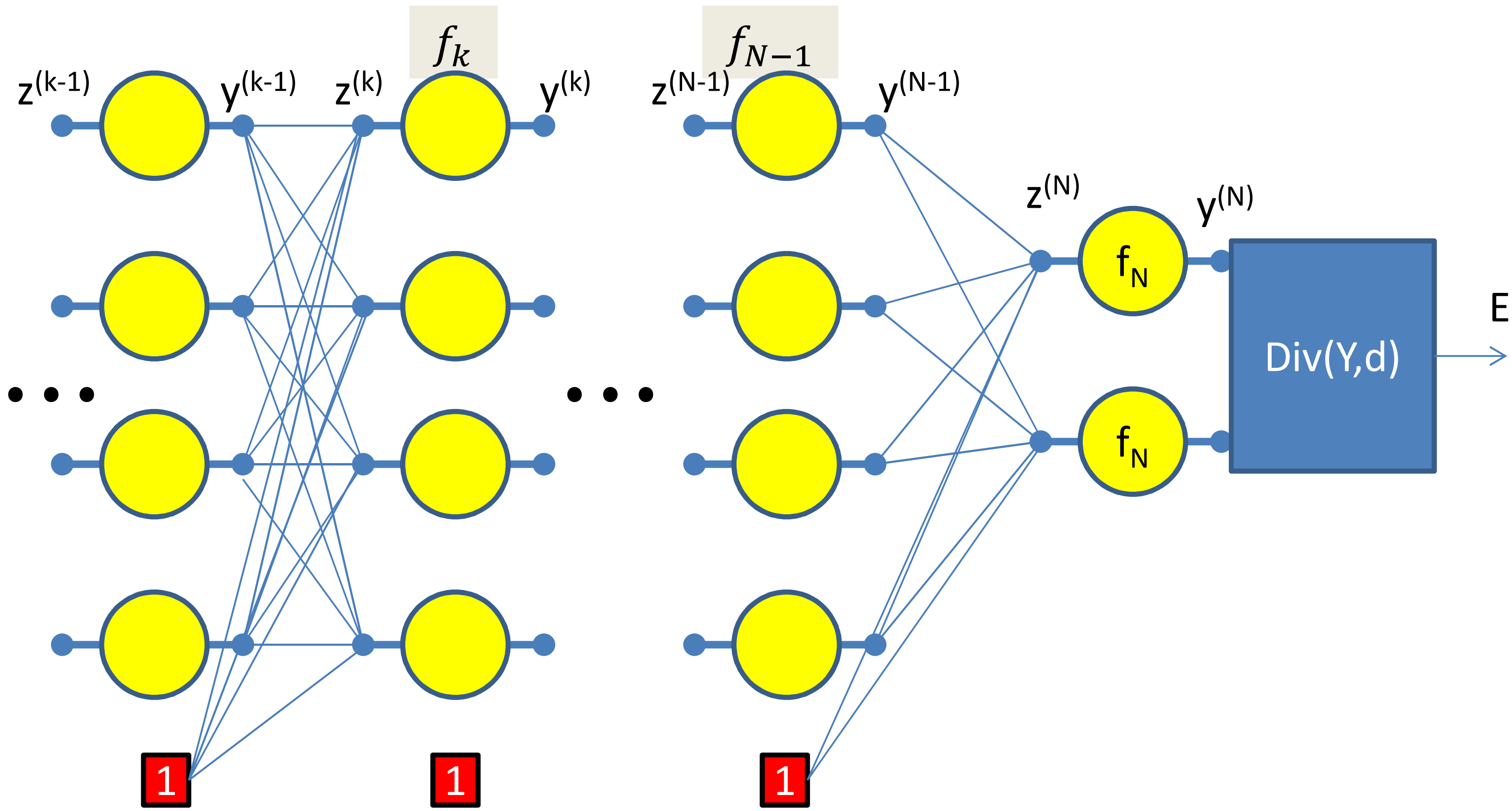
- Note: computation of the derivative requires intermediate and final output values of the network in response to the input

BP: Scalar Formulation



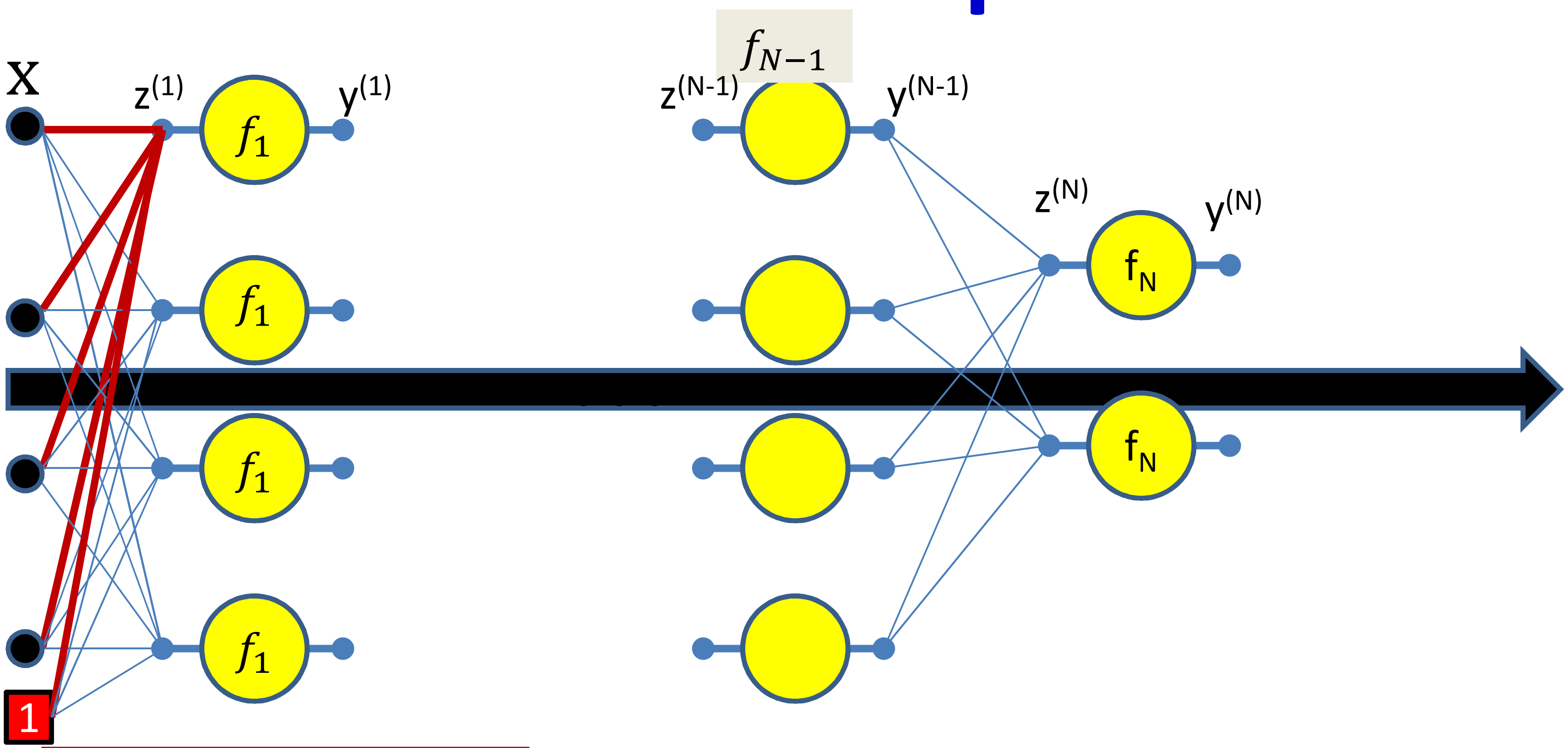
- The network again

Gradients: Local Computation



- Redrawn
- Separately label input and output of each node

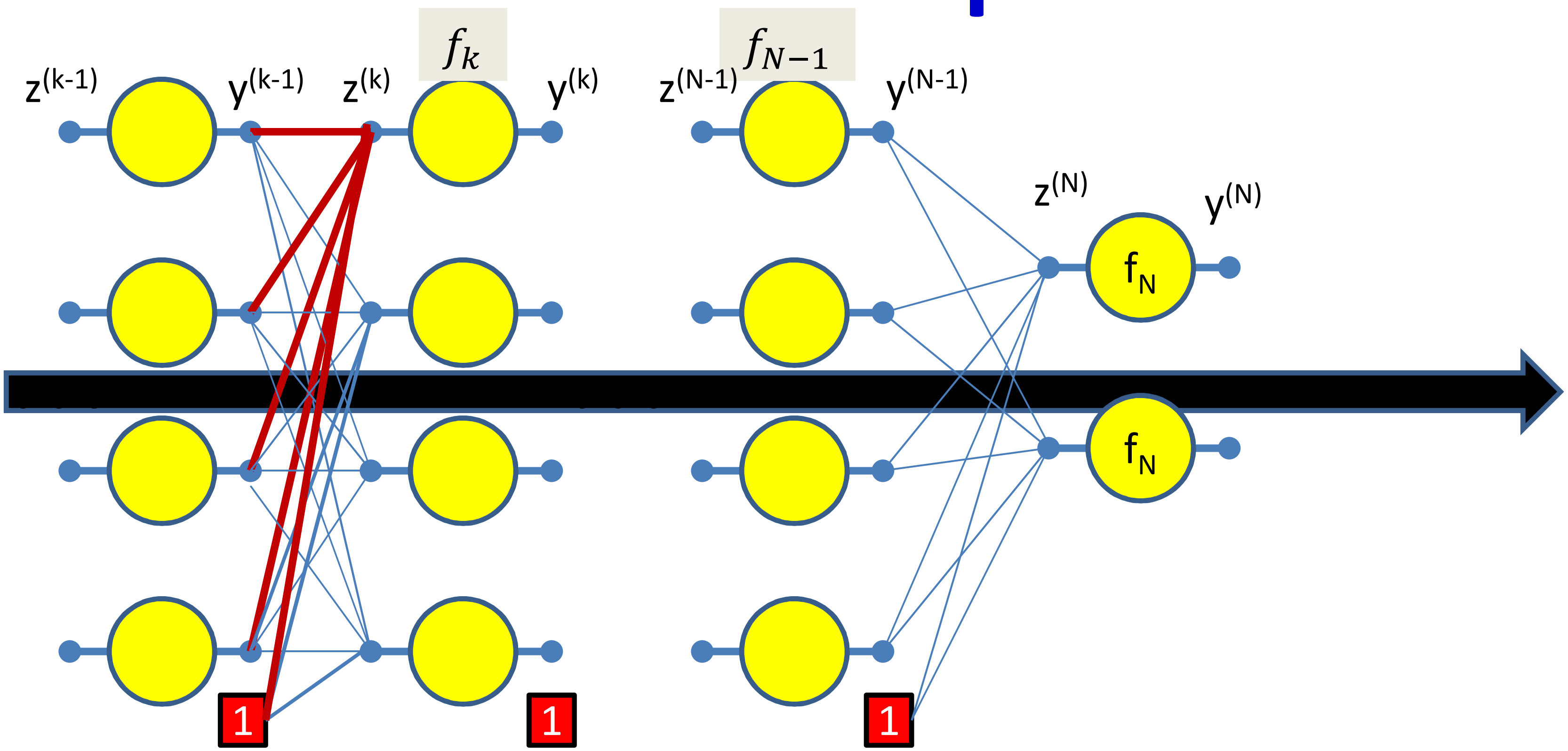
Forward Computation



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

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

Forward Computation

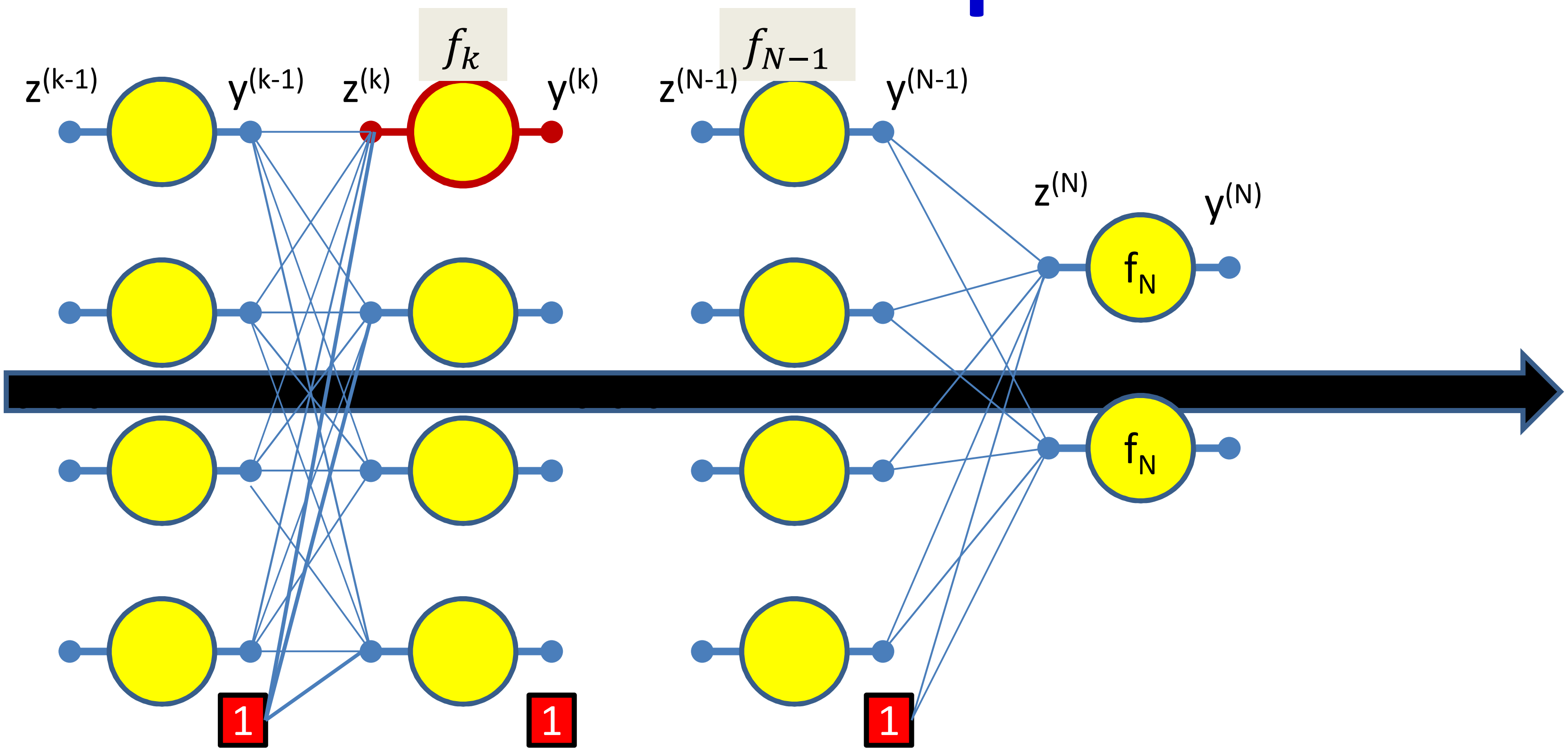


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i$$

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

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

Forward Computation

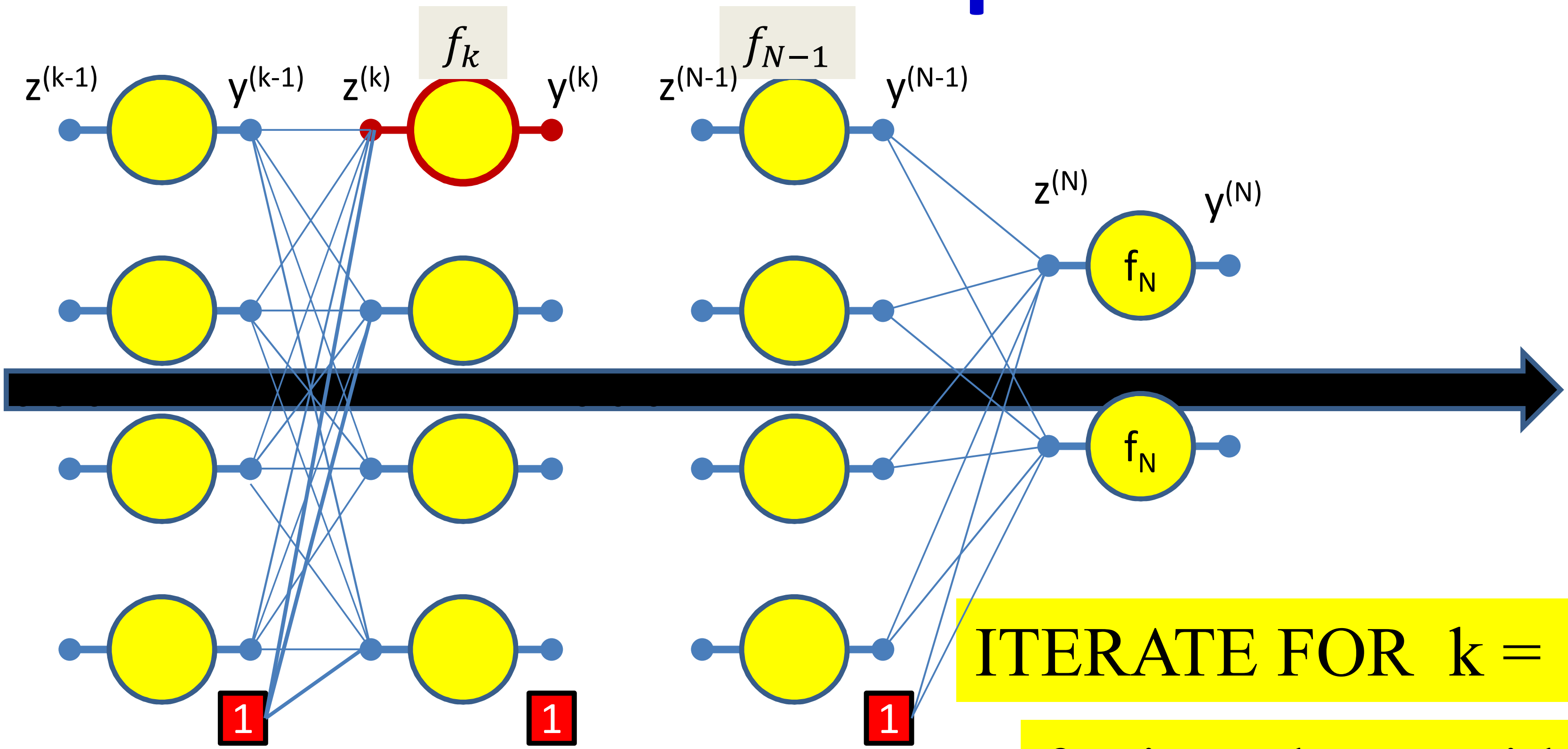


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i$$

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

$$\boxed{y_j^{(k)} = f_k\left(z_j^{(k)}\right)}$$

Forward Computation



ITERATE FOR k = 1:N

for j = 1:layer-width

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

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

$$y_j^{(k)} = f_k\left(z_j^{(k)}\right)$$

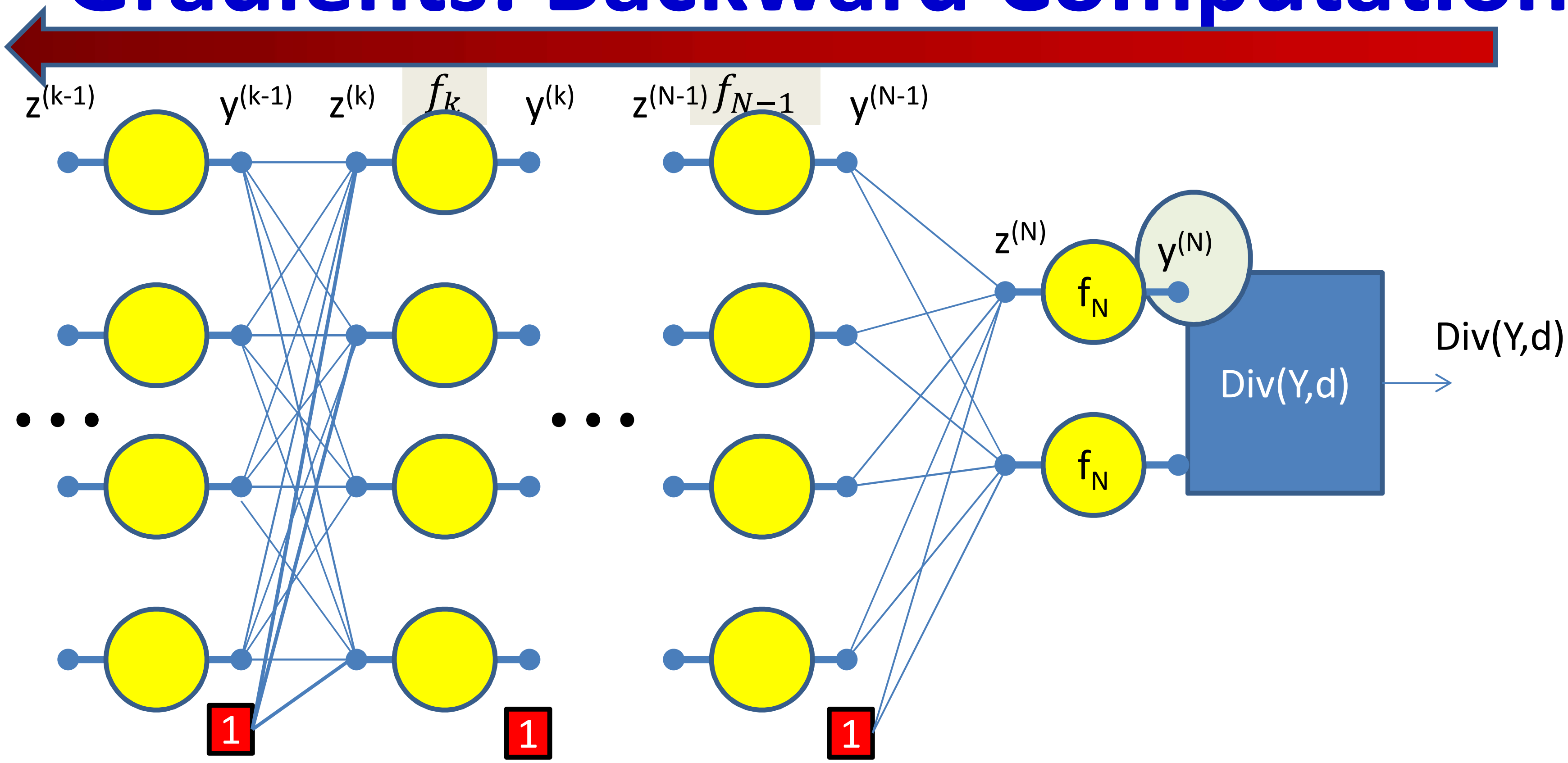
Forward "Pass"

- Input: D dimensional vector $\mathbf{x} = [x_j,\ j = 1 \dots D]$
- Set:
  - $D_0 = D,$ is the width of the 0th (input) layer
  - $y_j^{(0)} = x_j,\ j = 1 \dots D; \quad y_0^{(k=1\dots N)} = x_0 = 1$
- For layer $k = 1 \dots N$
  - For $j = 1 \dots D_k$

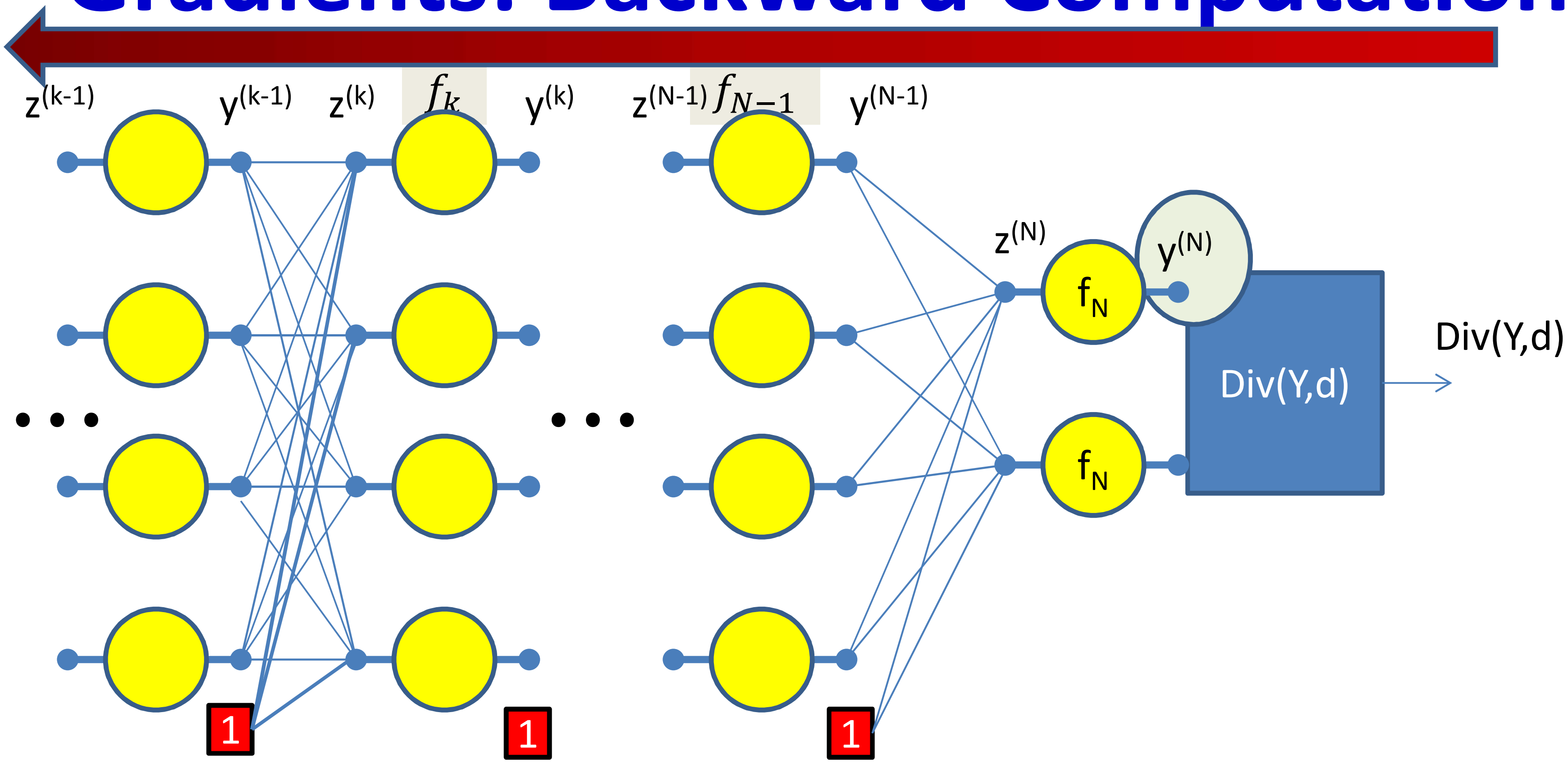
    D_k is the size of the kth layer

    - $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)} y_i^{(k-1)}$
    - $y_j^{(k)} = f_k\left(z_j^{(k)}\right)$
- Output:
  - $Y = y_j^{(N)}, j = 1..D_N$

Gradients: Backward Computation

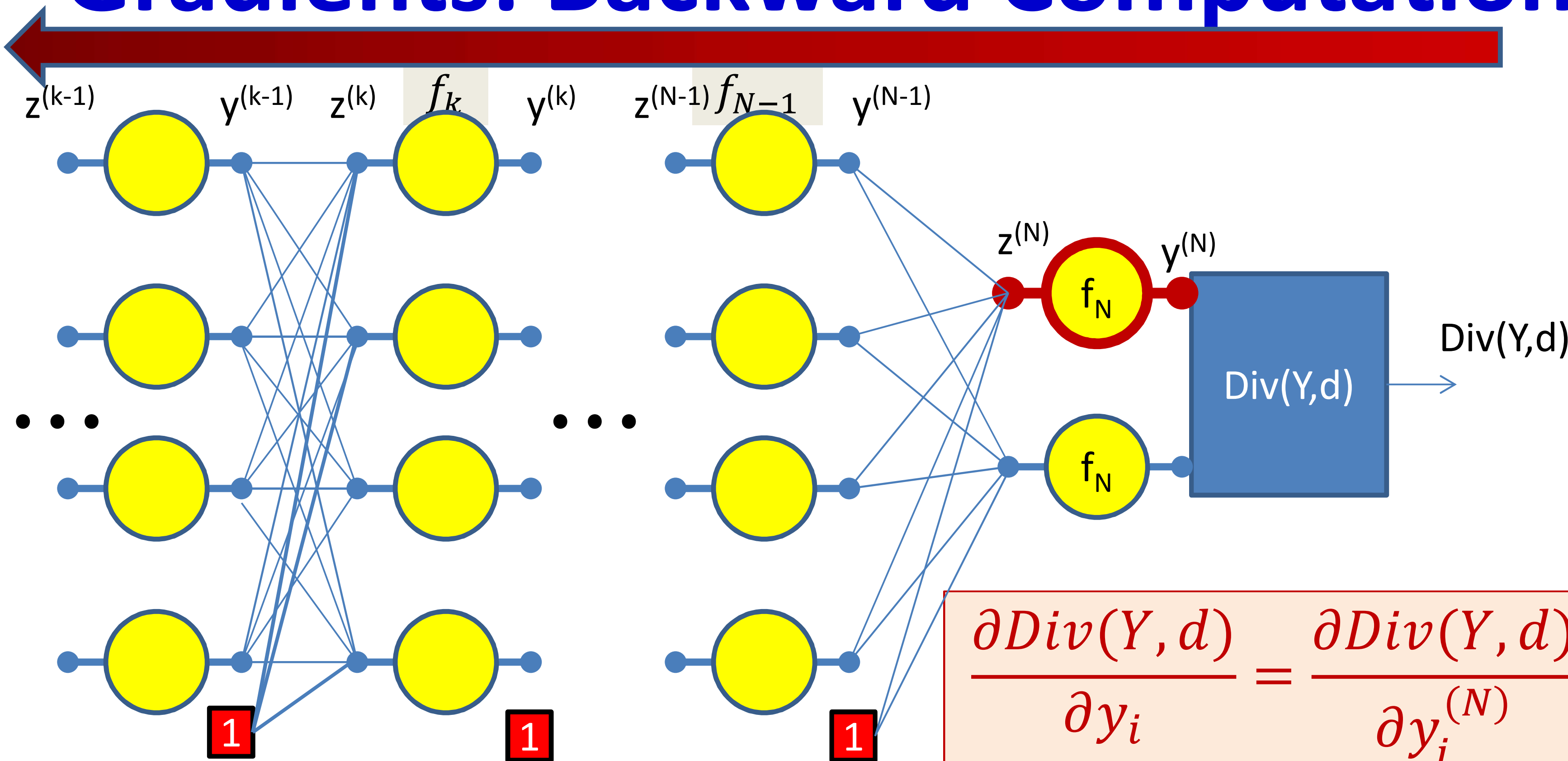


Gradients: Backward Computation



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

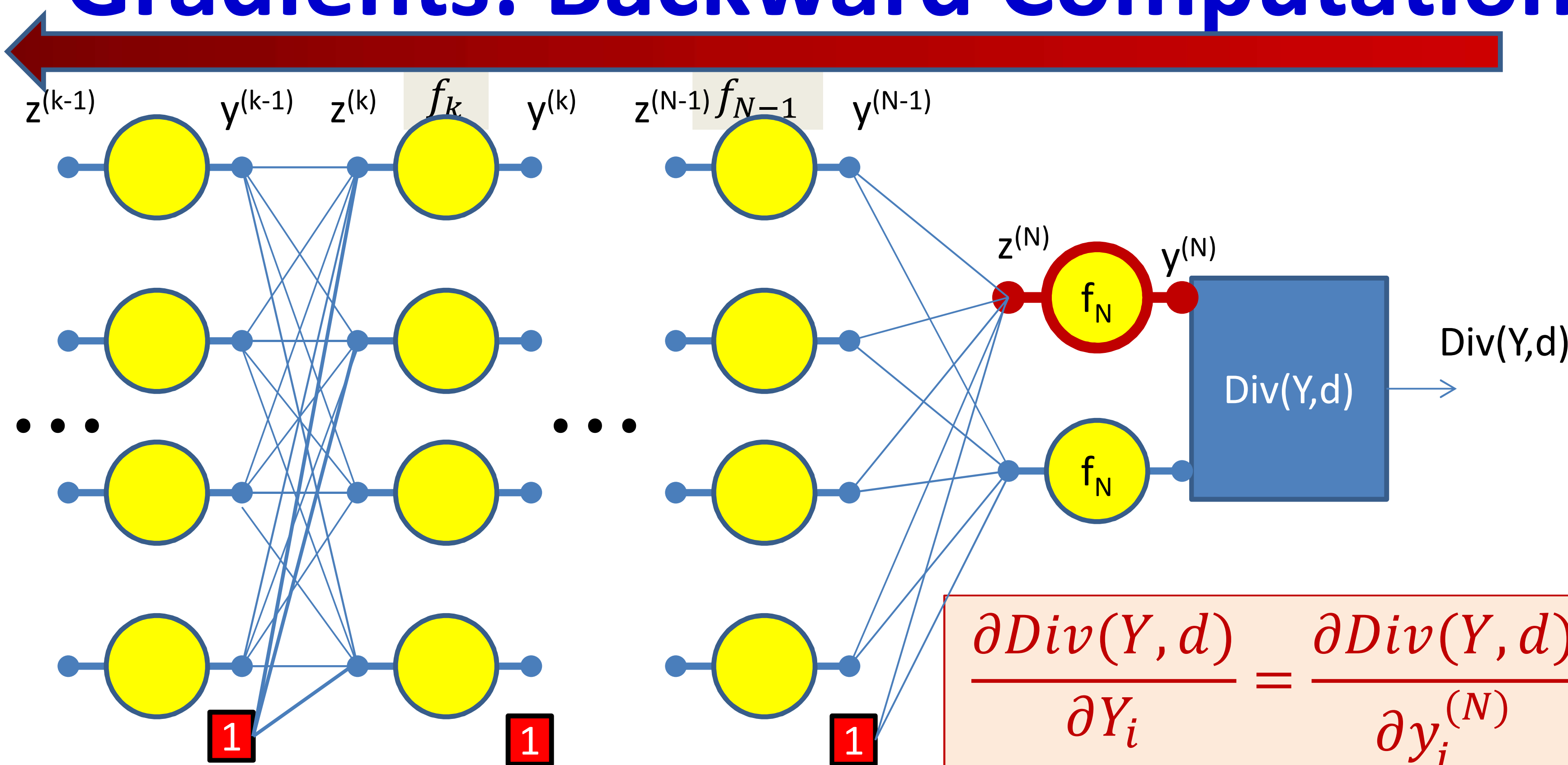
Gradients: Backward Computation



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

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

Gradients: Backward Computation

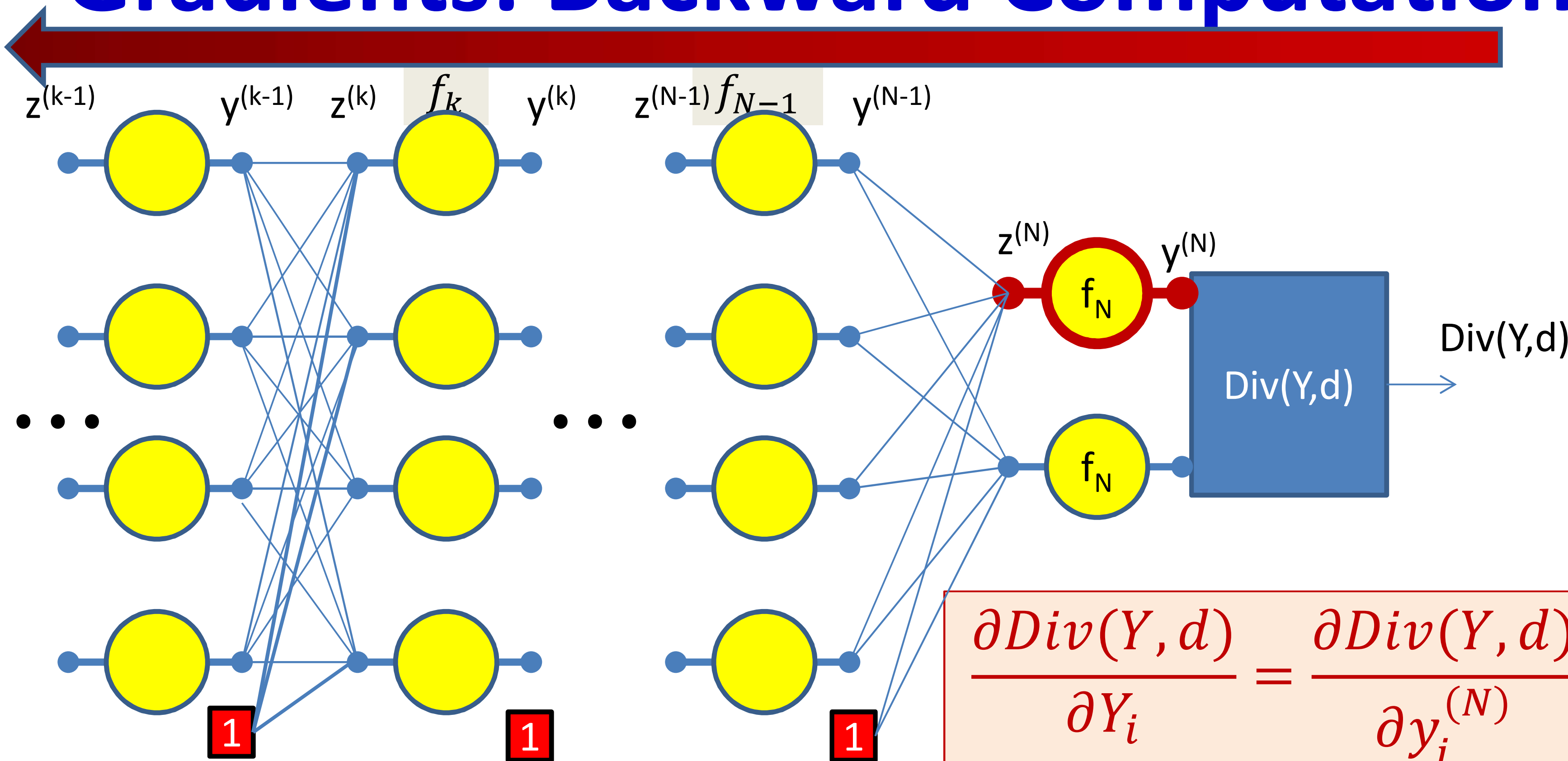


$$\frac{\partial Div(Y,d)}{\partial Y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

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

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

Gradients: Backward Computation

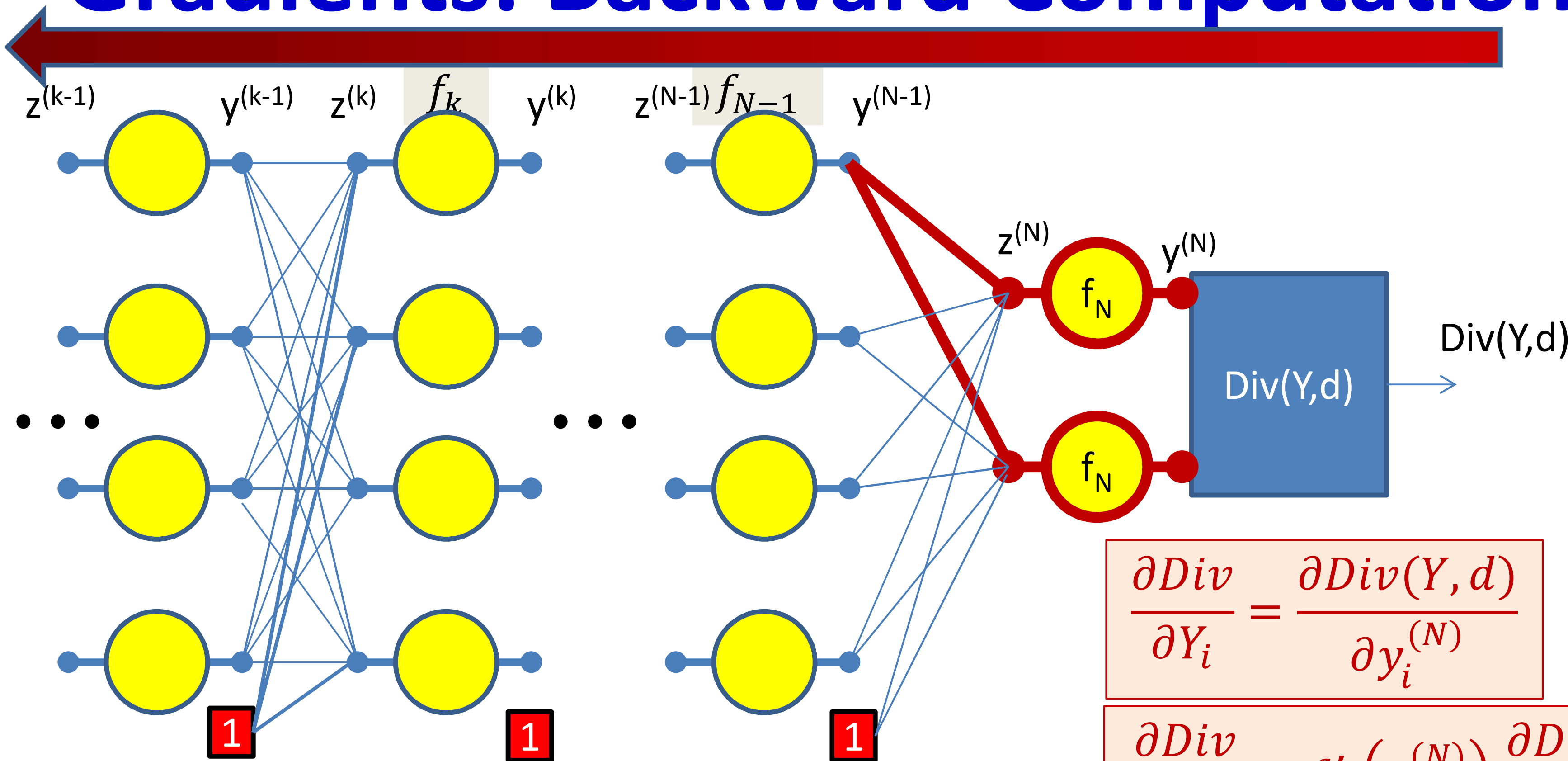


$$\frac{\partial Div(Y,d)}{\partial Y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

Derivative of the activation function of Nth layer

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

Gradients: Backward Computation



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

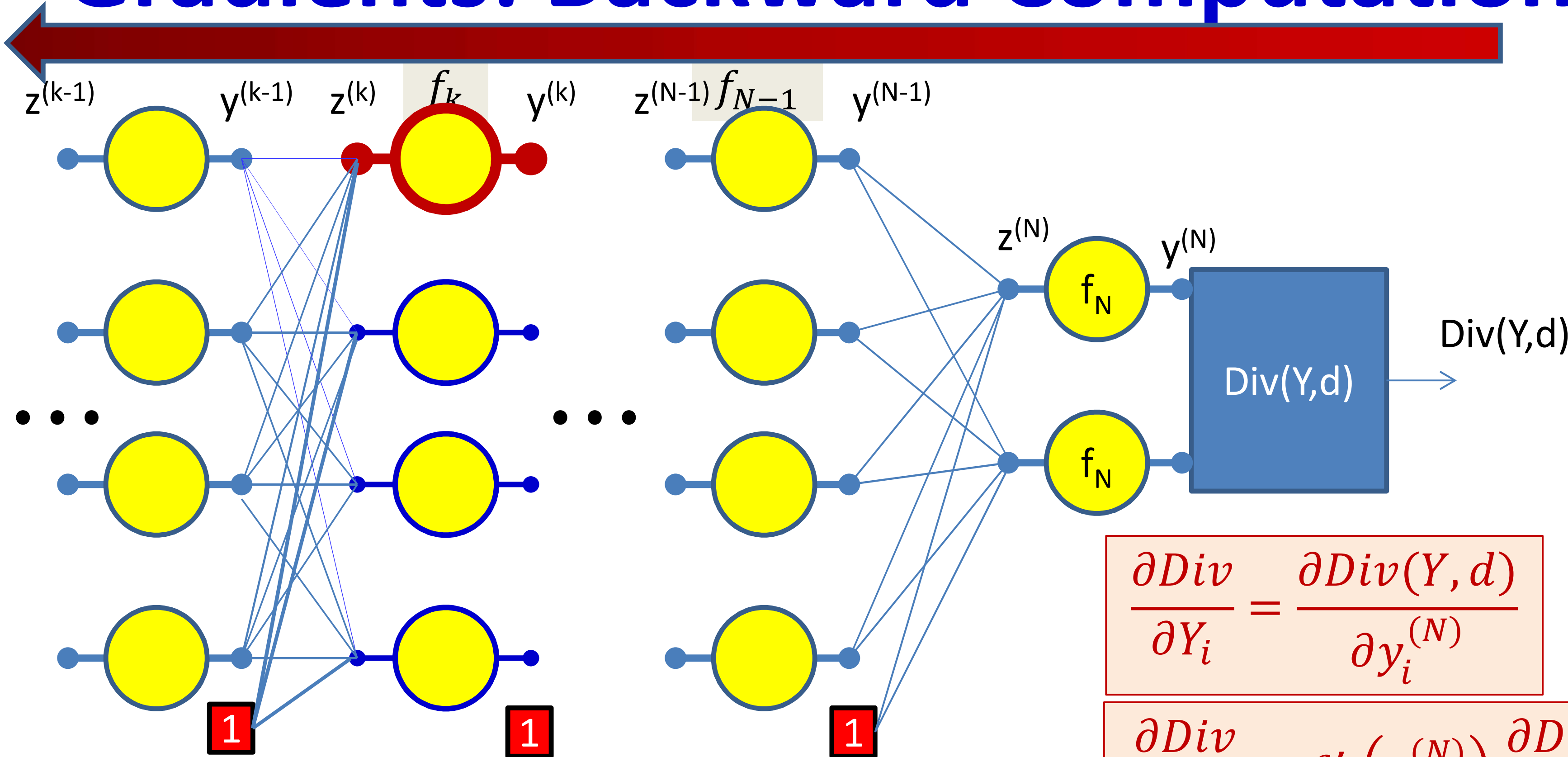
$$\frac{\partial Div}{\partial z_i^{(N)}} = f_N'\left(z_i^{(N)}\right)\frac{\partial Div}{\partial y_i^{(N)}}$$

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

Because :

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

Gradients: Backward Computation



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

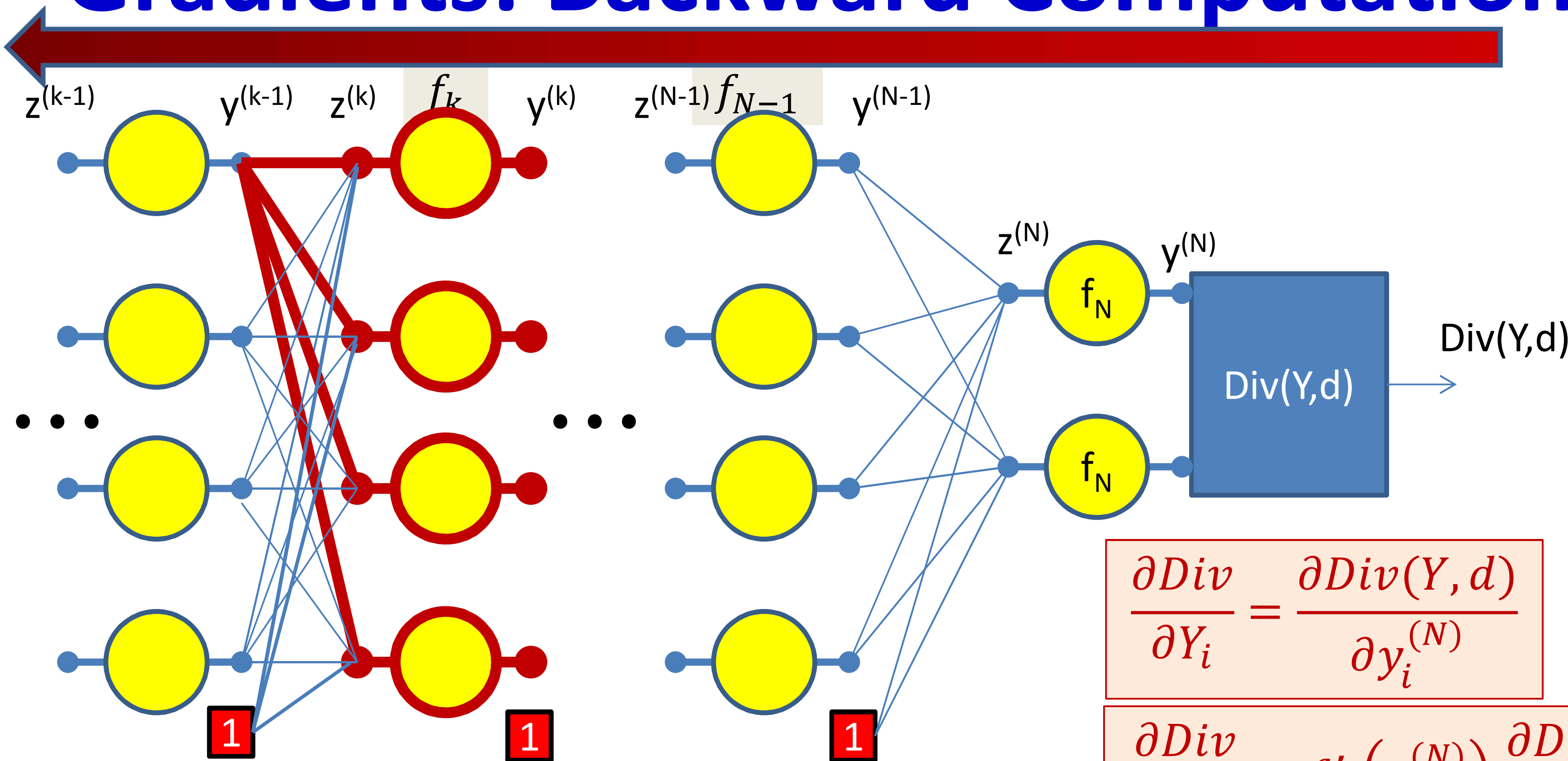
$$\frac{\partial Div}{\partial z_i^{(N)}} = f_N'\left(z_i^{(N)}\right)\frac{\partial Div}{\partial y_i^{(N)}}$$

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

computed during
the forward pass

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

Gradients: Backward Computation

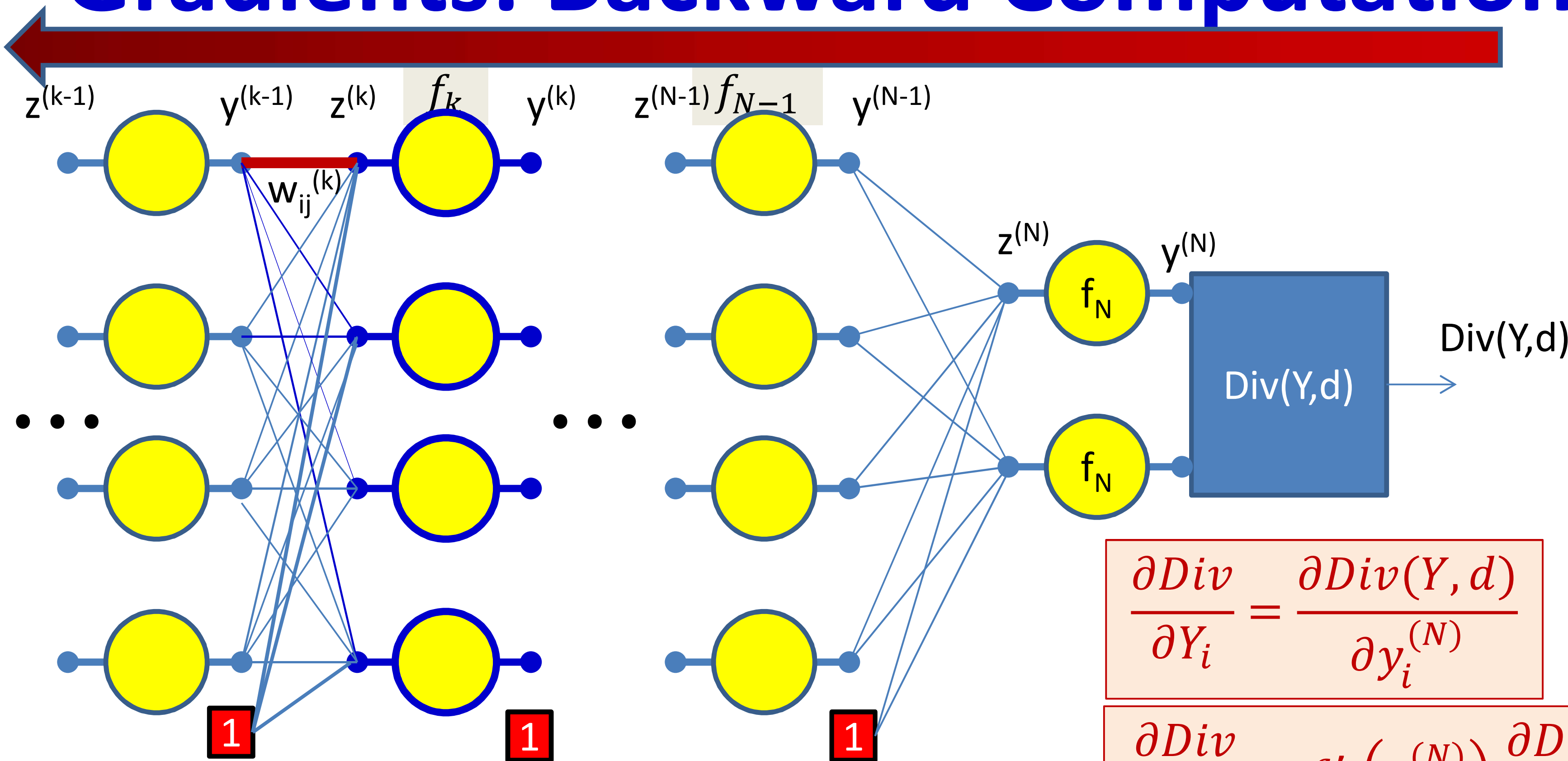


$$\frac{\partial Div}{\partial Y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

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

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

Gradients: Backward Computation



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

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

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

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

Gradients: Backward Computation

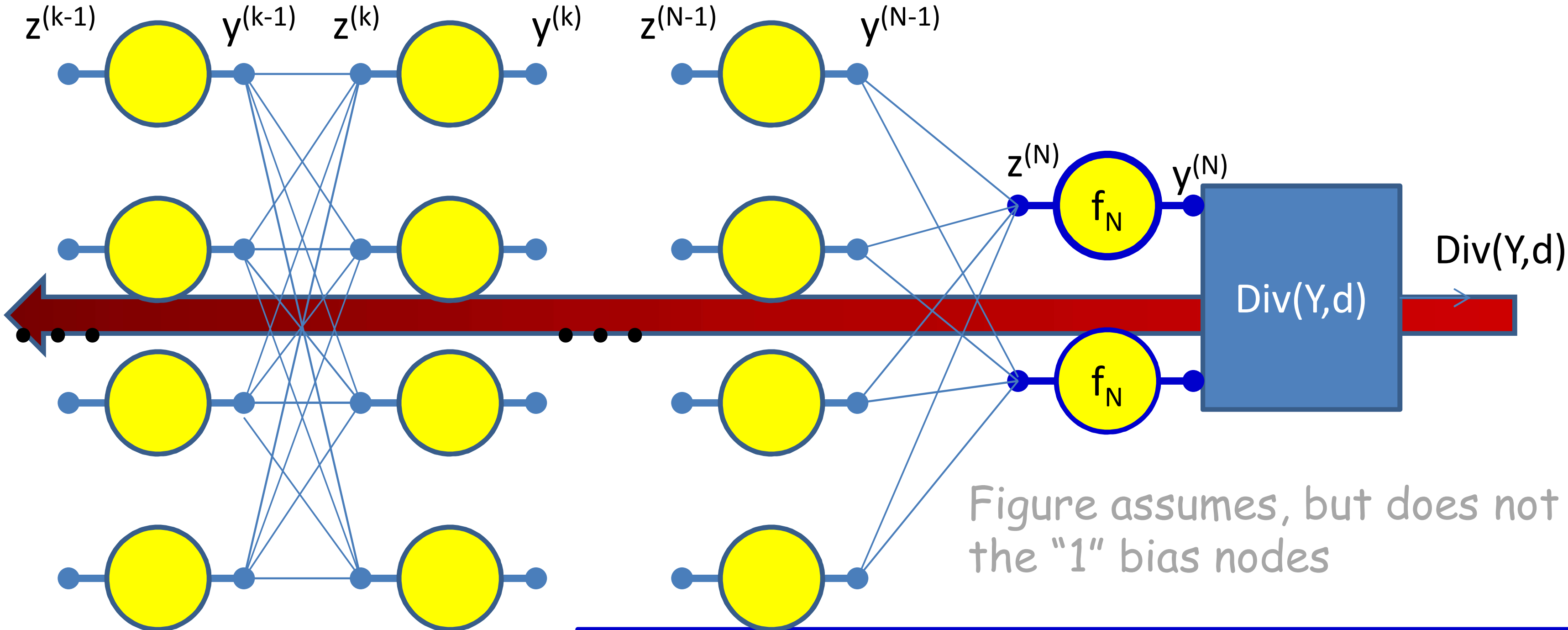


Figure assumes, but does not show the "1" bias nodes

Initialize: Gradient w.r.t network output

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

$For\ k\ =\ N..1$

$\quad For\ i\ =\ 1: layer-width$

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

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

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

Backward Pass

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

Backward Pass

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

Called "Backpropagation" because the derivative of the error is propagated "backwards" through the network

Very analogous to the forward pass:

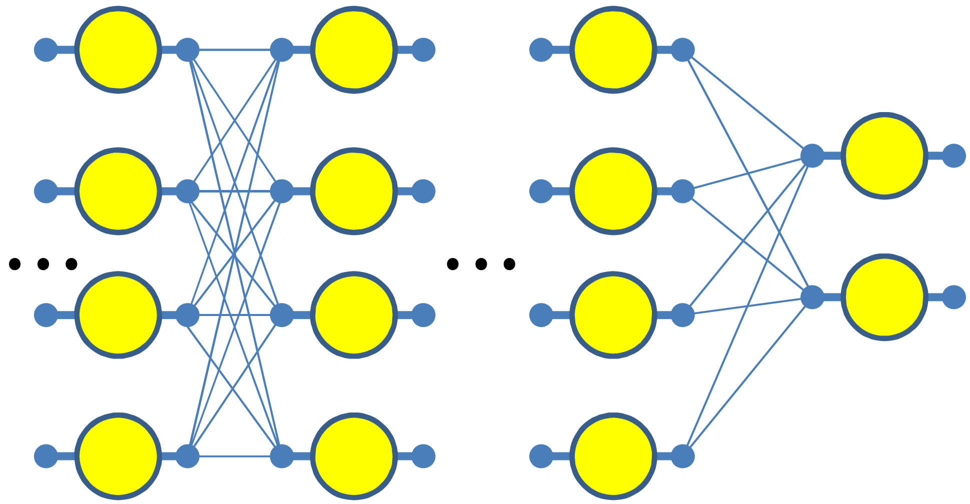
Backward weighted combination of next layer

Backward equivalent of activation

For comparison: the forward pass again

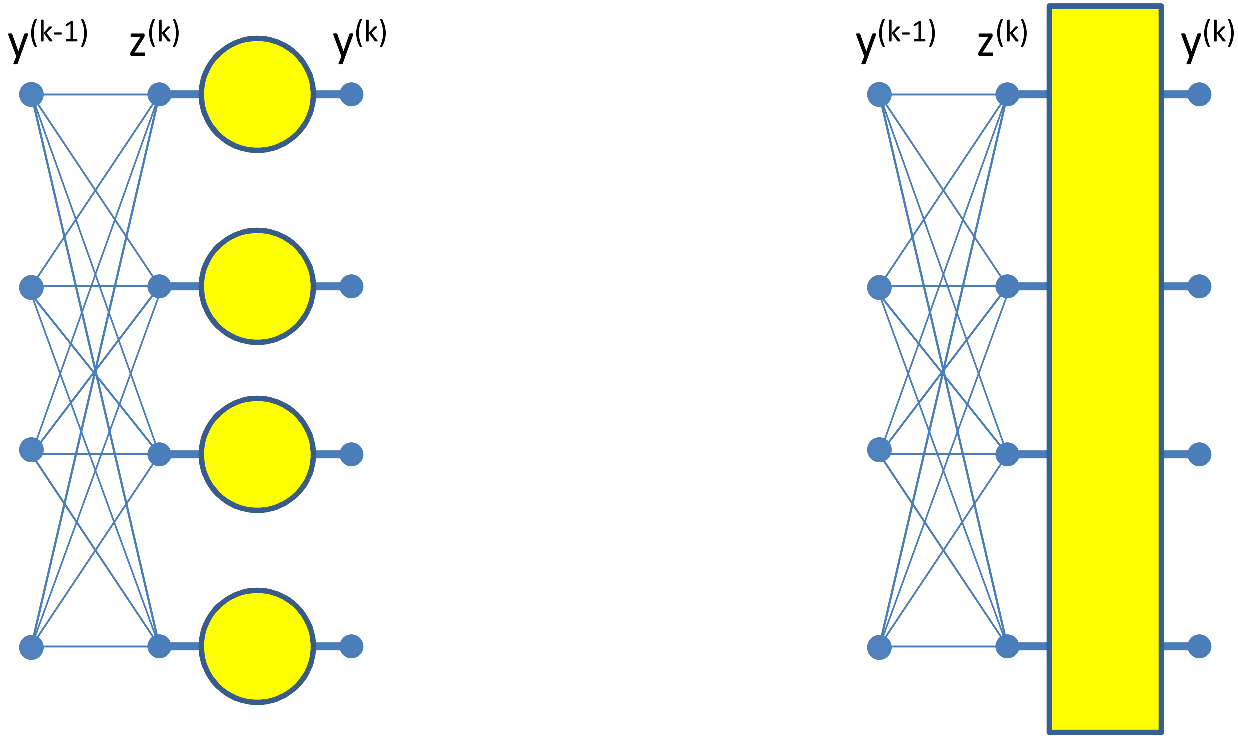
- Input: D dimensional vector $\mathbf{x} = [x_j,\ j = 1 \dots D]$
- Set:
  - $D_0 = D,$ is the width of the 0th (input) layer
  - $y_j^{(0)} = x_j,\ j = 1 \dots D; \qquad y_0^{(k=1\dots N)} = x_0 = 1$
- For layer $k = 1 \dots N$
  - For $j = 1 \dots D_k$
    - $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)}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

Special cases



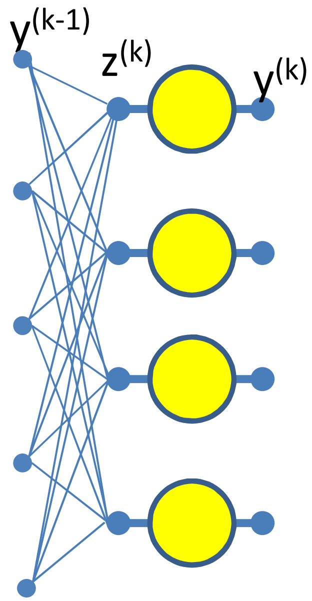
- Have assumed so far that
  1. The computation of the output of one neuron does not directly affect computation of other neurons in the same (or previous) layers
  2. Outputs of neurons only combine through weighted addition
  3. Activations are actually differentiable
  - All of these conditions are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

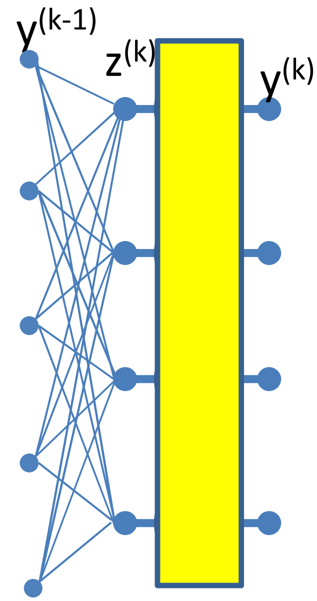
Special Case 1. Vector activations



- Vector activations: all outputs are functions of all inputs

Special Case 1. Vector activations





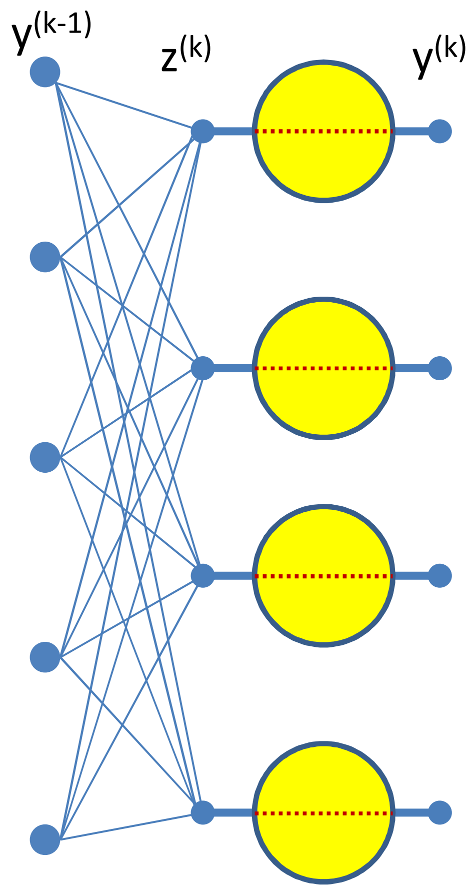
Scalar activation: Modifying a z_i only changes corresponding y_i

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

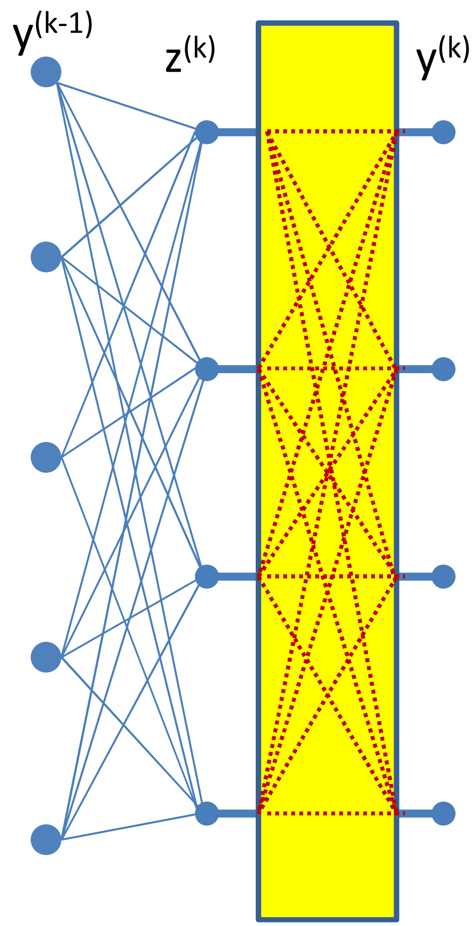
Vector activation: Modifying a z_i potentially changes all, $y_1 \ldots y_M$

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

“Influence” diagram

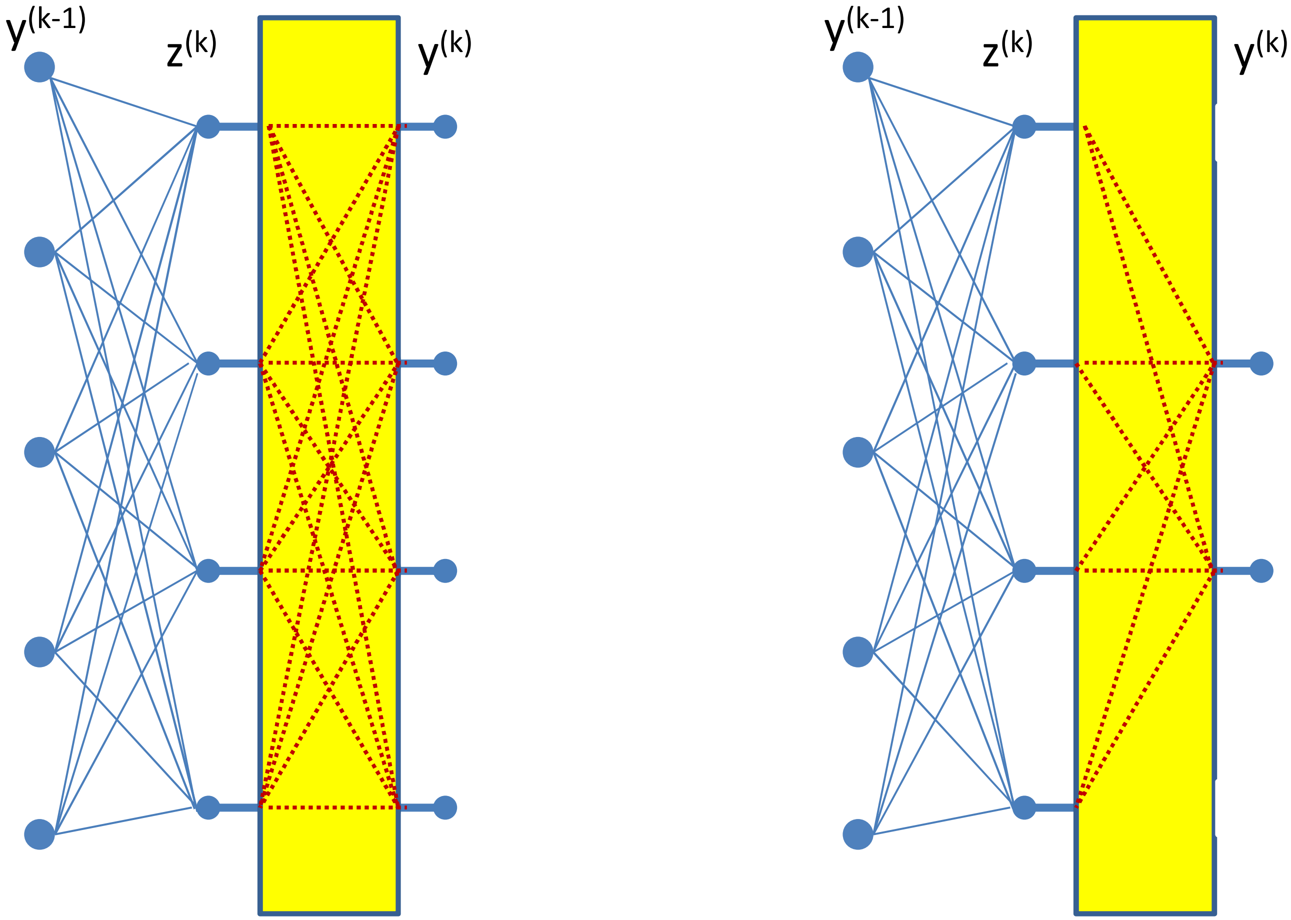


Scalar activation: Each z_i influences *one* y_i



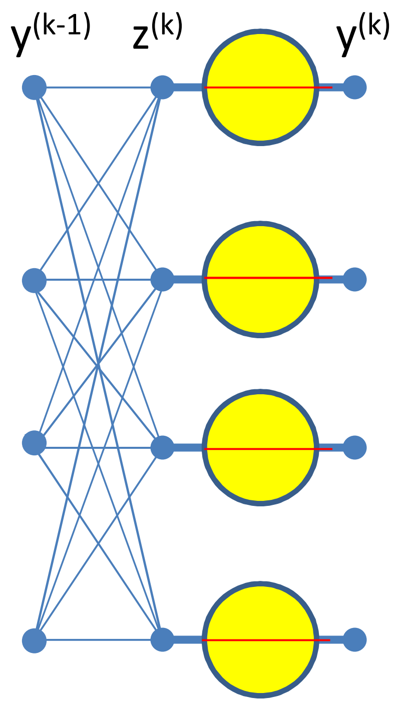
Vector activation: Each z_i influences all, $y_1 \ldots y_M$

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

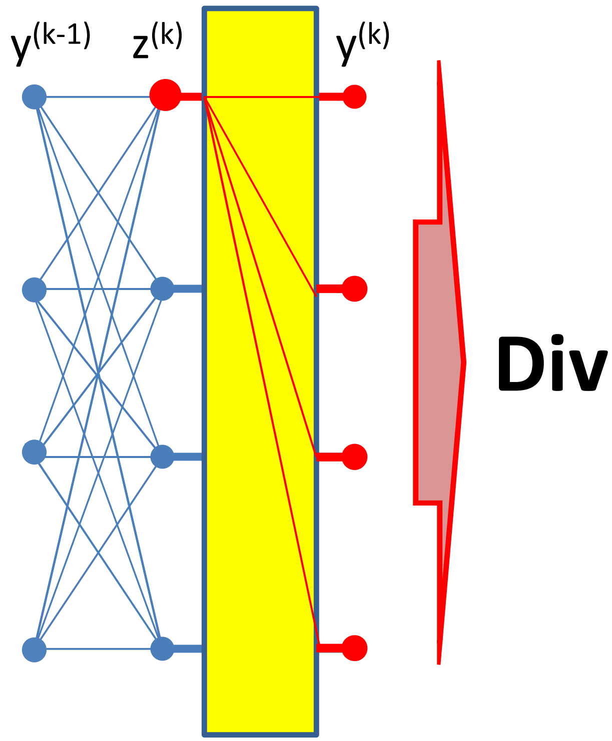
Scalar Activation: Derivative rule



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

- 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



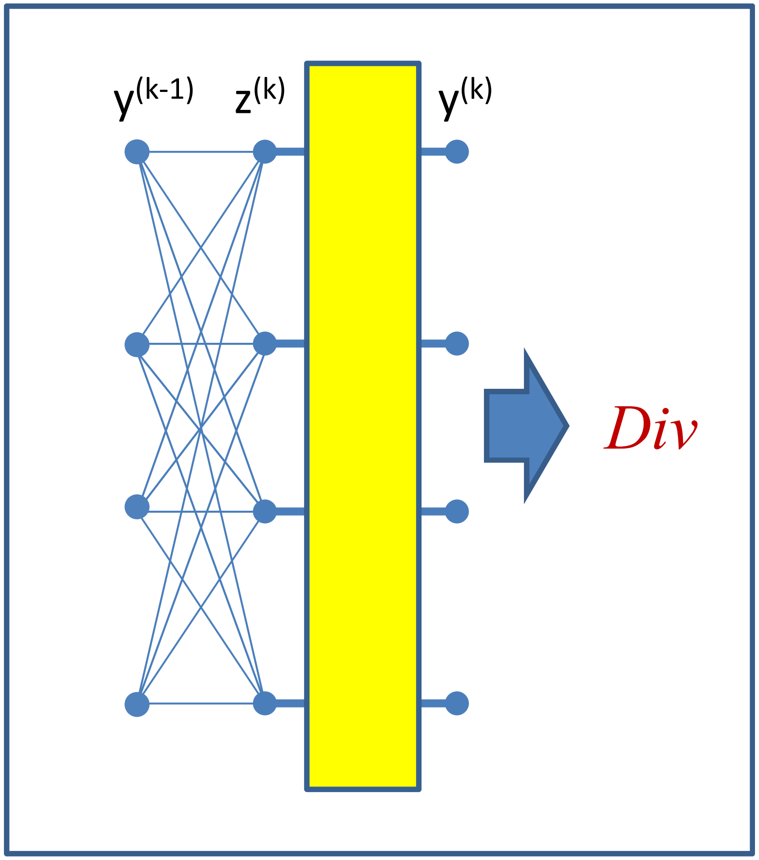
$$\frac{\partial Div}{\partial z_i^{(k)}} = \sum_j \frac{\partial Div}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}}$$

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

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

- 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



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

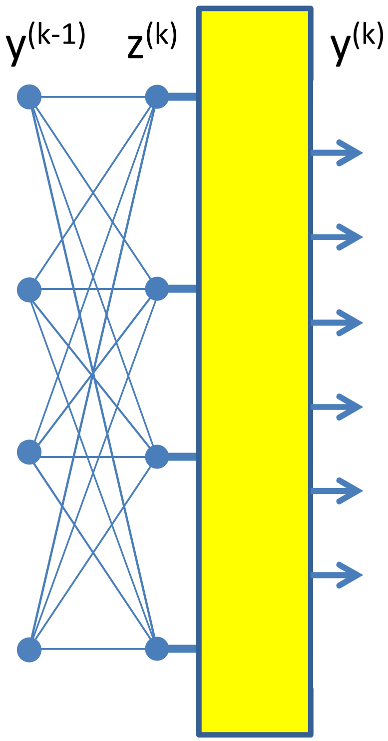
$$\boxed{\frac{\partial Div}{\partial z_i^{(k)}} = \sum_j \frac{\partial Div}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}}}$$

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

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

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

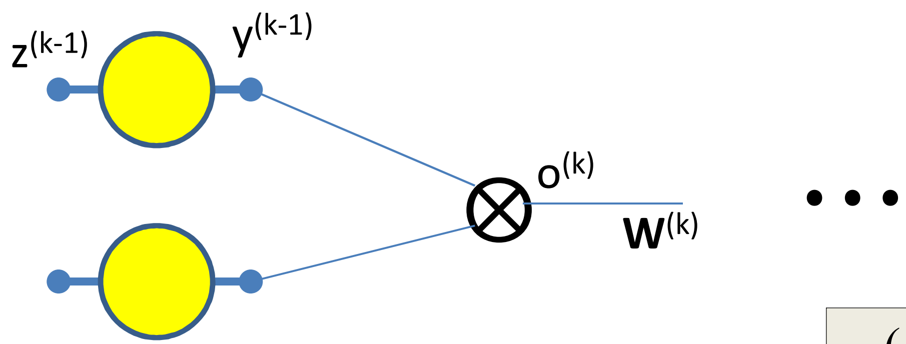
Vector Activations



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

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

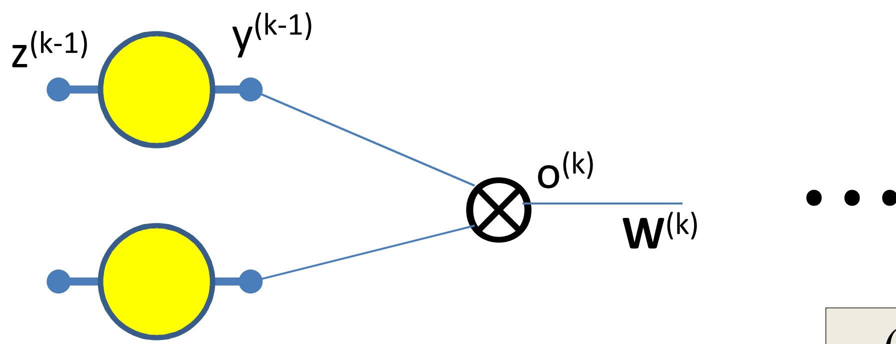
Special Case 2: Multiplicative networks



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

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

Backpropagation: Multiplicative Networks



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

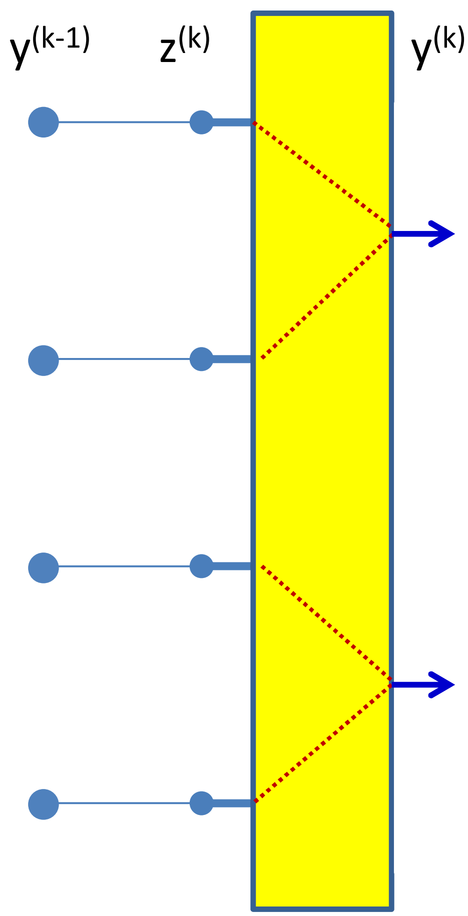
Backward:

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

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

- Some types of networks have *multiplicative* combination

Multiplicative combintion as a case of vector activations

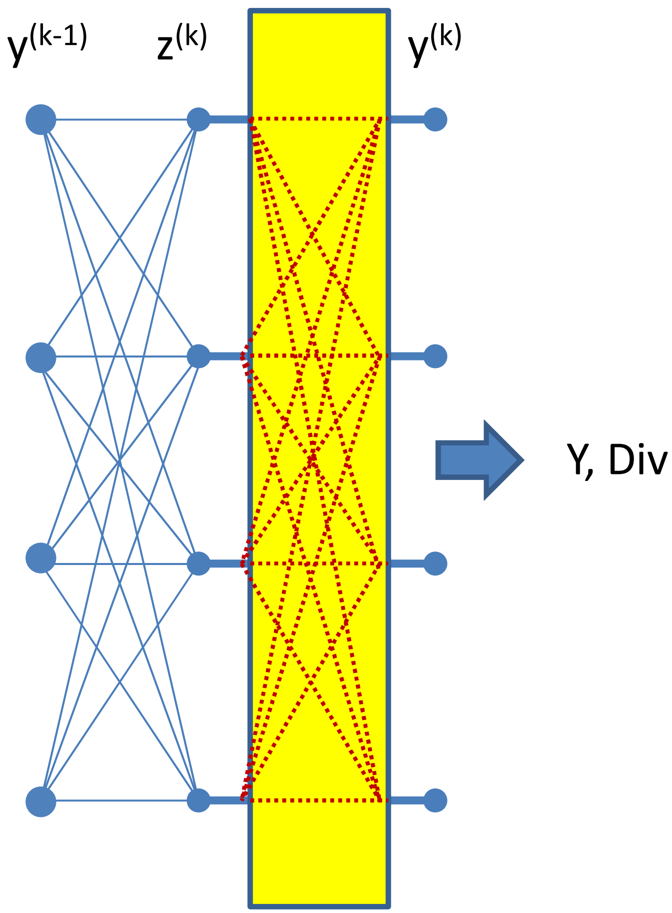


$$z_i^{(k)} = y_i^{(k-1)}$$

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

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

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



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

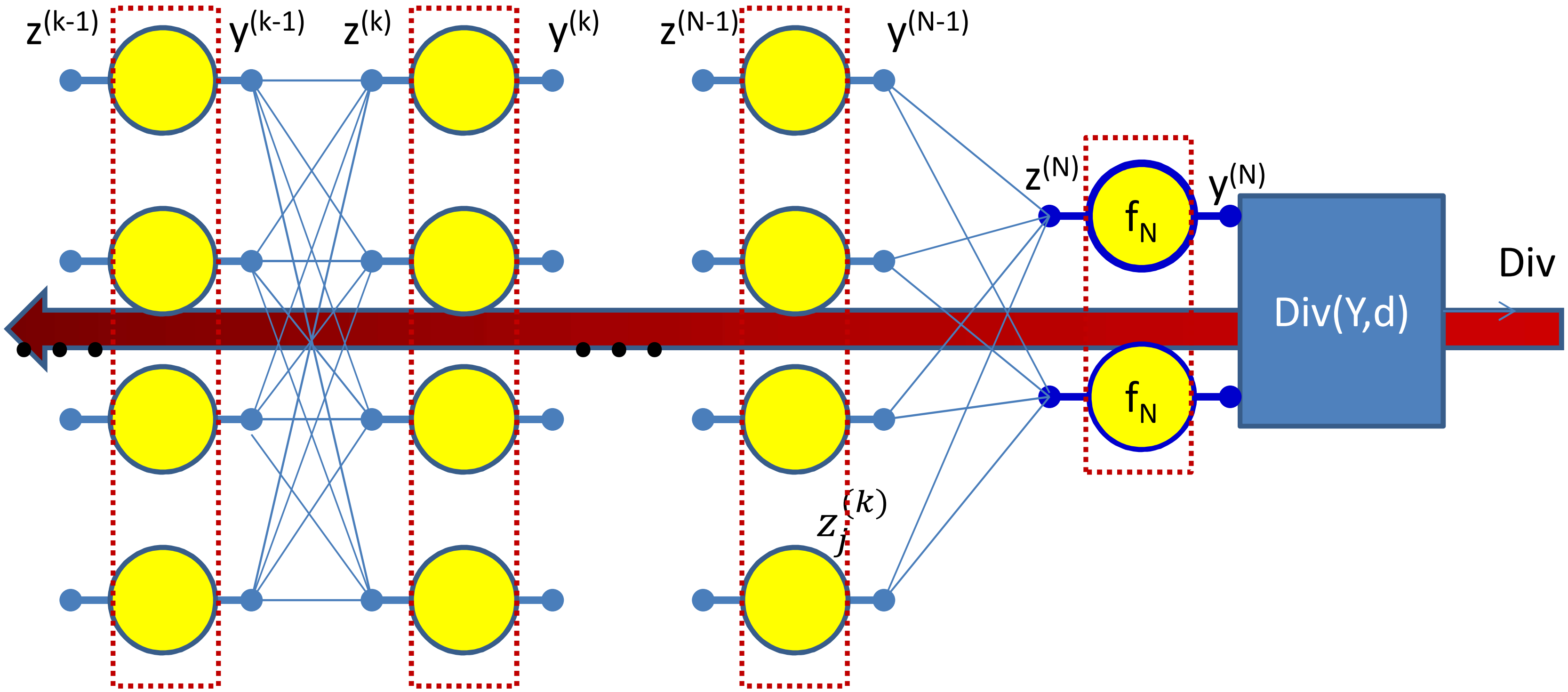
$$y_i^{(k)} = \prod_l \left(z_l^{(k)}\right)^{\alpha_{li}^{(k)}}$$

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

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

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

Gradients: Backward Computation



For $k = N...1$

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

If layer has vector activation

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

$\Leftrightarrow$

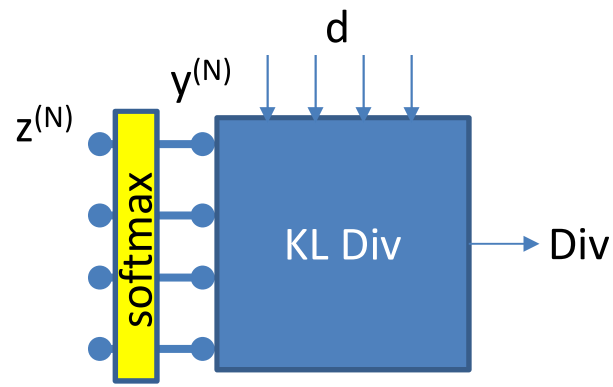
Else if activation is scalar

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

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

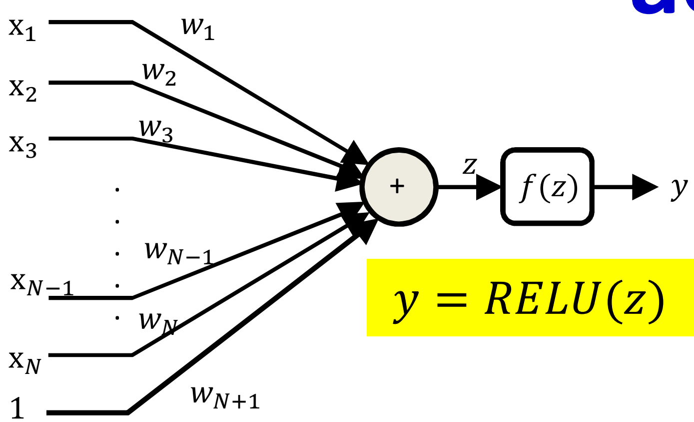
$$\boxed{\frac{\partial Div}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial Div}{\partial z_j^{(k)}}}$$

Backward Pass for *softmax output layer*

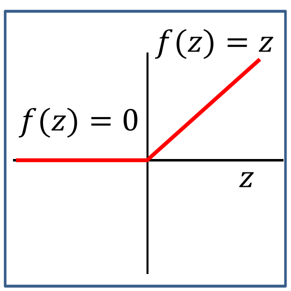


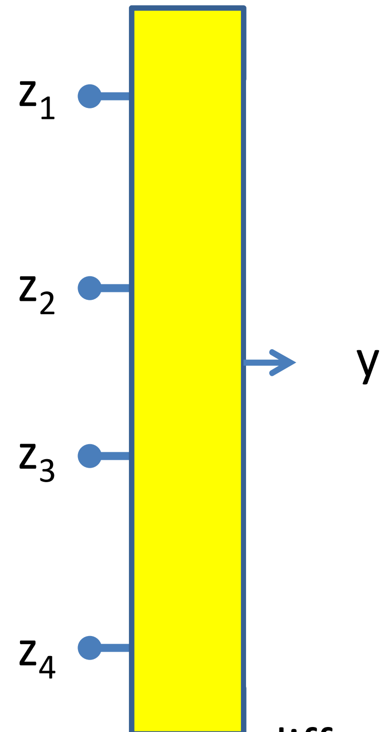
- Output layer (N) :
  - For $i = 1 \dots D_N$
    - $\frac{\partial Div}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$
    - $\frac{\partial Div}{\partial z_i^{(N)}} = \sum_j \frac{\partial D \quad (Y,d)}{\partial y_j^{(N)}} y_i^{(N)} \left(\delta_{ij} - y_j^{(N)}\right)$
- For layer $k = N - 1\ downto\ 0$
  - For $i = 1 \dots D_k$
    - $\frac{\partial Div}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial Div}{\partial z_j^{(k+1)}}$
    - $\frac{\partial Div}{\partial z_i^{(k)}} = f_k'\left(z_i^{(k)}\right) \frac{\partial Div}{\partial y_i^{(k)}}$
    - $\frac{\partial Di}{\partial w_{ij}^{(k+1)}} = y_j^{(k)} \frac{\partial Di}{\partial z_i^{(k+1)}}$ for $j = 1 \dots D_{k-1}$

Special Case 3: Non-differentiable activations



$$y = RELU(z)$$

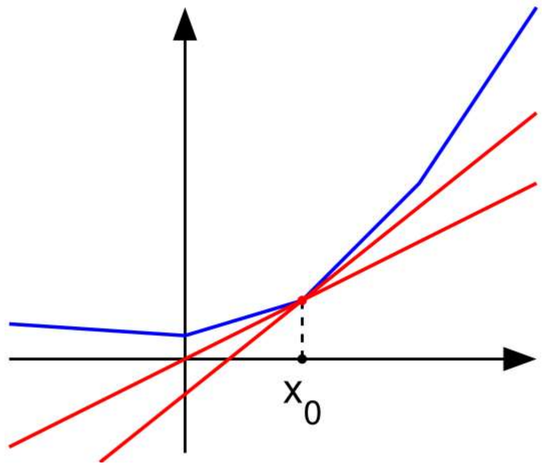




$$y = \max_j z_j$$

- 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

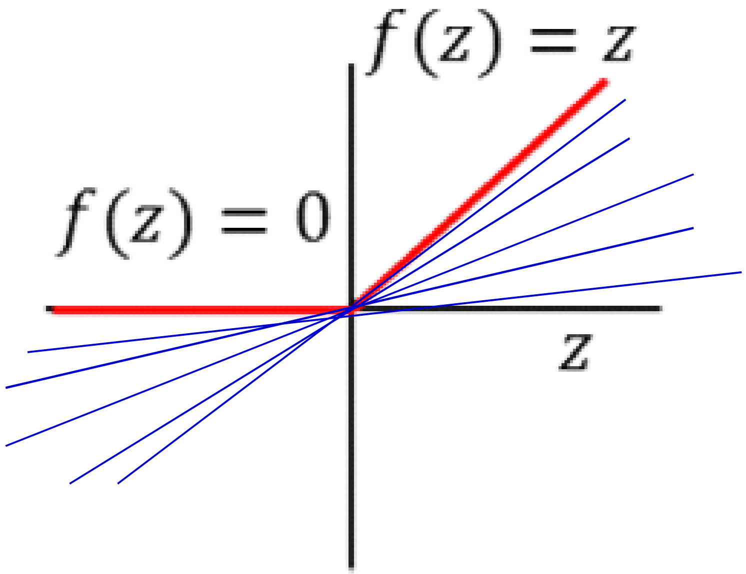


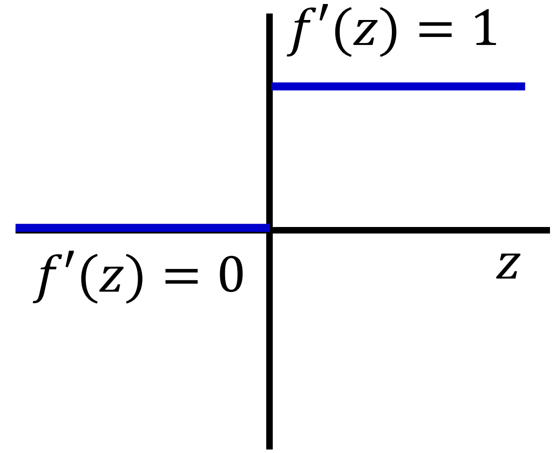
- A subgradient of a function $f(x)$ at a point x_0 is any vector v such that

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

- 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

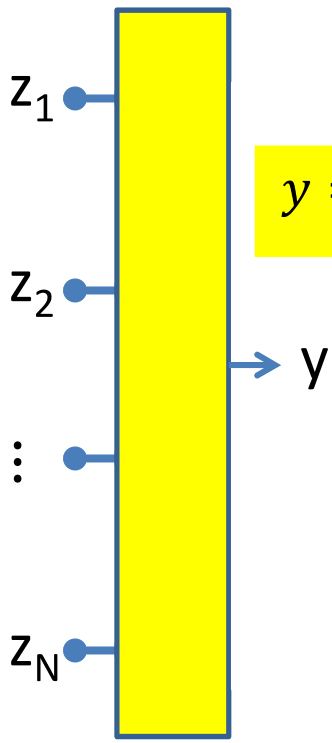




$$f'(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$$

- 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

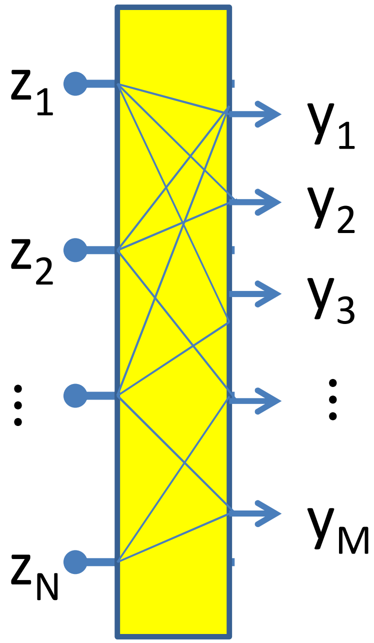


$$y = \max_j z_j$$

$$\frac{\partial y}{\partial z_i} = \begin{cases} 1, & i = \operatorname{argmax}_j z_j \\ 0, & otherwise \end{cases}$$

- 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



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

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

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

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

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

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

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

$$W \leftarrow W - \eta \nabla_W \mathbf{Err}$$

Training by BackProp

- Initialize all weights $(\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \dots, \boldsymbol{W}^{(K)})$
- 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 $\boldsymbol{Y_t}$
      - $Err \mathrel{+}= \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$
    - **Backward pass:** For all i, j, k:
      - Compute $\frac{d\boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{dw_{i,j}^{(k)}}$
      - Compute $\frac{dErr}{dw_{i,j}^{(k)}} \mathrel{+}= \frac{d\boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{dw_{i,j}^{(k)}}$
  - For all i, j, k, update:

$$w_{i,j}^{(k)} = w_{i,j}^{(k)} - \frac{\eta}{T} \frac{dErr}{dw_{i,j}^{(k)}}$$

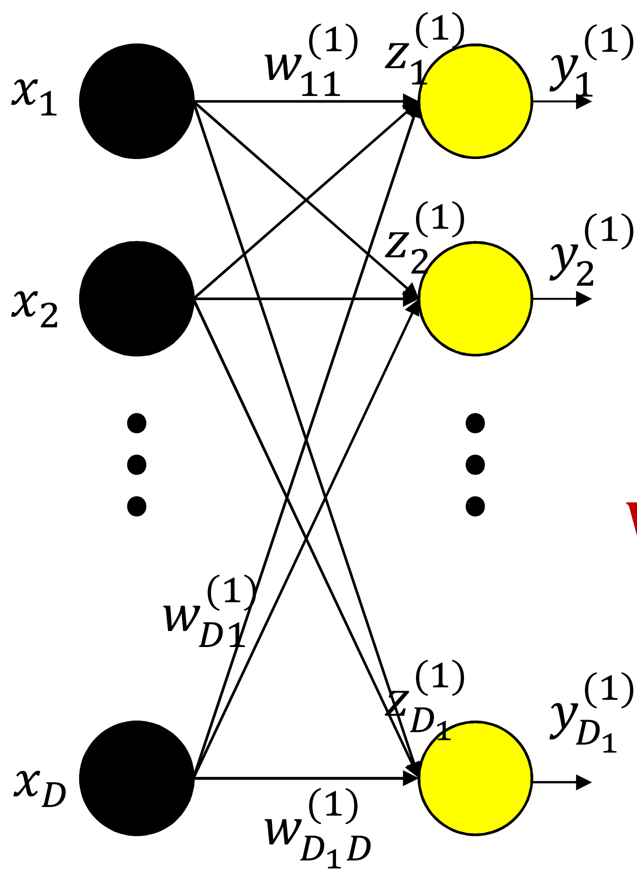
- Until Err has converged

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

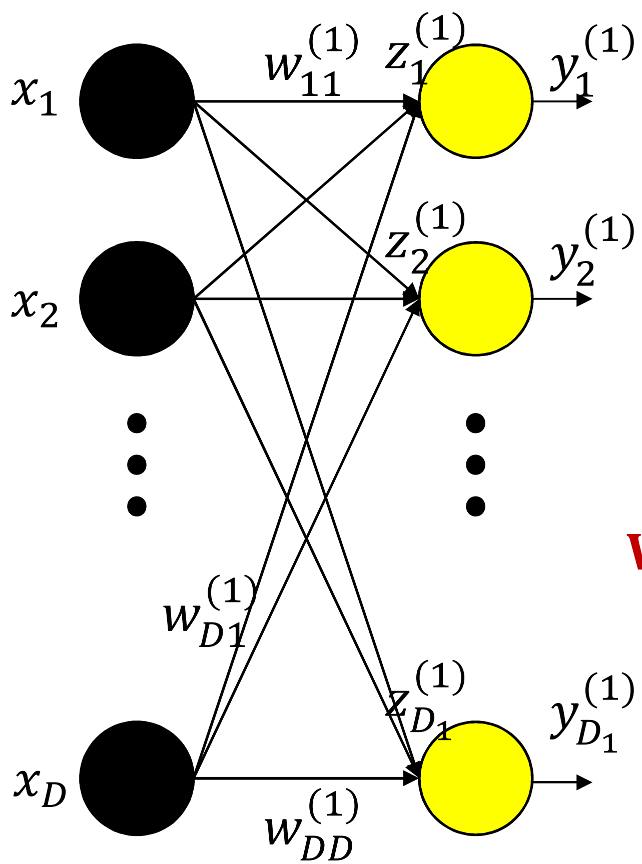


$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix} \qquad \mathbf{z}_k = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{D_k}^{(k)} \end{bmatrix} \qquad \mathbf{y}_k = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_{D_k}^{(k)} \end{bmatrix}$$

$$\mathbf{W}_k = \begin{bmatrix} w_{11}^{(k)} & w_{21}^{(k)} & \vdots & w_{D_k1}^{(k)} \\ w_{12}^{(k)} & w_{22}^{(k)} & \vdots & w_{D_k2}^{(k)} \\ \cdots & \cdots & \ddots & \vdots \\ w_{1D_{k+1}}^{(k)} & w_{1D_{k+1}}^{(k)} & \cdots & w_{1D_{k+1}}^{(k)} \end{bmatrix} \qquad \mathbf{b}_k = \begin{bmatrix} b_1^{(k)} \\ b_2^{(k)} \\ \vdots \\ b_{D_{k+1}}^{(k)} \end{bmatrix}$$

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

Vector formulation



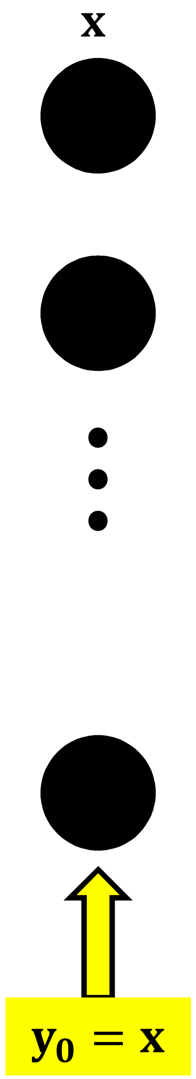
$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix} \qquad \mathbf{z_k} = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{D_k}^{(k)} \end{bmatrix} \qquad \mathbf{y_k} = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_{D_k}^{(k)} \end{bmatrix}$$

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

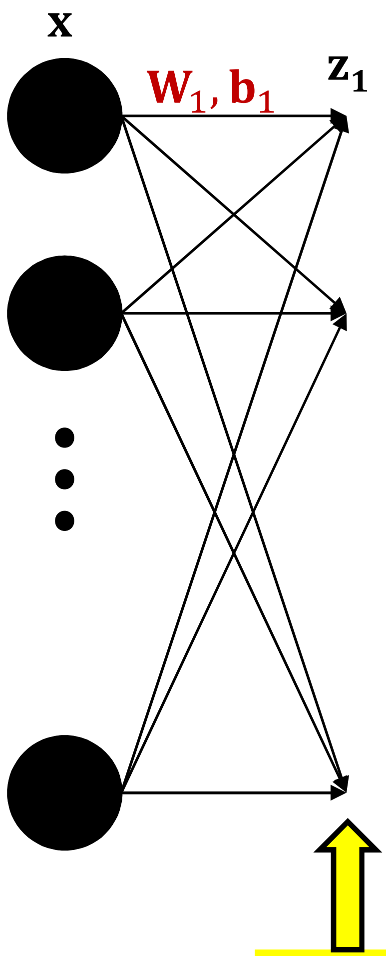
- The computation of a single layer is easily expressed in matrix notation as (setting $\mathbf{y_0} = \mathbf{x}$):

$$\boxed{\mathbf{z_k} = \mathbf{W}_k \mathbf{y_{k-1}} + \mathbf{b_k}} \qquad \boxed{\mathbf{y_k} = \boldsymbol{f}_k(\mathbf{z_k})}$$

The forward pass: Evaluating the network

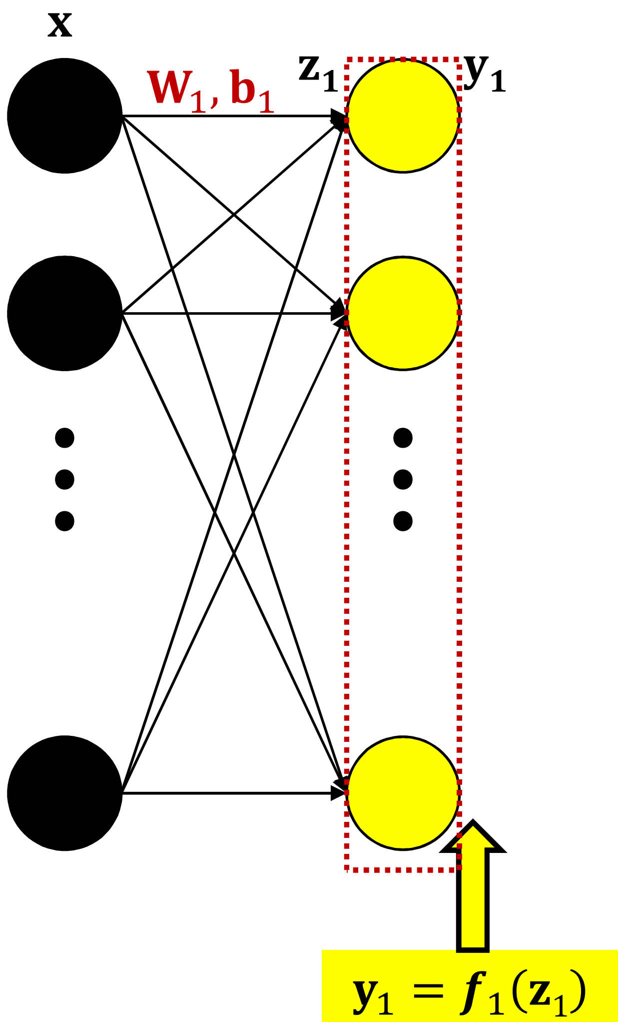


The forward pass



$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$

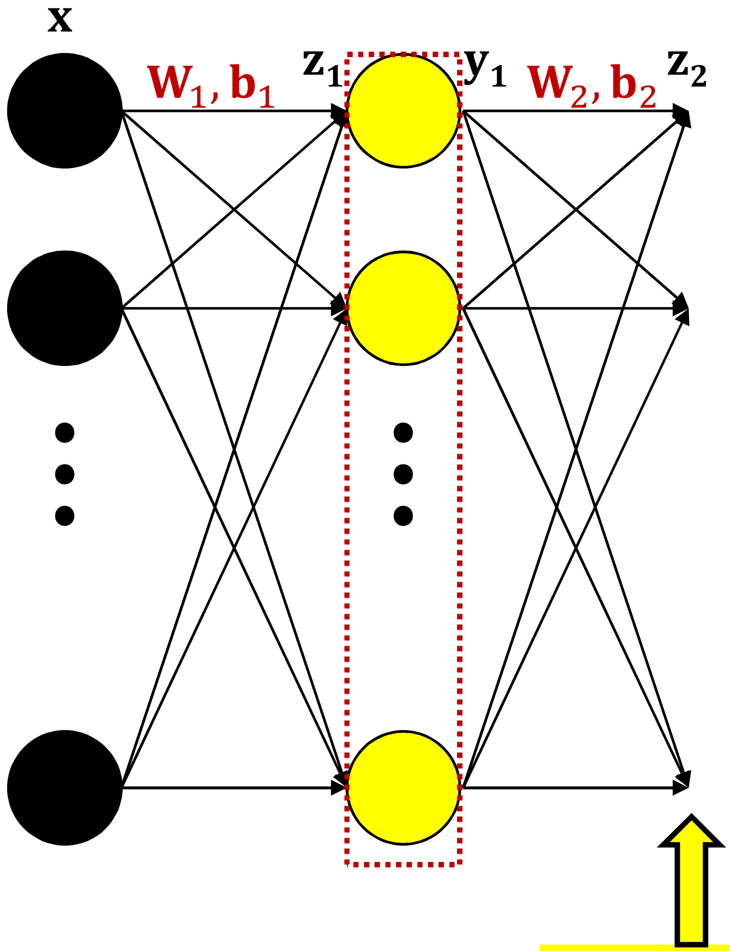
The forward pass



The Complete computation

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

The forward pass

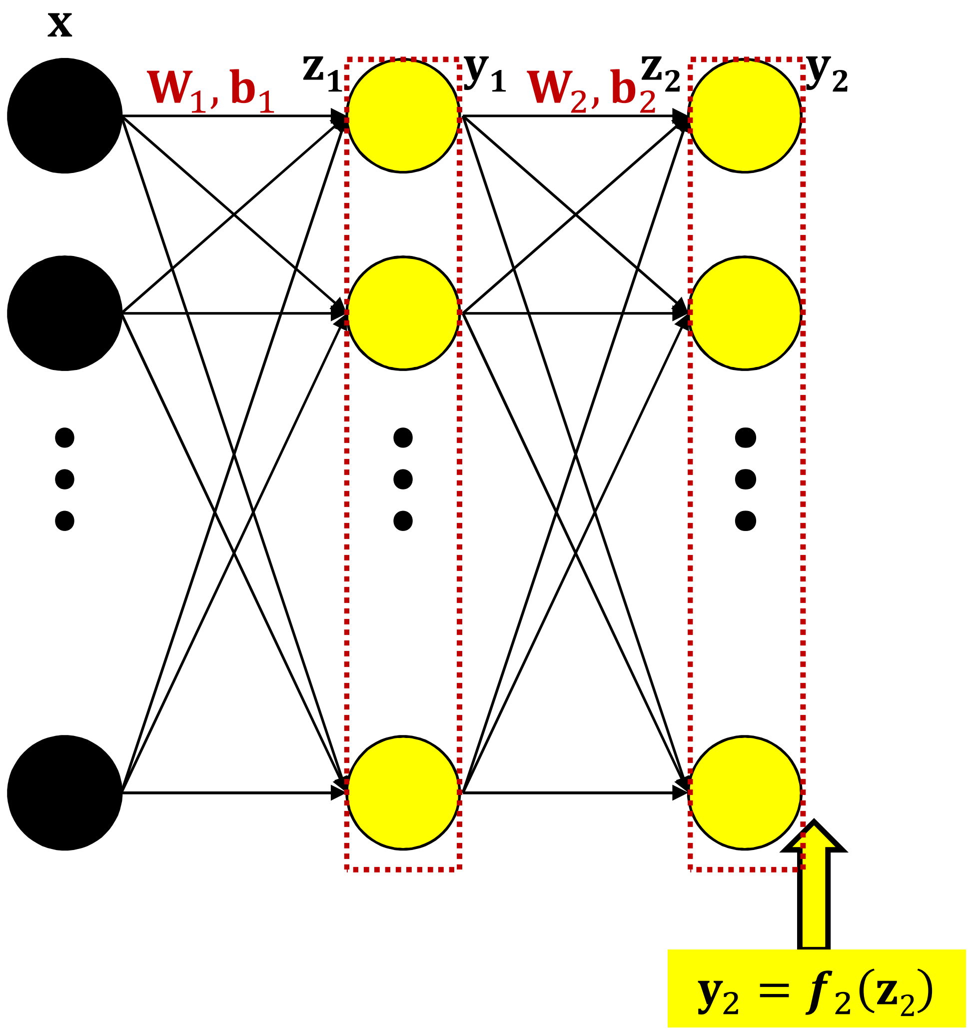


$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{y}_1 + \mathbf{b}_2$$

The Complete computation

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

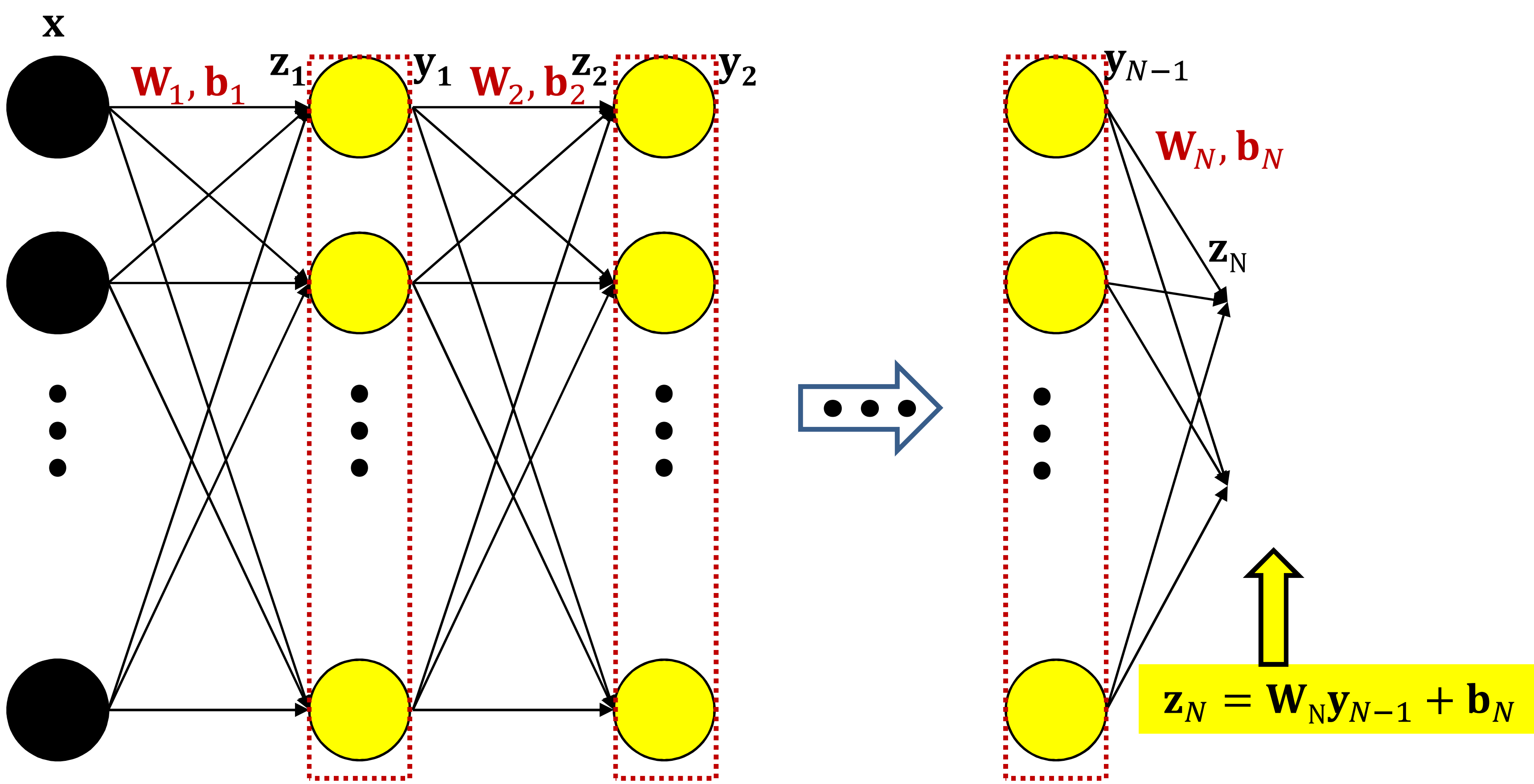
The forward pass



The Complete computation

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

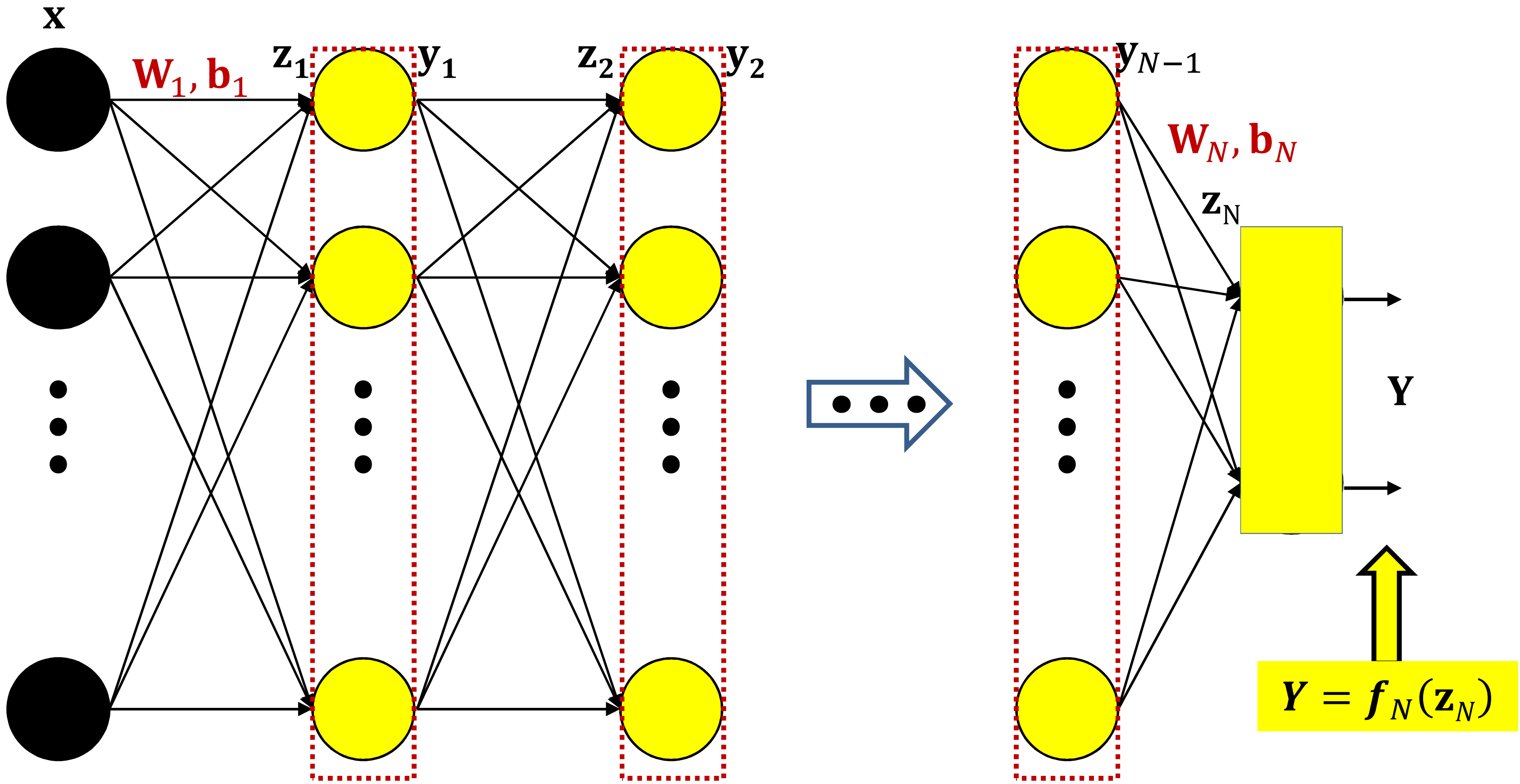
The forward pass



The Complete computation

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

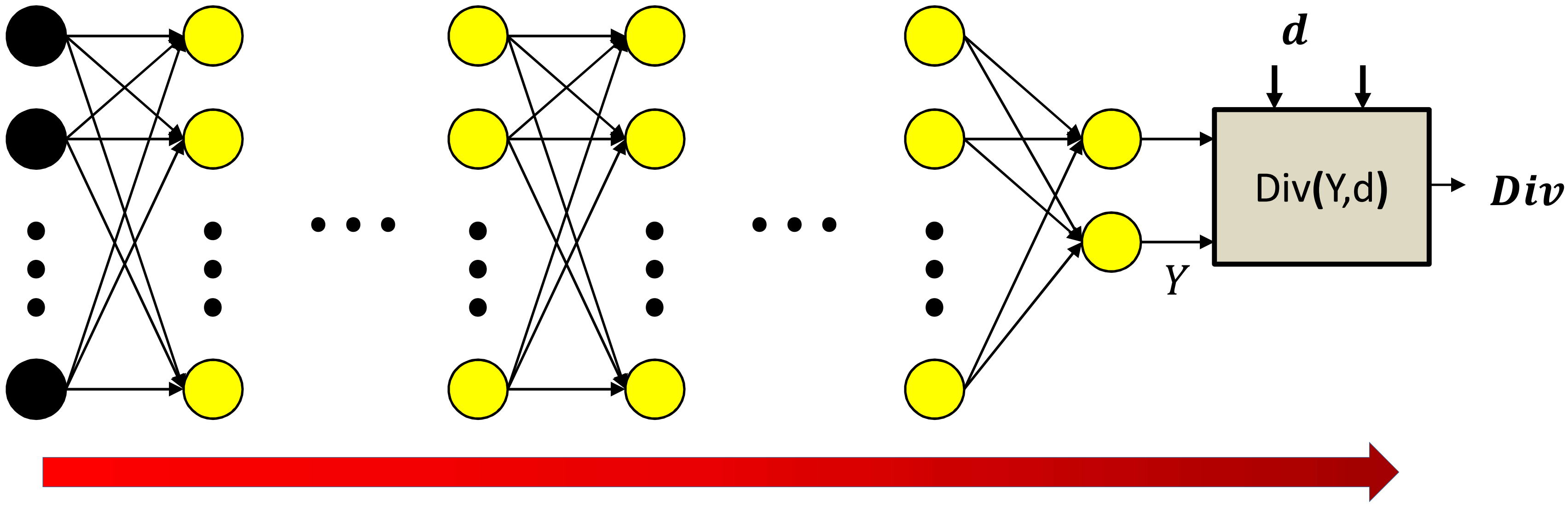
The forward pass



The Complete computation

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

Forward pass



Forward pass:

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

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

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

The Forward Pass

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

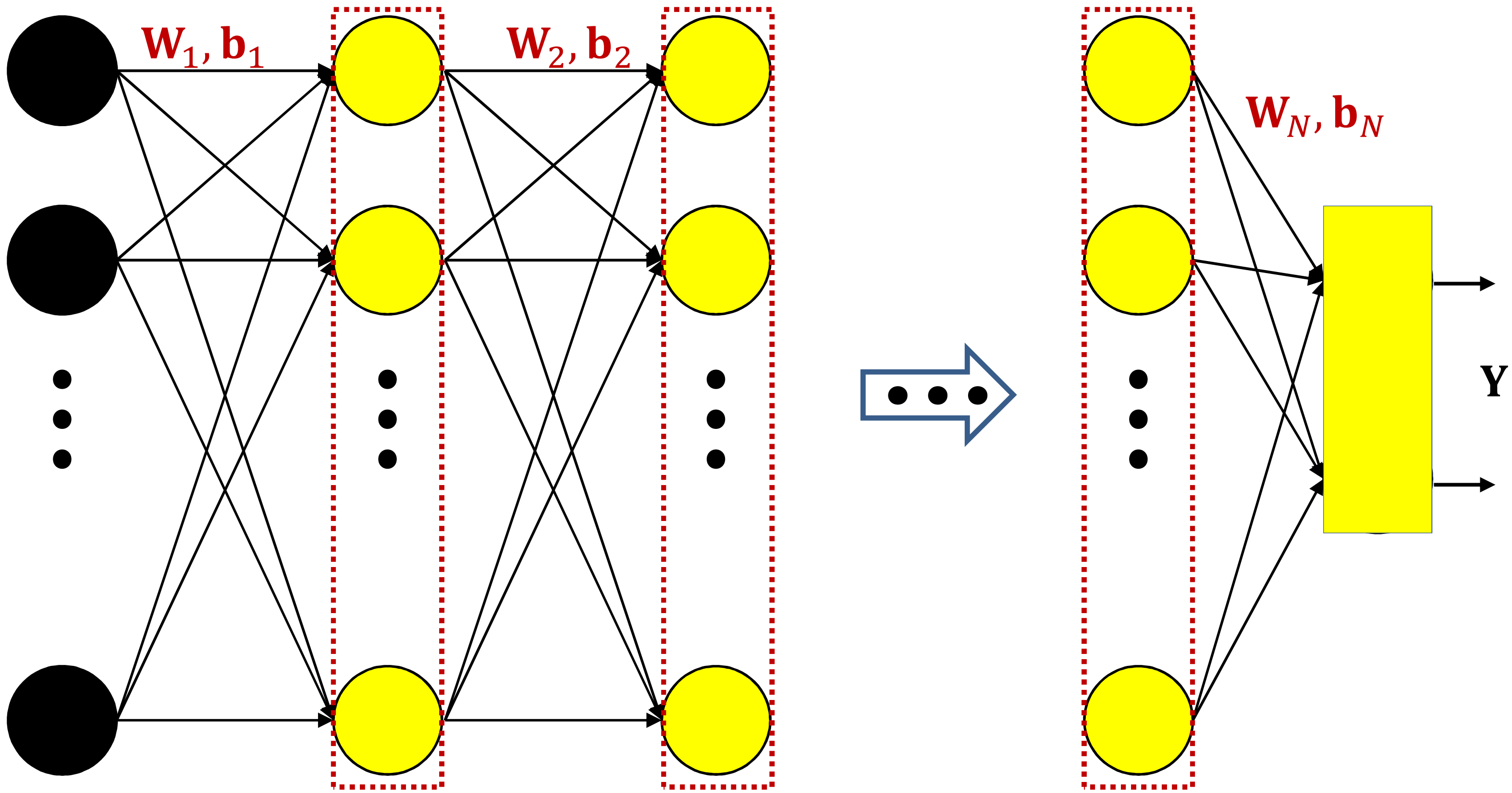
- For layer k = 1 to N:
  - Recursion:

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

- 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

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

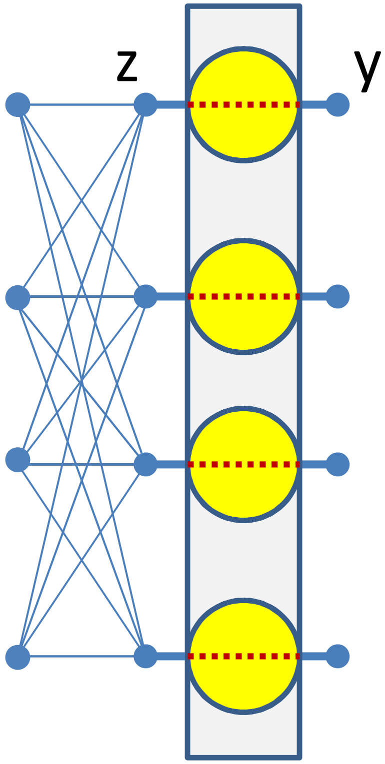
Using vector notation

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

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

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

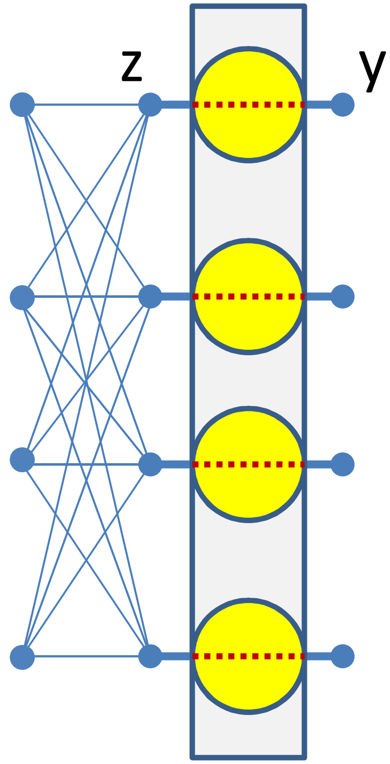
Jacobians can describe the derivatives of neural activations w.r.t their input



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

- **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

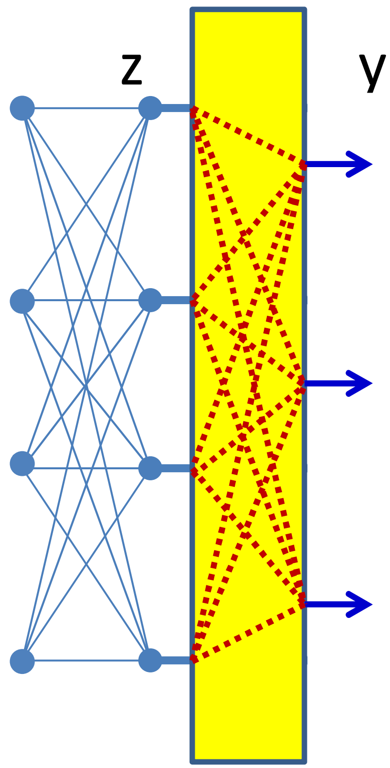


$$\boxed{y_i = f(z_i)}$$

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

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

For *Vector* activations



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

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

Special case: Affine functions

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

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

- 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:**

$$\mathbf{y} = \boldsymbol{f}(\boldsymbol{g}(\mathbf{x}))$$

$$\mathbf{z} = \boldsymbol{g}(\mathbf{x})$$

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

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

Check

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

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

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

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

Vector derivatives: Chain rule

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

$$D = f\left(\boldsymbol{g}(\mathbf{x})\right)$$

$$\mathbf{z} = \boldsymbol{g}(\mathbf{x})$$

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

$$\nabla_{\mathbf{x}} D = \nabla_{\mathbf{z}}(D) J_{\mathbf{z}}(\mathbf{x})$$

Check

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

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

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

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

Special Case

- Scalar functions of Affine functions

$$\boxed{D = f(\mathbf{W}\mathbf{y} + \mathbf{b})}$$

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

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

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

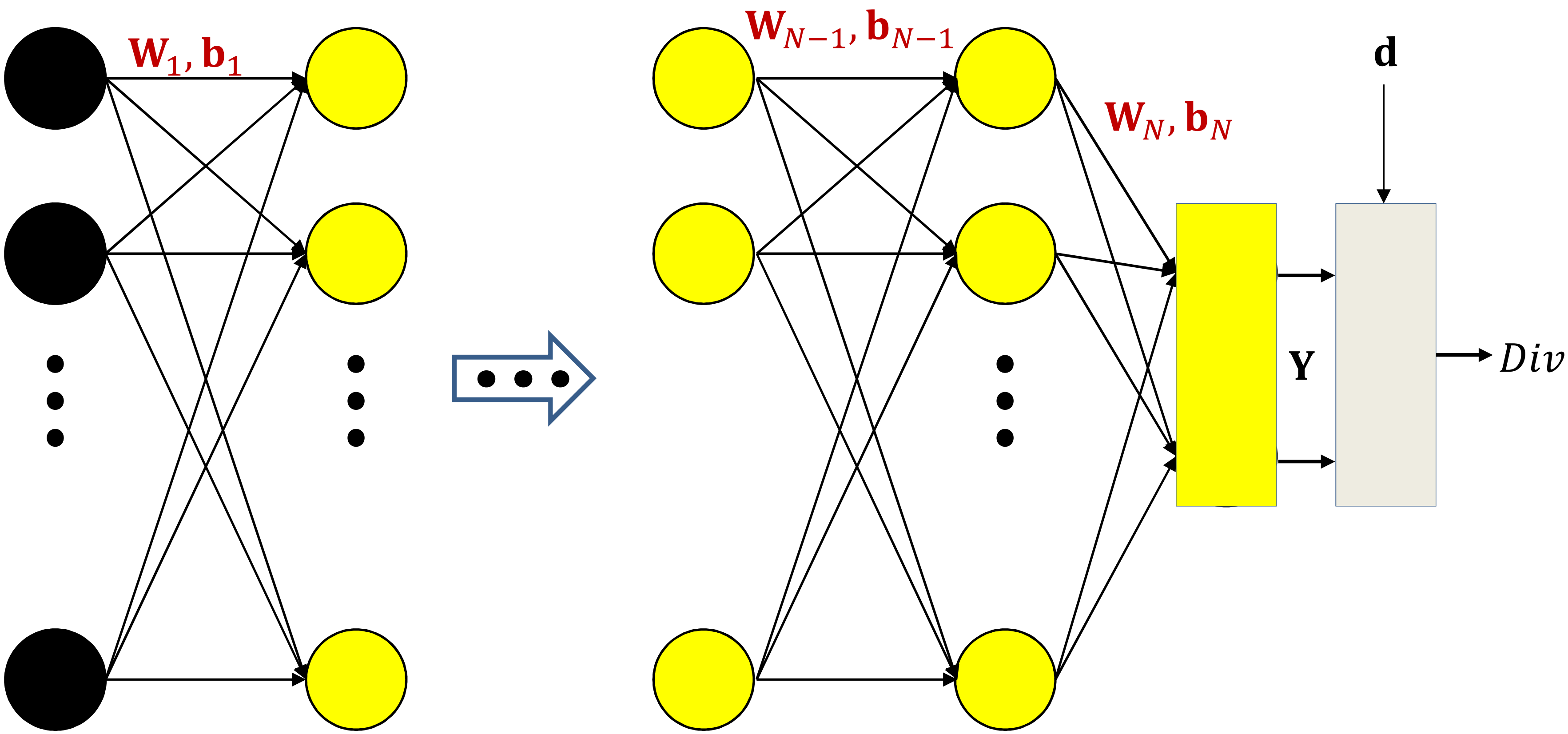
$$\boxed{\nabla_{\mathbf{b}} D = \nabla_{\mathbf{z}}(D)}$$

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

Derivatives w.r.t parameters

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

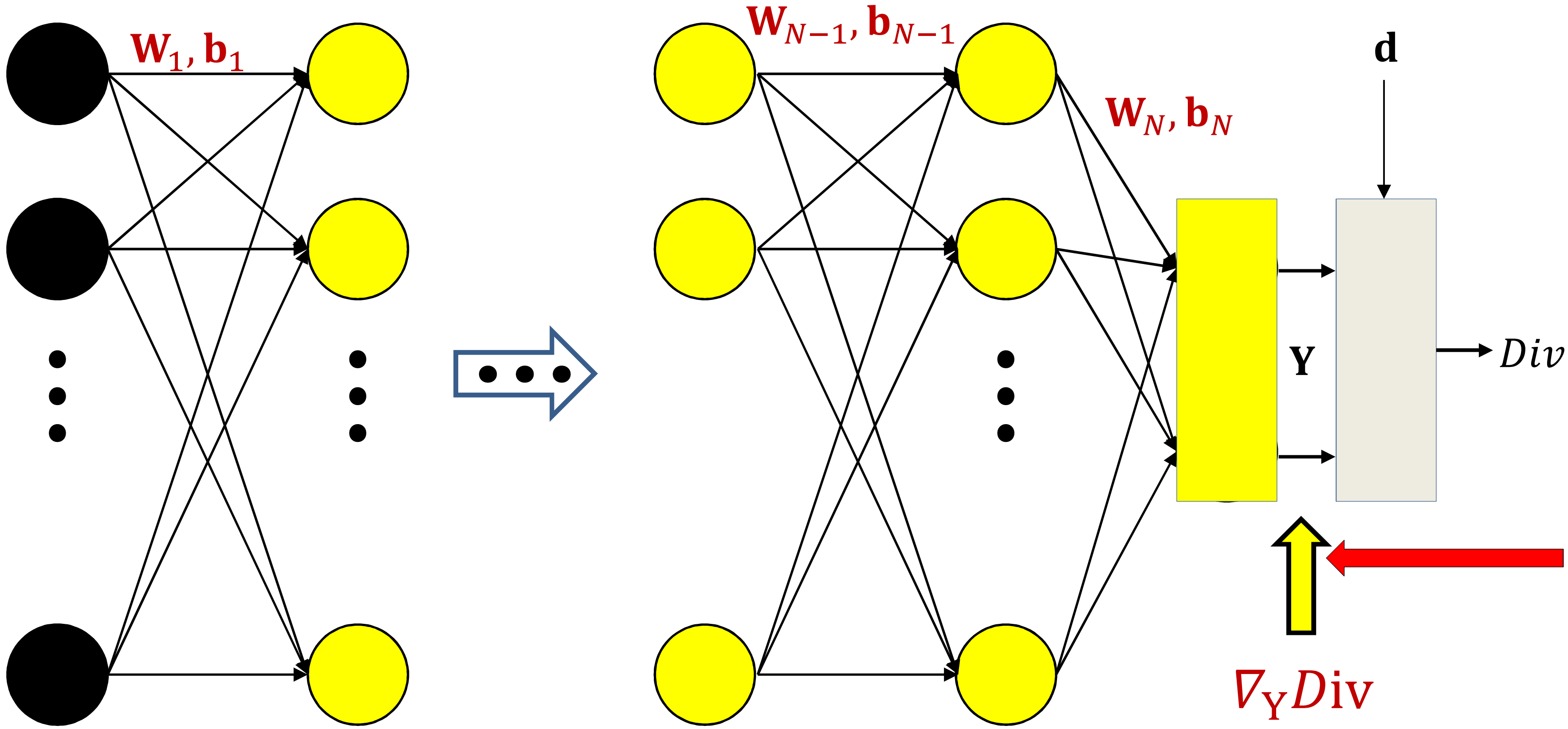
The backward pass



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

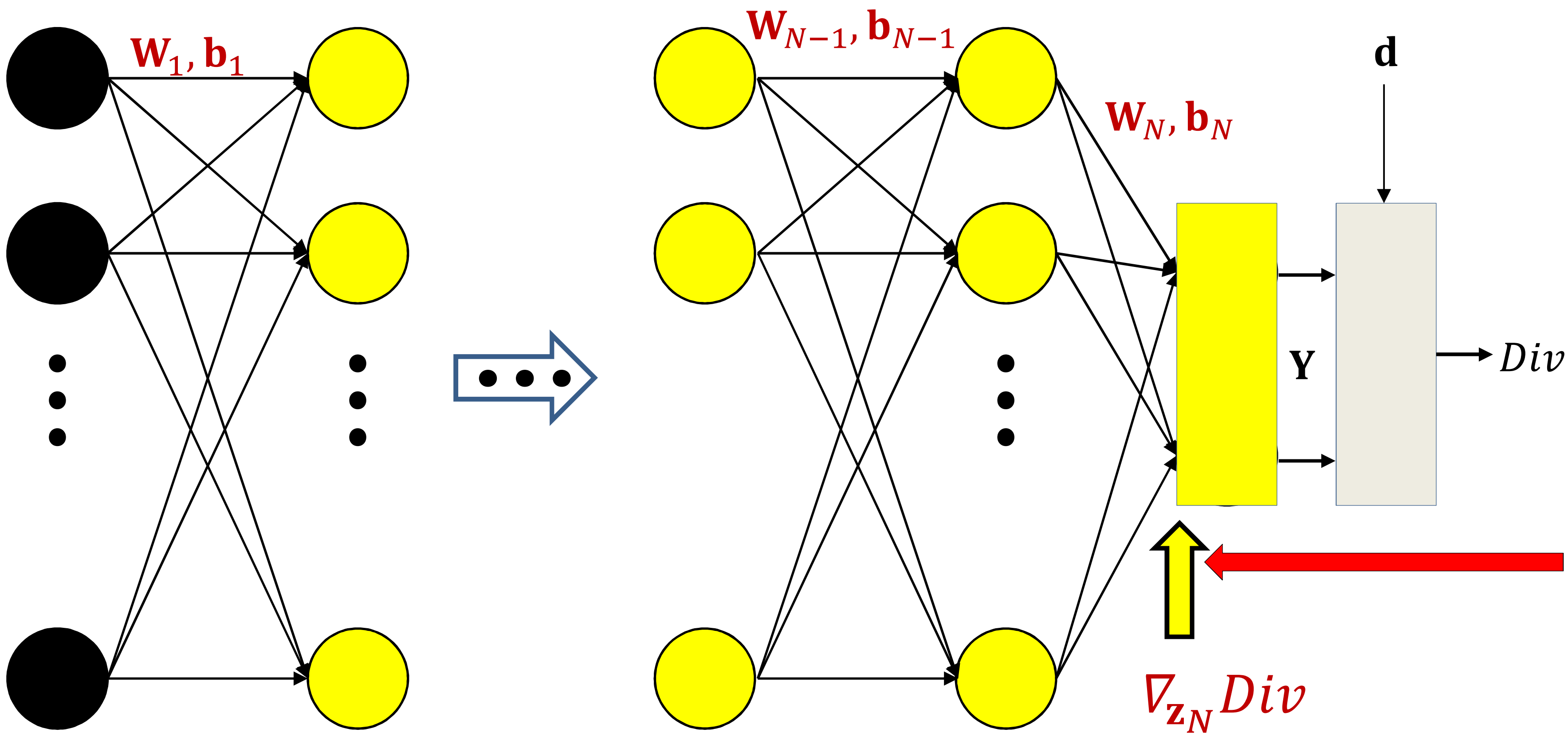
In general $\nabla_{\mathbf{a}}\mathbf{b}$ represents a derivative of $\mathbf{b}$ w.r.t. $\mathbf{a}$ and could be a gradient (for scalar $\mathbf{b}$) Or a Jacobian (for vector $\mathbf{b}$)

The backward pass



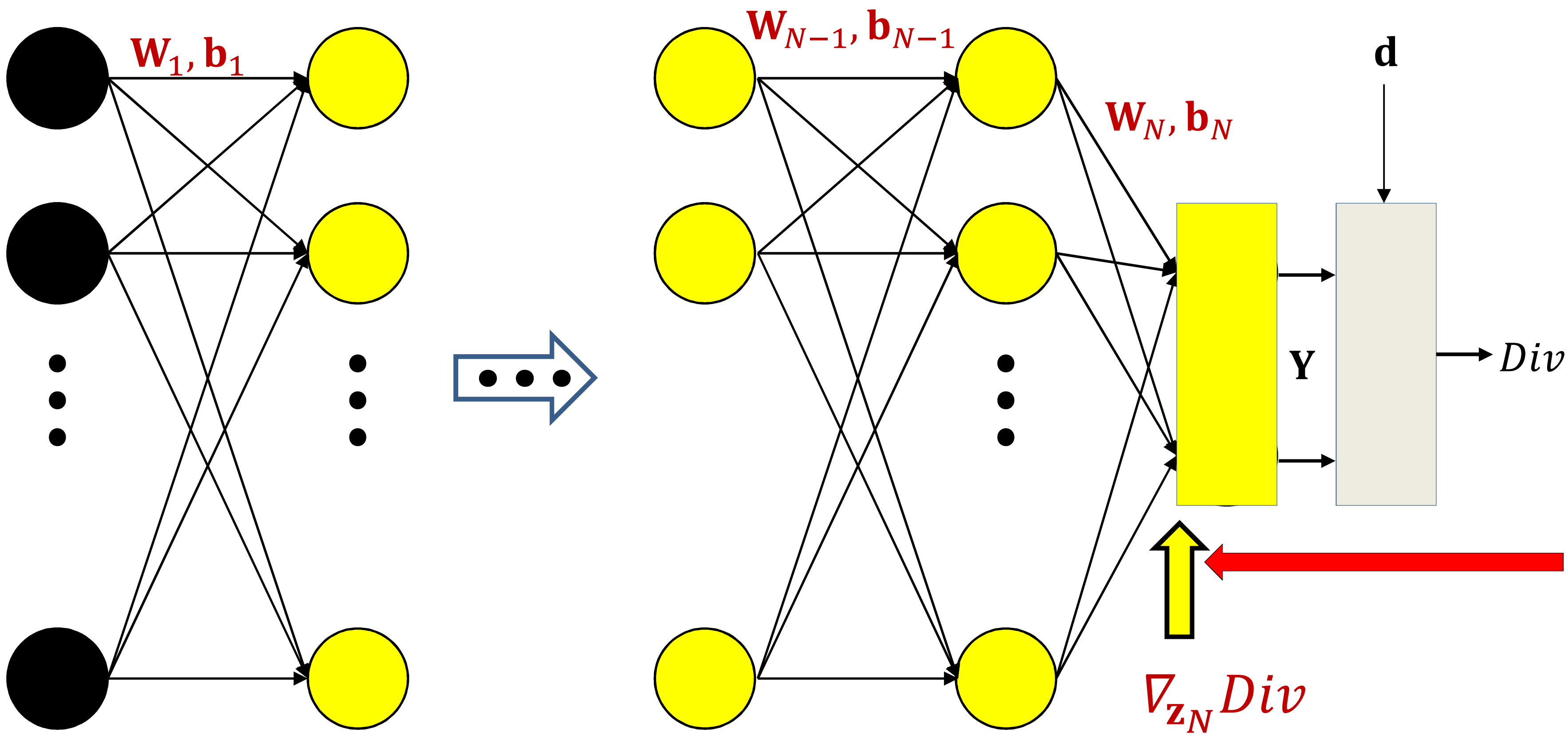
First compute the gradient of the divergence w.r.t. Y.
The actual gradient depends on the divergence function.

The backward pass



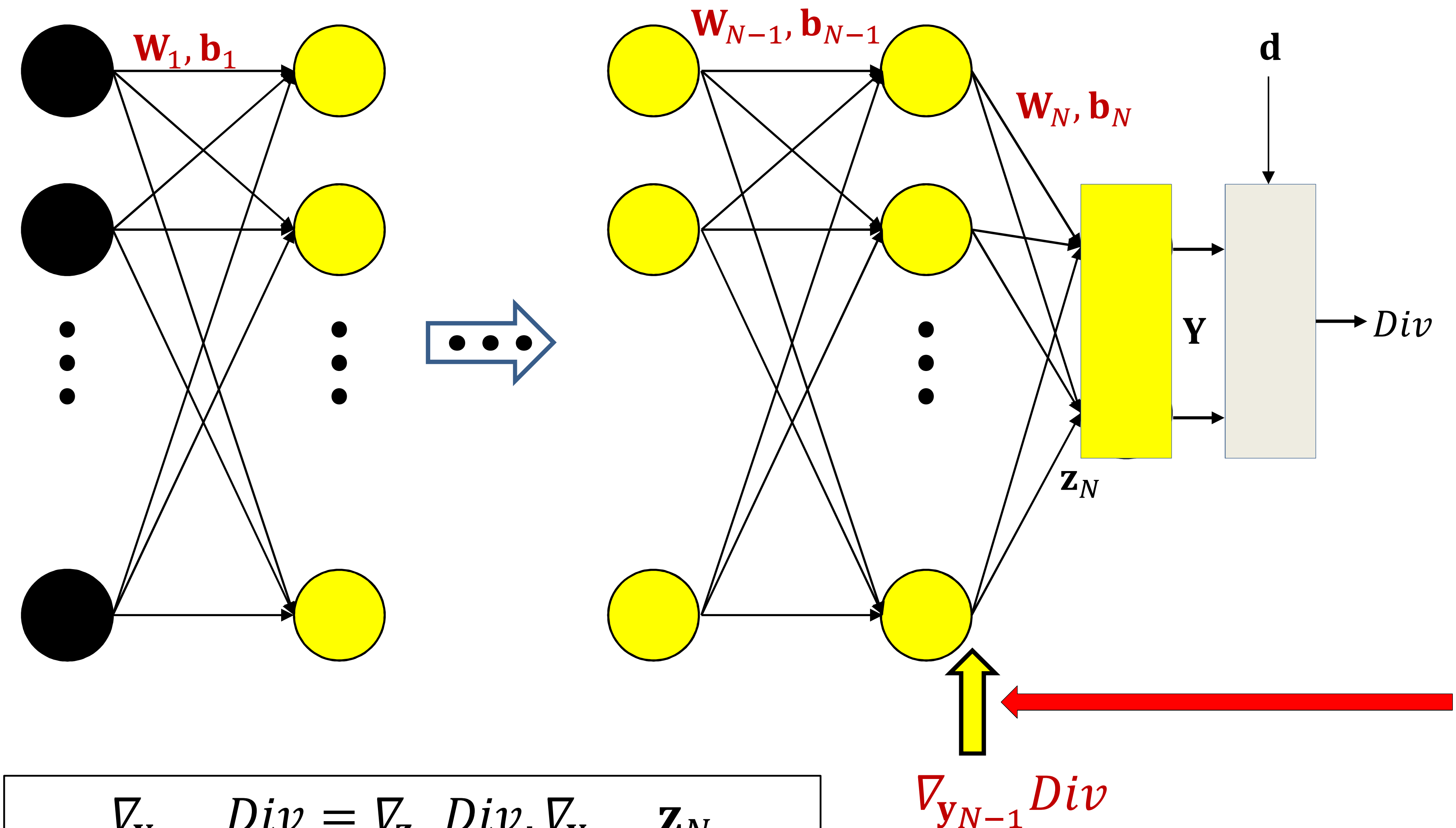
$$\boxed{\nabla_{\mathbf{z}_N} Div = \nabla_{\mathrm{Y}} Div . \nabla_{\mathbf{z}_N} \mathbf{Y}}$$

The backward pass



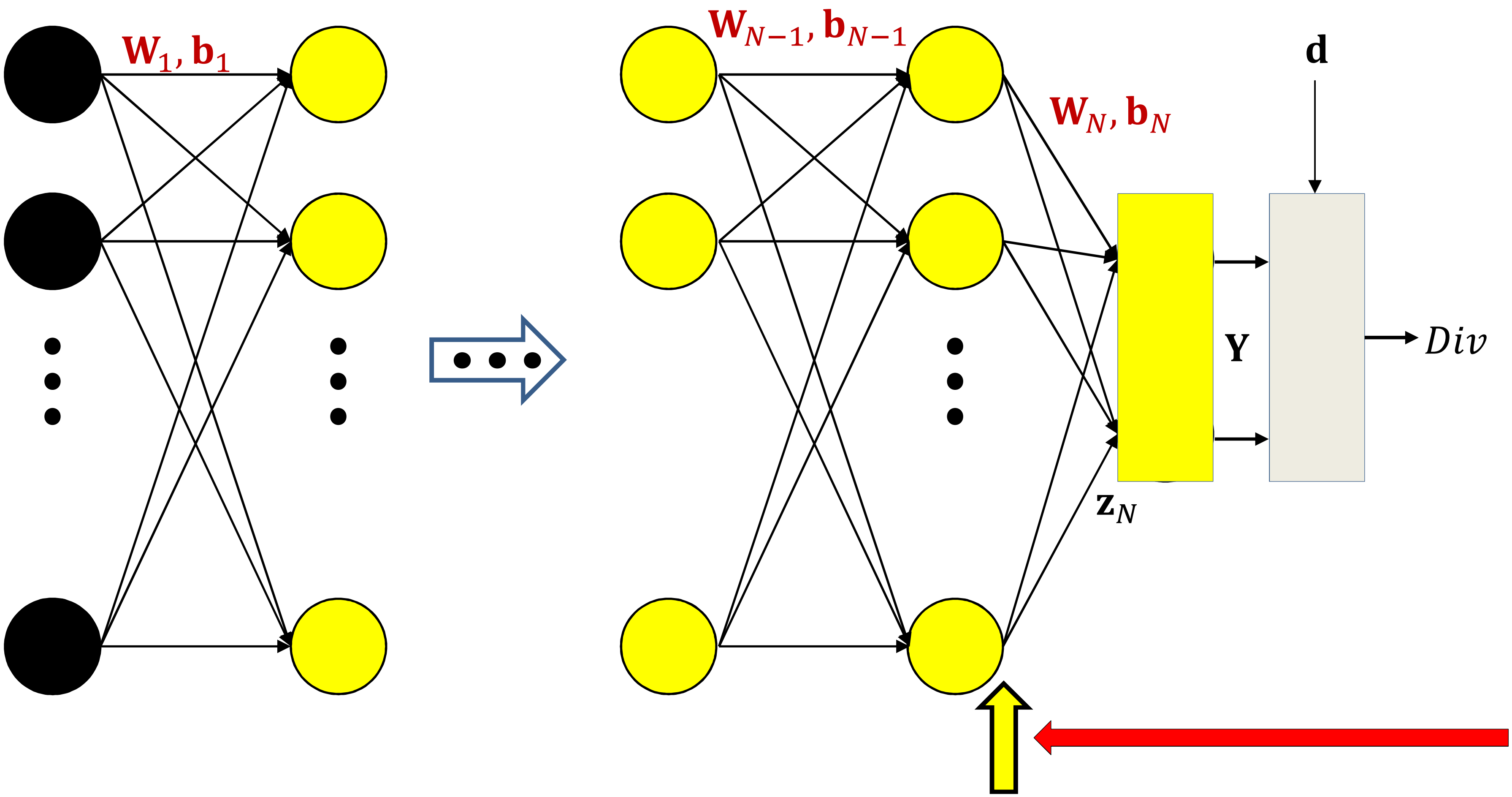
$$\nabla_{\mathbf{z}_N} Div = \nabla_{\mathbf{Y}} Div \, J_{\mathbf{Y}}(\mathbf{z}_N)$$

The backward pass



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

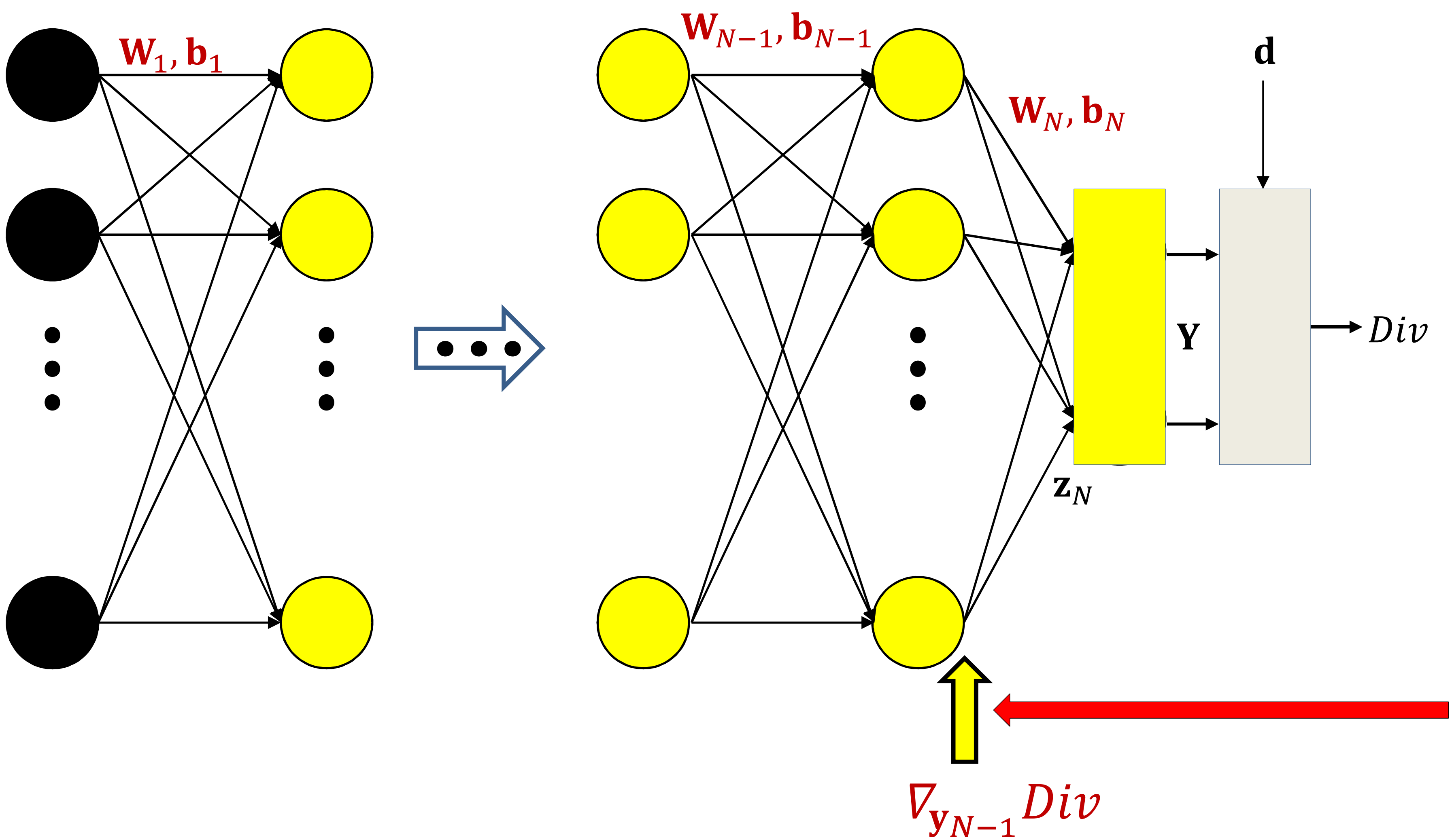
The backward pass



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

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

The backward pass

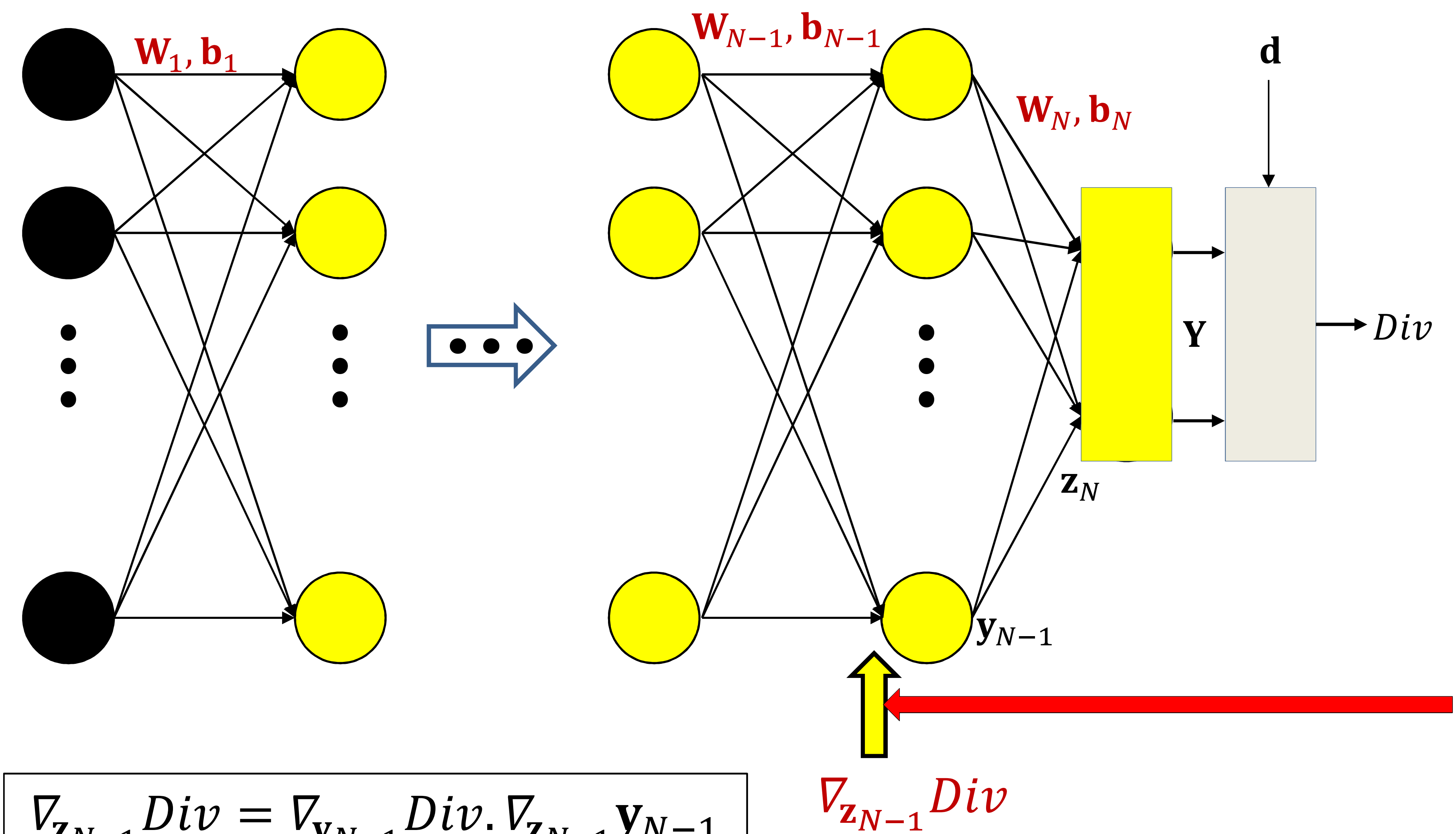


$$\nabla_{\mathbf{y}_{N-1}} Div = \nabla_{\mathbf{z}_N} Div\, \mathbf{W}_N$$

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

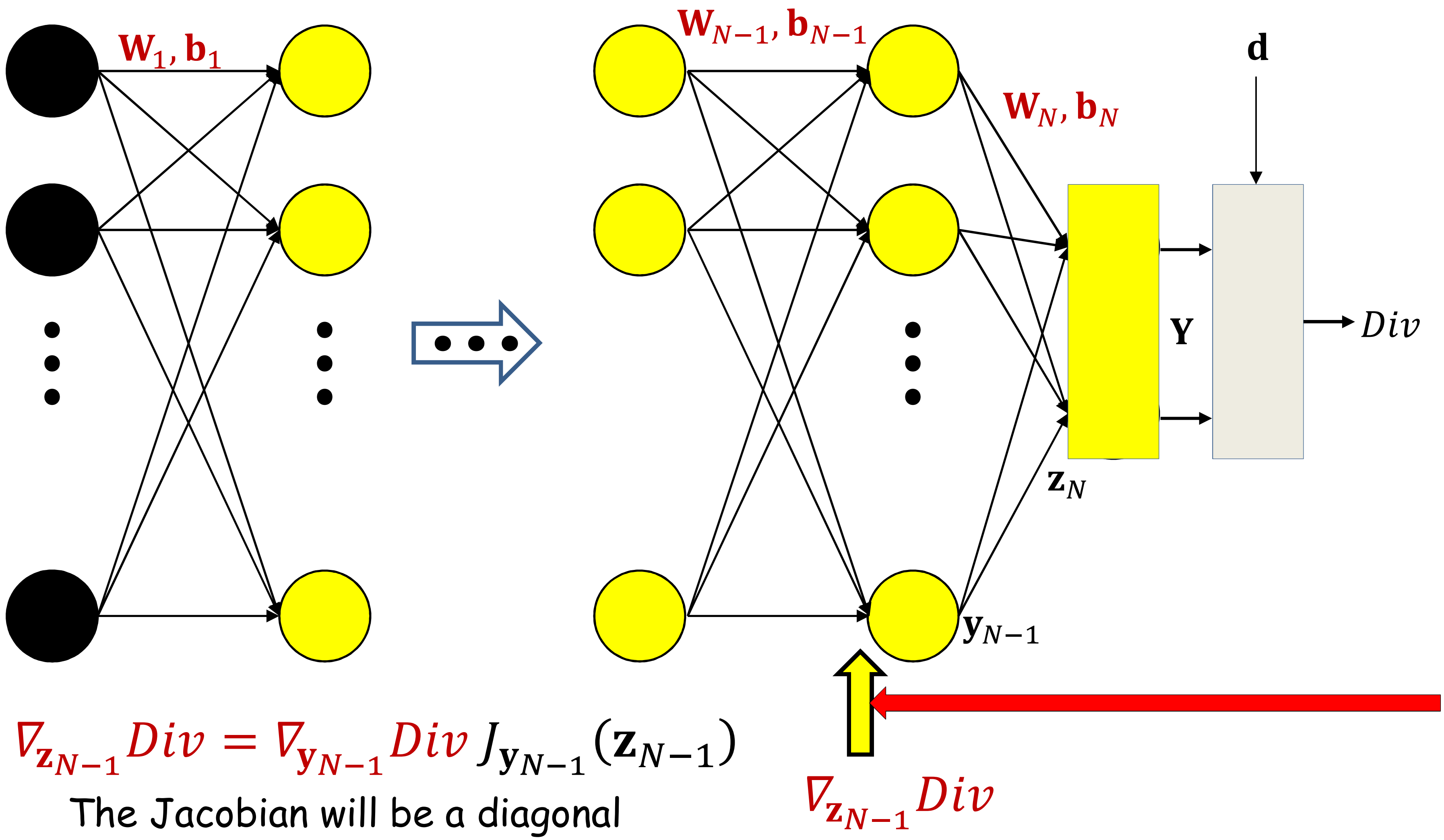
$$\boxed{\nabla_{\mathbf{b}_N} Div = \nabla_{\mathbf{z}_N} Div}$$

The backward pass



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

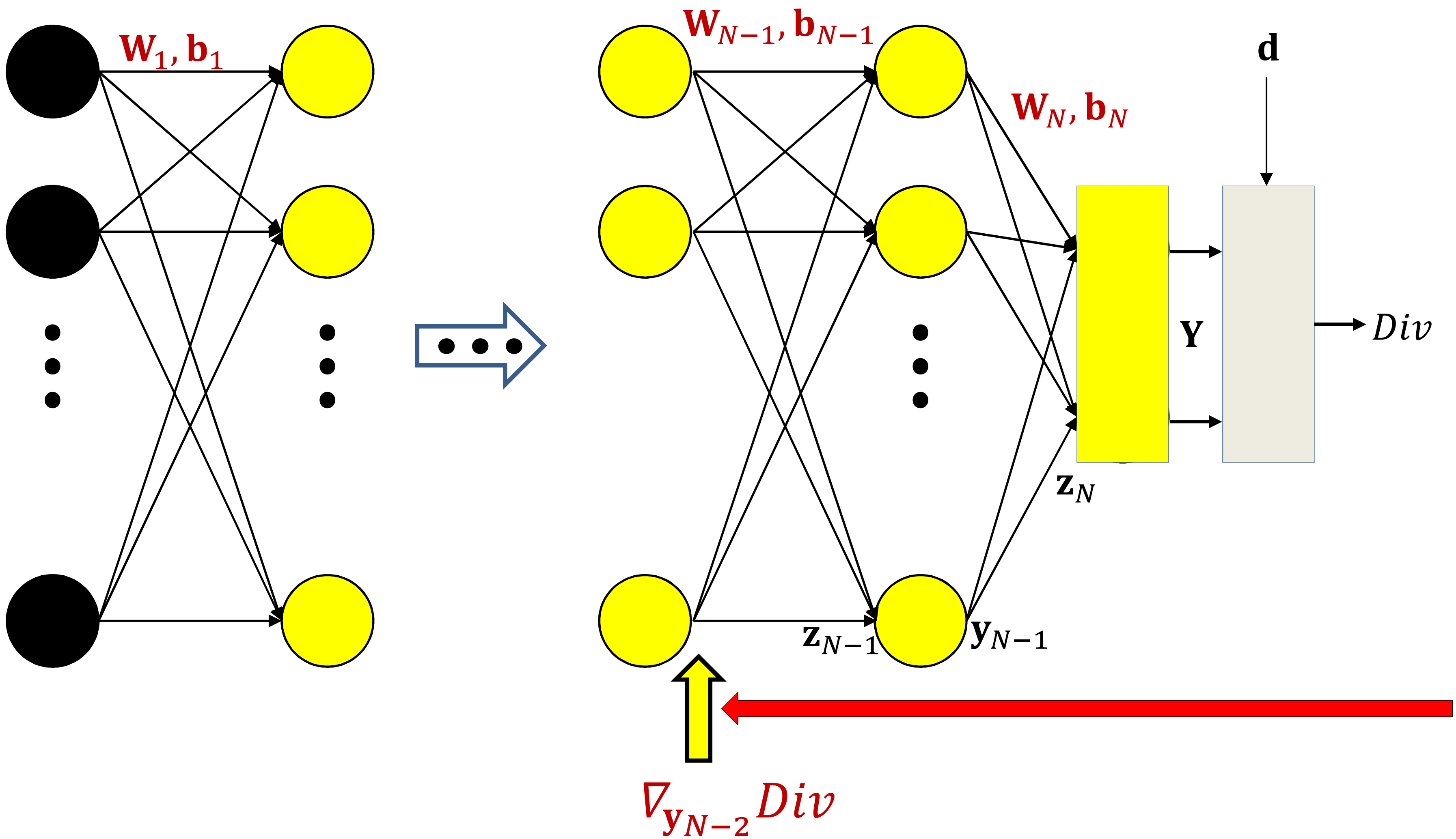
The backward pass



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

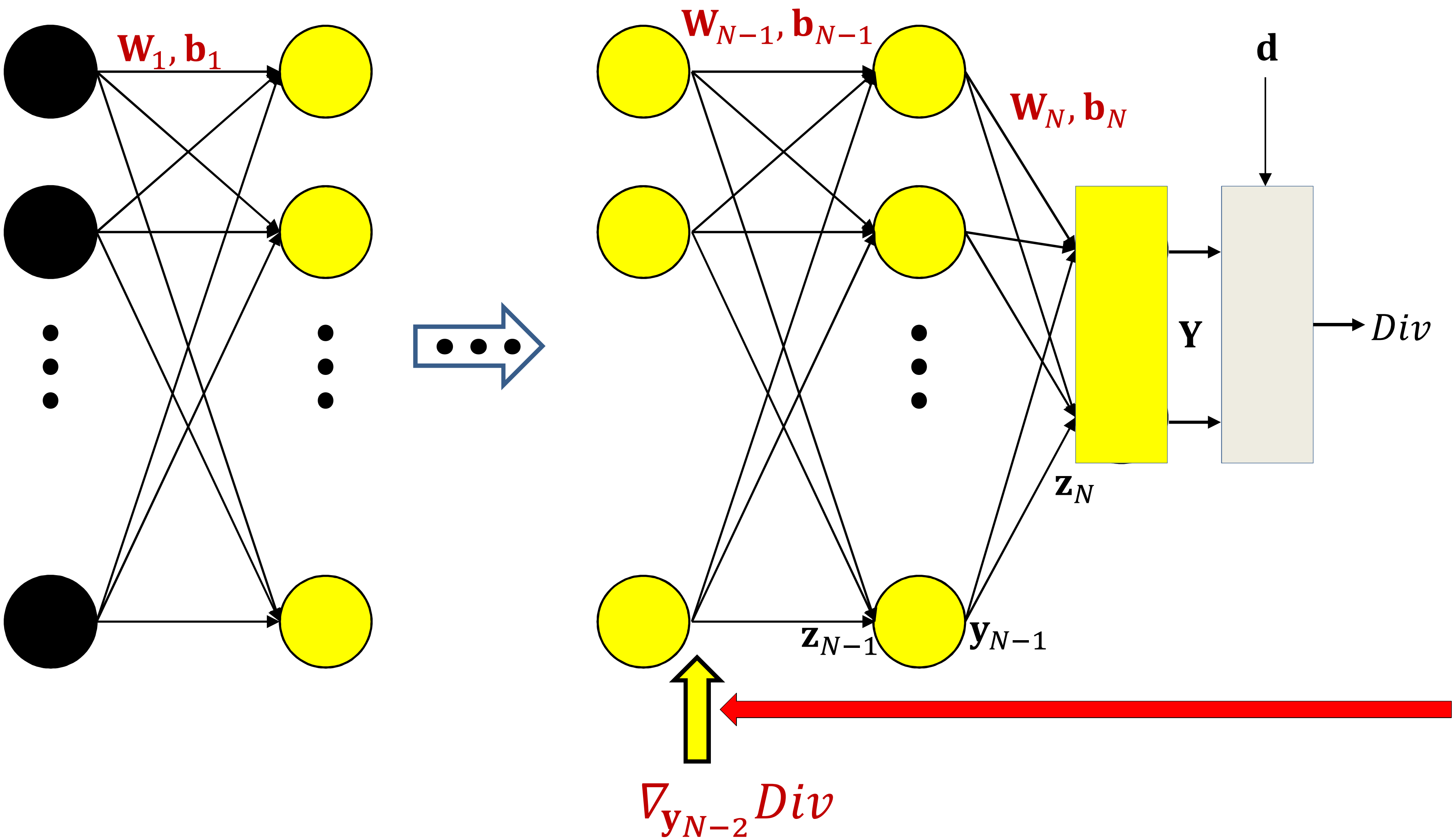
The Jacobian will be a diagonal matrix for scalar activations

The backward pass



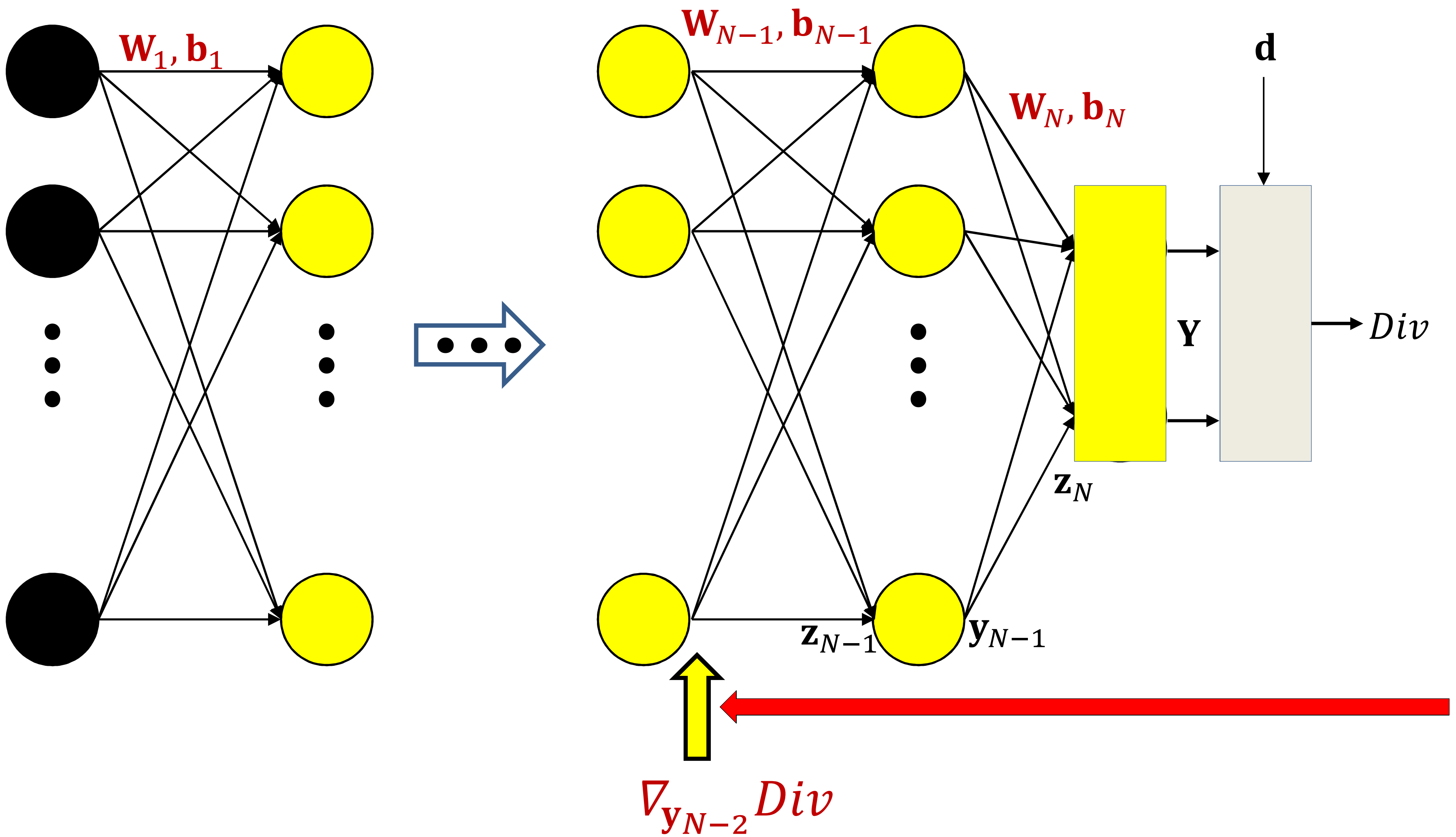
$$\nabla_{\mathbf{y}_{N-2}} Div = \nabla_{\mathbf{z}_{N-1}} Div . \nabla_{\mathbf{y}_{N-2}} \mathbf{z}_{N-1}$$

The backward pass



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

The backward pass

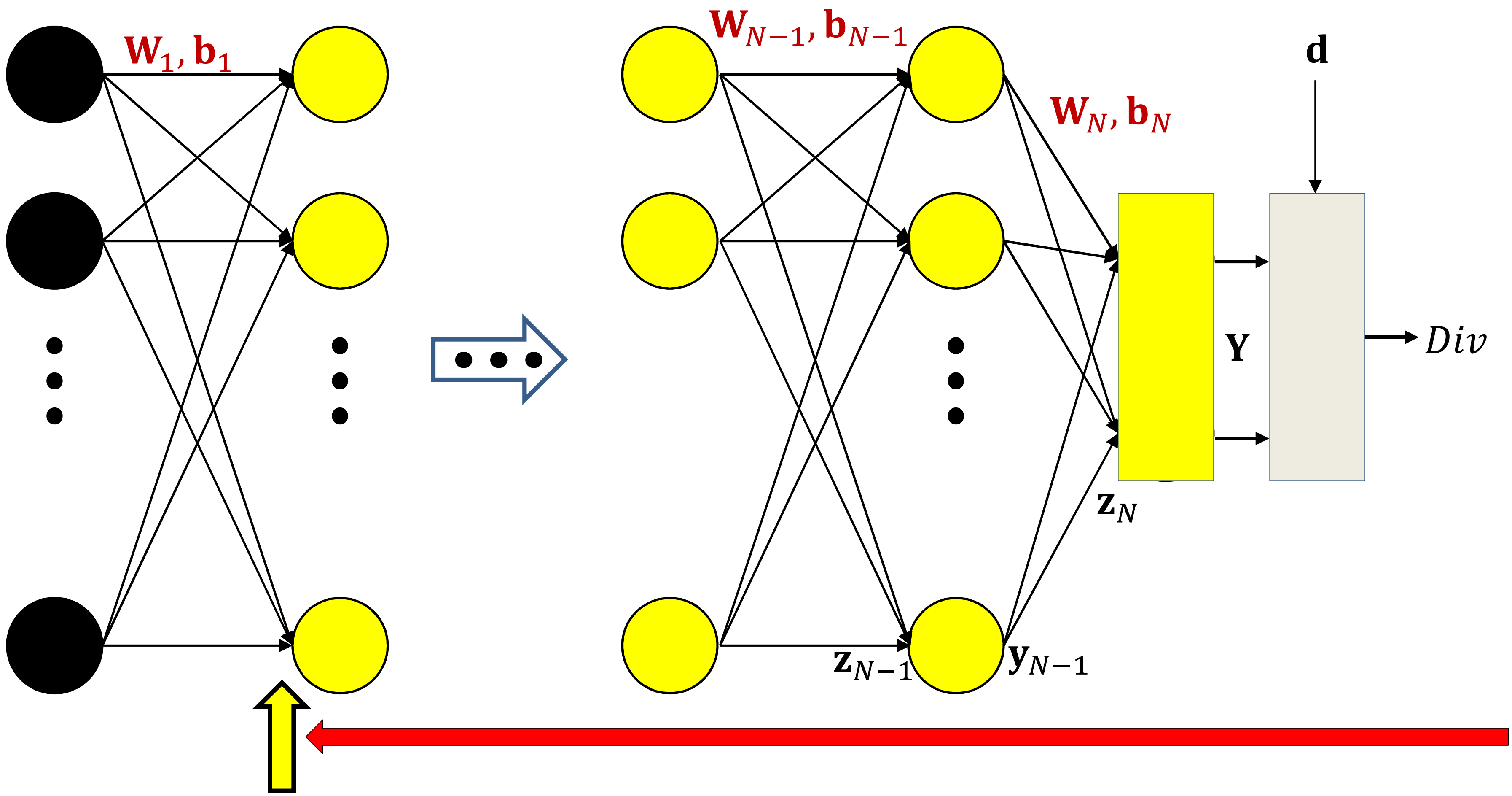


$$\nabla_{\mathbf{y}_{N-2}} Div = \nabla_{\mathbf{z}_{N-1}} Div \; \mathbf{W}_{N-1}$$

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

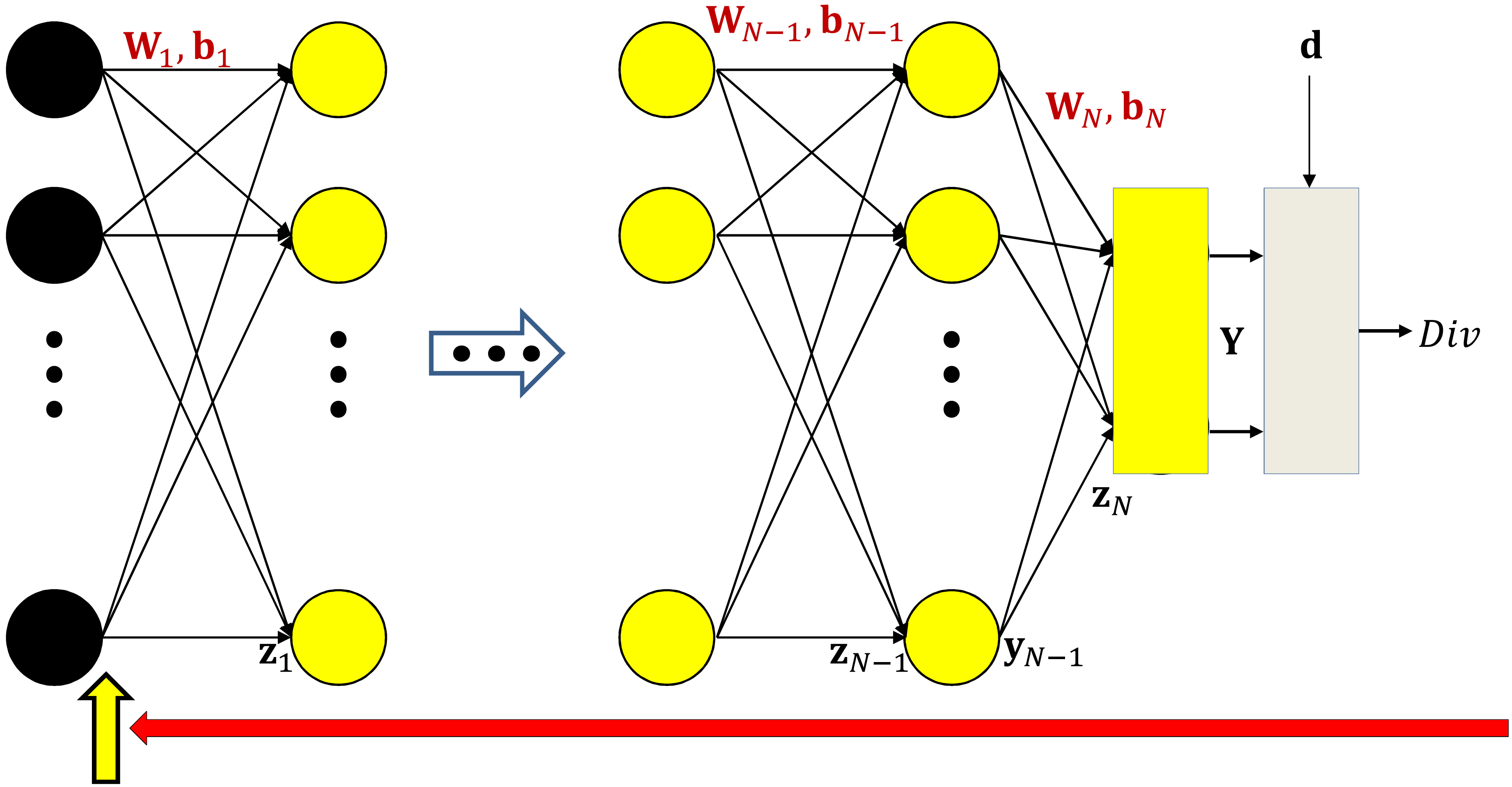
$$\boxed{\nabla_{\mathbf{b}_{N-1}} Div = \nabla_{\mathbf{z}_{N-1}} Div}$$

The backward pass



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

The backward pass



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

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

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

The Backward Pass

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

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

- Output:

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

Neural network training algorithm

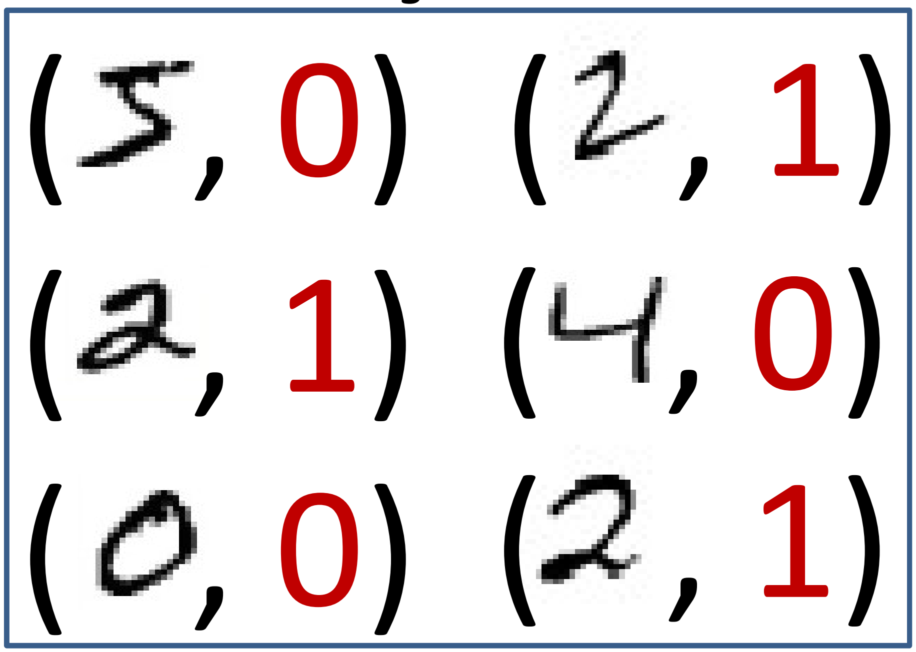
- Initialize all weights and biases $(\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \dots, \mathbf{W}_N, \mathbf{b}_N)$
- Do:
  - $Err = 0$
  - For all k, initialize $\nabla_{\mathbf{W}_k} Err = 0$, $\nabla_{\mathbf{b}_k} Err = 0$
  - For all $t = 1:T$
    - Forward pass : Compute
      - Output $\boldsymbol{Y}(\boldsymbol{X}_t)$
      - Divergence $\boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$
      - $Err += \boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$
    - Backward pass: For all k compute:
      - $\nabla_{\mathbf{W}_k} \boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$; $\nabla_{\mathbf{b}_k} \boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$
      - $\nabla_{\mathbf{W}_k} Err += \nabla_{\mathbf{W}_k} \boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$; $\nabla_{\mathbf{b}_k} Err += \nabla_{\mathbf{b}_k} \boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$
  - For all k, update:

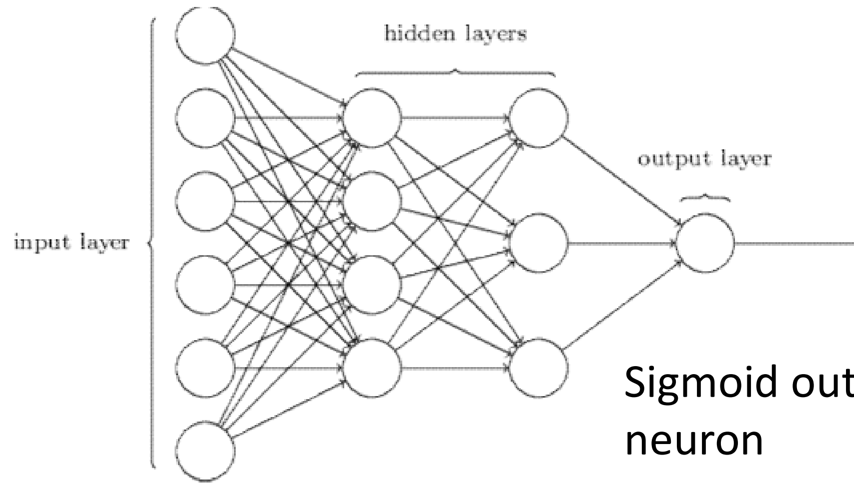
$$\mathbf{W}_k = \mathbf{W}_k - \frac{\eta}{T}\left(\nabla_{\mathbf{W}_k} Err\right)^T; \qquad \mathbf{b}_k = \mathbf{b}_k - \frac{\eta}{T}\left(\nabla_{\mathbf{W}_k} Err\right)^T$$

- Until Err has converged

Setting up for digit recognition

Training data



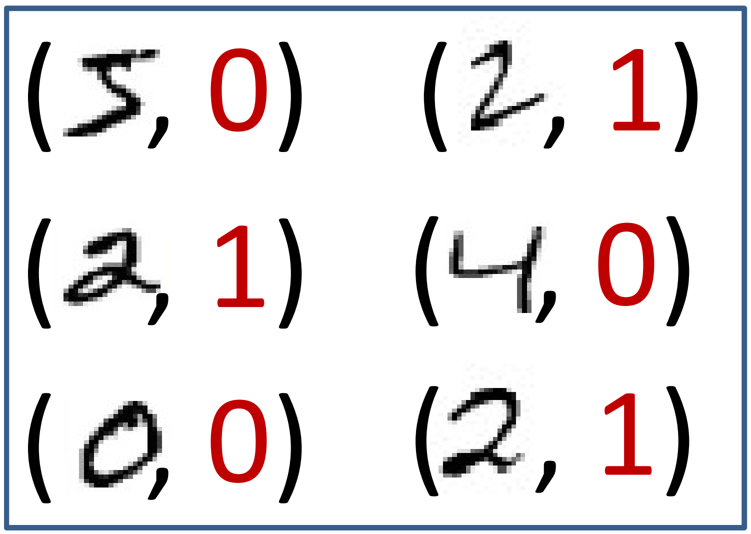


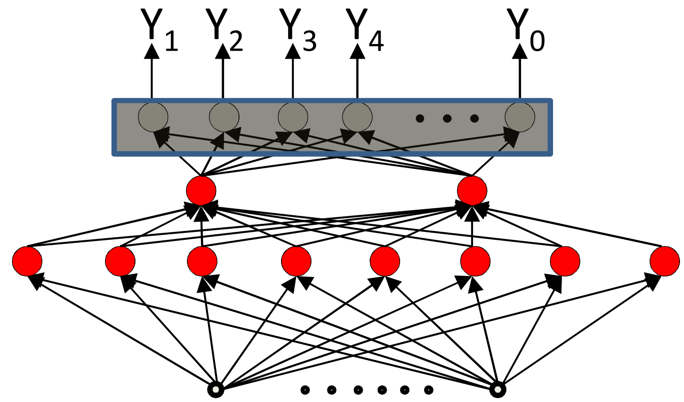
Sigmoid output neuron

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

Recognizing the digit

Training data





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