Machine learning

Introduction to Neural Networks

Mohammad-Reza A. Dehaqani

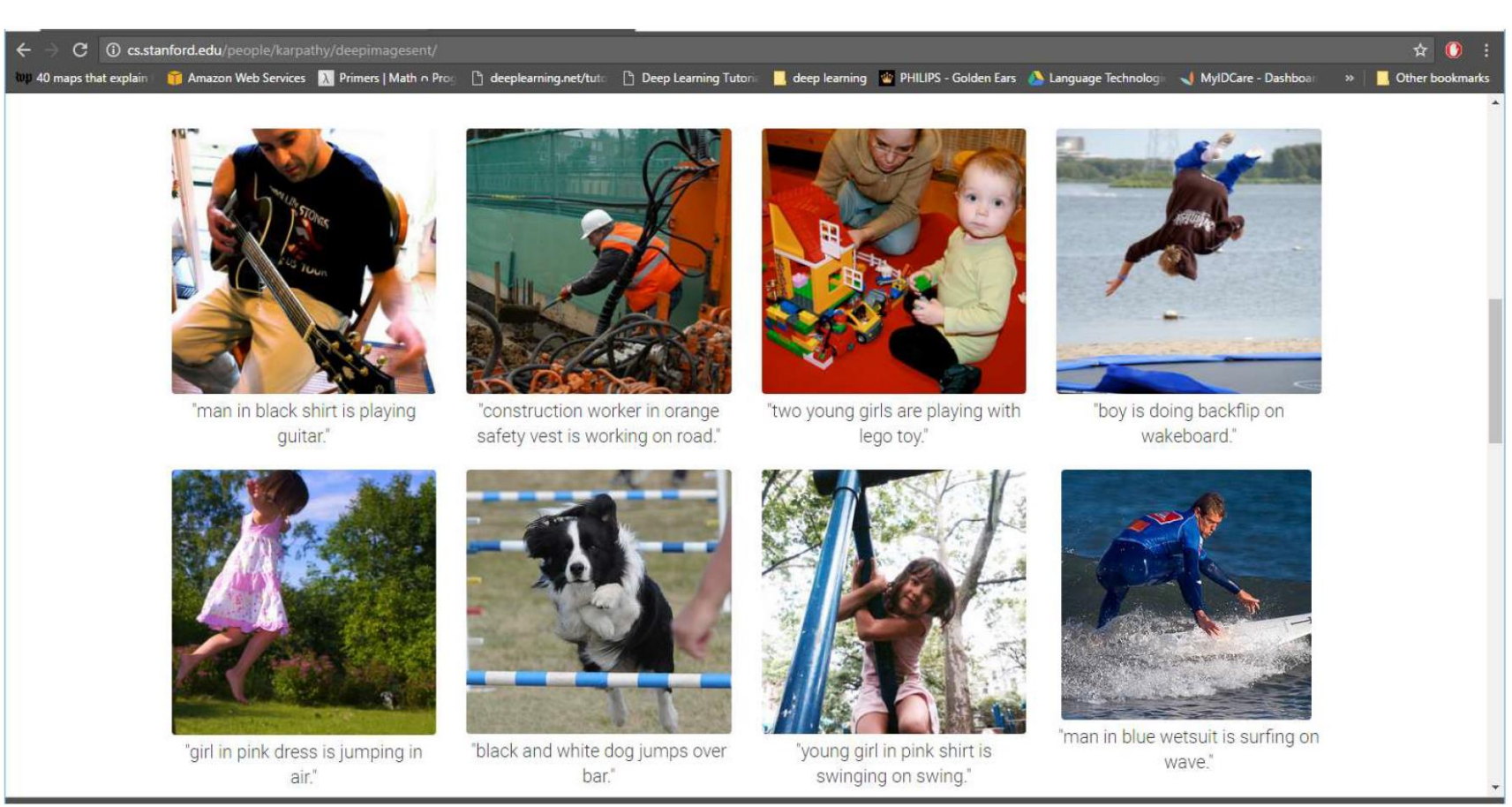
dehaqani@ut.ac.ir

Slides are adopted form CMU deep NN course

Neural Networks are taking over!

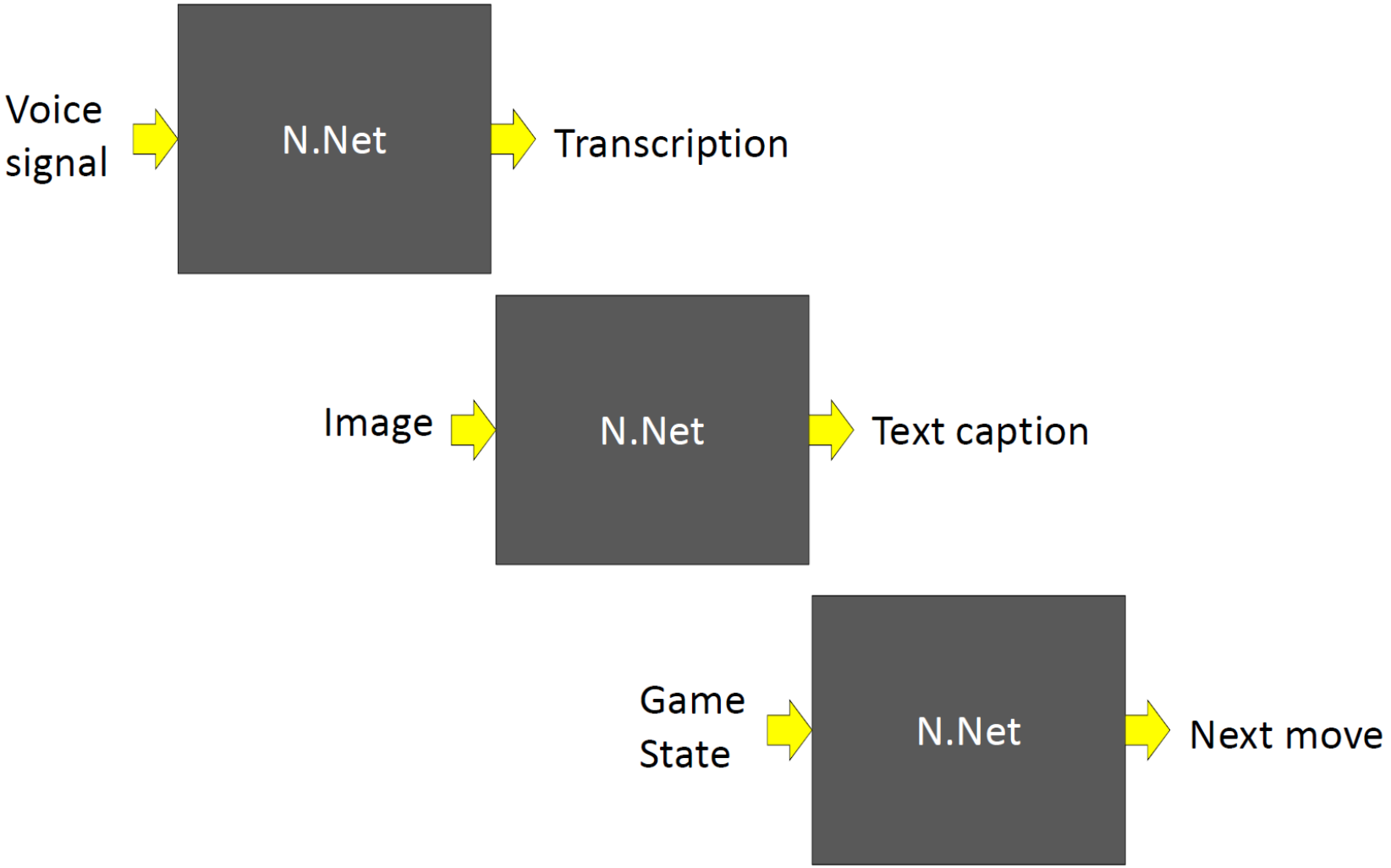
- Neural networks have become one of the major areas recently in various pattern recognition, prediction, and analysis problems
  - In many problems they have established the **state of the art**
  - Often exceeding previous benchmarks by large margins
- Neural nets can **do anything**

Breakthroughs with neural networks



- Captions generated entirely by a neural network

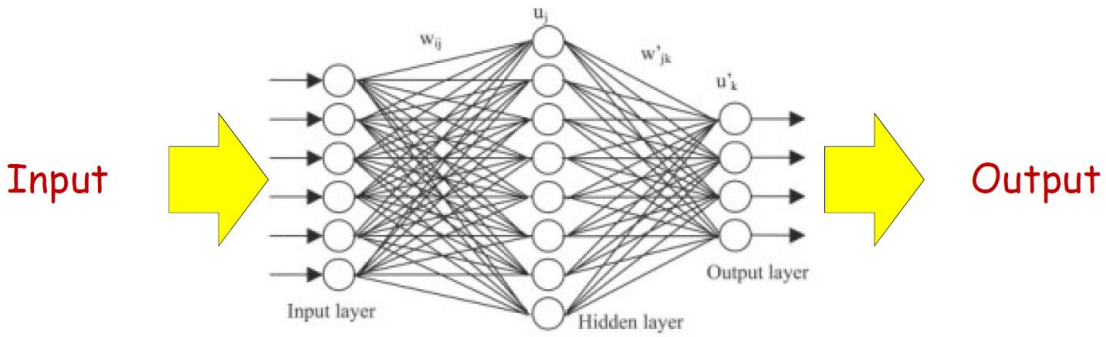
So what are neural networks??



- What are these **boxes**?

NNets in AI

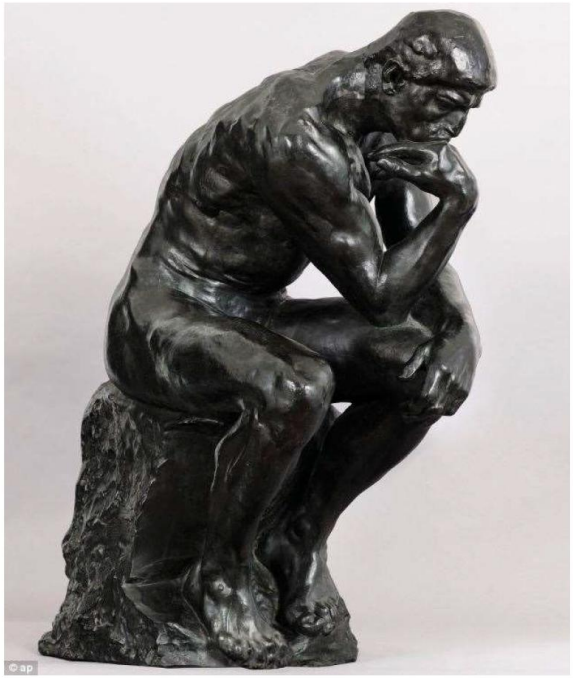
- Model memory
  - Loopy networks can "remember" patterns
- Represent probability distributions
- Over integer, real and complex-valued domains
- MLPs can model both a posteriori and a priori distributions of data
- The network as a function



- Inputs are numeric vectors: Numeric representation of input, e.g. audio, image, game state, etc.
- Outputs are numeric scalars or vectors: Numeric "encoding" of output from which actual output can be derived
  - E.g. a score, which can be compared to a threshold to decide if the input is a face or not
- Output may be multi-dimensional, if task requires it

The magical capacity of humans

- Humans can
  - Learn
  - Solve problems
  - Recognize patterns
  - Create
  - Cogitate
  - ...



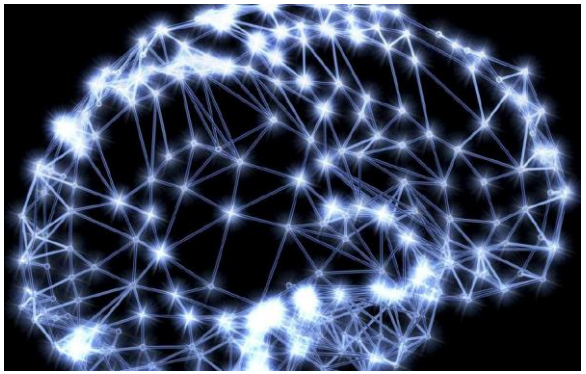
"The Thinker!"
by Augustin Rodin

- "If the brain was simple enough to be understood - we would be too simple to understand it!"
  - Marvin Minsky

Early Models of Human Cognition

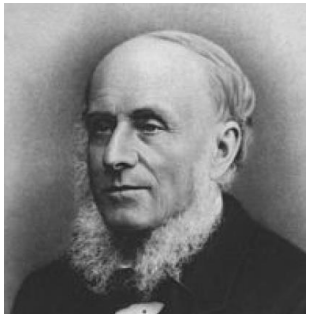
- **Associationism**: Humans learn **through association**
  - 400BC-1900AD: Plato, David Hume, Ivan Pavlov..
- "**Pairs** of thoughts become associated based on the organism's **past experience**"
- Learning is a mental process that forms associations between **temporally related** phenomena
- Aristotle's four laws of association:
  - The law of **contiguity**. Things or events that occur close together in space or time get linked together
  - The law of **frequency**. The more often two things or events are linked, the more powerful that association.
  - The law of **similarity**. If two things are similar, the thought of one will trigger the thought of the other
  - The law of **contrast**. Seeing or recalling something may trigger the recollection of something opposite

But how do we store them? Dawn of Connectionism



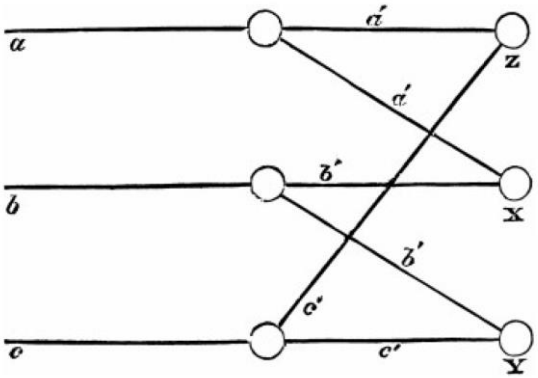
- Observation: The Brain: is a mass of **interconnected neurons** (Mid 1800s)
- Many neurons connect in to each neuron
- Each neuron connects out to many neurons

- **Alexander Bain**:
- philosopher, mathematician, logician, linguist, professor
- 1873: The information **is in the connections** (The mind and body (1873))



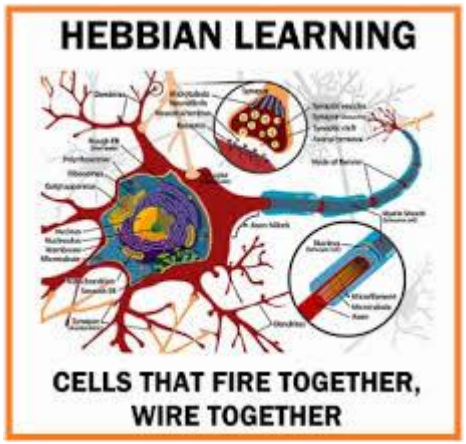
Bain's Idea

- Neural **Groupings**:
  - Neurons excite and stimulate each other
  - Different combinations of inputs can result in different outputs



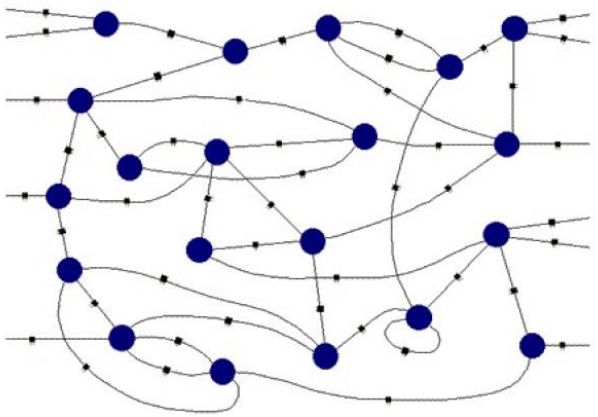
- Making **Memories**
  - "when two impressions concur, or closely succeed one another, the nerve currents find some bridge or place of continuity, better or worse, according to the abundance of nerve matter available for the transition."

- Predicts "Hebbian" learning (three quarters of a century before Hebb!)



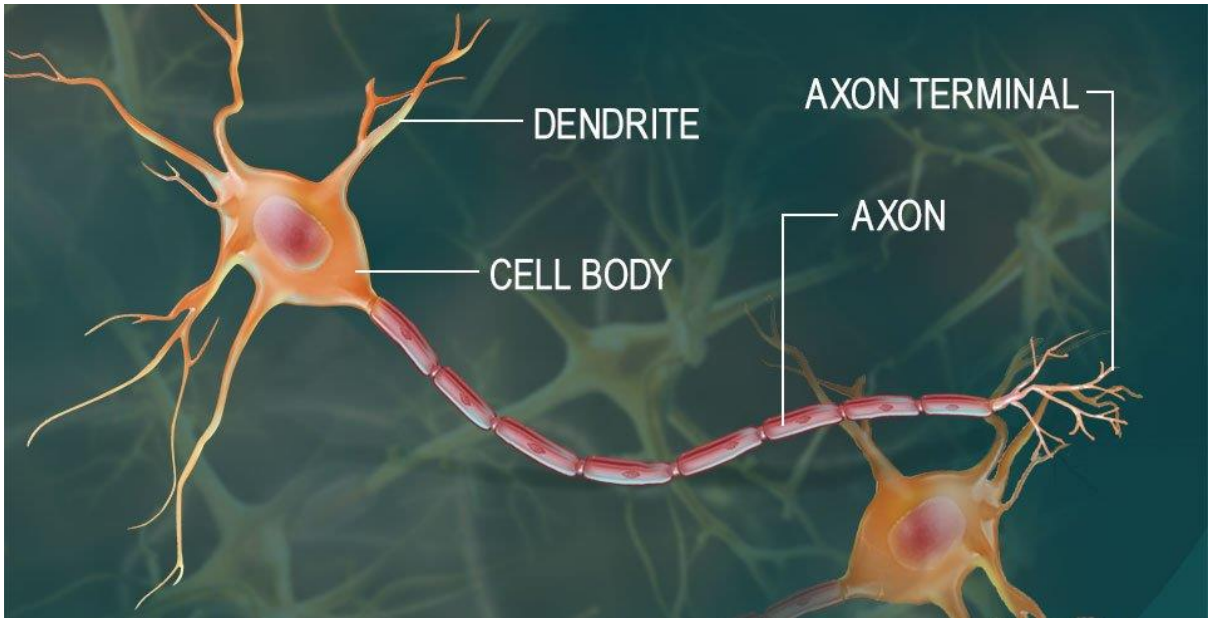
Connectionist Machines

- Current neural network models are connectionist machines
- Requirements for a connectionist system
- (Bechtel and Abrahamson, 91)
  - The connectivity of units
  - The activation function of units
  - The nature of the learning procedure that modifies the connections between units, and
  - How the network is interpreted semantically



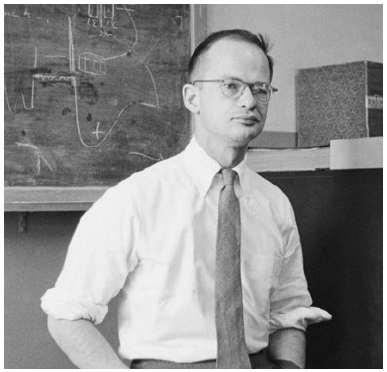
Modelling the brain

- What are the units?

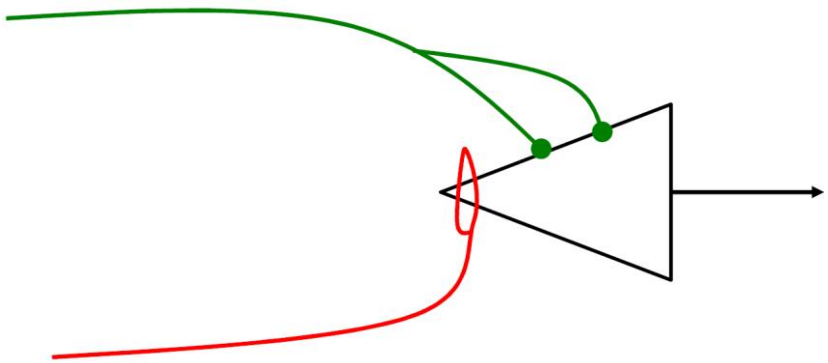


- Signals come in through the dendrites into the Soma
- A signal goes out via the axon to other neurons

McCullough and Pitts



- The Doctor and the Hobo..
  - Warren McCulloch: Neurophysician
  - Walter Pitts: Homeless wannabe logician



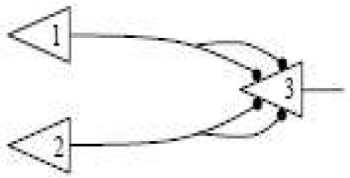
- The McCulloch and Pitts model
  - Pitts was only 20 years old
  - Neuron is an ‘‘all-or-none’’ process
    - Excitatory synapse: Transmits weighted input to the neuron
    - Inhibitory synapse: Any signal from an inhibitory synapse forces output to zero
    - The activity of any inhibitory synapse absolutely prevents excitation of the neuron at that time. Regardless of other inputs at this time

McCulloch, Warren S., and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity." *The bulletin of mathematical biophysics* 5.4 (1943): 115-133.

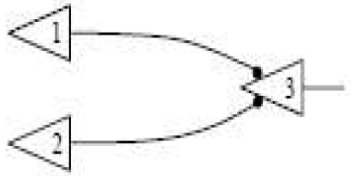
McCulloch and Pitts Model

- Could compute **arbitrary Boolean propositions**
  - Since any Boolean function can be emulated, any Boolean function can be composed
- Models for memory
  - Networks with loops can "remember"
  - Lawrence Kubie (1930): Closed loops in the central nervous system explain memory
- Didn't provide a learning mechanism

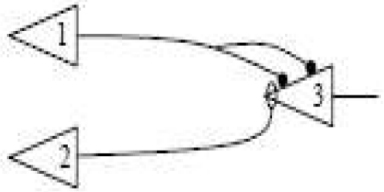
$N_3(t) = N_1(t\text{-}1)$ or $N_2(t\text{-}1)$

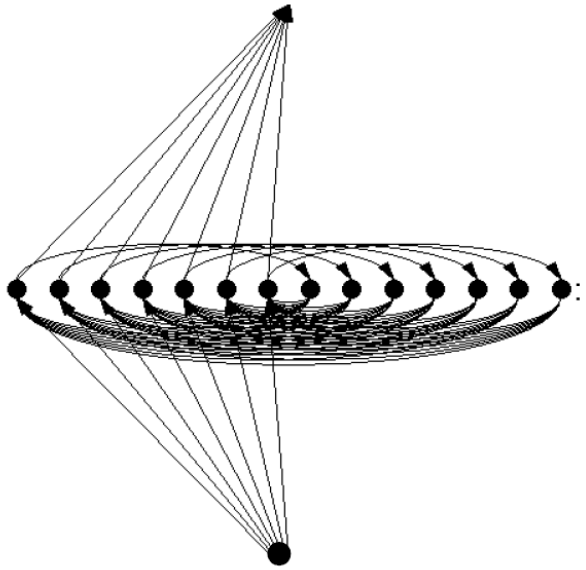


$N_3(t) = N_1(t\text{-}1)$ and $N_2(t\text{-}1)$



$N_3(t) = N_1(t\text{-}1)$ and $\sim N_2(t\text{-}1)$





Donald Hebb

- When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic chan takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."
- As A repeatedly excites B, its ability to excite B improves
- – Neurons that fire together wire together

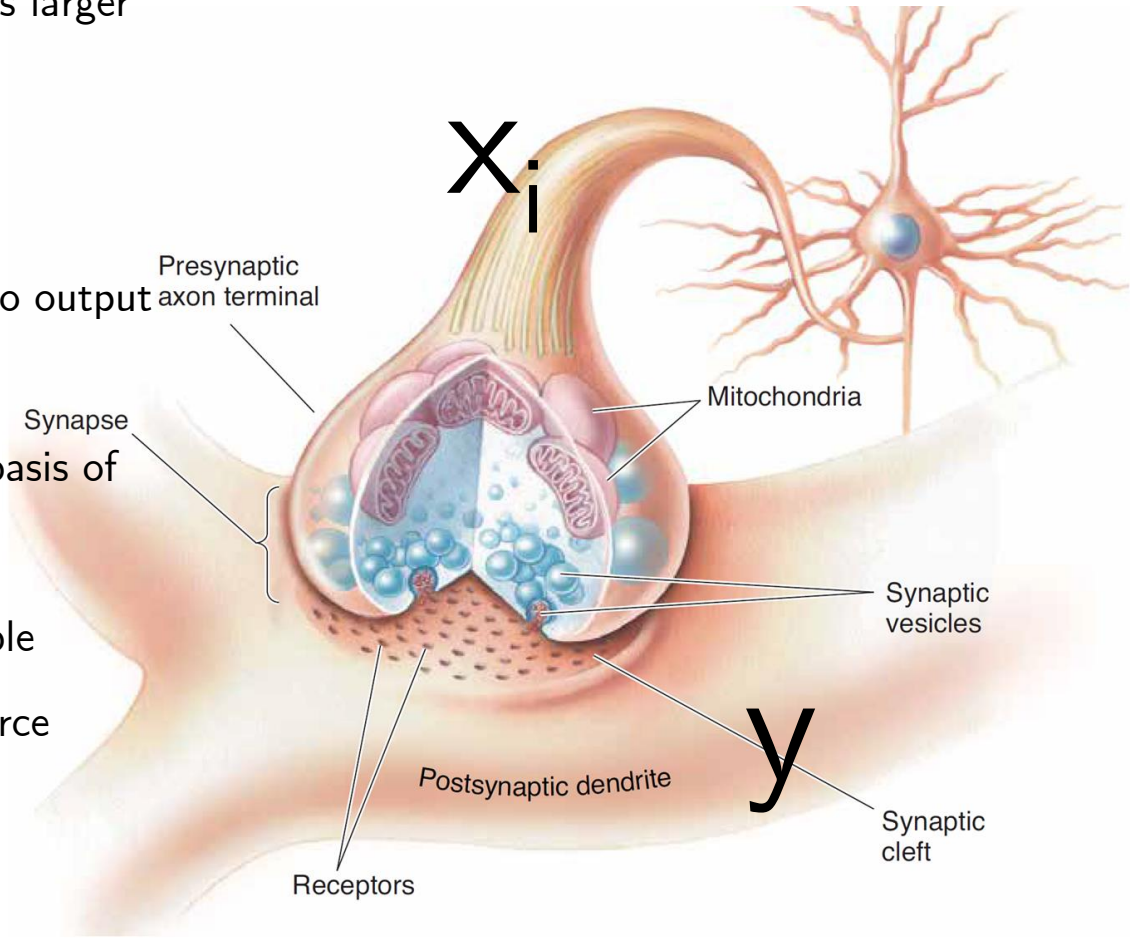
"Organization of behavior", 1949

Hebbian Learning

- If neuron x_i repeatedly triggers neuron y, the synaptic knob connecting x_i to y gets larger
- In a mathematical model:

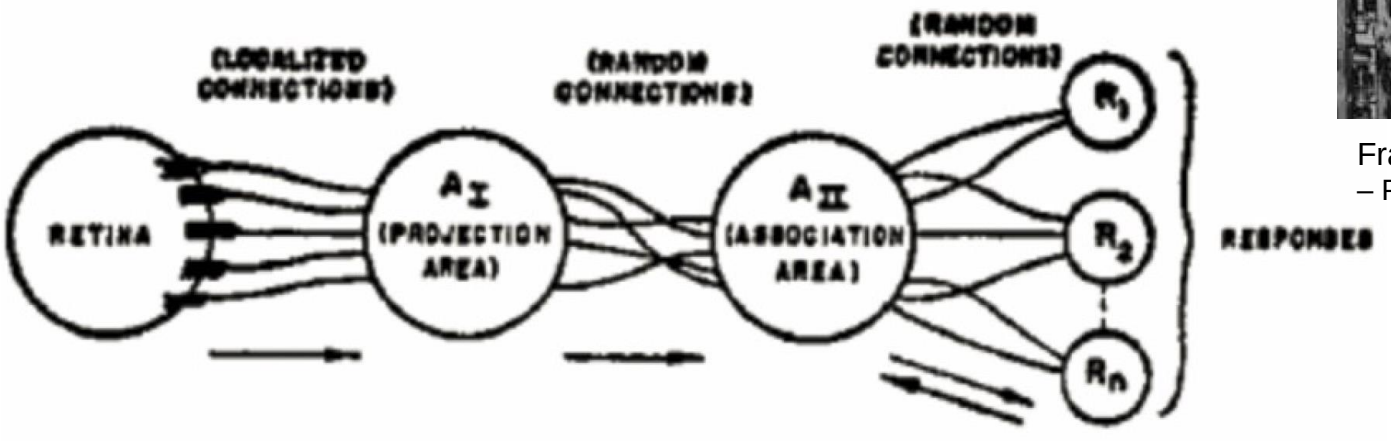
$$w_i = w_i + \eta x_i y$$

- Weight w_i of the i^{th} neuron's input to output neuron y
- This simple formula is actually the basis of many learning algorithms in ML
- This models is fundamentally unstable
  - Stronger connections will enforce themselves
  - No notion of "competition"
  - No reduction in weights
  - Learning is unbounded

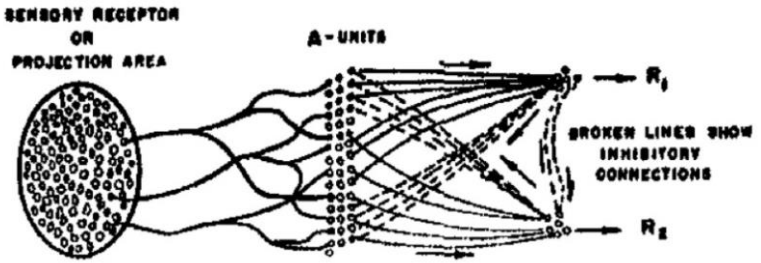


Rosenblatt's perceptron

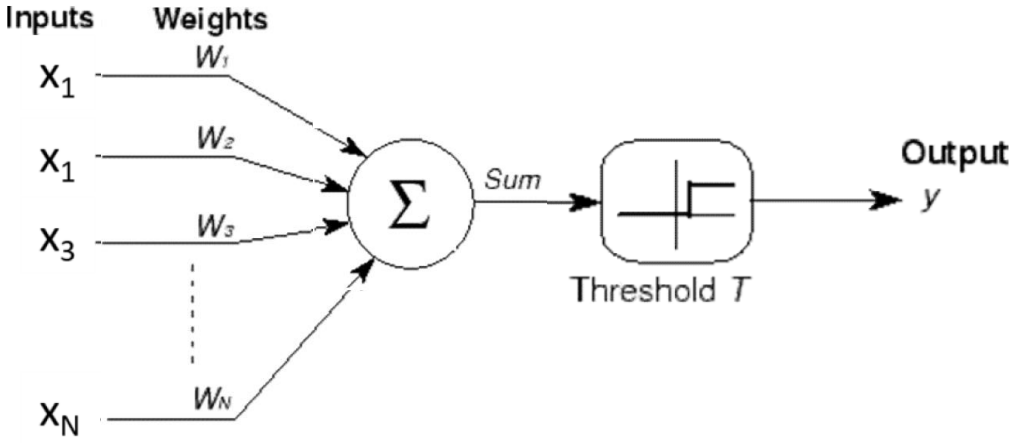
Frank Rosenblatt
– Psychologist, Logician



- Original perceptron model
  - Groups of sensors (S) on retina combine onto cells in association area A1
  - Groups of A1 cells combine into Association cells A2
  - Signals from A2 cells combine into response cells R
  - All connections may be excitatory or inhibitory
- Even included **feedback** between A and R cells



Simplified mathematical model



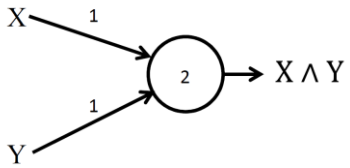
- Number of inputs combine linearly
- – Threshold logic: Fire if combined input exceeds threshold

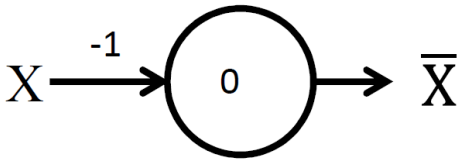
$$Y = \begin{cases} 1 \;\; if \;\; \sum_i w_i x_i - T > 0 \\ 0 \qquad\qquad\qquad else \end{cases}$$

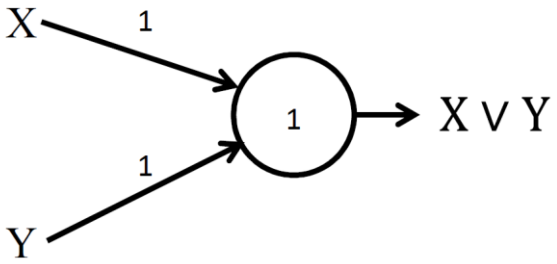
Also provided a learning algorithm

$$\mathbf{W} = \mathbf{W} + \eta(d(\mathbf{x}) - y(\mathbf{x}))\mathbf{x}$$

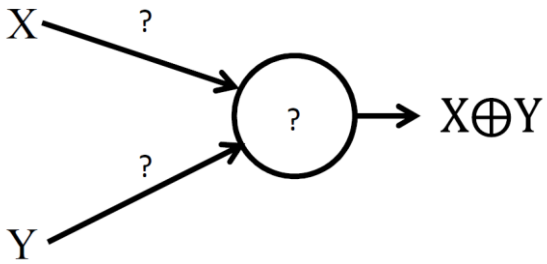
- Sequential Learning:
  - $d(\mathbf{x})$ is the desired **output** in response to input $\mathbf{x}$
  - $Y(\mathbf{x})$ is the actual output in response to $\mathbf{x}$
- Boolean tasks
  - Update the weights whenever the perceptron output is wrong
  - Proved convergence for linearly separable classes
- Easily shown to mimic any Boolean gate







- **But**



No solution for XOR!
Not universal!

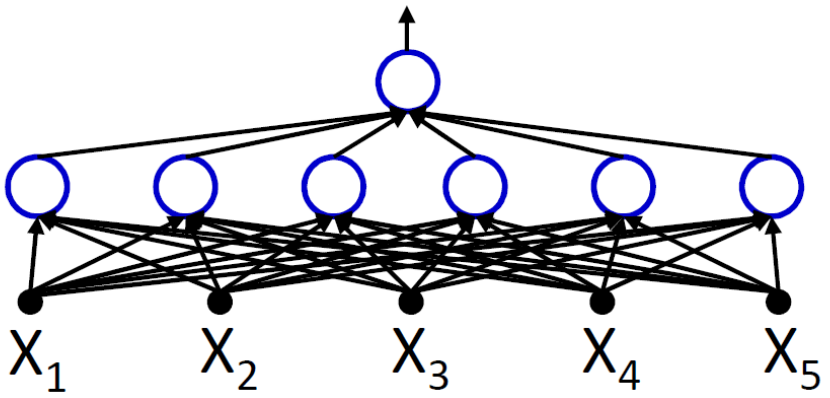
- Minsky and Papert, 1968

A one-hidden-layer MLP is a Universal Boolean Function

Truth Table

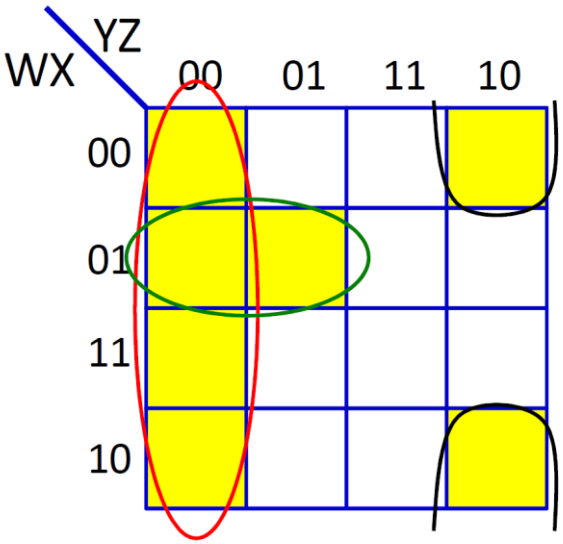
X_1	X_2	X_3	X_4	X_5	Y
0	0	1	1	0	1
0	1	0	1	1	1
0	1	1	0	0	1
1	0	0	0	1	1
1	0	1	1	1	1
1	1	0	0	1	1

$$Y = \bar{X}_1\bar{X}_2X_3X_4\bar{X}_5 + \bar{X}_1X_2\bar{X}_3X_4X_5 + \bar{X}_1X_2X_3\bar{X}_4\bar{X}_5 + X_1\bar{X}_2\bar{X}_3\bar{X}_4X_5 + X_1\bar{X}_2X_3X_4X_5 + X_1X_2\bar{X}_3\bar{X}_4X_5$$

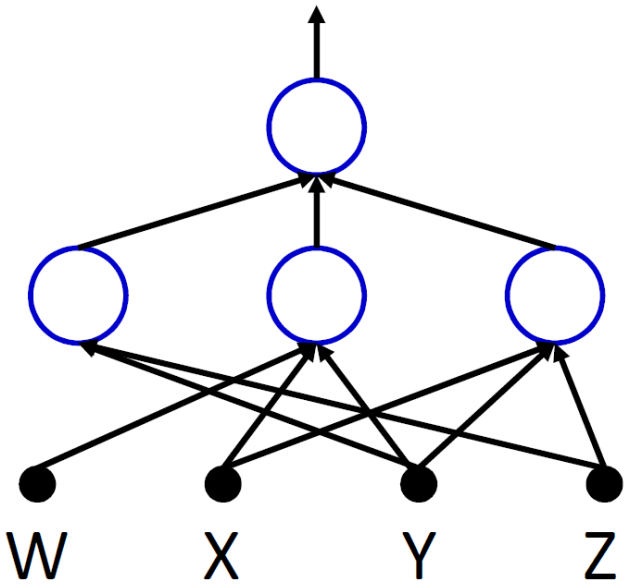


Any truth table can be expressed in this manner!

Reducing a Boolean Function

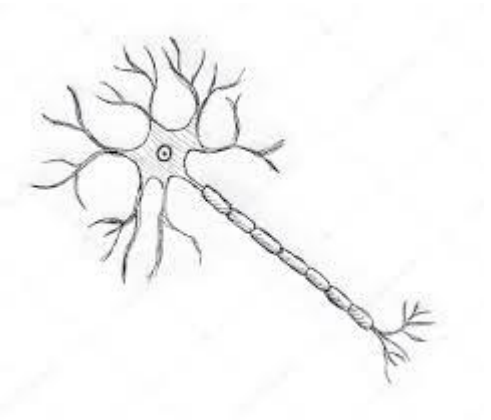


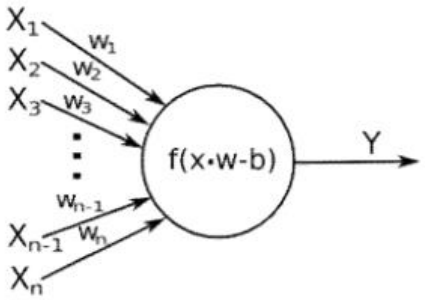
$$O = \bar{Y}\bar{Z} + \bar{W}X\bar{Y} + \bar{X}Y\bar{Z}$$

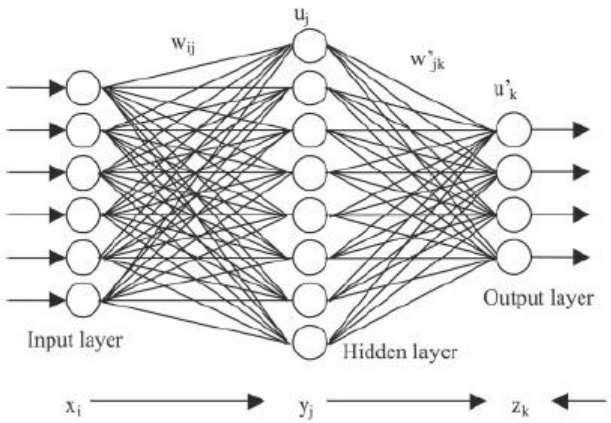


- Reduced DNF form:
  - Find groups
  - Express as reduced DNF
  - Boolean network for this function needs only 3 hidden units
- Reduction of the DNF reduces the size of the one-hidden-layer network

A single neuron is not enough



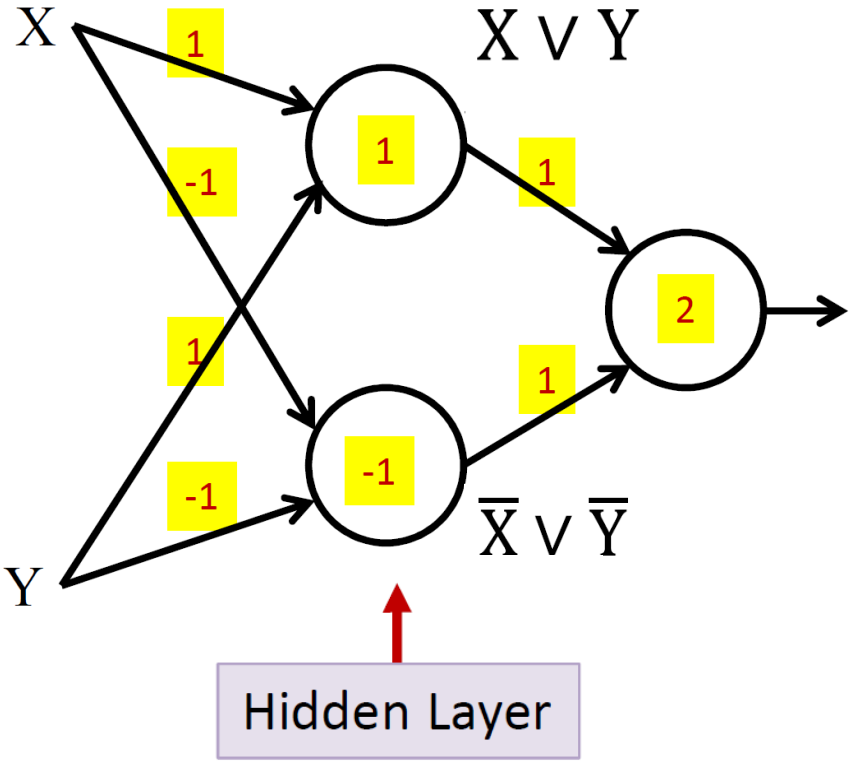




- Individual elements are weak computational elements
- – Marvin Minsky and Seymour Papert, 1969, Perceptrons:

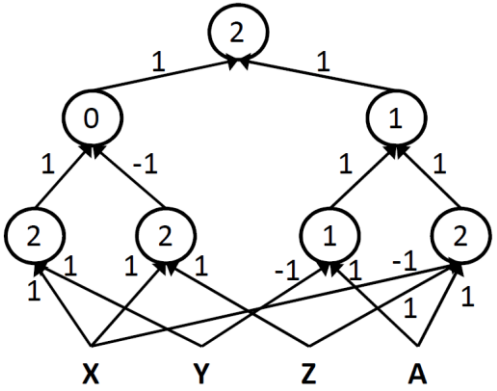
  An Introduction to Computational Geometry
- Networked elements are required

Multi-layer Perceptron!



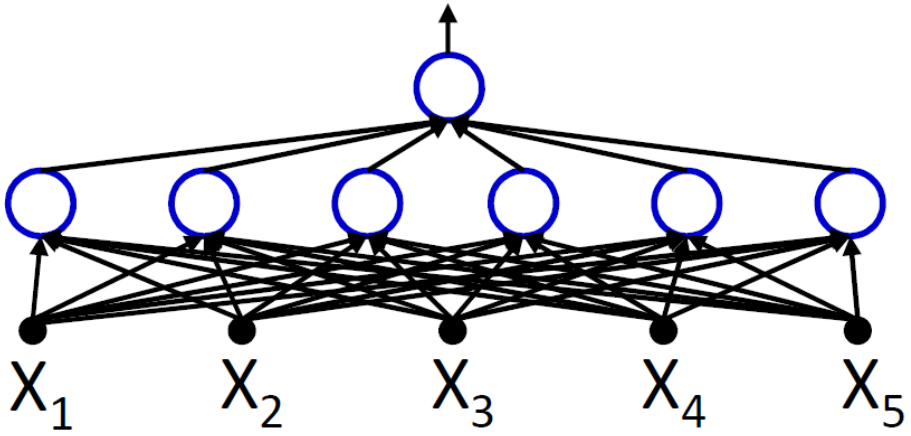
$X \oplus Y$

$((A\&\bar{X}\&Z)|(A\&\bar{Y}))\&((X\ \&\ Y)|\overline{(X\&Z)})$



- XOR
  - The first layer is a "hidden" layer
  - Also originally suggested by Minsky and Paper 1968
- Can compose arbitrarily complicated Boolean functions

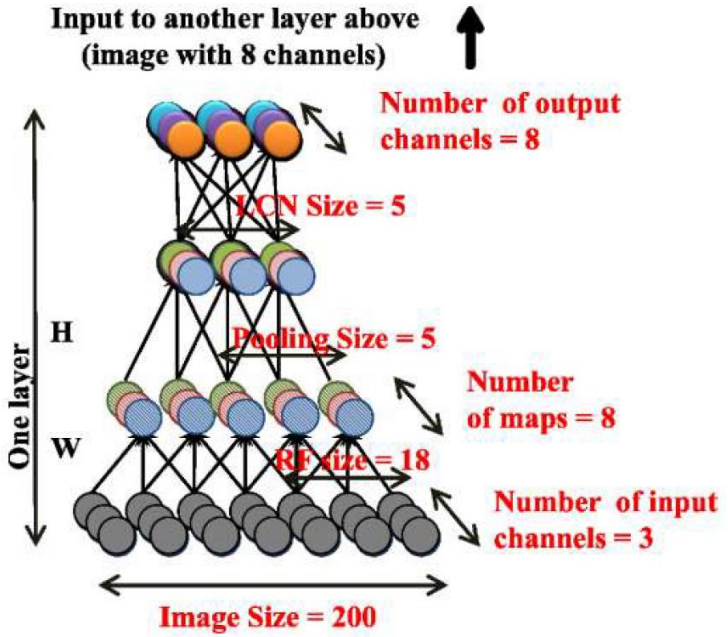
The actual number of parameters in a network

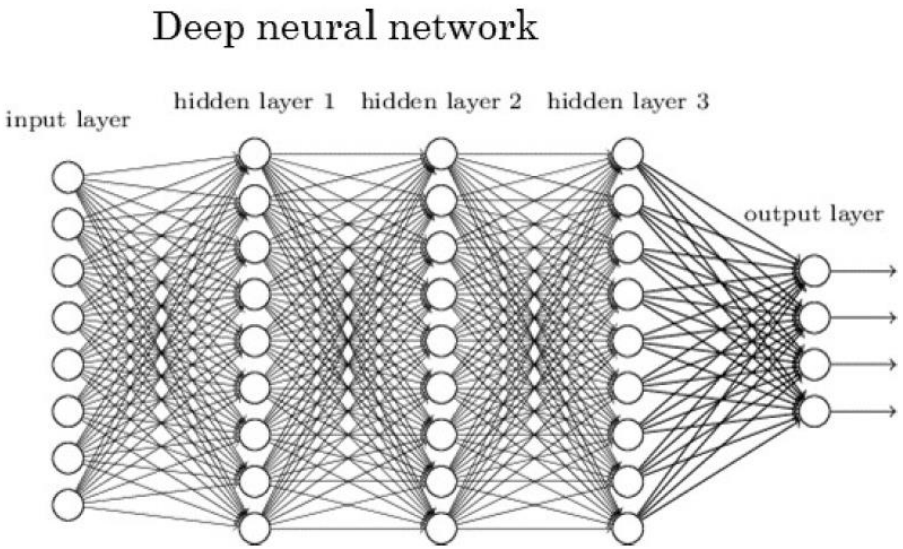


- The actual number of parameters in a network is the number of connections
  - In this example there are 30
- This is the number that really matters in software or hardware implementations
- Having a few extra layers can greatly reduce network size

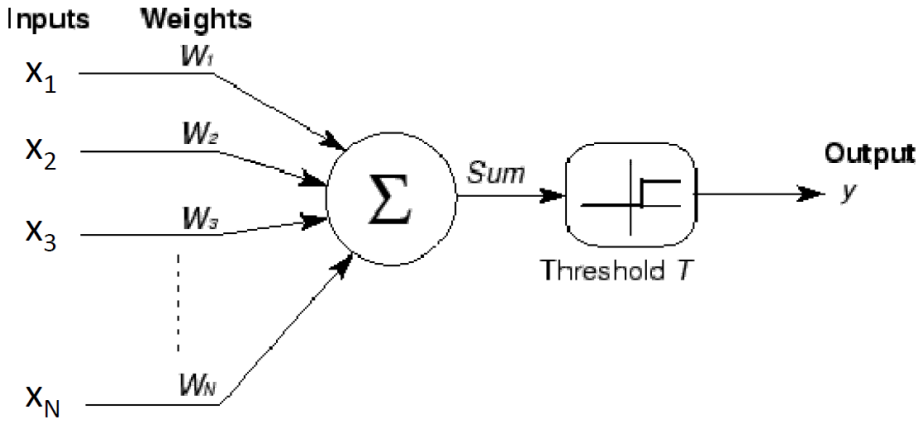
Deep Structures

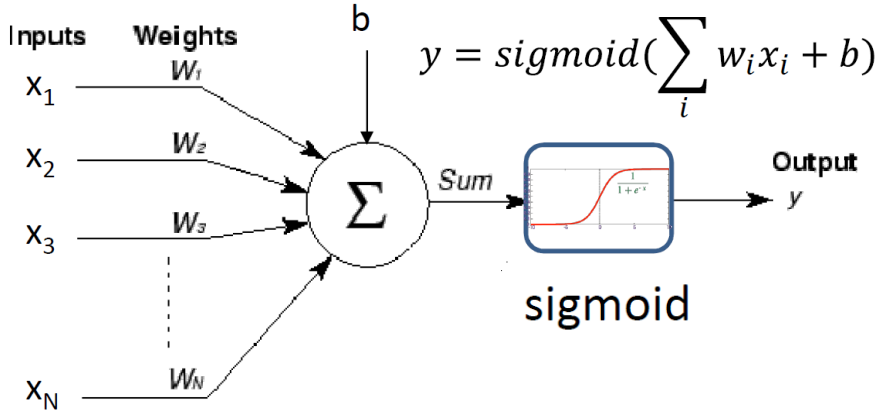
- "depth" is the length of the longest path from a source to a sink
- Layered deep structure
- "Deep" $\mapsto$ Depth > 2



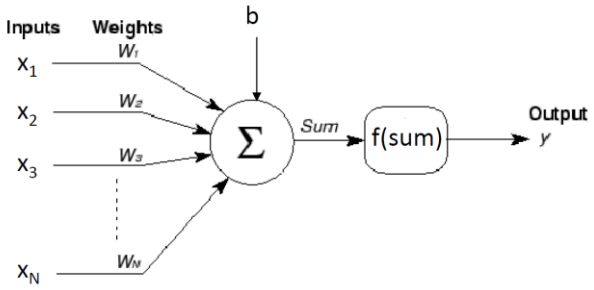


But our brain is not Boolean: The perceptron with real inputs



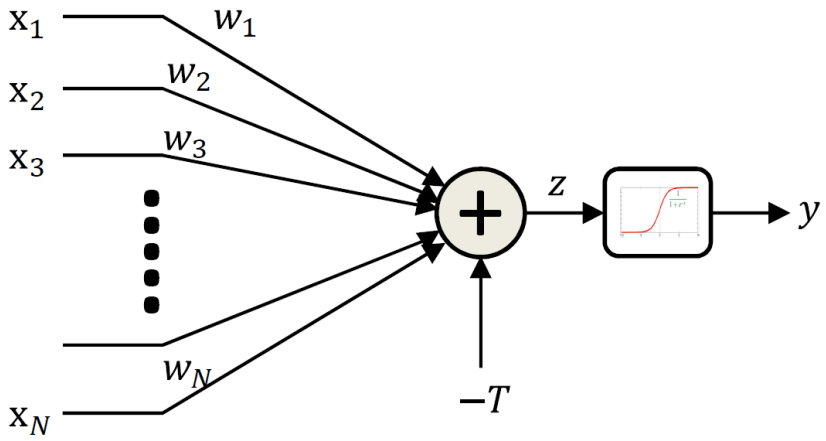


- $x_1 \ldots x_N$ are real valued
- $w_1 \ldots w_N$ are real valued
- Unit "fires" if weighted input exceeds a threshold
- The output y can also be real valued
  - Sometimes viewed as the "probability" of firing



- Any real-valued "activation" function may operate on the weighted sum input
- The perceptron maps real-valued inputs to real-valued outputs

The "soft" perceptron (logistic)

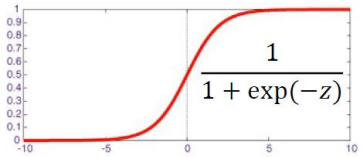


$$z = \sum_i w_i x_i - T$$

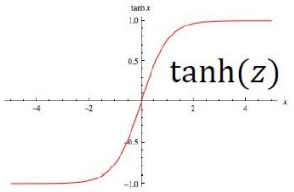
$$y = \frac{1}{1 + exp(-z)}$$

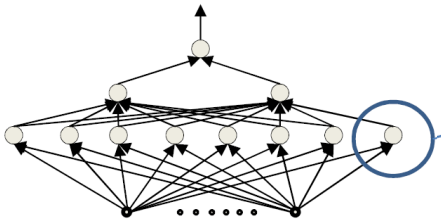
- A "squashing" function instead of a threshold at the output
  - The sigmoid "activation" replaces the threshold
- Activation: The function that acts on the weighted combination of inputs (and threshold)
- Other "activations"

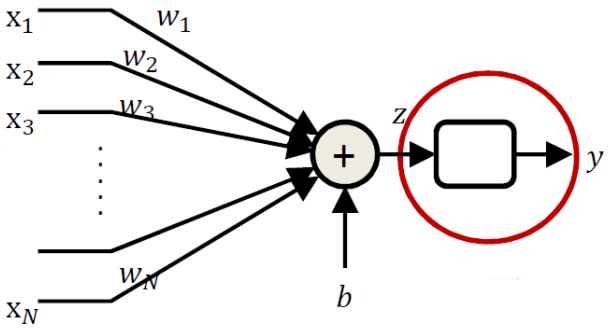
sigmoid

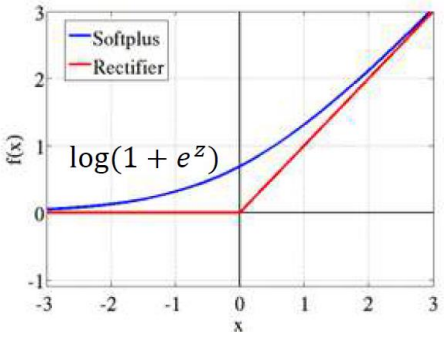


tanh

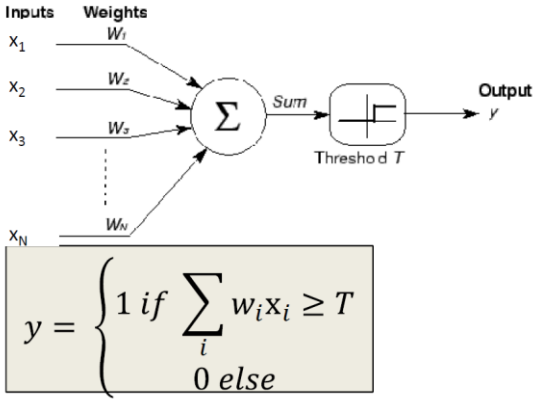




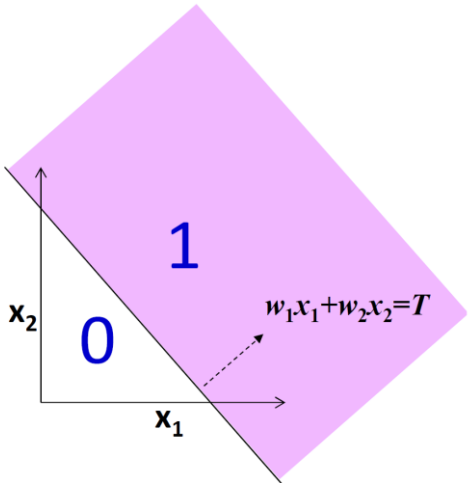


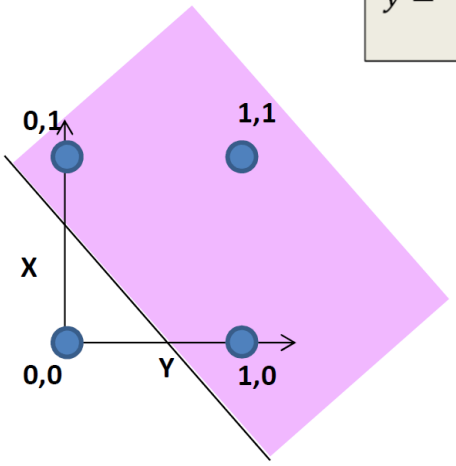


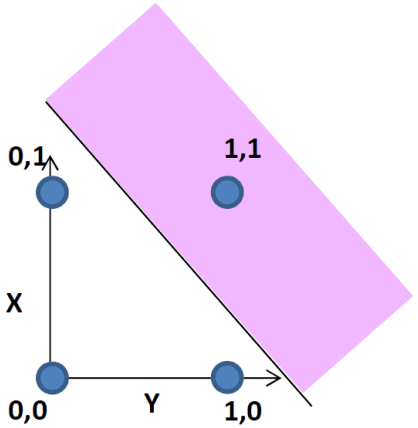
A Perceptron on Reals is Linear classifier

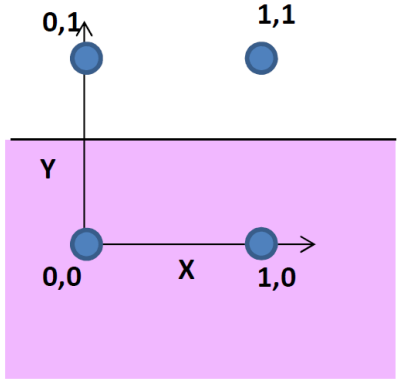


$$y = \begin{cases} 1 \text{ if } \sum_i w_i x_i \geq T \\ 0 \text{ else} \end{cases}$$





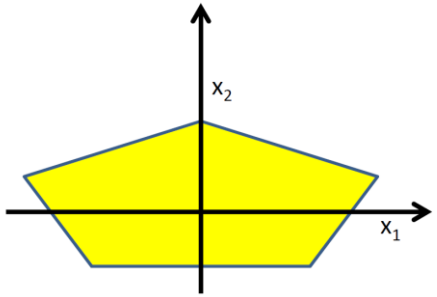


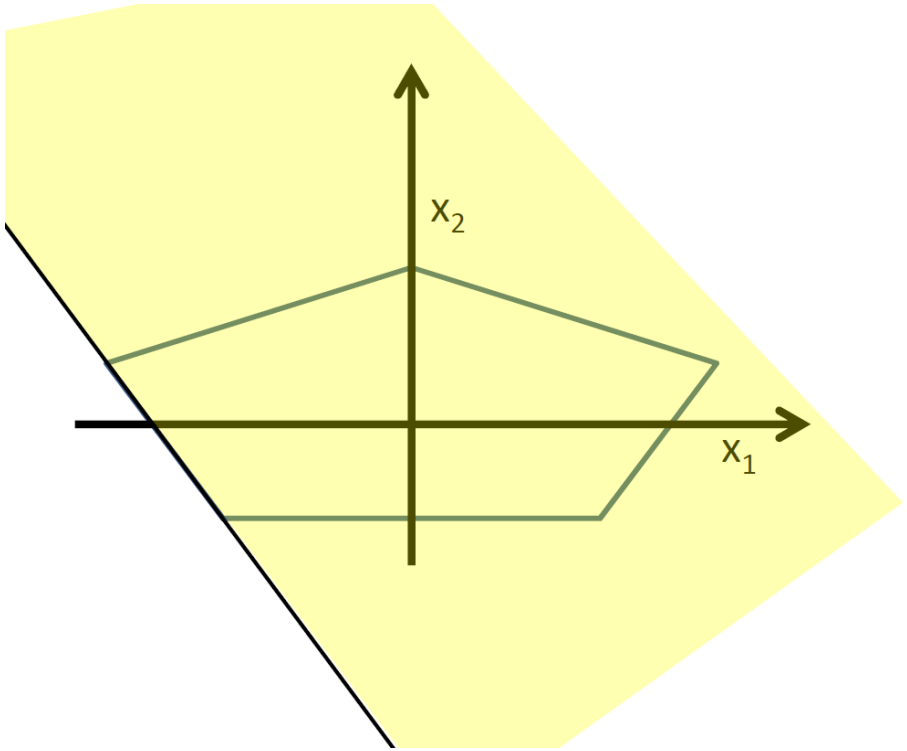


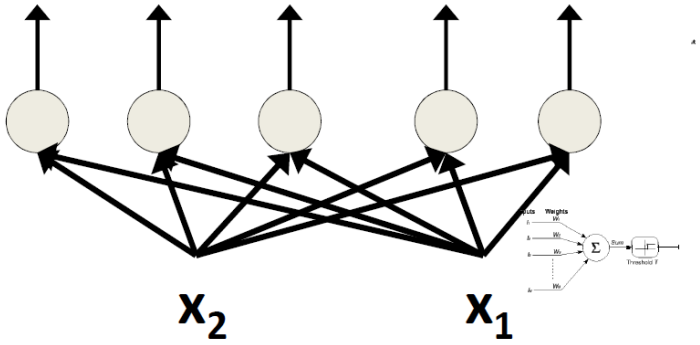
- Boolean perceptrons are also linear classifiers
  - Purple regions have output 1 in the figures
  - What are these functions
  - Why can we not compose an XOR?

Composing complicated "decision" boundaries

- Can now be composed into"networks" to compute arbitrary classification "boundaries"

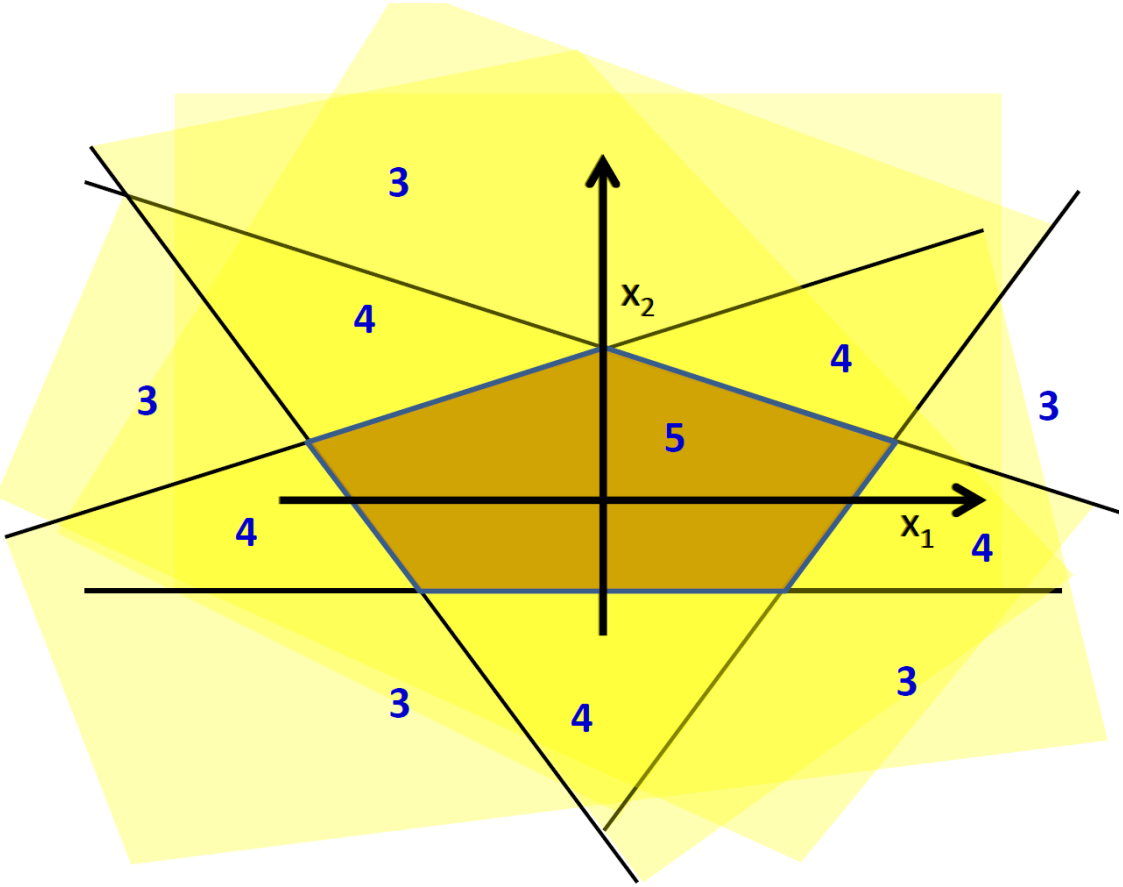


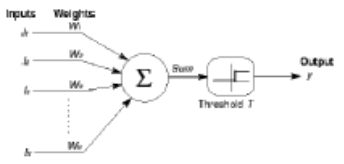


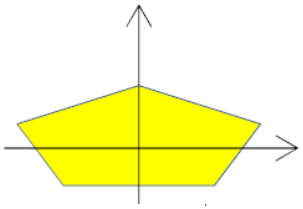


- The network must fire if the input is in the coloured area

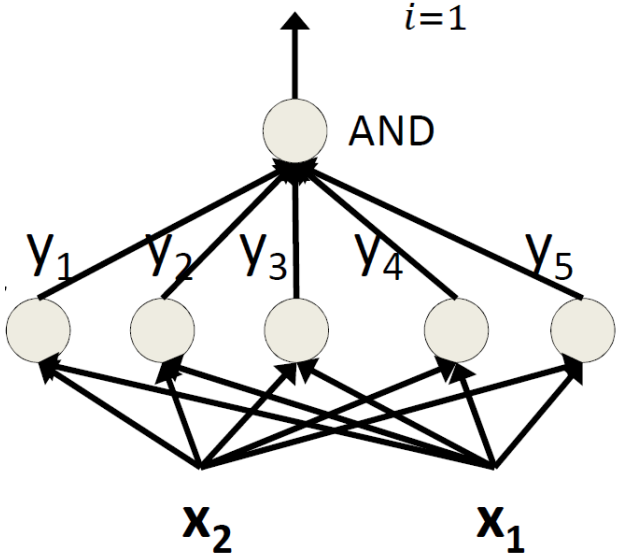
Booleans over the reals; MLPs as universal classifiers





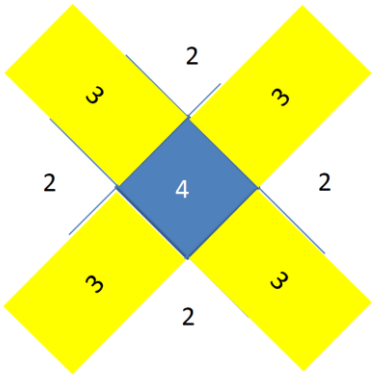


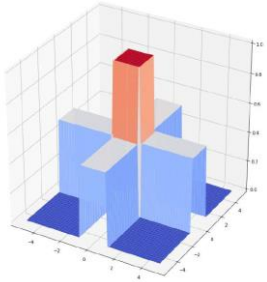
$$\sum_{i=1}^{N} y_i \geq 5?$$

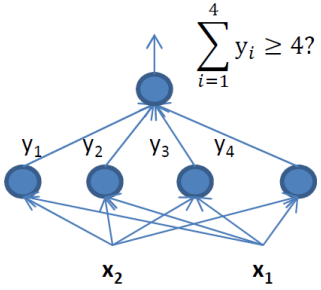


The network must fire if the input is in the coloured area 102

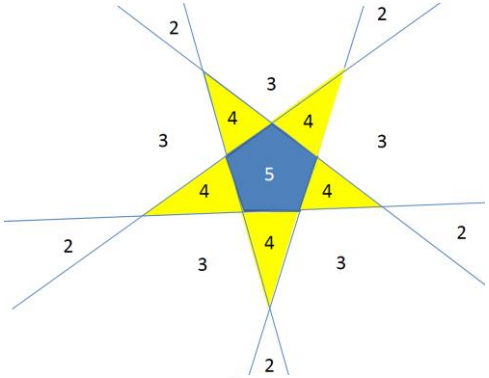
Composing a Square decision boundary

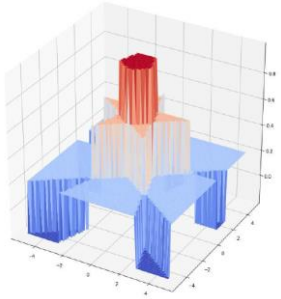


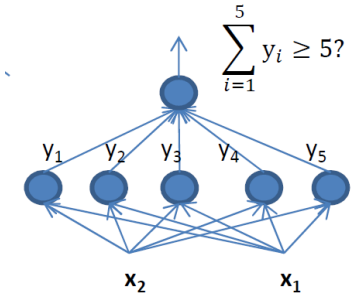




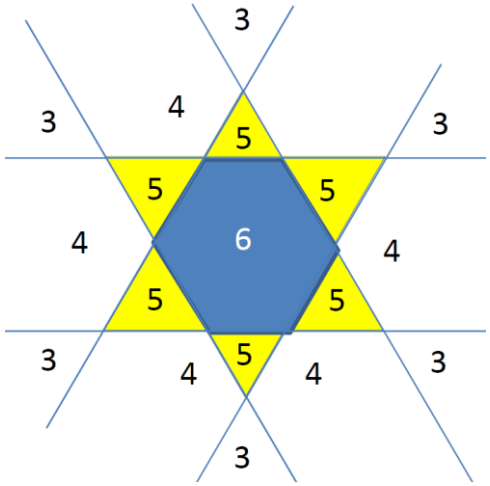
Composing a pentagon

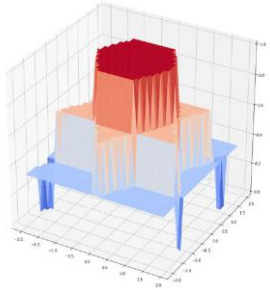


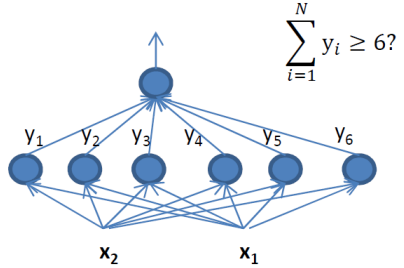




Composing a hexagon

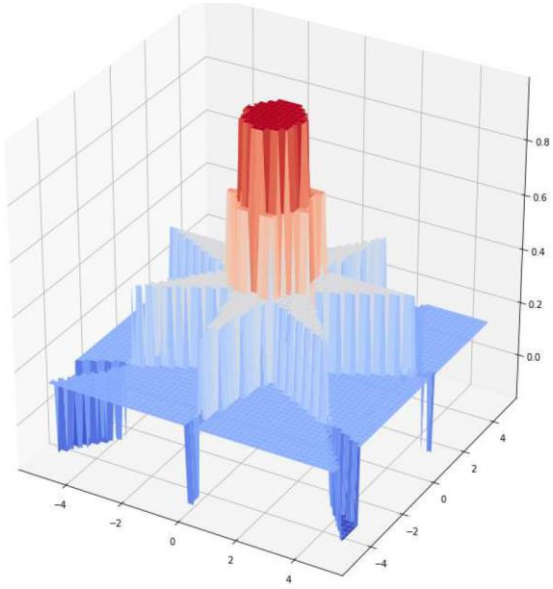




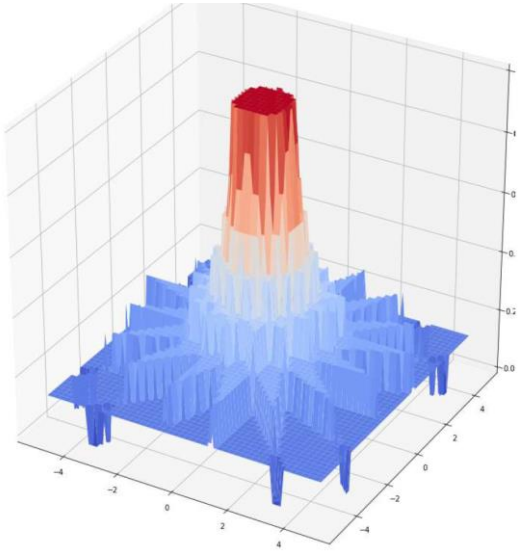


Increasing the number of sides reduces the area outside the polygon that have N/2 < Sum < N

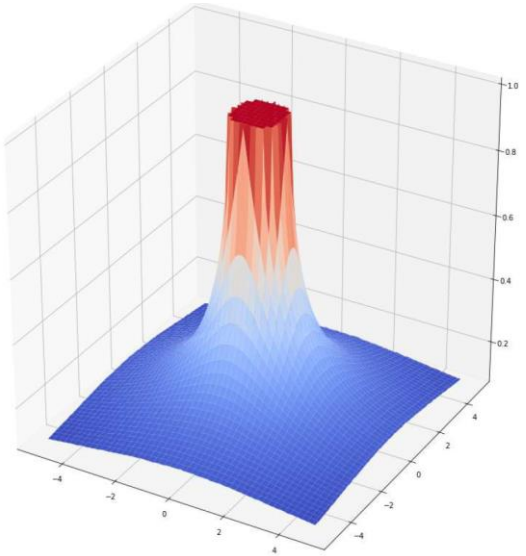
heptagon

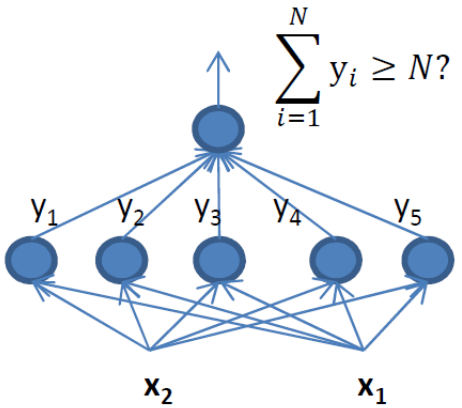


16 sides

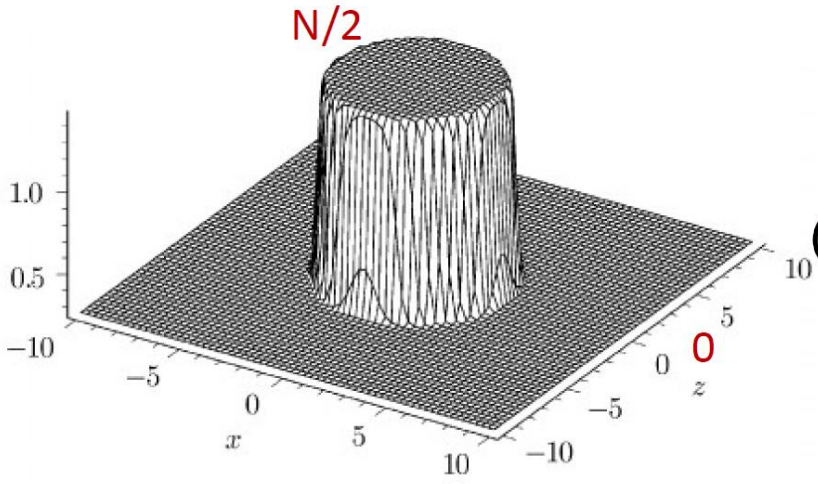


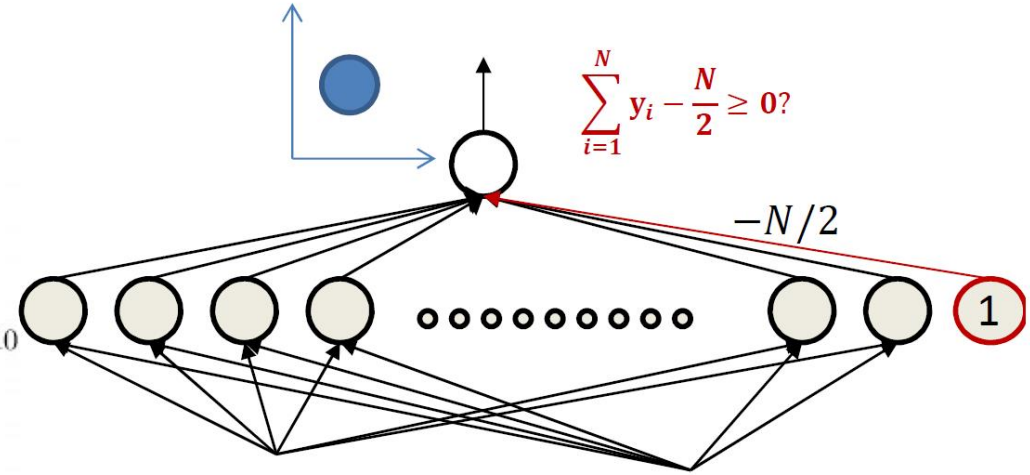
1000 sides





In the limit: Composing a circle

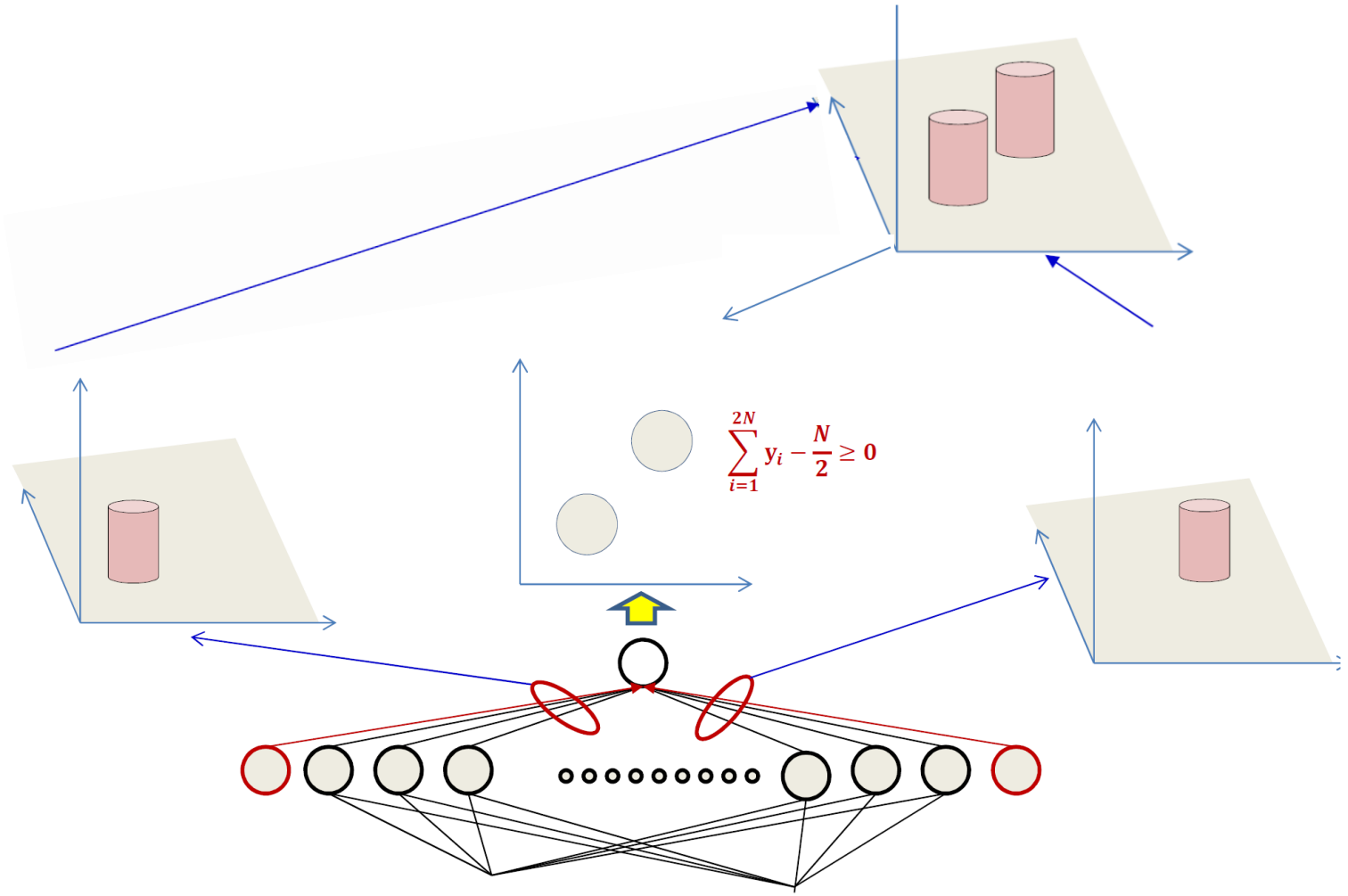




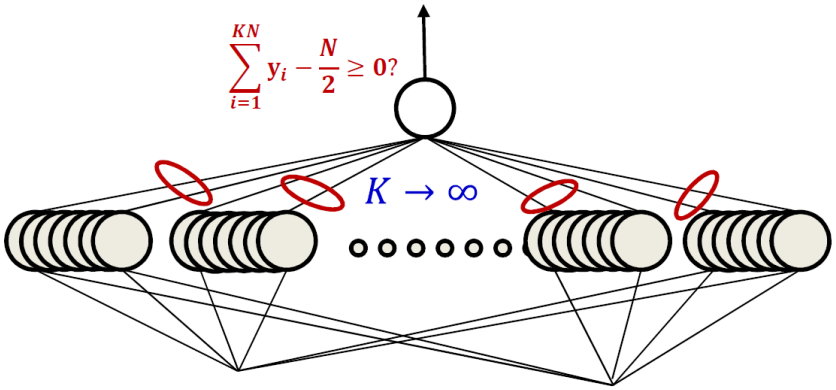
- The circle net
  - Very large number of neurons
  - Sum is N inside the circle, N/2 outside almost everywhere
  - Circle can be at any location

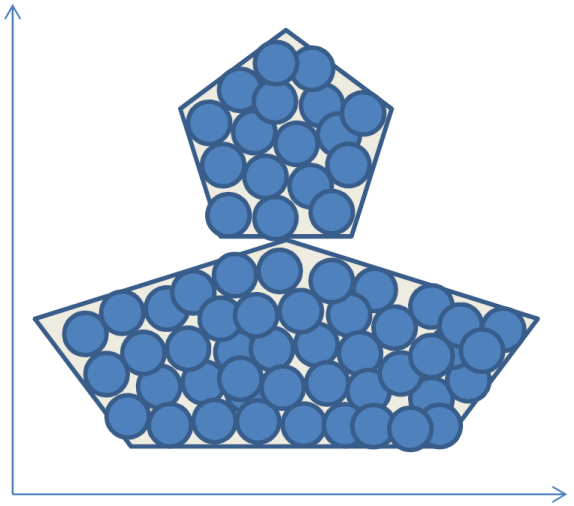
Sum is N/2 inside the circle, 0 outside almost everywhere

Adding circles



Composing an arbitrary figure

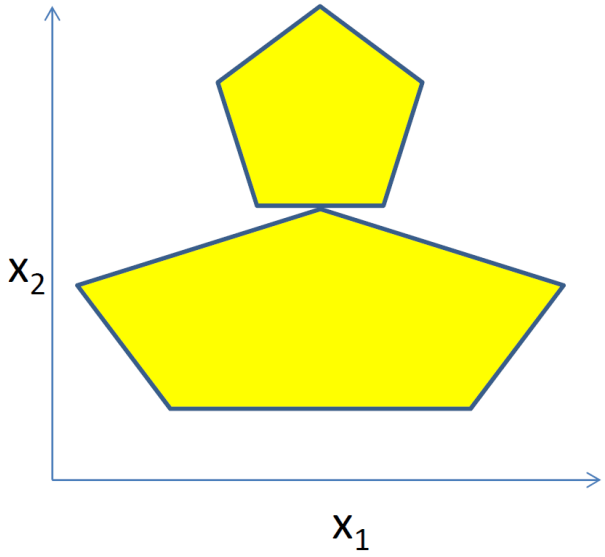


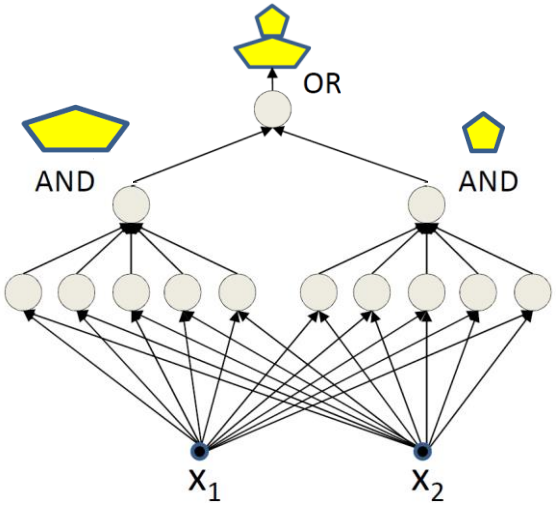


- Just fit in an arbitrary number of circles
  - More accurate approximation with greater number of smaller circles
  - Can achieve arbitrary precision

MLP: Universal classifier

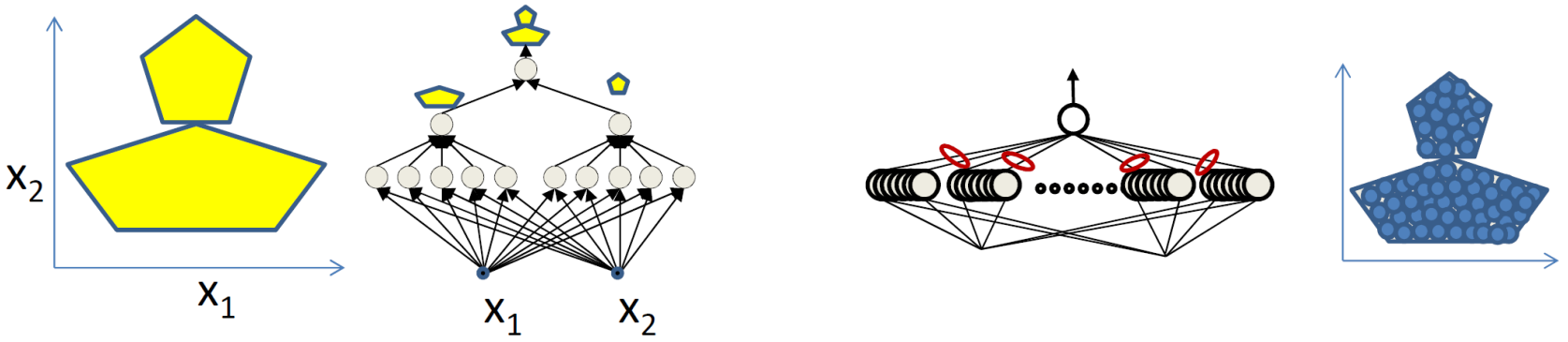
More complex decision boundaries
The need for depth





- Network to fire if the input is in the yellow area
  - "OR" two polygons
  - A third layer is required
- Can compose very **complex** decision boundaries
- Classification problems: **finding** decision boundaries in high-dimensional space
  - Can be performed by an MLP
- MLPs can classify real-valued inputs

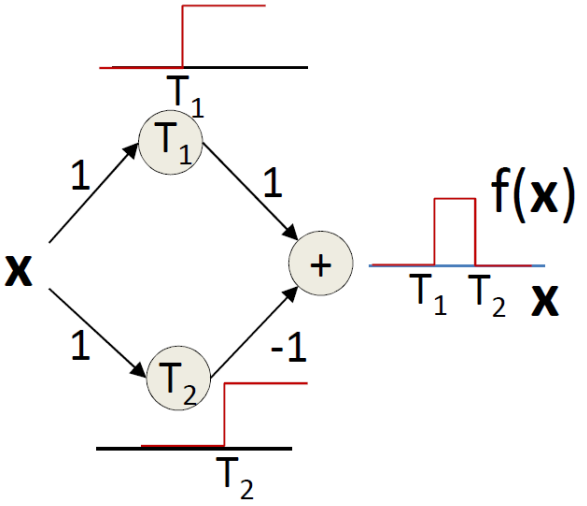
Depth vs. width

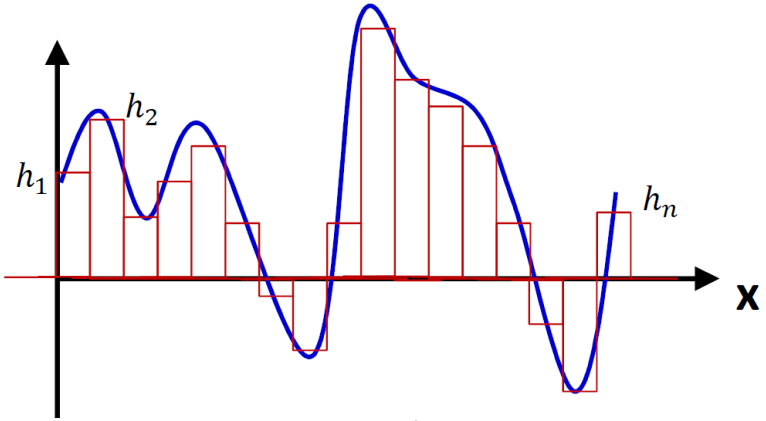


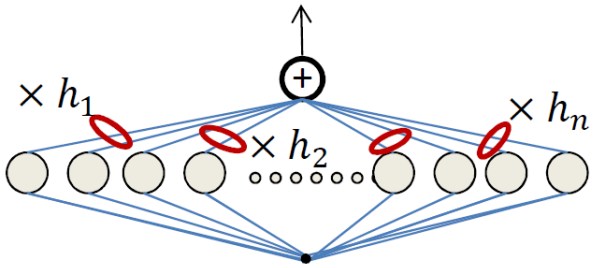
- Deeper networks can require far fewer neurons
- A polynomial of degree n requires a network of depth $\log_2(n)$
- The number of neurons required in a shallow network is potentially exponential in the dimensionality of the input
  - (this is the worst case)

MLP as a continuous-valued regression; MLPs as universal approximators

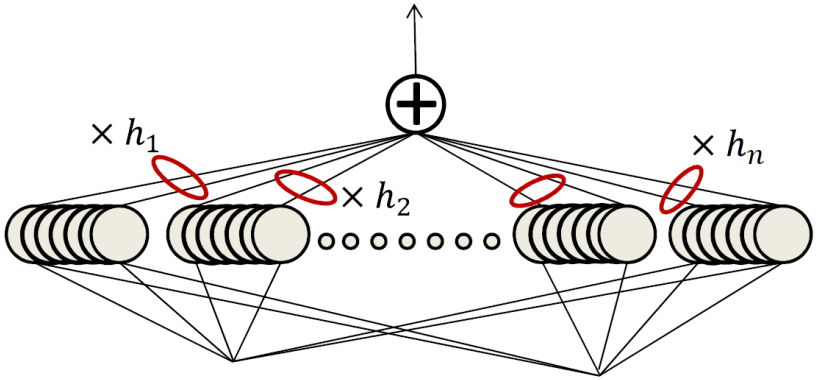
- A simple 3-unit MLP with a "summing" output unit can generate a "square pulse" over an input
  - Output is 1 only if the input lies between T1 and T2
  - T1 and T2 can be arbitrarily specified
- An MLP with many units can model an arbitrary function over an input
  - To arbitrary precision
- Simply make the individual pulses narrower
- This generalizes to functions of any number of inputs

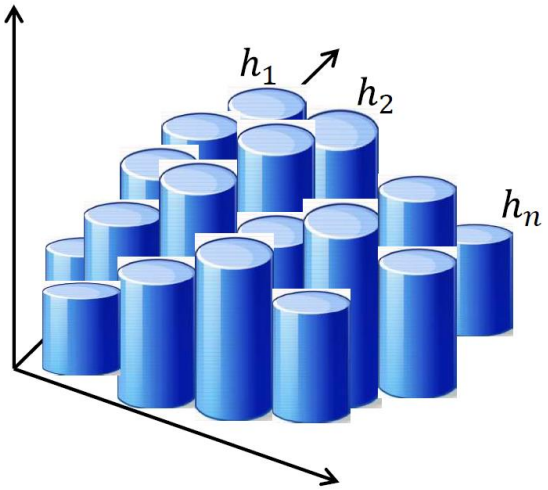






MLP as a continuous-valued function

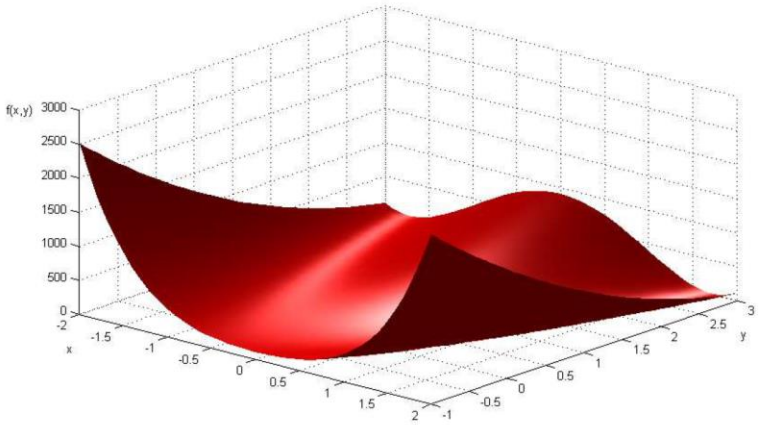




- MLPs can actually compose arbitrary functions in any number of dimensions!
  - Even with only one layer
- As sums of scaled and shifted cylinders
  - To arbitrary precision
- By making the cylinders thinner
  - The MLP is a **universal approximator**!

Story so far

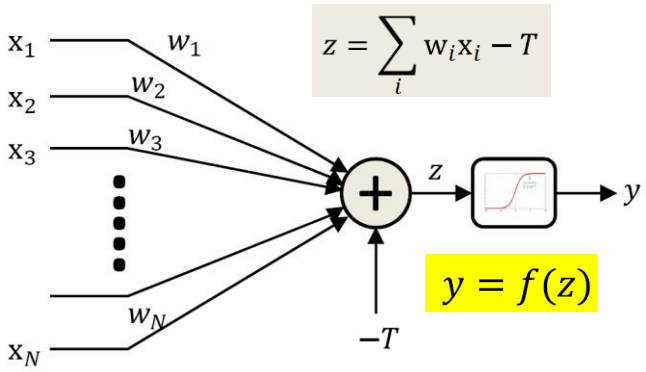
- MLPs are connectionist computational models
  - Individual perceptrons are computational equivalent of neurons
  - The MLP is a layered composition of many perceptrons
- MLPs can model Boolean functions
  - Individual perceptrons can act as Boolean gates
  - Networks of perceptrons are Boolean functions
- MLPs are Boolean machines
  - They represent Boolean functions over linear boundaries
  - They can represent arbitrary decision boundaries
  - They can be used to classify data

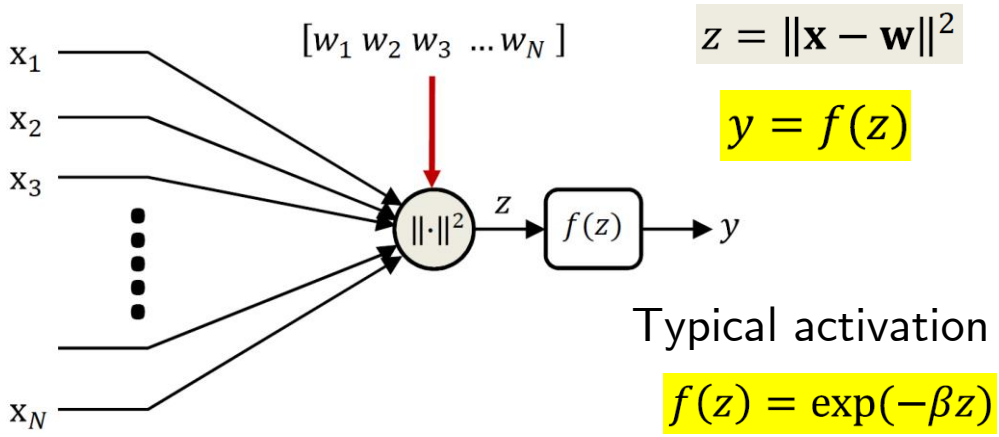


- MLPs are universal classifiers
- MLPs are universal function approximators
- A single-layer MLP can approximate anything to arbitrary precision
  - But could be exponentially or even infinitely wide in its inputs size
- Deeper MLPs can achieve the same precision with far fewer neurons
  - Deeper networks are more expressive

Brief segue: RBF networks

Perceptrons so far

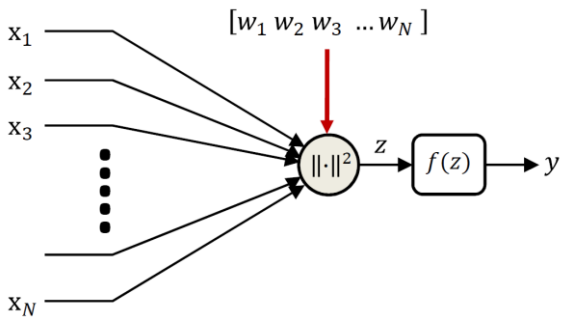


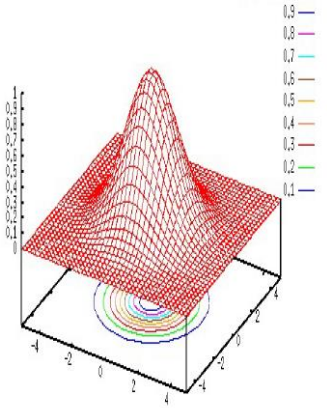


- The output is a function of the distance of the input from a "center"
  - The "center" is the parameter specifying the unit
  - The most common activation is the exponent
- β is a "bandwidth" parameter
  - But other similar activations may also be used
- Key aspect is radial symmetry, instead of linear symmetry

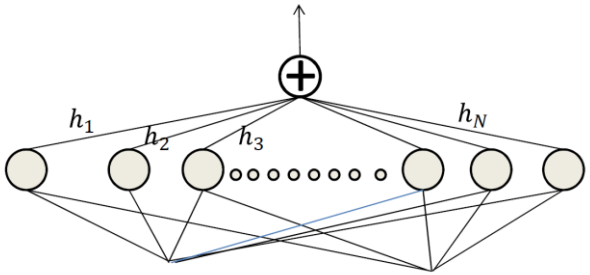
RBF networks as universal approximators

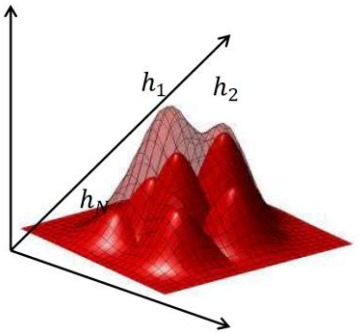
- Radial basis functions can compose cylinder-like outputs with just a single unit with appropriate choice of bandwidth (or activation function)
  - As opposed to $N \rightarrow \infty$ units for the linear perceptron





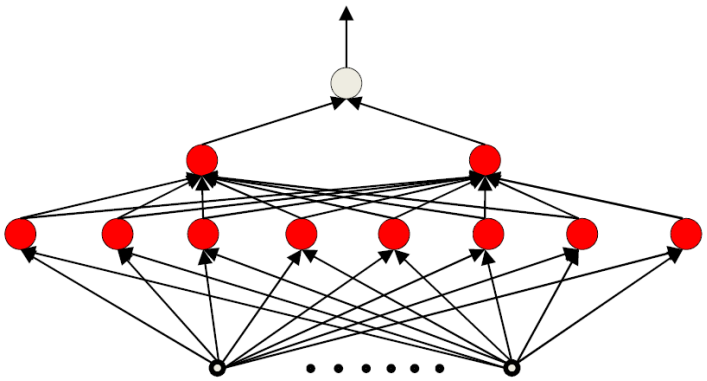
- RBF networks are more **effective approximators** of continuous-valued functions
  - A one-hidden-layer net only requires one unit per "cylinder"

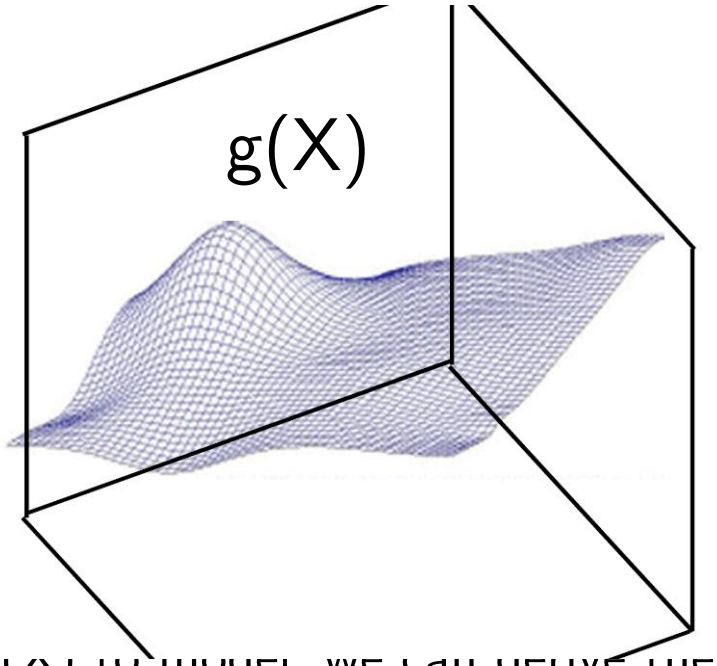




The MLP can represent **anything;** But how do we **construct** it?

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





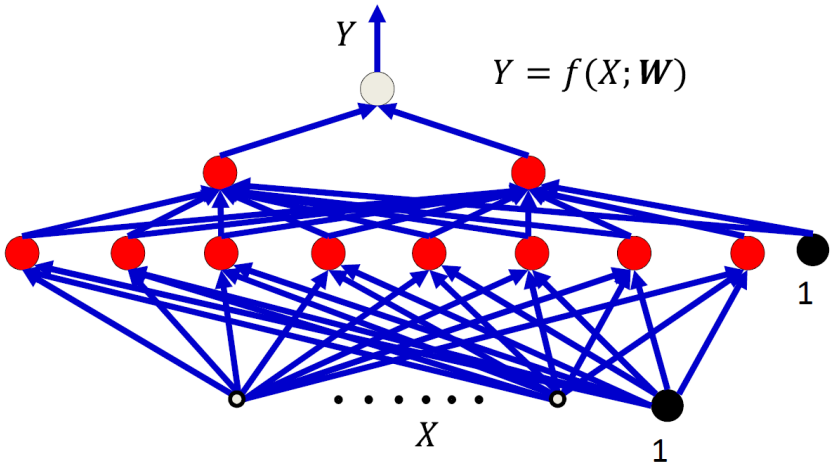
- More generally, given the function g(X) to model, we can derive the parameters of the network to model it, through computation
- When Y=f(X;**W**) has the capacity to exactly represent g(X)

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

- div() is a divergence function that goes to zero when g(X) =f(X;**W**)

Learning

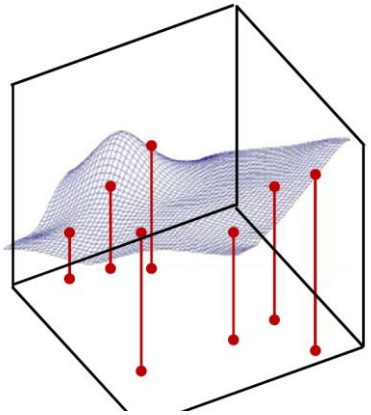
- Preliminaries:
  - How do we represent the input?
  - How do we represent the output?
- How do we compose the network that performs the requisite function?
- **What we learn: The parameters of the network**

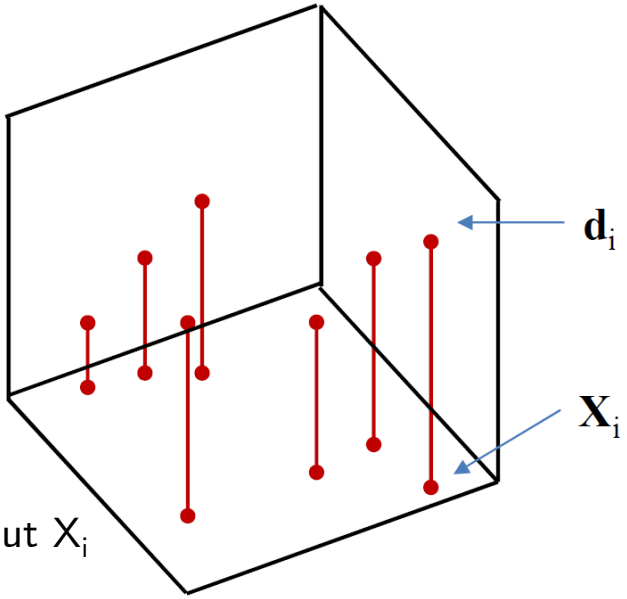


- Given: the architecture of the network
- The parameters of the network: The weights and biases
  - The weights associated with the blue arrows in the picture
- Learning the network : Determining the values of these parameters such that the network computes the desired function

Sampling the function

- g(X) is unknown and we draw "input-output" training instances from the function and estimate network parameters to "fit" the input-output relation at these instances

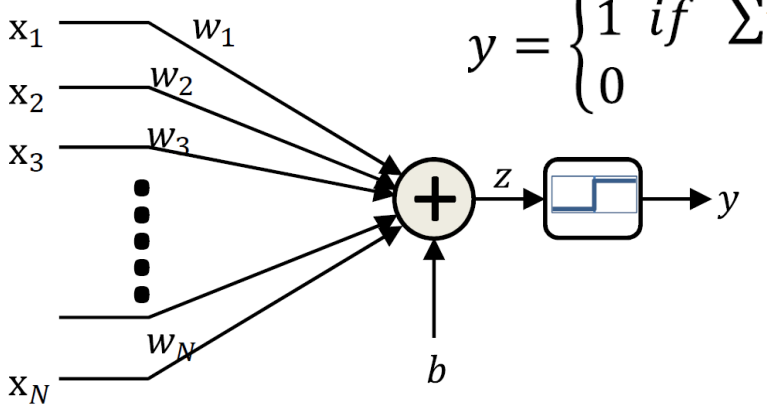




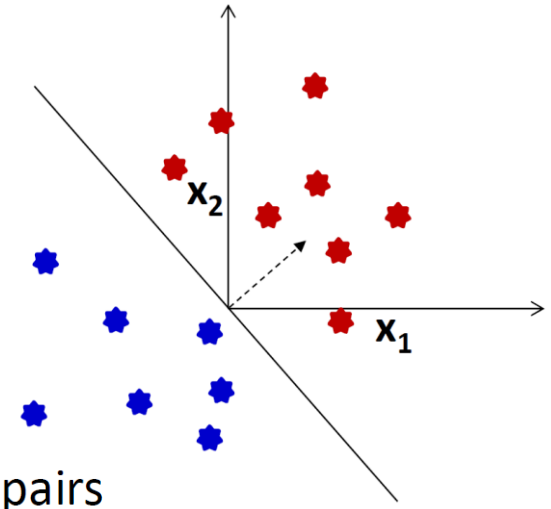
- Sample g(X)
  - Basically, get input-output pairs for a number of samples of input X_i

    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) (distribution of X)
- We must learn the entire function from these few examples
- The "training" samples Estimate the network parameters to "fit" the training points exactly
  - Assuming network architecture is sufficient for such a fit
  - Assuming unique output d at any X

Learning the perceptron



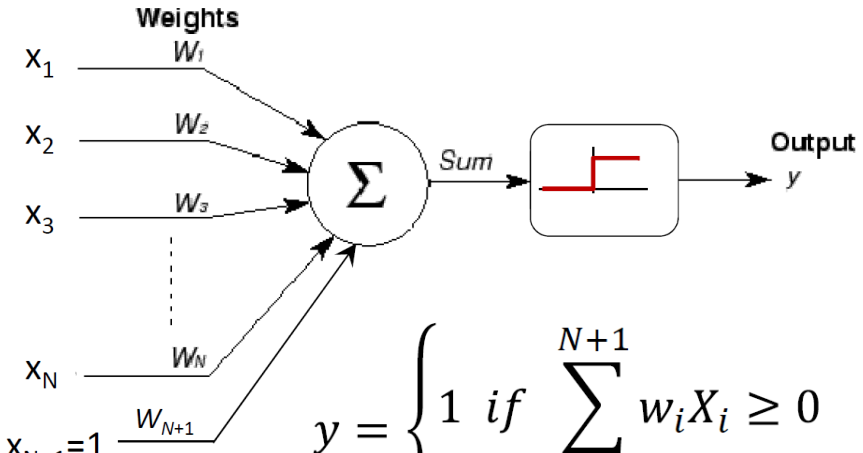
$$y = \begin{cases} 1 & if \ \sum_{i=1}^{N} w_i X_i + b \geq 0 \\ 0 & otherwise \end{cases}$$



Learn $W = [w_1 .. w_N]$ and b, given several (X, y) pairs

Restating the perceptron equation by adding another dimension to X ($x_{n+1} = 1$)

$\sum_{i=1}^{N+1} w_i X_i \geq 0$ is now a hyperplane through origin



Recap from lecture 12:

Criterion function is the **number of samples misclassified** by **W**.

$$y = \begin{cases} 1 & if \ \sum_{i=1}^{N+1} w_i X_i \geq 0 \\ 0 & otherwise \end{cases}$$

A Simple Method: The Perceptron Algorithm

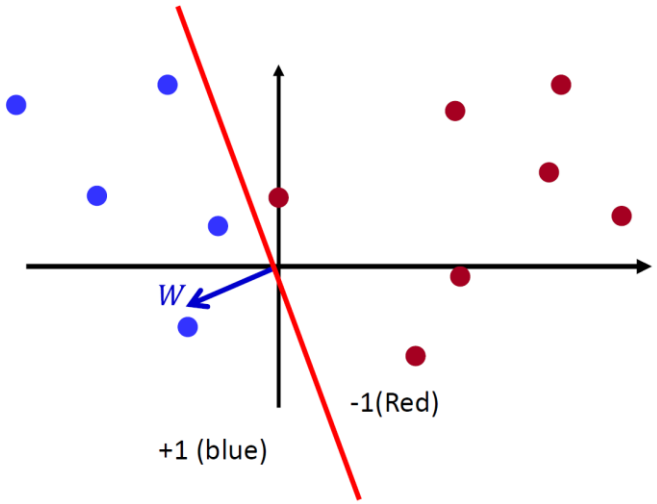
- Initialize: Randomly initialize the hyperplane (i.e. randomly initialize the normal vector **W**)
- Classification rule sign($\mathbf{W}^T\mathbf{x}$)
  - Vectors on the same side of the hyperplane as **W** will be assigned +1 class, and those on the other side will be assigned -1

- If instance misclassified
  - If instance is positive class

$$W = W + X_i$$

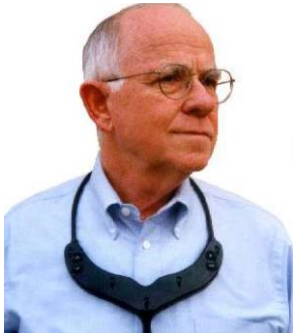
  - If instance is negative class

$$W = W - X_i$$



Perfect classification, no more updates

History: ADALINE (**Adaptive Linear Neuron**)



Bernard Widrow

- Scientist, Professor, Entrepreneur
- Inventor of most useful things in signal processing and machine learning!

- During learning, minimize **the squared error** assuming to be real output
- Error for a single input
- Online learning rule

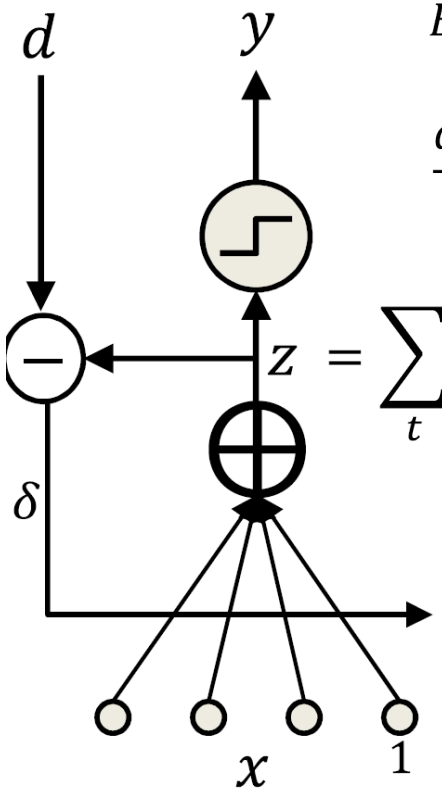
  After each input , that has target (binary) output , compute and update

$$\delta = d - z$$

$$w_i = w_i + \eta \delta x_i$$

- This is the famous delta rule
  - Also called the **LMS update** rule

$$out = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases}$$



$$Err(x) = \frac{1}{2}(d - z)^2$$

$$\frac{dErr(x)}{dw_i} = -(d - z)x_i$$

$$w_i = w_i + \eta(d - z)x_i$$

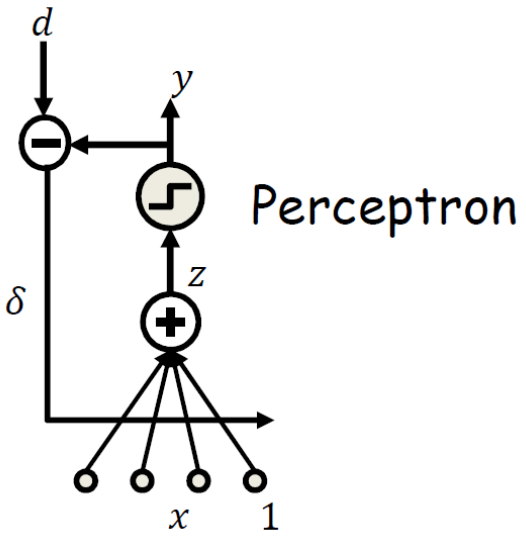
Generalized delta rule

- For any differentiable activation function the following update rule is used

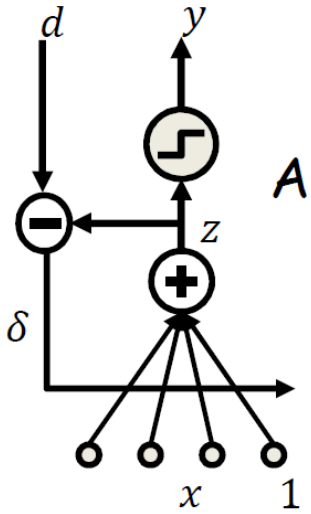
$$\delta = d - y$$

$$w_i = w_i + \eta \delta f'(z) x_i$$

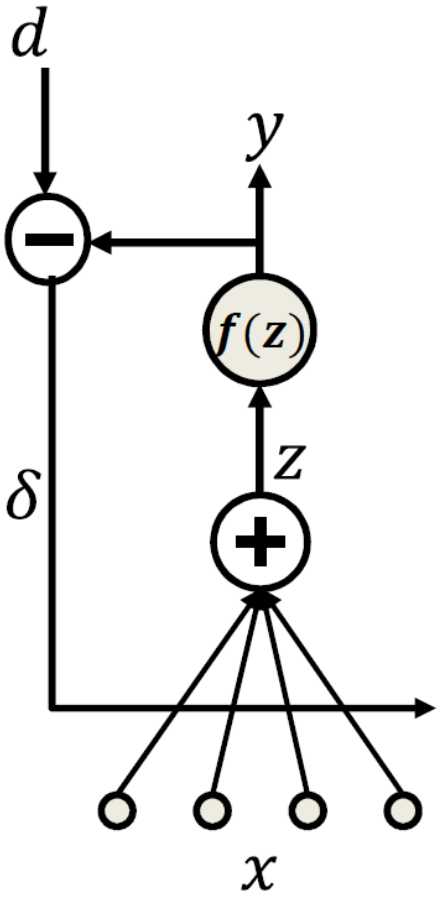
- This is exactly backpropagation in multilayer nets if we let f(z) represent the entire network between z and y



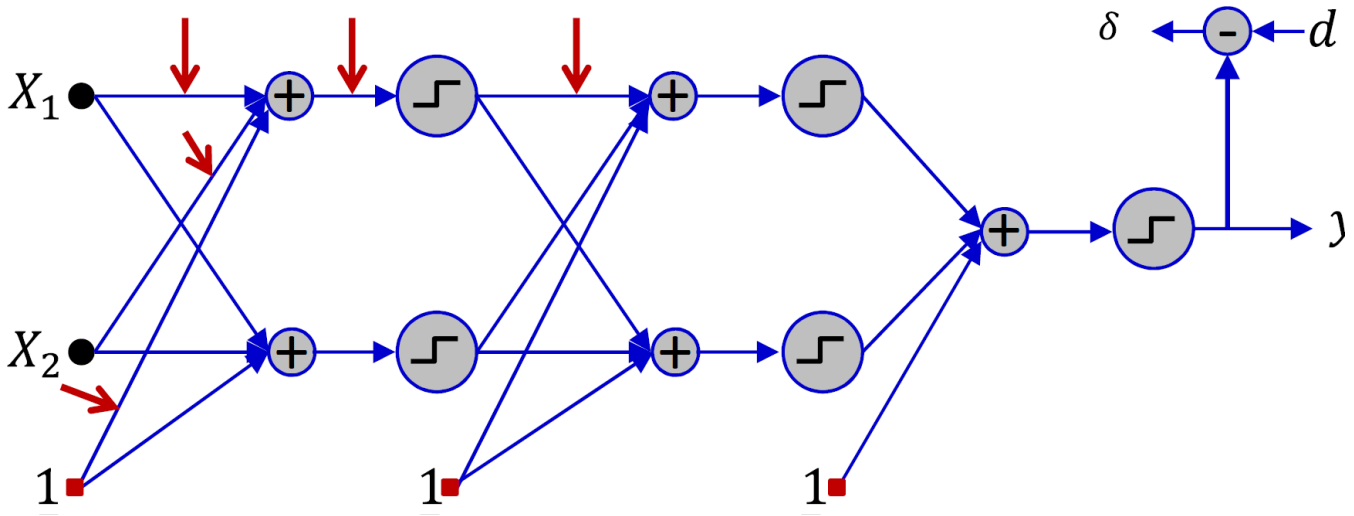
Perceptron



ADALINE



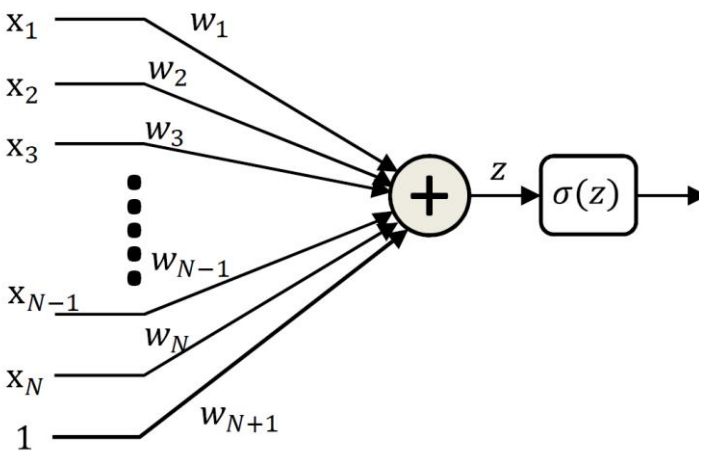
Multilayer perceptron: MADALINE; training

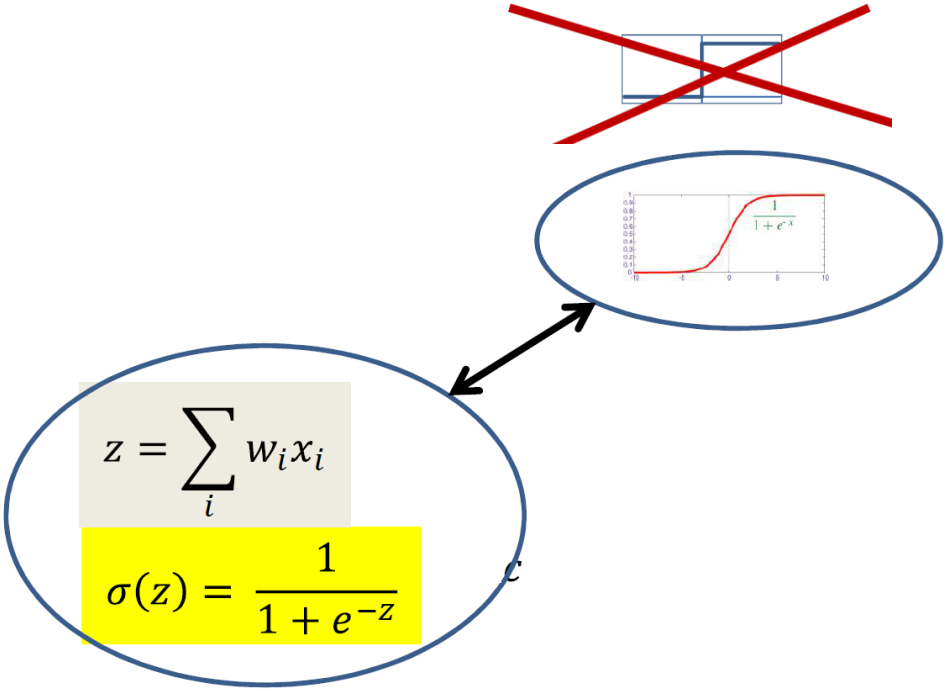


- Update only on error
- If error, find the z that is closest to 0
- Flip the output of corresponding unit and compute new output
- If error reduces:
  - Set the desired output of the unit to the flipped value
  - Apply ADALINE rule to update weights of the unit
- For threshold activations, this is an NP-complete combinatorial optimization problem

combinatorial optimization problem stalled development of neural networks for well over a decade!

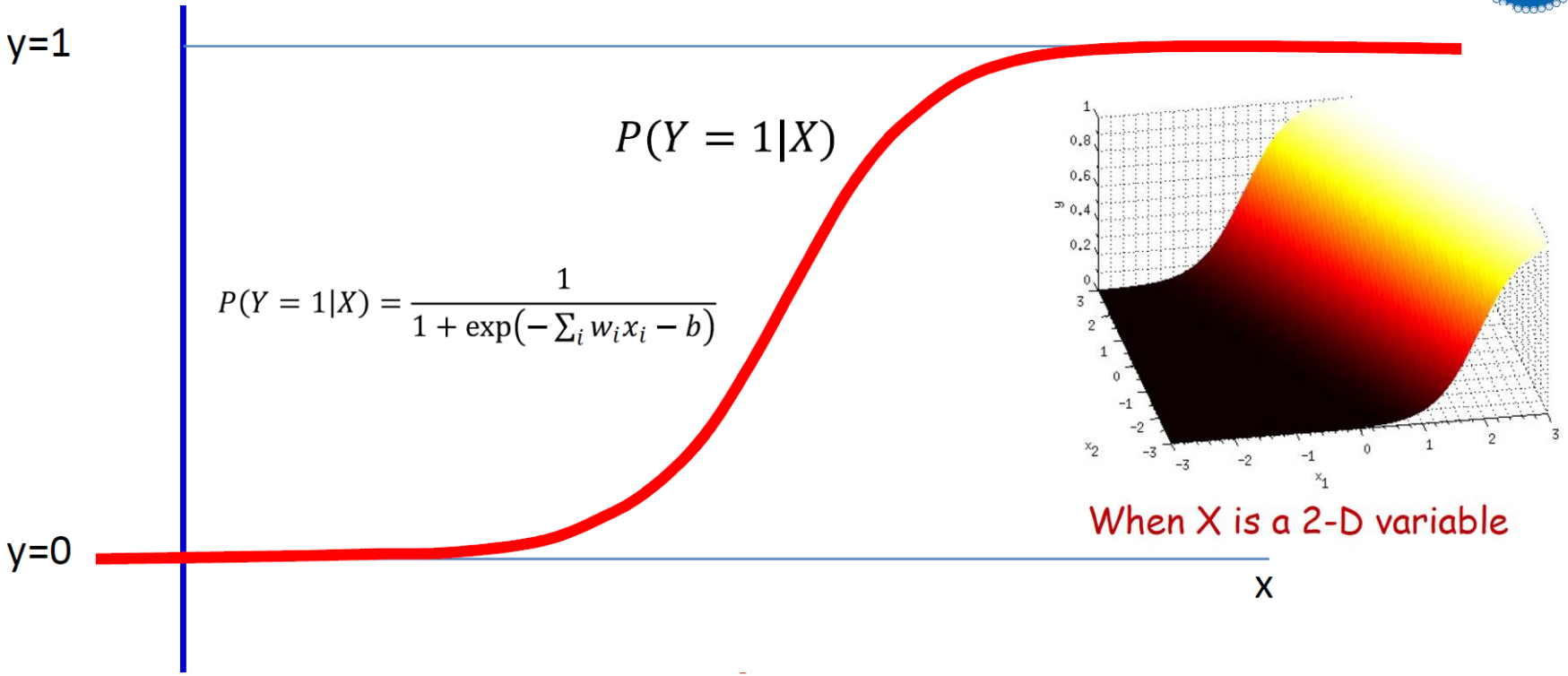
Lets make the neuron differentiable





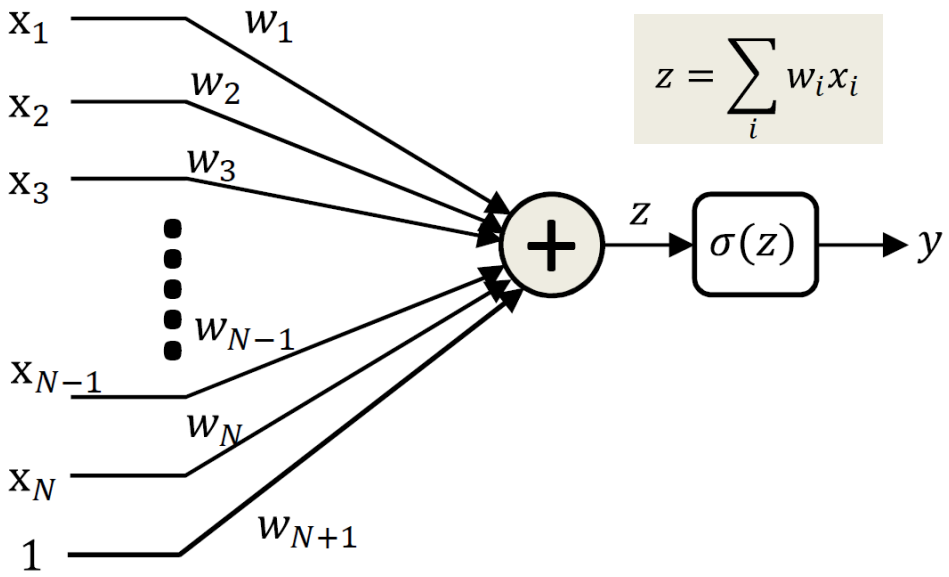
- The simple MLP is a flat, non-differentiable function
  - continuous activation functions with non-zero derivatives
- This enables us to estimate the parameters using gradient descent techniques
- This makes the output of the network differentiable w.r.t **every parameter** in the network

The probability of y=1



- Consider this differently: at each point look at a small window around that point
- **The logistic regression model**:
- Class 1 becomes increasingly probable going left to right
  - Very typical in many problems
- The perceptron with a sigmoid activation (It actually computes the probability that the input belongs to class 1)
- The logistic activation perceptron actually computes the a posteriori probability of the output given the input

Perceptrons with differentiable activation functions



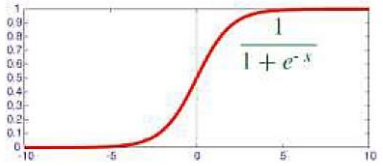
$$z = \sum_i w_i x_i$$

$$\frac{dy}{dw_i} = \frac{dy}{dz}\frac{dz}{dw_i} = \sigma'(z)x_i$$

$$\frac{dy}{dx_i} = \frac{dy}{dz}\frac{dz}{dx_i} = \sigma'(z)w_i$$

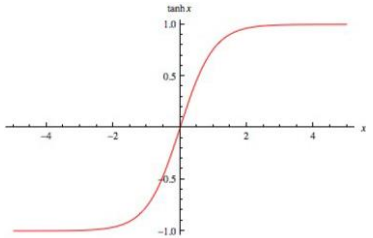
- $\sigma(z)$ is a differentiable function of z
- **Using the chain rule**, y is a differentiable function of both inputs x_i and weight w_i
- This means that we can compute the change in the output for small changes in either the input or the weights

Some popular activations and their derivatives



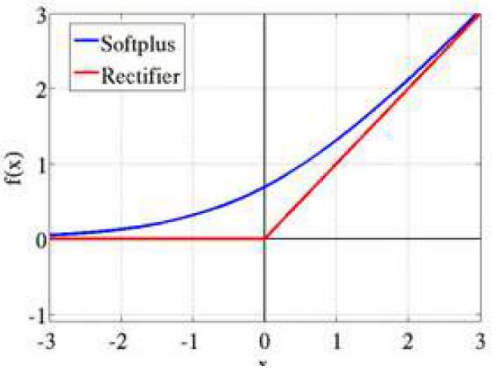
$$f(z) = \frac{1}{1 + \exp(-z)}$$

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



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

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



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

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

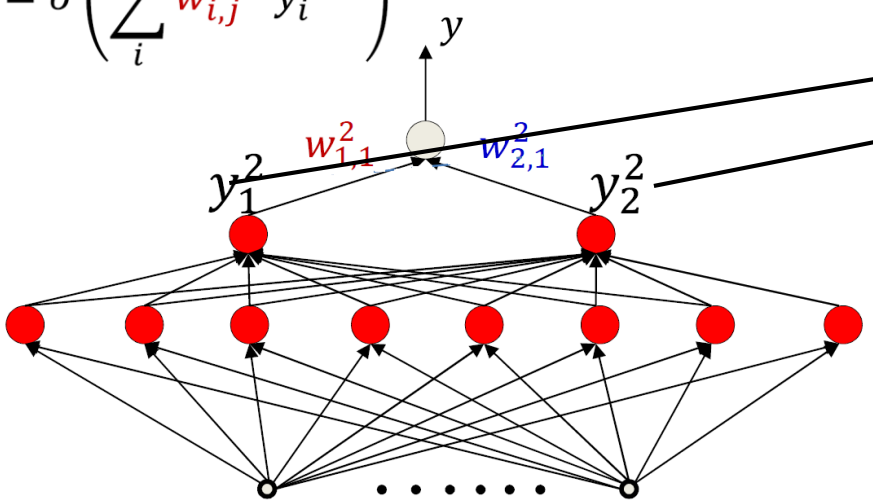
$$f(z) = \log(1 + \exp(z))$$

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

Overall network is differentiable

$$y_j^k = \sigma\left(\sum_i w_{i,j}^{k-1} y_i^{k-1}\right)$$

$$y = \sigma(w_{1,1}^2 y_1^2 + w_{2,1}^2 y_2^2 + w_{3,1}^2)$$



y = output of overall network

$w_{i,j}^k$ = weight connecting the ith unit of the kth layer to the jth unit of the k+1-th layer

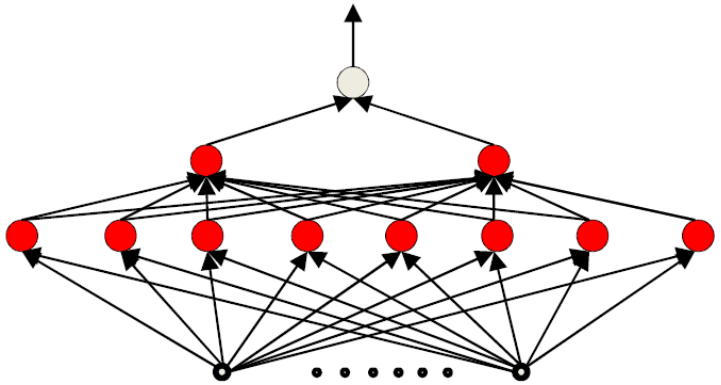
y_i^k = output of the ith unit of the kth layer

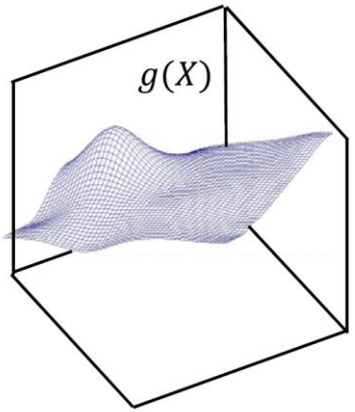
$\sigma()$ is differentiable w.r.t both w and y_i^k

- Every individual perceptron is differentiable w.r.t its inputs and its weights (including "bias" weight)
- By the chain rule, the overall function is differentiable w.r.t every parameter (weight or bias)
  - Small changes in the parameters result in measurable changes in output

Recap: Learning the function

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





- When f(X;**W**) has the capacity to exactly represent g(X)

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

- div() is a divergence function that goes to zero when f(X;**W**) = g(X)
- More generally, assuming is a random variable

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

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

Sampling the function and empirical risk

- Sample g(X)
- 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 will be drawn from P(X)
- Estimate function from the samples
- The **expected** error is the average error over the entire input space

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

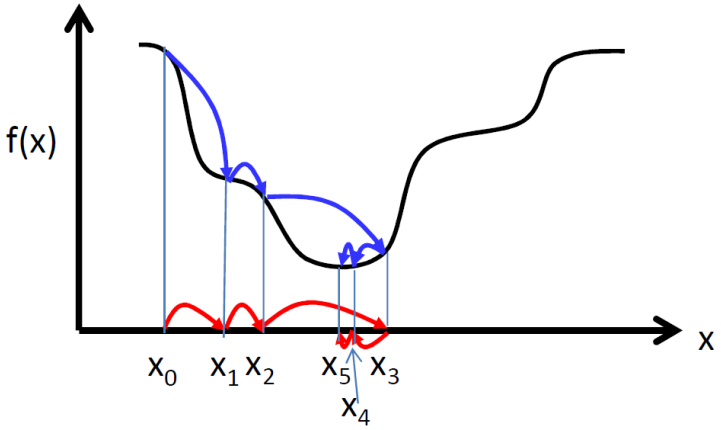
- The empirical estimate of the expected error is the average error over the samples

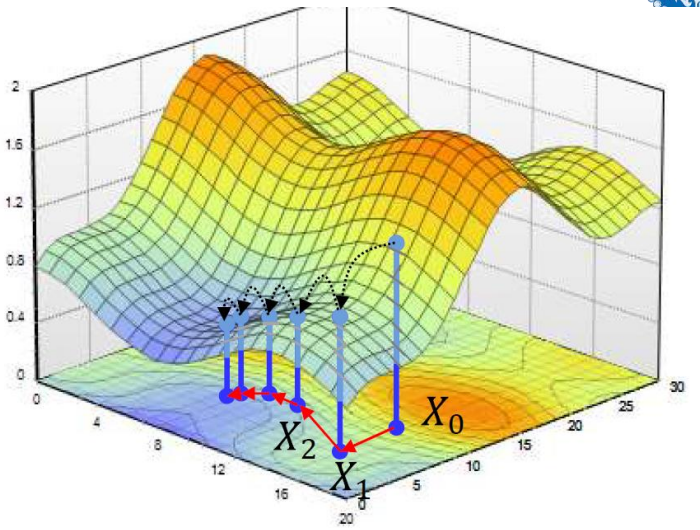
$$E\big[div\big(f(X;W),g(X)\big)\big] \approx \frac{1}{N}\sum_{i=1}^{N} div(f(X_i;W),d_i) = Err(W)$$

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

An instance of **optimization:**
- Gradient descent and other iterative solutions

The Approach of Gradient Descent





- Iterative solution: Trivial algorithm

  Initialize x^0

  While $\left\|\nabla_x f(x^k)\right\| > \varepsilon$ (or while $\left|f\left(x^{k+1}\right) - f\left(x^k\right)\right| > \varepsilon$)

  $$x^{k+1} = x^k - \eta^k \nabla_x f(x^k)$$

  η^k is the "step size"

Problem Setup: Things to define

- Given a training set of input-output pairs (X_1,d_1), (X_2,d_2), (X_3,d_3), …, (X_T,d_T),

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

- What is f() and what are its parameters W?
  - Typical network; Multi-layer perceptron
    - A differentiable activation function applied to an affine combination of the input

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

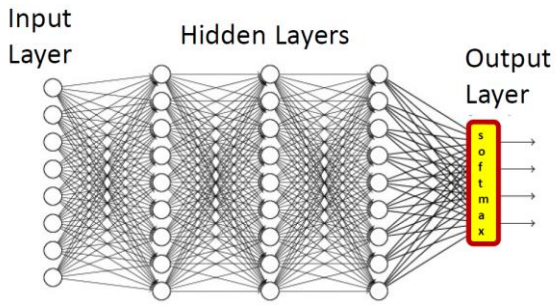
$$[y_1, y_2, \ldots, y_l] = f(x_1, x_2, \ldots, x_k; W)$$

- Function operates on set of inputs to produce set of outputs (vector activation)

  – Modifying a single parameter in will affect all outputs

Representing the output; One-hot representations

- 5-class example: 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:
  - 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: $[0\ 0\ 0\ 0\ 1]^T$
- Softmax vector activation is often used at the output of multi-class classifier nets

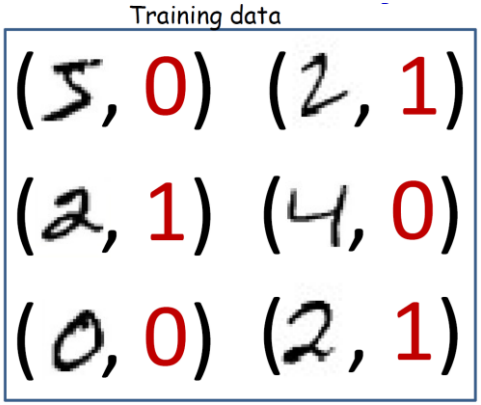


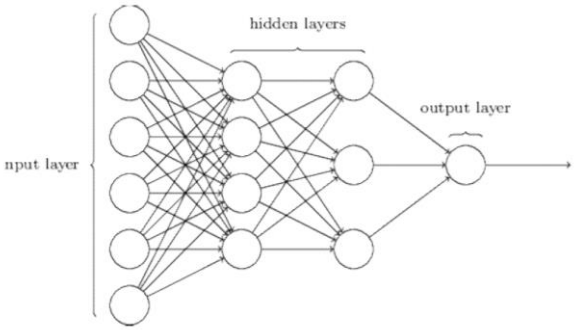
$$z_i = \sum_j w_{ji}^{(n)} y_j^{(n-1)}$$

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

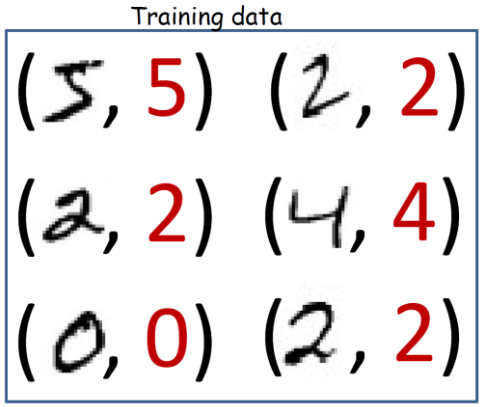
- This can be viewed as the probability $y_i = P(class = i|X)$

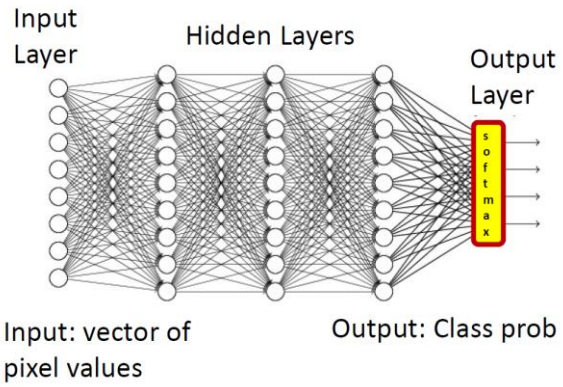
binary classification





multiclass classification





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

Examples of divergence functions

- For real-valued output vectors, the (scaled) L_2 divergence is popular
- Squared Euclidean distance between true and desired output

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

  - Note: this is differentiable

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

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

Training Neural Nets through Gradient Descent

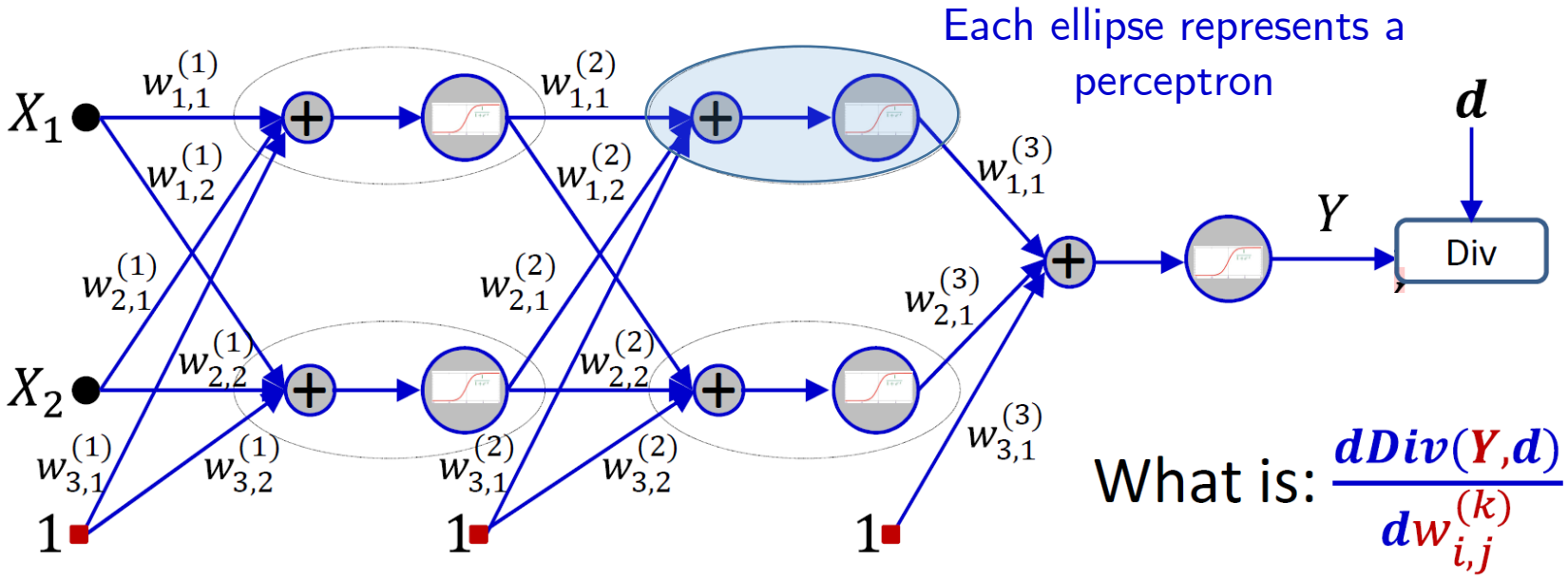
- Total training error: $$Err = \frac{1}{T}\sum_{t} Div(\boldsymbol{Y}_t, \boldsymbol{d}_t)$$
- Gradient descent algorithm:
  - Initialize all weights and biases $\{w_{ij}^{(k)}\}$ (Assuming the bias is also represented as a weight)
  - Do:
    - For every layer k for all I, j, update

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

How to compute

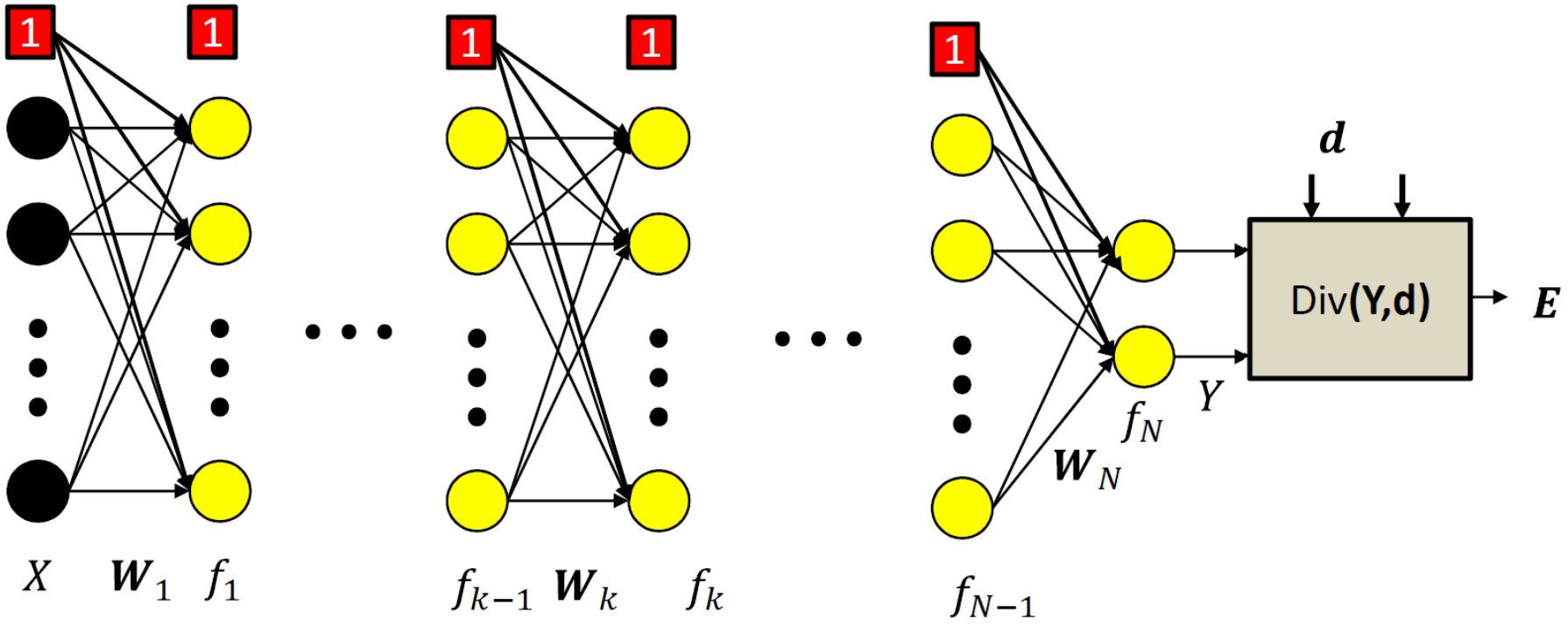
  - Until has converged

A closer look at the network

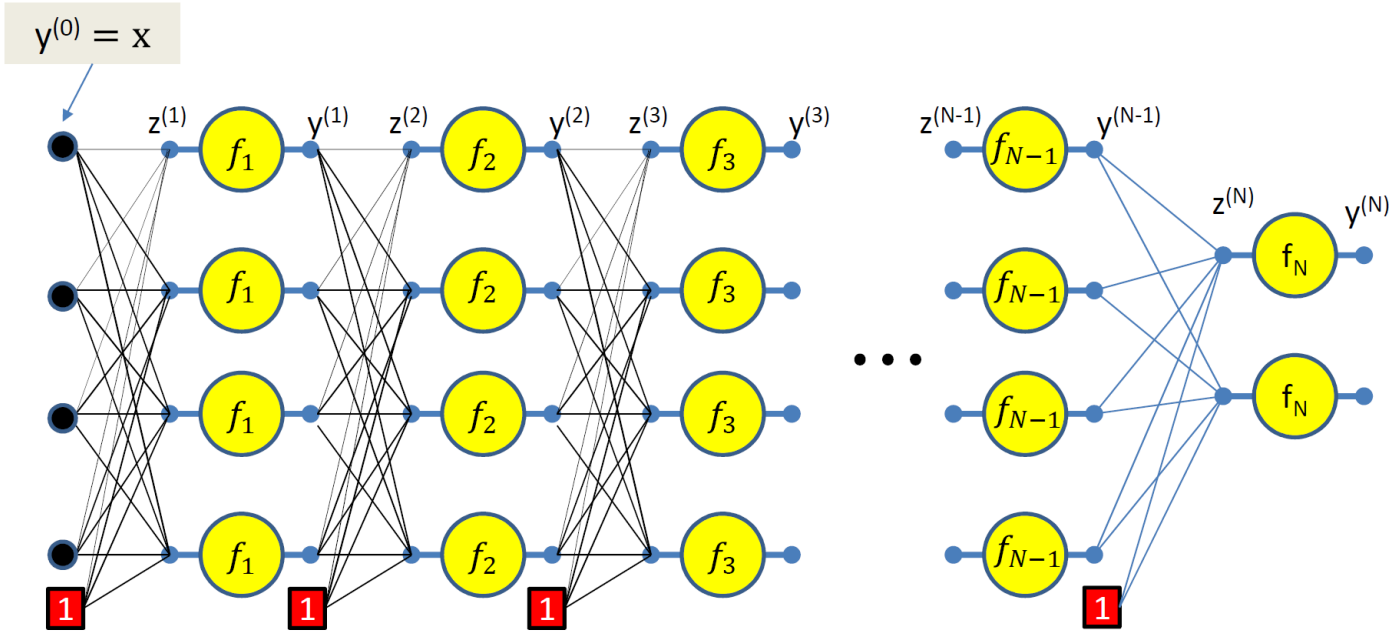


- Expanded with all weights and activations shown
- The overall function is differentiable w.r.t every weight, bias and input
- Computation of the derivative requires intermediate and final output values of the network in response to the input

The network again: modular view

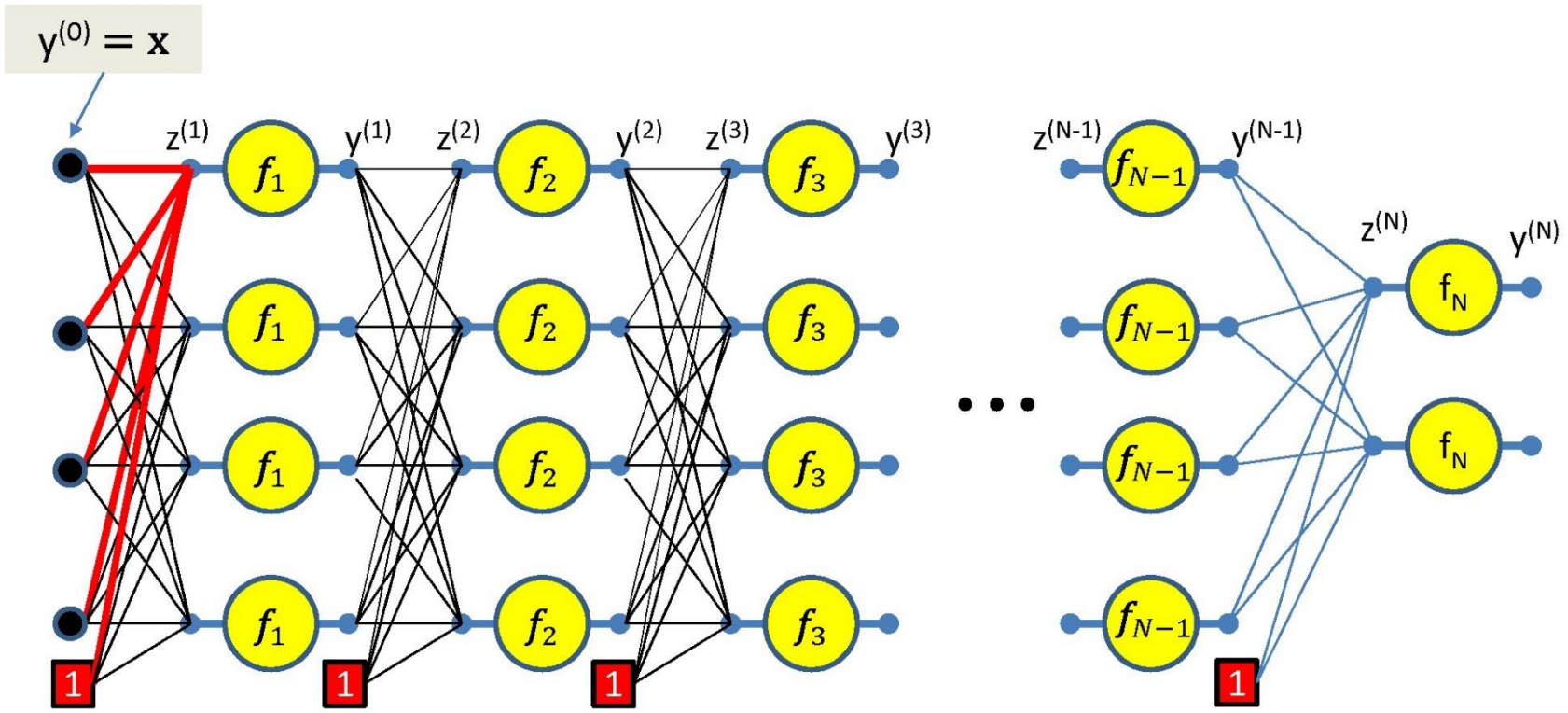


Expanding it out



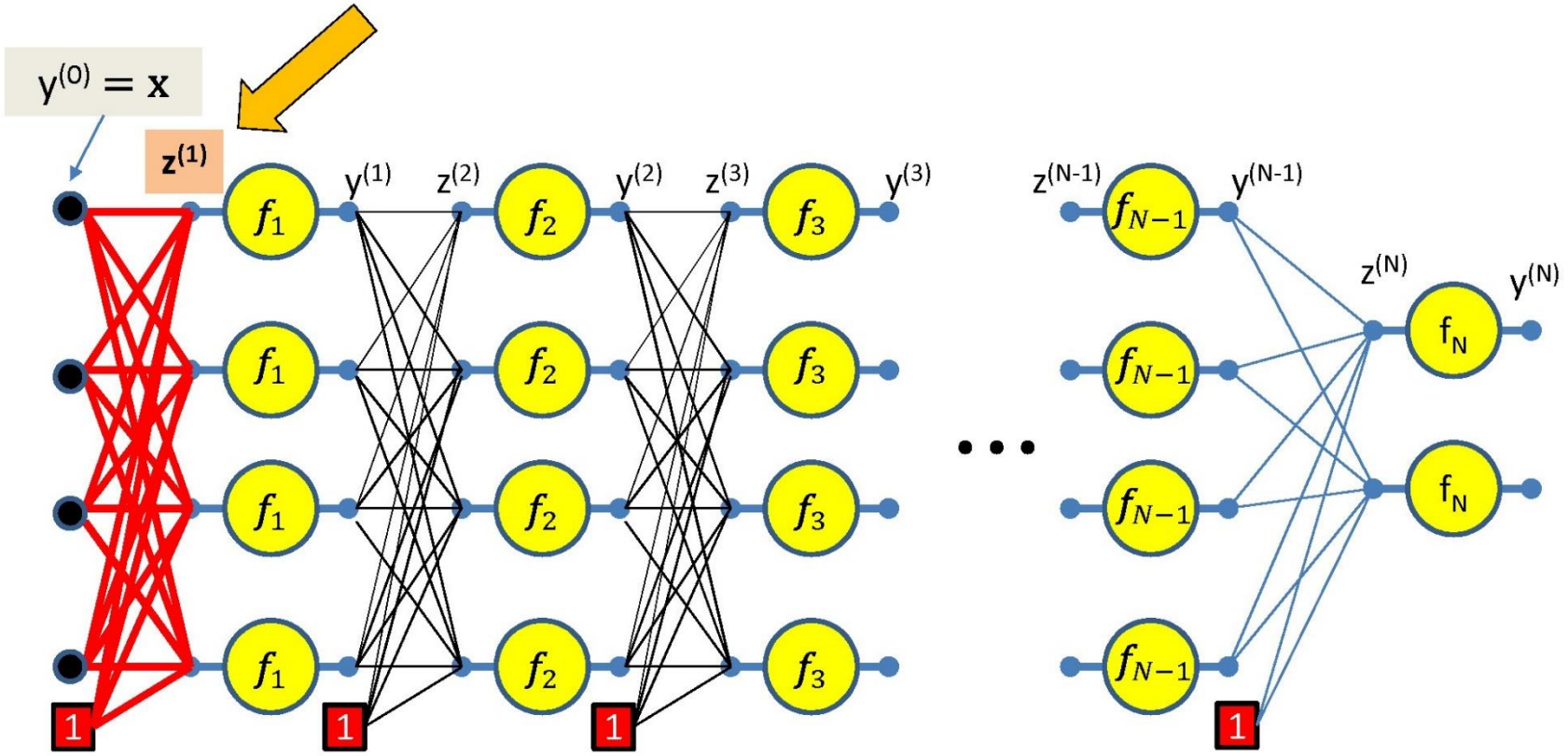
- Setting $y_i^{(0)} = x_i$ for notational convenience
- Assuming $w_{0j}^{(k)} = b_j^{(k)}$ and $y_0^{(k)} = 1$
- Assuming the **bias is a weight** and extending the output of every layer by a constant 1, to account for the biases

Forward Computation



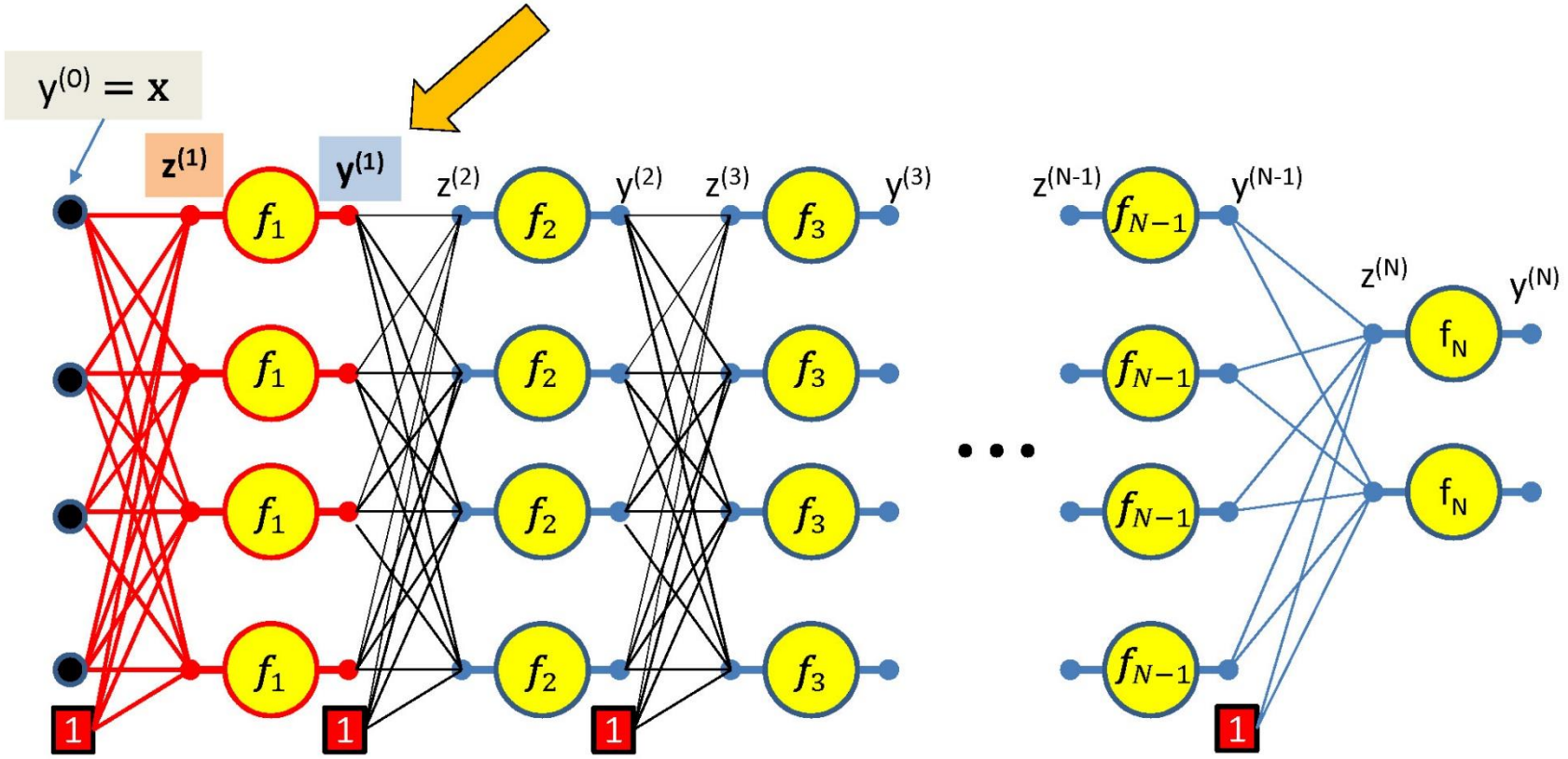
$$z_1^{(1)} = \sum_i w_{i1}^{(1)} y_i^{(0)}$$

Forward Computation



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

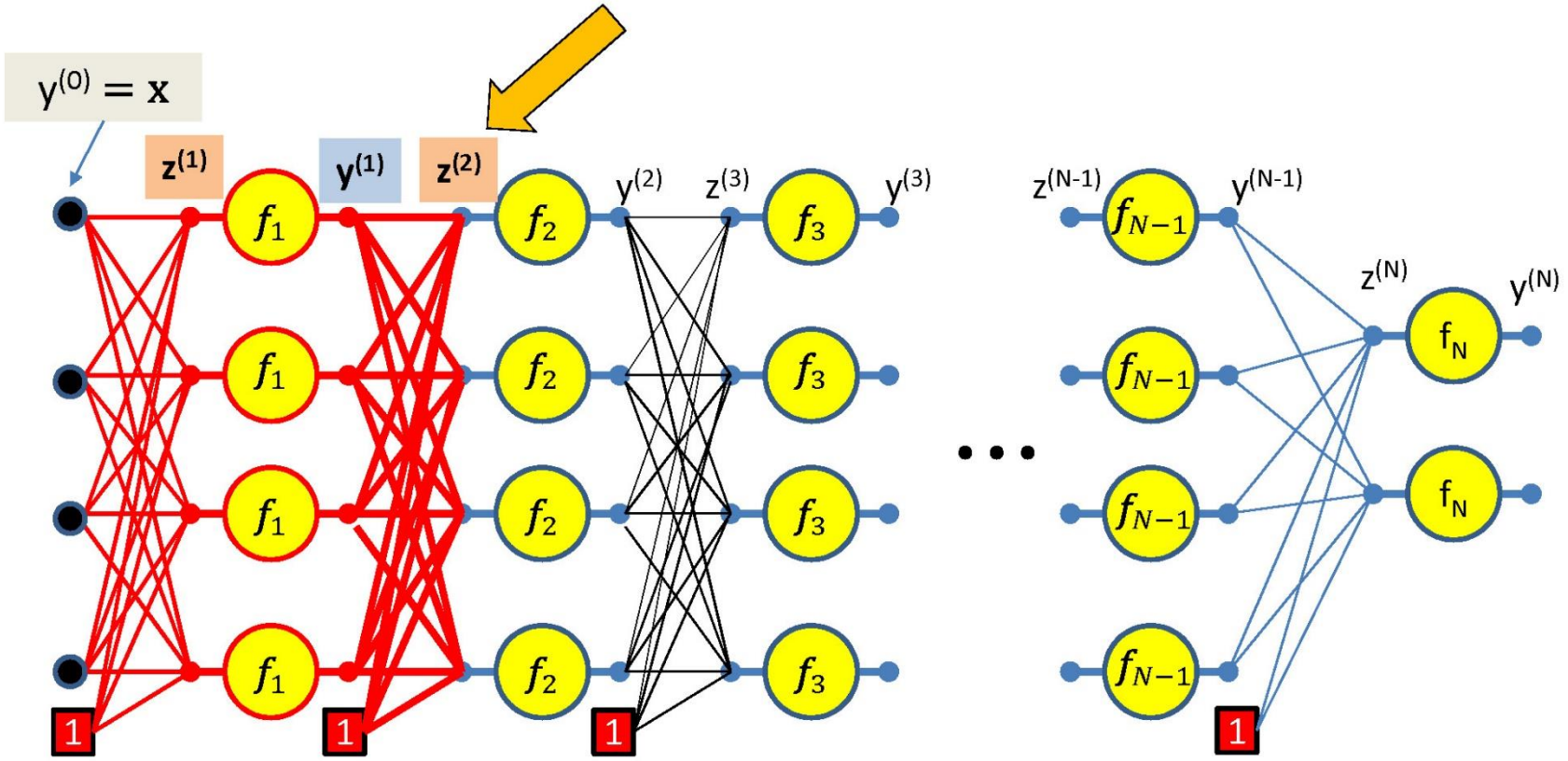
Forward Computation



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

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

Forward Computation

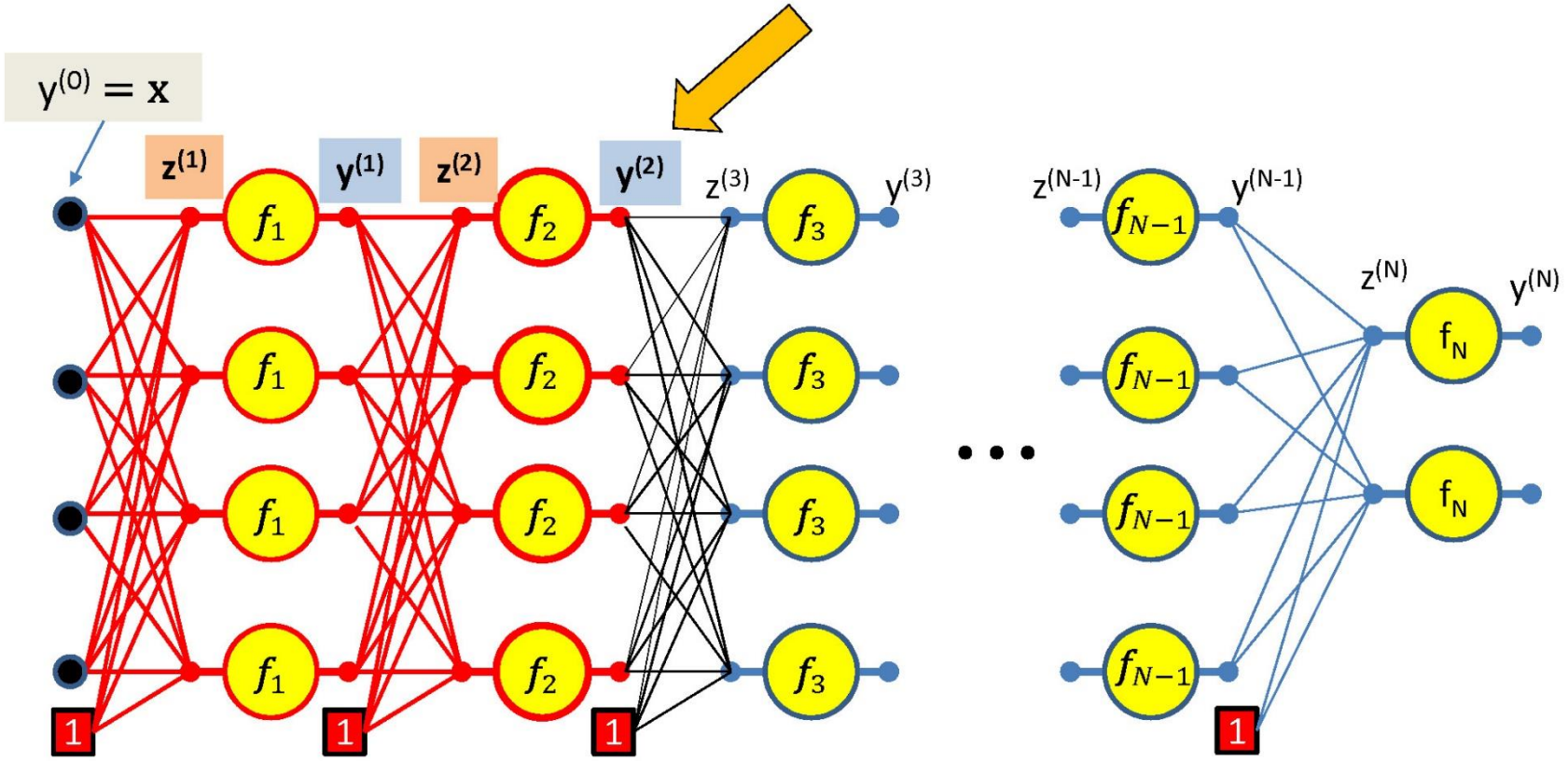


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

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

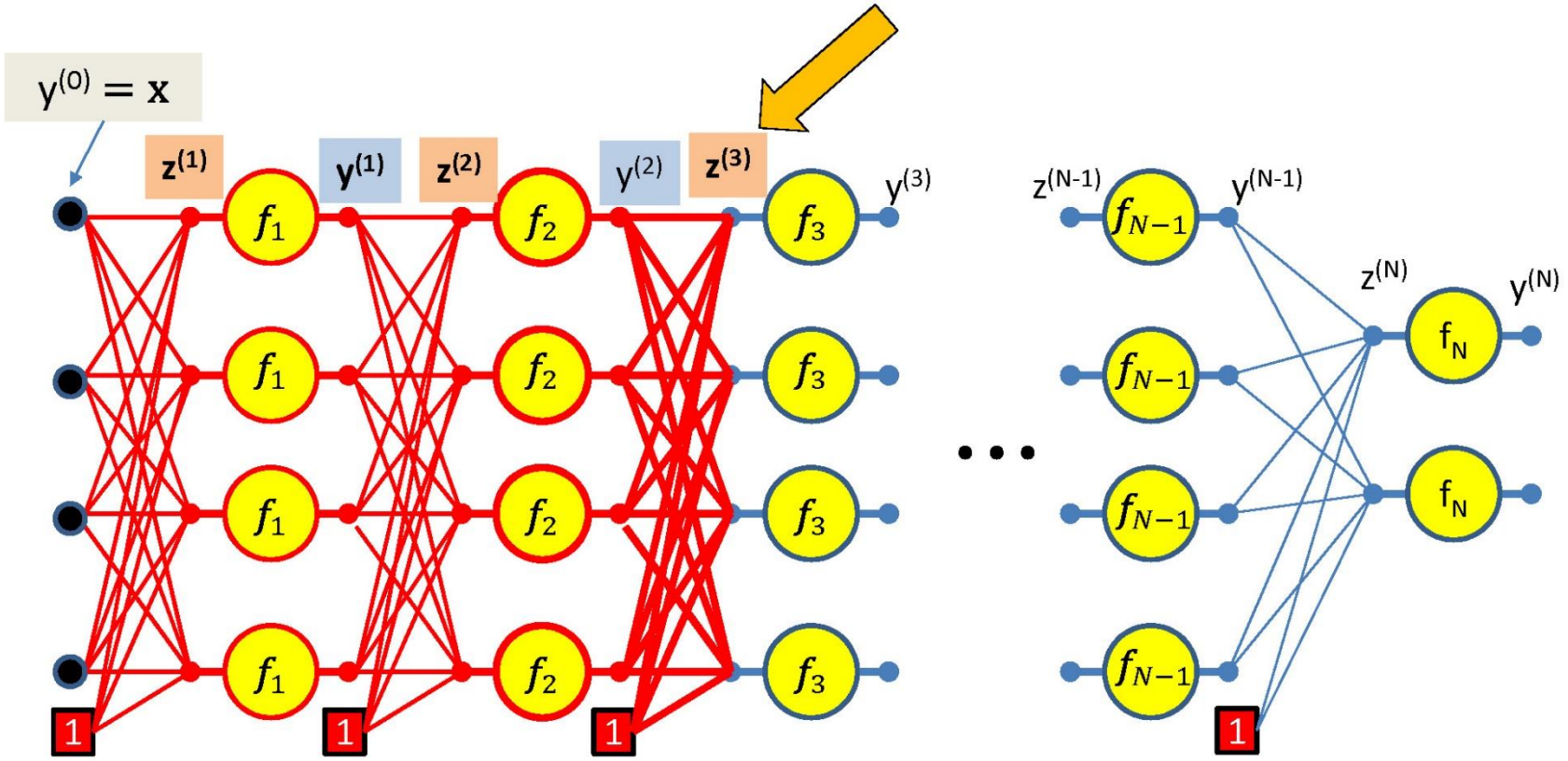
$$z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)}$$

Forward Computation



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

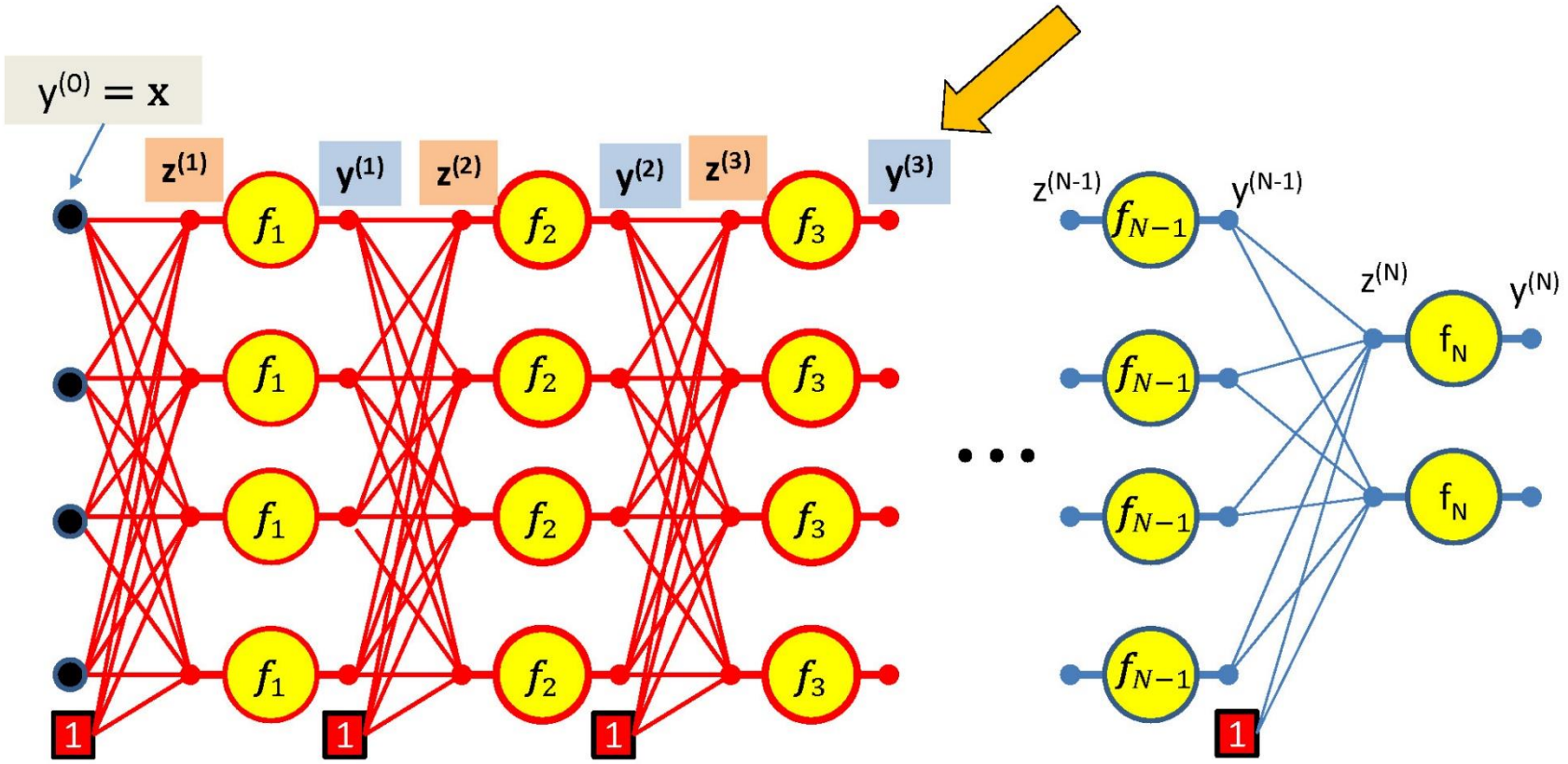
Forward Computation



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

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

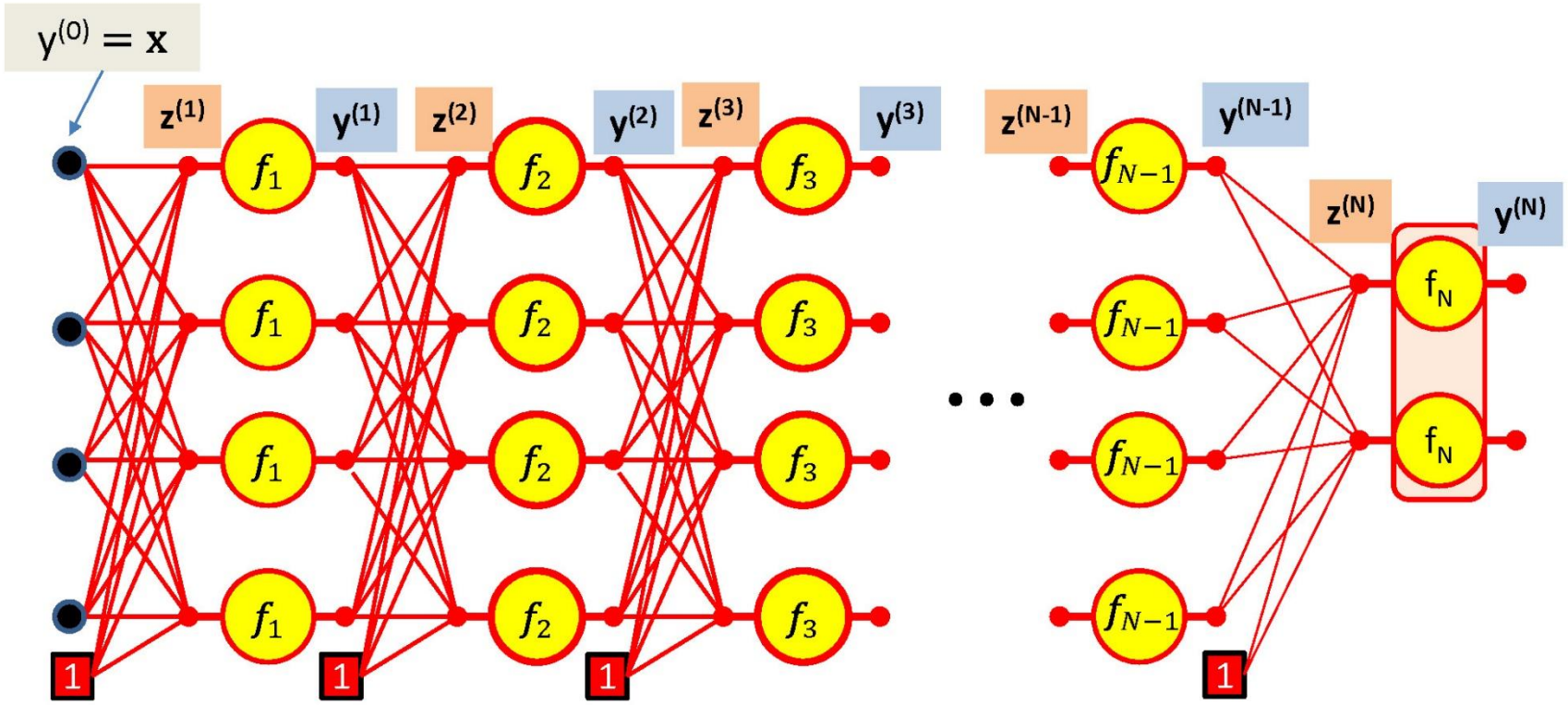
Forward Computation



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

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

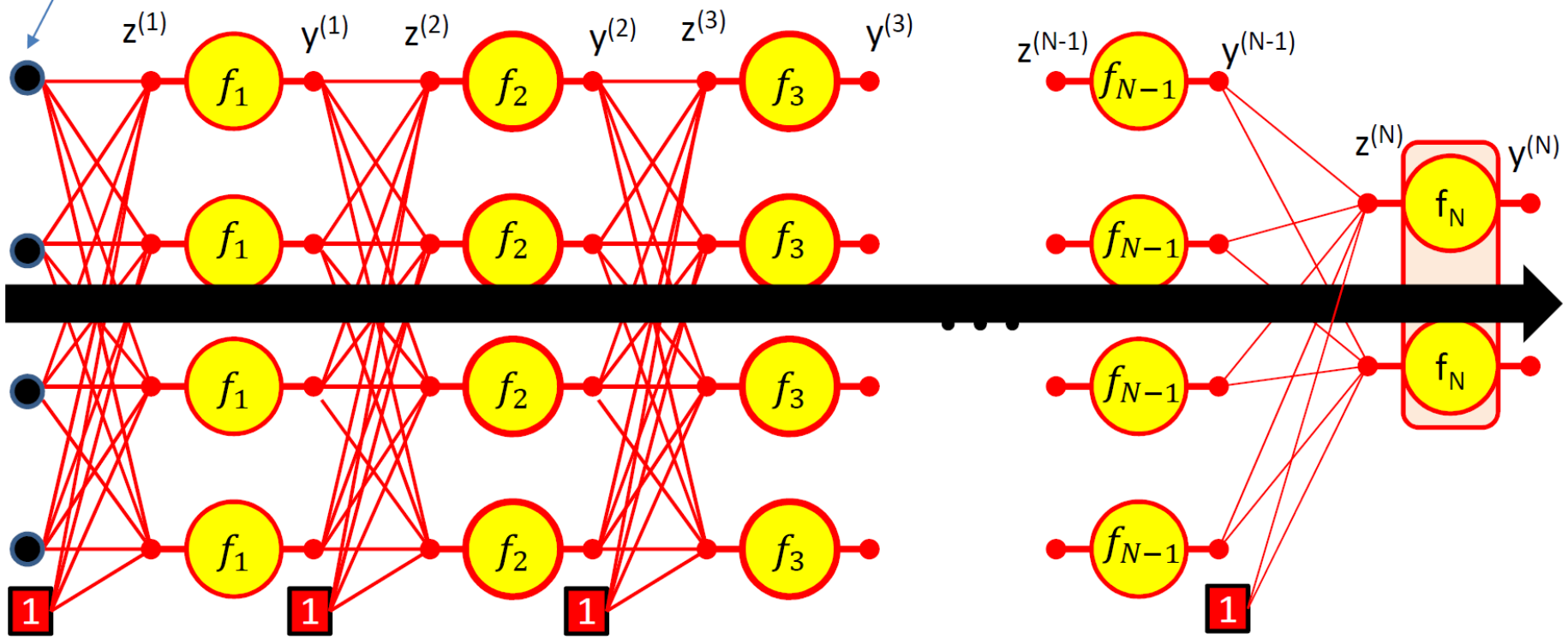
Forward Computation



$$y_j^{(N-1)} = f_{N-1}\left(z_j^{(N-1)}\right) \qquad z_j^{(N)} = \sum_i w_{ij}^{(N)} y_i^{(N-1)} \qquad \boldsymbol{y}^{(N)} = f_N\left(\boldsymbol{z}^{(N)}\right)$$

Forward Computation

$y^{(0)} = x$ $\quad y_i^{(0)} = x_i$



- Iterate for k = 1:N
  - for j = 1:layer-width

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

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

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
- $y_j^{(0)} = x_j,\ j = 1 \dots D;\qquad y_0^{(k=1\dots N)} = x_0 = 1$

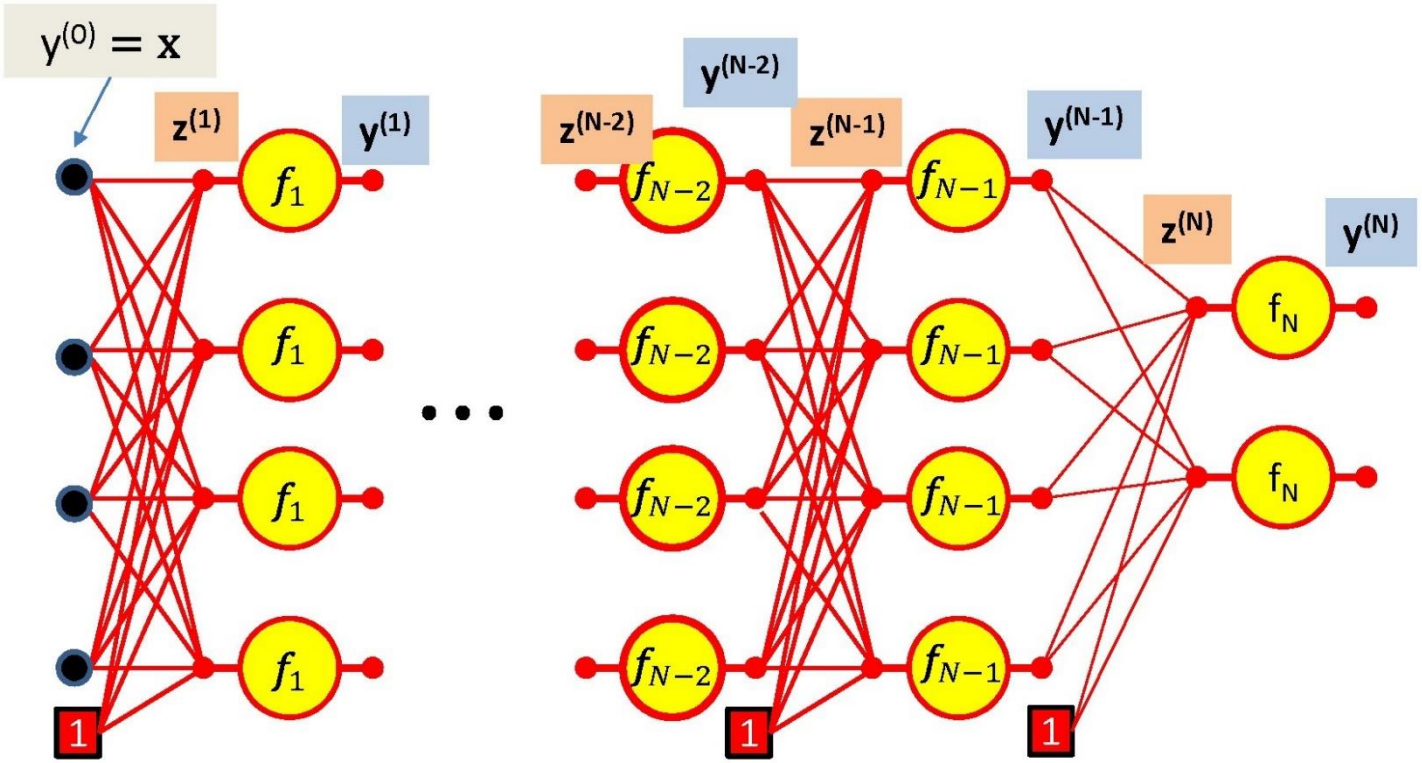
For layer $k = 1 \dots N$

- For $j = 1 \dots D_k$ D_k is the size of the kth layer
  - $z_j^{(k)} = \sum_{i=0}^{D_{k-1}} w_{i,j}^{(k)} y_i^{(k-1)}$
  - $y_j^{(k)} = f_k\left(z_j^{(k)}\right)$

Output:

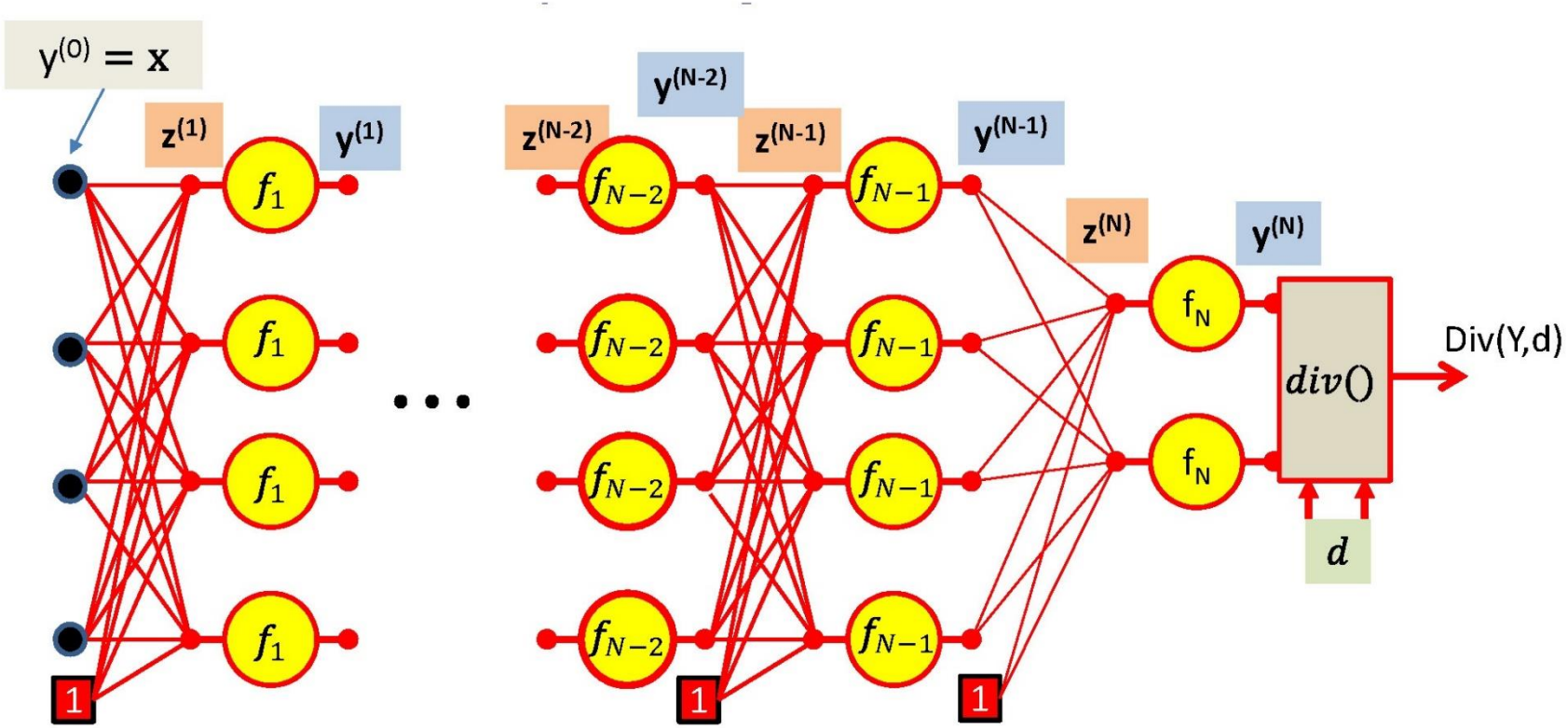
- $Y = y_j^{(N)}, j = 1..D_N$

Computing derivatives



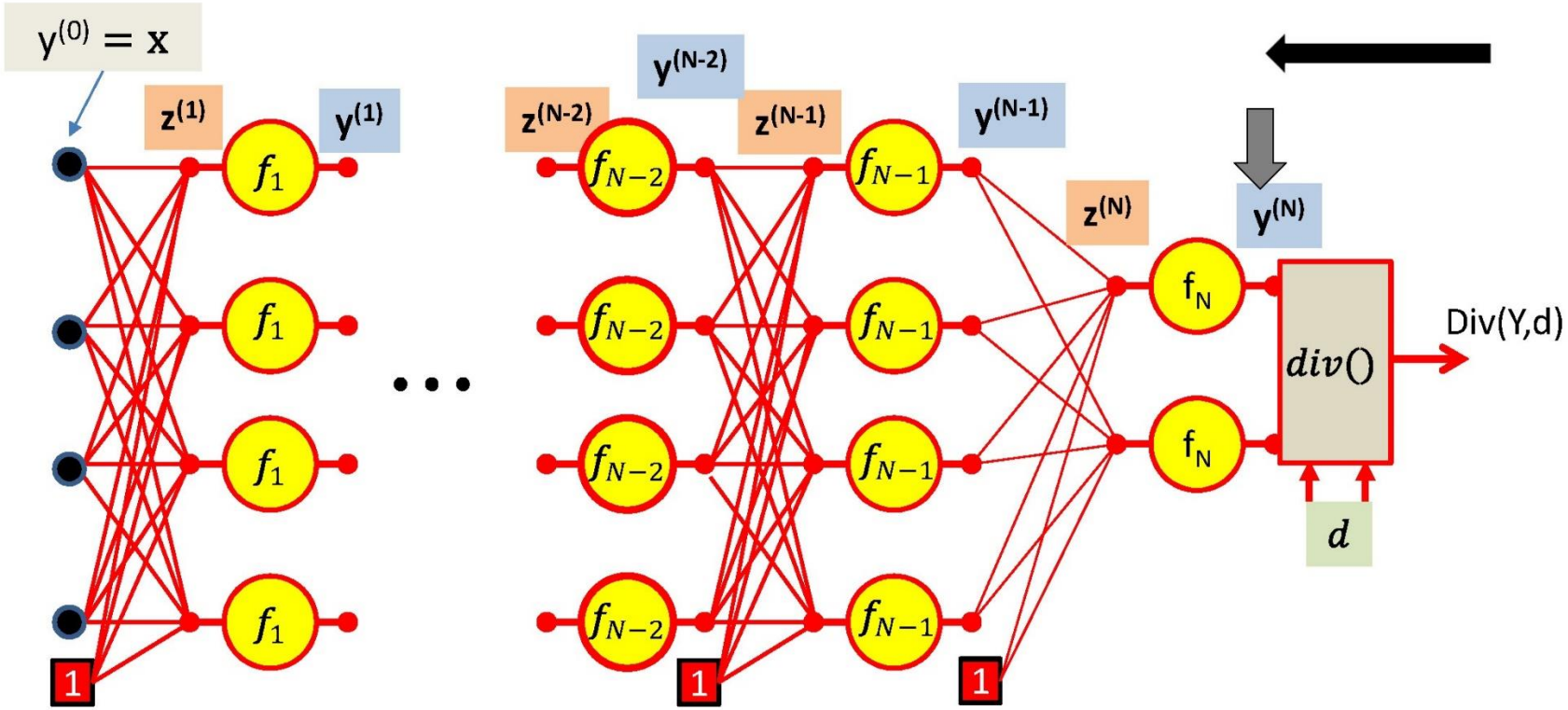
- We have computed all these intermediate values in the forward computation
- We must remember them – we will need them to compute the derivatives

Computing derivatives



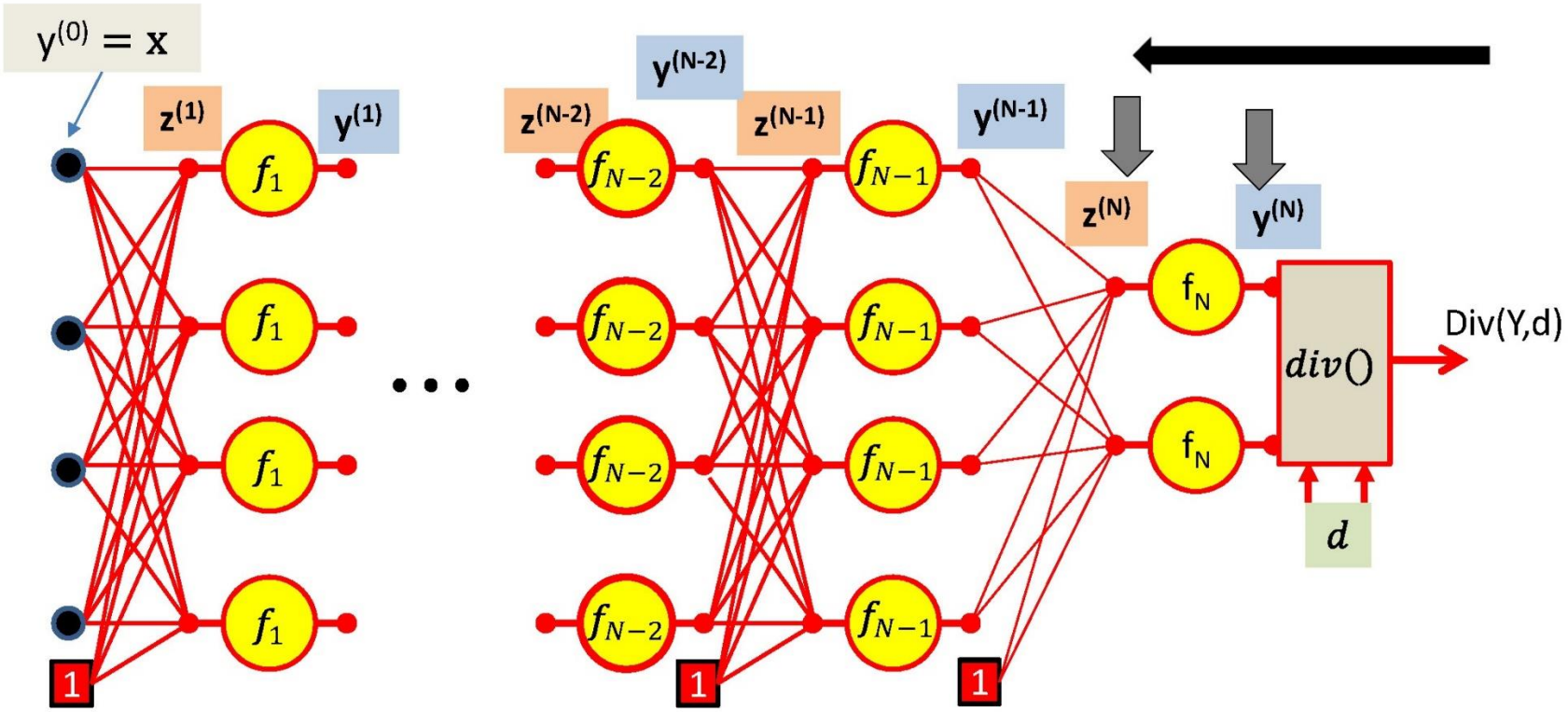
- First, we compute the divergence between the output of the net $y = y^{(N)}$ and the desired output d

Computing derivatives



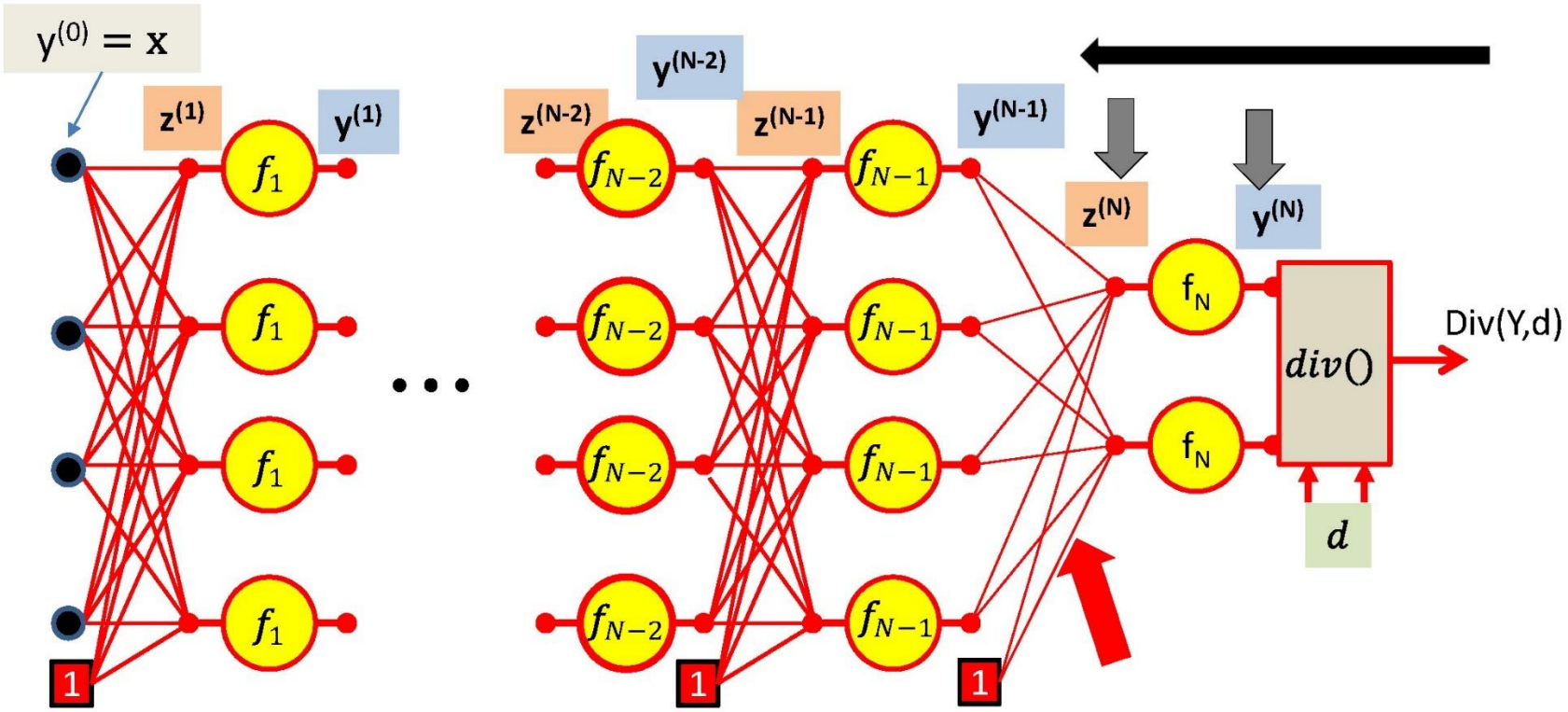
- We then compute $\nabla_{Y(N)}\text{div}(.)$ the derivative of the divergence w.r.t. the final output of the network $y^{(N)}$

Computing derivatives



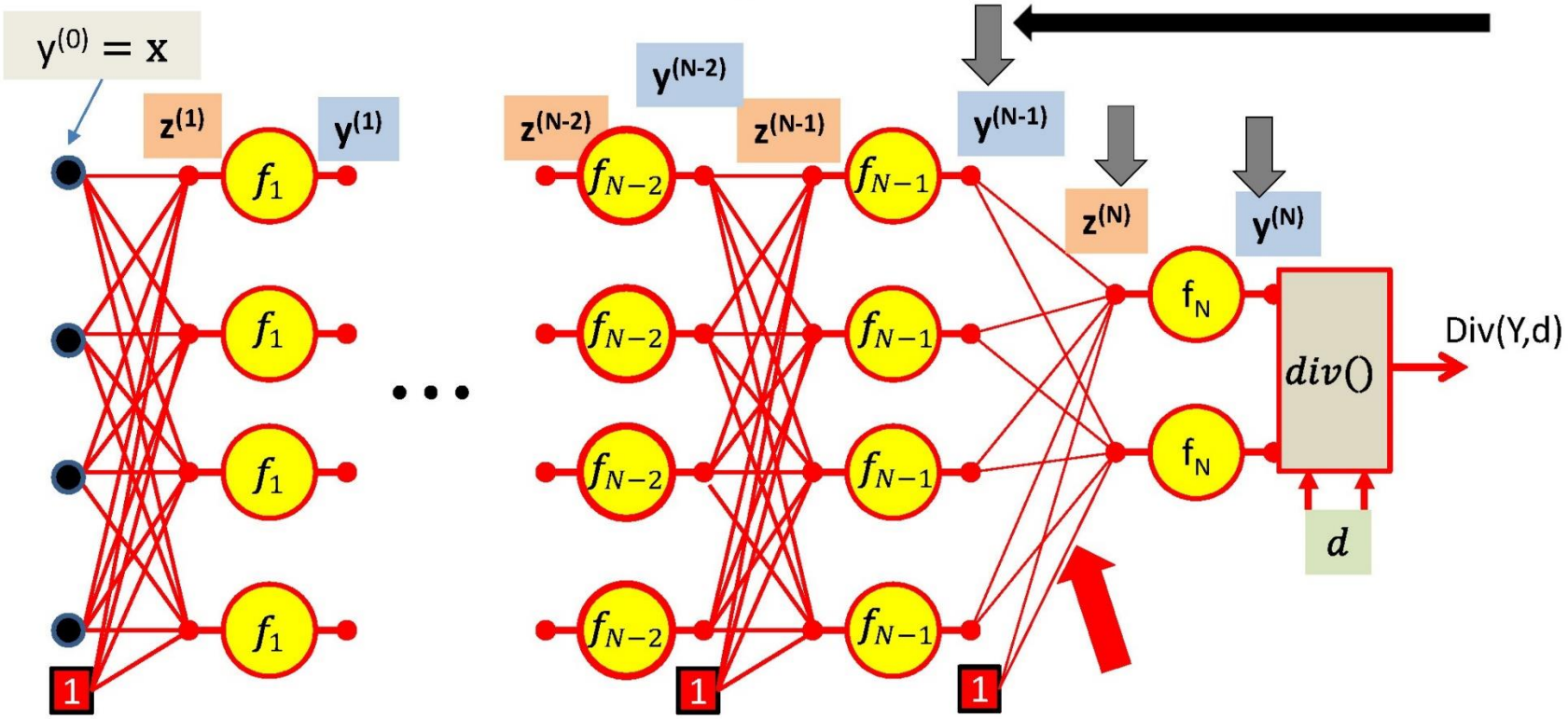
- We then compute $\nabla_{Y(N)}$div(.) the derivative of the divergence w.r.t. the final output of the network y(N)
- We then compute $\nabla_{Z(N)}$div(.) the derivative of the divergence w.r.t. the pre-activation affine combination z(N) using the chain rule

Computing derivatives



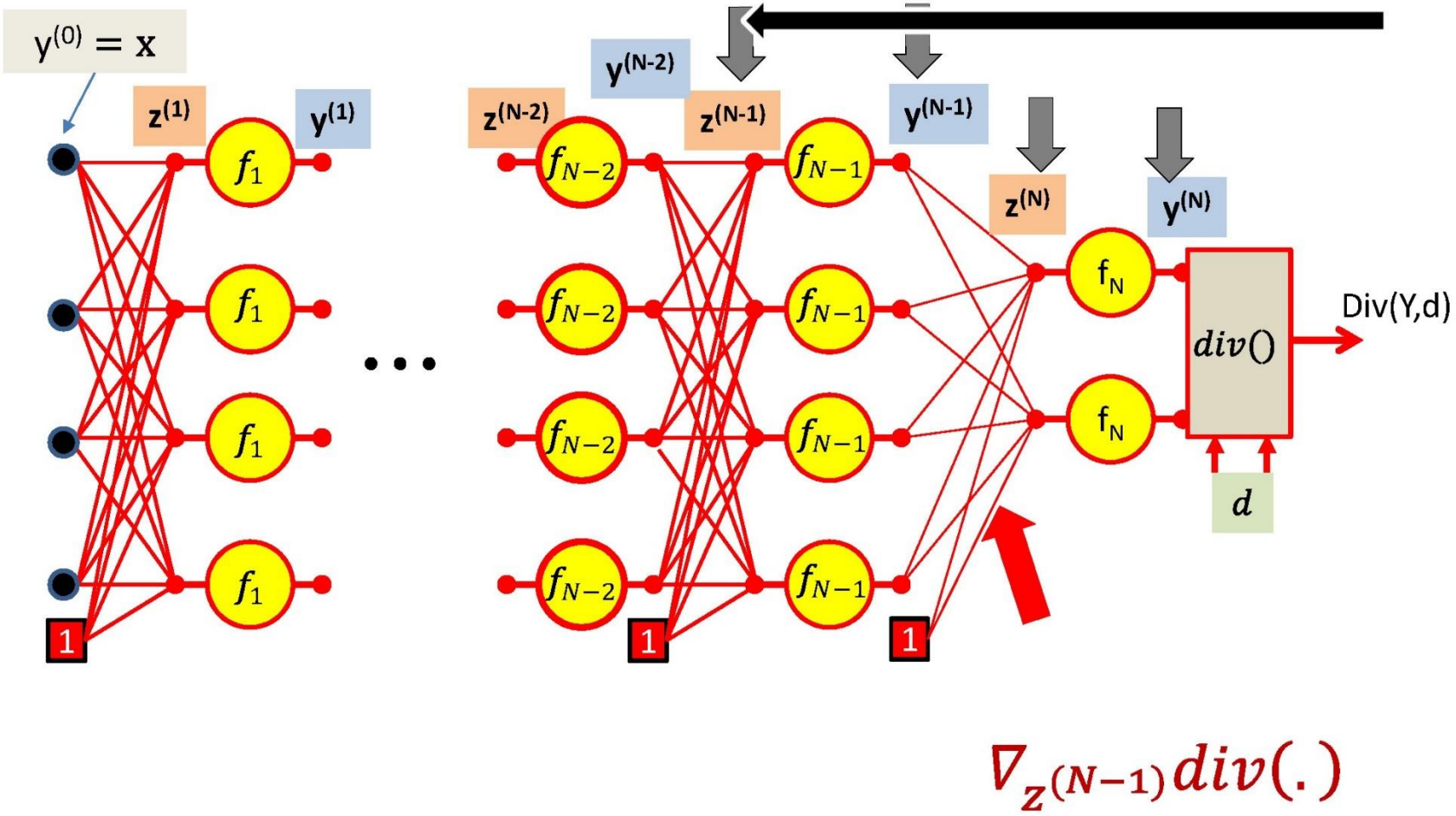
- Continuing on, we will compute $\nabla_{W(N)}$div(.) the derivative of the divergence with respect

Computing derivatives



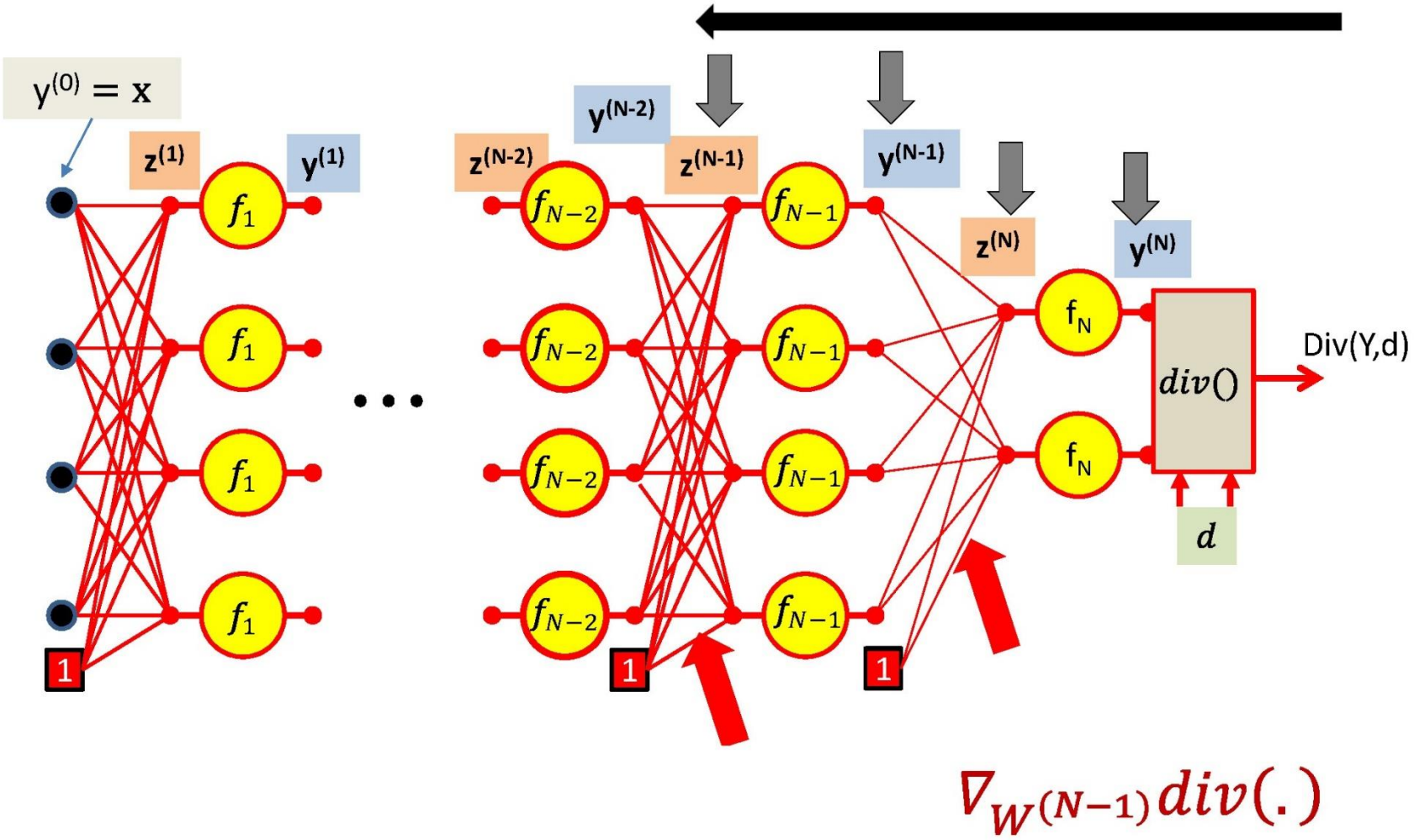
- Continuing on, we will compute $\nabla_{W(N)}\text{div}(.)$ the derivative of the divergence with respect to the weights of the connections to the output layer
- Then continue with the chain rule to compute $\nabla_{Y(N-1)}\text{div}(.)$ the derivative of the divergence w.r.t. the output of the N-1th layer

Computing derivatives



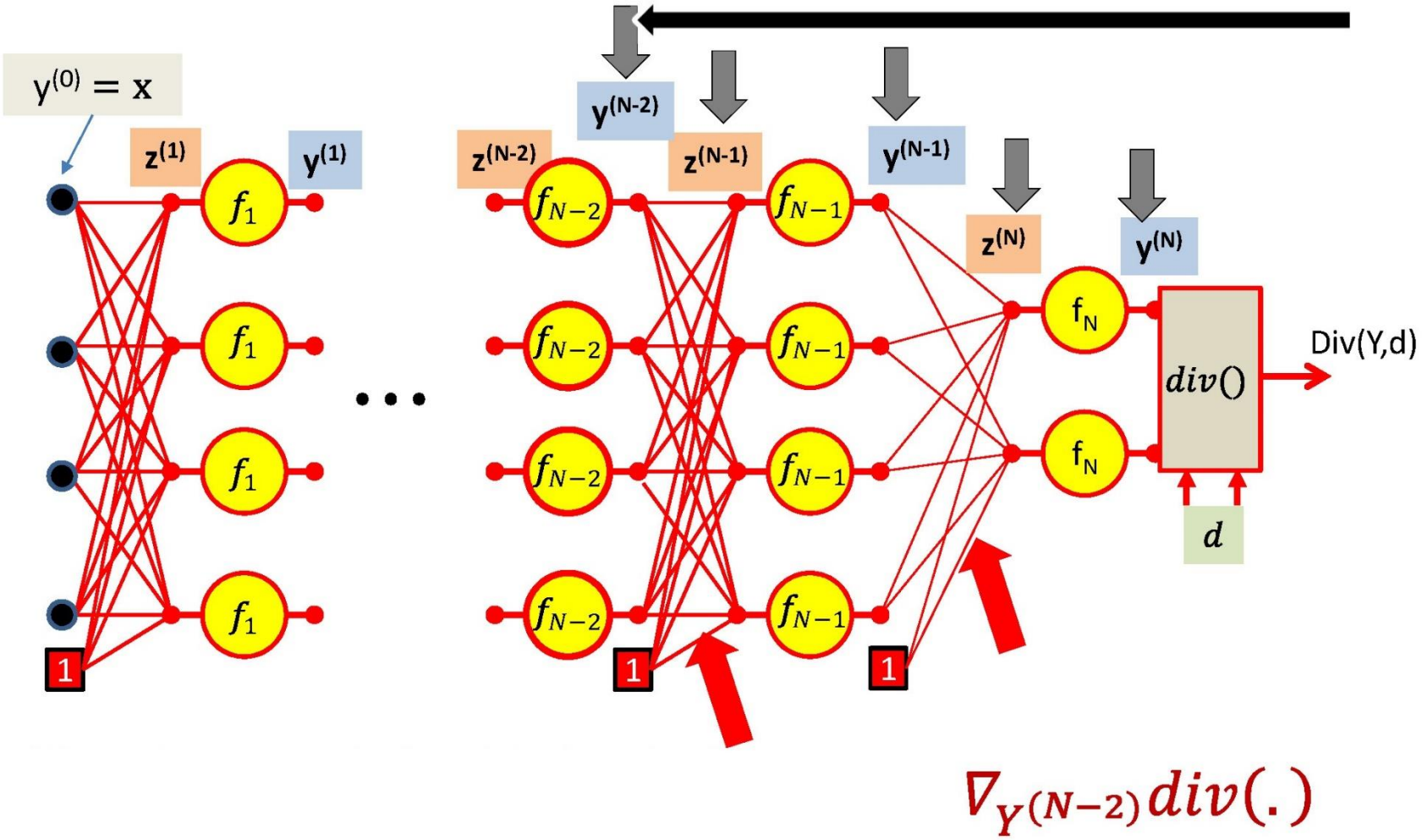
- We continue our way backwards in the order shown

Computing derivatives



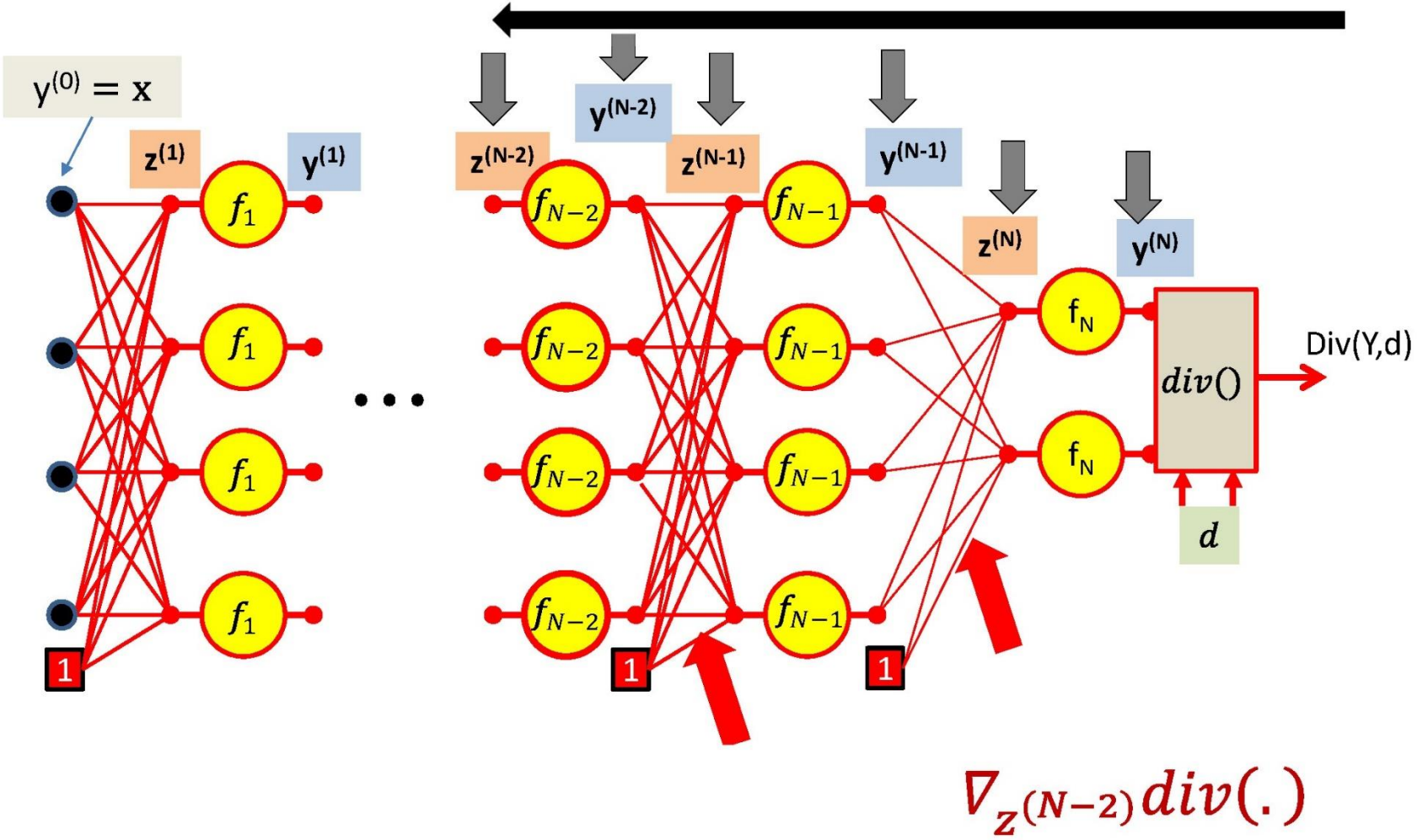
- We continue our way backwards in the order shown

Computing derivatives



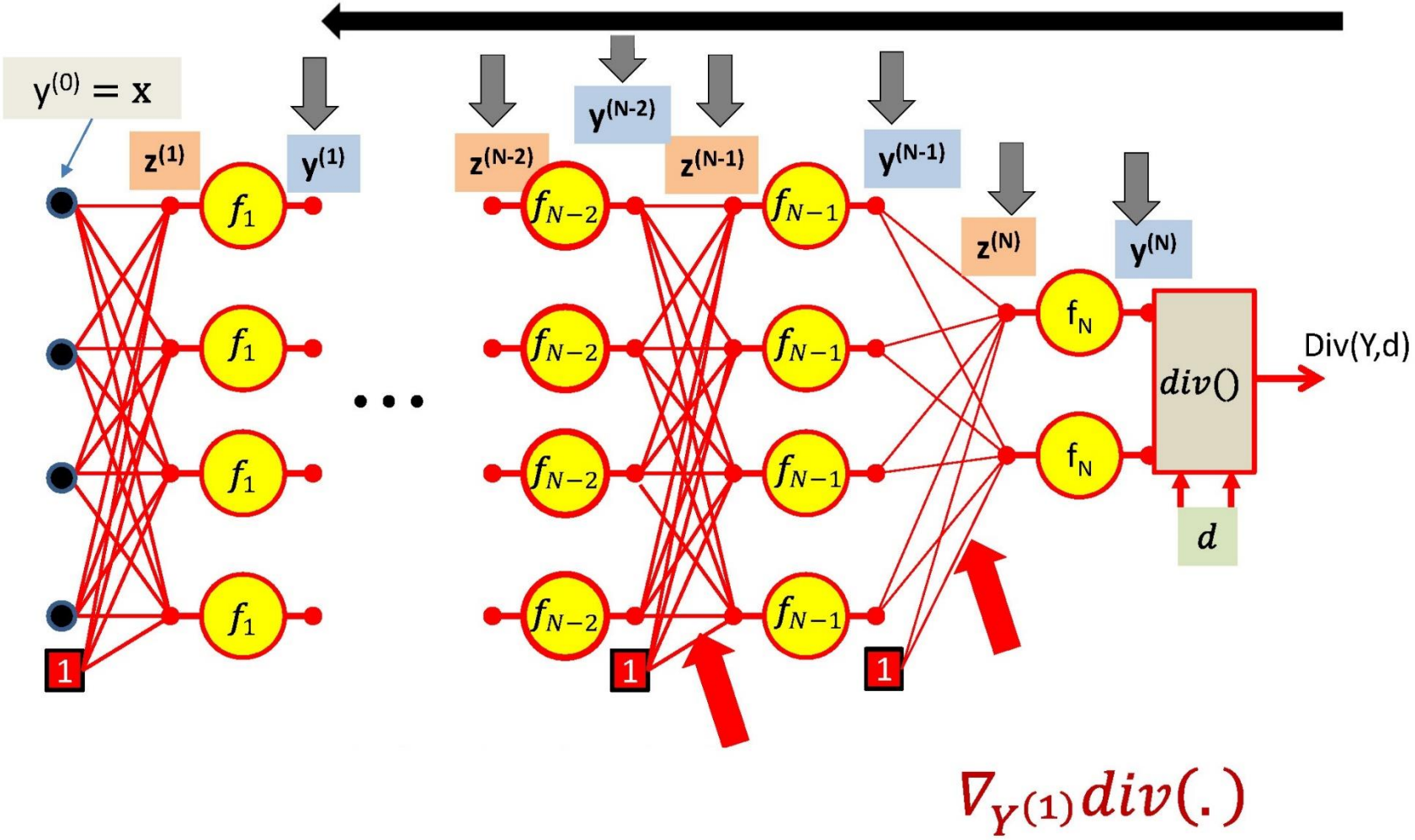
- We continue our way backwards in the order shown

Computing derivatives



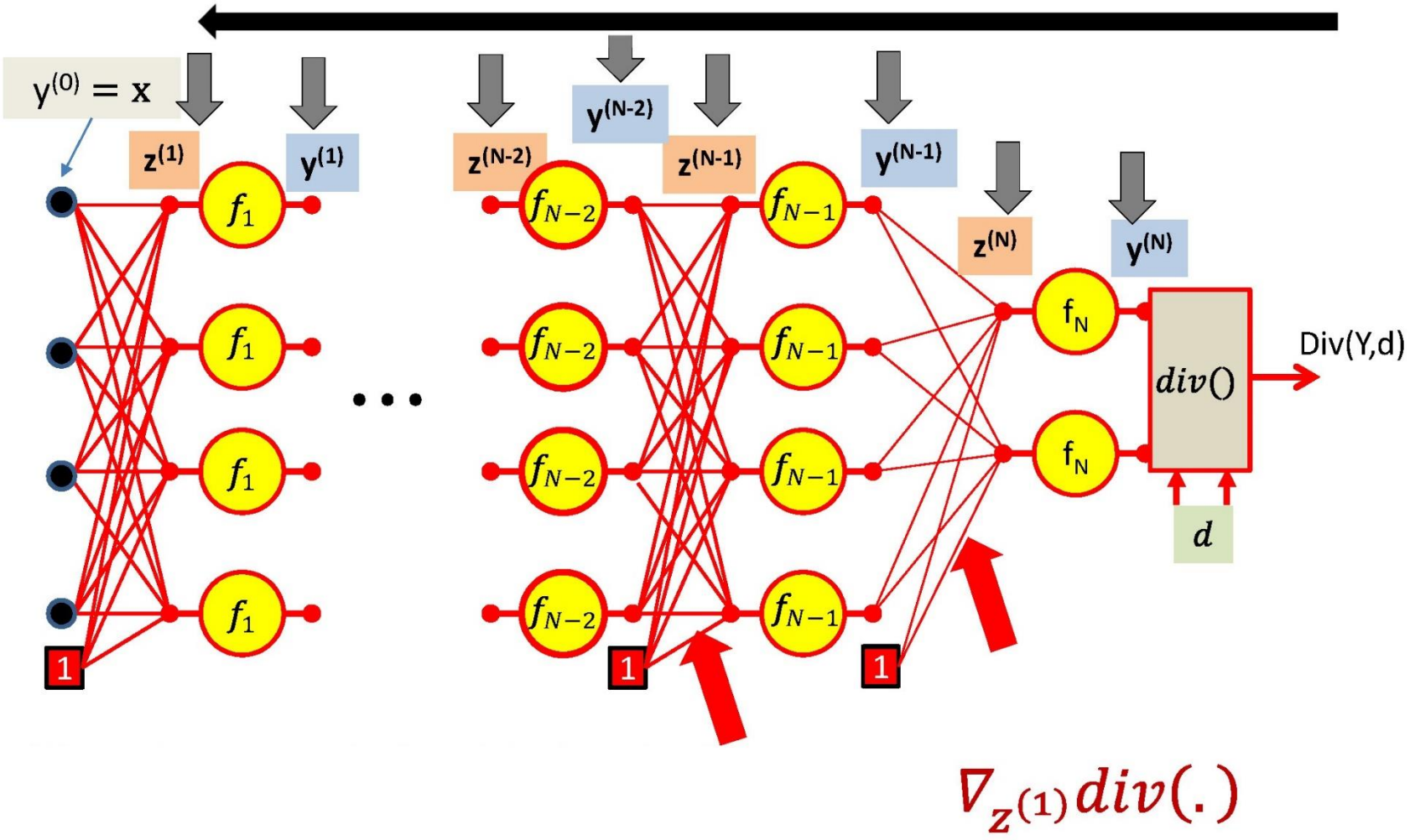
- We continue our way backwards in the order shown

Computing derivatives



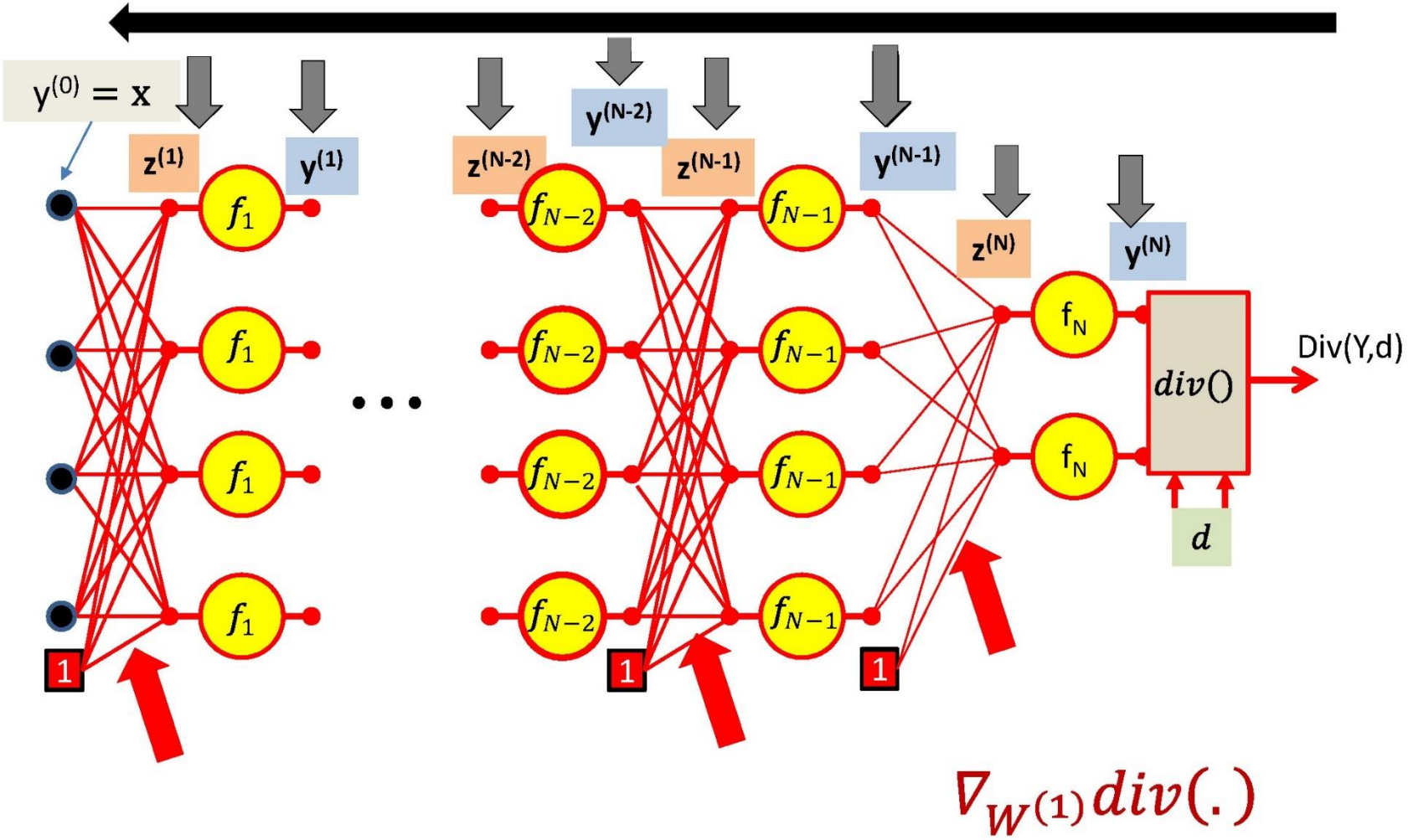
- We continue our way backwards in the order shown

Computing derivatives



- We continue our way backwards in the order shown

Computing derivatives

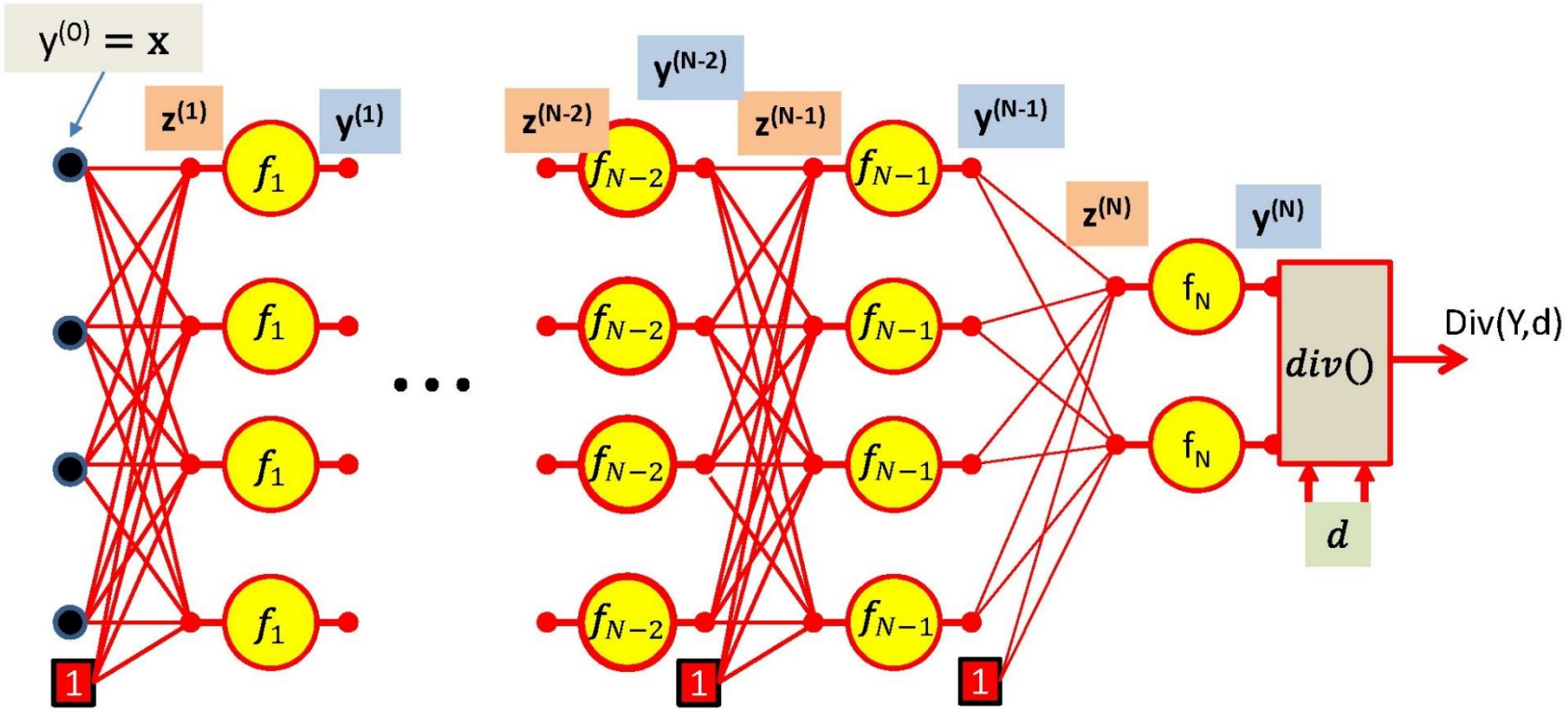


- We continue our way backwards in the order shown

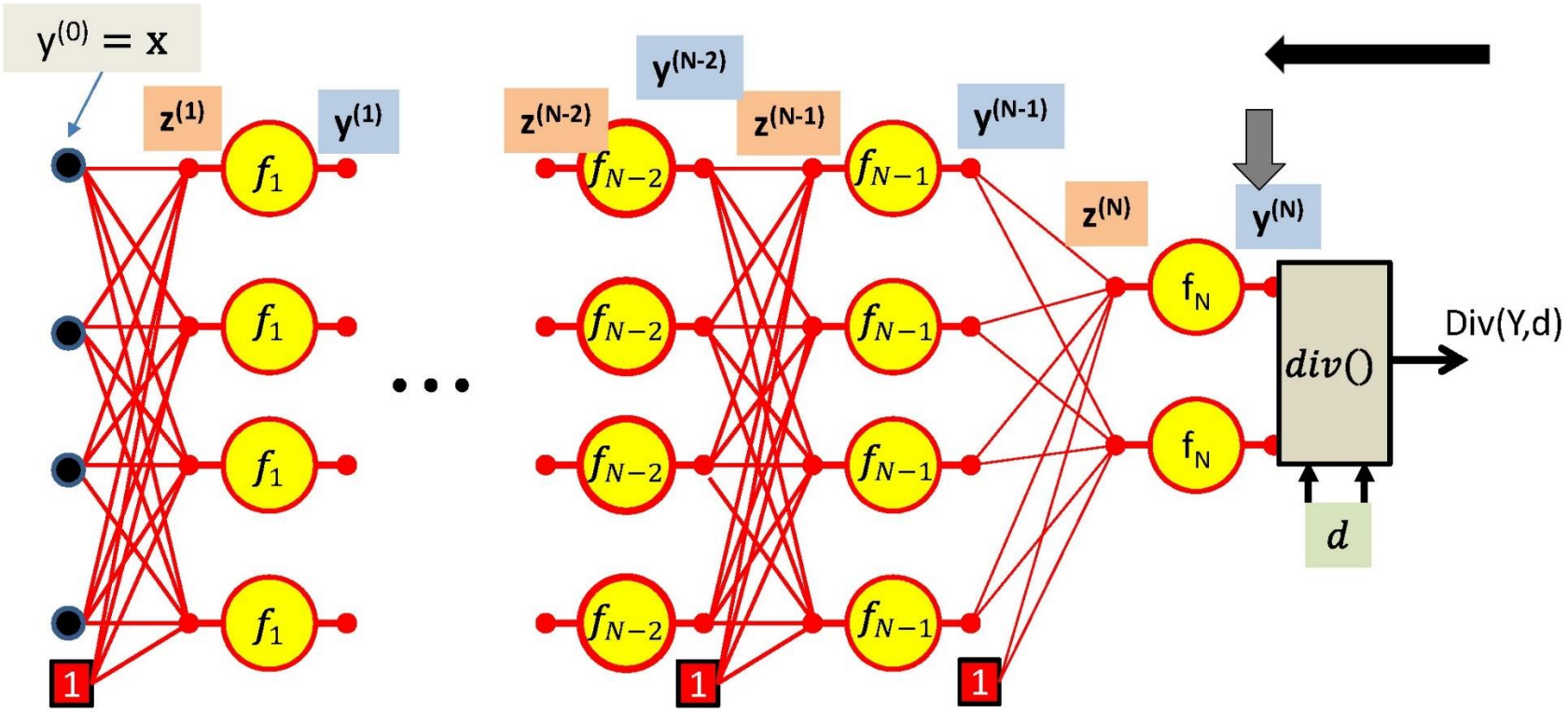
Backward Gradient Computation

- Lets actually see the math . . .

Backward Gradient Computation



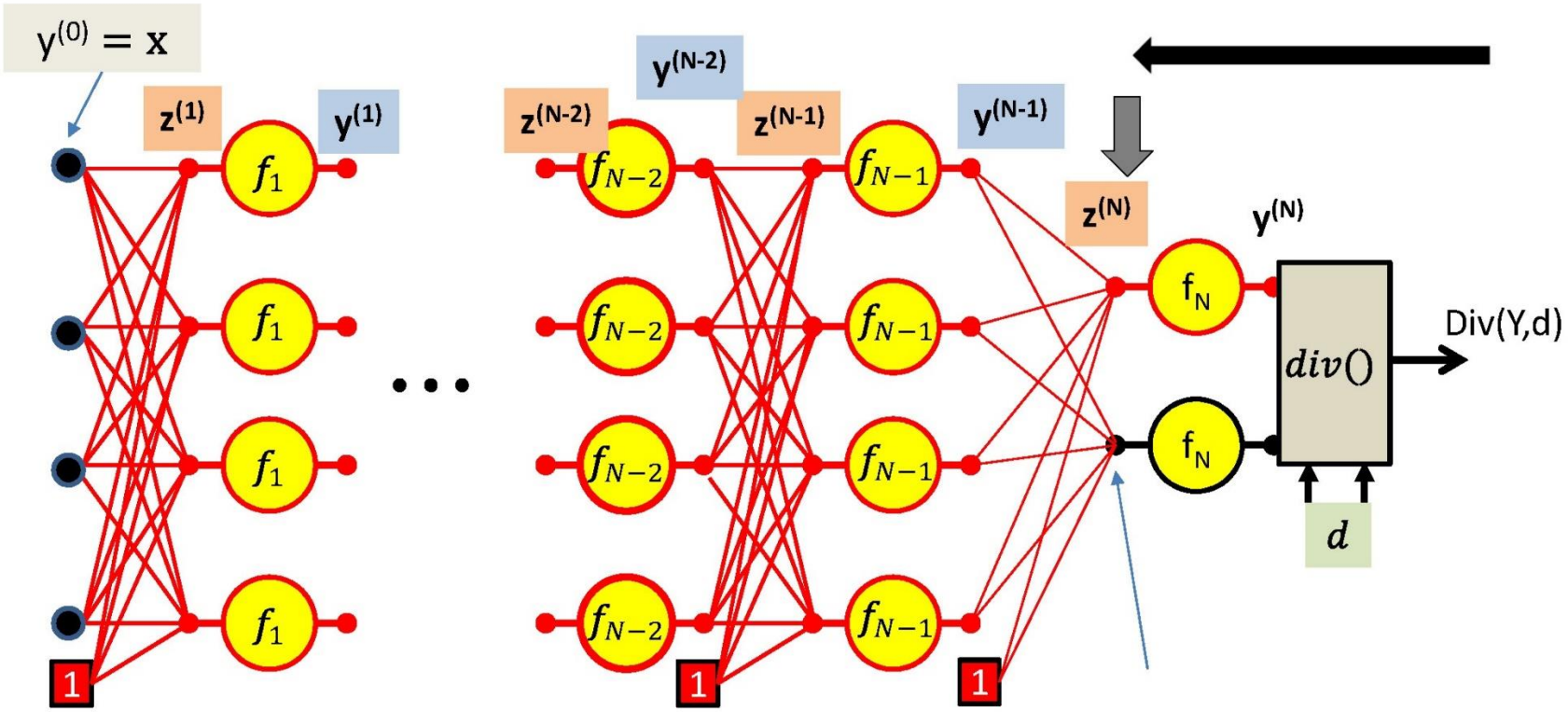
Backward Gradient Computation



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

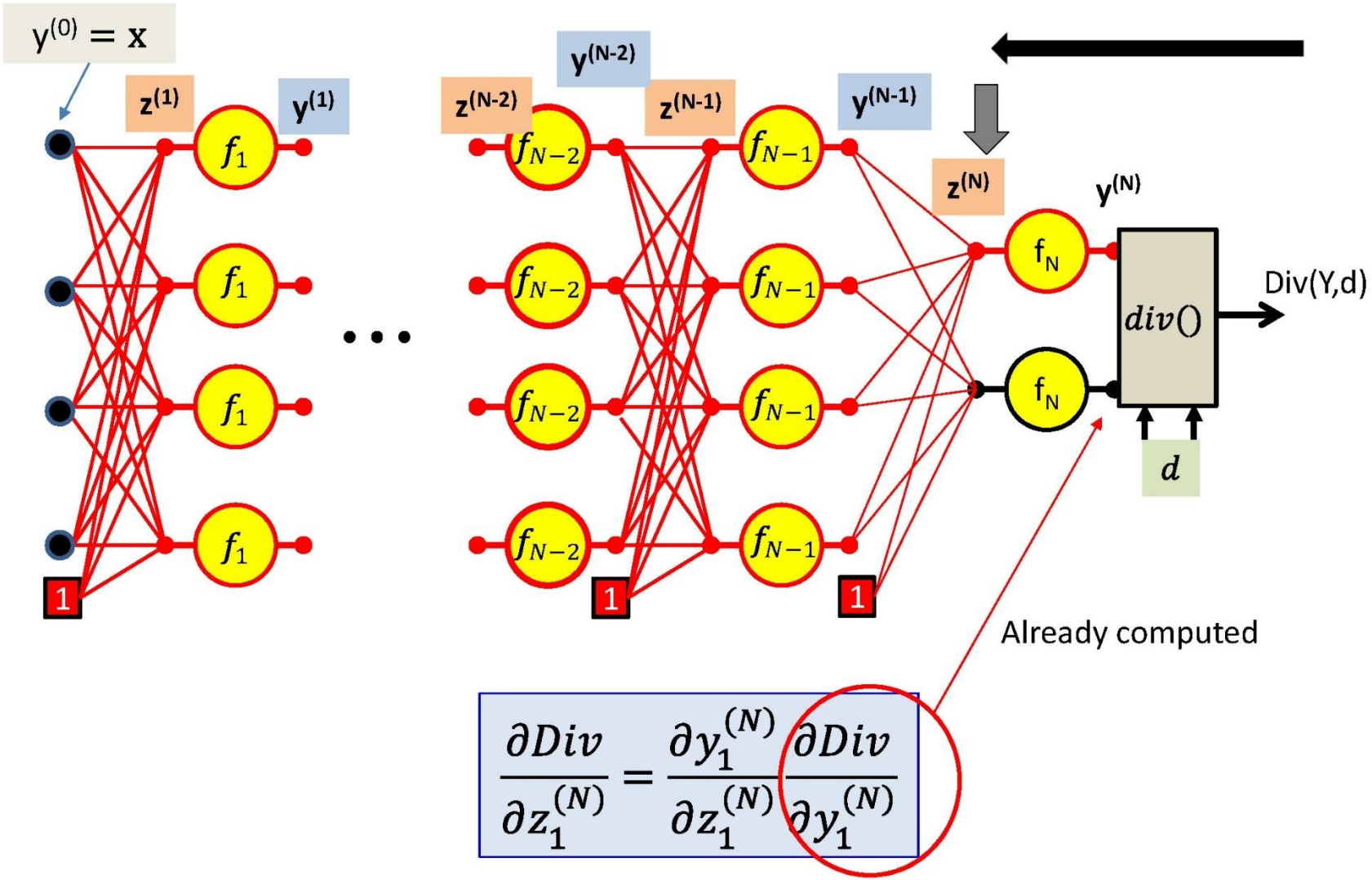
$$\frac{\partial Div(Y,d)}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

Backward Gradient Computation

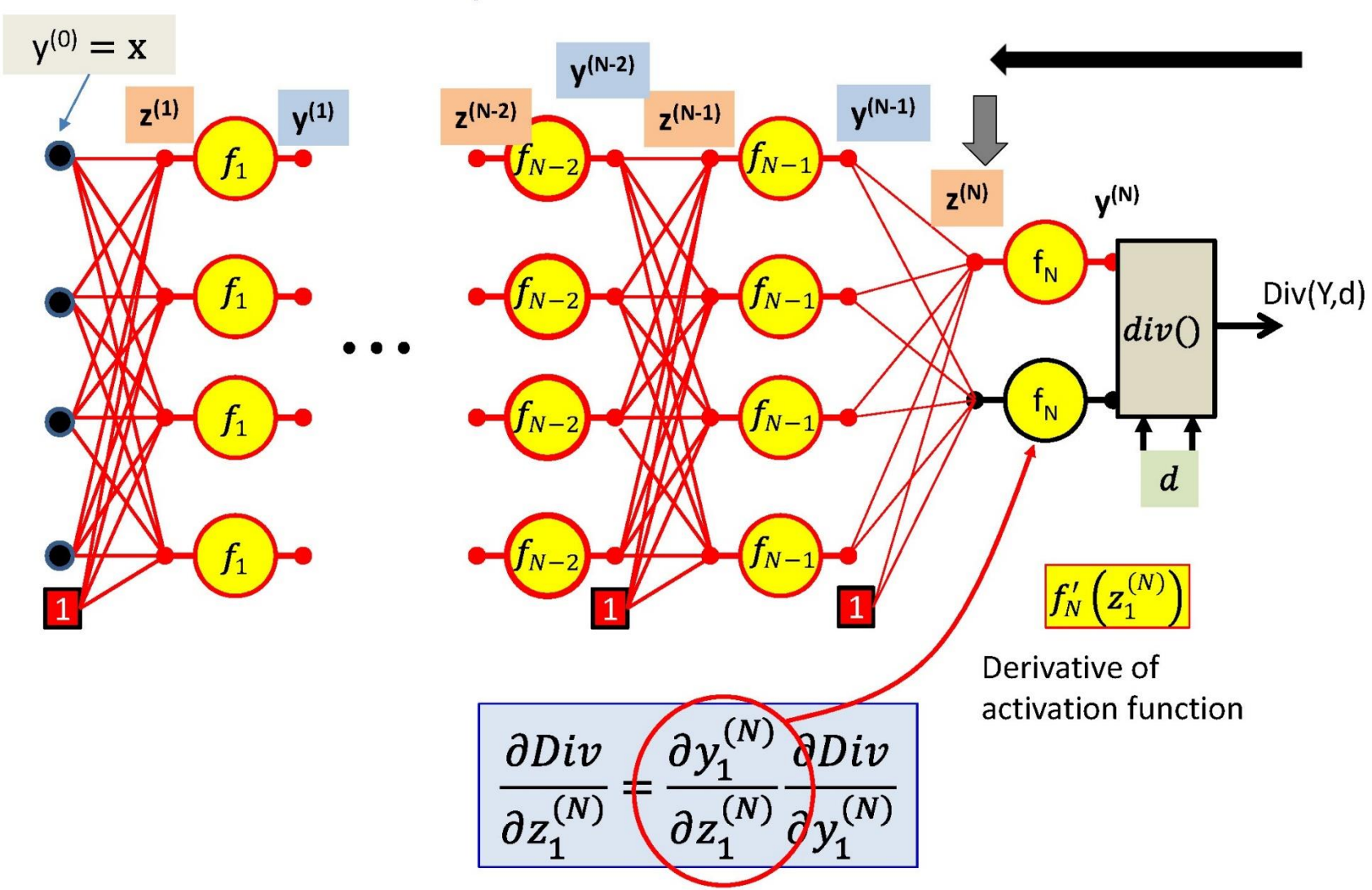


$$\frac{\partial Div}{\partial z_1^{(N)}} = \frac{\partial y_1^{(N)}}{\partial z_1^{(N)}} \frac{\partial Div}{\partial y_1^{(N)}}$$

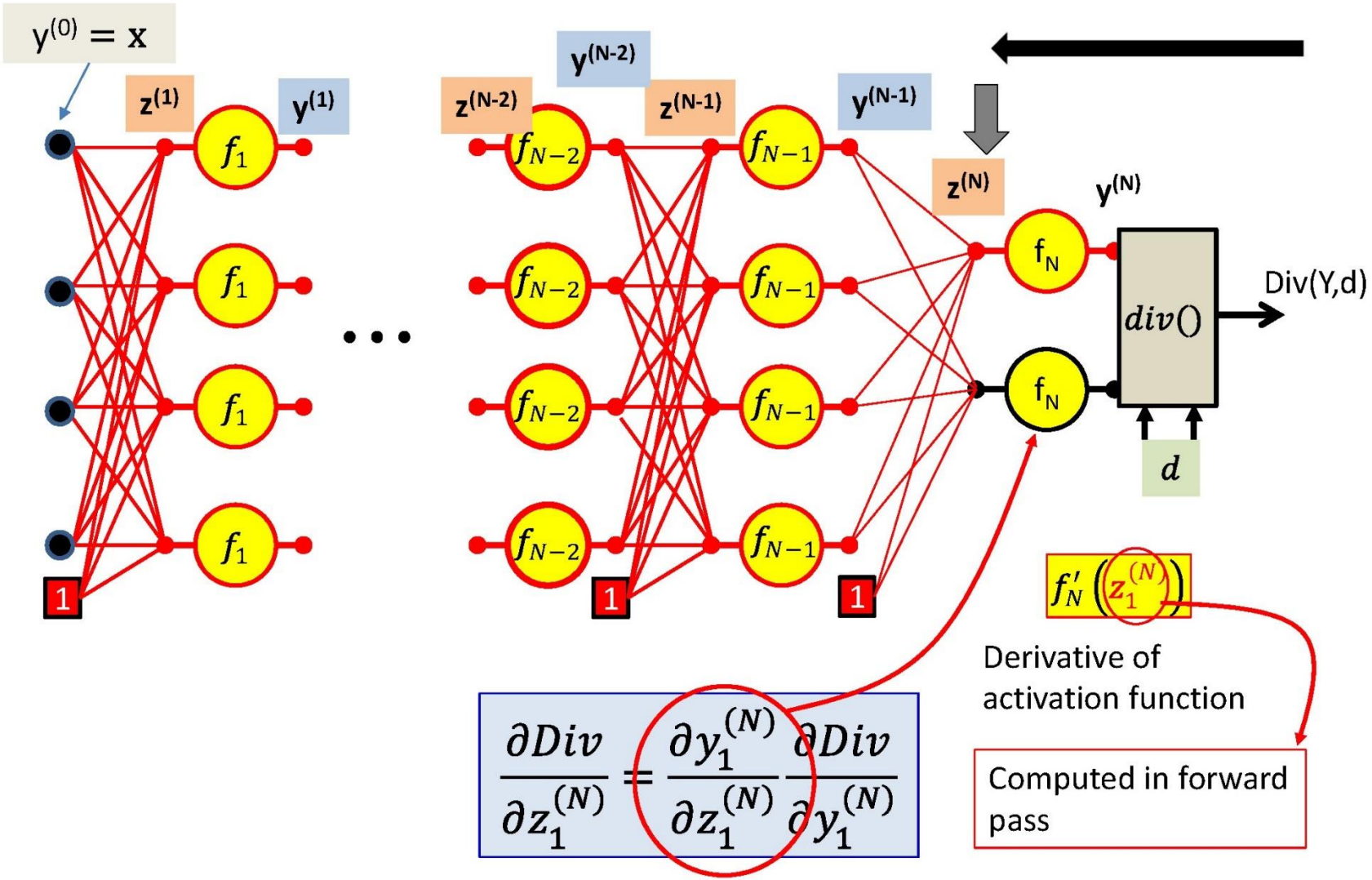
Backward Gradient Computation



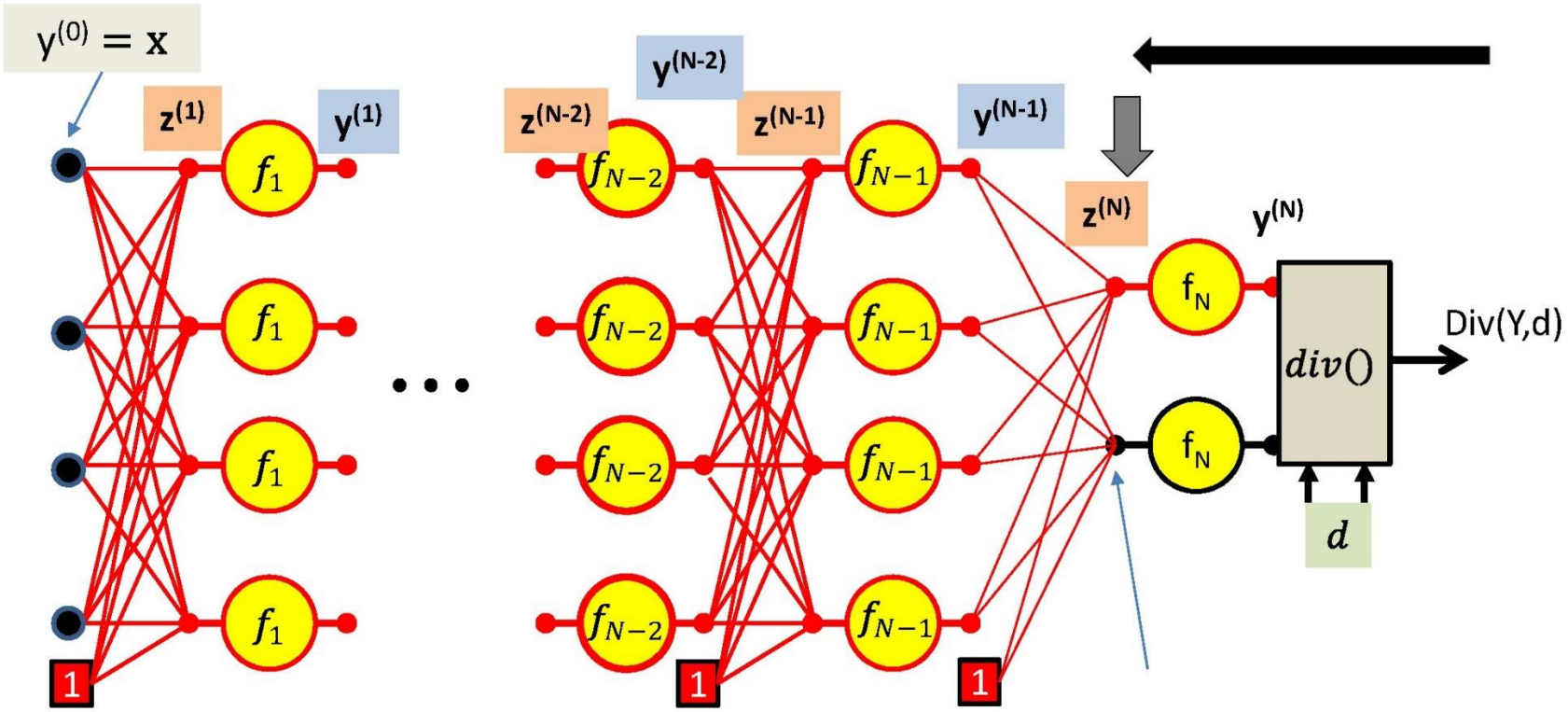
Backward Gradient Computation



Backward Gradient Computation

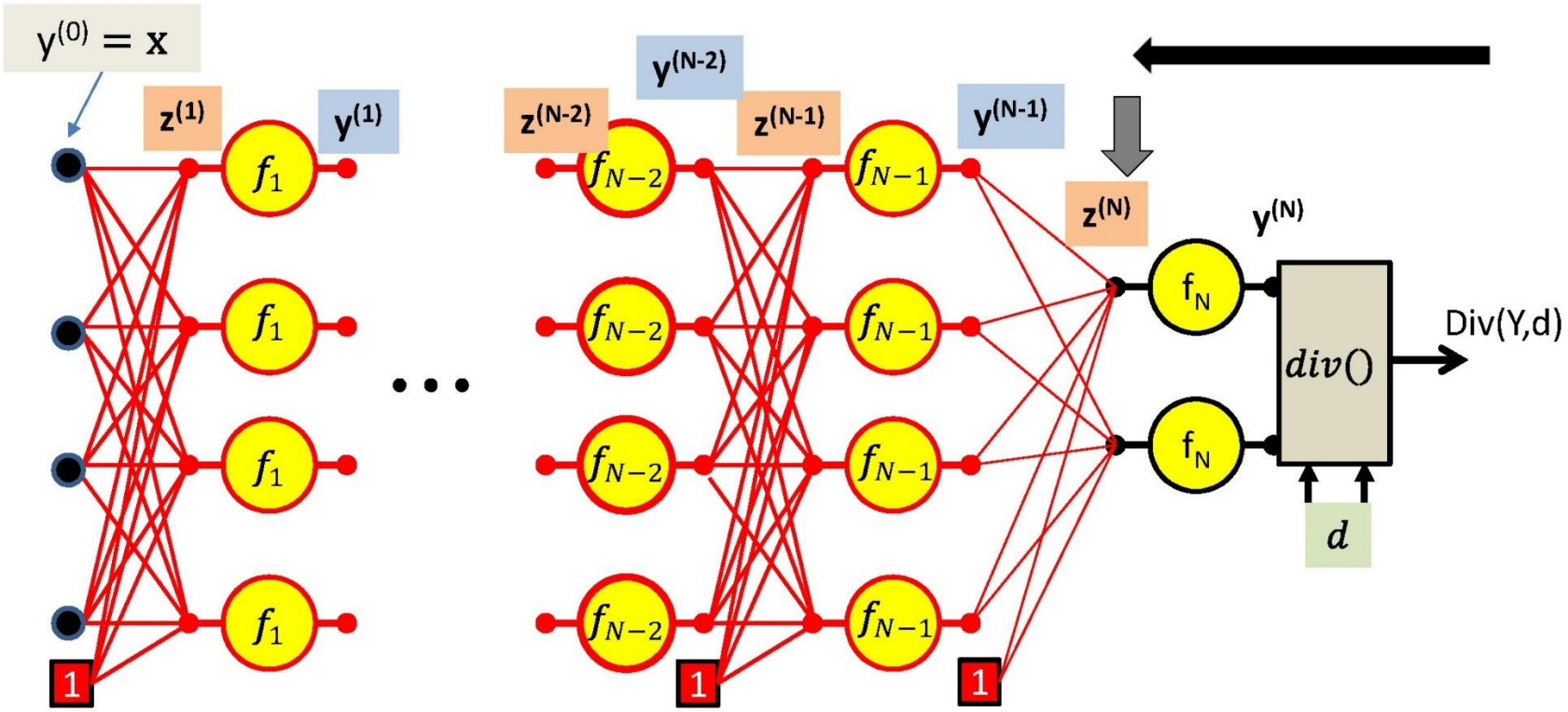


Backward Gradient Computation



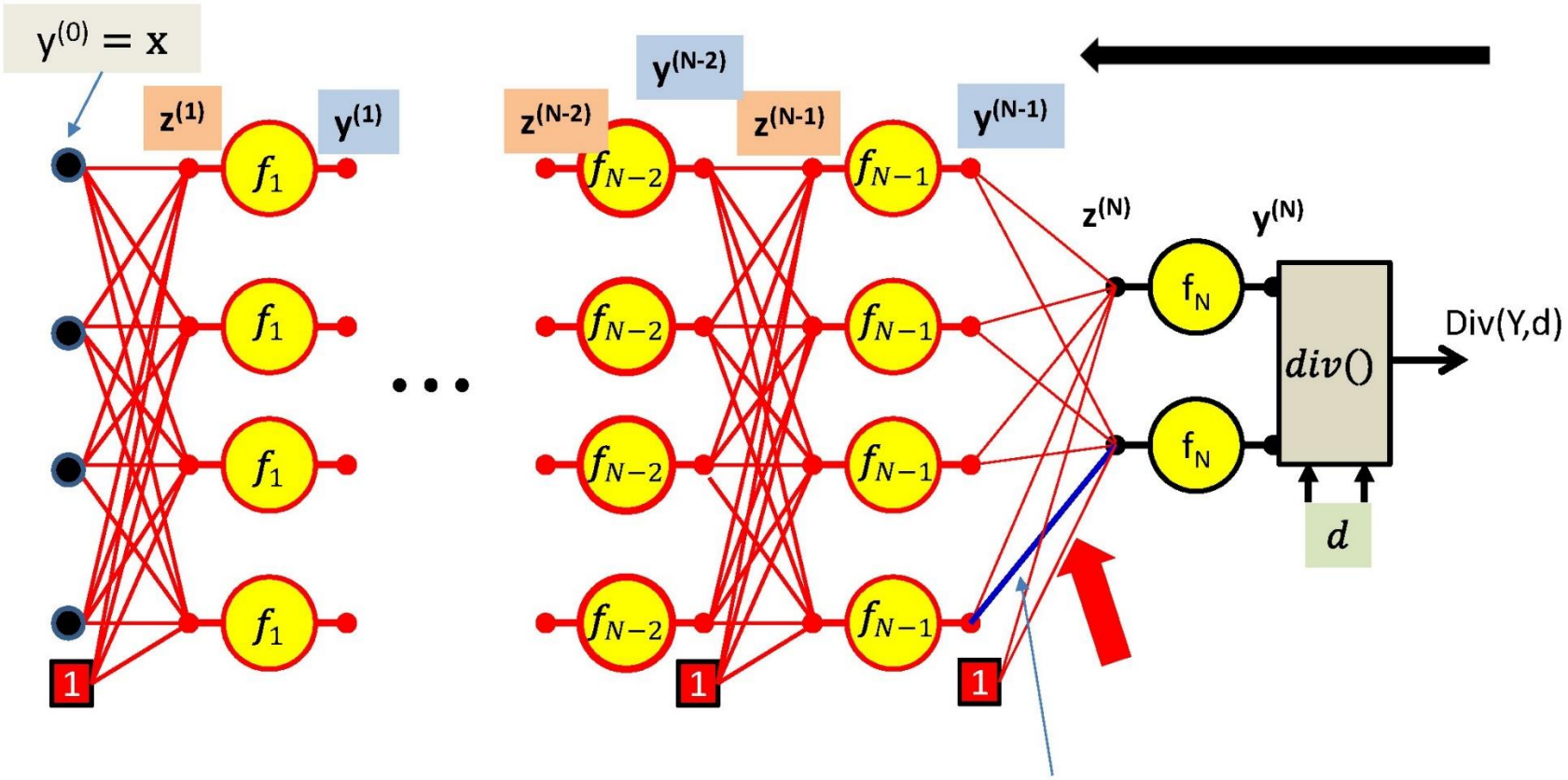
$$\frac{\partial Div}{\partial z_1^{(N)}} = f_N'\left(z_1^{(N)}\right)\frac{\partial Div}{\partial y_1^{(N)}}$$

Backward Gradient Computation



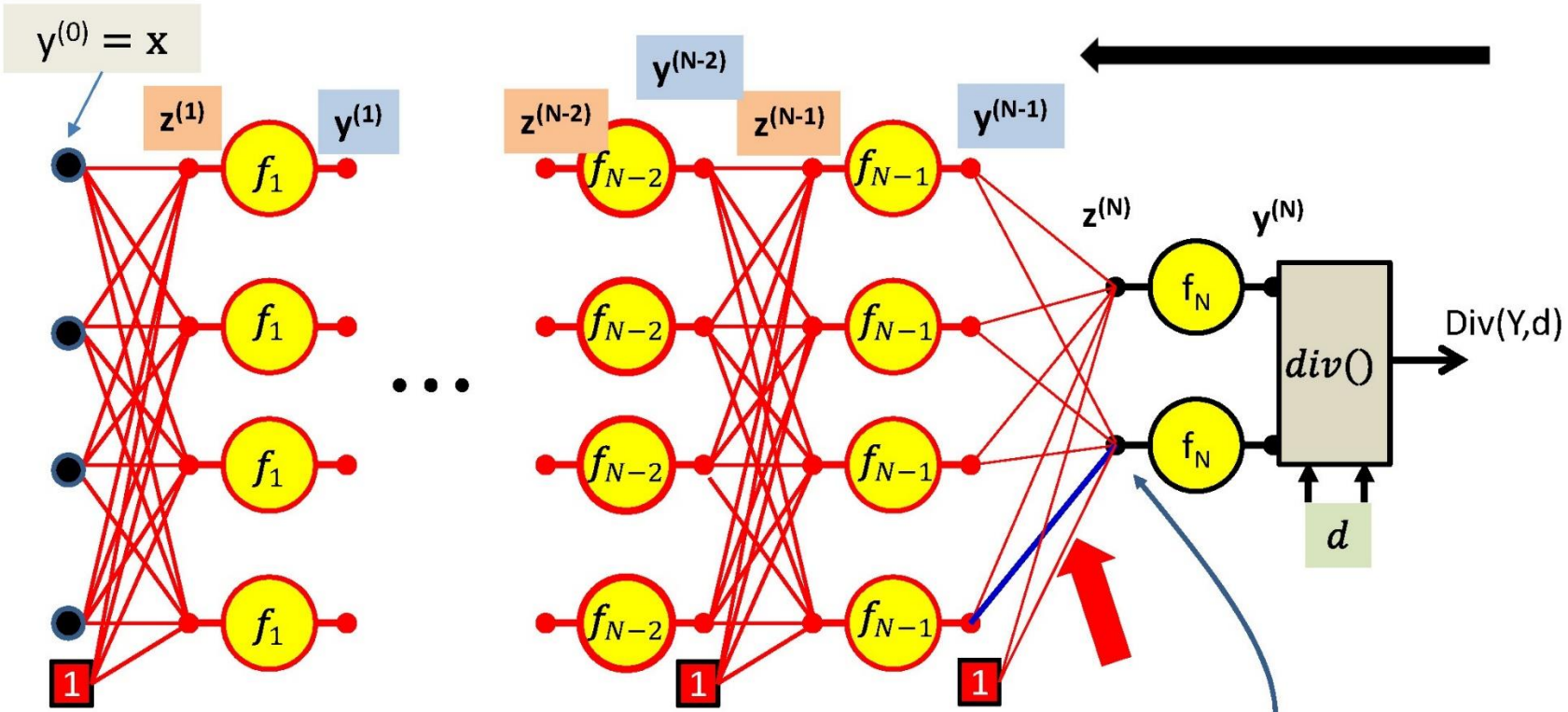
$$\frac{\partial Div}{\partial z_i^{(N)}} = f_N'\left(z_i^{(N)}\right)\frac{\partial Div}{\partial y_i^{(N)}}$$

Backward Gradient Computation



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

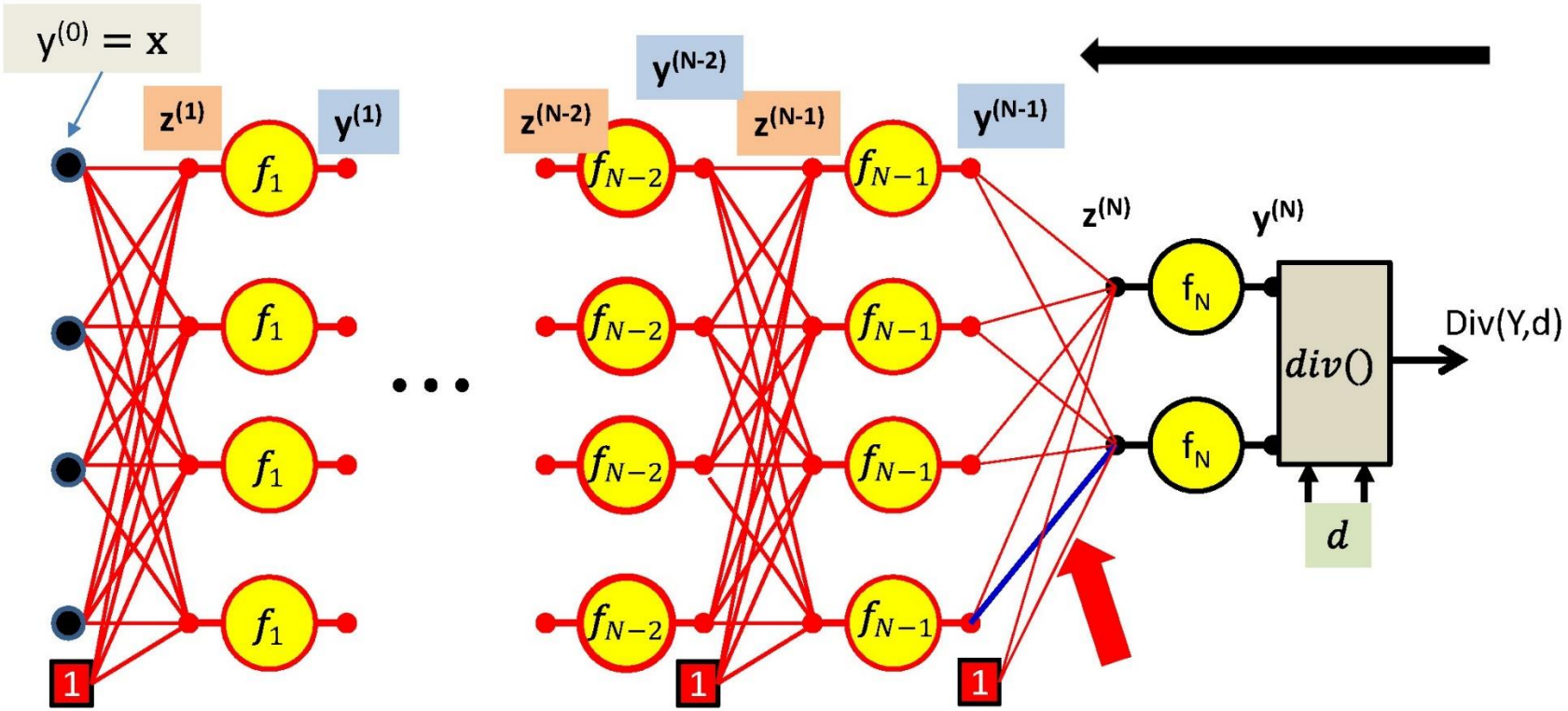
Backward Gradient Computation



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

Just computed

Backward Gradient Computation



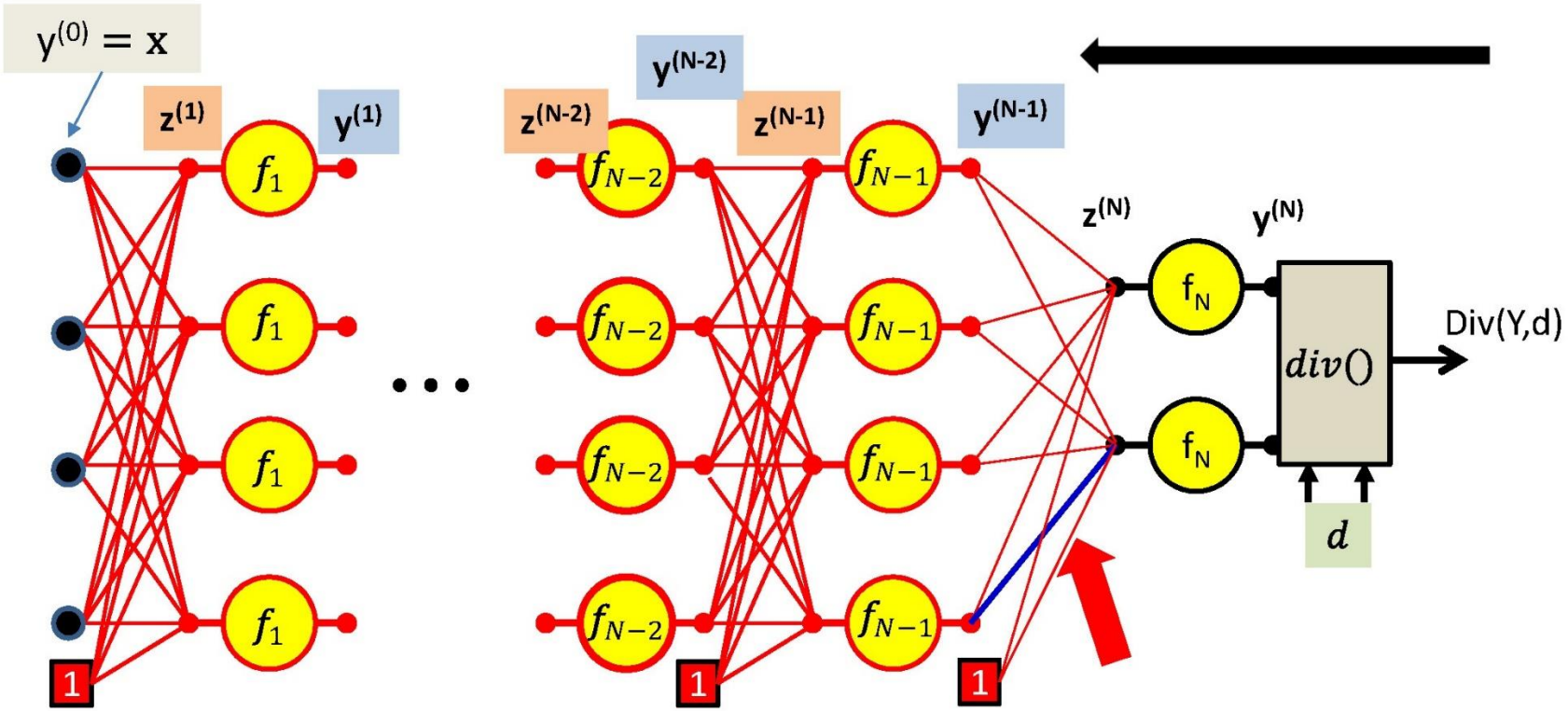
$$\frac{\partial Div}{\partial w_{11}^{(N)}} = \frac{\partial z_1^{(N)}}{\partial w_{11}^{(N)}} \frac{\partial Div}{\partial z_1^{(N)}}$$

$y_1^{(N-1)}$

Because

$$z_1^{(N)} = w_{11}^{(N)} y_1^{(N-1)} + \text{other terms}$$

Backward Gradient Computation



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

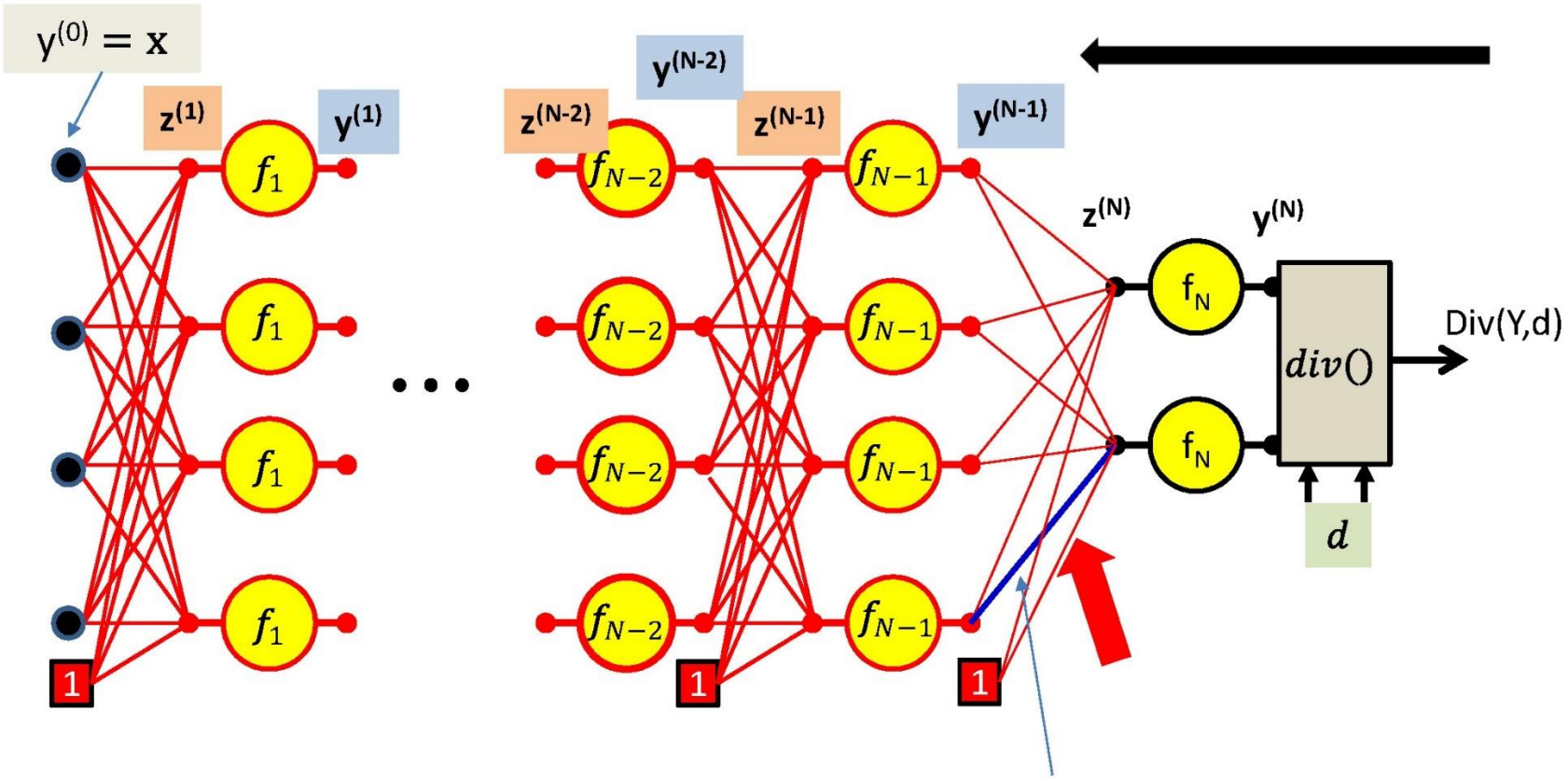
$y_1^{(N-1)}$

Because

$$z_1^{(N)} = w_{11}^{(N)} y_1^{(N-1)} + \text{other terms}$$

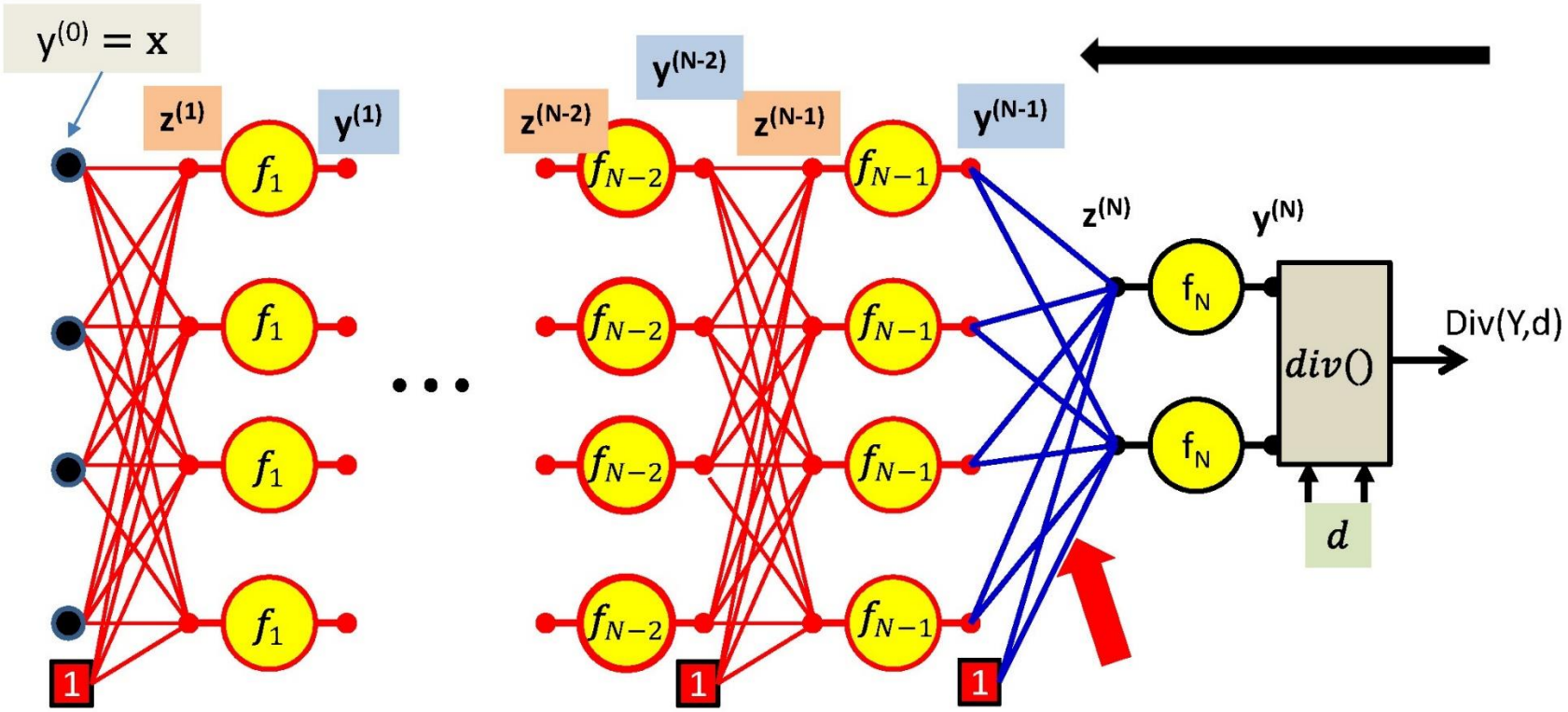
Computed in forward pass

Backward Gradient Computation



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

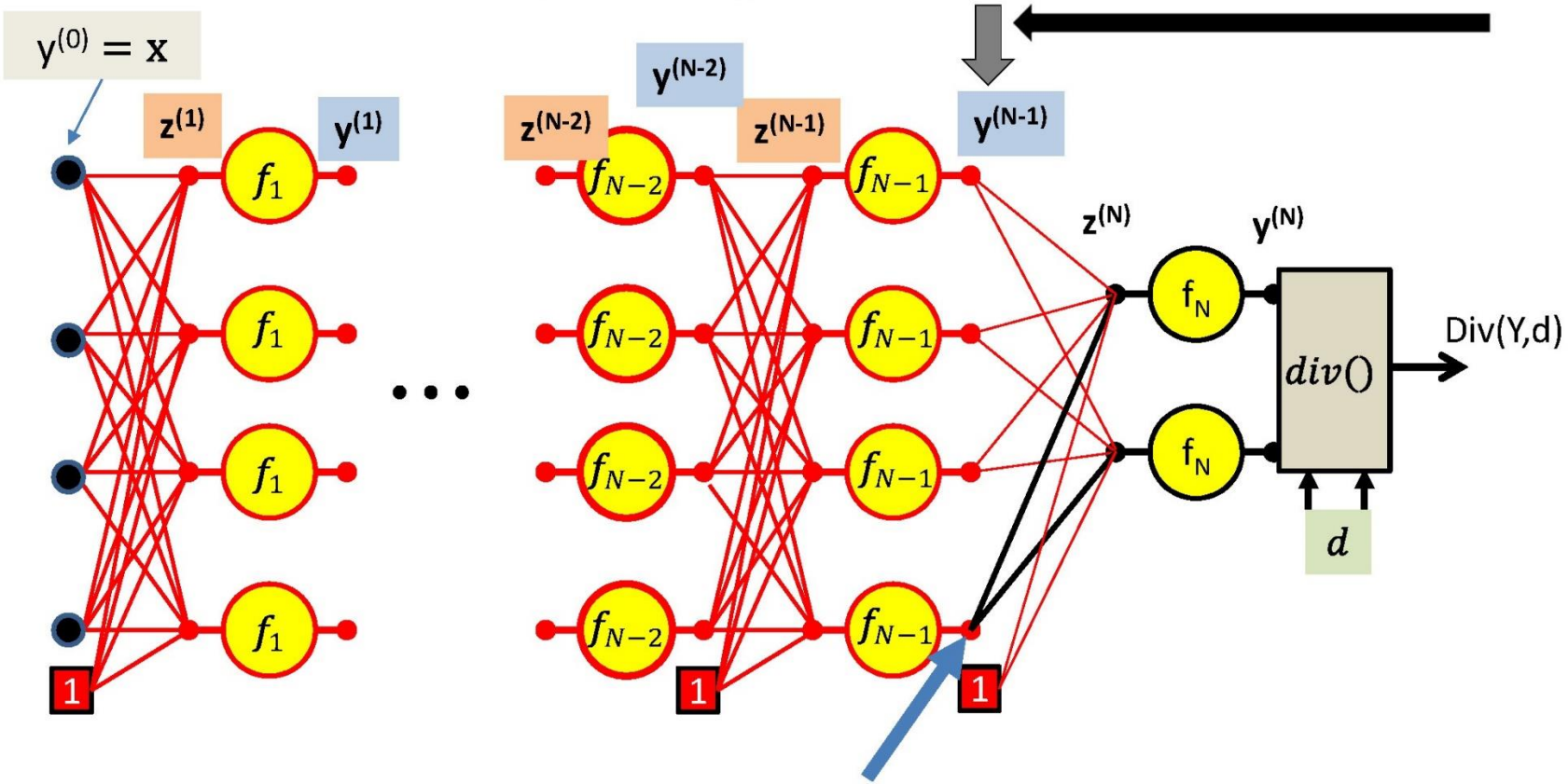
Backward Gradient Computation



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

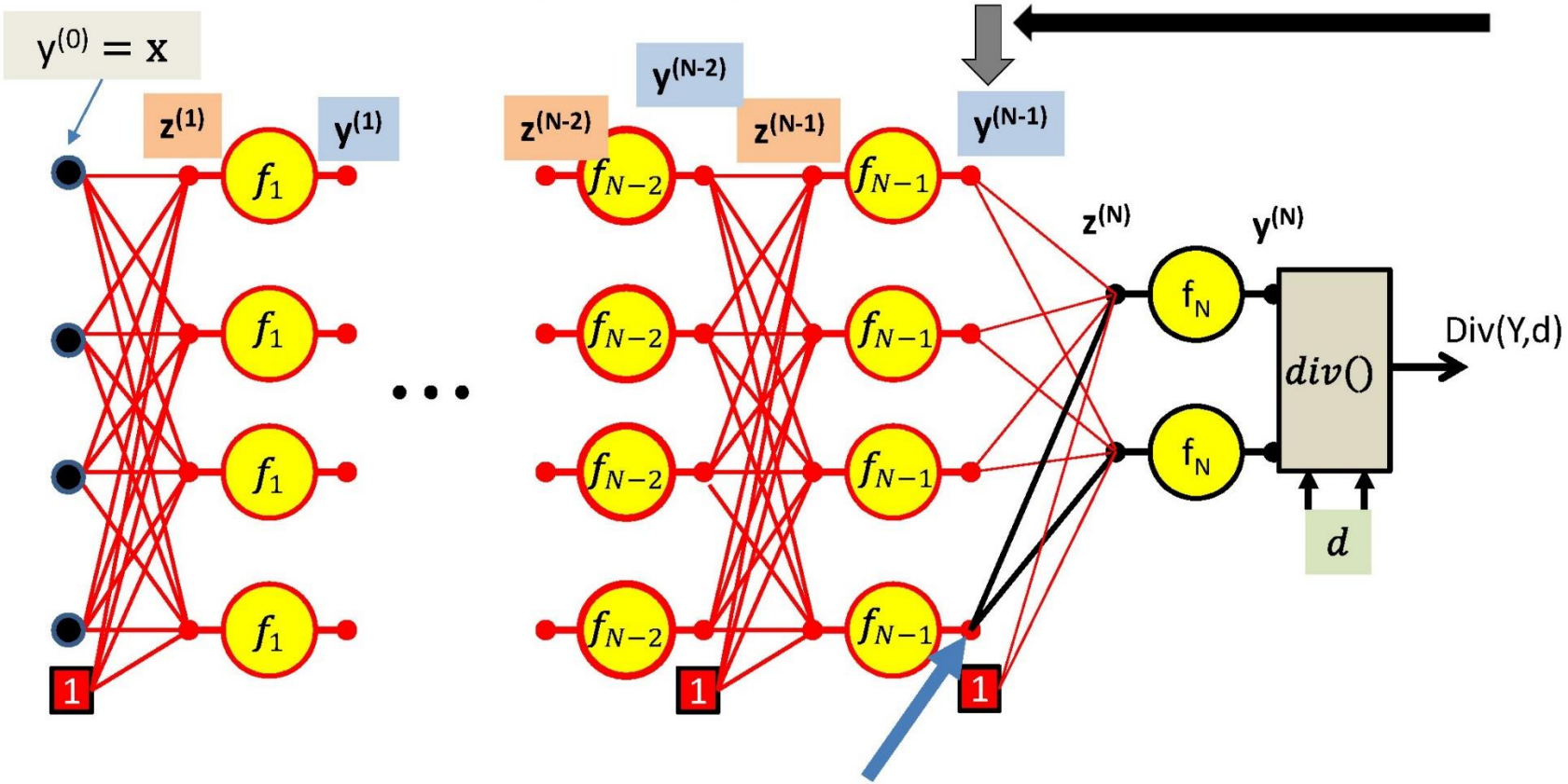
For the bias term $y_0^{(N-1)} = 1$

Backward Gradient Computation



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

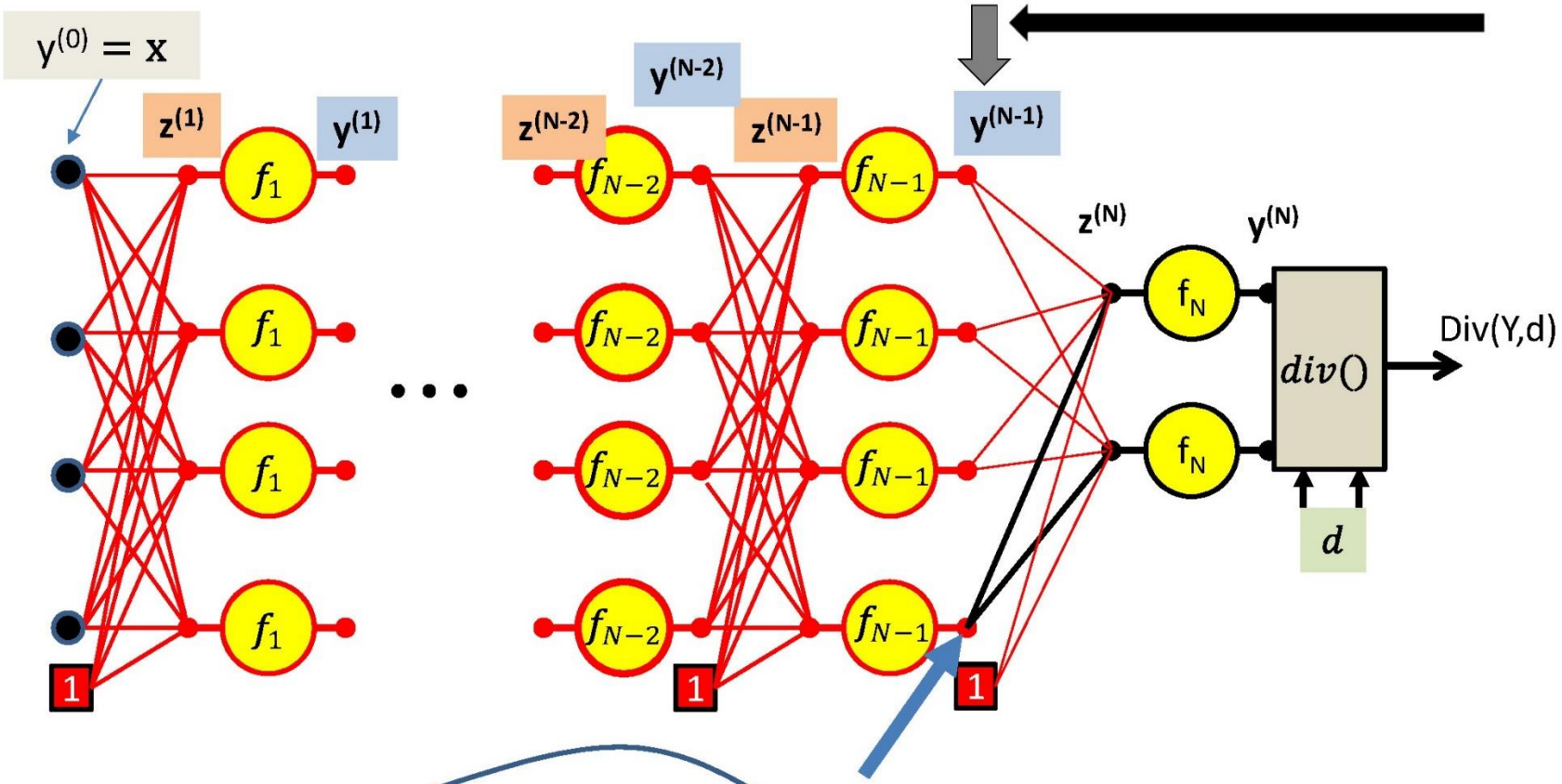
Backward Gradient Computation



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

Already computed

Backward Gradient Computation



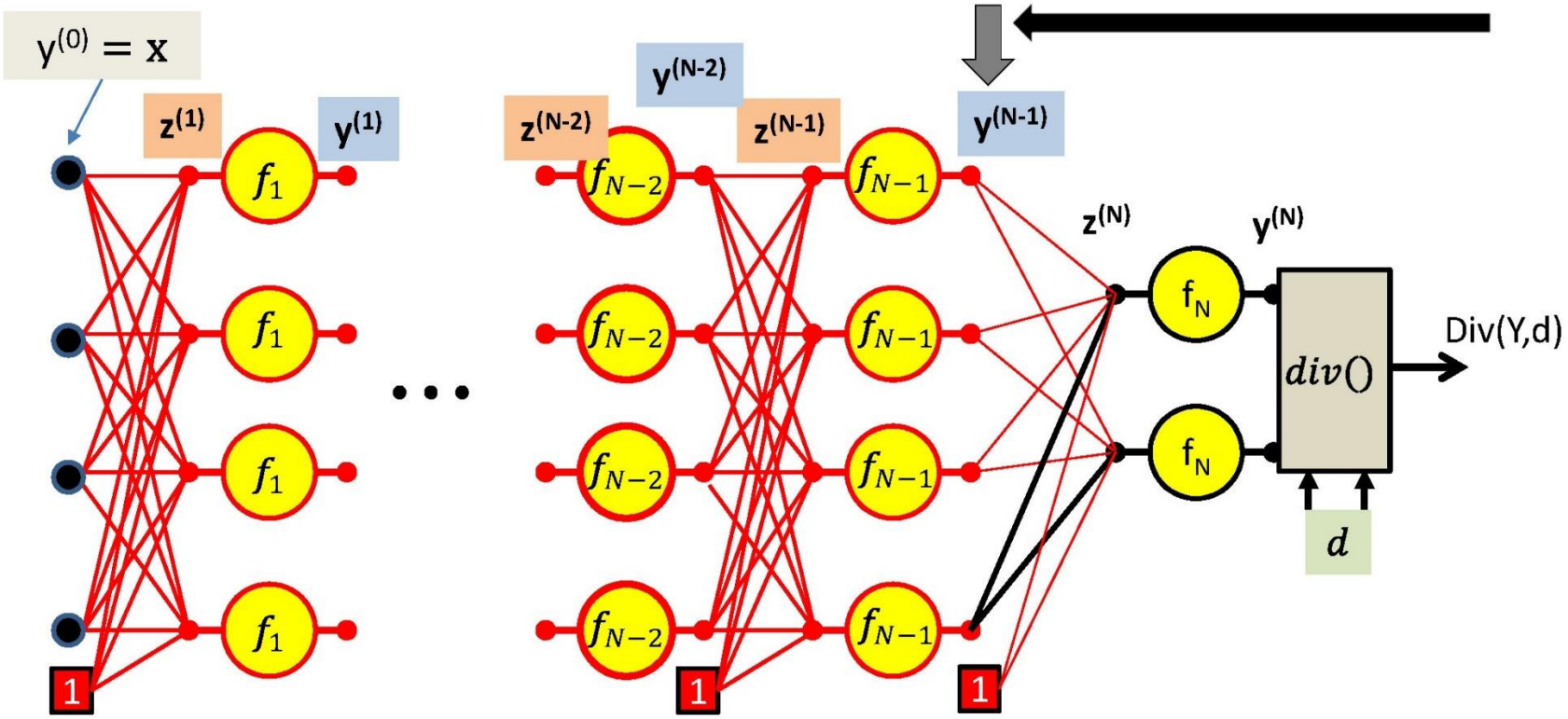
$$\frac{\partial Div}{\partial y_1^{(N-1)}} = \sum_j \frac{\partial z_j^{(N)}}{\partial y_1^{(N-1)}} \frac{\partial Div}{\partial z_j^{(N)}}$$

$w_{1j}^{(N-1)}$

Because

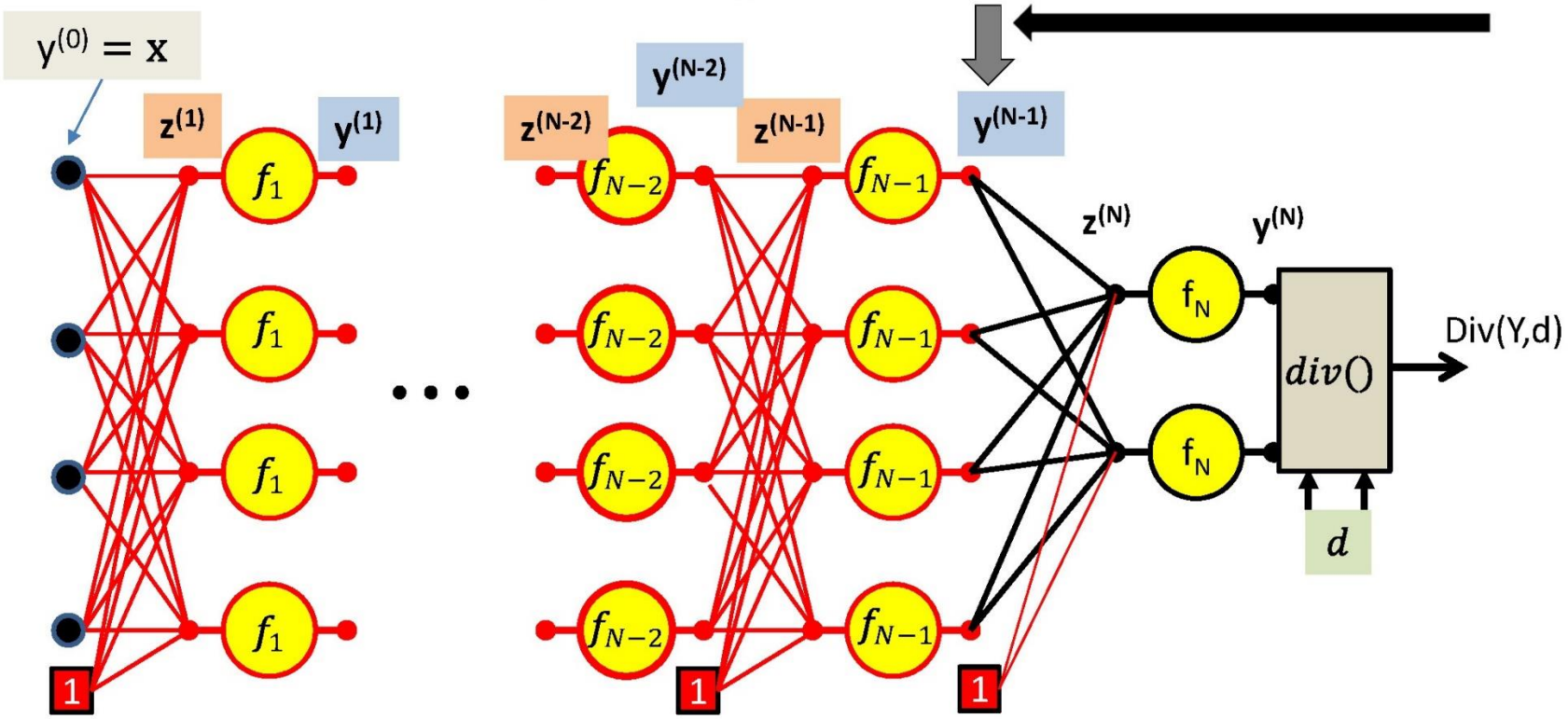
$$z_j^{(N)} = w_{1j}^{(N)} y_1^{(N-1)} + \text{other terms}$$

Backward Gradient Computation



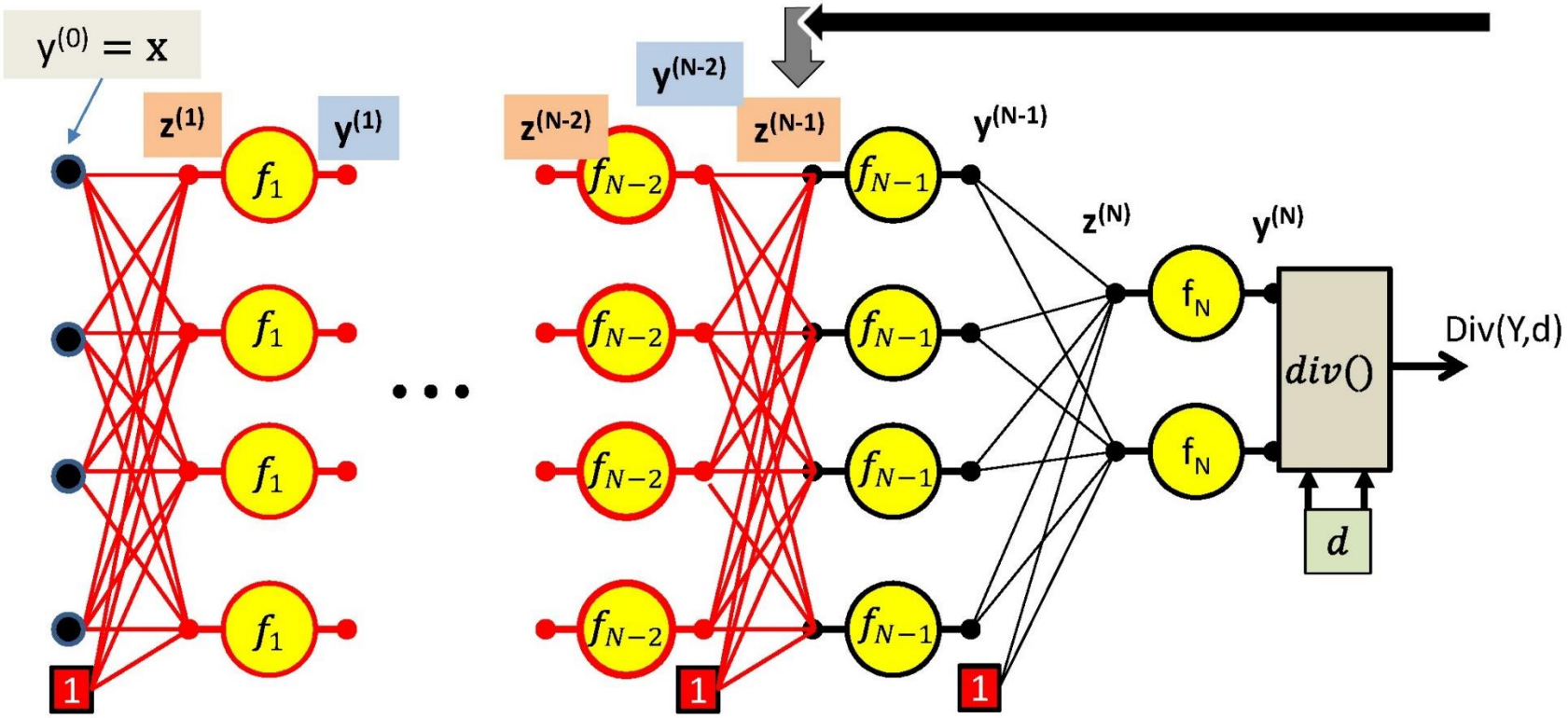
$$\frac{\partial Div}{\partial y_1^{(N-1)}} = \sum_j w_{1j}^{(N)} \frac{\partial Div}{\partial z_j^{(N)}}$$

Backward Gradient Computation



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

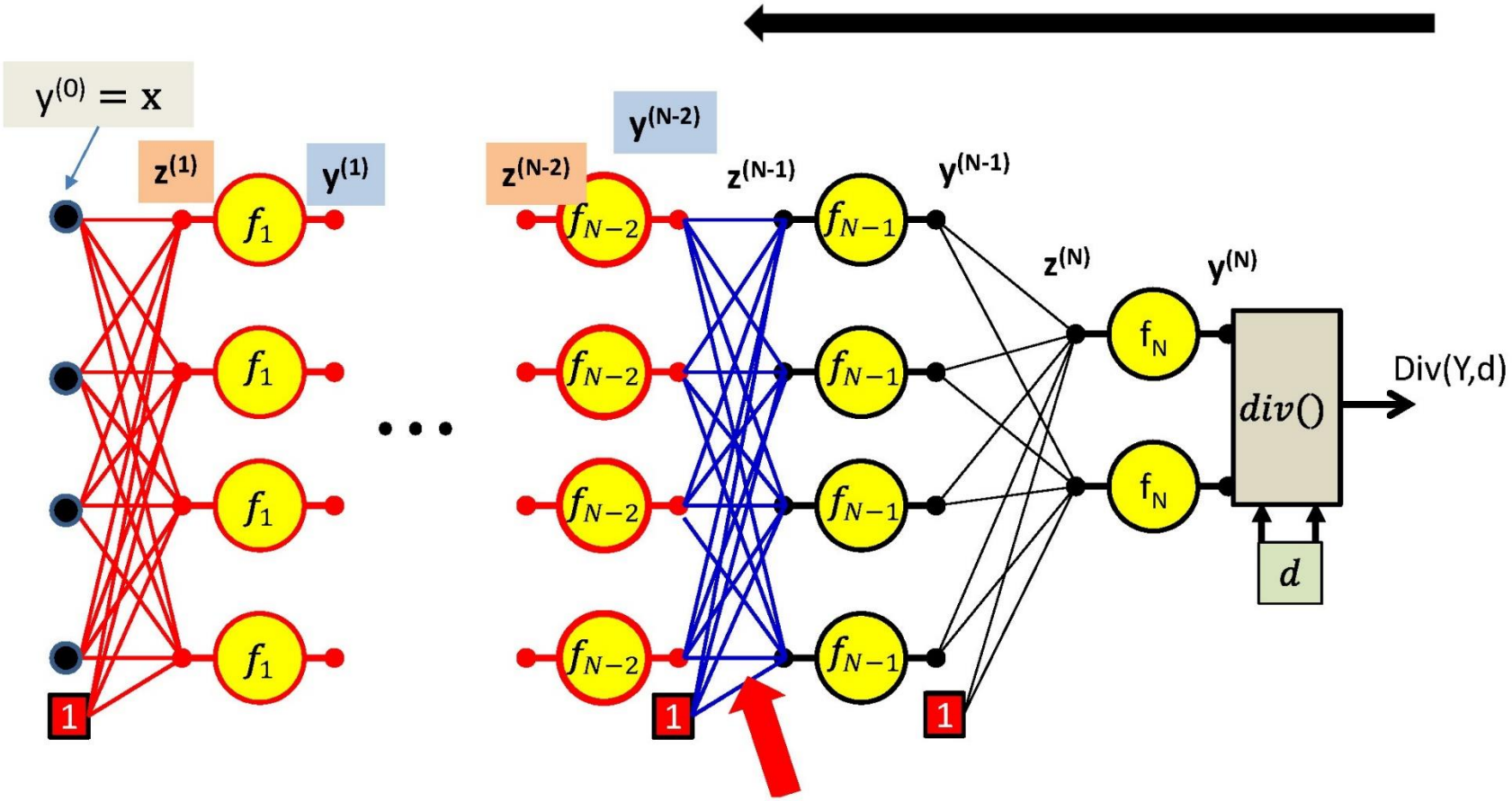
Backward Gradient Computation



We continue our way backwards in the order shown

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

Backward Gradient Computation

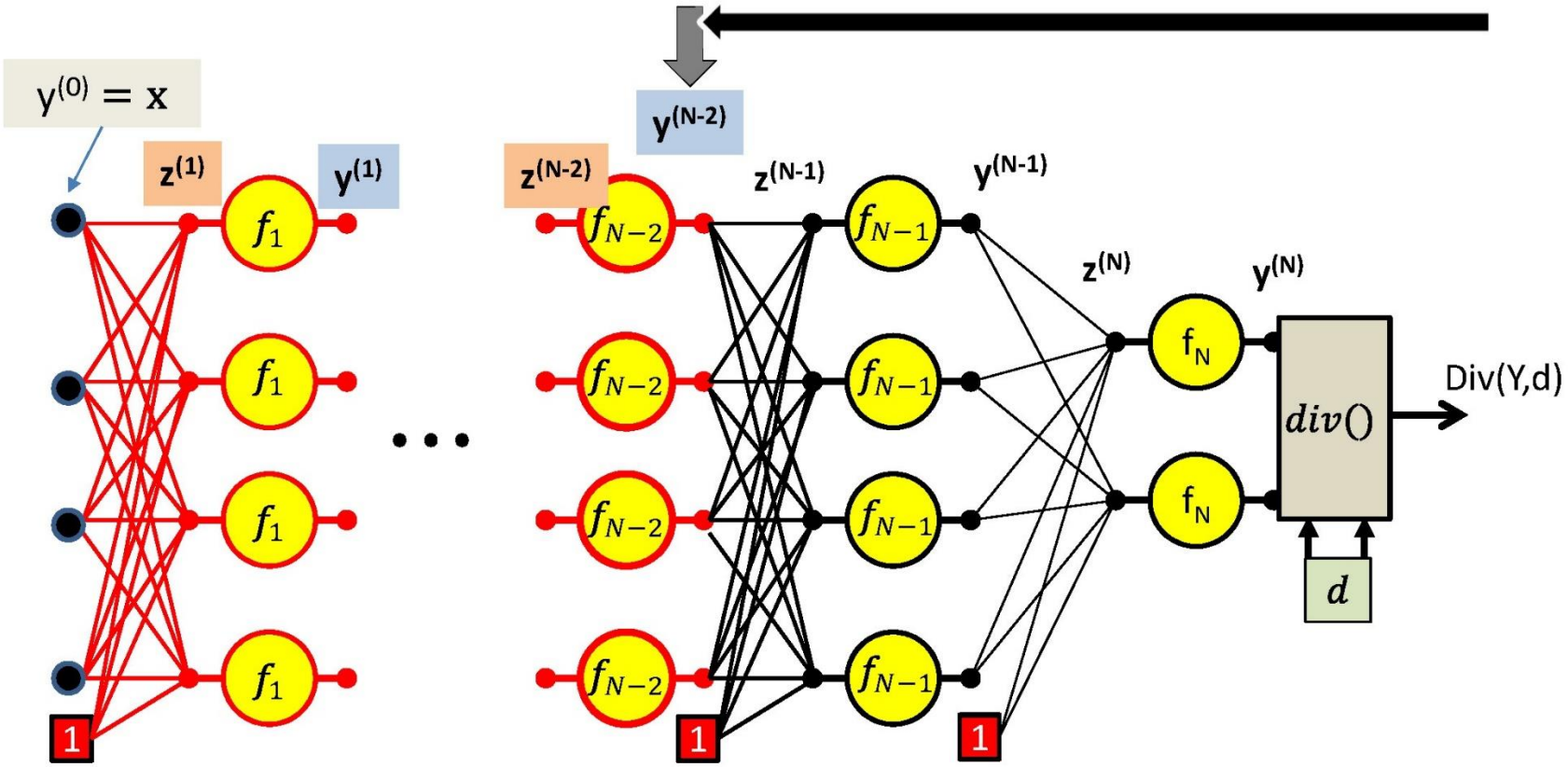


$$\frac{\partial Div}{\partial w_{ij}^{(N-1)}} = y_i^{(N-2)} \frac{\partial Div}{\partial z_j^{(N-1)}}$$

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

We continue our way backwards in the order shown

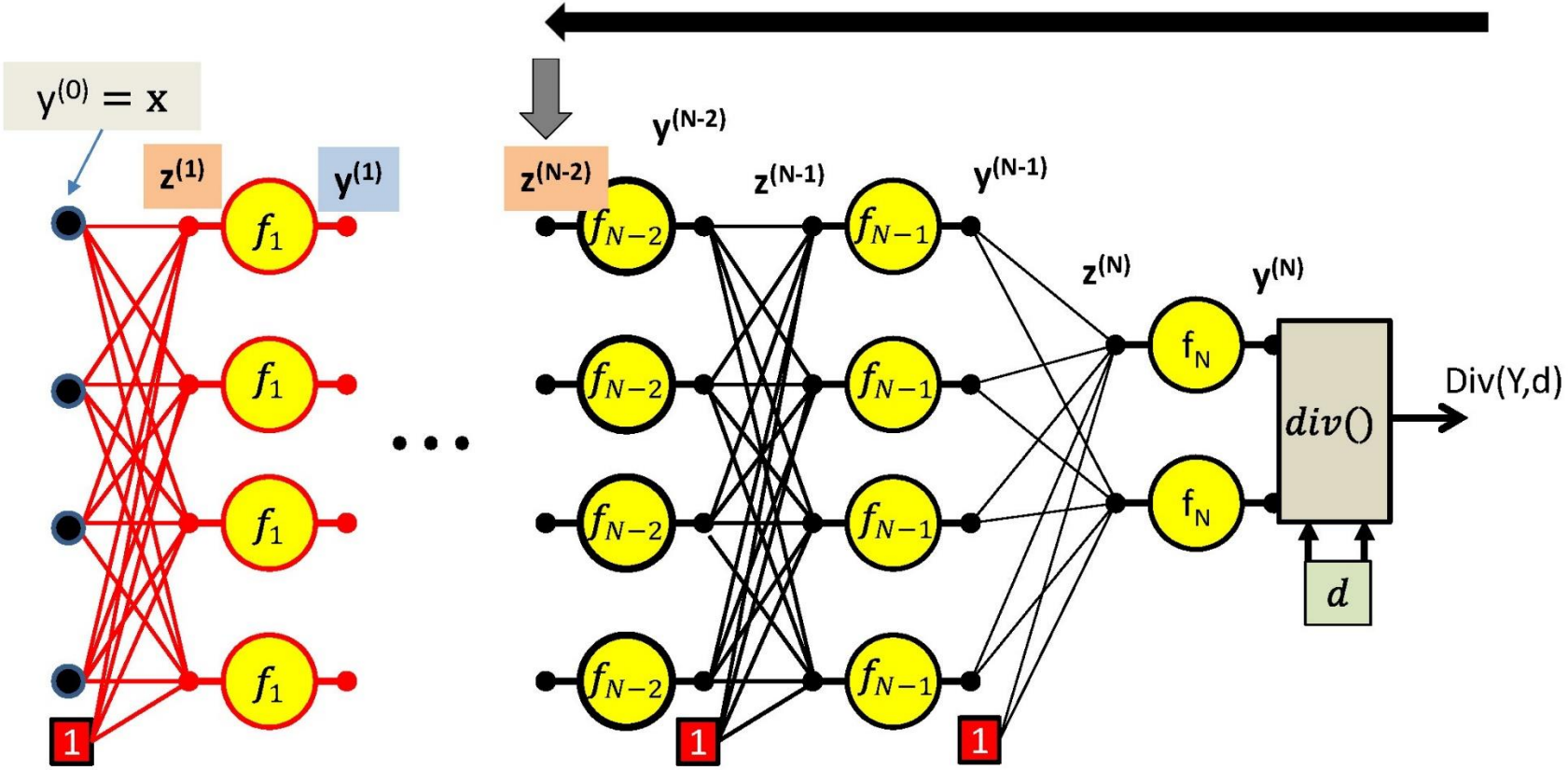
Backward Gradient Computation



We continue our way backwards in the order shown

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

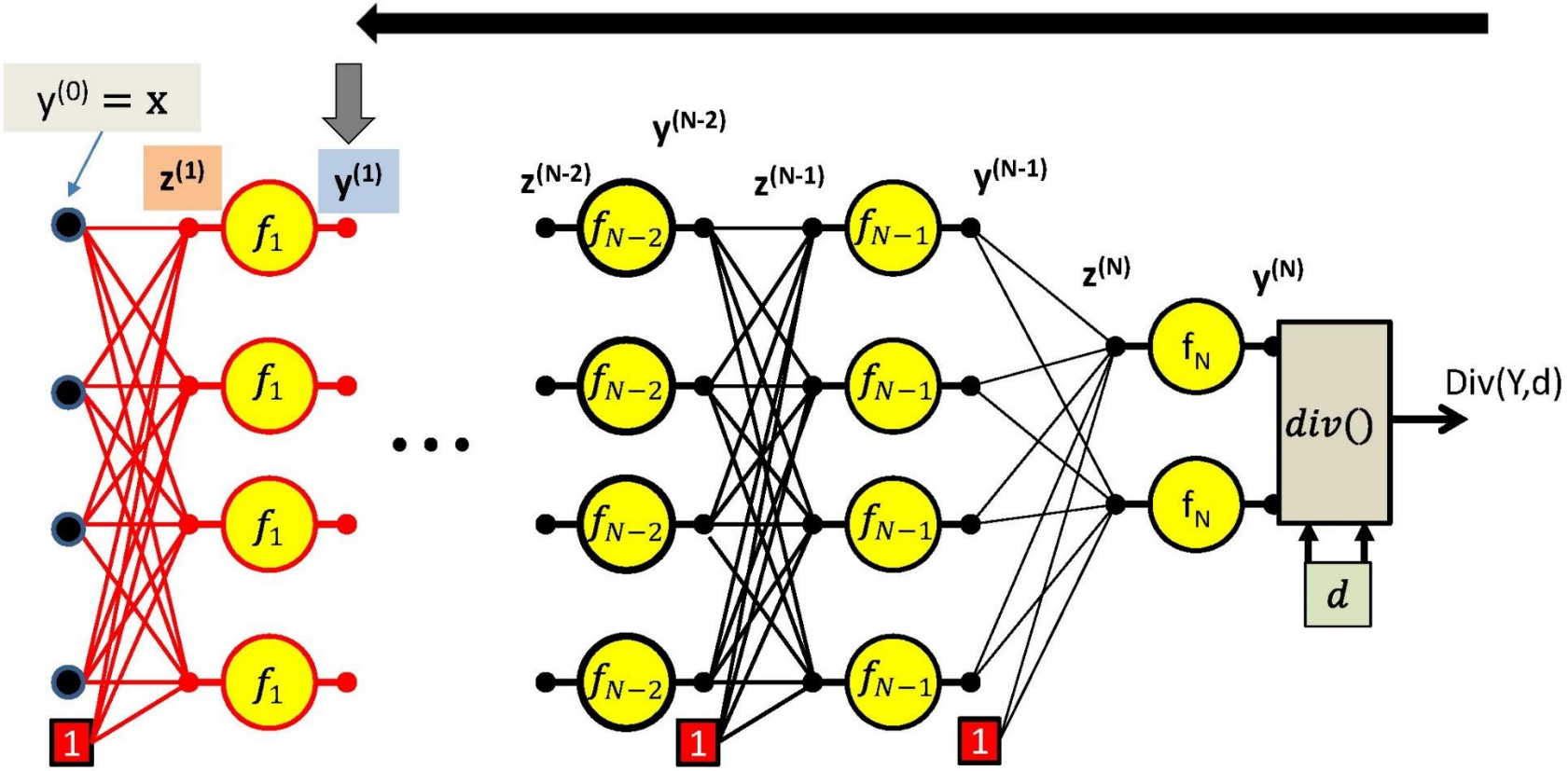
Backward Gradient Computation



We continue our way backwards in the order shown

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

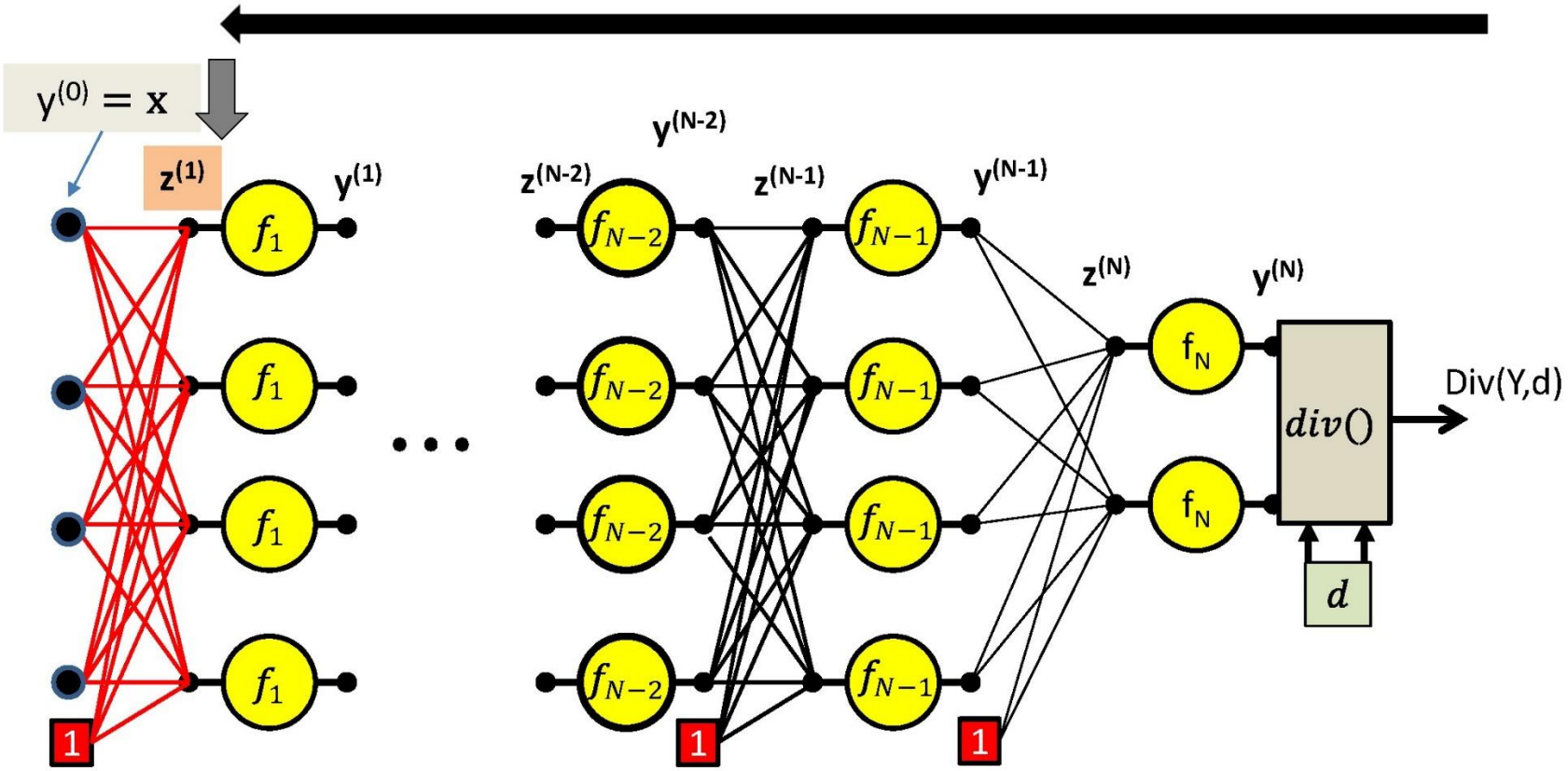
Backward Gradient Computation



We continue our way backwards in the order shown

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

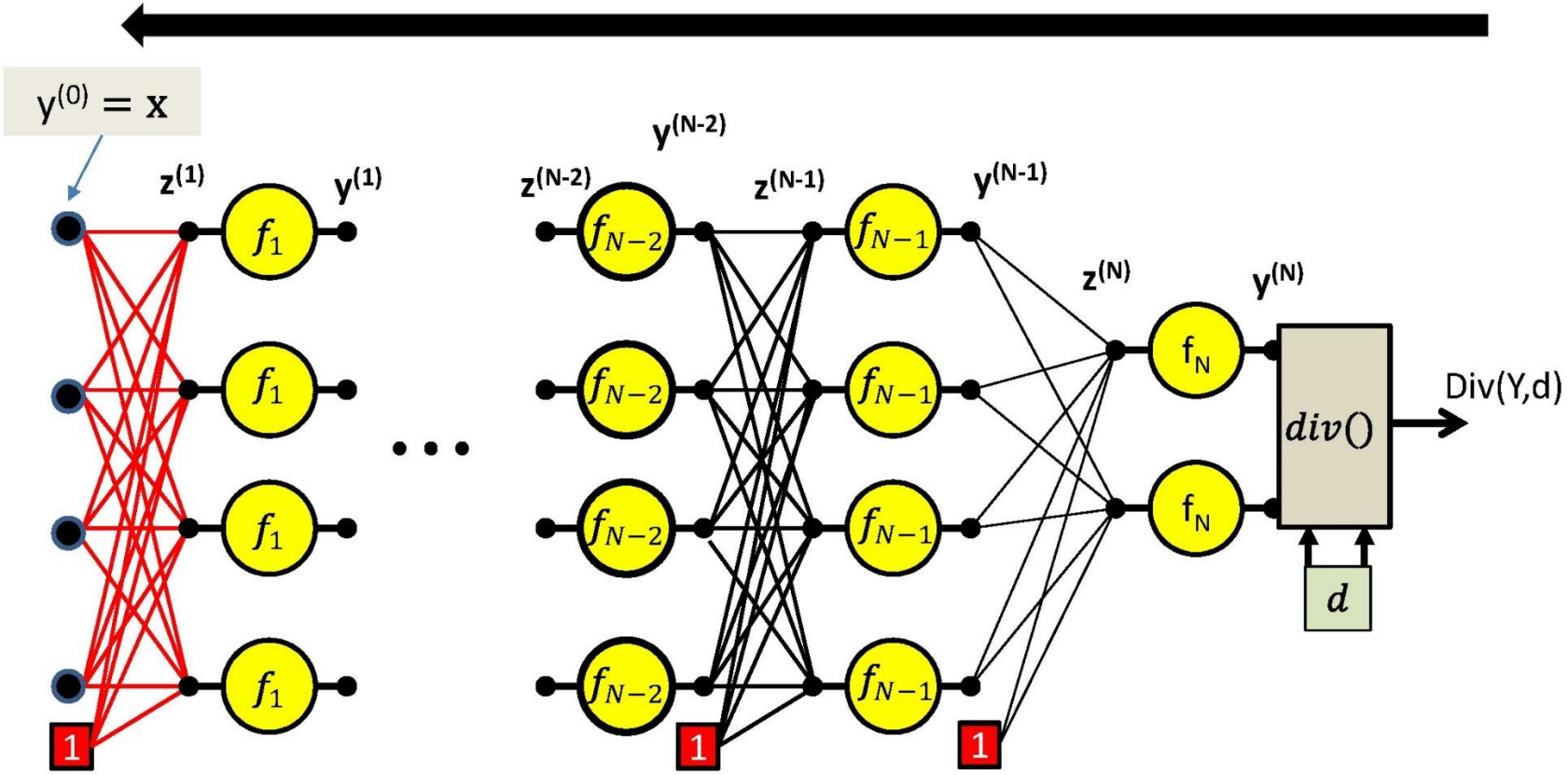
Backward Gradient Computation



We continue our way backwards in the order shown

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

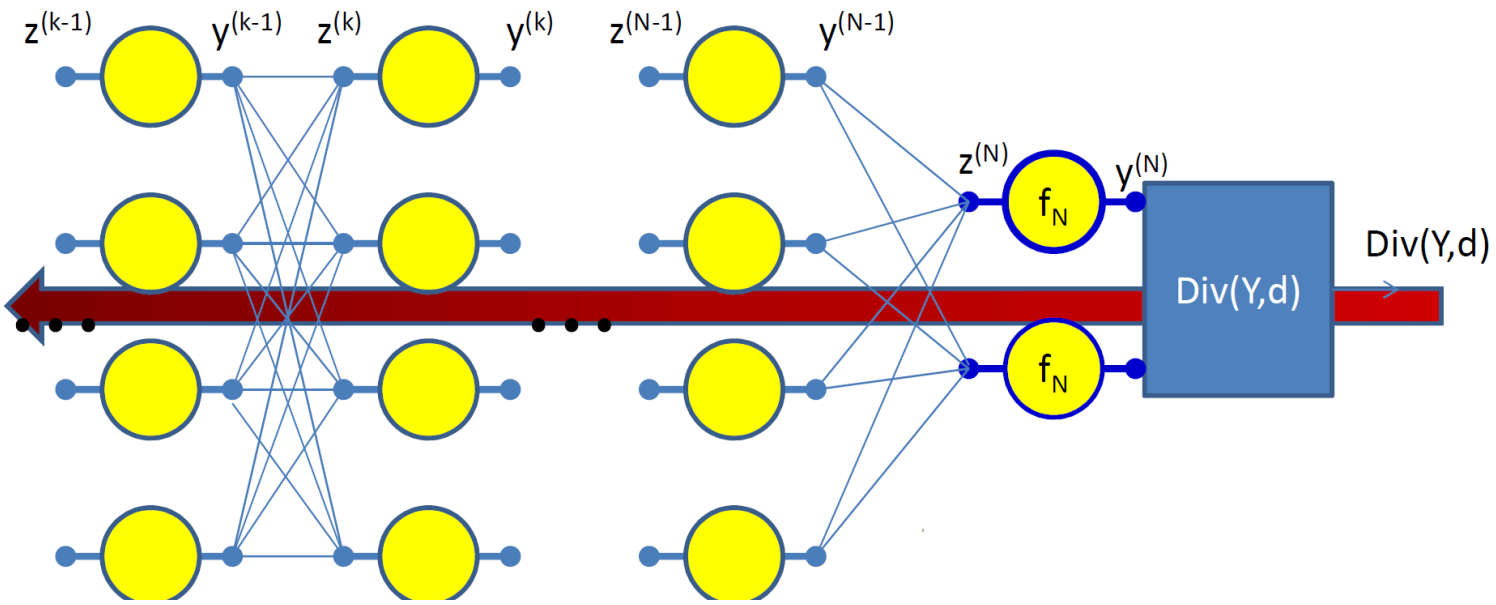
Backward Gradient Computation



We continue our way backwards in the order shown

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

Gradients: Backward Computation



Initialize: Gradient w.r.t network output

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

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

For k= N-1 . . . 0

For i= 1 . . . Layer width

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

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

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

Backward Pass

Output layer (N) :

For $i = 1 \ldots D_N$

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

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

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

Very analogous to the forward pass:

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

For $i = 1 \ldots D_k$

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

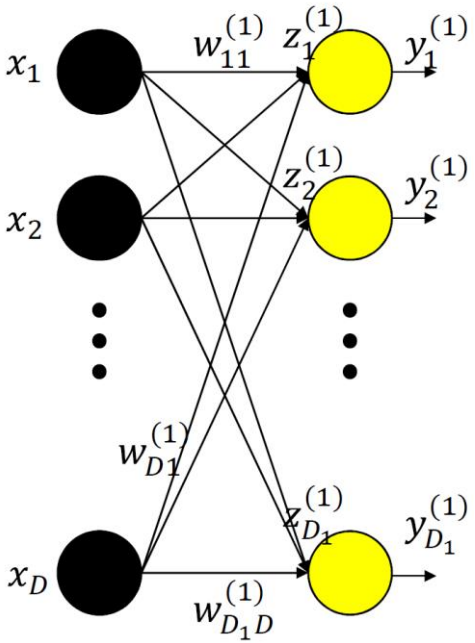
Backward weighted combination of next layer

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

Backward equivalent of activation

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

Vector formulation



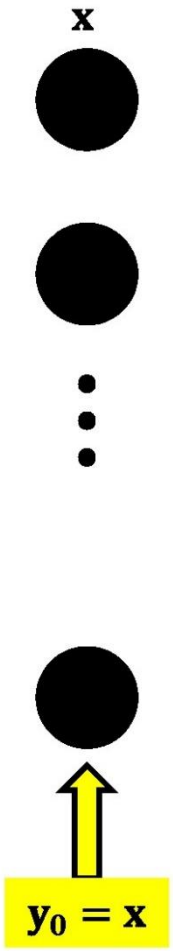
$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix} \quad \mathbf{z_k} = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{D_k}^{(k)} \end{bmatrix} \quad \mathbf{y_k} = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_{D_k}^{(k)} \end{bmatrix}$$

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

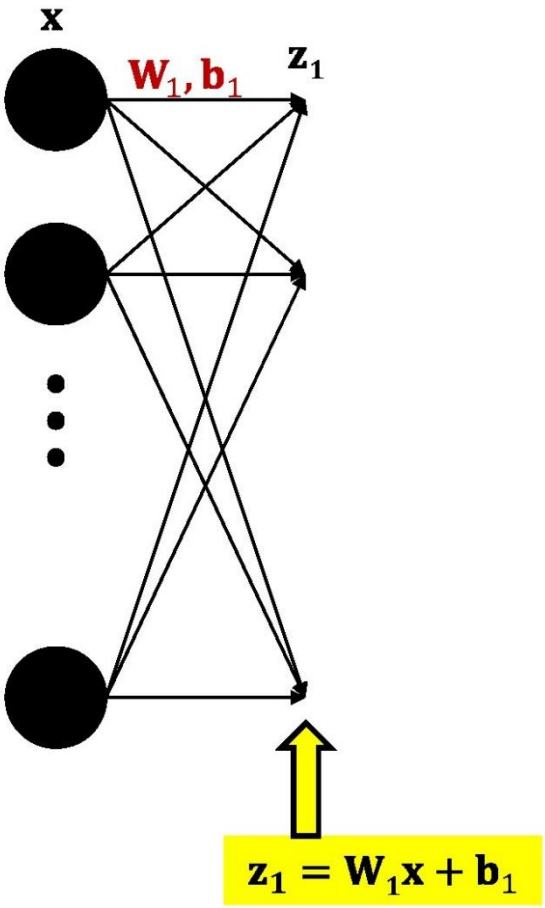
- Arrange all inputs to the network in a vector X
- Arrange the inputs to neurons of the kth layer as a vector Z_k
- Arrange the outputs of neurons in the kth layer as a vector y_k
- Arrange the weights to any layer as a matrix W_k (Similarly with biases)
- The computation of a single layer is easily expressed in matrix notation as

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

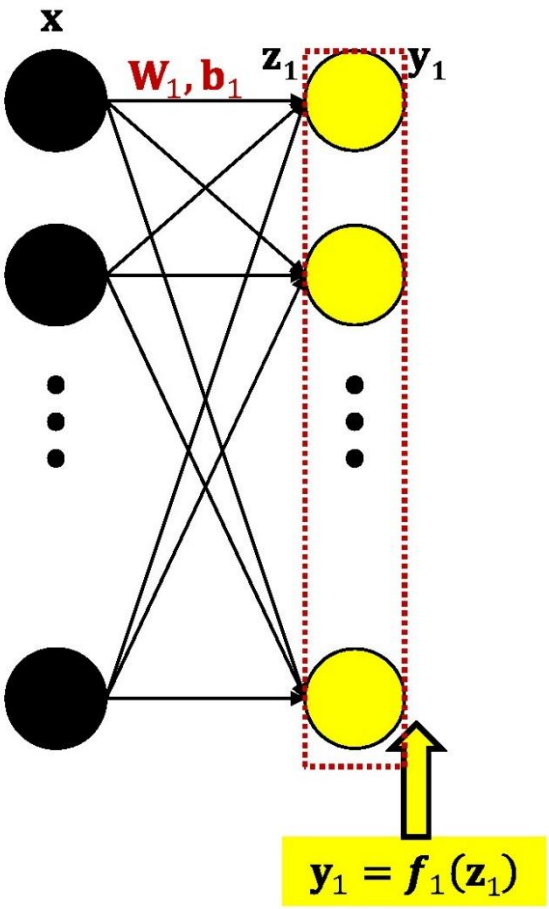
The forward pass: Evaluating the network



The forward pass



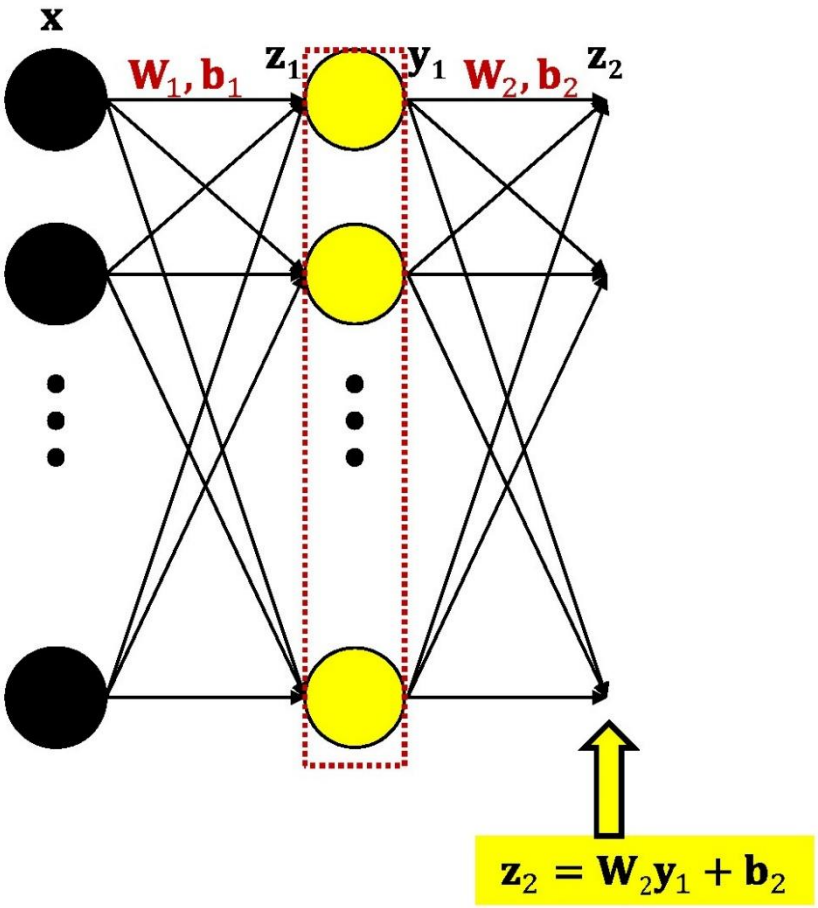
The forward pass



The Complete computation

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

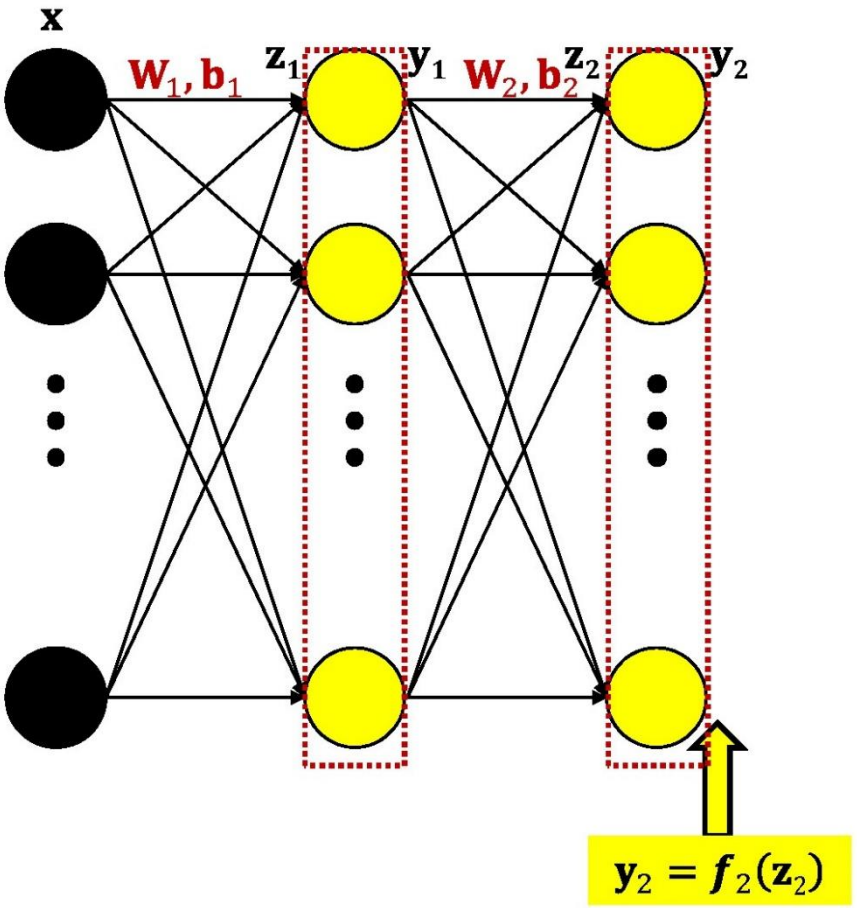
The forward pass



The Complete computation

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

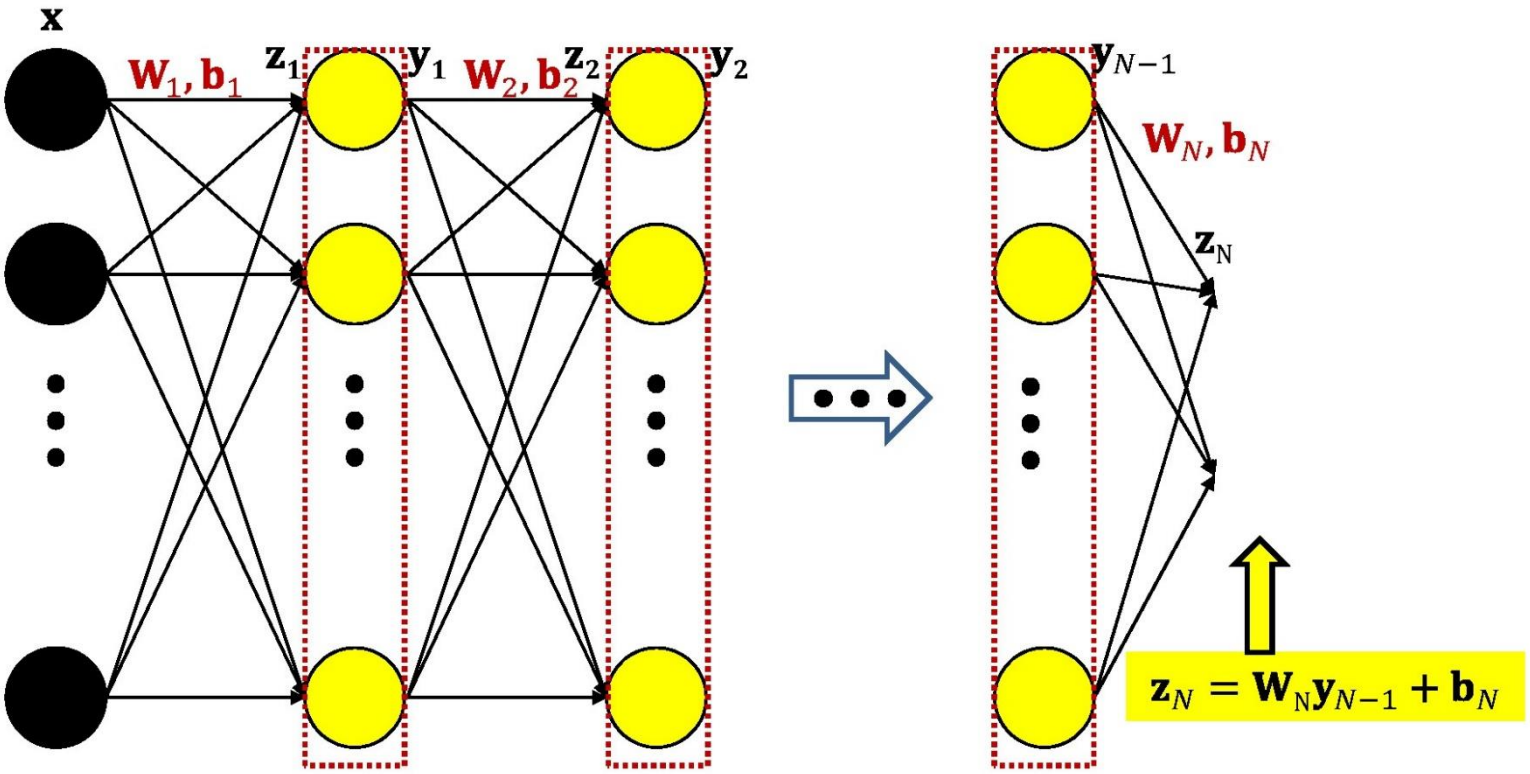
The forward pass



The Complete computation

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

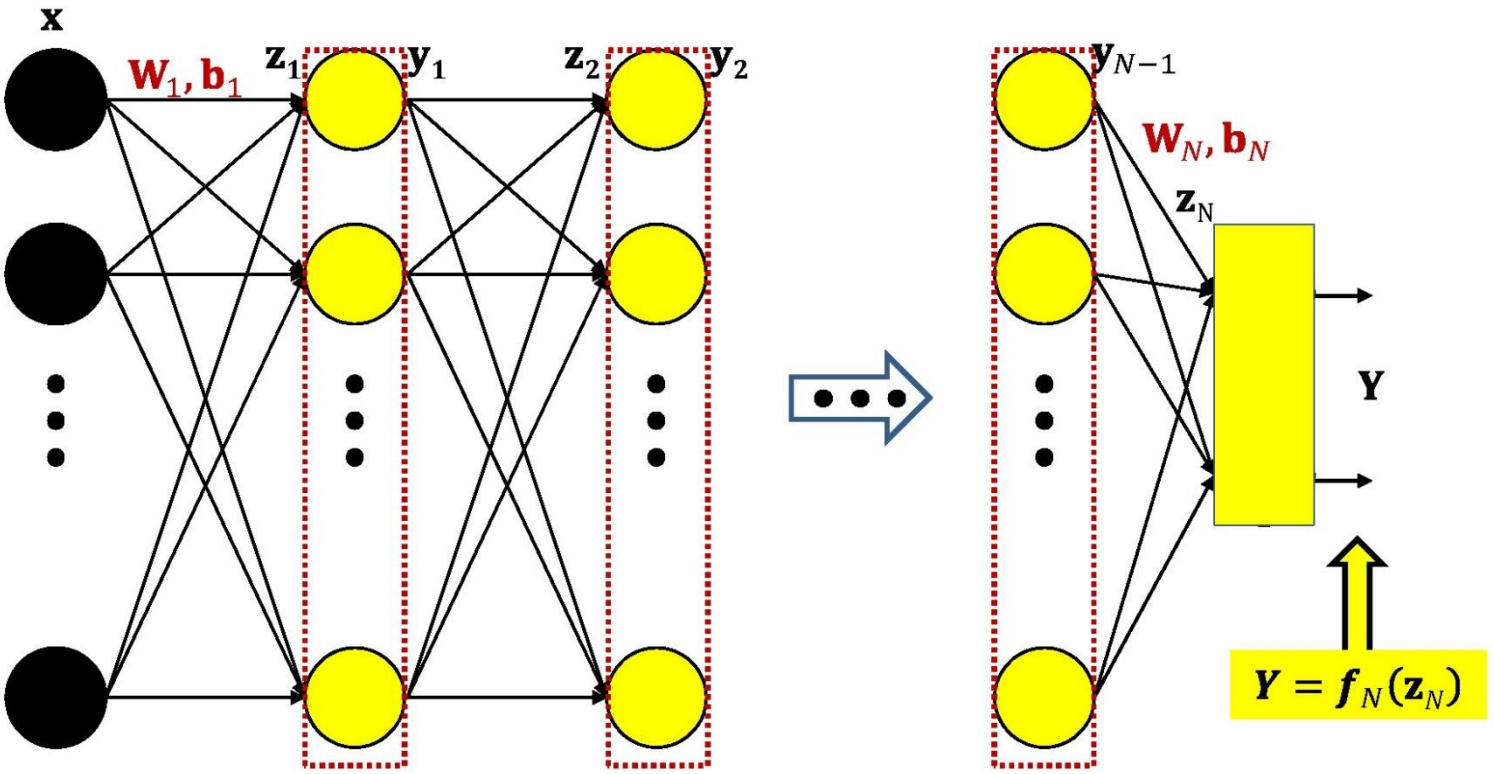
The forward pass



The Complete computation

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

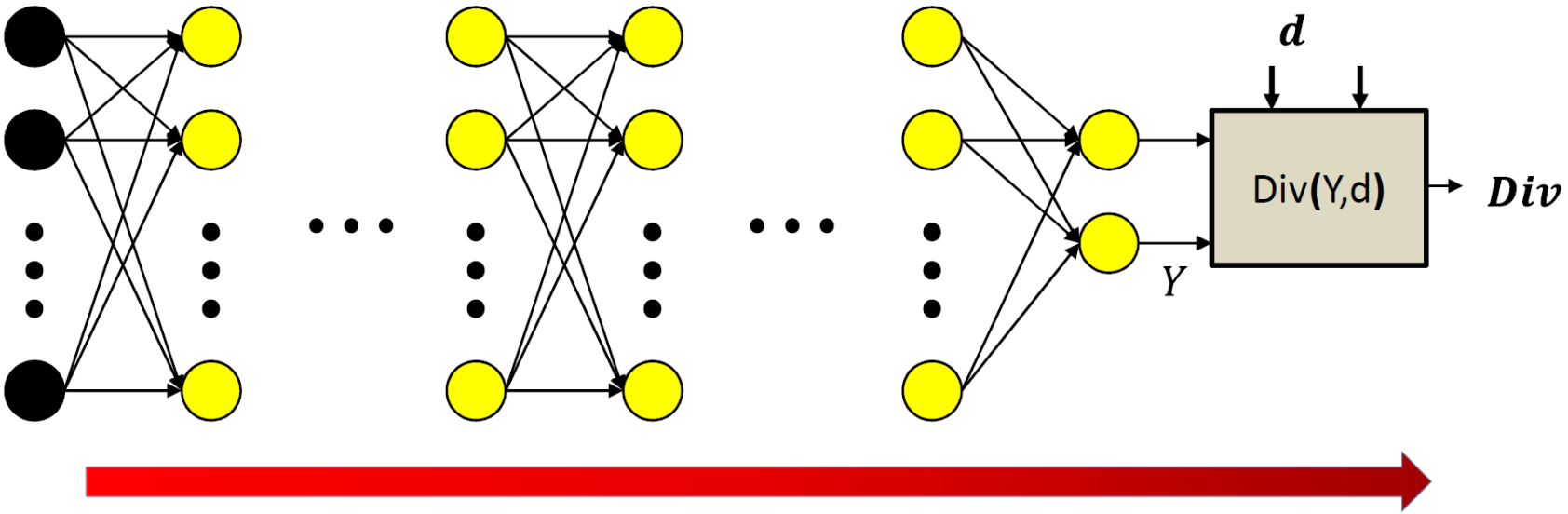
The forward pass



The Complete computation

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

The forward pass

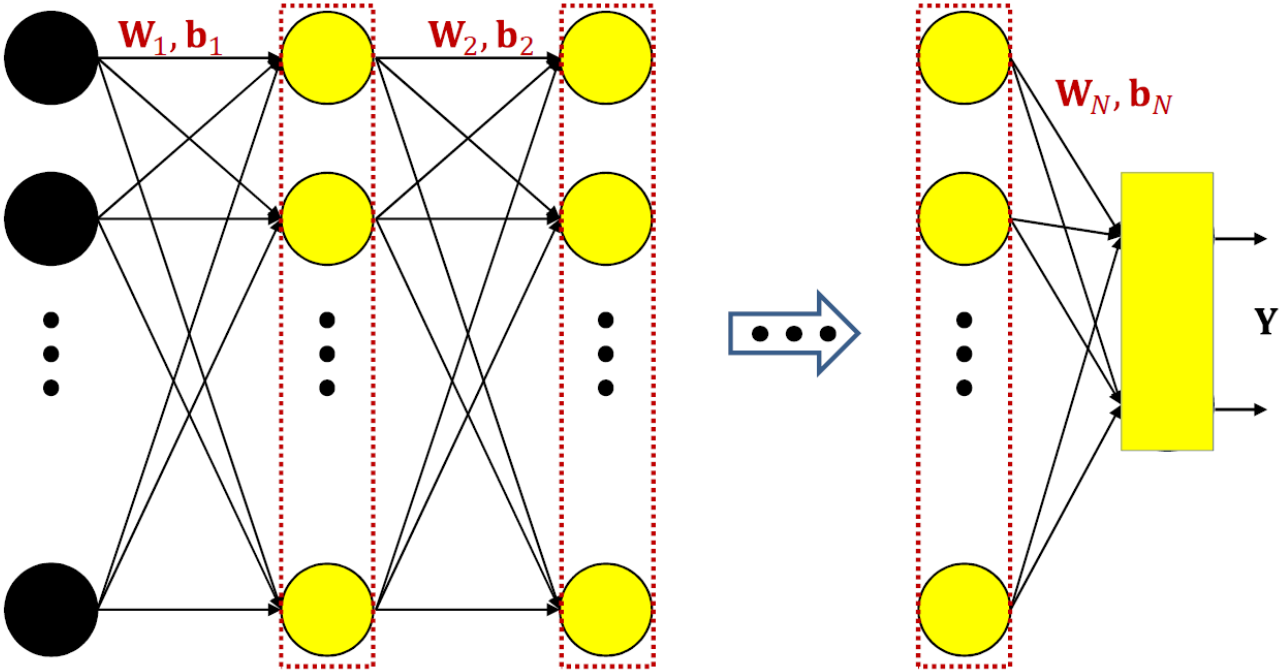


- Forward pass:
  - Initialize $\mathbf{y}_0 = \mathbf{x}$
  - For k = 1 to N: (Recursion)

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

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

The backward pass



- The network is a nested function

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

- The error for any is also a nested function

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

The Jacobian

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

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

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

- Using vector notation

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

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

Special case: Affine functions

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

- Matrix **W** and bias **b** operating on vector **y** to produce vector
  - The Jacobian of **z** w.r.t **y** is simply the matrix **W**
- Chain rule for **Jacobians**

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

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

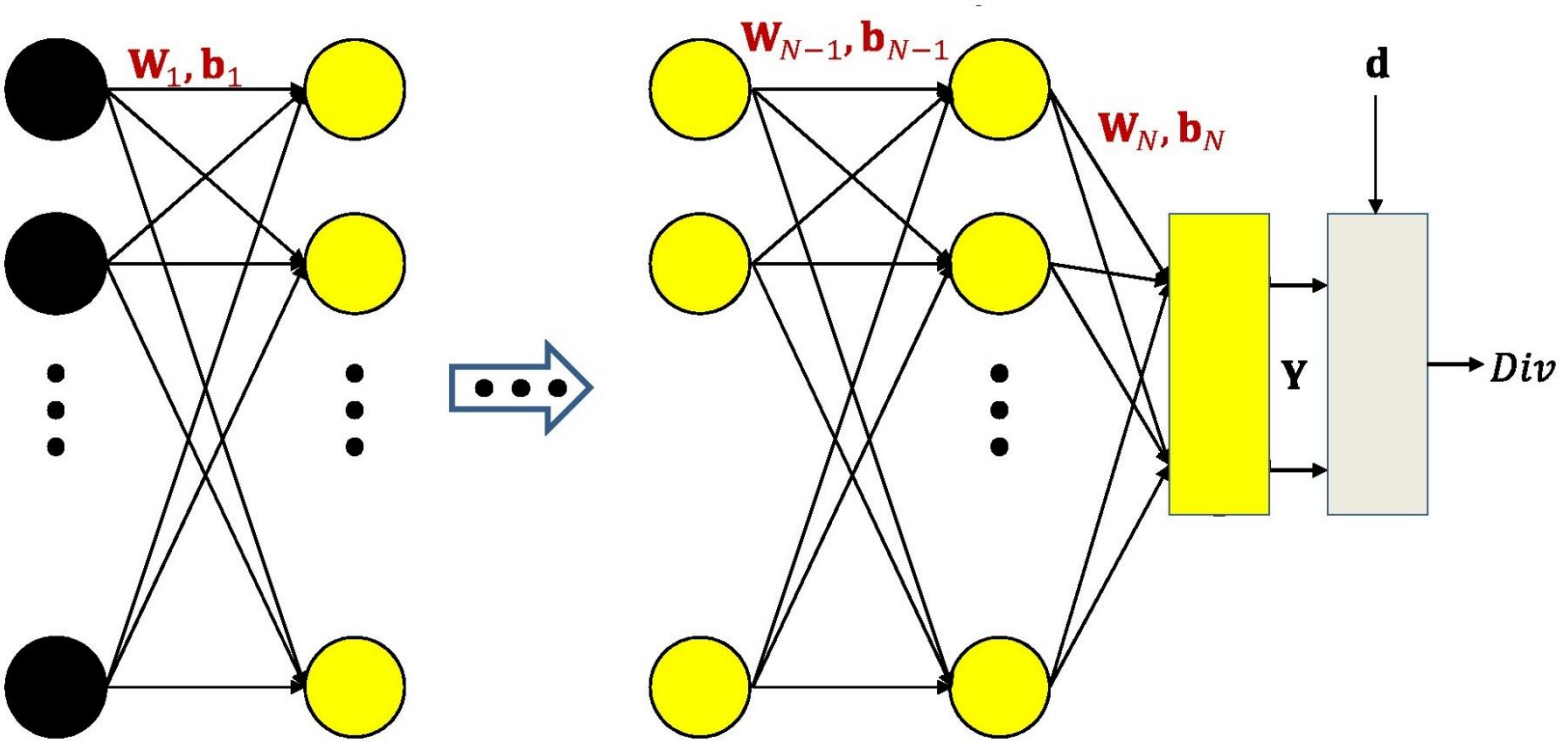
$$D = f(\mathbf{z})$$

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

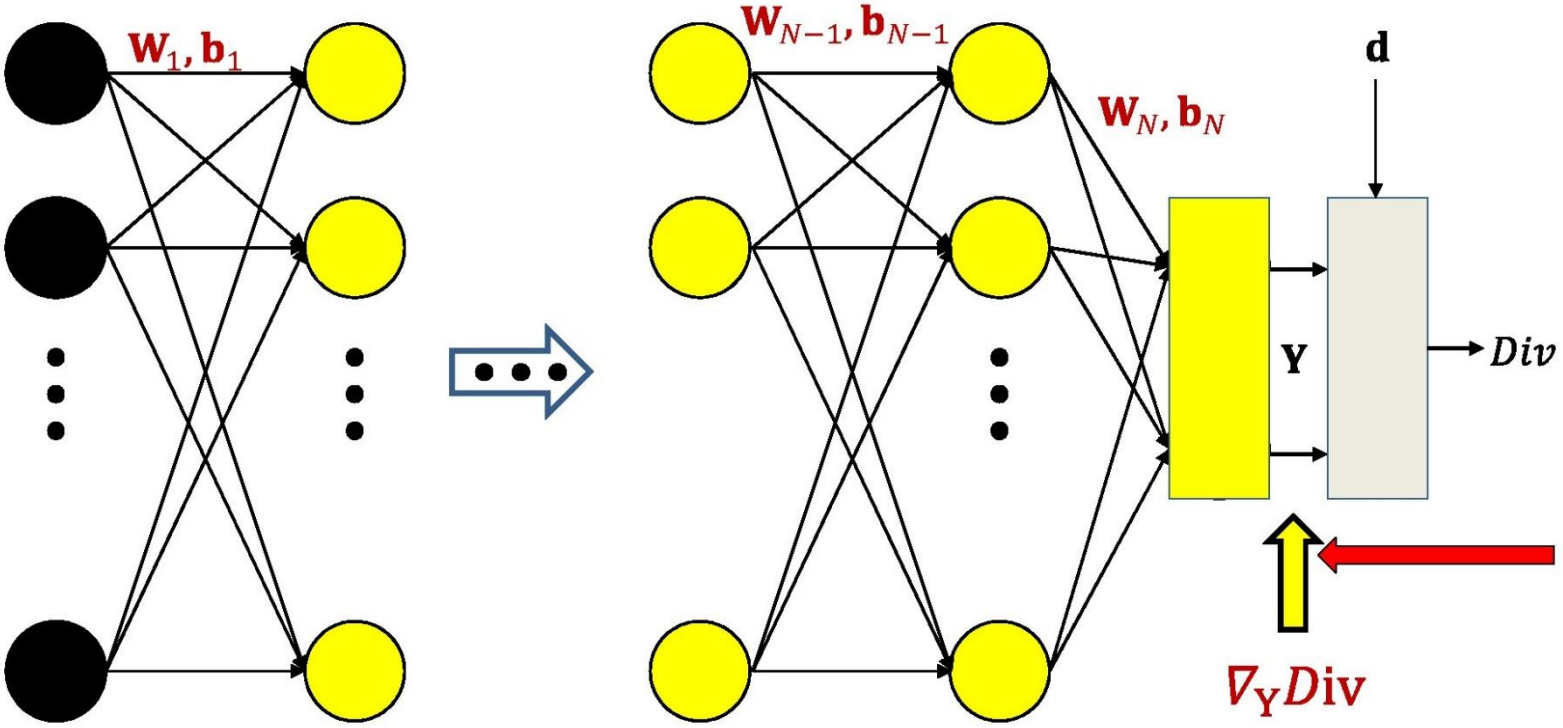
$$\Delta\mathbf{z} = J_{\mathbf{z}}(\mathbf{x})\Delta\mathbf{x}$$

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

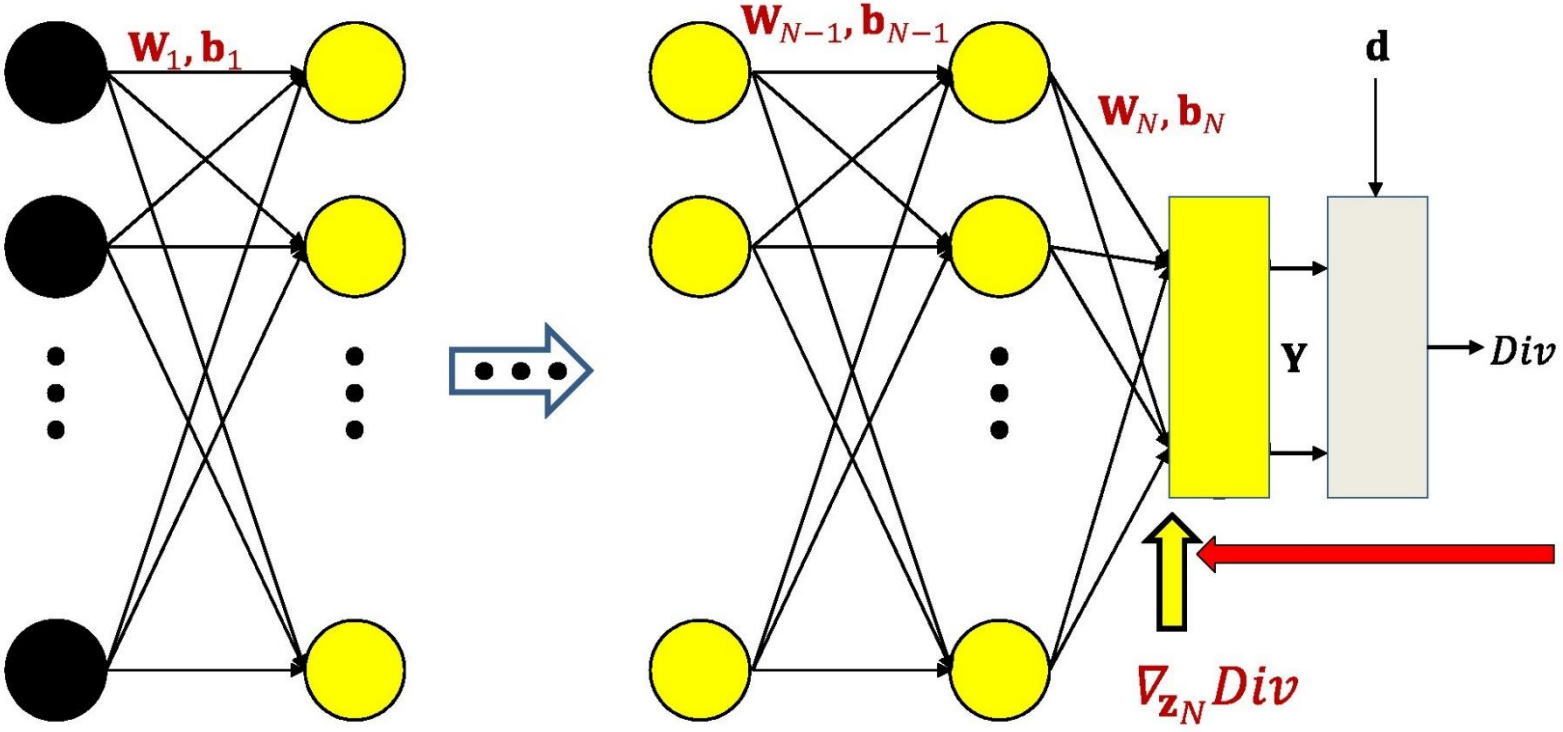
$$\Delta D = \nabla_{\mathbf{z}}(D) J_{\mathbf{z}}(\mathbf{x})\Delta\mathbf{x} = \nabla_{\mathbf{x}} D\,\Delta\mathbf{x}$$



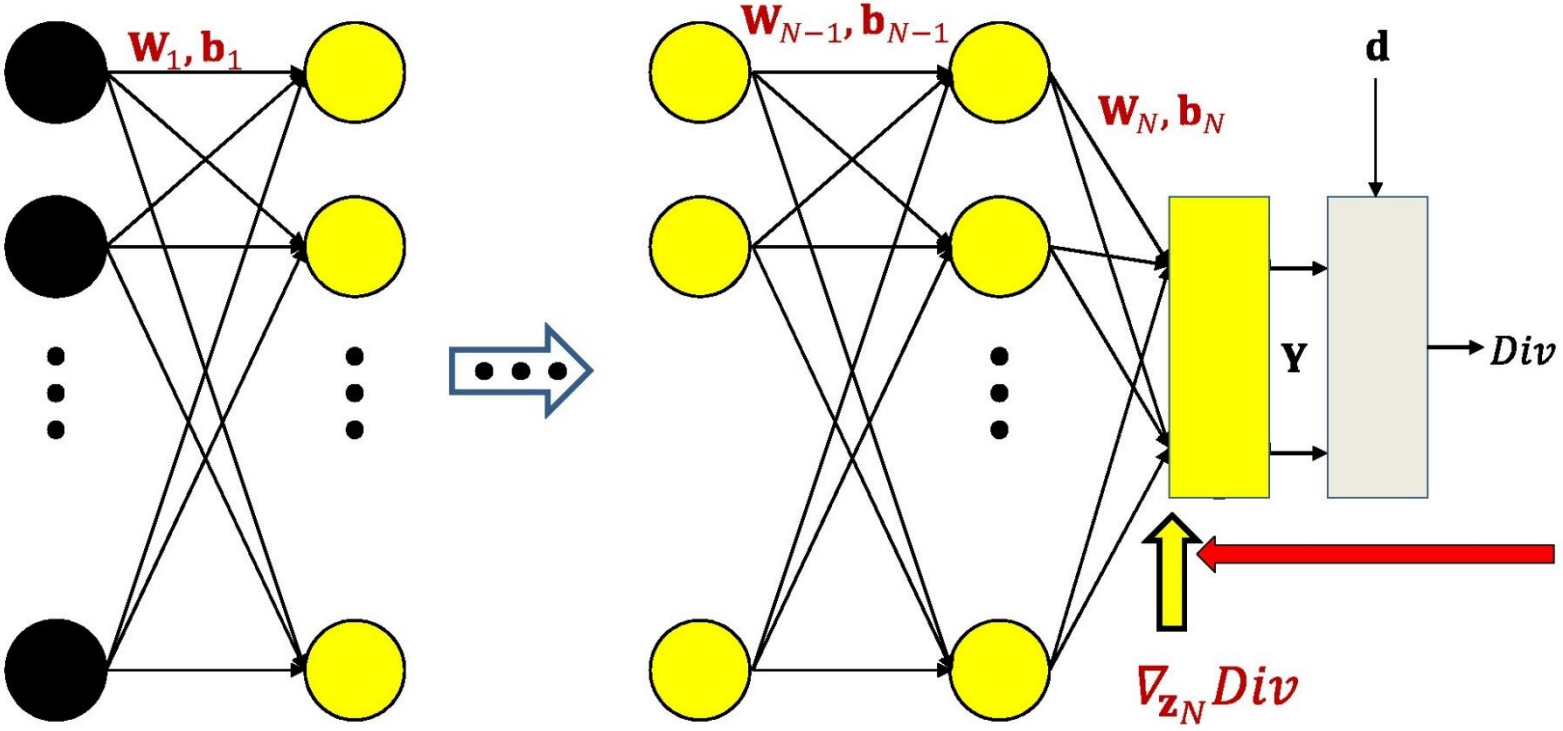
In general $\nabla_{\mathbf{a}}\mathbf{b}$ represents a derivative of **b** w.r.t. **a** and could be a gradient (for scalar **b**) Or a Jacobian (for vector **b**)



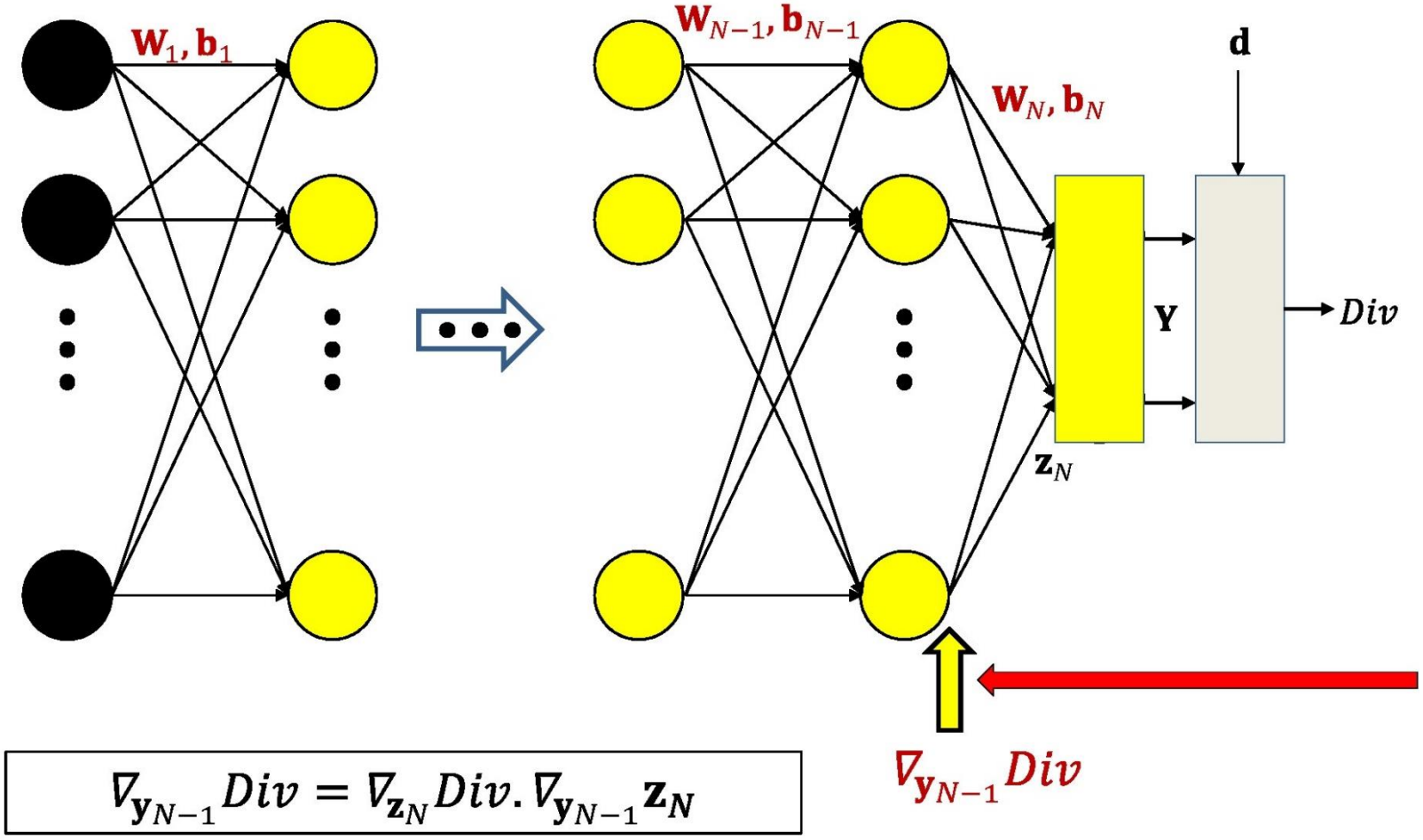
- First compute the gradient of the divergence w.r.t. **Y**
- The actual gradient depends on the divergence function



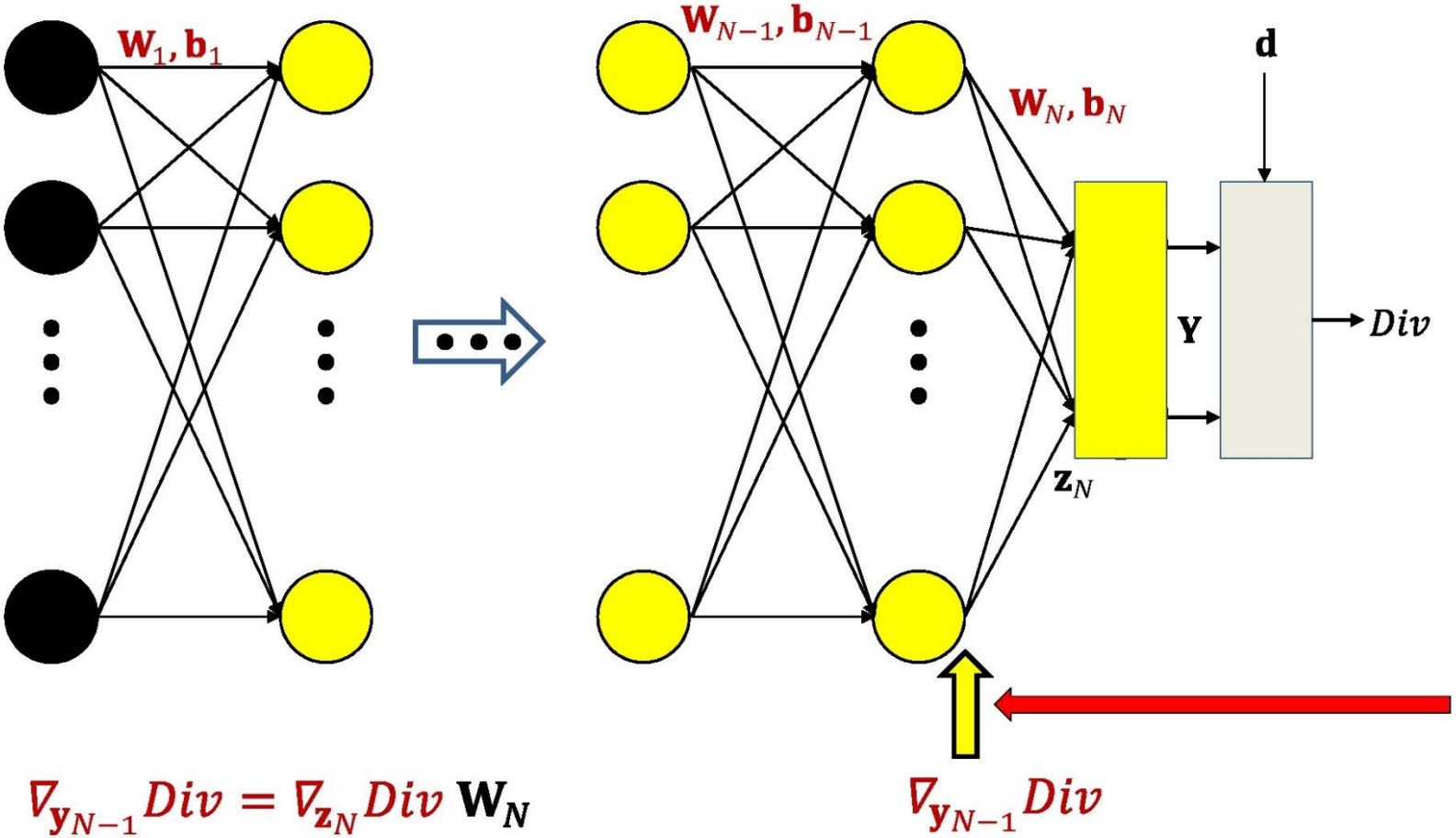
$$\nabla_{\mathbf{z}_N} Div = \nabla_{\mathbf{Y}} Div . \nabla_{\mathbf{z}_N} \mathbf{Y}$$

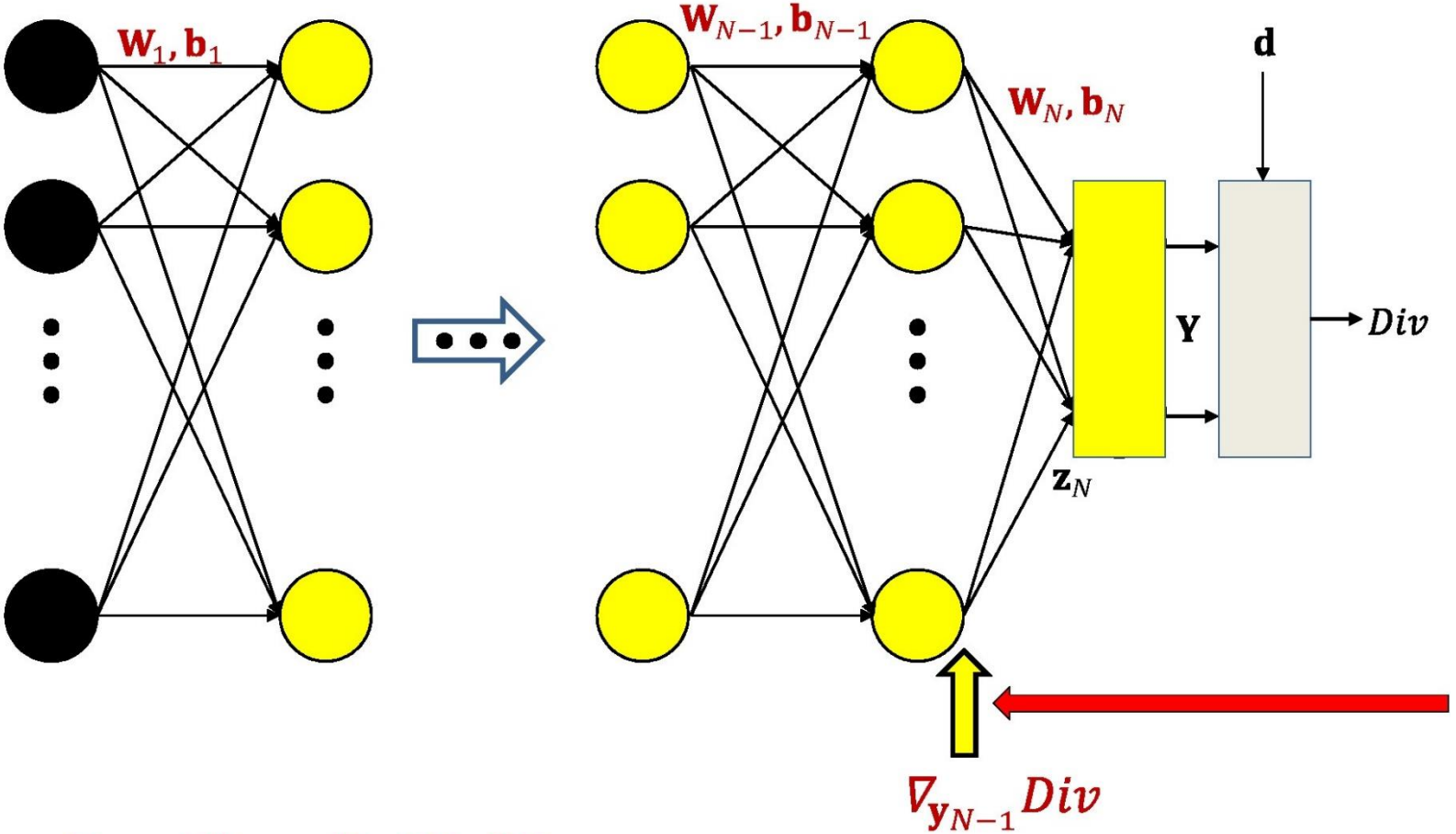


$$\nabla_{\mathbf{z}_N} Div = \nabla_{\mathbf{Y}} Div \, J_{\mathbf{Y}}(\mathbf{z}_N)$$



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

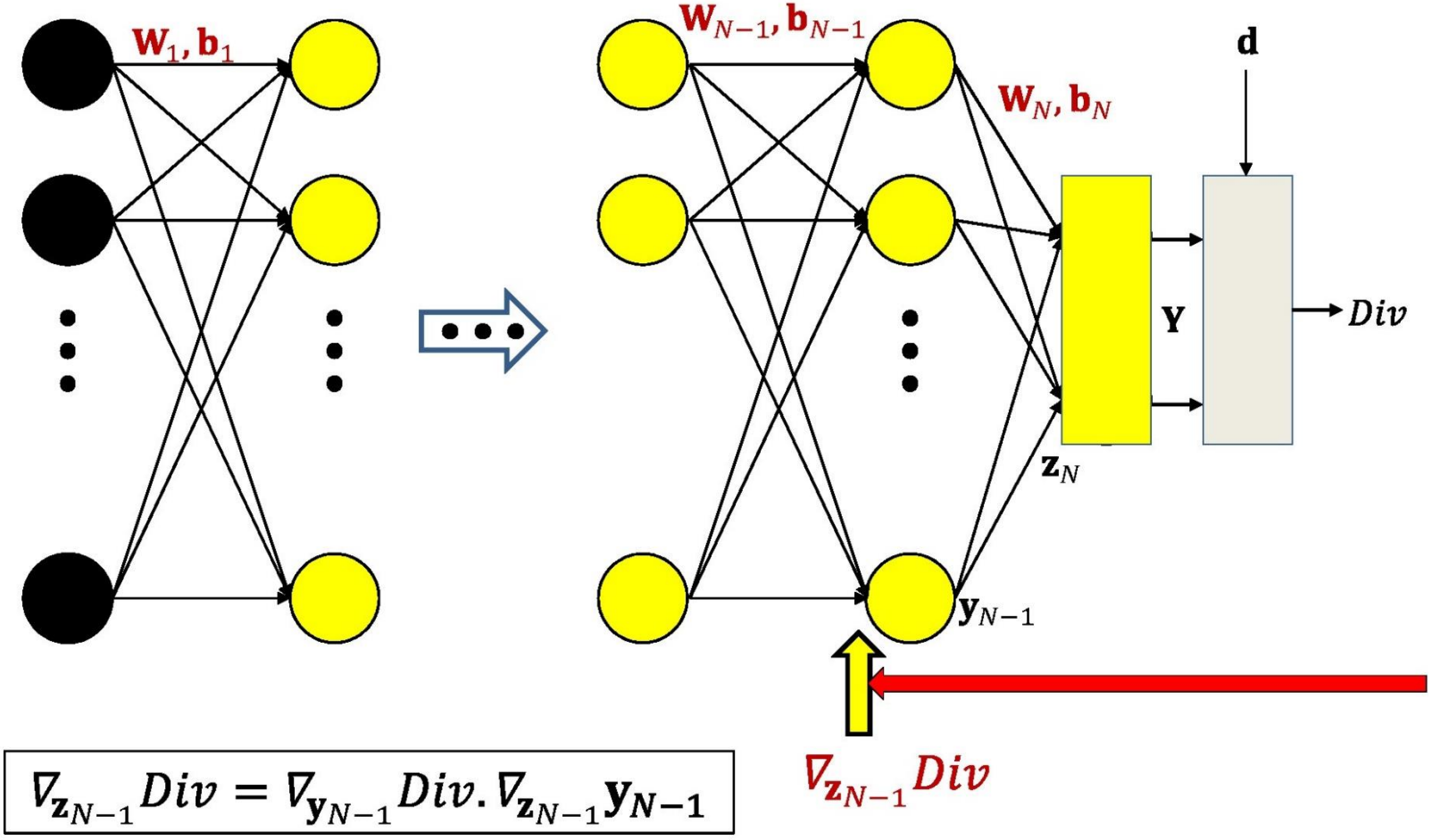

$\mathbf{W}_1, \mathbf{b}_1$
$\mathbf{W}_{N-1}, \mathbf{b}_{N-1}$
$\mathbf{W}_N, \mathbf{b}_N$
$\mathbf{d}$
$\mathbf{Y}$
Div
$\mathbf{z}_N$
$\nabla_{\mathbf{y}_{N-1}} Div = \nabla_{\mathbf{z}_N} Div \, \mathbf{W}_N$
$\nabla_{\mathbf{y}_{N-1}} Div$



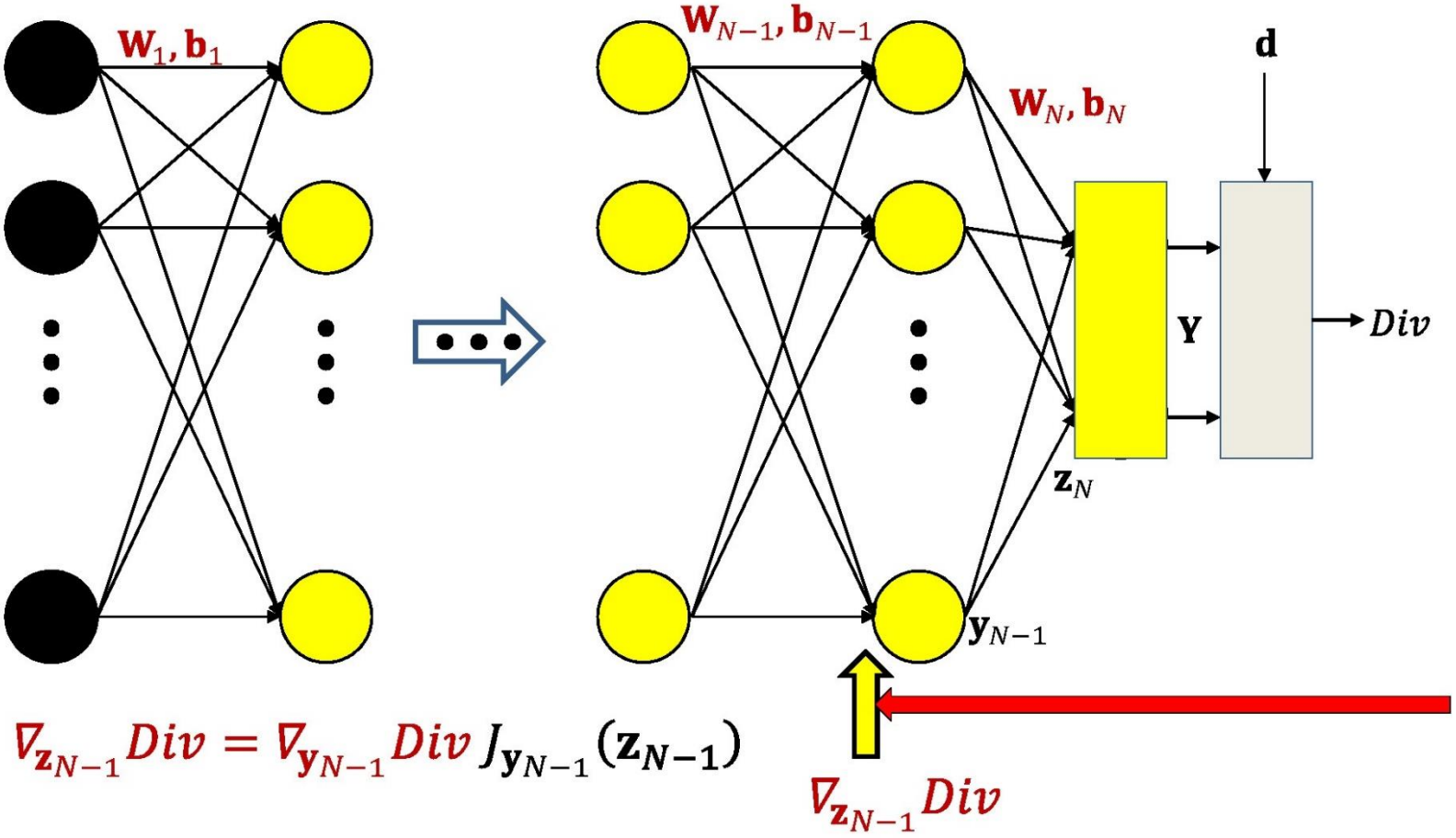
$$\nabla_{\mathbf{y}_{N-1}} Div = \nabla_{\mathbf{z}_N} Div \, \mathbf{W}_N$$

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

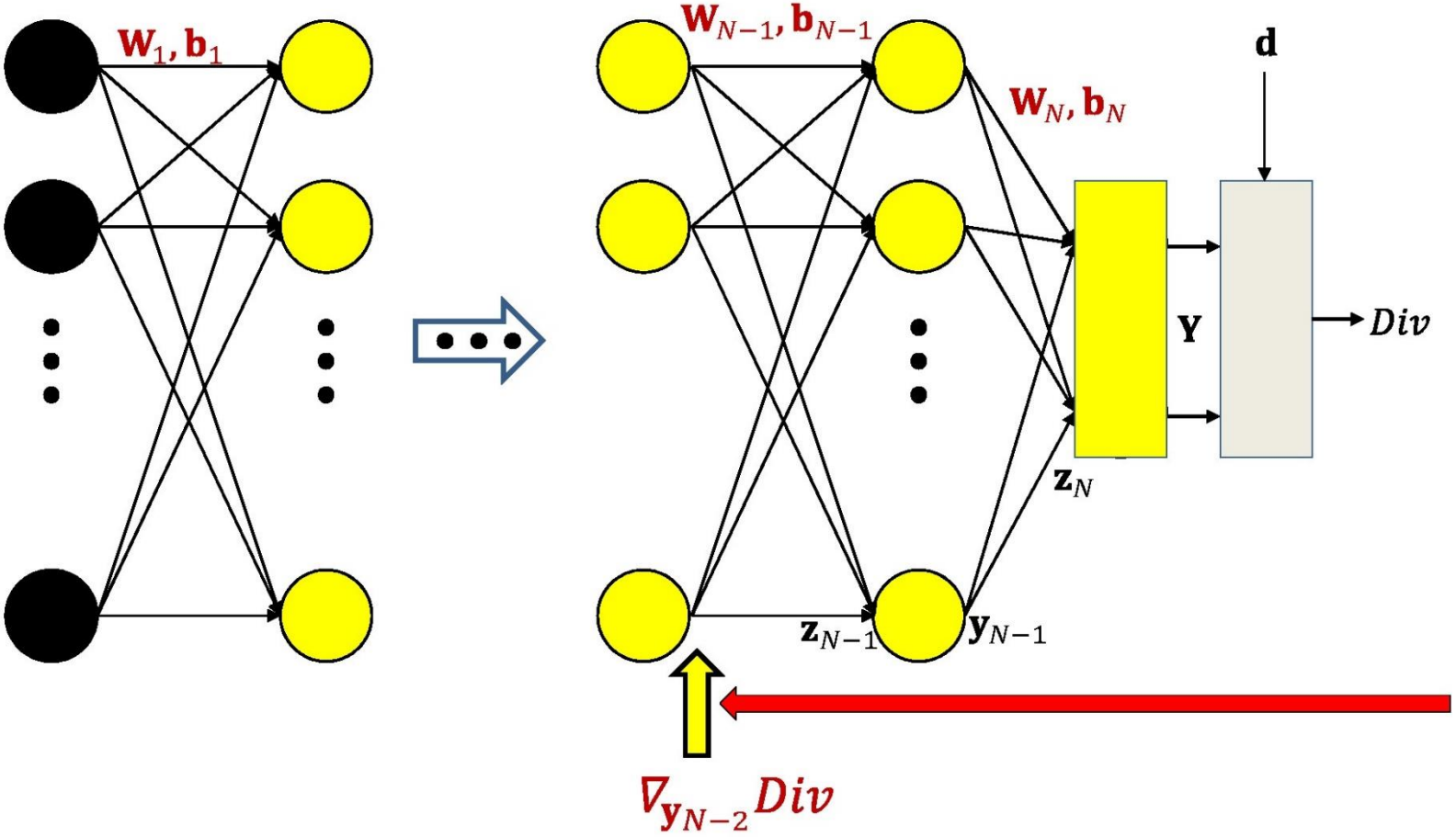
$$\nabla_{\mathbf{b}_N} Div = \nabla_{\mathbf{z}_N} Div$$



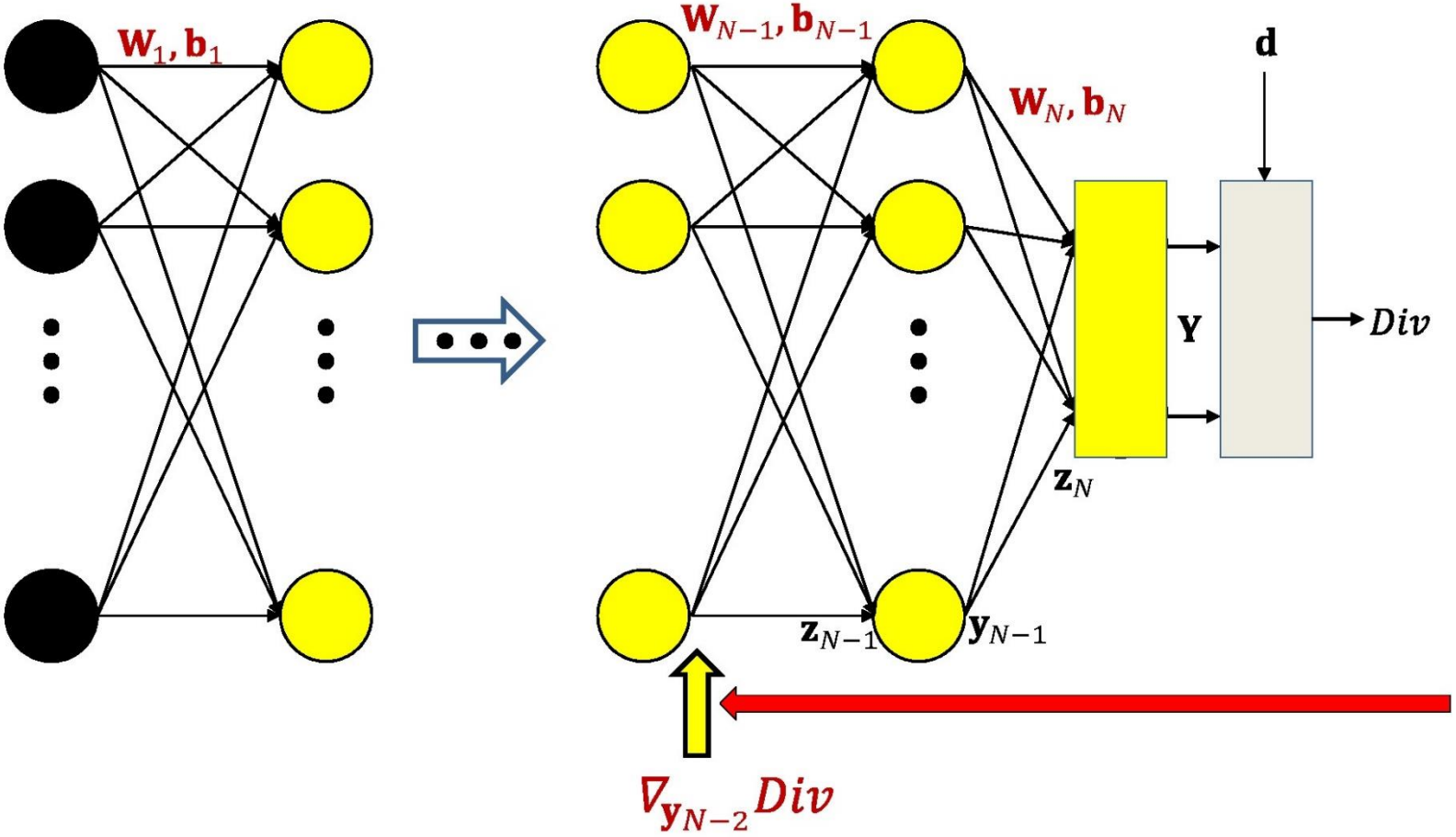
$$\nabla_{\mathbf{z}_{N-1}} Div = \nabla_{\mathbf{y}_{N-1}} Div . \nabla_{\mathbf{z}_{N-1}} \mathbf{y}_{N-1}$$



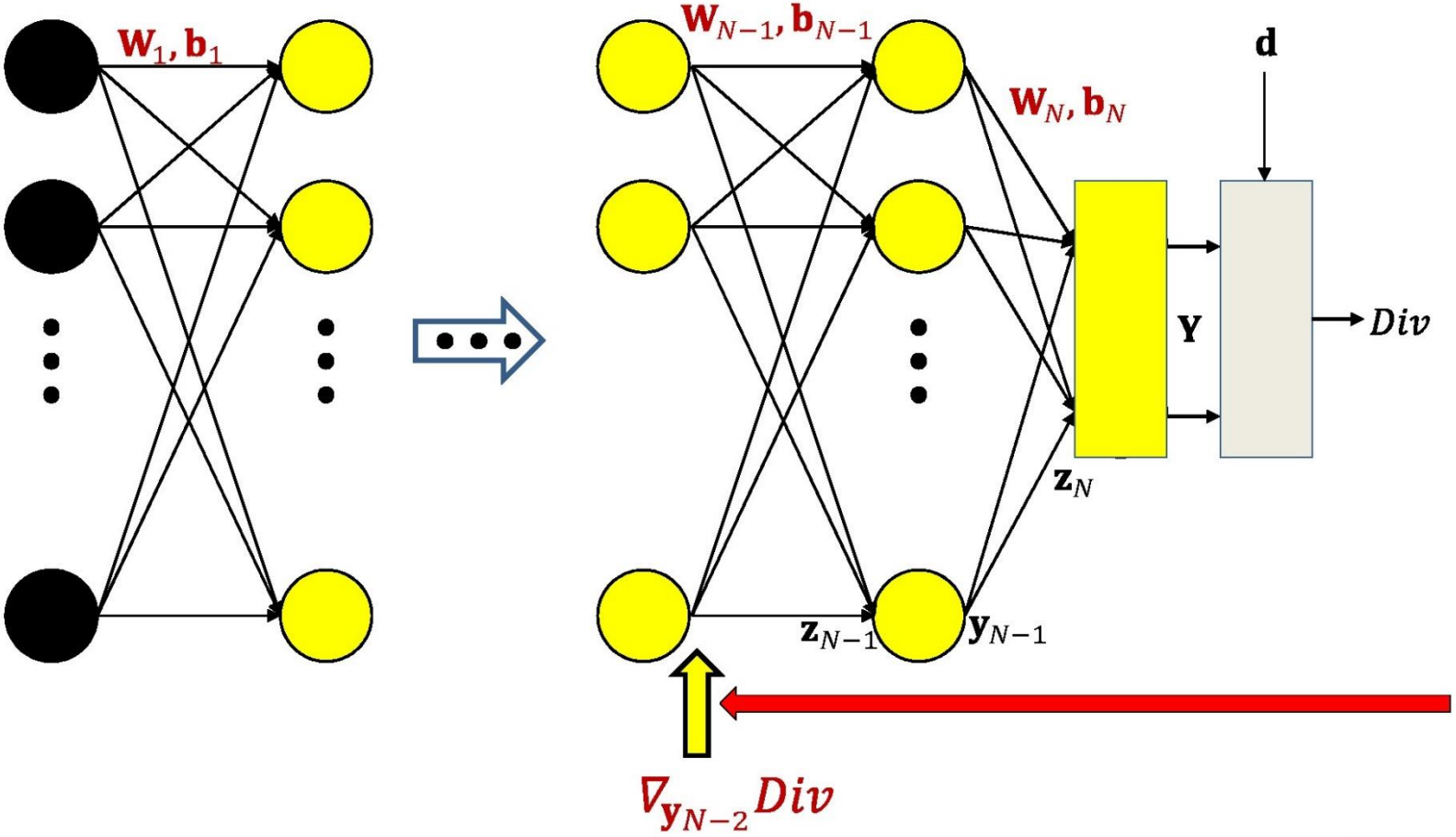
The Jacobian will be a diagonal matrix for scalar activations



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



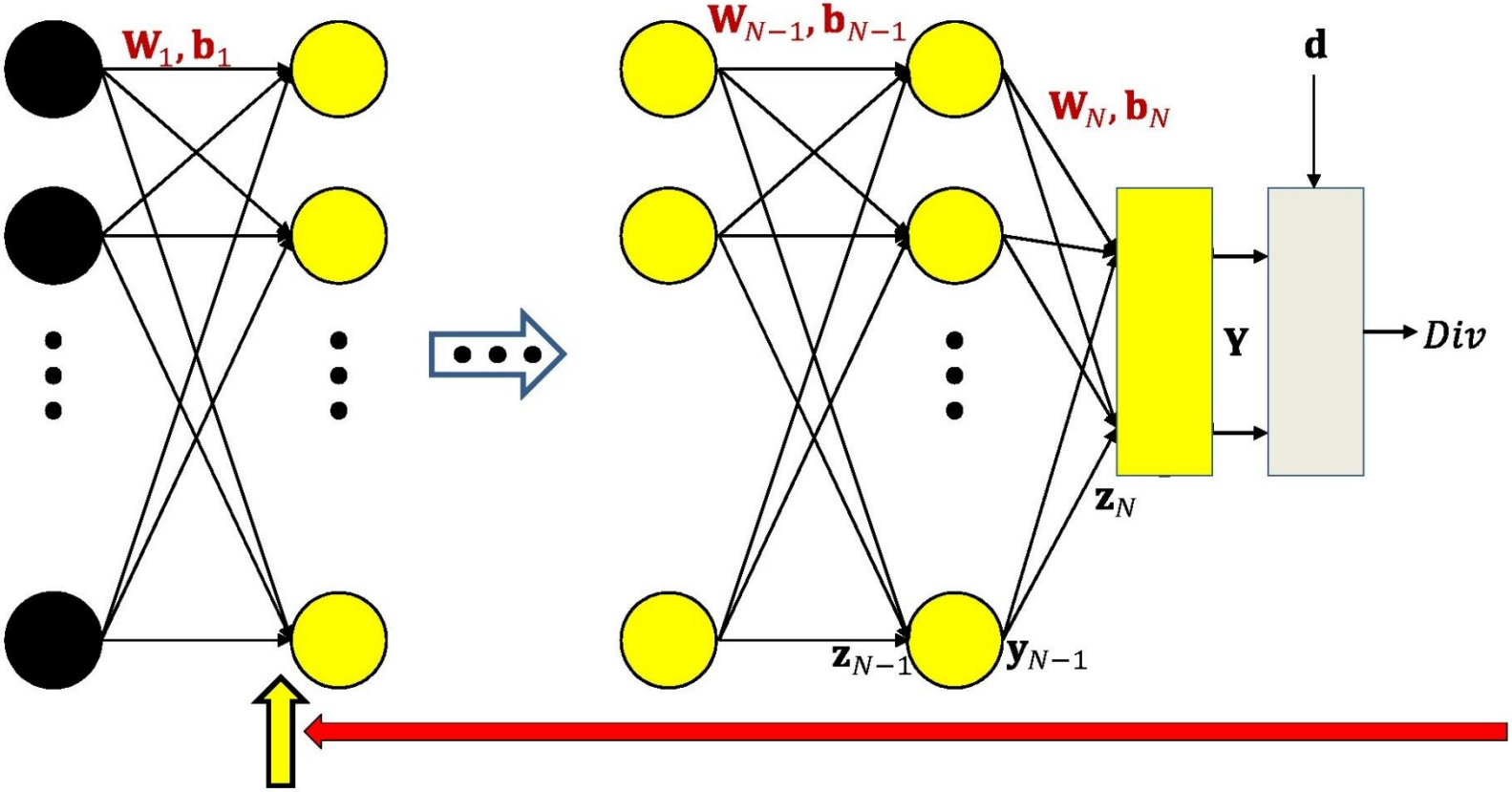
$$\nabla_{\mathbf{y}_{N-2}} Div = \nabla_{\mathbf{z}_{N-1}} Div \, \mathbf{W}_{N-1}$$



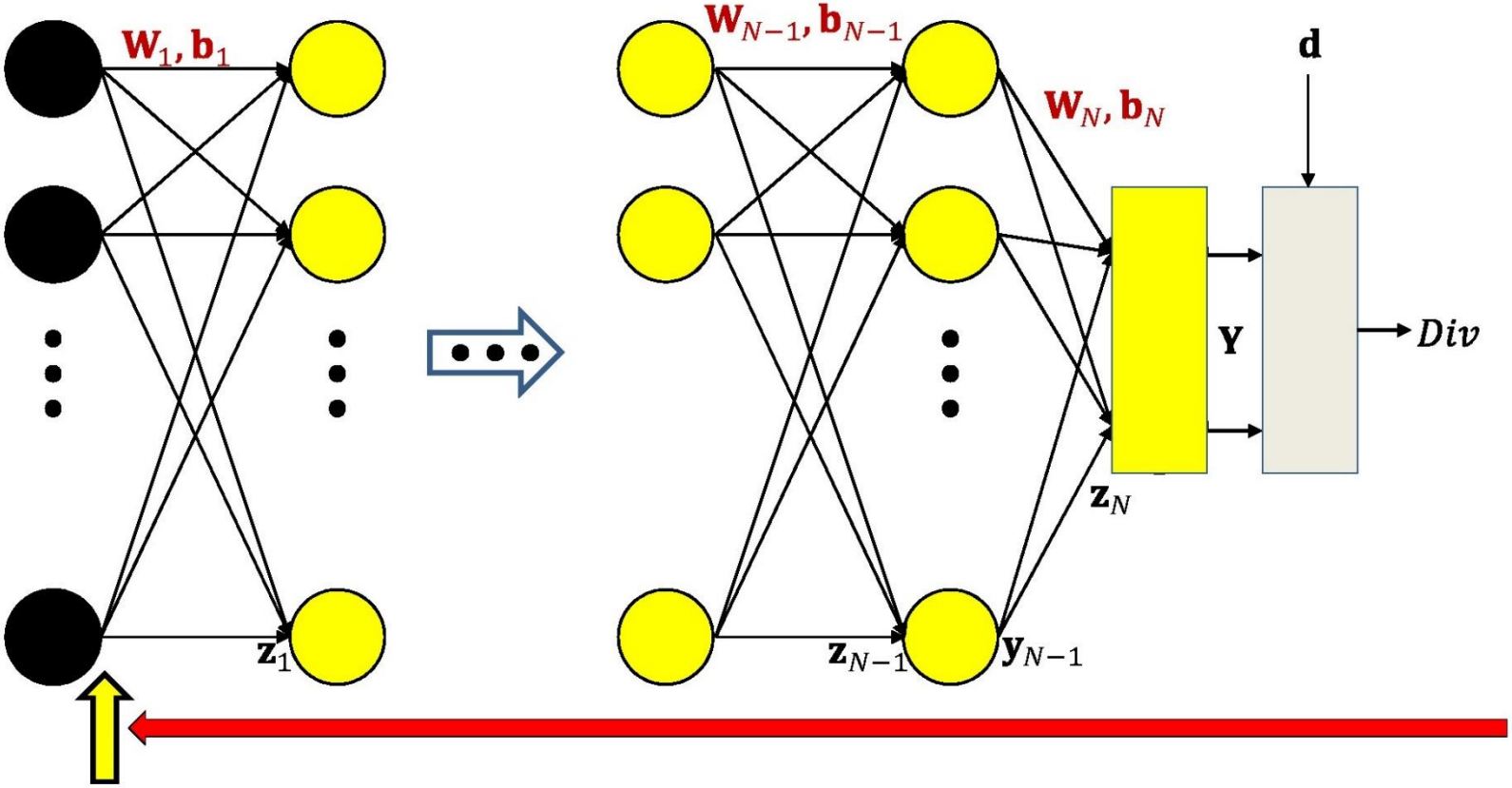
$$\nabla_{\mathbf{y}_{N-2}} Div = \nabla_{\mathbf{z}_{N-1}} Div \, \mathbf{W}_{N-1}$$

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

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



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



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

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

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:

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:

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

Neural network training algorithm

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

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 $\boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$
    - $Err \mathrel{+}= \boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$
  - Backward pass: For all k compute:
    - $\nabla_{\mathbf{y}_k} Div = \nabla_{\mathbf{z}_k+1} Div \, \mathbf{W}_k$
    - $\nabla_{\mathbf{z}_k} Div = \nabla_{\mathbf{y}_k} Div \, J_{\mathbf{y}_k}(\mathbf{z}_k)$
    - $\nabla_{\mathbf{W}_k} \boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$; $\nabla_{\mathbf{b}_k} \boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$
    - $\nabla_{\mathbf{W}_k} Err \mathrel{+}= \nabla_{\mathbf{W}_k} \boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$; $\nabla_{\mathbf{b}_k} Err \mathrel{+}= \nabla_{\mathbf{b}_k} \boldsymbol{Div}(\boldsymbol{Y}_t, \boldsymbol{d}_t)$
- 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