Machine learning

Applying Neural Networks

Mohammad-Reza A. Dehaqani

dehaqani@ut.ac.ir

Based on MIT 6.S191

Example Problem

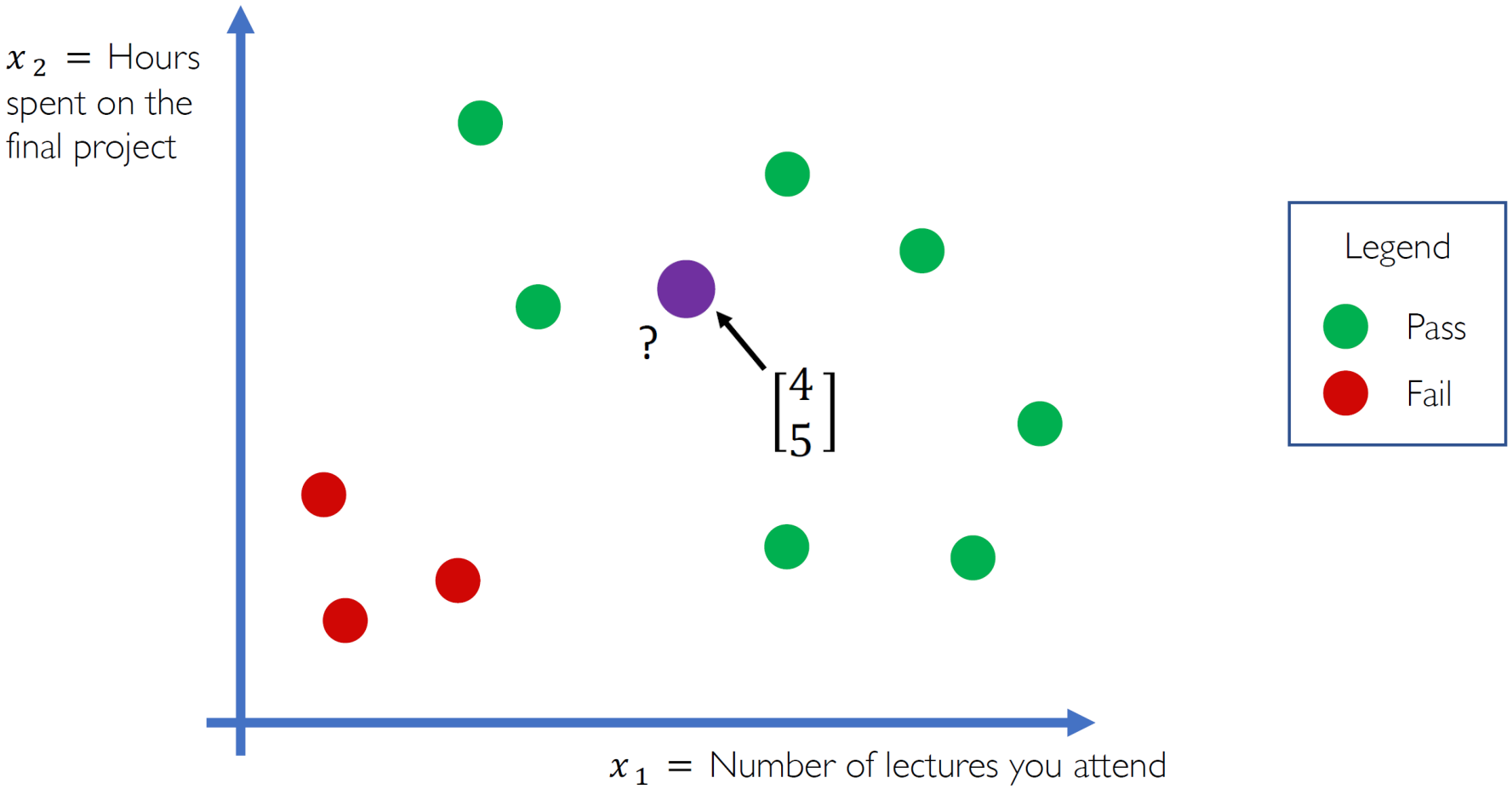
Will I pass this class?

Let's start with a simple two feature model

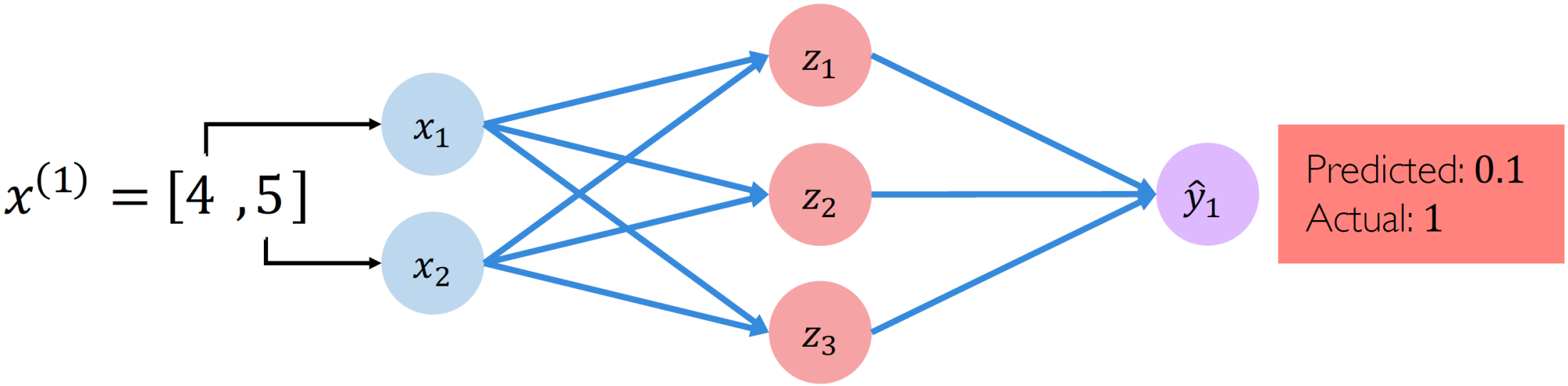
x_1 = Number of lectures you attend

x_2 = Hours spent on the final project

Example Problem: Will I pass this class?

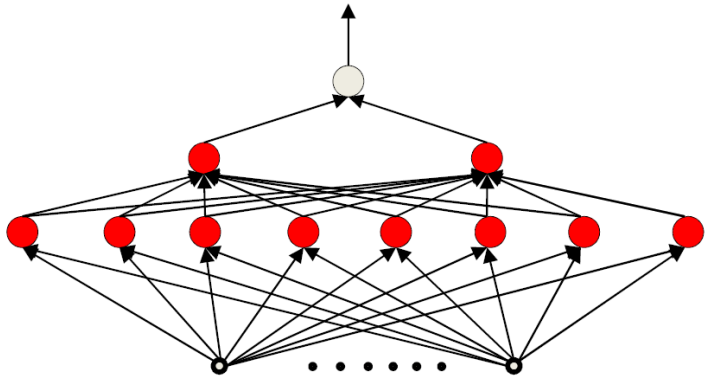


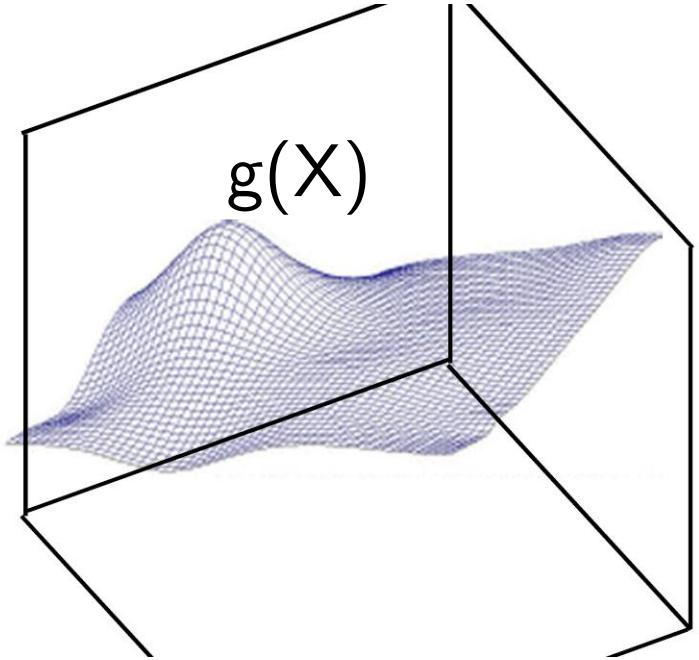
Example Problem: Will I pass this class?



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

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





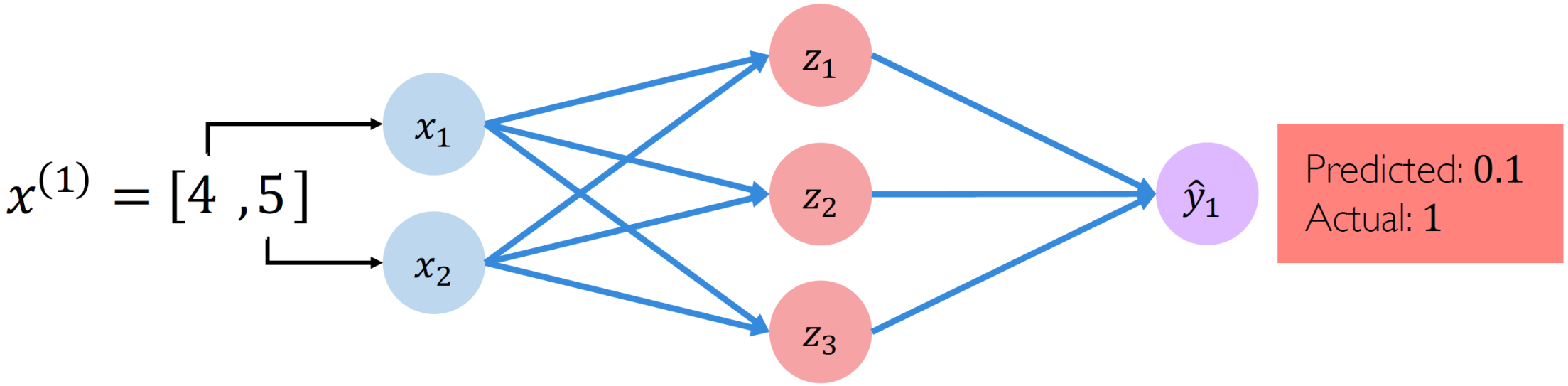
- More generally, given the function g(X) to model, we can derive the parameters of the network to model it, through computation
- When Y=f(X;**W**) has the capacity to exactly represent g(X)

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

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

Quantifying Loss

*The **loss** of our network measures the cost incurred from incorrect predictions*



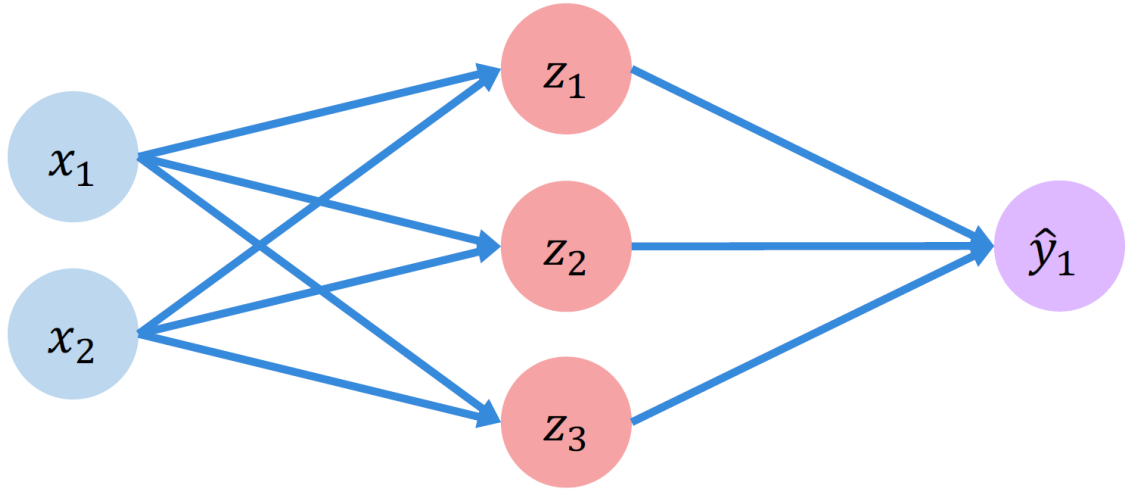
$$\mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Empirical Loss

*The **empirical loss** measures the total loss over our entire dataset*

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$$f(x) \quad y$$

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \quad \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

Also known as:

- Objective function
- Cost function
- Empirical Risk

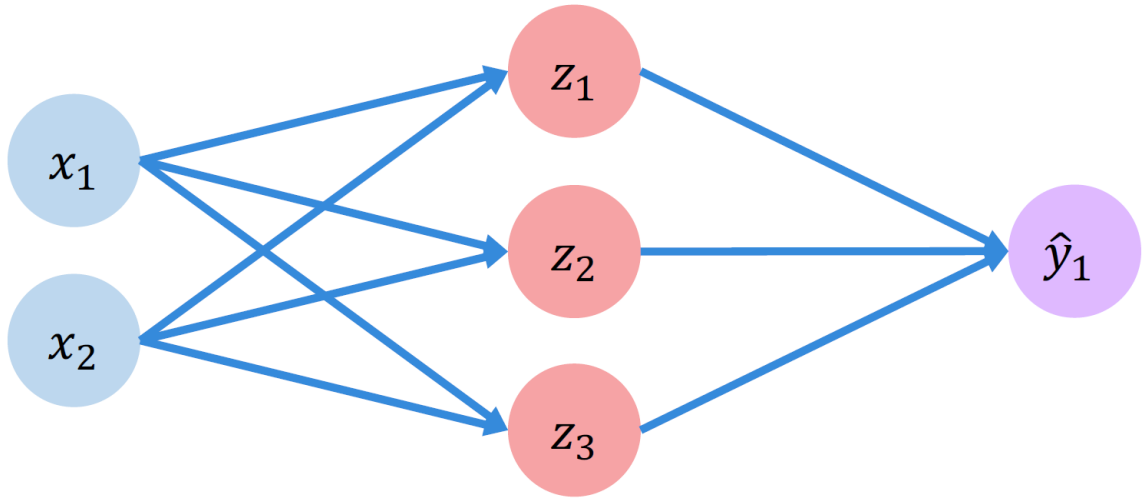
$$\boldsymbol{J(W)} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

Predicted Actual

Binary Cross Entropy Loss

Cross entropy loss *can be used with models that output a probability between 0 and 1*

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$f(x)$

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix}$$

y

$$\begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

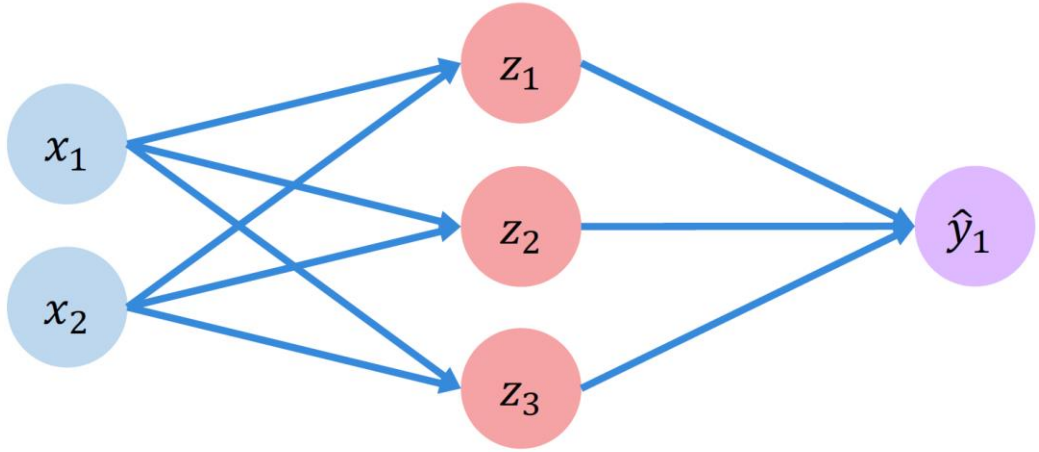
$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log\left(f\left(x^{(i)}; \boldsymbol{W}\right)\right) + (1 - y^{(i)}) \log\left(1 - f\left(x^{(i)}; \boldsymbol{W}\right)\right)$$

Actual Predicted Actual Predicted

Mean Squared Error Loss

Mean squared error loss *can be used with regression models that output continuous real numbers*

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$



$$\boldsymbol{J(W)} = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f\left(x^{(i)}; \boldsymbol{W}\right)\right)^2$$

Actual Predicted

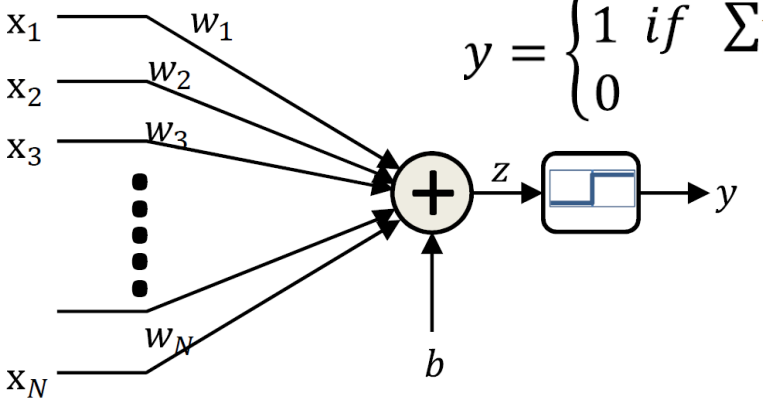
$f(x)$ y

$$\begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix} \quad \begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix}$$

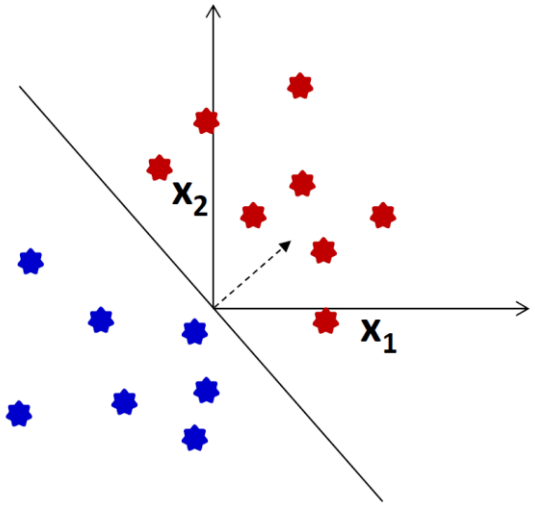
Final Grades (percentage)

Training Neural Networks

Learning the perceptron



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

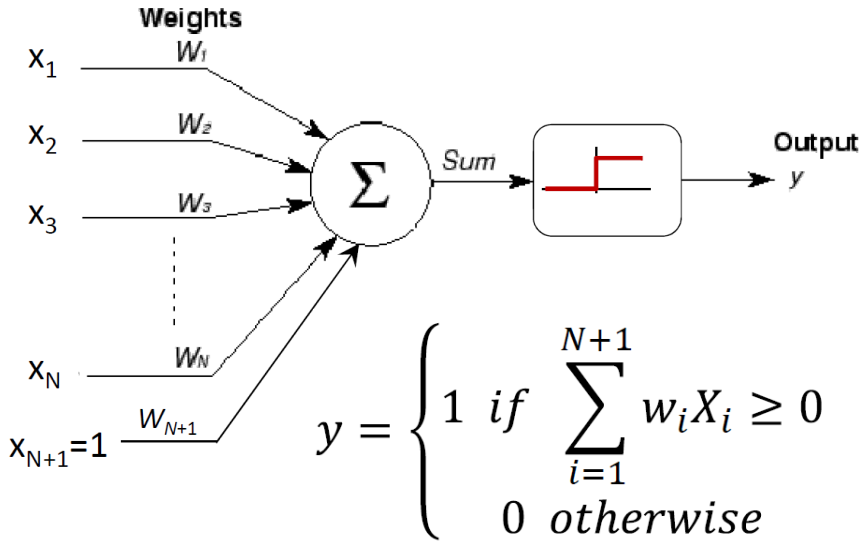


Learn $W = [w_1 .. w_N]$ and b, given several (X, y) pairs

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

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

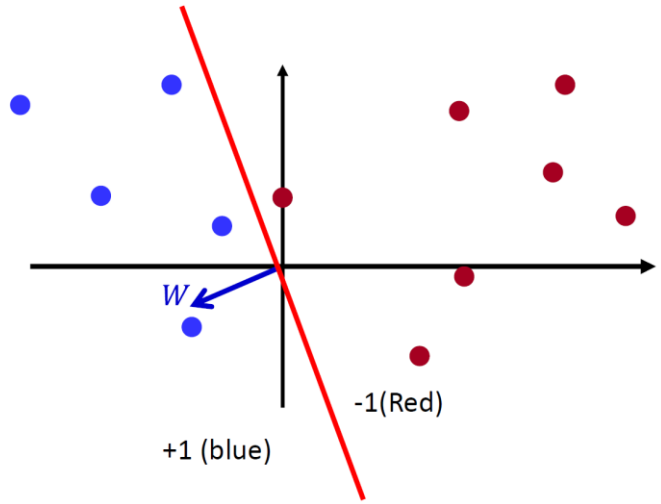
- Initialize: Randomly initialize the hyperplane (i.e. randomly initialize the normal vector W)
- Classification rule sign($W^T x$)
  - Vectors on the same side of the hyperplane as W will be assigned +1 class, and those on the other side will be assigned -1

- If instance misclassified
  - If instance is positive class

$$W = W + X_i$$

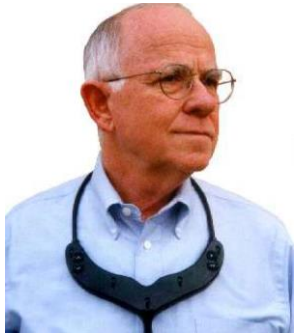
  - If instance is negative class

$$W = W - X_i$$



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

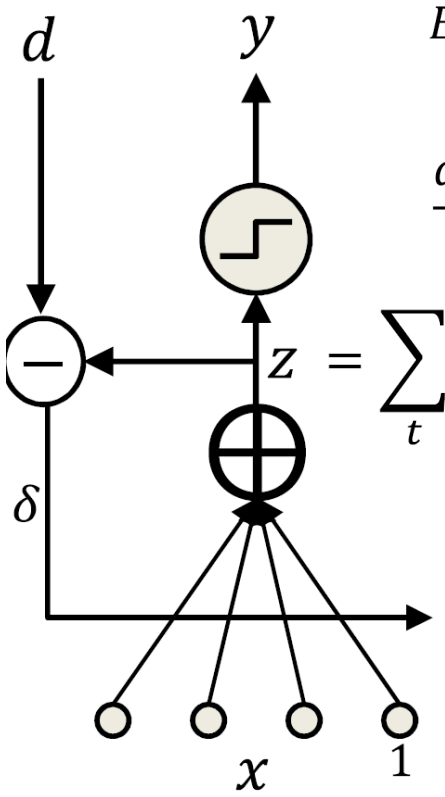
  After each input , that has target (binary) output , compute and update

$$\delta = d - z$$

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

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

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



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

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

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

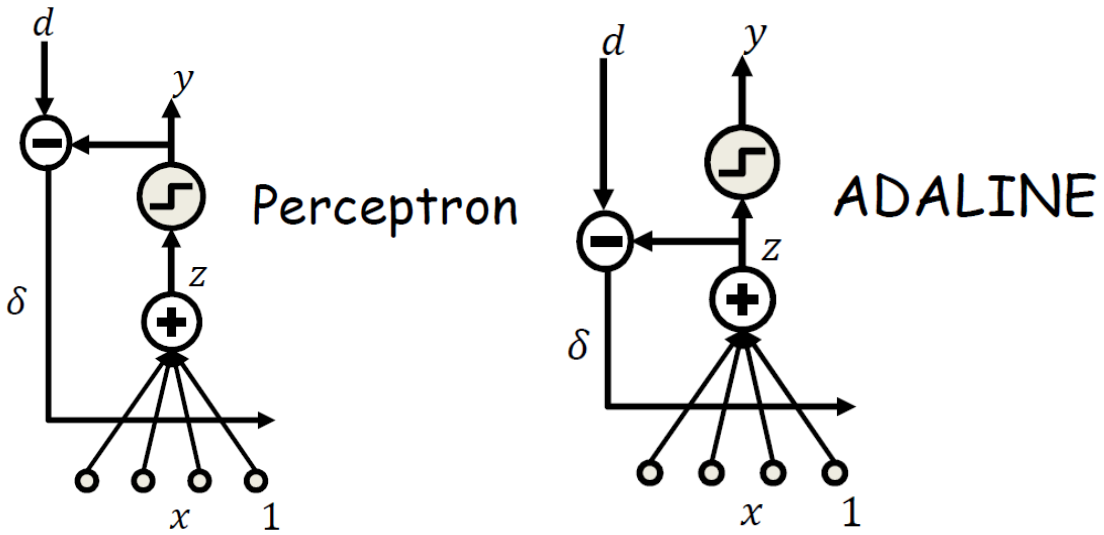
Generalized delta rule

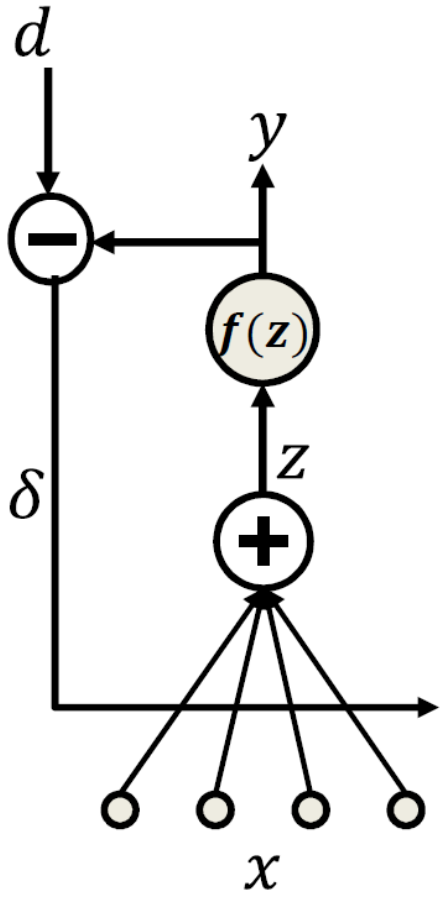
- For any differentiable activation function the following update rule is used

$$\delta = d - y$$

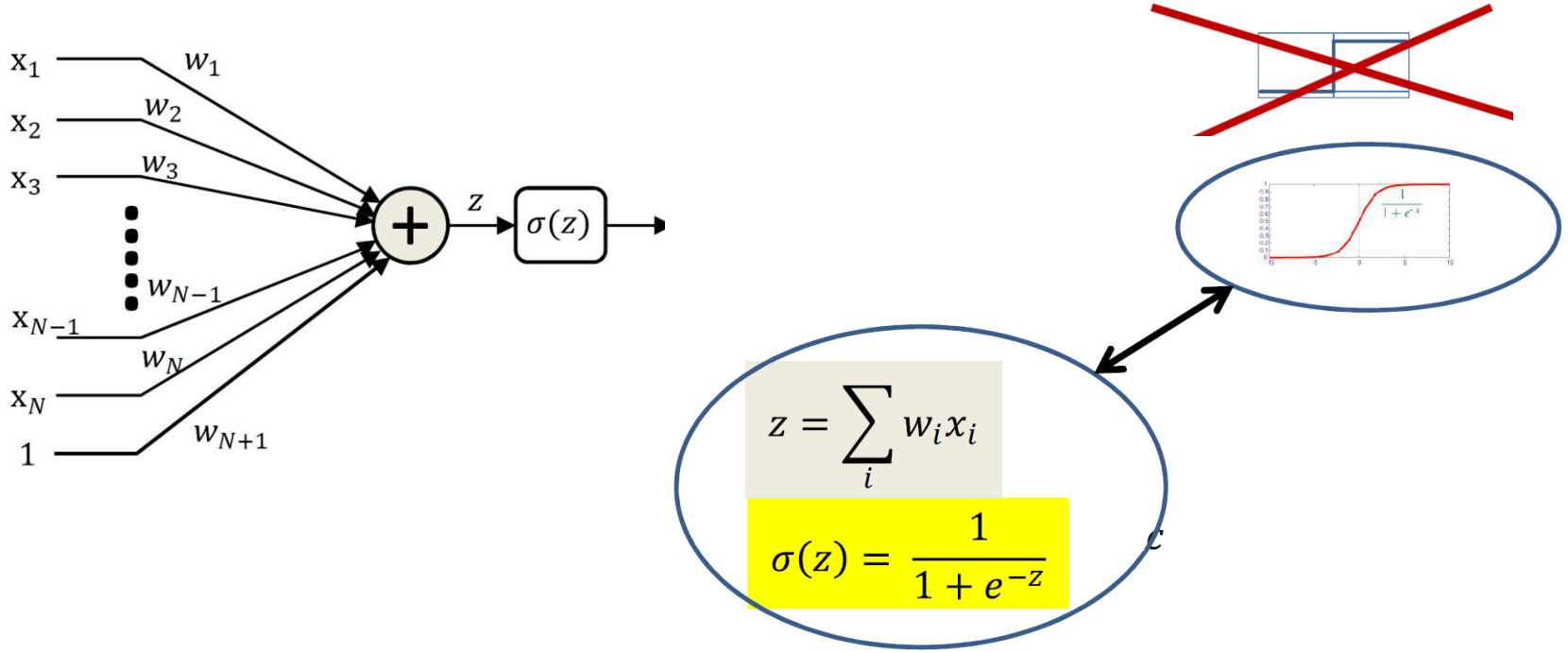
$$w_i = w_i + \eta \delta f'(z) x_i$$

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



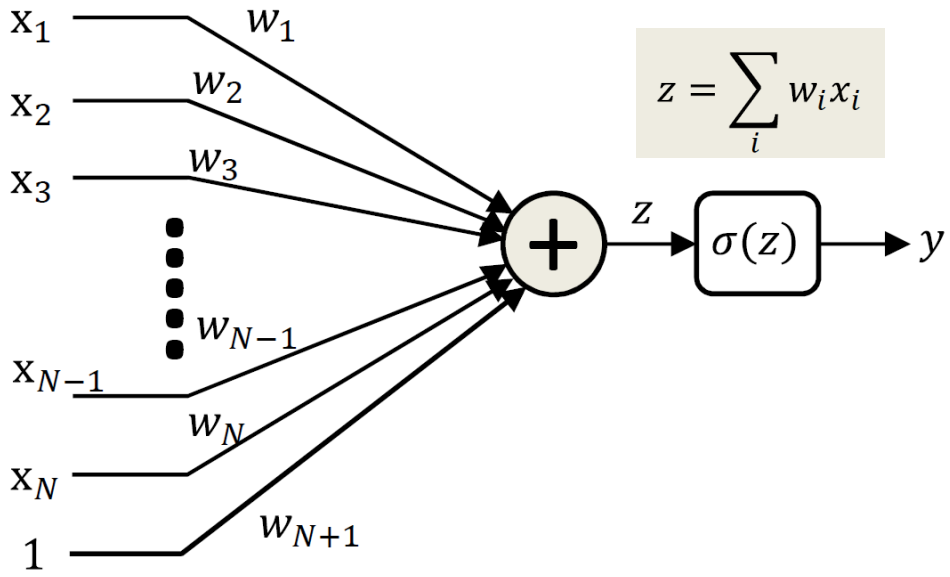


Lets make the neuron differentiable



- 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

Perceptrons with differentiable activation functions



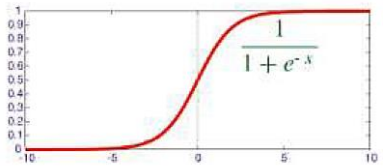
$$z = \sum_i w_i x_i$$

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

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

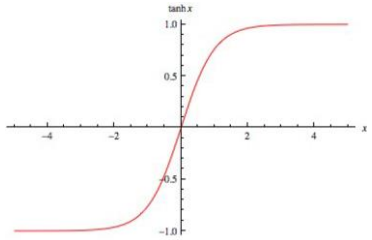
- $\sigma(z)$ is a differentiable function of z
- Using the chain rule, y is a differentiable function of both inputs x_i and weight w_i
- This means that we can compute the change in the output for small changes in either the input or the weights

Some popular activations and their derivatives



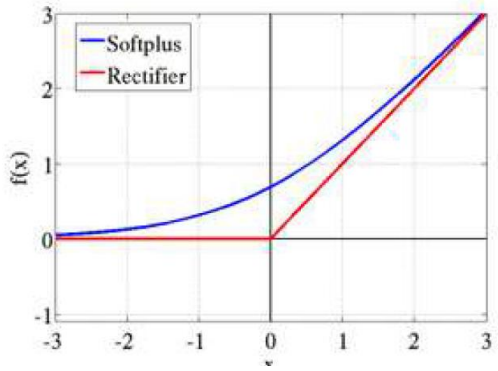
$$f(z) = \frac{1}{1 + \exp(-z)}$$

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



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

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



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

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

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

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

Loss Optimization

We want to find the network weights that ***achieve the lowest loss***

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

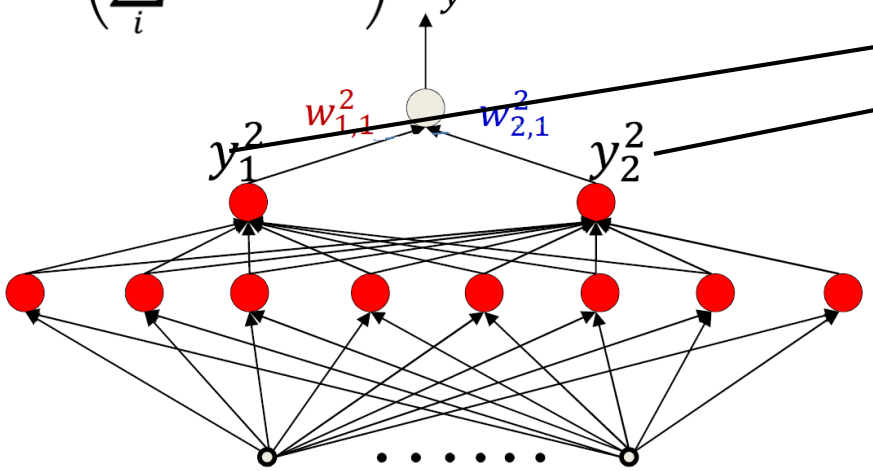
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

Remember:

$$\boldsymbol{W} = \{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}, \dots\}$$

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

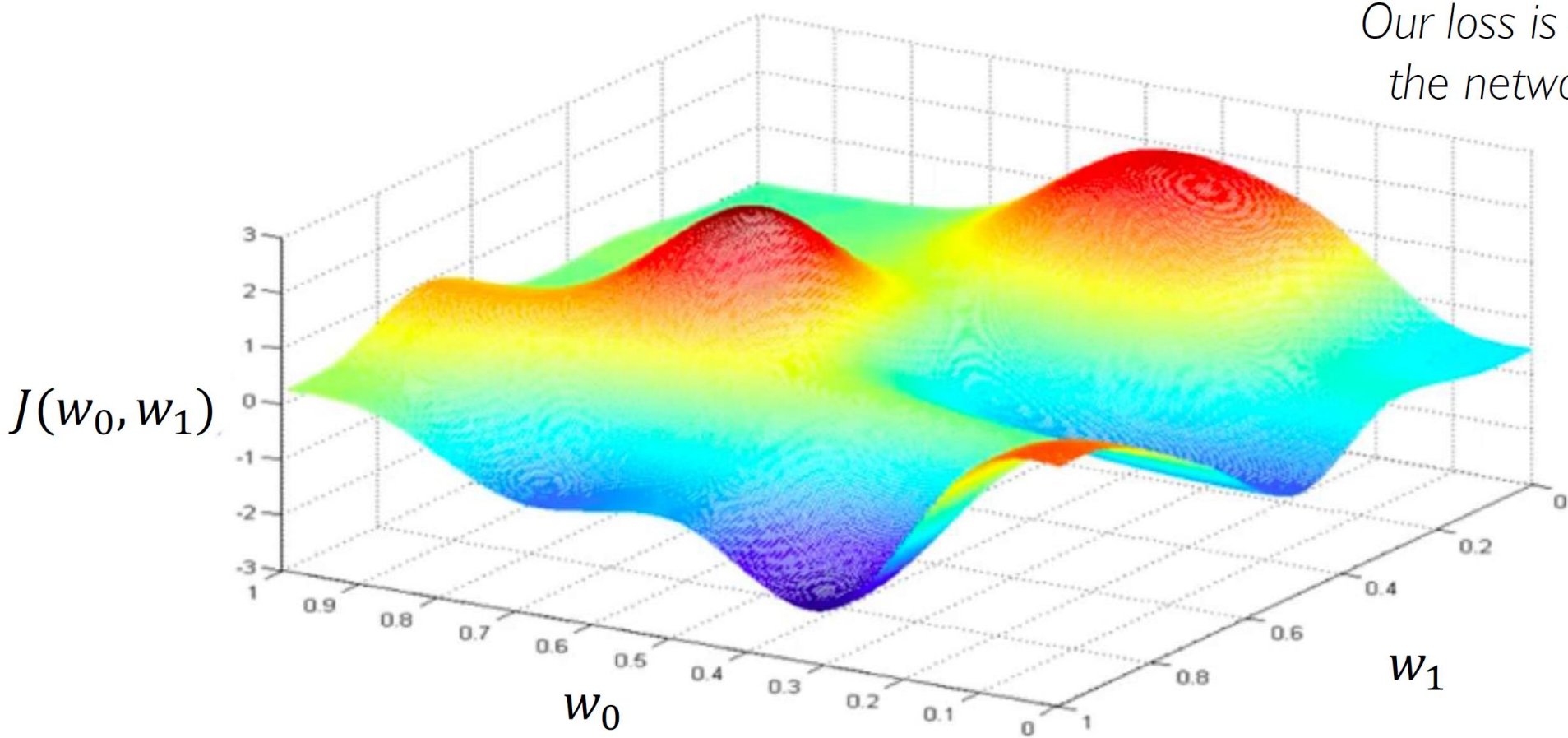
- Every individual perceptron is differentiable w.r.t its inputs and its weights (including "bias" weight)
- By the chain rule, the overall function is differentiable w.r.t every parameter (weight or bias)
  - Small changes in the parameters result in measurable changes in output

Loss Optimization

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W})$$

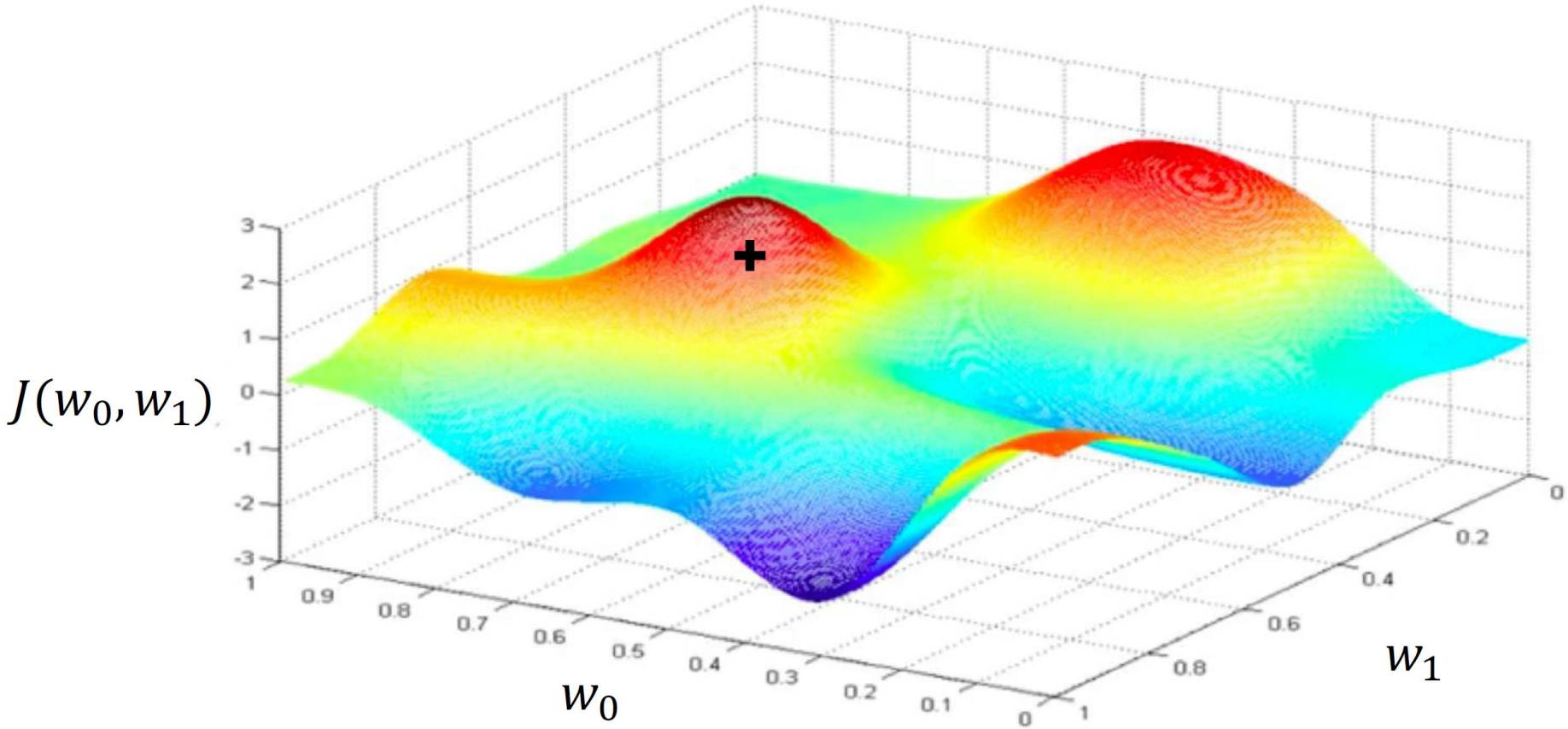
Remember:

Our loss is a function of the network weights!



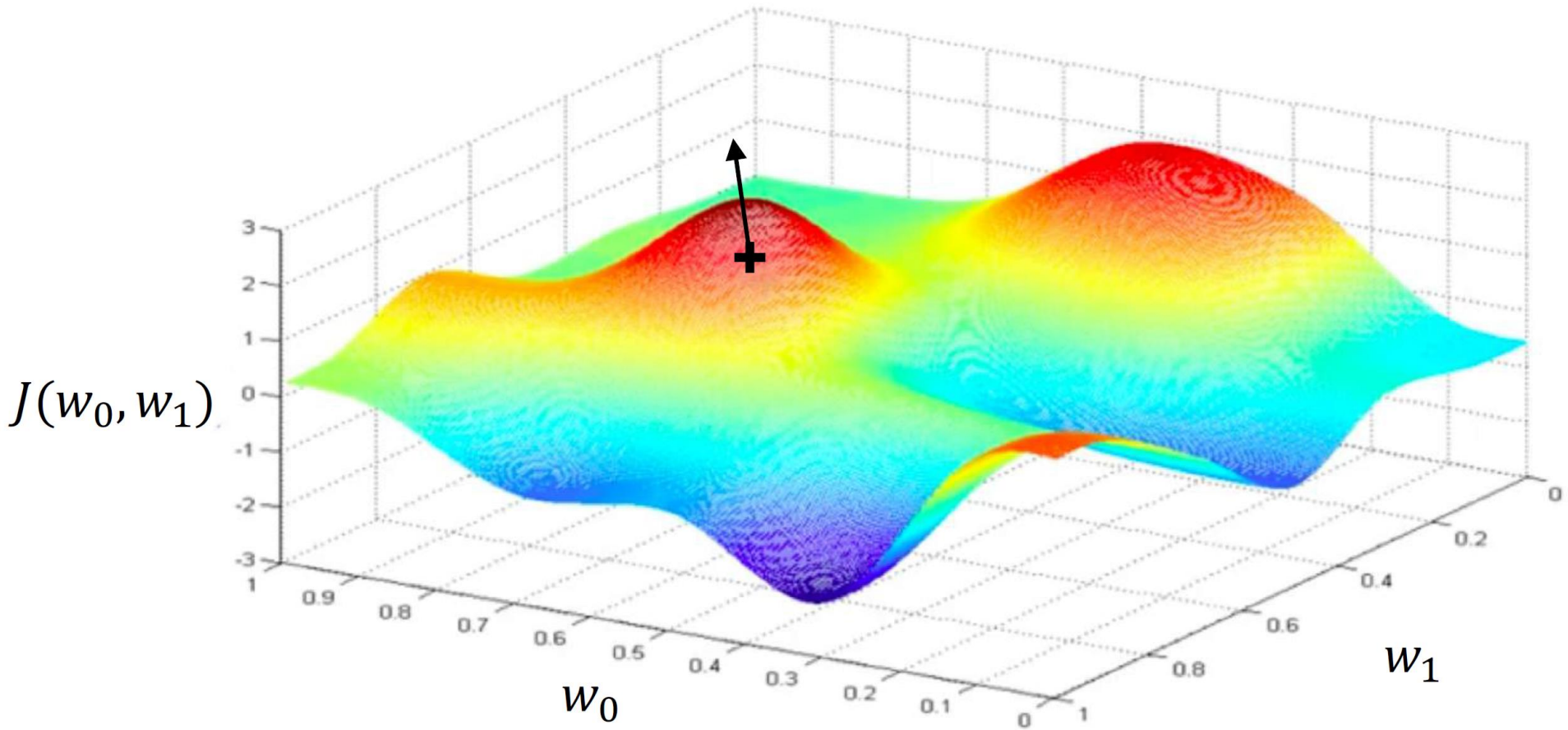
Loss Optimization

Randomly pick an initial (w_0, w_1)



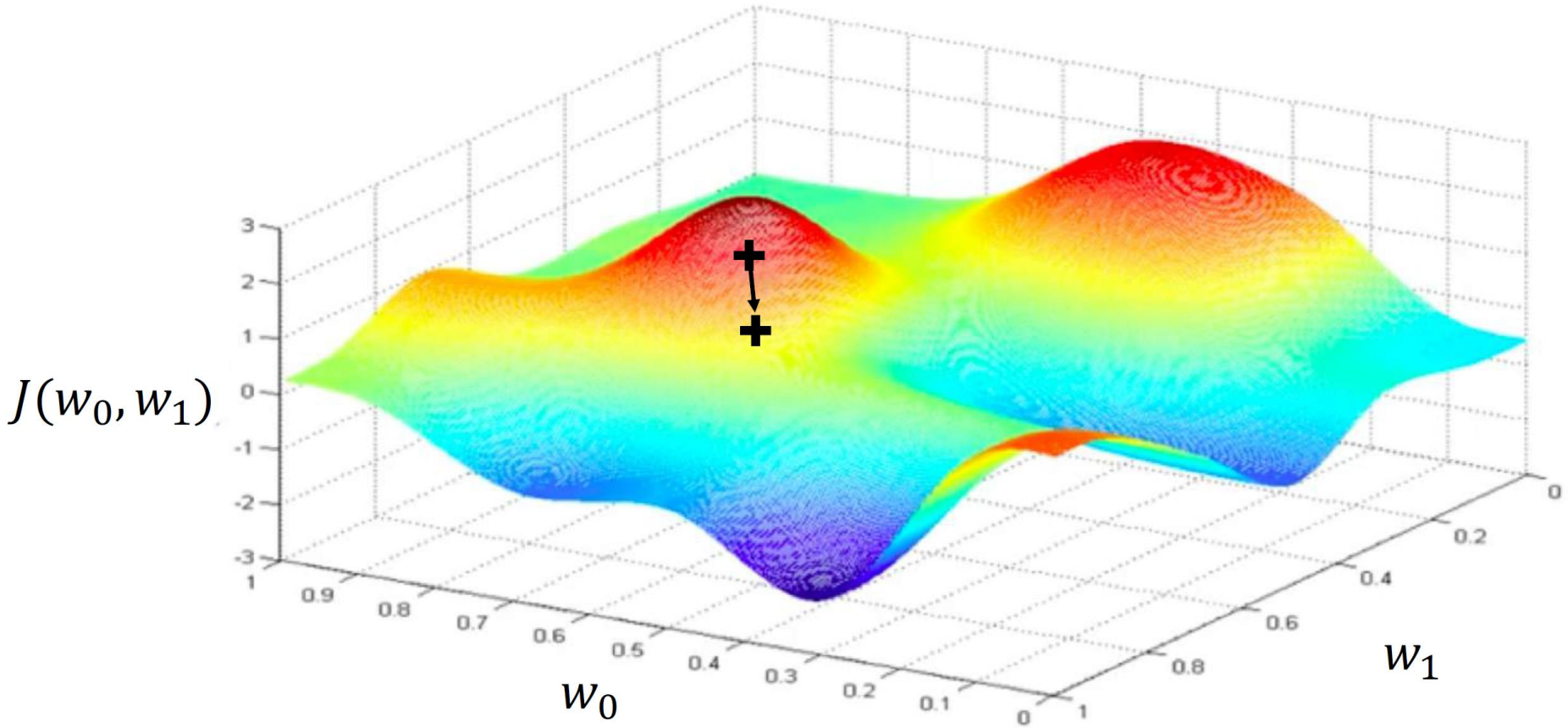
Loss Optimization

Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$



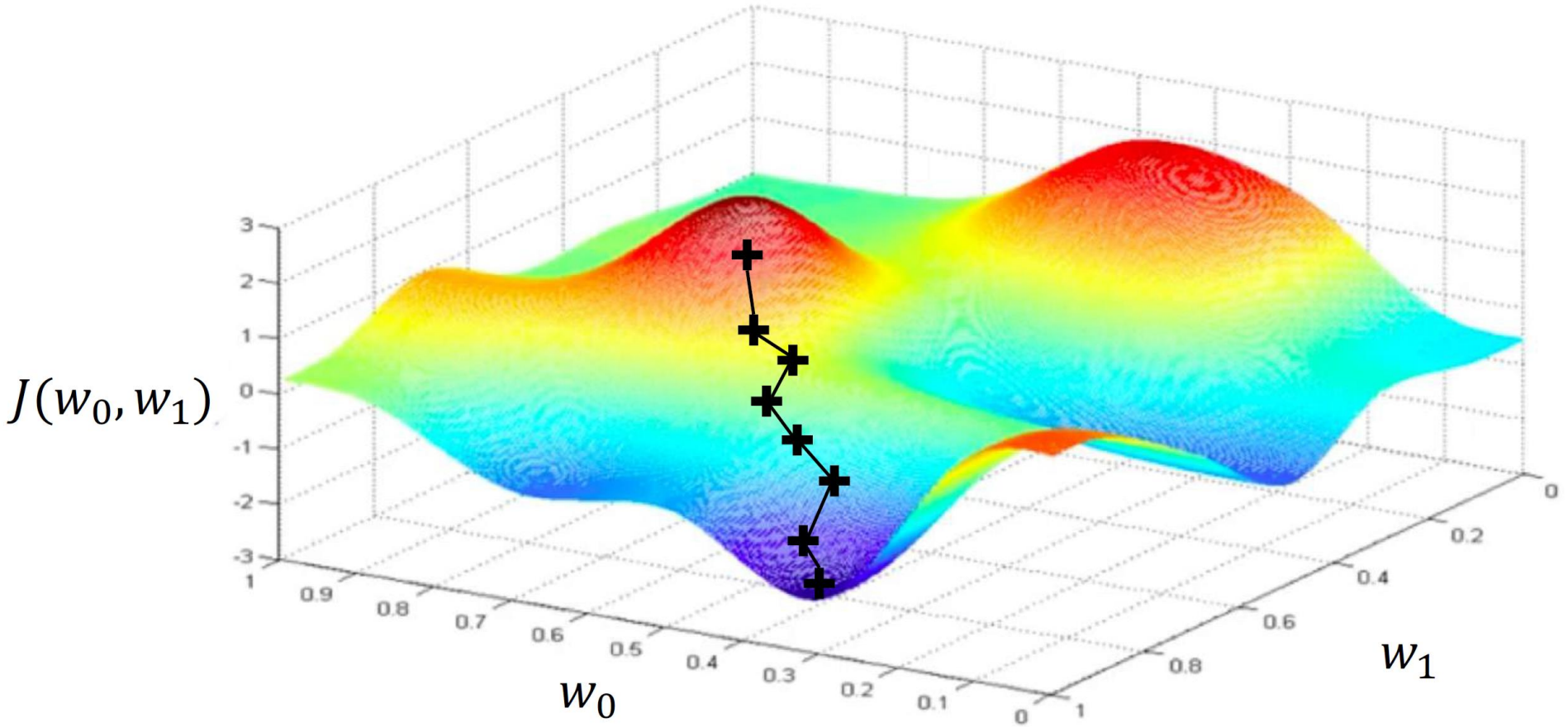
Loss Optimization

Take small step in opposite direction of gradient



Gradient Descent

Repeat until convergence

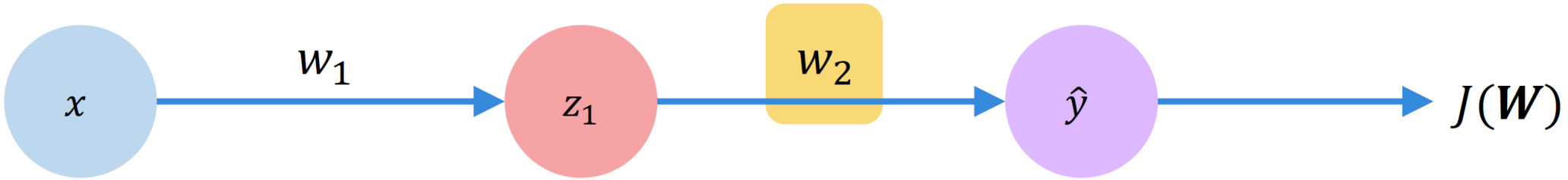


Gradient Descent

Algorithm

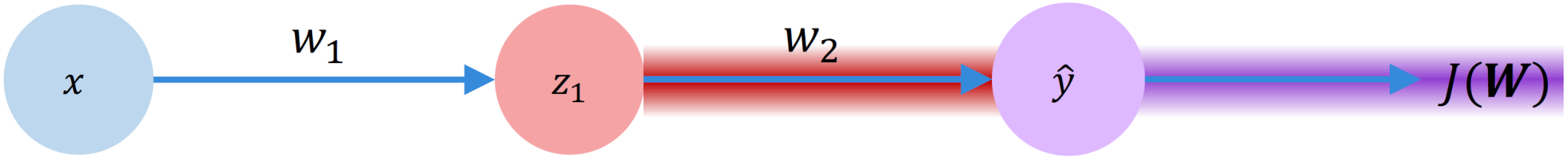
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence: While $\left\|\nabla_x f(x^k)\right\| > \varepsilon$ (or while $\left|f\left(x^{k+1}\right) - f\left(x^k\right)\right| > \varepsilon$)
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

Computing Gradients: Backpropagation



How does a small change in one weight (ex. $\boldsymbol{w_2}$) affect the final loss $\boldsymbol{J(W)}$?

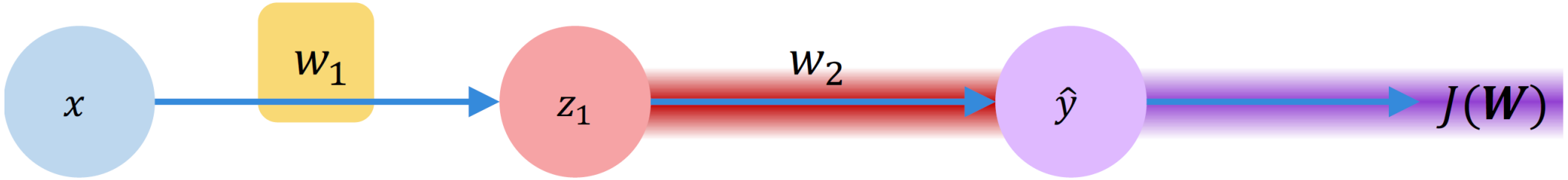
Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

Let's use the chain rule!

Computing Gradients: Backpropagation

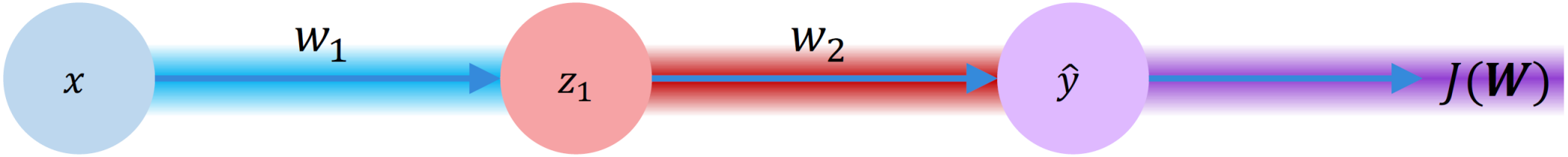


$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1}$$

Apply chain rule!

Apply chain rule!

Computing Gradients: Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Repeat this for every weight in the network using gradients from later layers

Chain Rule

$$\frac{d}{dx}\left[f\left(g(x)\right)\right] = f'\left(g(x)\right)g'(x)$$

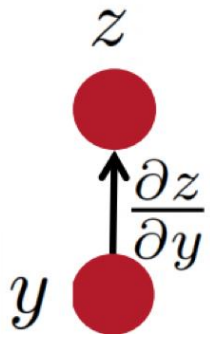
$$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$$

Some facts from real analysis

- First let's get the notation right:

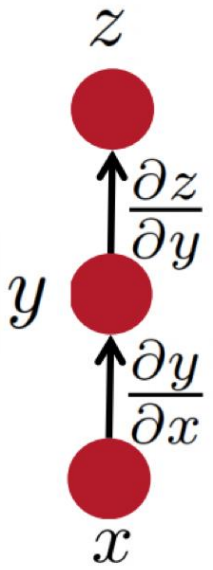
- The arrow shows functional dependence of z on y
  - i.e. given y, we can calculate z.
  - e.g., for example: $z(y) = 2y^2$

The derivative of z, with respect to y.



Some facts from real analysis

- Simple chain rule
  - If z is a function of y, and y is a function of x
    - Then z is a function of x, as well.
  - Question: how to find $\frac{\partial z}{\partial x}$

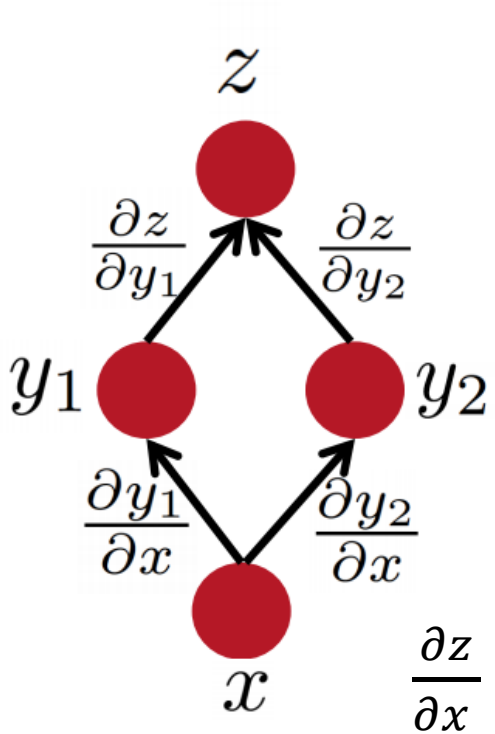


$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

We will use these facts to derive the details of the Backpropagation algorithm.
z will be the error (loss) function. - We need to know how to differentiate z
Intermediate nodes use a logistics function (or another differentiable step function). - We need to know how to differentiate it.

Some facts from real analysis

- Multiple path chain rule

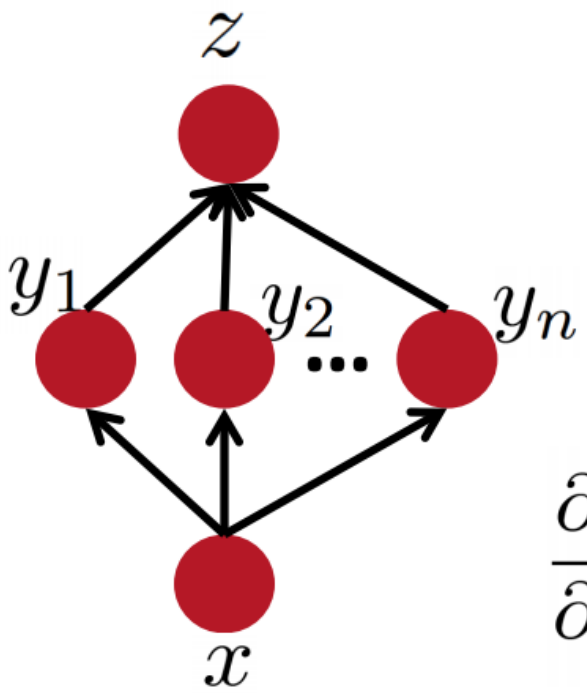


$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

Slide Credit: Richard Socher

Some facts from real analysis

- Multiple path chain rule: general

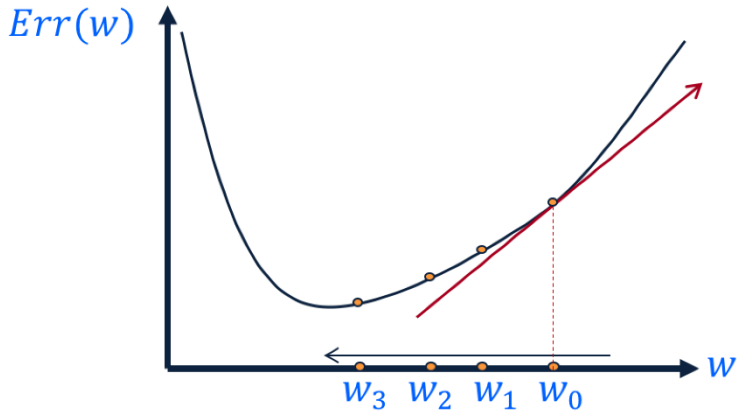


$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

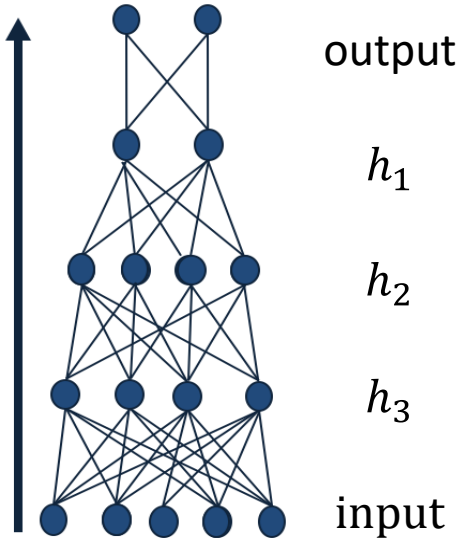
Slide Credit: Richard Socher

Key Intuitions Required for BP

- Gradient Descent
  - Change the weights in the direction of gradient to minimize the error function.
- Chain Rule
  - Use the chain rule to calculate the weights of the intermediate weights
- Dynamic Programming (Memoization)
  - Memoize the weight updates to make the updates faster.

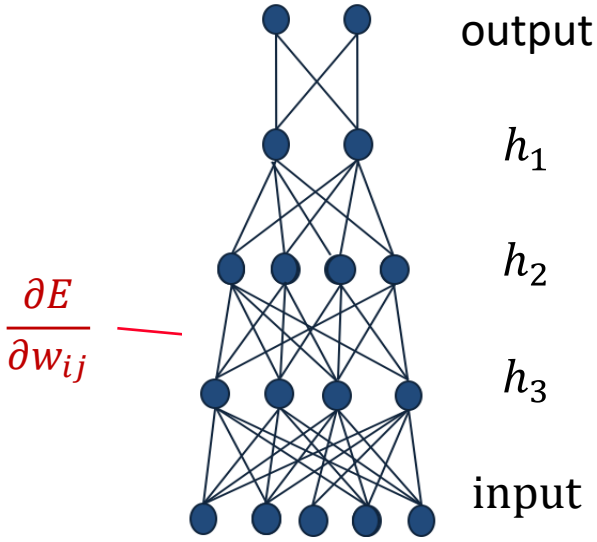


$\frac{\partial E}{\partial w_{ij}}$



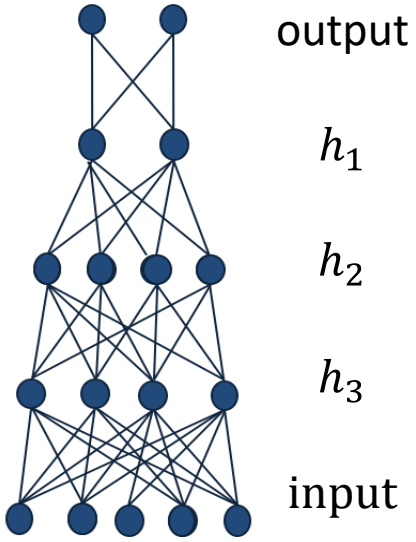
Backpropagation: the big picture

- Loop over instances:
  1. The forward step
     - Given the input, make predictions layer-by-layer, starting from the first layer)
  2. The backward step
     - Calculate the error in the output
     - Update the weights layer-by-layer, starting from the final layer



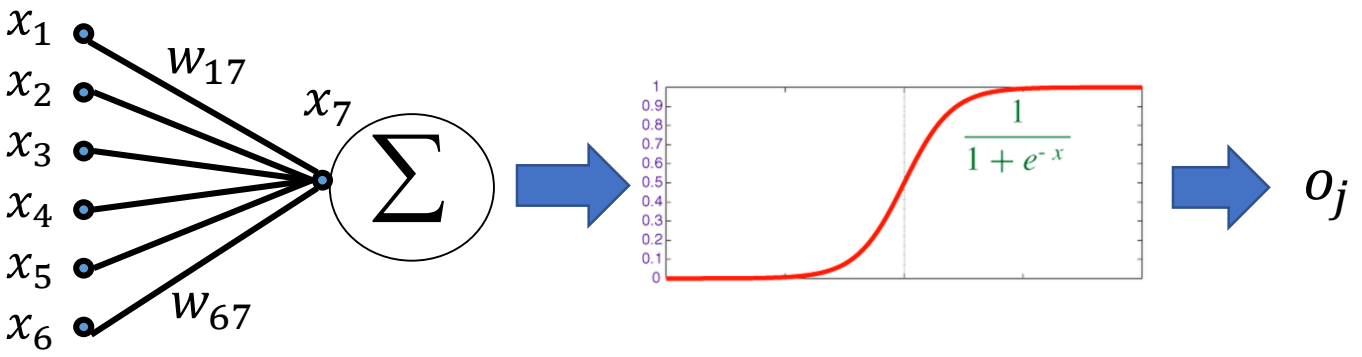
Questions

- What is the purpose of forward step?
  - To make predictions, given an input.
- What is the purpose of backward step?
  - To update the weights, given an output error.
- Why do we use the chain rule?
  - To calculate gradient in the intermediate layers.
- Why backpropagation could be efficient?
  - Because it can be parallelized.



Reminder: Model Neuron (Logistic)

- Neuron is modeled by a unit j connected by weighted links w_{ij} to other units i.



- Use a non-linear, differentiable output function such as the sigmoid or logistic function
- Net input to a unit is defined as:

$$\text{net}_j = \sum w_{ij}.x_i$$

- Output of a unit is defined as:

$$o_j = \frac{1}{1+\exp\left(-(\text{net}_j - T_j)\right)}$$

Note:
Other gates, beyond Sigmoid, can be used (TanH, ReLu)
Other Loss functions, beyond LMS, can be used.

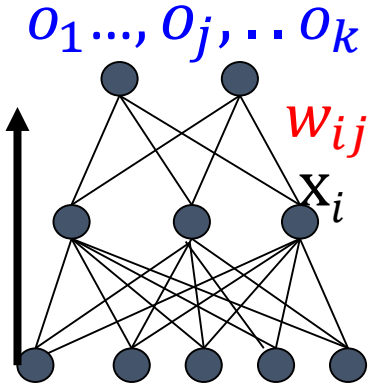
Derivation of Learning Rule

- The weights are updated incrementally; the error is computed for each example and the weight update is then derived.

$$E_d(\boldsymbol{w}) = \frac{1}{2}\sum_{k\in K}(t_k - o_k)^2$$

- w_{ij} influences the output o_j only through net_j

$$o_j = \frac{1}{1+\exp\{-(\text{net}_j - T)\}} \quad \text{and} \quad \text{net}_j = \sum w_{ij}.x_i$$

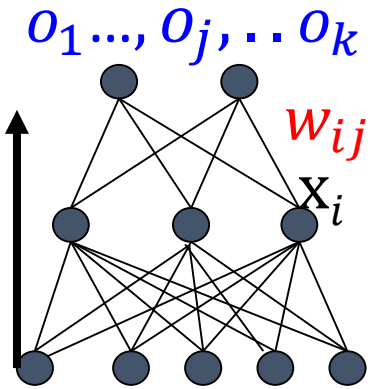


- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial o_j}\,\frac{\partial o_j}{\partial \text{net}_j}\,\frac{\partial \text{net}_j}{\partial w_{ij}}$$

Derivatives

- Function 1 (error):
  - $E = \frac{1}{2}\sum_{k \in K}(t_k - o_k)^2$
  - $\frac{\partial E}{\partial o_i} = -(t_i - o_i)$
- Function 2 (linear gate):
  - $\text{net}_j = \sum w_{ij} \cdot x_i$
  - $\frac{\partial \text{net}_j}{\partial w_{ij}} = x_i$
- Function 3 (differentiable activation function):
  - $o_i = \frac{1}{1+\exp\{-(\text{net}_j - T)\}}$
  - $\frac{\partial o_i}{\partial \text{net}_j} = \frac{\exp\{-(\text{net}_j - T)\}}{(1+\exp\{-(\text{net}_j - T)\})^2} = o_i(1 - o_i)$



$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}}$$

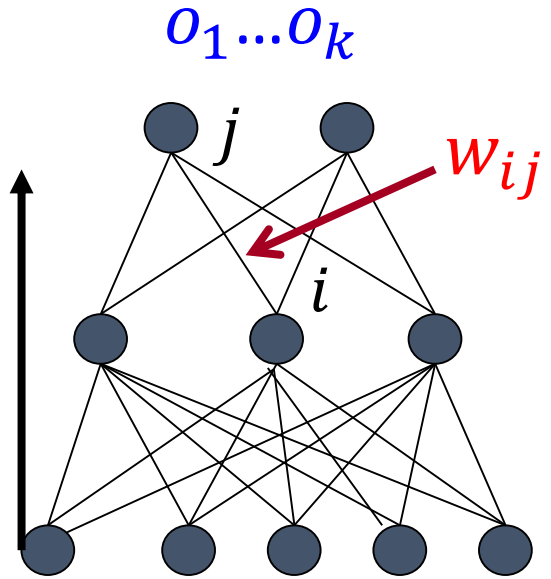
$$\sigma(x) = \frac{1}{1+e^{-x}}.$$

$$\frac{d\sigma(x)}{d(x)} = \sigma(x) \cdot (1 - \sigma(x)).$$

Derivation of Learning Rule (2)

- Weight updates of output units:
  - w_{ij} influences the output only through net_j
- Therefore:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial o_j}\,\frac{\partial o_j}{\partial \text{net}_j}\,\frac{\partial \text{net}_j}{\partial w_{ij}} = -(t_j - o_j)\; o_j(1-o_j)\; x_i$$



$$E_d(\boldsymbol{w}) = \frac{1}{2}\sum_{k \in K}(t_k - o_k)^2$$

$$\frac{\partial o_j}{\partial \text{net}_j} = o_j(1-o_j)$$

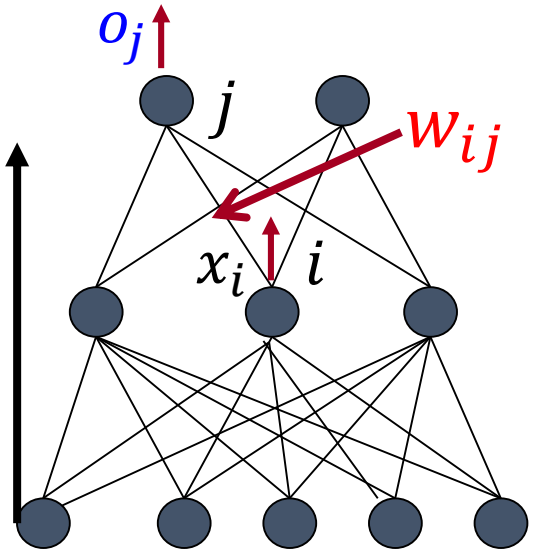
$$o_j = \frac{1}{1+\exp\{-(\text{net}_j - T_j)\}}$$

$$\sum w_{ij}.x_i$$

Derivation of Learning Rule (3)

- Weights of output units:
  - w_{ij} is changed by:

$$\Delta w_{ij} = \eta(t_j - o_j)o_j(1 - o_j)x_i$$
$$= \eta \delta_j x_i$$



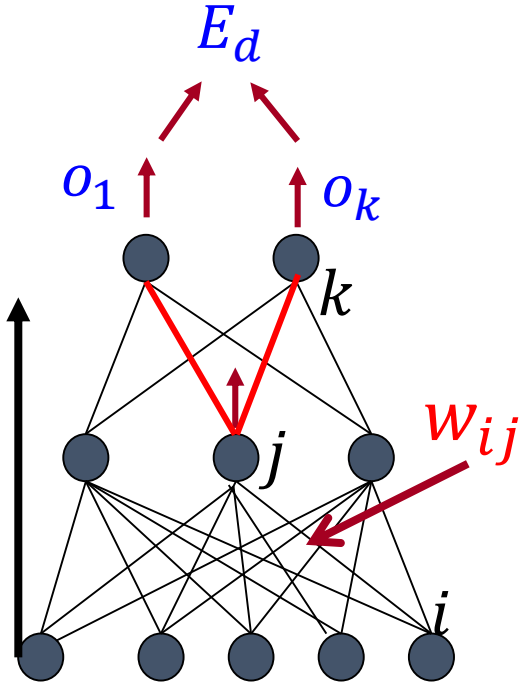
Where we defined:

$$\delta_j = \frac{\partial E_d}{\partial \text{net}_j} = (t_j - o_j)o_j(1 - o_j)$$

Derivation of Learning Rule (4)

- Weights of hidden units:
  - w_{ij} Influences the output only through all the units whose direct input include j

$$\frac{\partial E_d}{\partial w_{ij}}$$



Derivation of Learning Rule (4)

- Weights of hidden units:
  - w_{ij} Influences the output only through all the units whose direct input include j

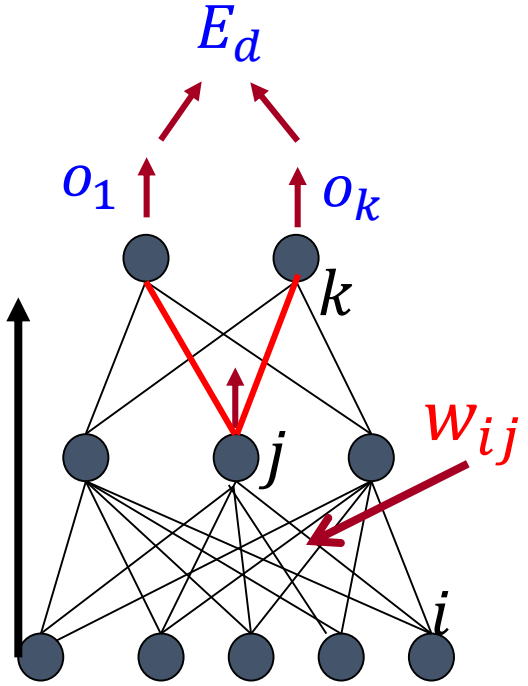
$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}} =$$

$$\text{net}_j = \sum w_{ij}.x_i$$

$$= \frac{\partial E_d}{\partial \text{net}_j} x_i =$$

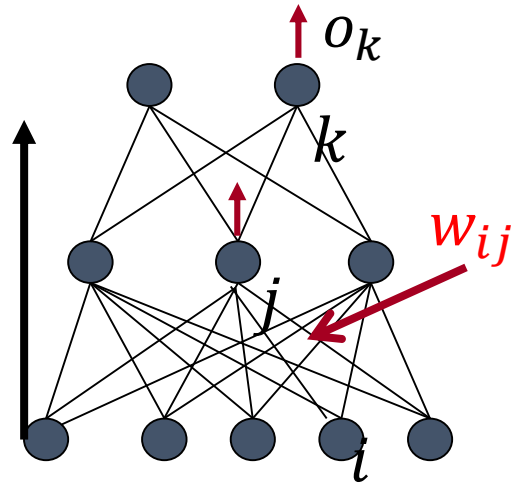
$$= \sum_{k \in parent(j)} \frac{\partial E_d}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial \text{net}_j} x_i$$

$$= \sum_{k \in parent(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} x_i$$



Derivation of Learning Rule (5)

- Weights of hidden units:
  - w_{ij} influences the output only through all the units whose direct input include j



$$\frac{\partial E_d}{\partial w_{ij}} = \sum_{k \in parent(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} x_i =$$

$$= \sum_{k \in parent(j)} -\delta_k \frac{\partial \text{net}_k}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} x_i$$

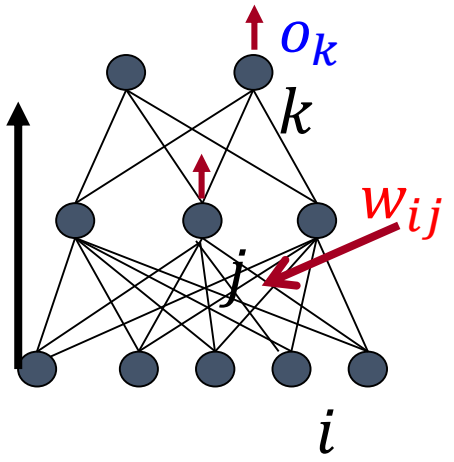
$$= \sum_{k \in parent(j)} -\delta_k \; w_{jk} \; o_j(1 - o_j) \; x_i$$

Derivation of Learning Rule (6)

- Weights of hidden units:
  - w_{ij} is changed by:

$$\Delta w_{ij} = \eta o_j (1 - o_j) . \left(\sum_{k \in parent(j)} -\delta_k \ w_{jk} \right) x_i$$

$$= \eta \delta_j x_i$$



- Where

$$\delta_j = o_j (1 - o_j) . \left(\sum_{k \in parent(j)} -\delta_k \ w_{jk} \right)$$

- First determine the error for the output units.
- Then, backpropagate this error layer by layer through the network, changing weights appropriately in each layer.

The Backpropagation Algorithm

- Create a fully connected three layer network. Initialize weights.
- Until all examples produce the correct output within ϵ (or other criteria)

For each example in the training set do:

1. Compute the network output for this example
2. Compute the error between the output and target value

$$\delta_k = (t_k - o_k)o_k(1 - o_k)$$

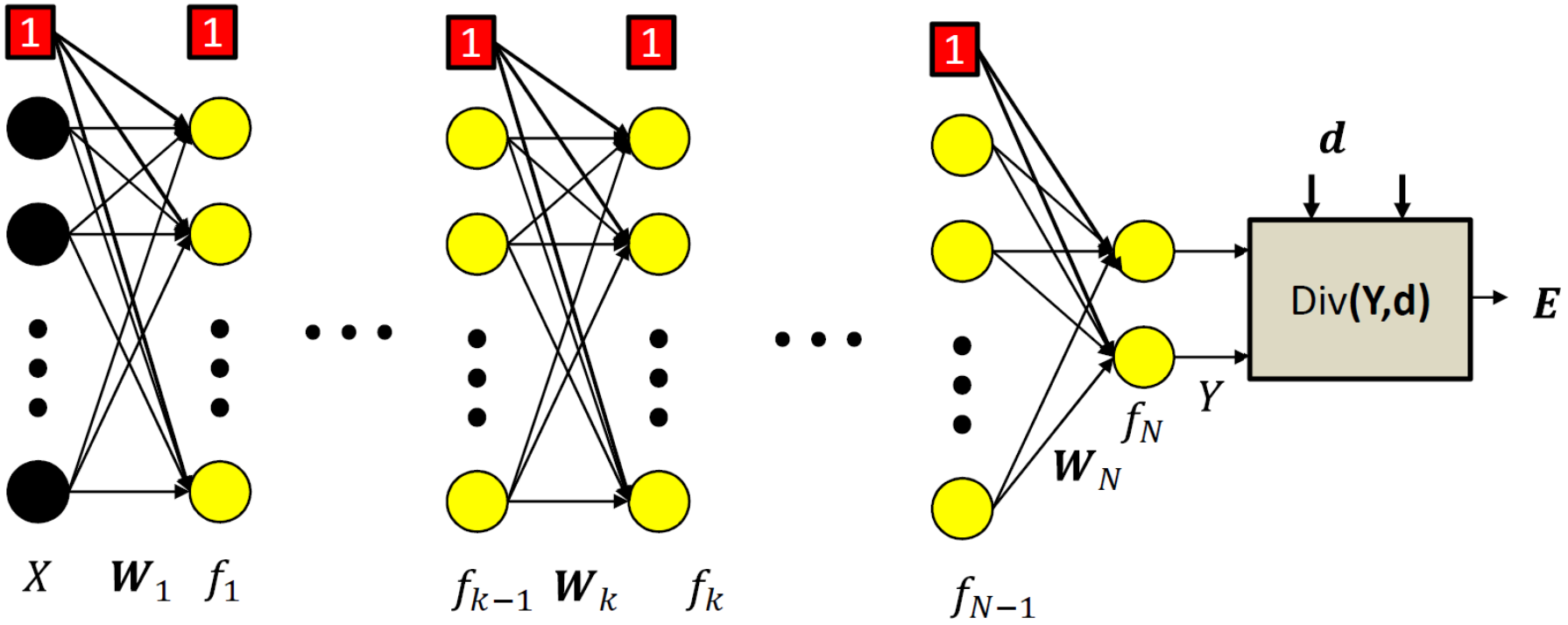
1. For each output unit k, compute error term

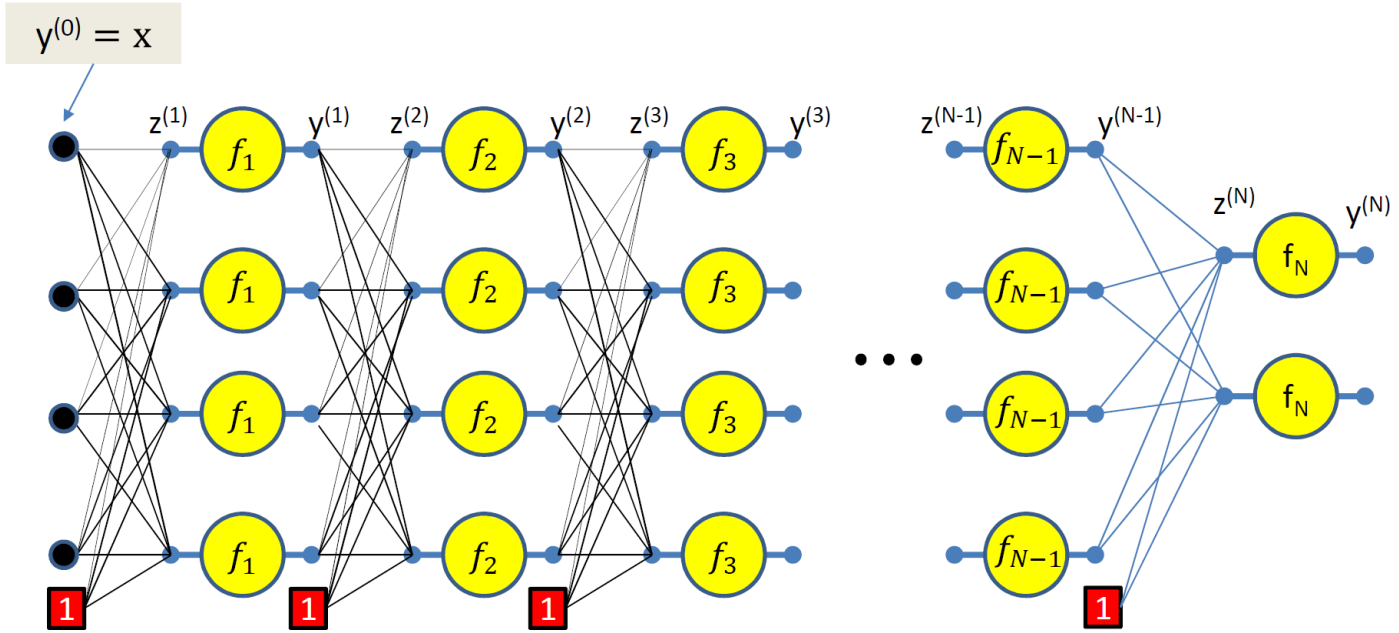
$$\delta_j = o_j(1 - o_j). \sum_{k \in downstream(j)} -\delta_k \, w_{jk}$$

1. For each hidden unit, compute error term: $\Delta w_{ij} = \eta \delta_j x_i$
2. Update network weights with Δw_{ij}

End epoch

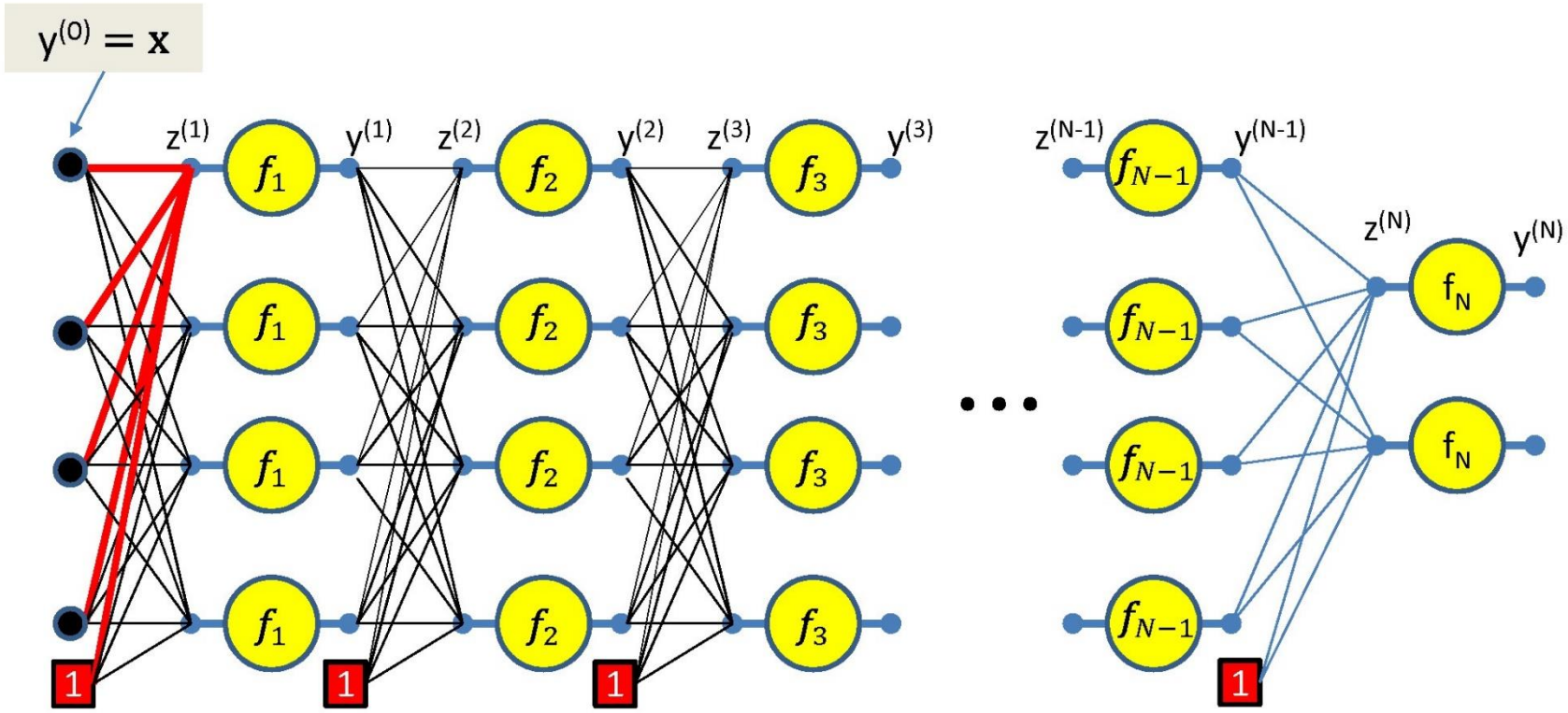
The network again: modular view





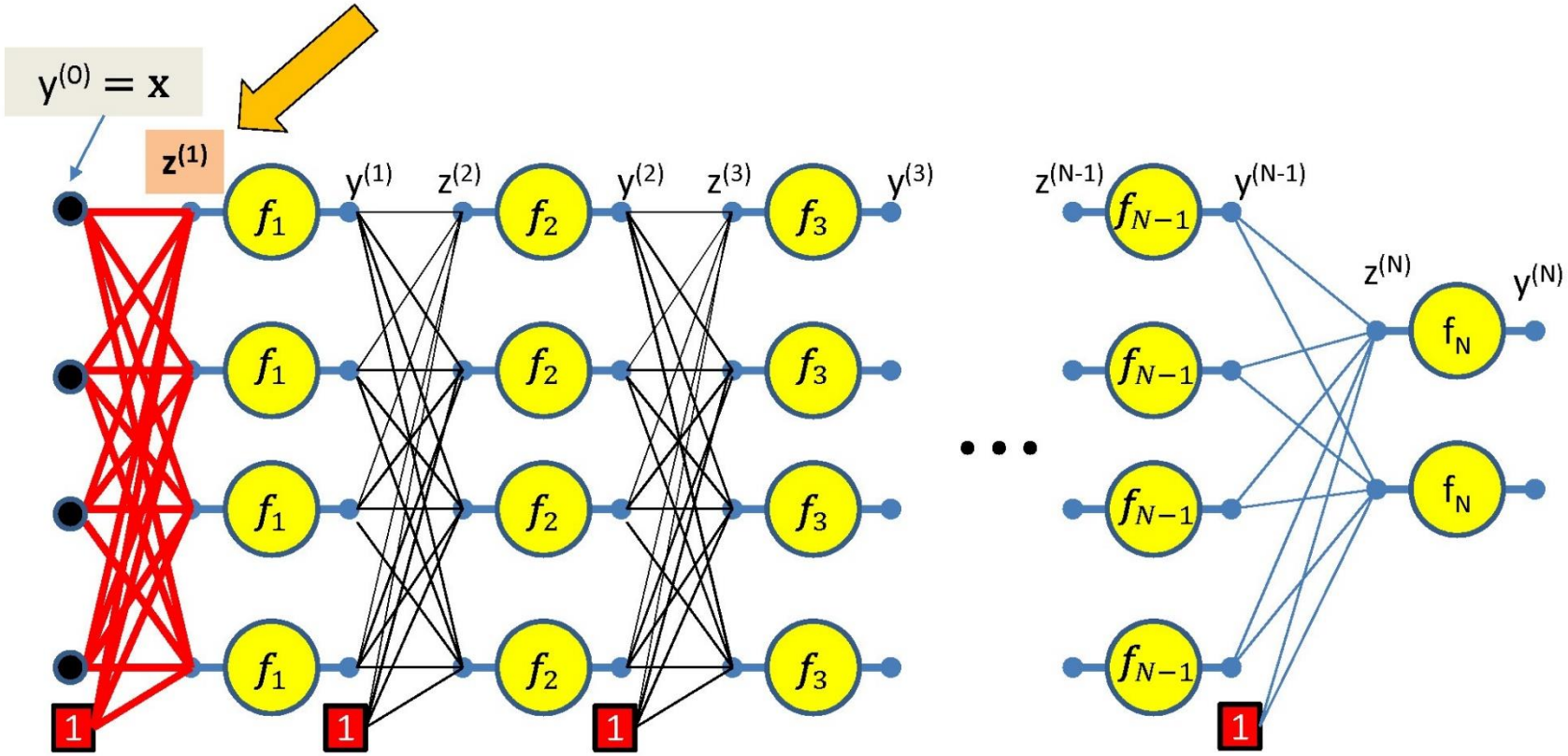
- Setting $y_i^{(0)} = x_i$ for notational convenience
- Assuming $w_{0j}^{(k)} = b_j^{(k)}$ and $y_0^{(k)} = 1$
- Assuming the bias is a weight and extending the output of every layer by a constant 1, to account for the biases

Forward Computation



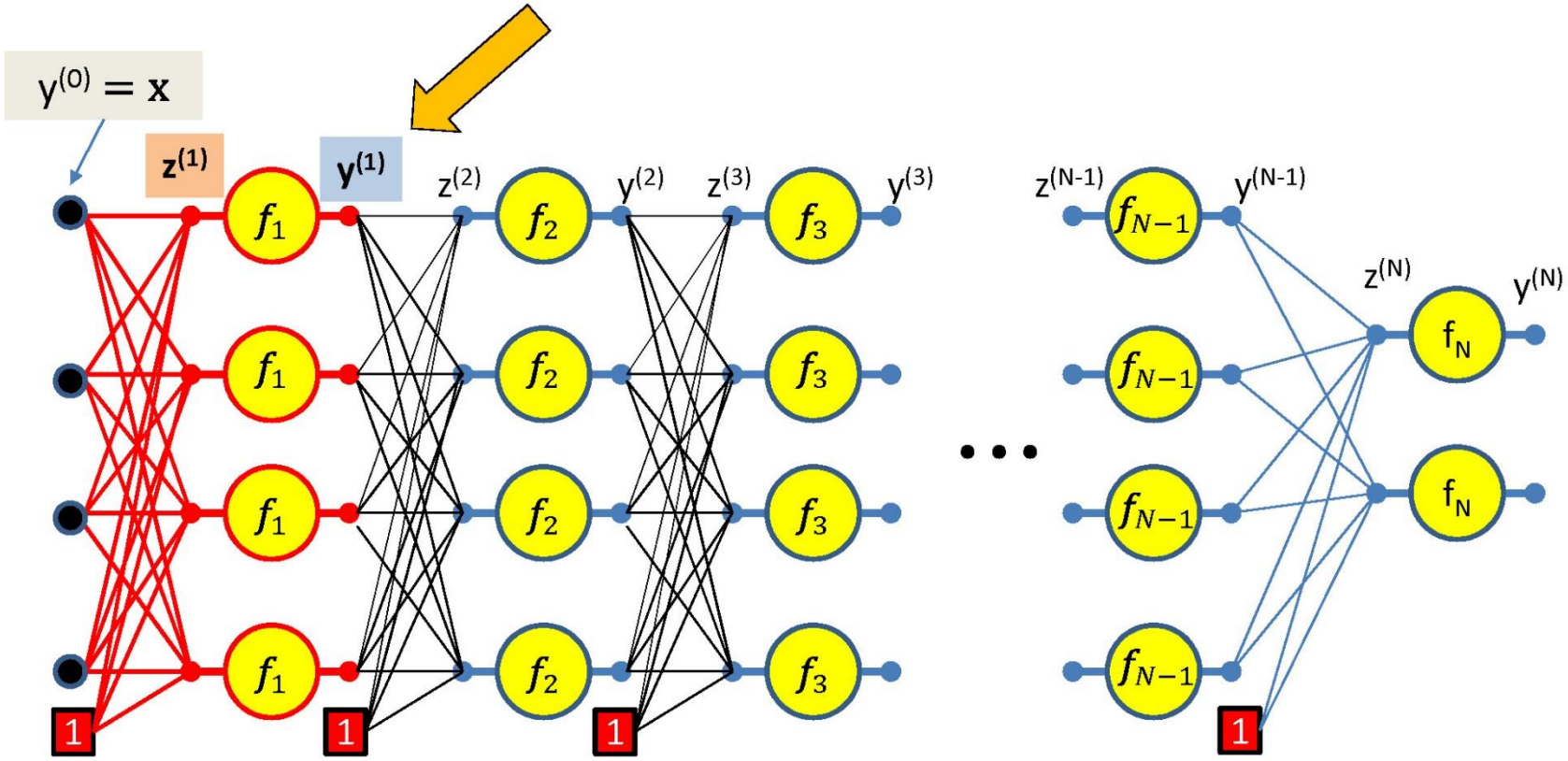
$$z_1^{(1)} = \sum_i w_{i1}^{(1)} y_i^{(0)}$$

Forward Computation



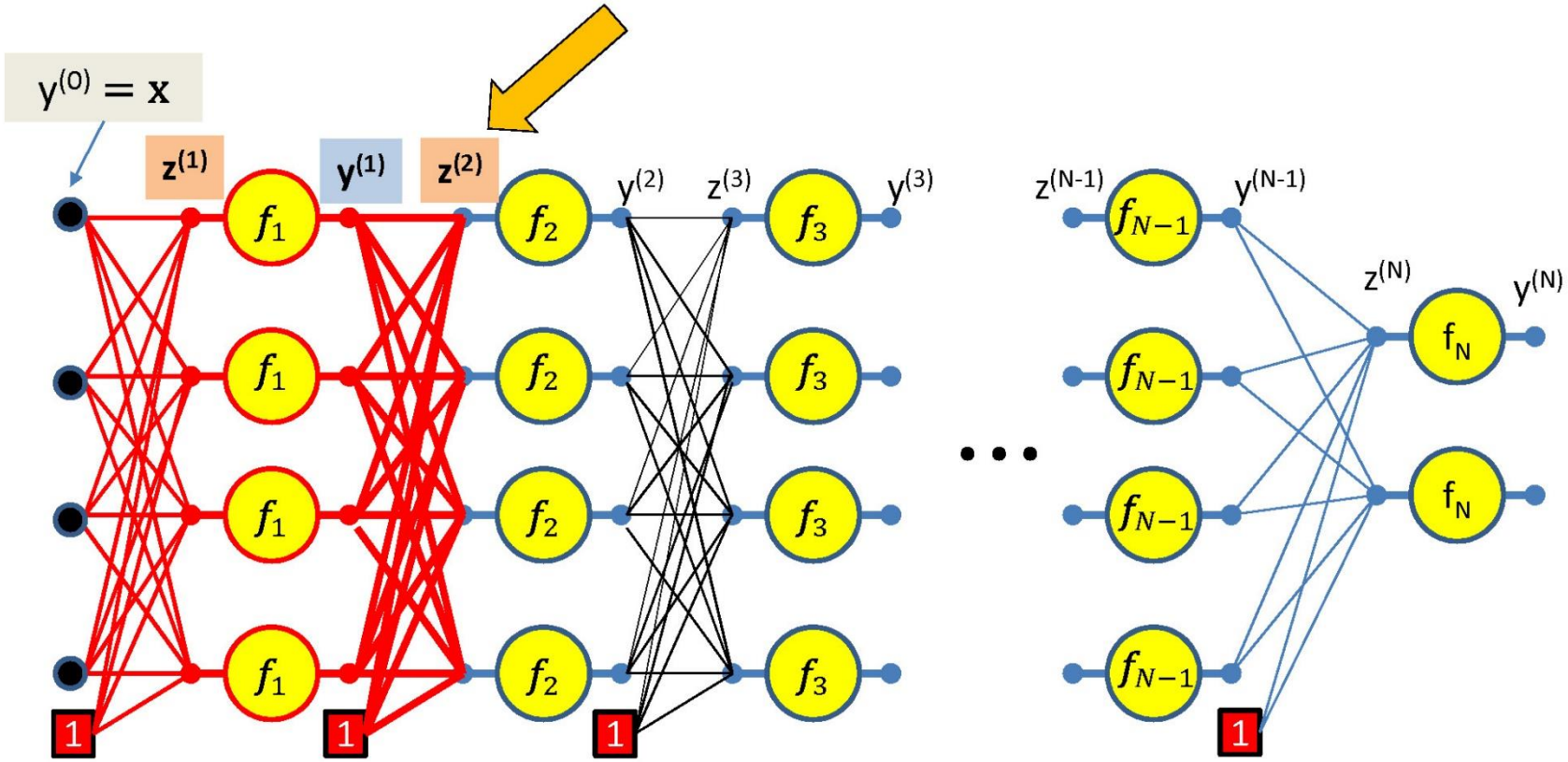
$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

Forward Computation



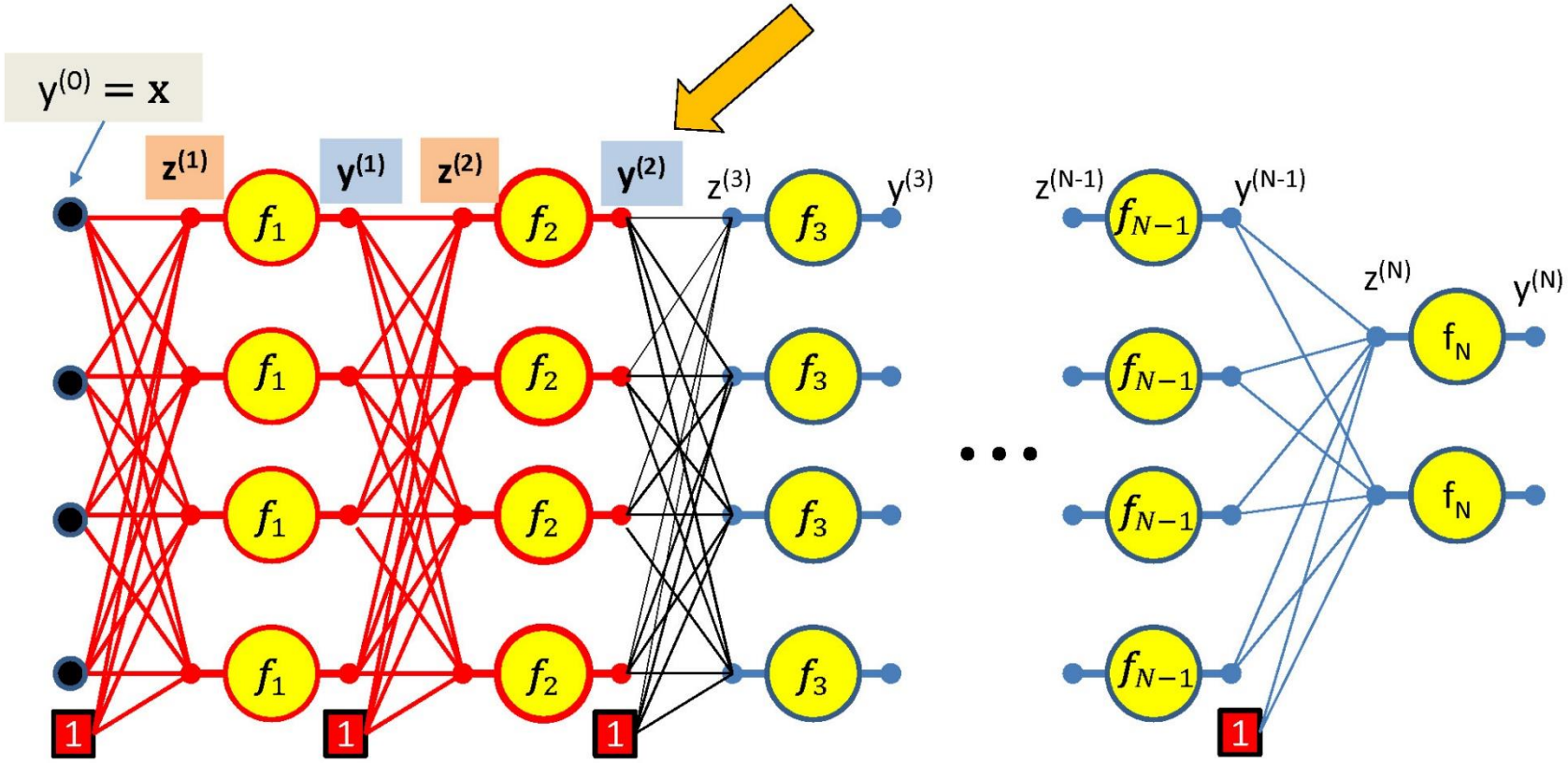
$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)} \qquad y_j^{(1)} = f_1\left(z_j^{(1)}\right)$$

Forward Computation



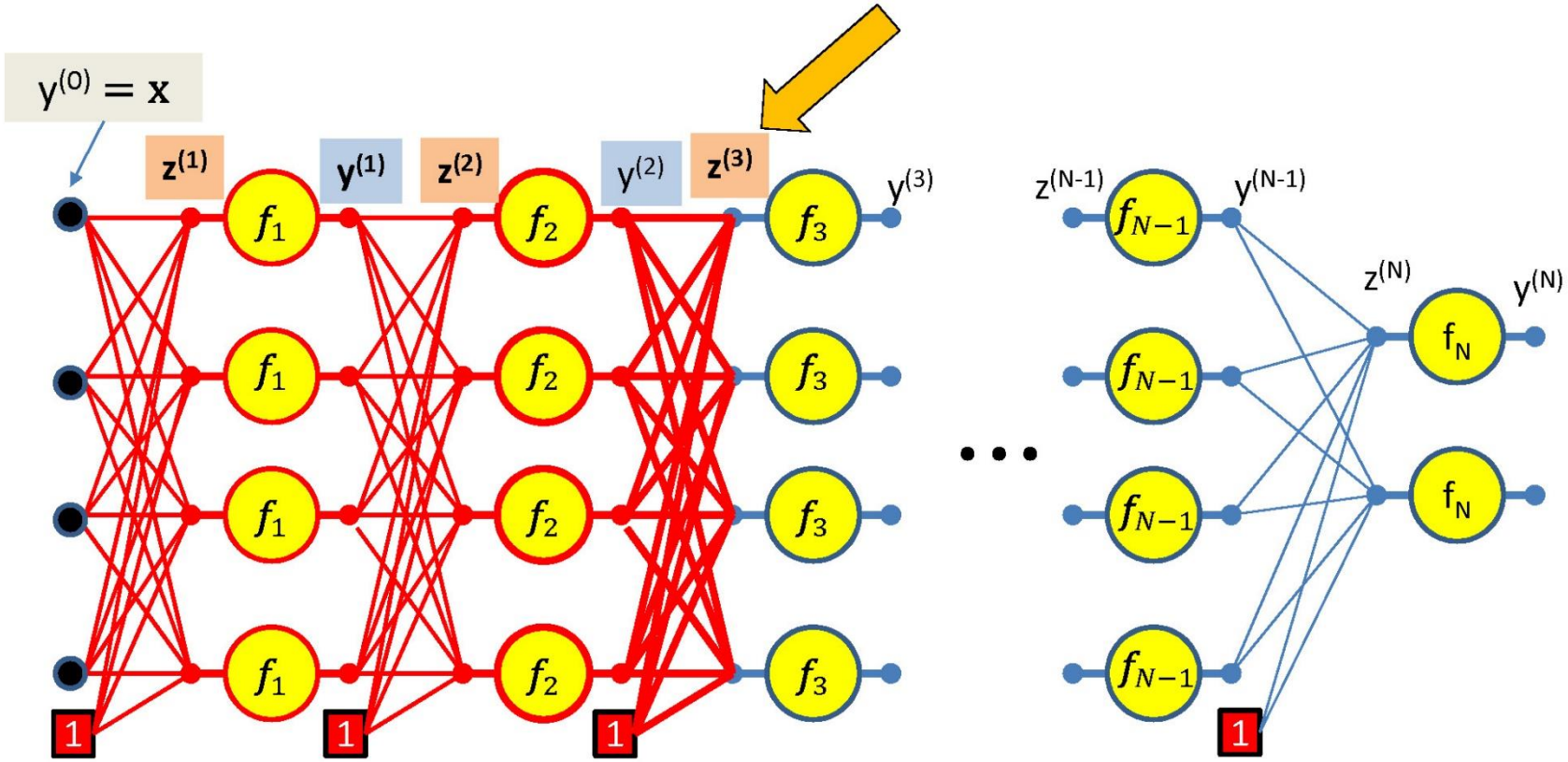
$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)} \qquad y_j^{(1)} = f_1\left(z_j^{(1)}\right) \qquad z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)}$$

Forward Computation



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

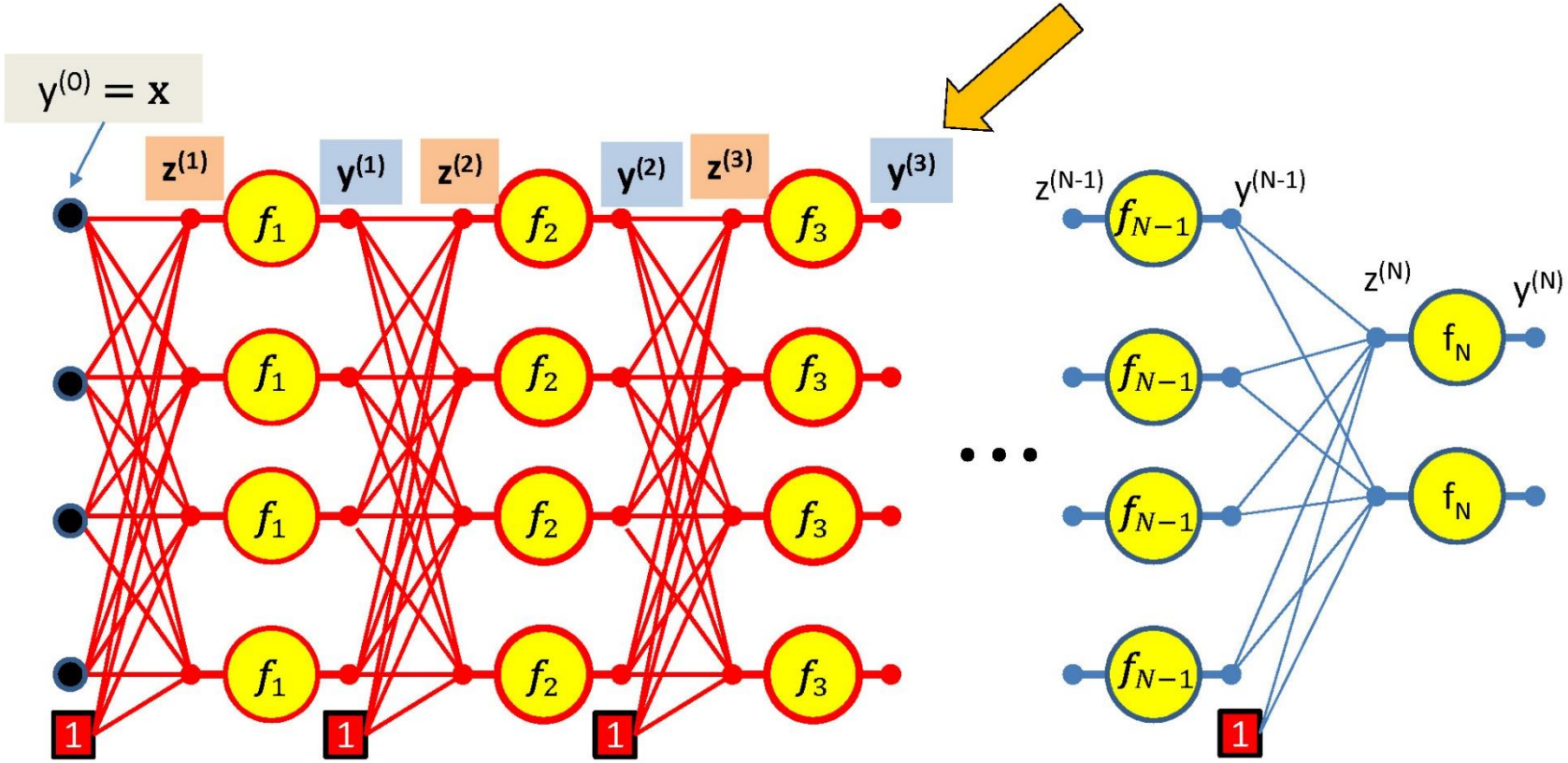
Forward Computation



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

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

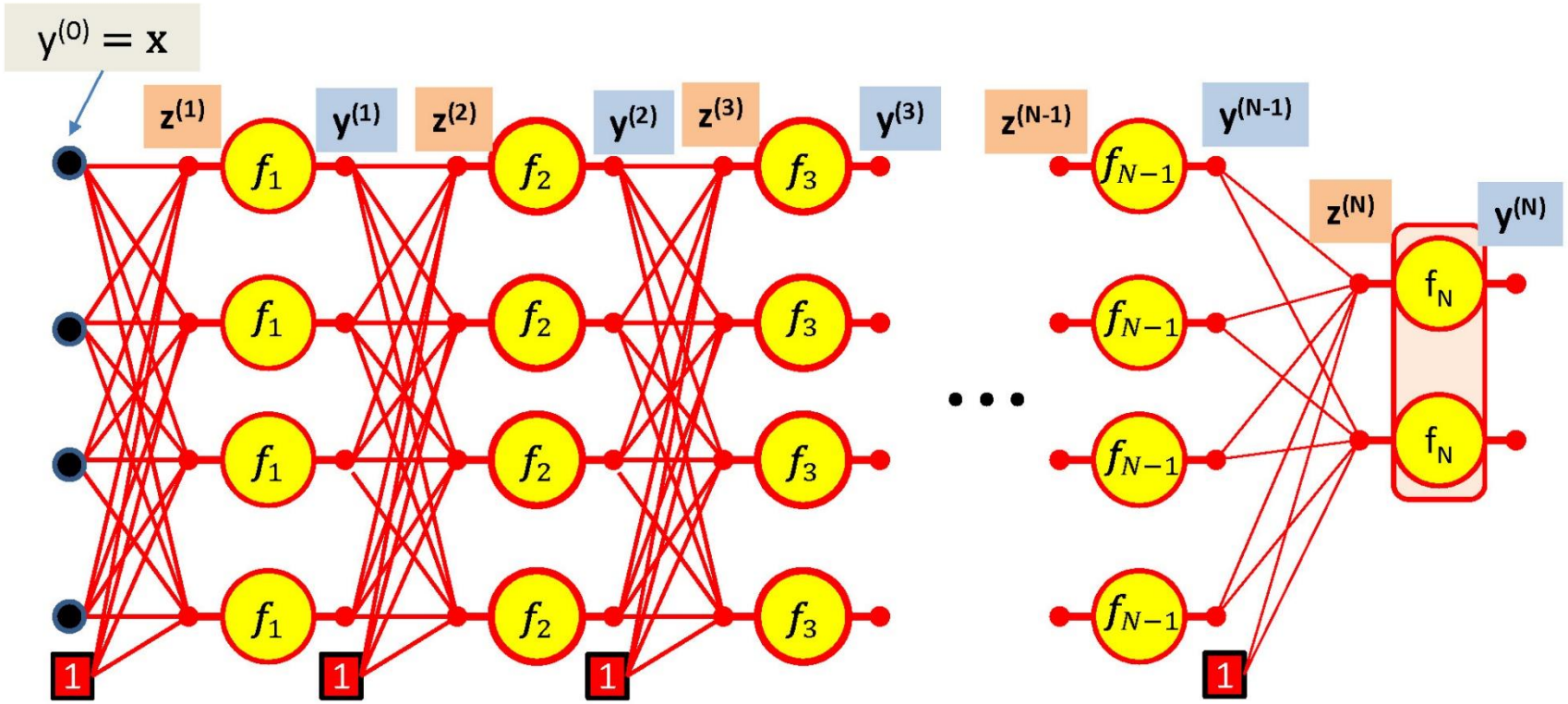
Forward Computation



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

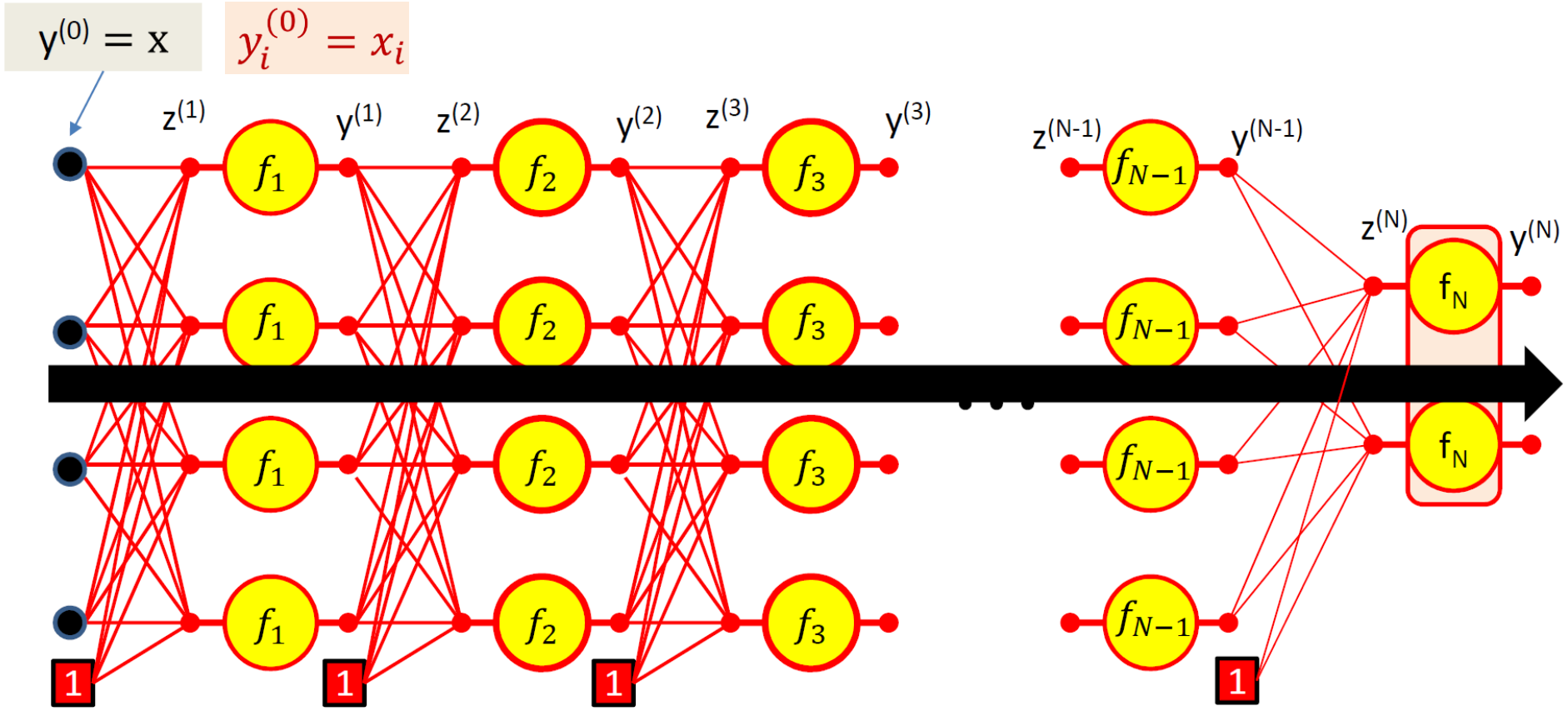
$$z_j^{(3)} = \sum_i w_{ij}^{(3)} y_i^{(2)} \qquad \boxed{y_j^{(3)} = f_3\left(z_j^{(3)}\right)} \qquad \cdots$$

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



- 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 $\mathbf{x} = [x_j,\ j = 1 \ldots D]$

Set:

- $D_0 = D$, is the width of the 0th (input) layer
- $y_j^{(0)} = x_j,\ j = 1 \ldots D;\quad y_0^{(k=1\ldots N)} = x_0 = 1$

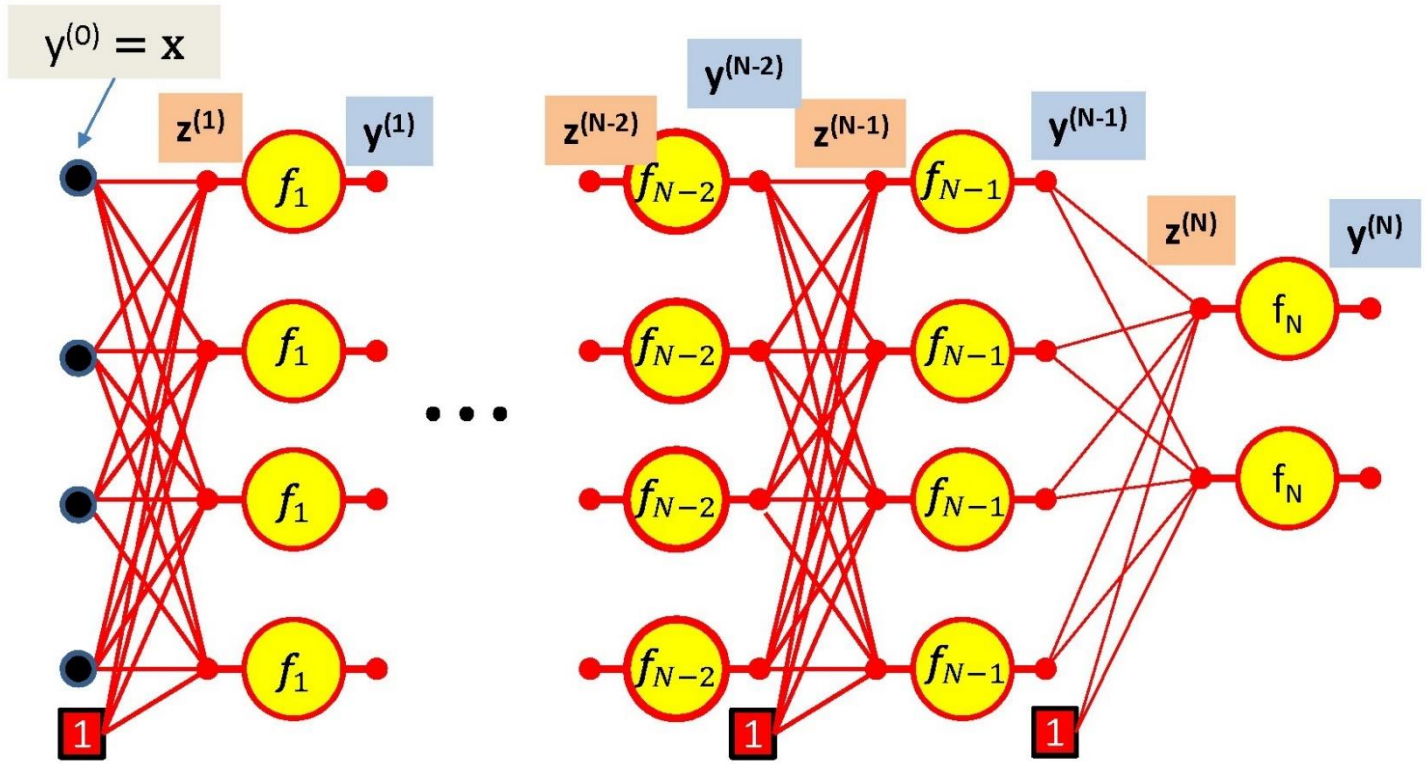
For layer $k = 1 \ldots N$

- For $j = 1 \ldots D_k$ 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:

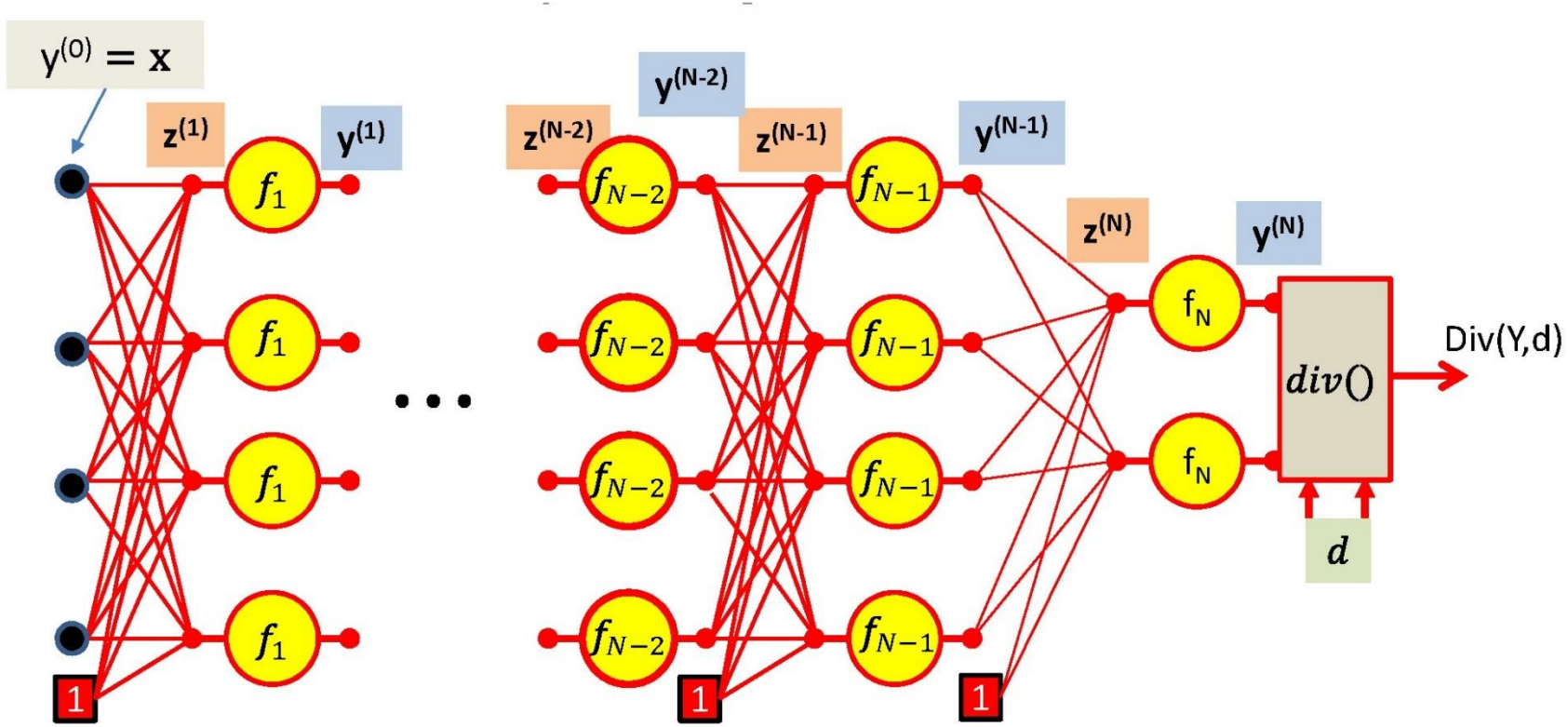
- $Y = y_j^{(N)}, j = 1..D_N$

Computing derivatives



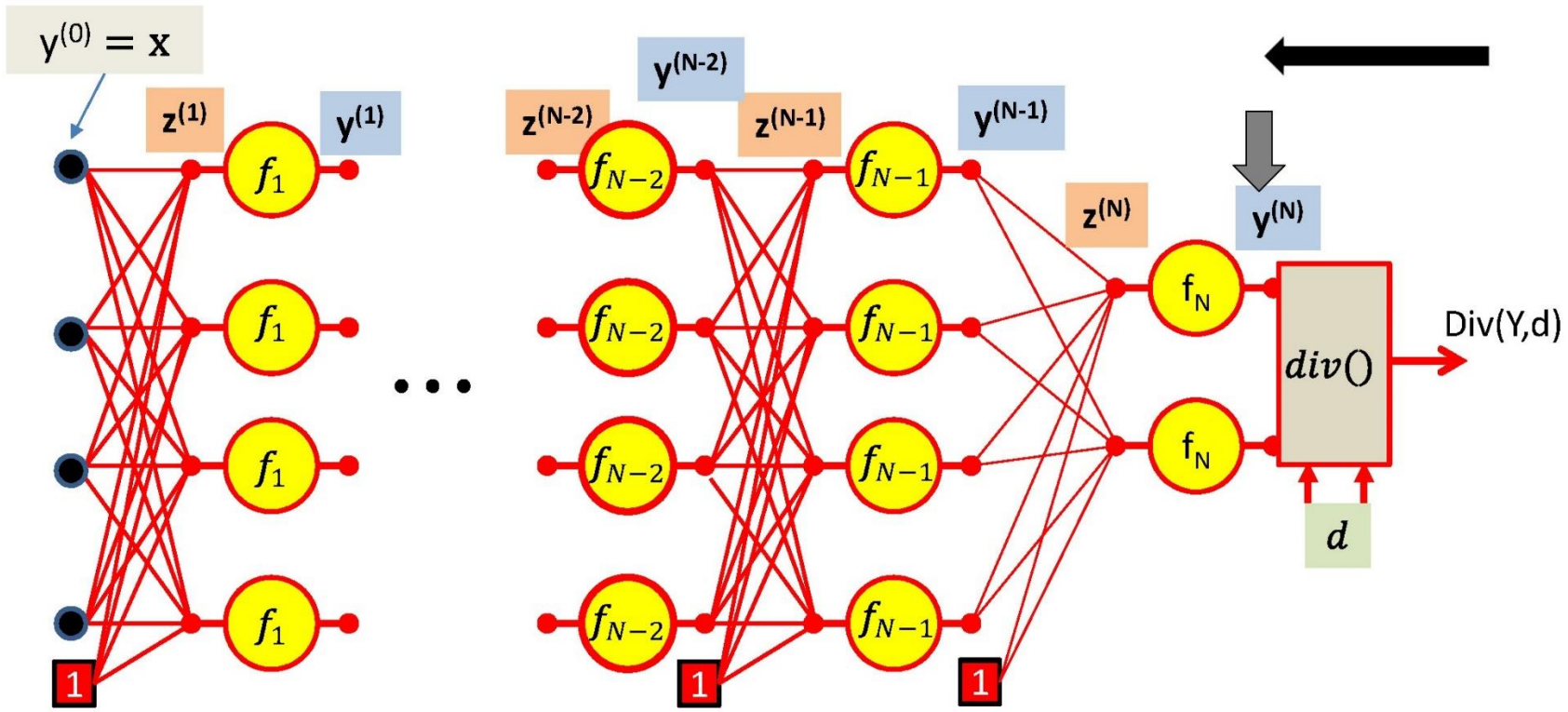
- We have computed all these intermediate values in the forward computation
- We must remember them – we will need them to compute the derivatives

Computing derivatives



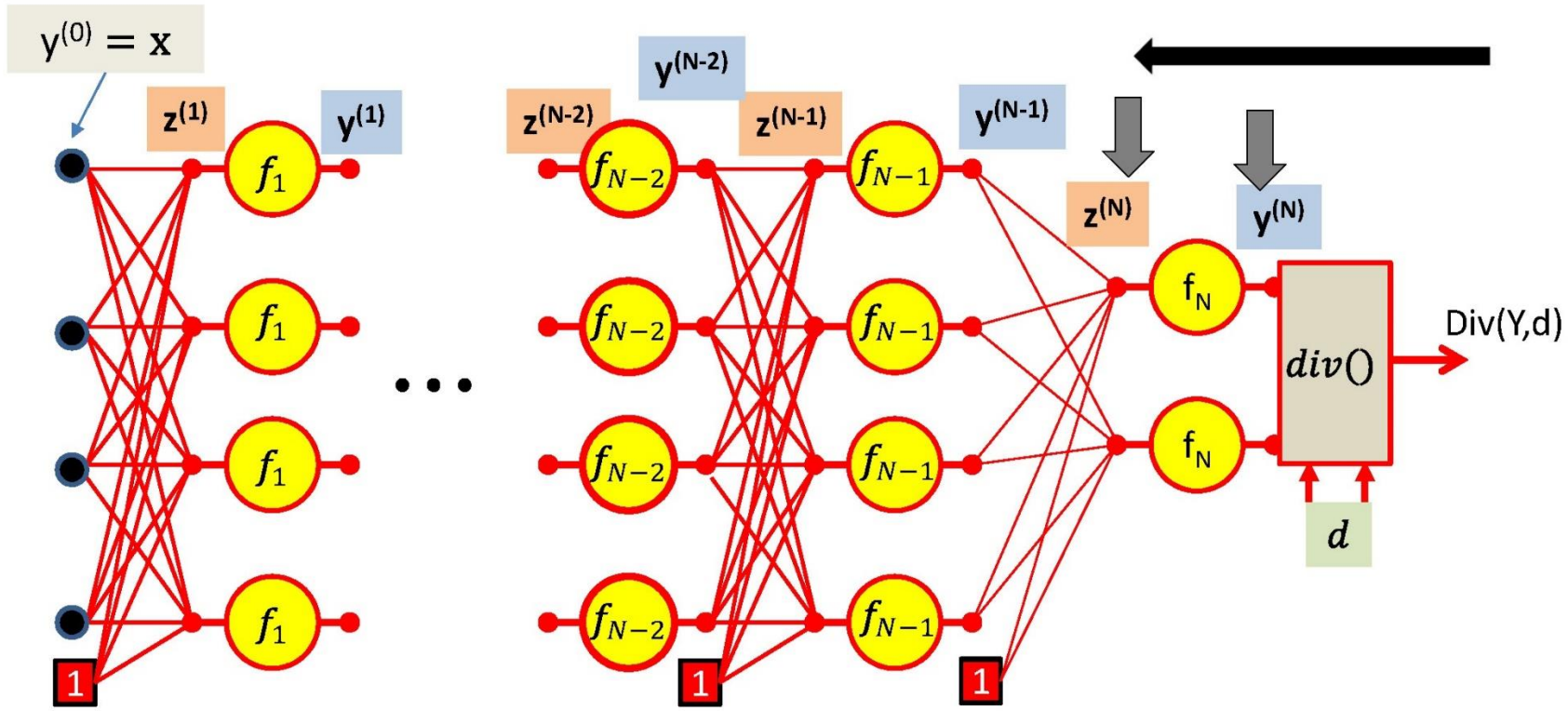
- First, we compute the divergence between the output of the net $y = y^{(N)}$ and the desired output d

Computing derivatives



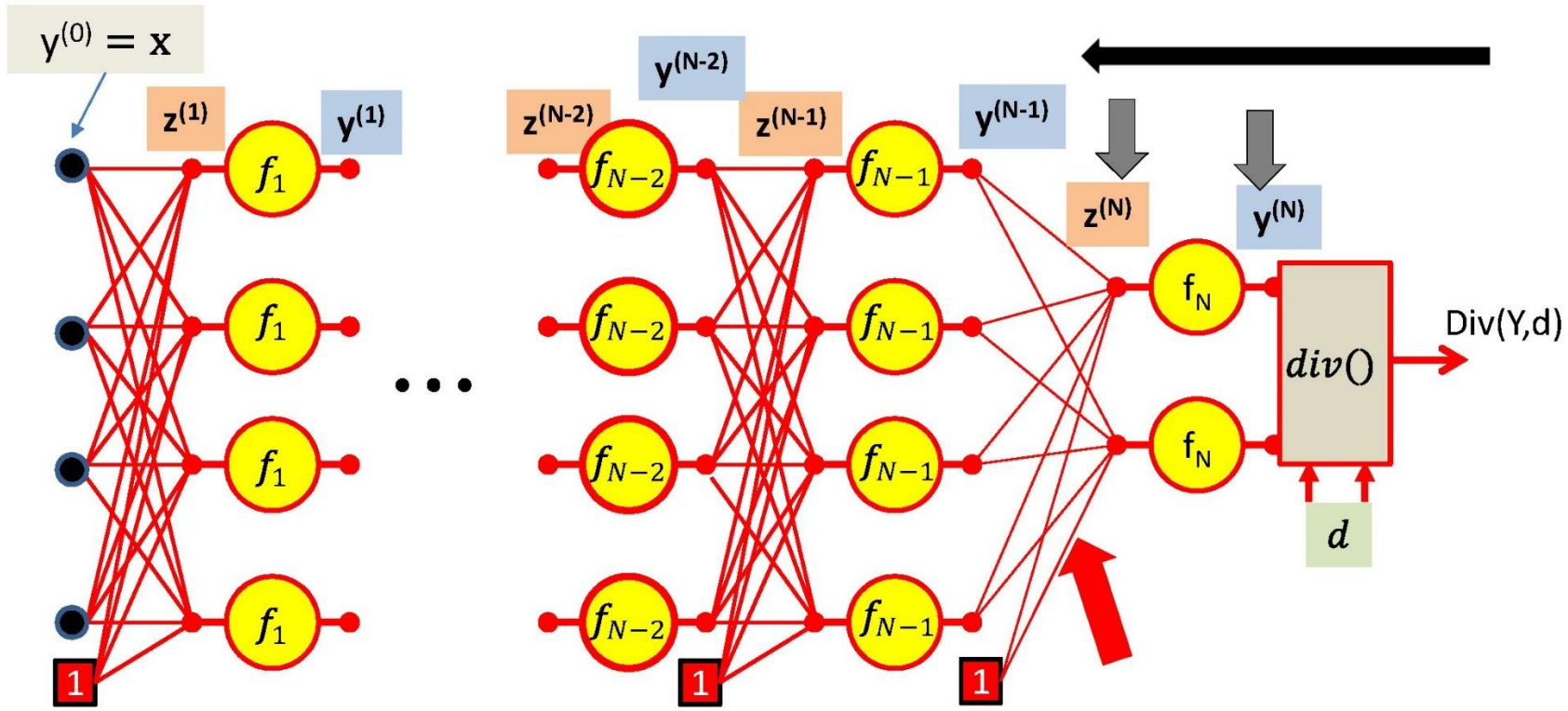
- We then compute $\nabla_{Y(N)}\text{div}(.)$ the derivative of the divergence w.r.t. the final output of the network $y^{(N)}$

Computing derivatives



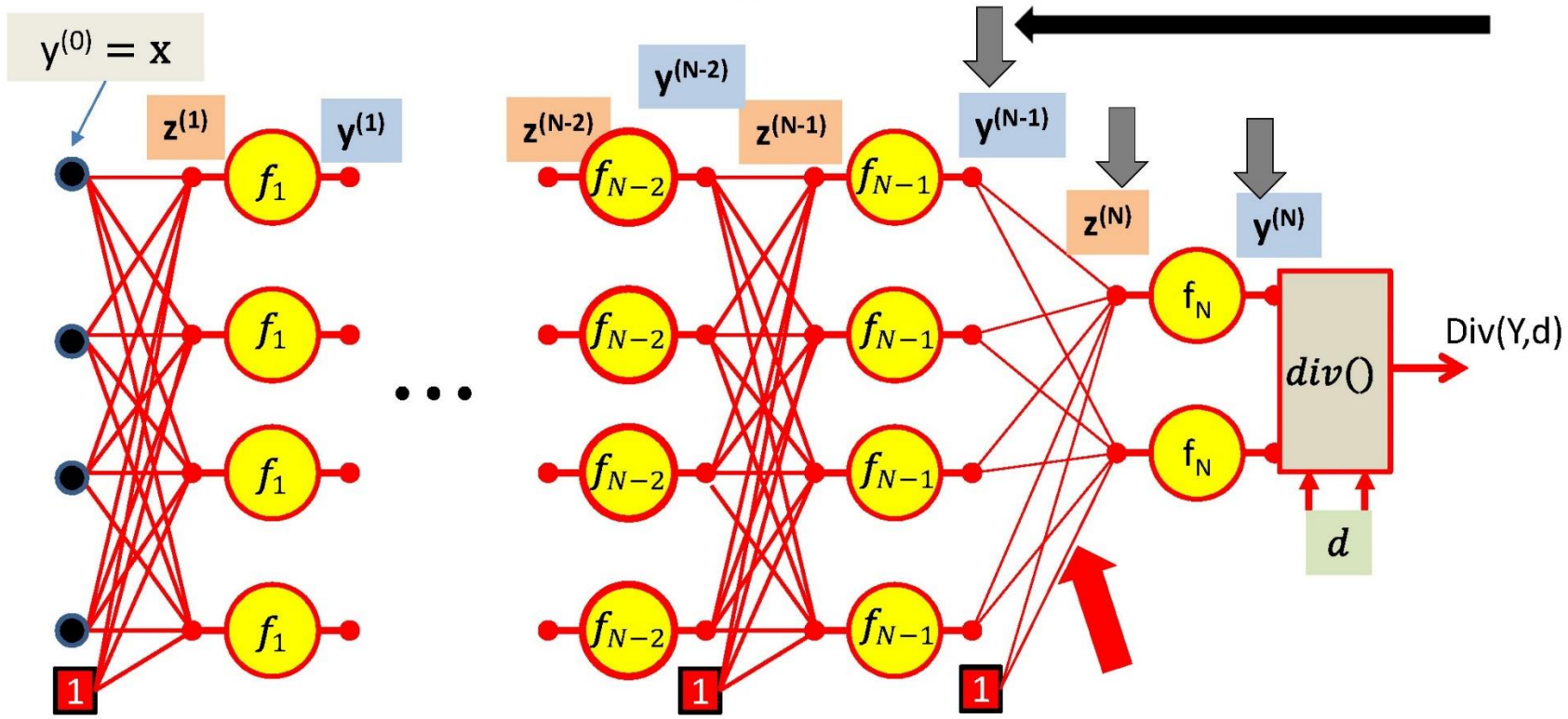
- We then compute $\nabla_{Y(N)}$div(.) the derivative of the divergence w.r.t. the final output of the network y(N)
- We then compute $\nabla_{Z(N)}$div(.) the derivative of the divergence w.r.t. the pre-activation affine combination z(N) using the chain rule

Computing derivatives



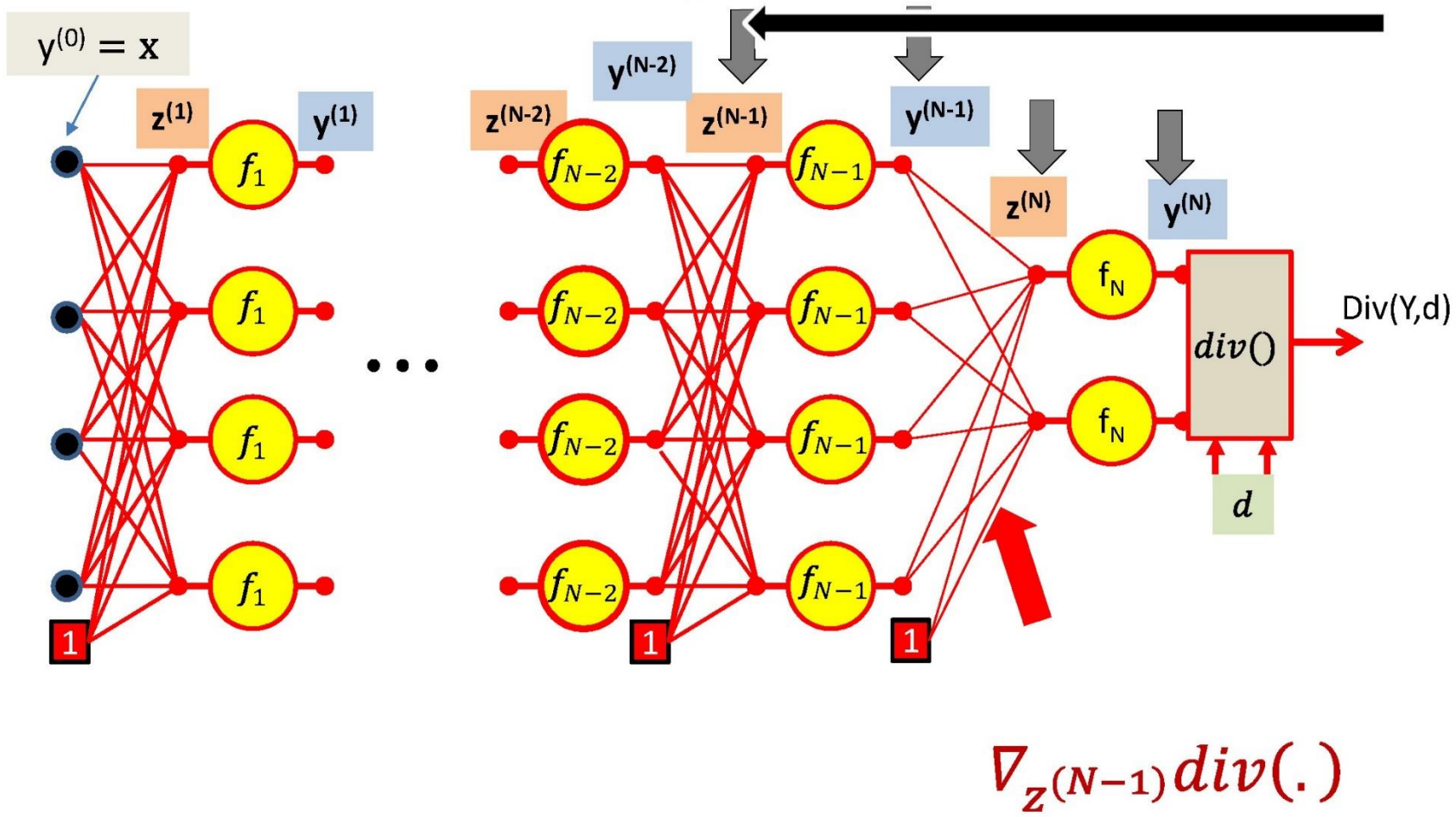
- Continuing on, we will compute $\nabla_{W(N)}$div(.) the derivative of the divergence with respect

Computing derivatives



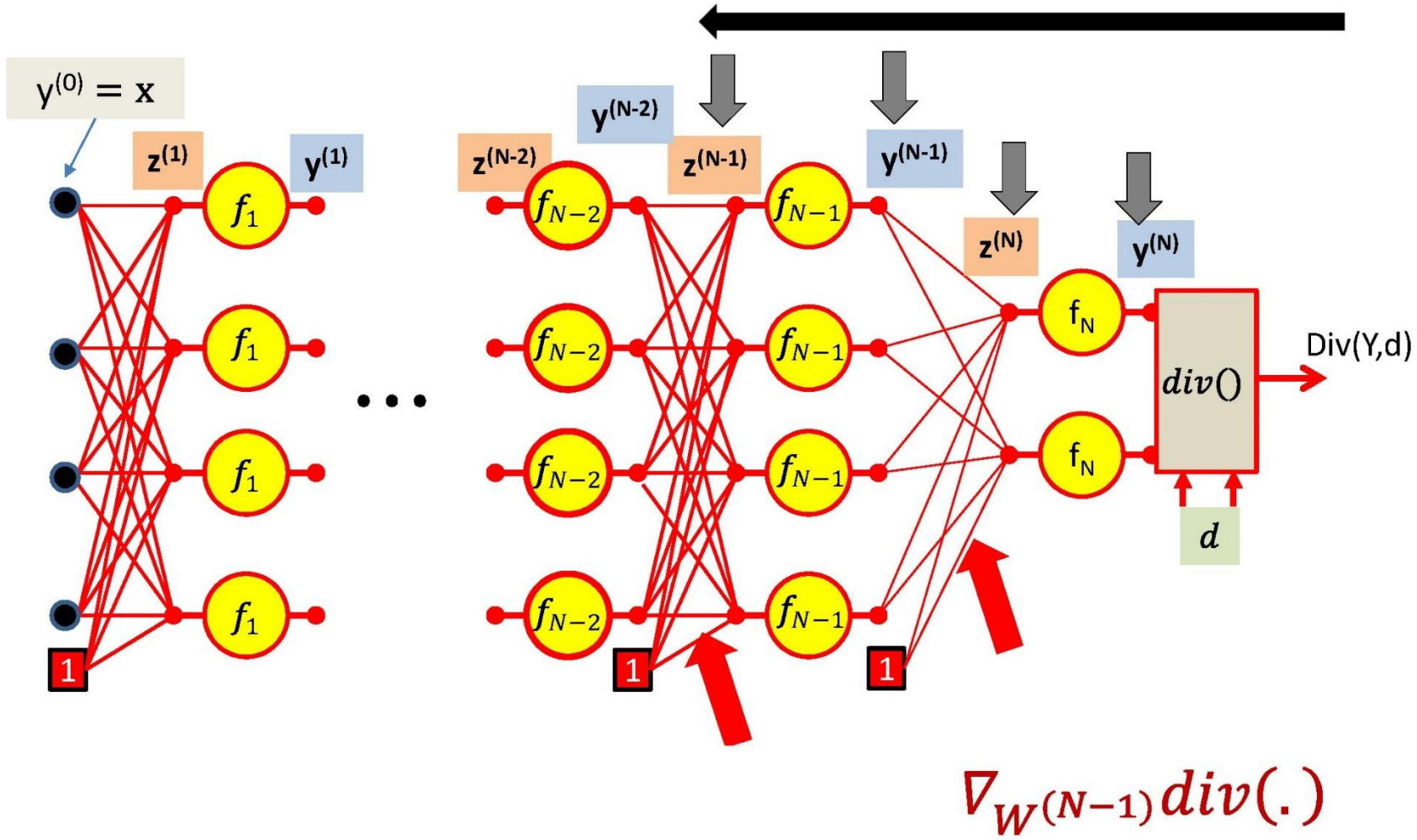
- Continuing on, we will compute $\nabla_{W(N)}\text{div}(.)$ the derivative of the divergence with respect to the weights of the connections to the output layer
- Then continue with the chain rule to compute $\nabla_{Y(N-1)}\text{div}(.)$ the derivative of the divergence w.r.t. the output of the N-1th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

Computing derivatives



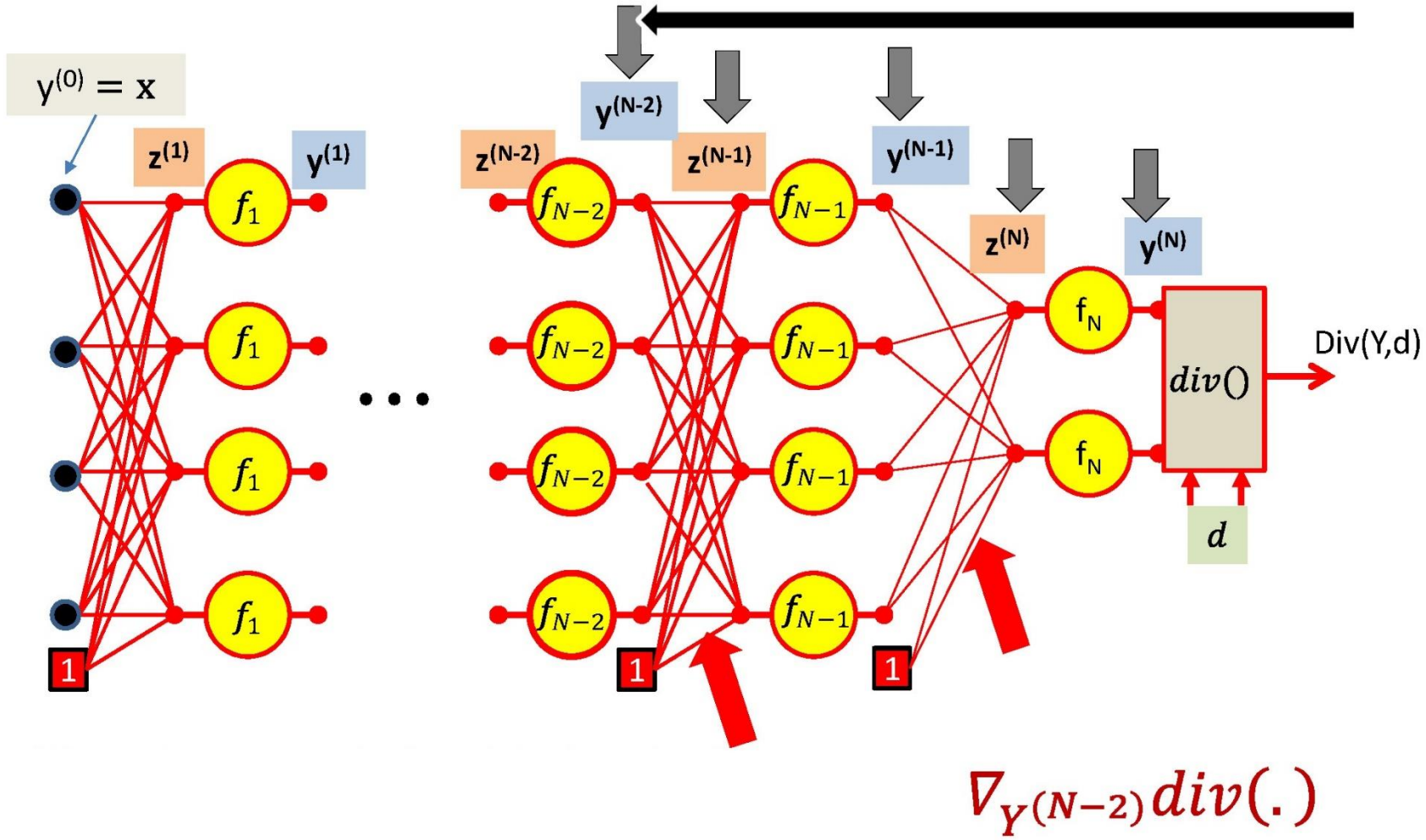
- We continue our way backwards in the order shown

Computing derivatives



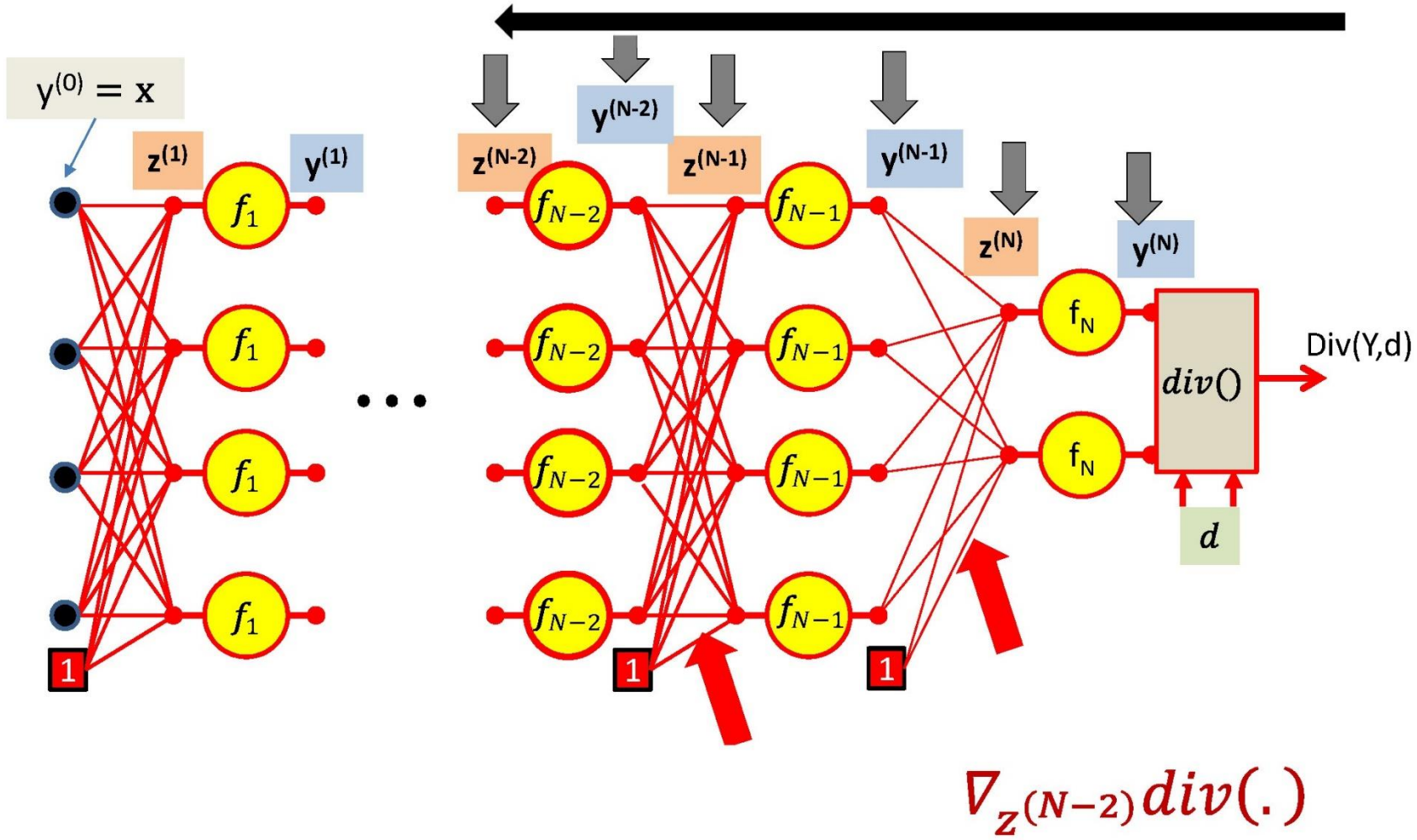
- We continue our way backwards in the order shown

Computing derivatives



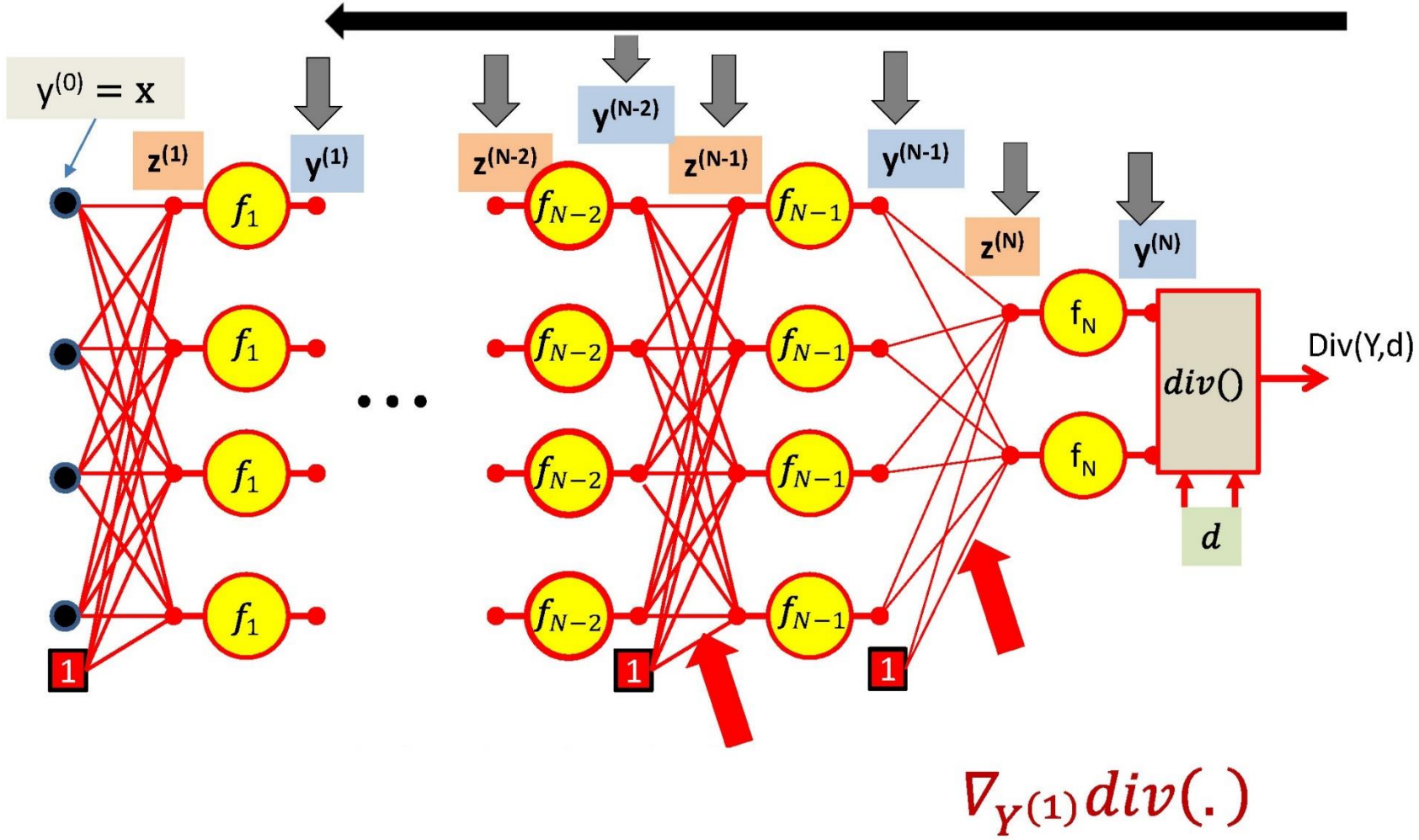
- We continue our way backwards in the order shown

Computing derivatives



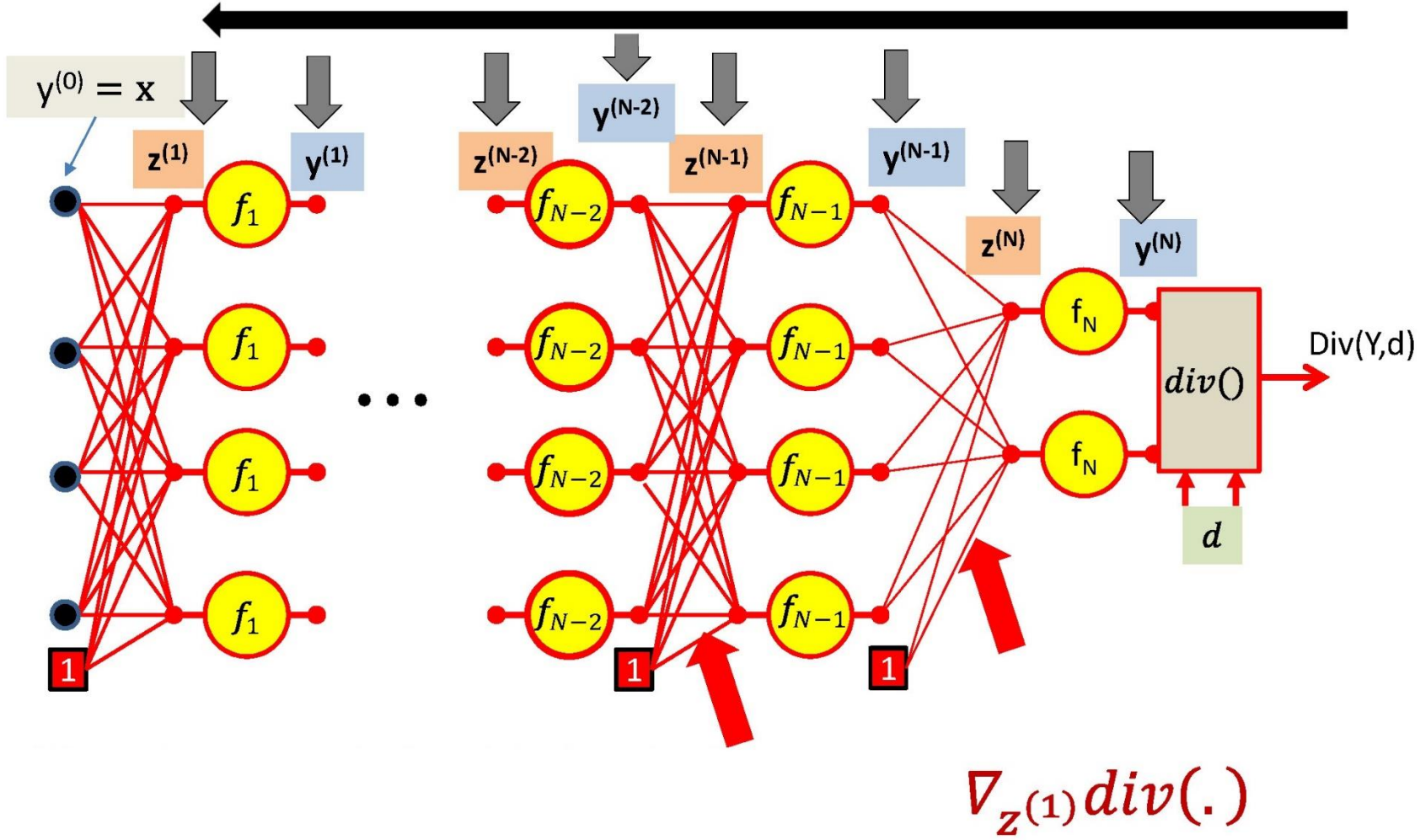
- We continue our way backwards in the order shown

Computing derivatives



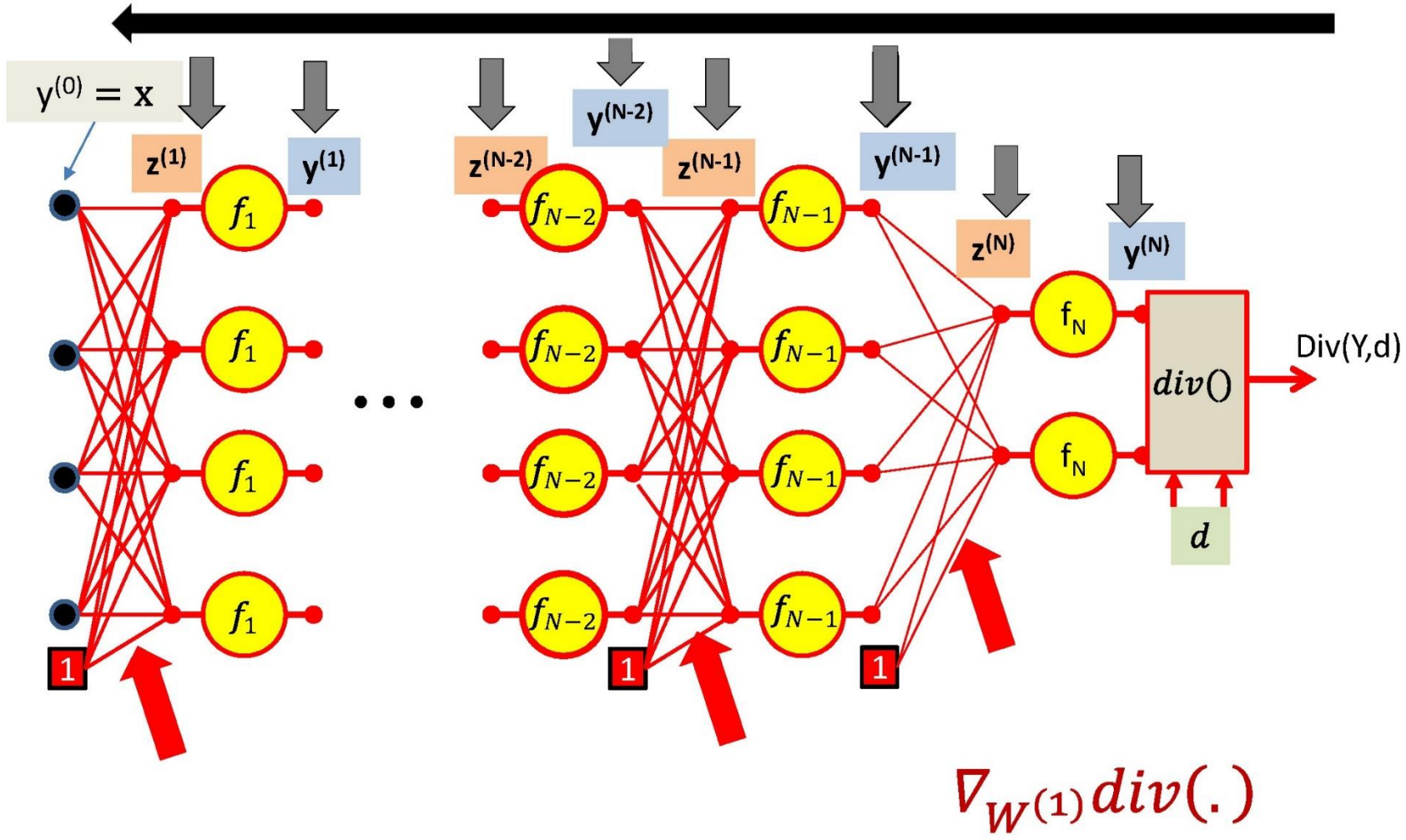
- We continue our way backwards in the order shown

Computing derivatives



- We continue our way backwards in the order shown

Computing derivatives

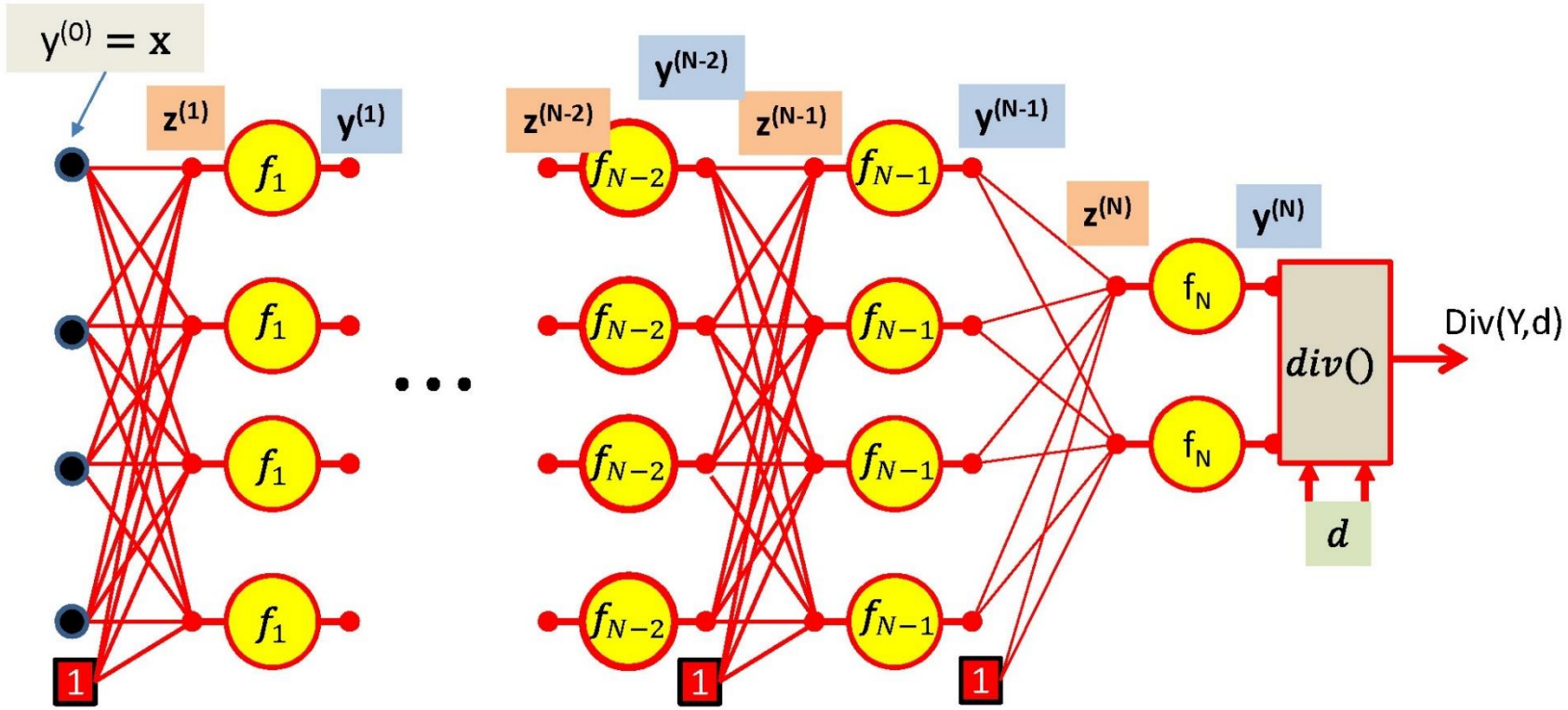


- We continue our way backwards in the order shown

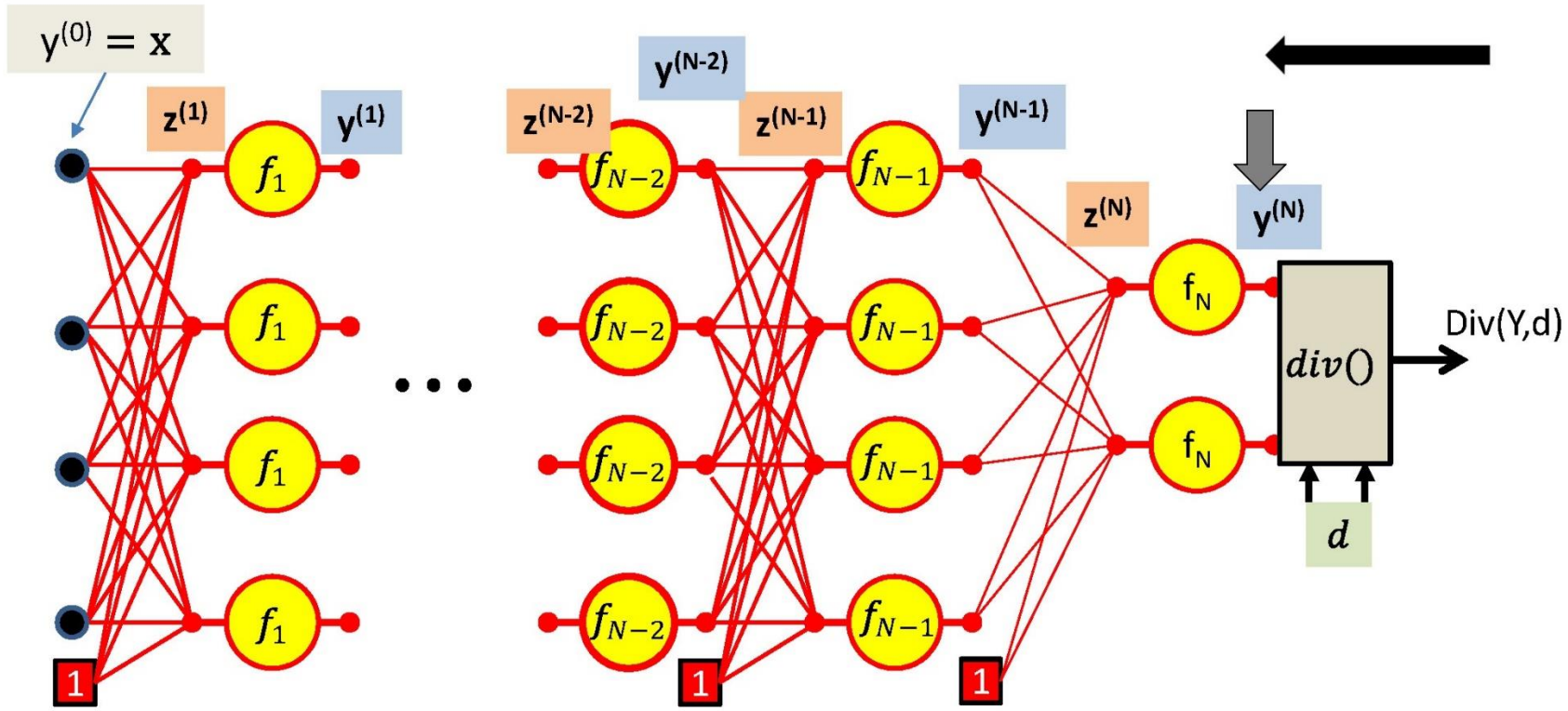
Backward Gradient Computation

- Lets actually see the math …

Backward Gradient Computation



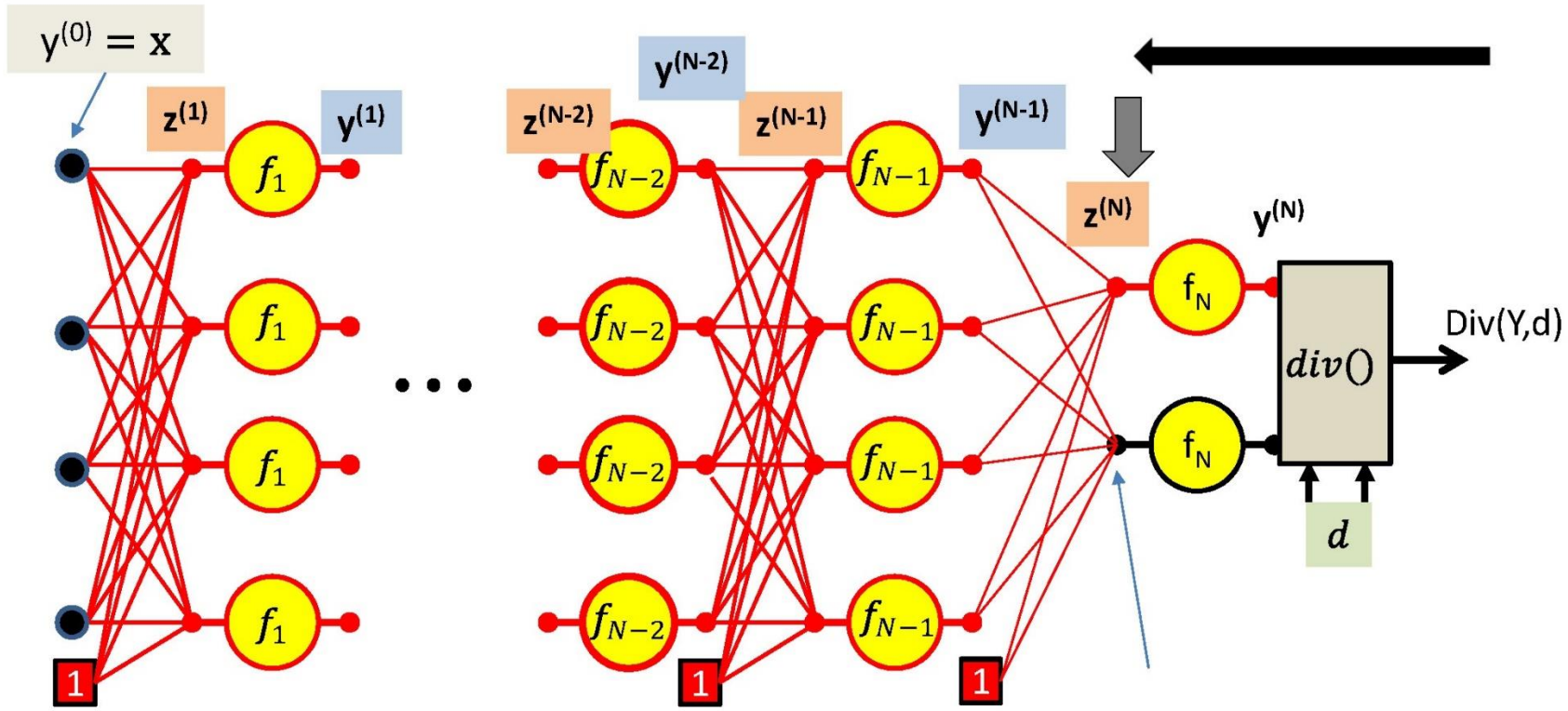
Backward Gradient Computation



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

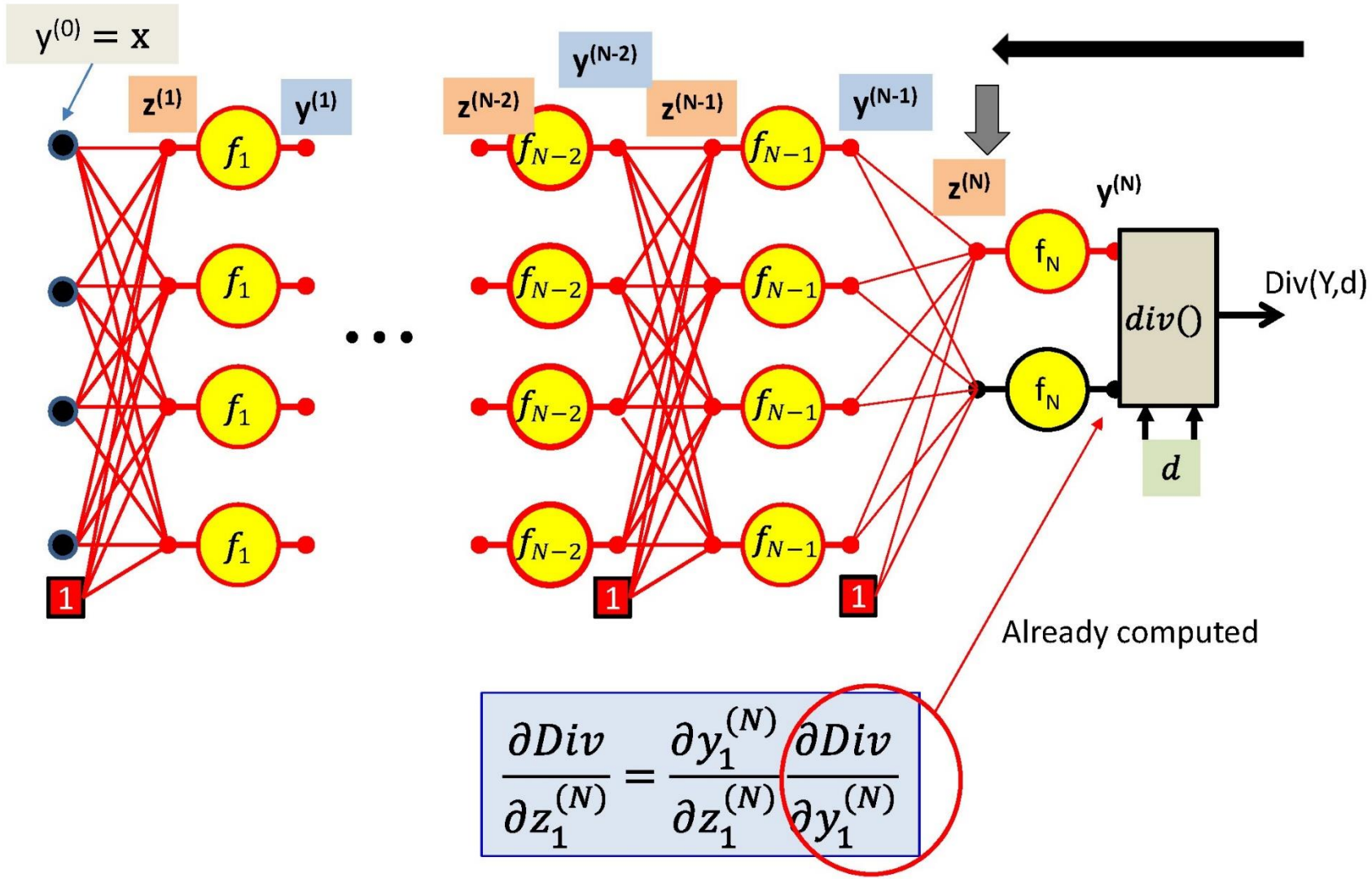
$$\frac{\partial Div(Y,d)}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

Backward Gradient Computation

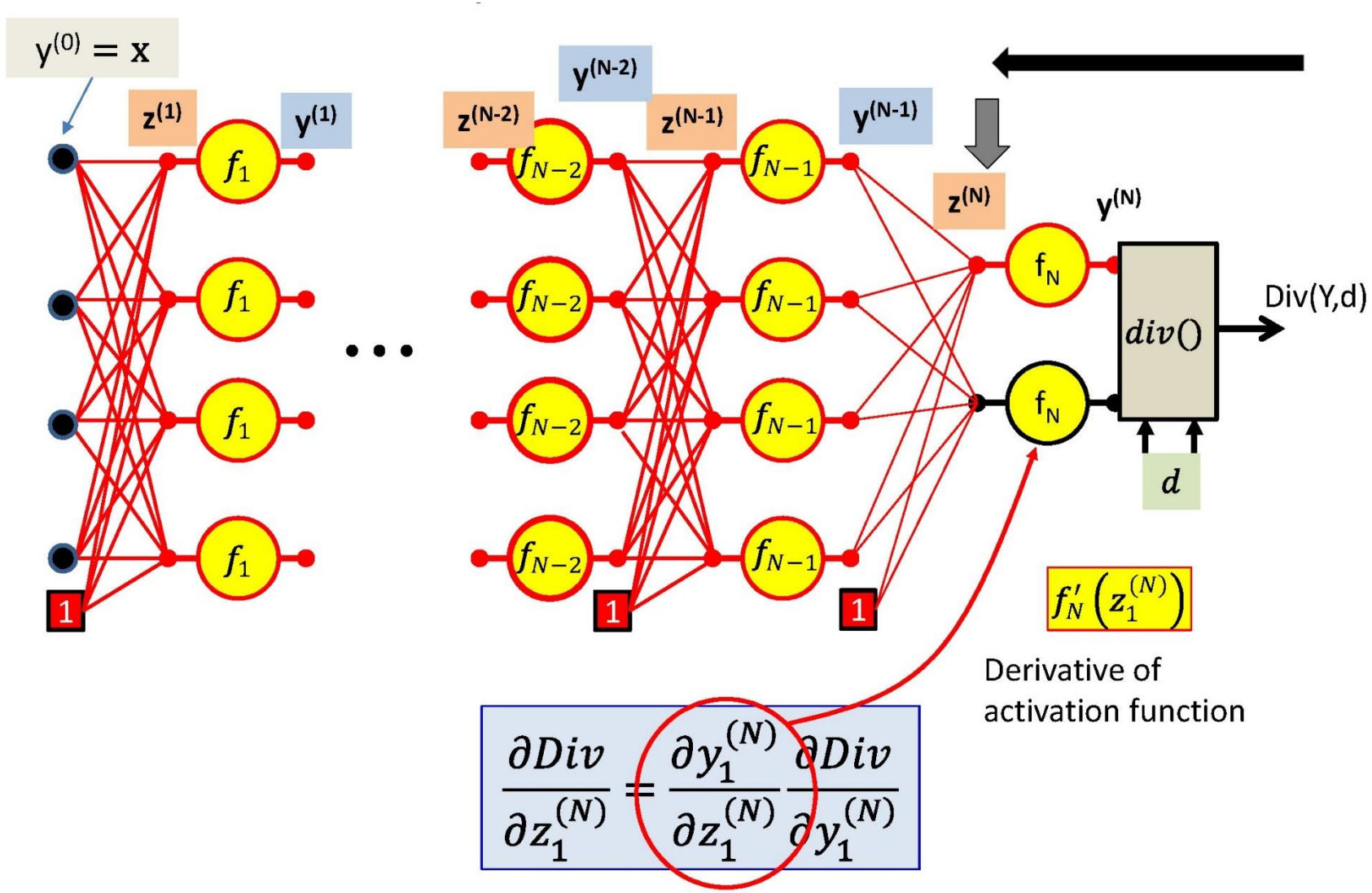


$$\frac{\partial Div}{\partial z_1^{(N)}} = \frac{\partial y_1^{(N)}}{\partial z_1^{(N)}} \frac{\partial Div}{\partial y_1^{(N)}}$$

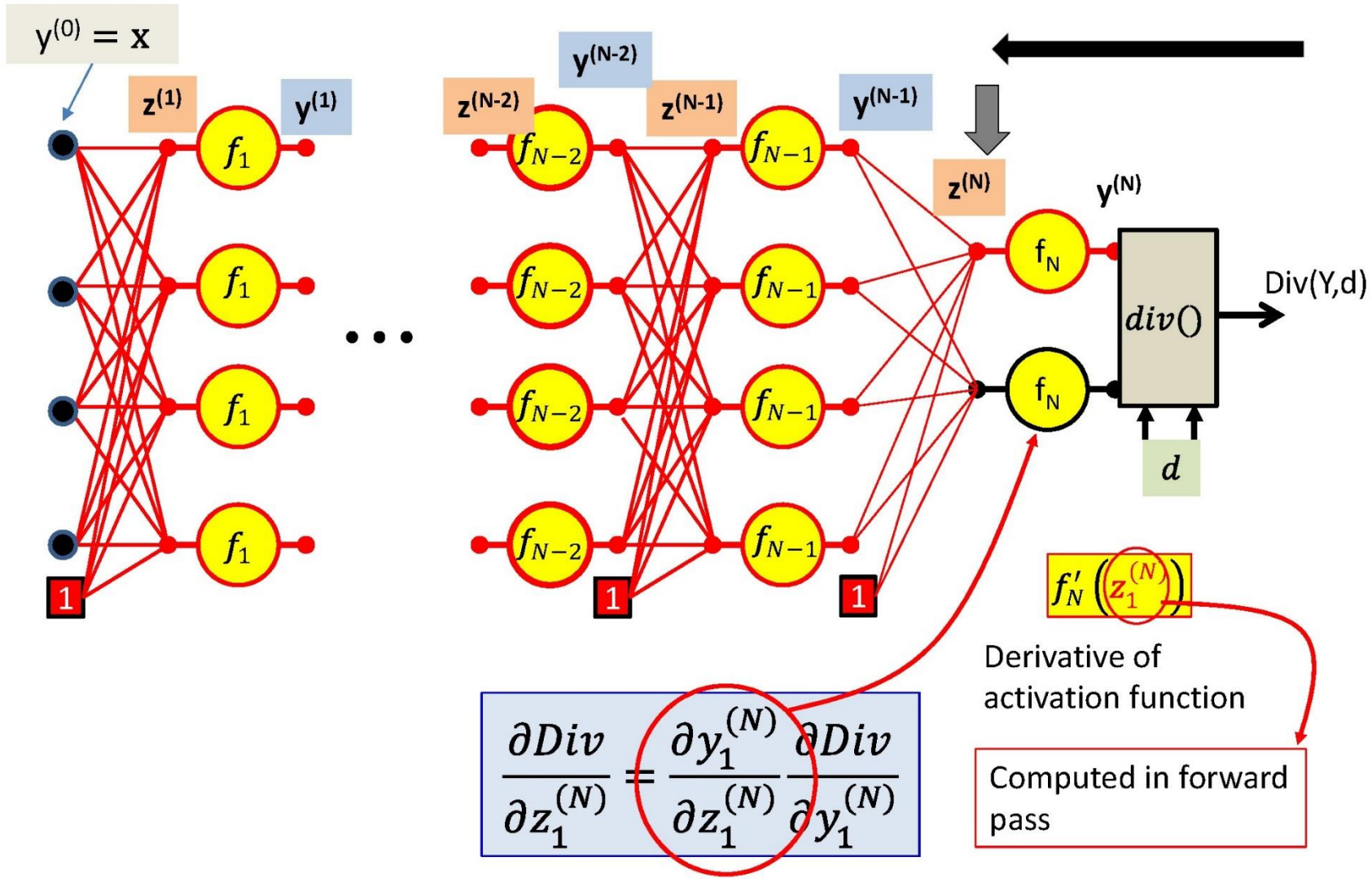
Backward Gradient Computation



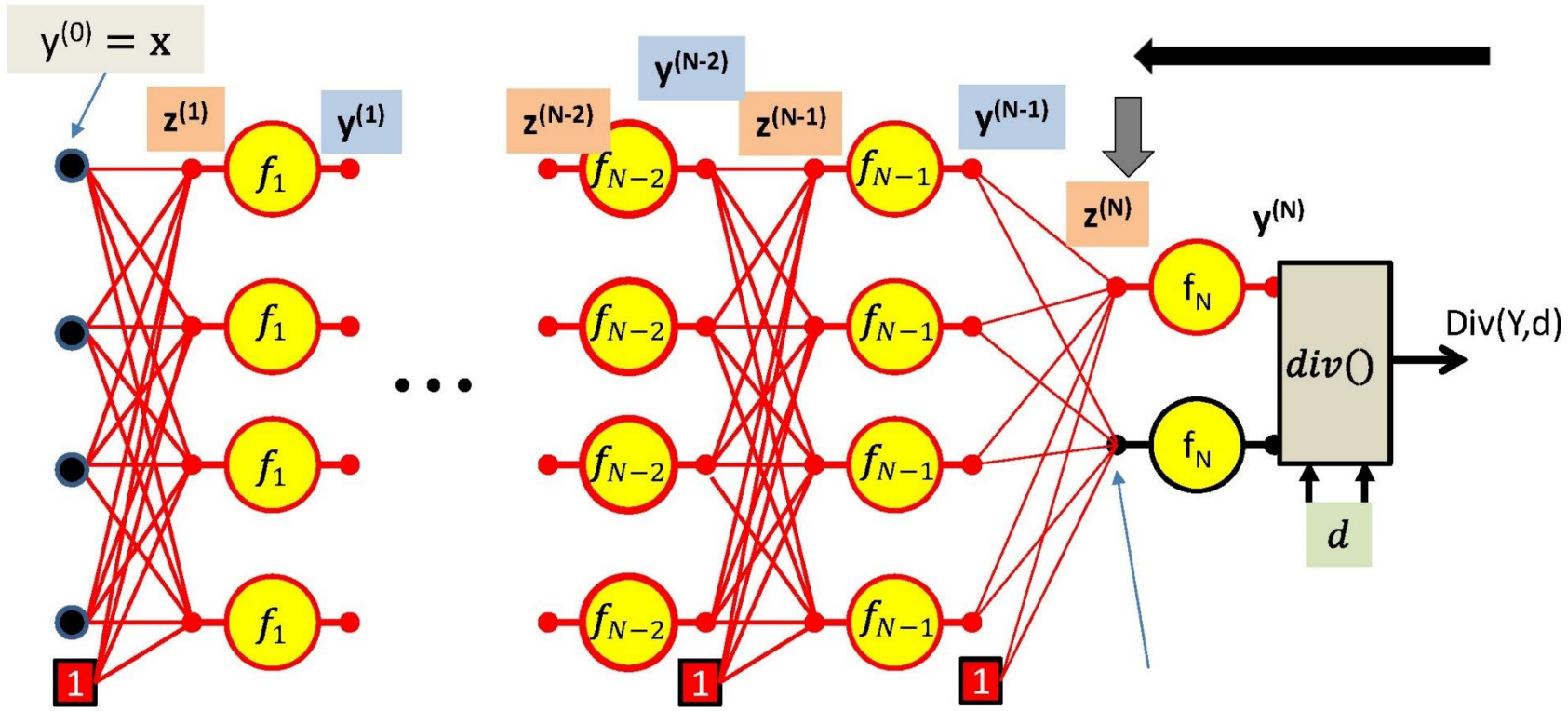
Backward Gradient Computation



Backward Gradient Computation

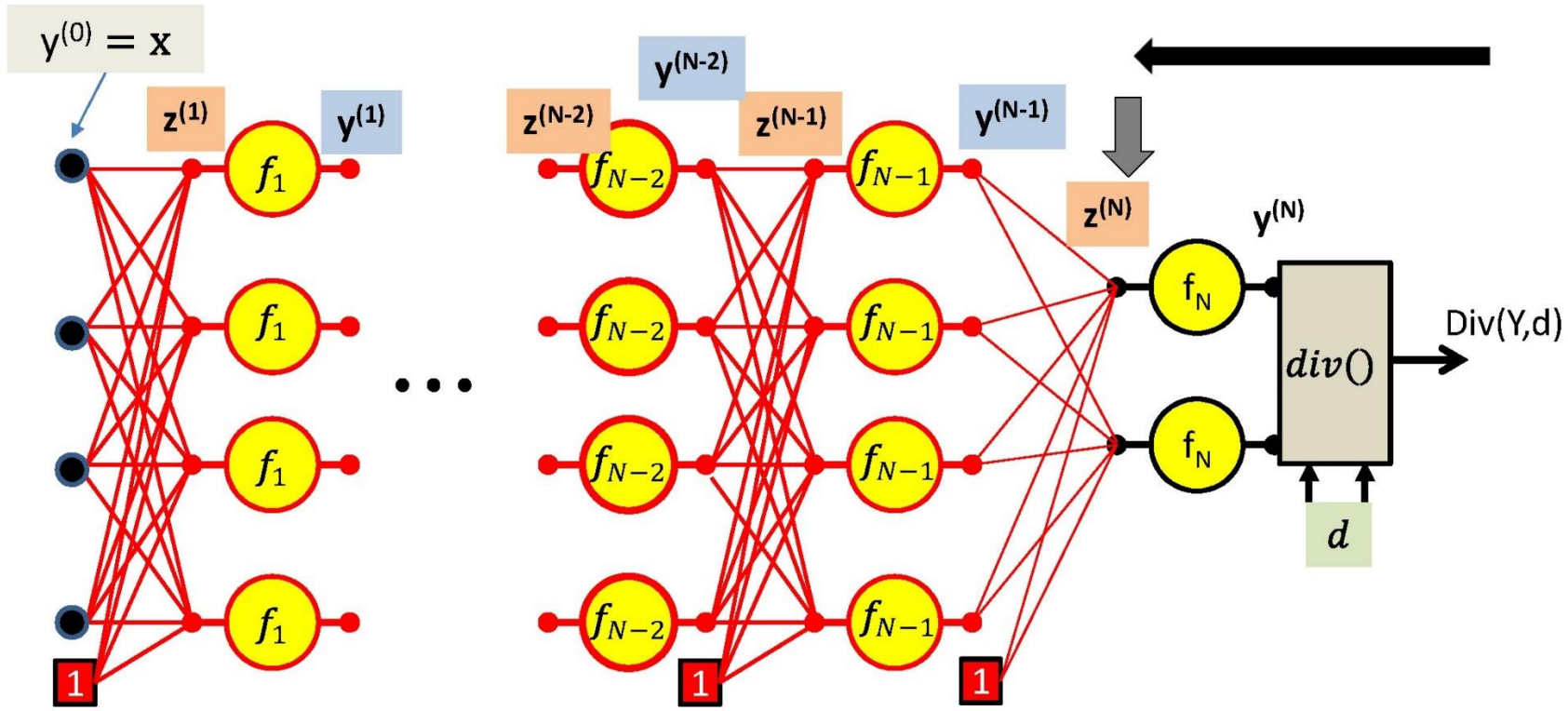


Backward Gradient Computation



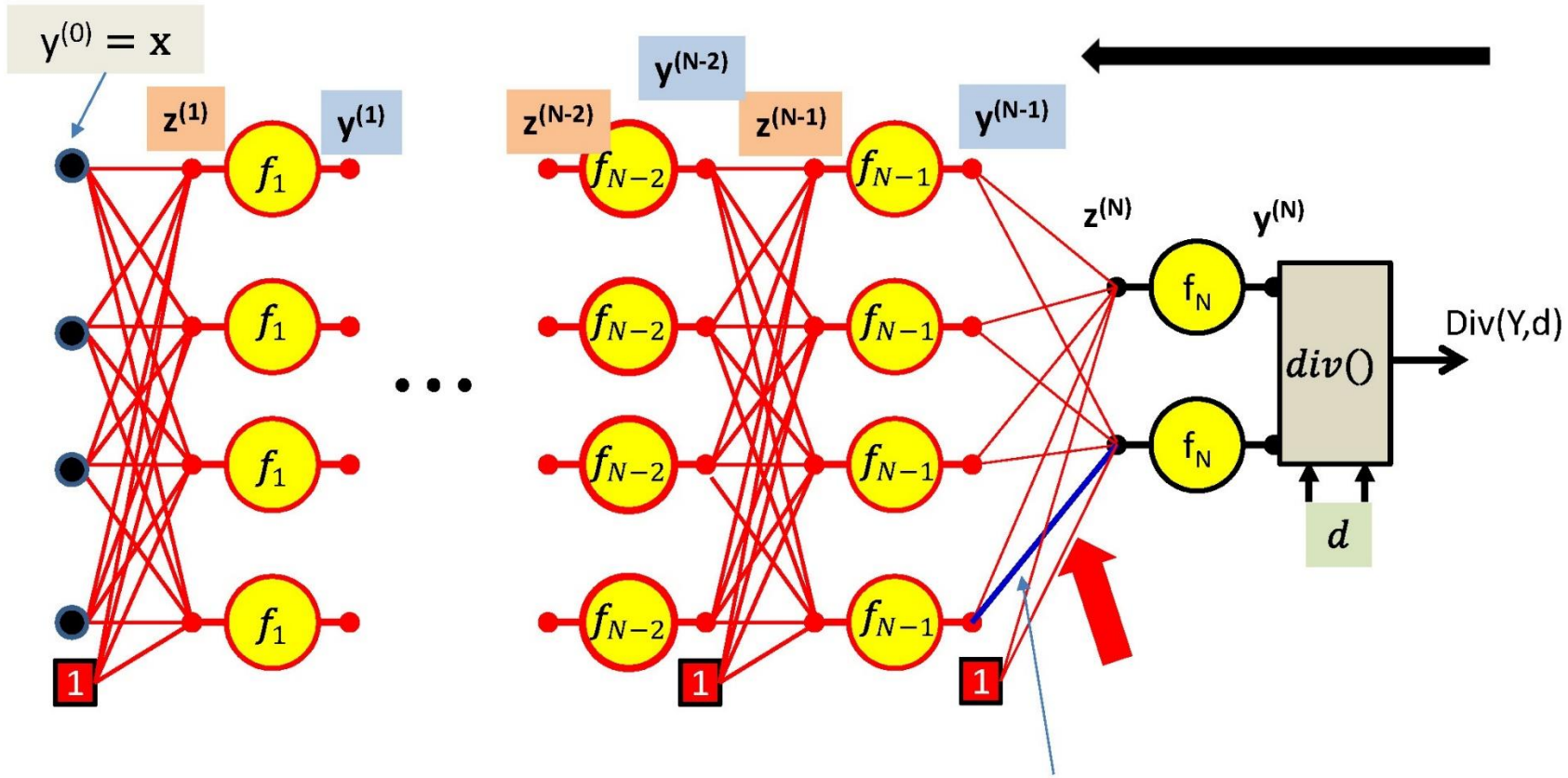
$$\frac{\partial Div}{\partial z_1^{(N)}} = f_N'\left(z_1^{(N)}\right)\frac{\partial Div}{\partial y_1^{(N)}}$$

Backward Gradient Computation



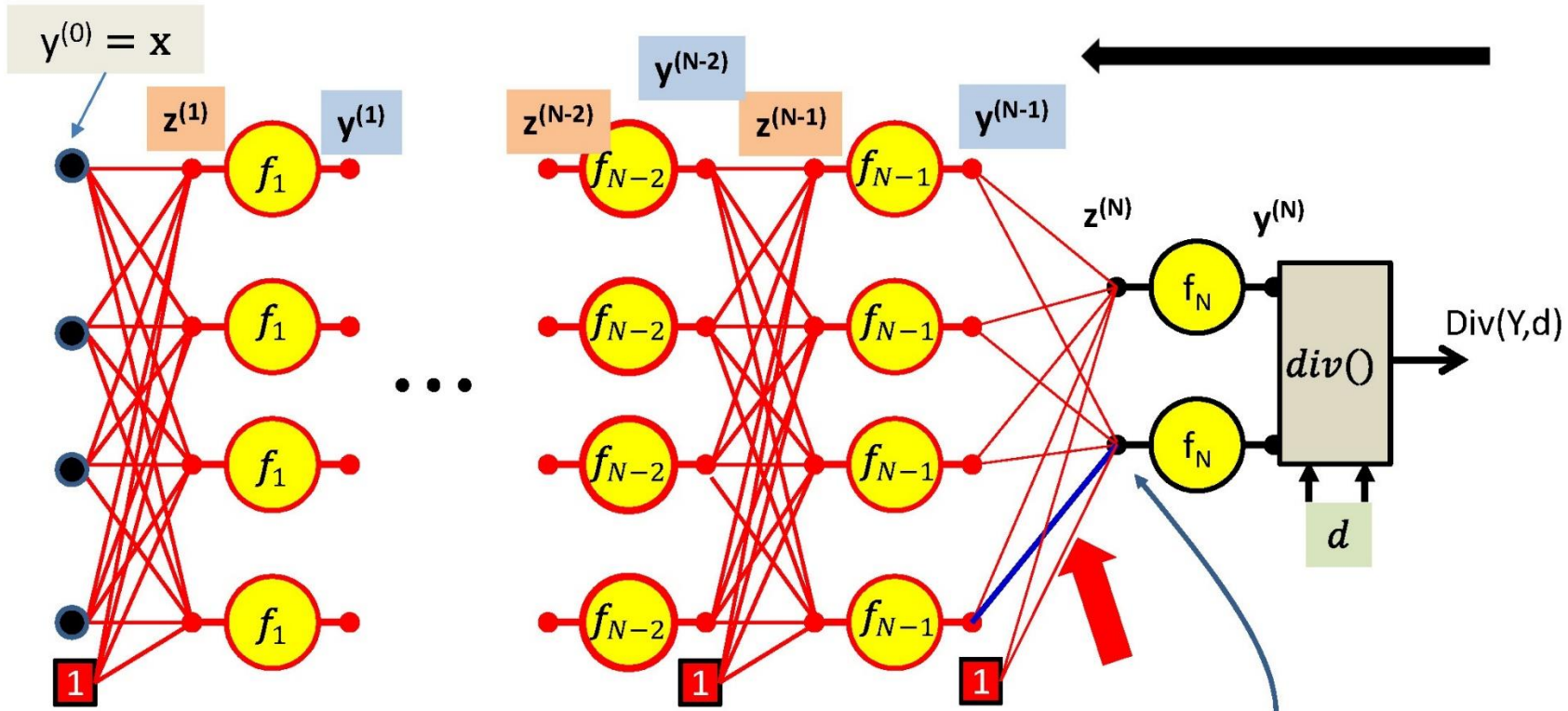
$$\frac{\partial Div}{\partial z_i^{(N)}} = f_N'\left(z_i^{(N)}\right)\frac{\partial Div}{\partial y_i^{(N)}}$$

Backward Gradient Computation



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

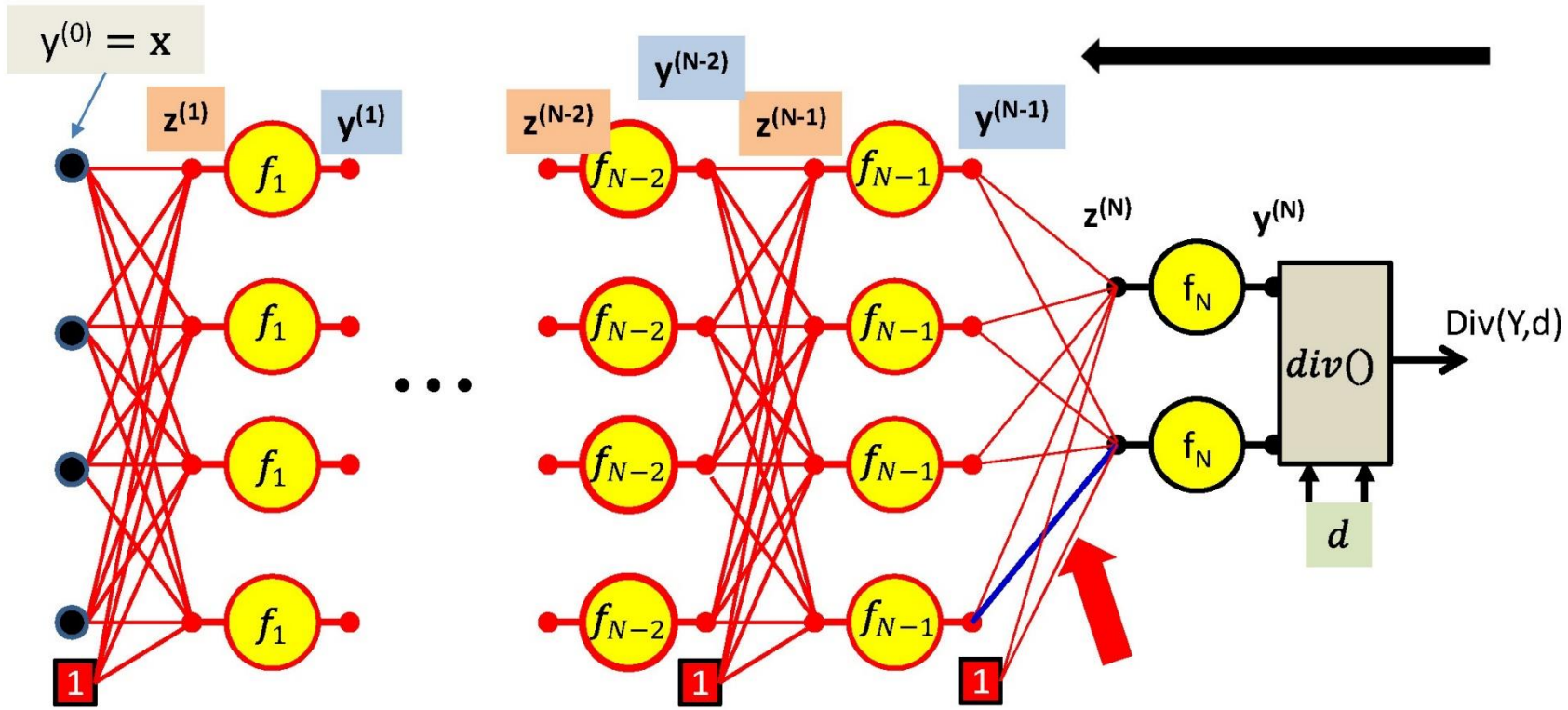
Backward Gradient Computation



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

Just computed

Backward Gradient Computation



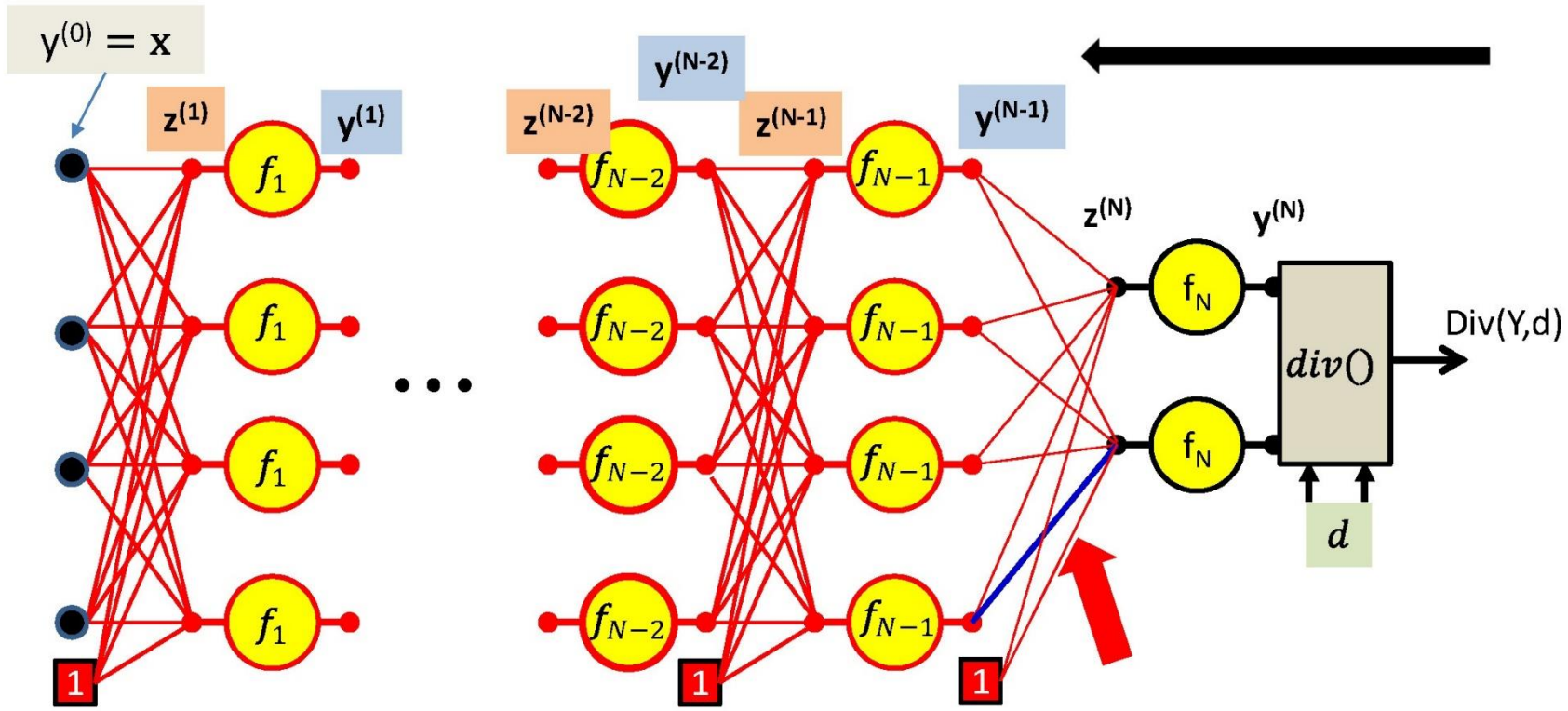
$$\frac{\partial Div}{\partial w_{11}^{(N)}} = \frac{\partial z_1^{(N)}}{\partial w_{11}^{(N)}} \frac{\partial Div}{\partial z_1^{(N)}}$$

$y_1^{(N-1)}$

Because

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

Backward Gradient Computation



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

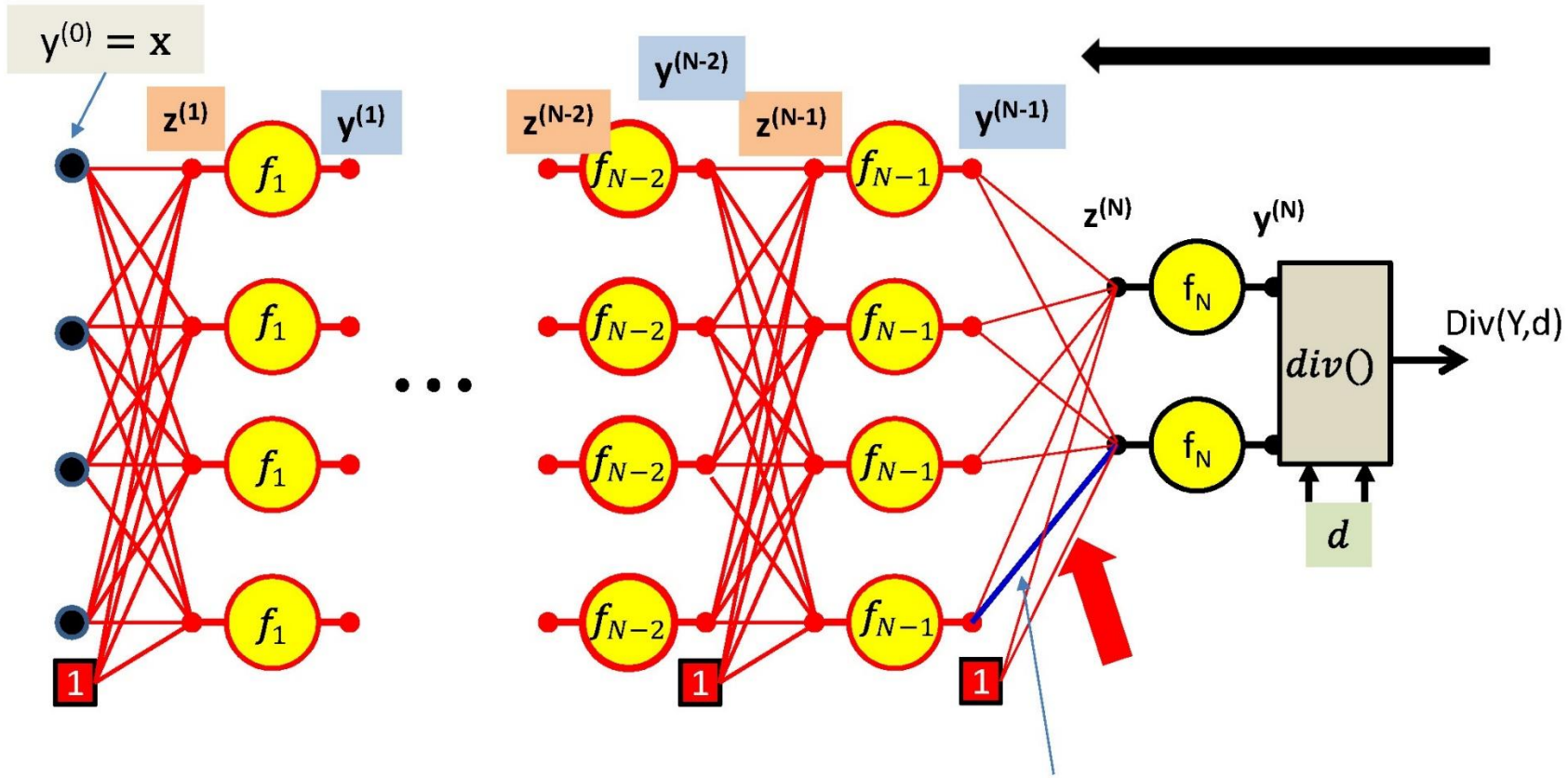
$y_1^{(N-1)}$

Because

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

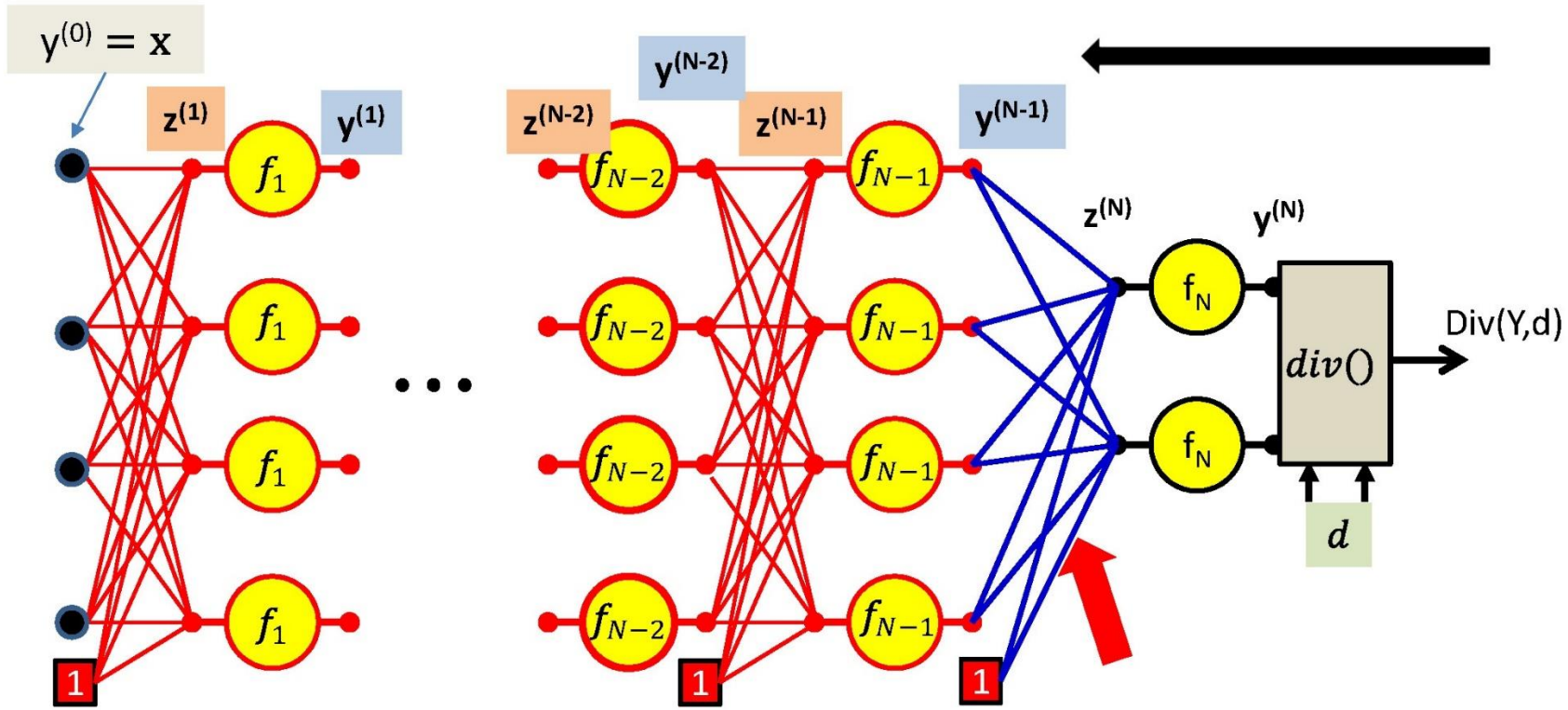
Computed in forward pass

Backward Gradient Computation



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

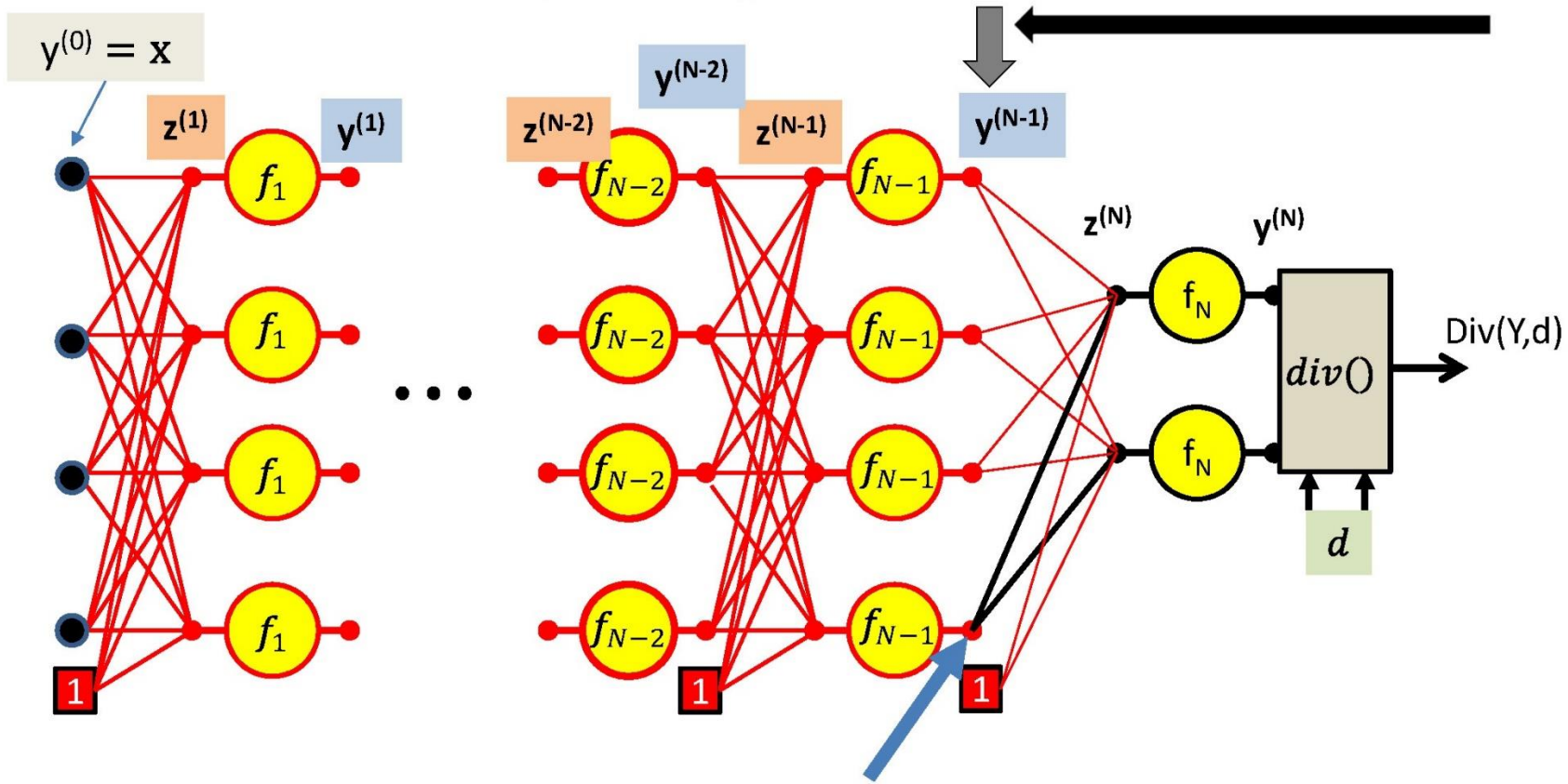
Backward Gradient Computation



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

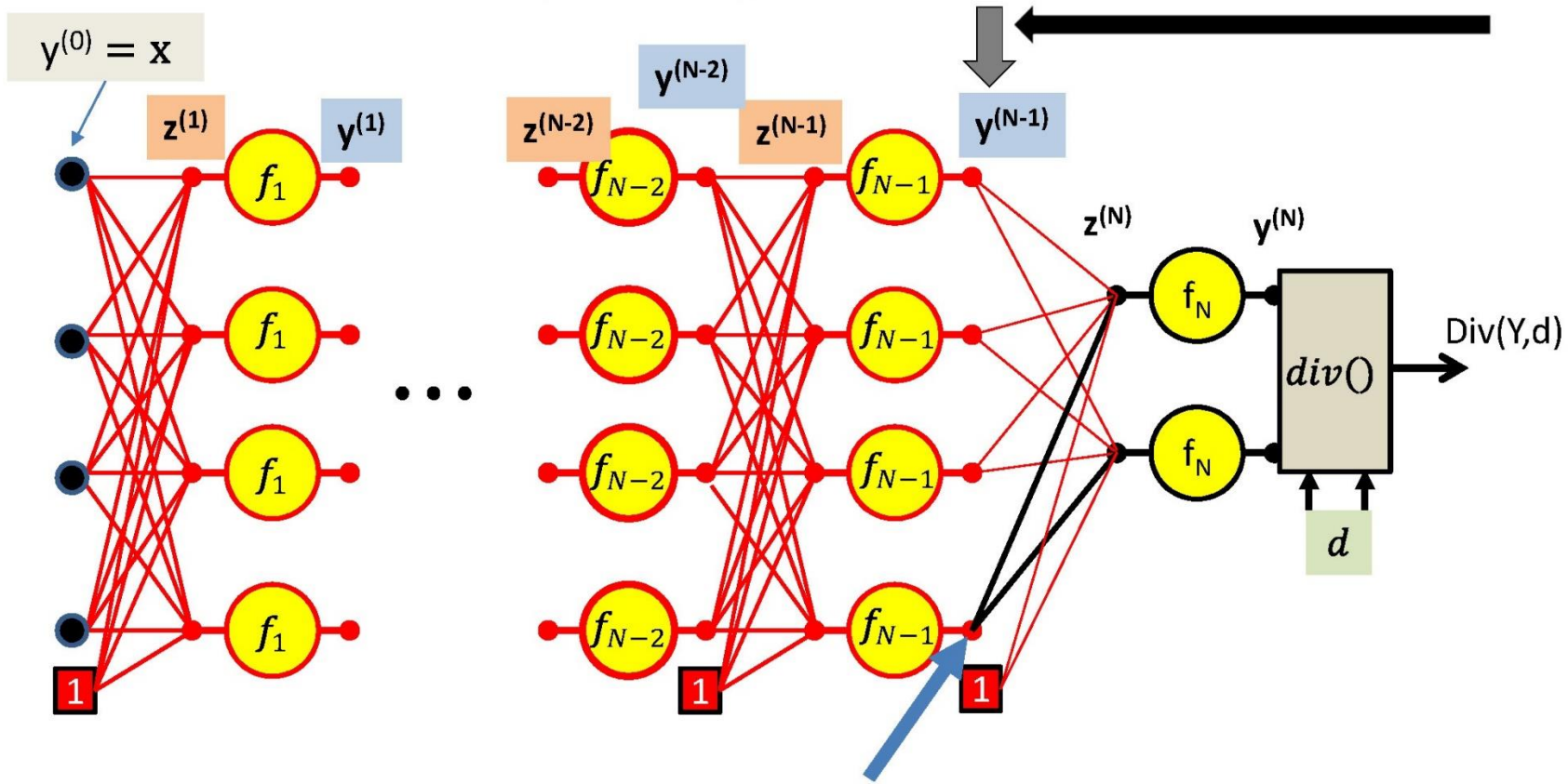
For the bias term $y_0^{(N-1)} = 1$

Backward Gradient Computation



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

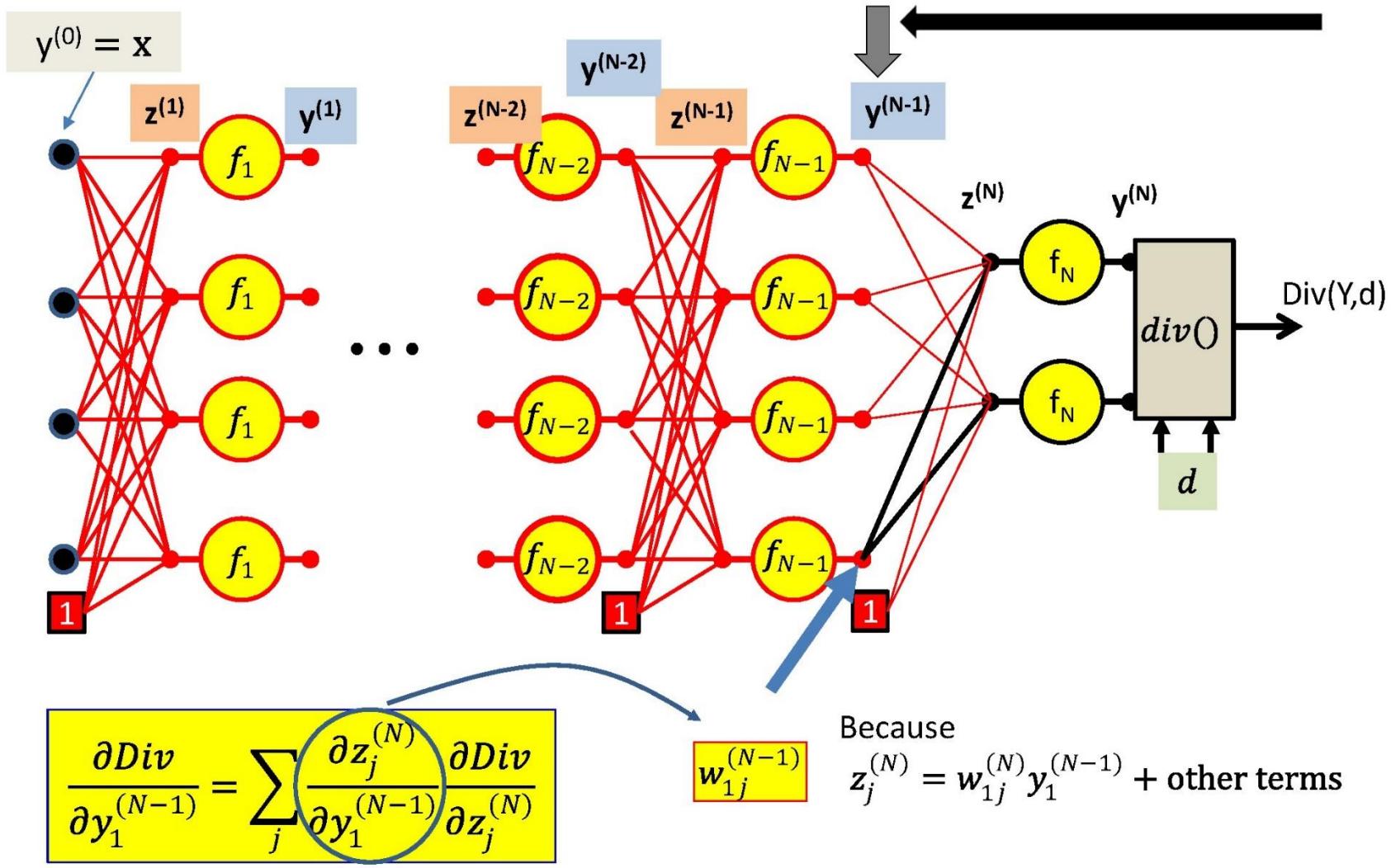
Backward Gradient Computation



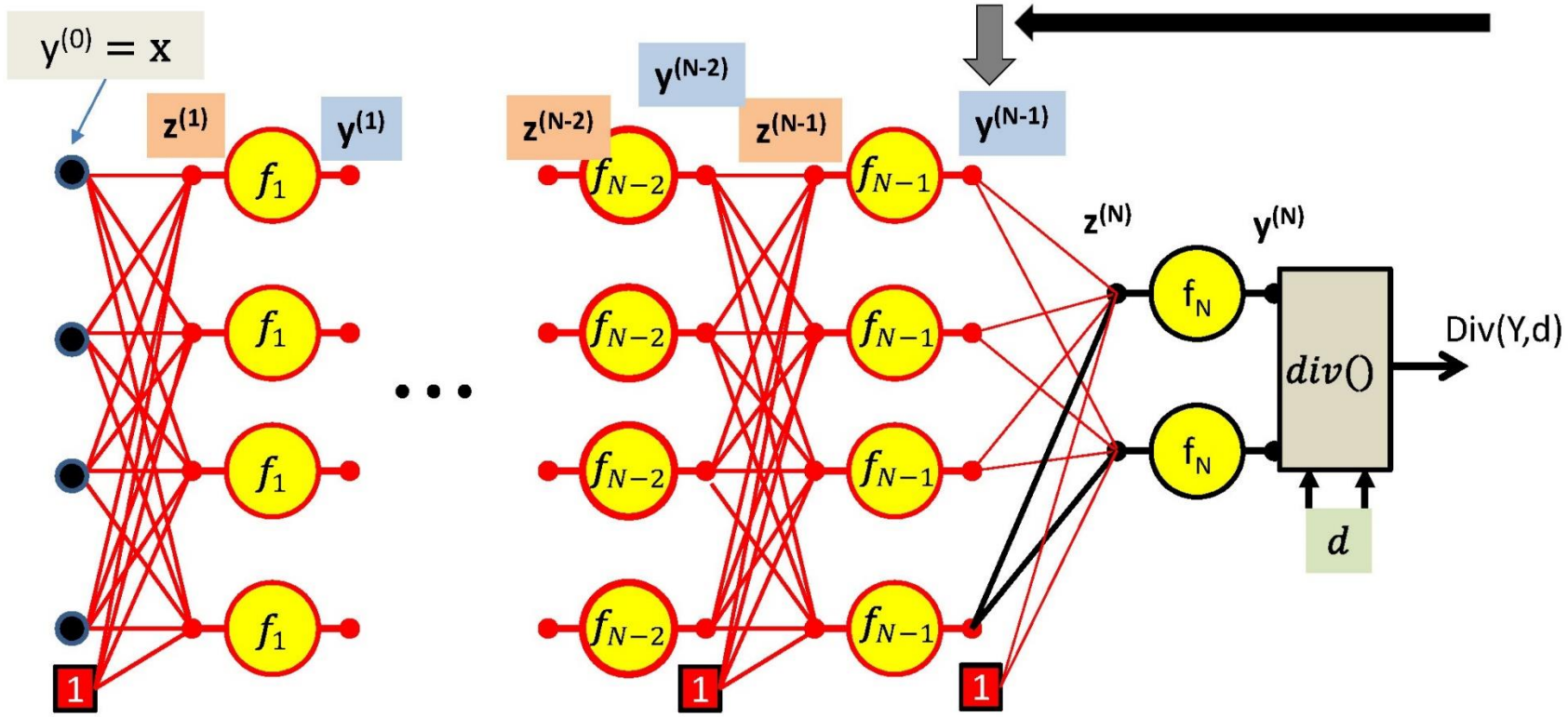
$$\frac{\partial Div}{\partial y_1^{(N-1)}} = \sum_j \frac{\partial z_j^{(N)}}{\partial y_1^{(N-1)}} \frac{\partial Div}{\partial z_j^{(N)}}$$

Already computed

Backward Gradient Computation

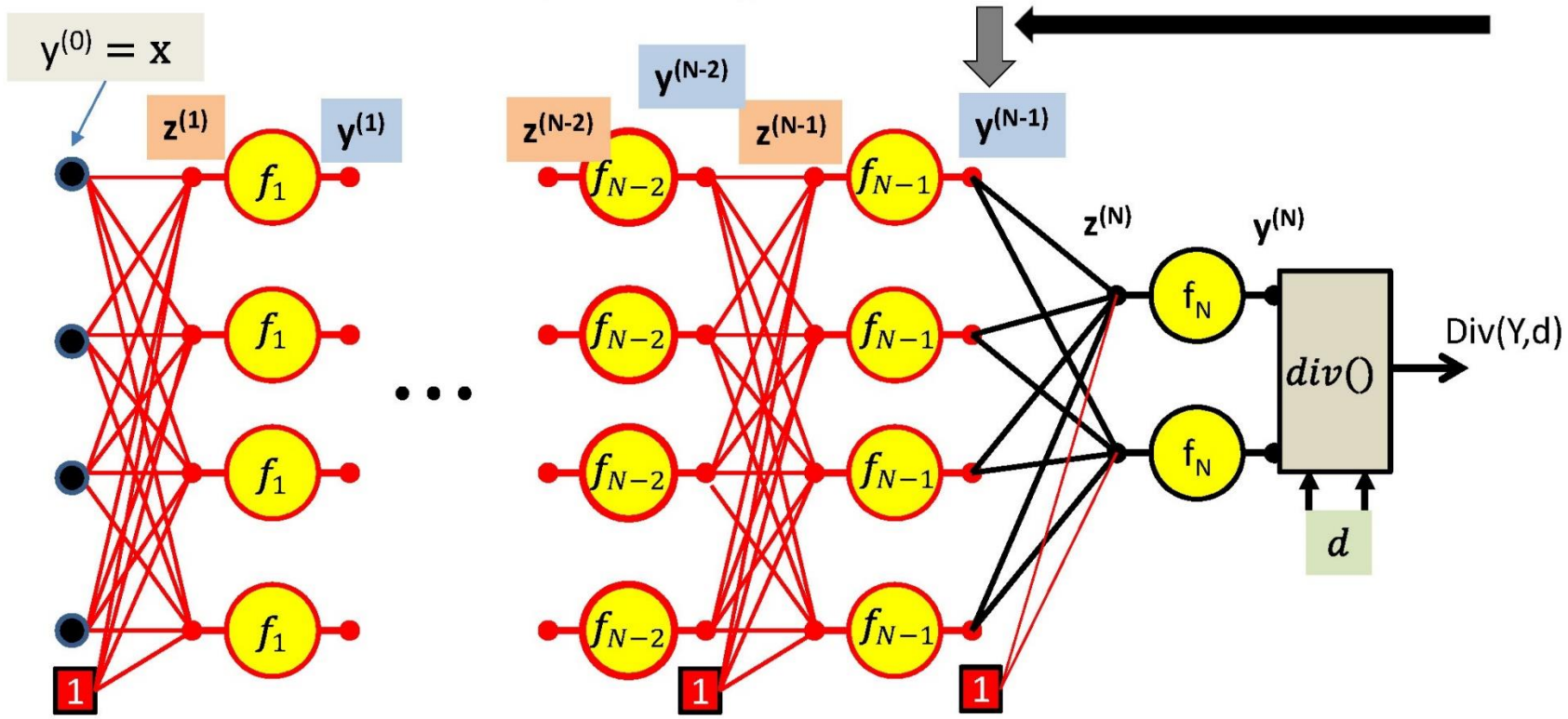


Backward Gradient Computation



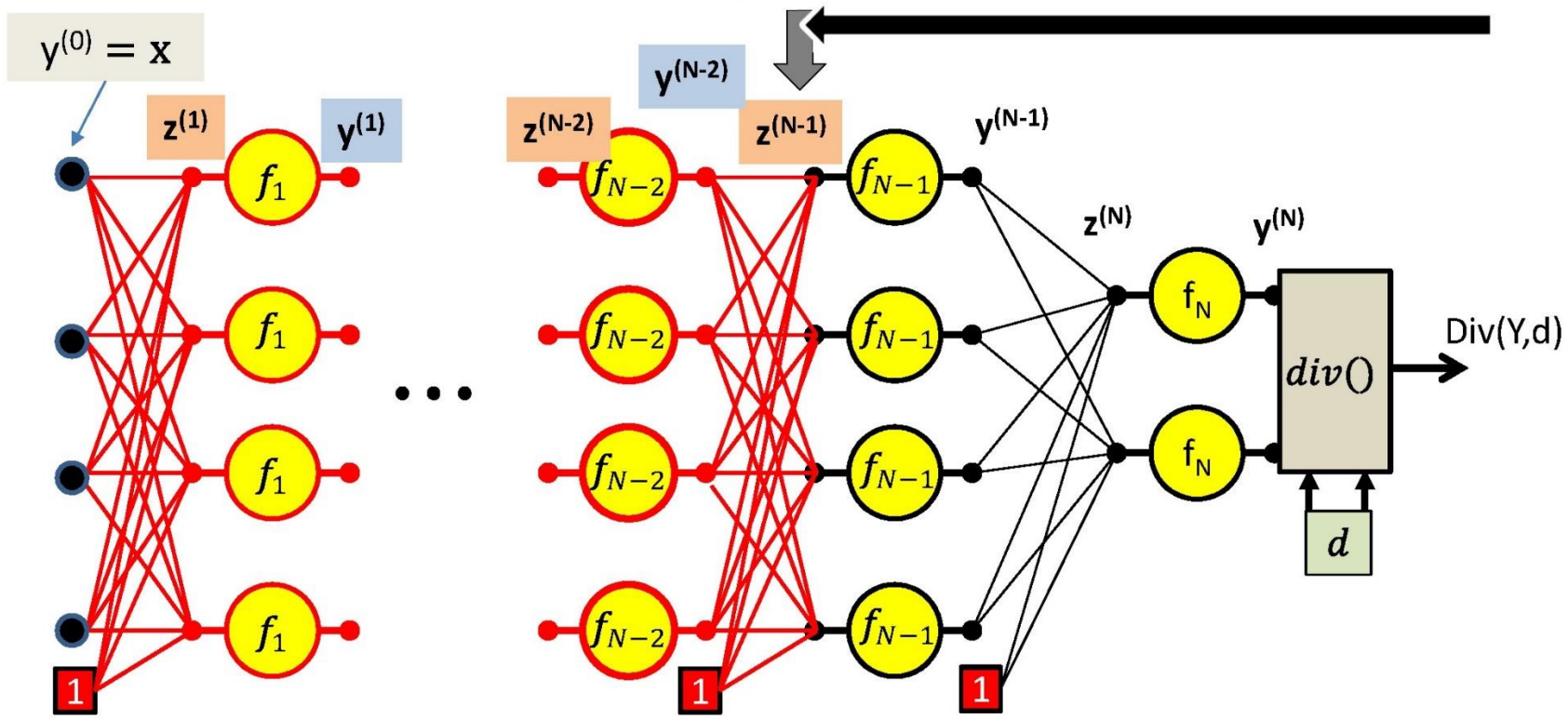
$$\frac{\partial Div}{\partial y_1^{(N-1)}} = \sum_j w_{1j}^{(N)} \frac{\partial Div}{\partial z_j^{(N)}}$$

Backward Gradient Computation



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

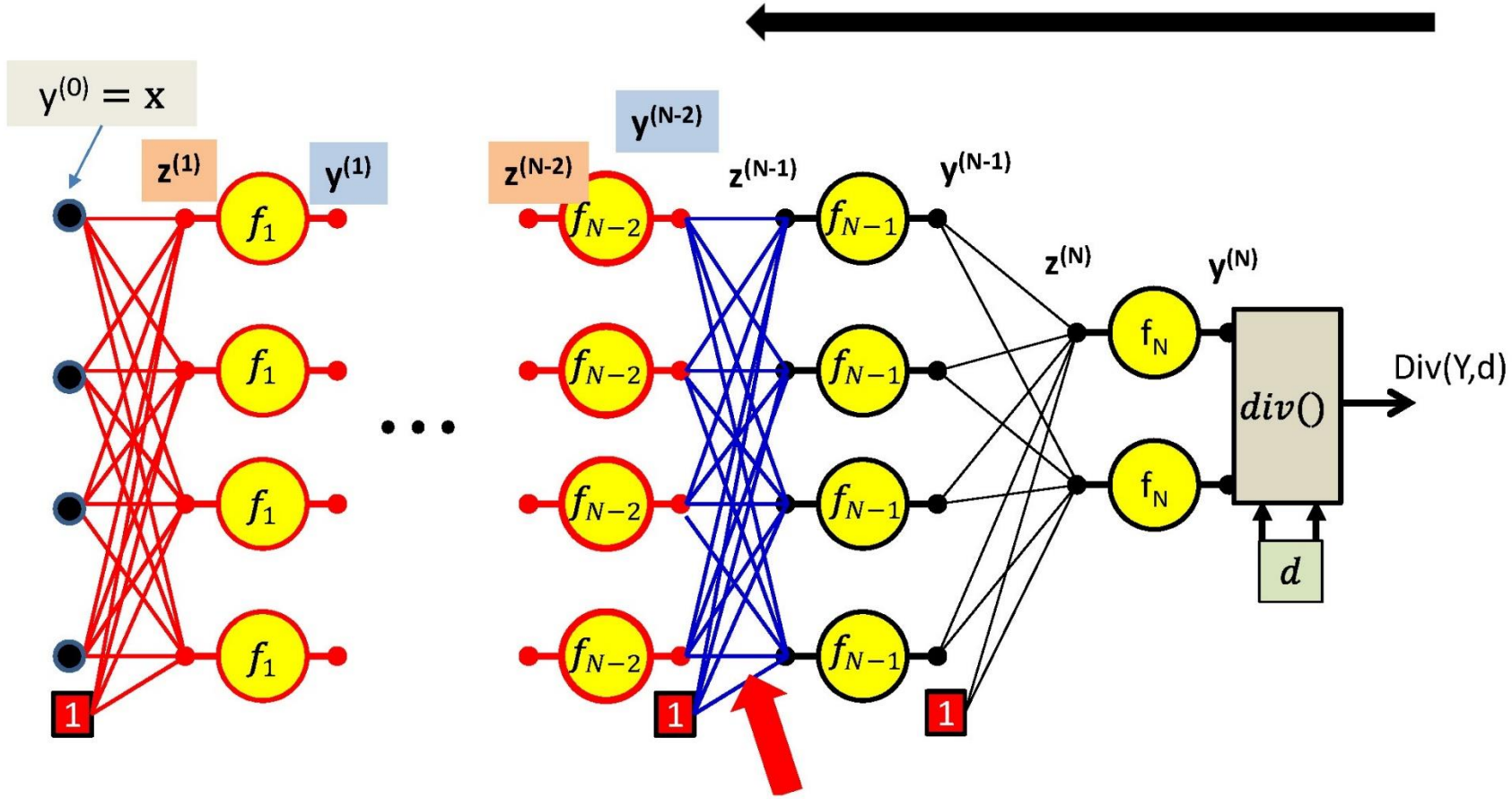
Backward Gradient Computation



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

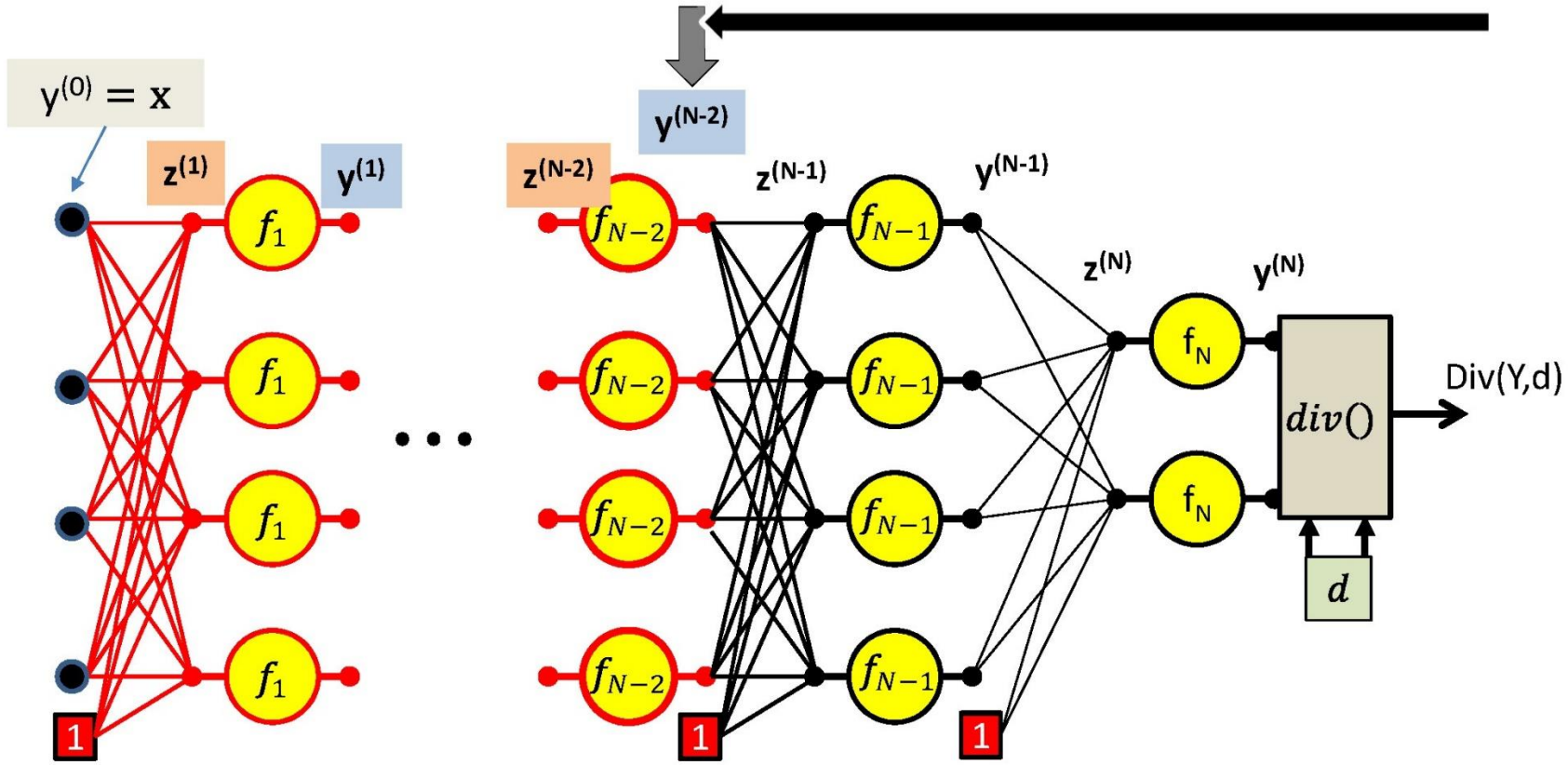


$$\frac{\partial Div}{\partial w_{ij}^{(N-1)}} = y_i^{(N-2)} \frac{\partial Div}{\partial z_j^{(N-1)}}$$

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

We continue our way backwards in the order shown

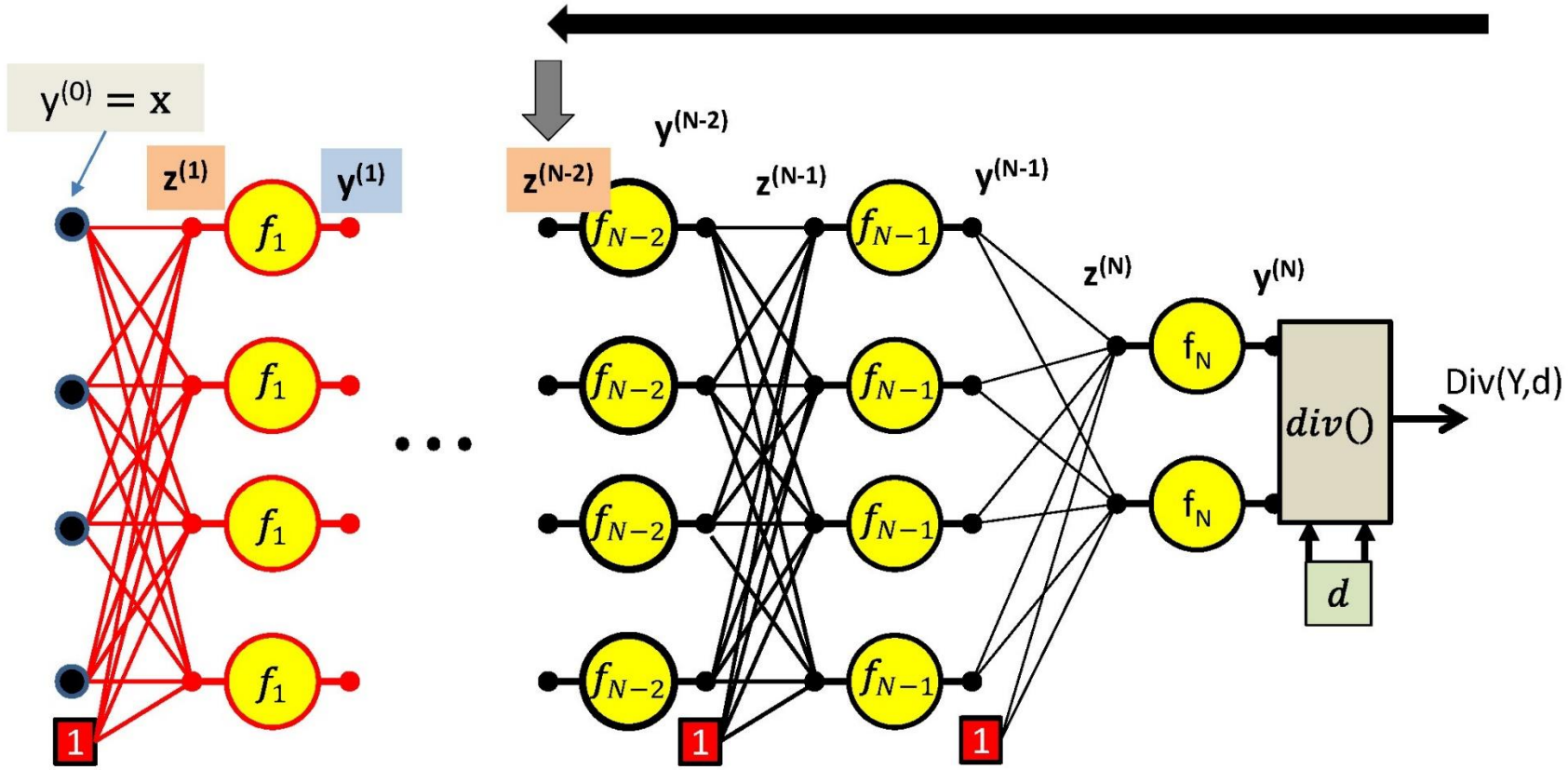
Backward Gradient Computation



We continue our way backwards in the order shown

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

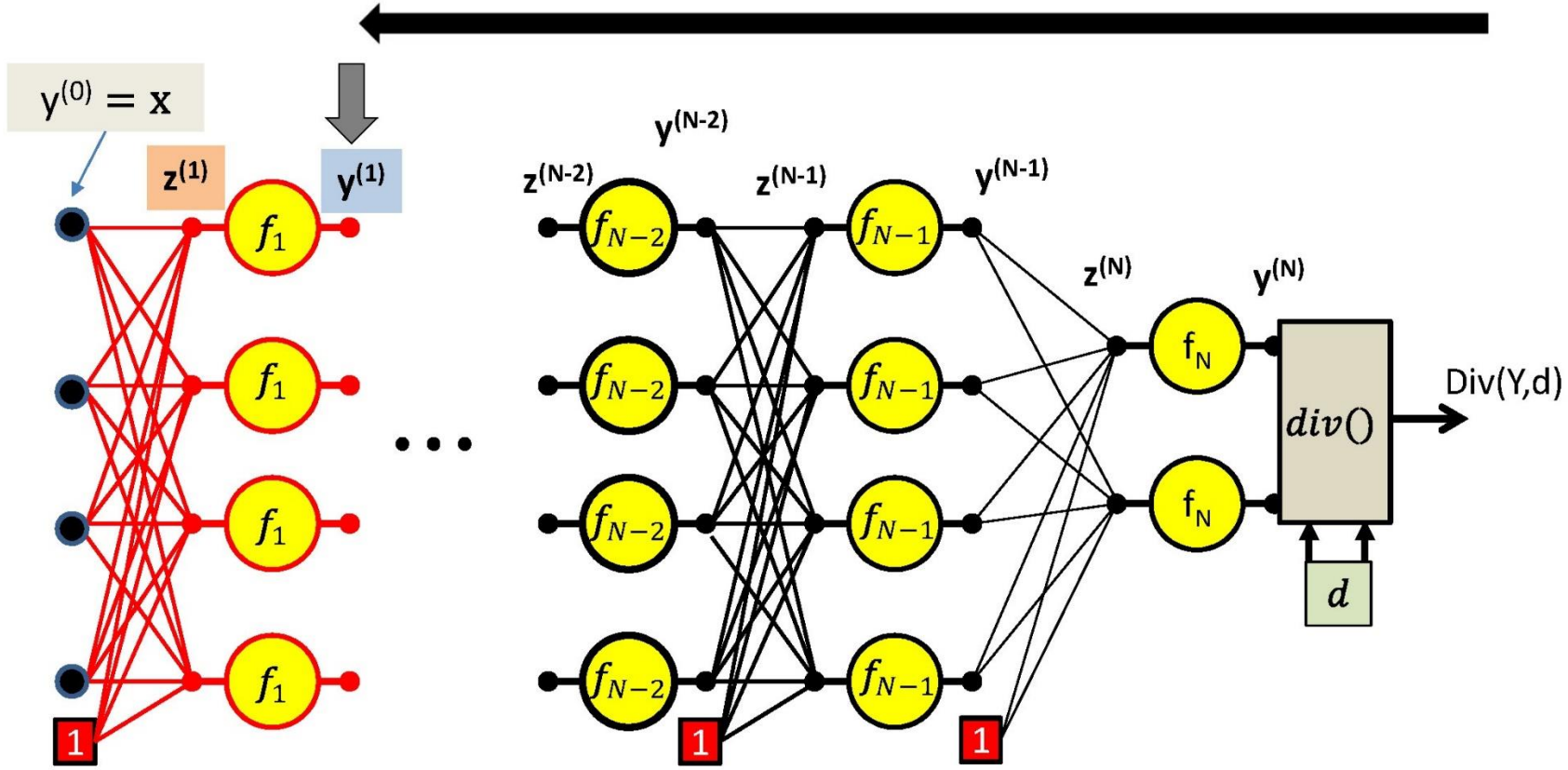
Backward Gradient Computation



We continue our way backwards in the order shown

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

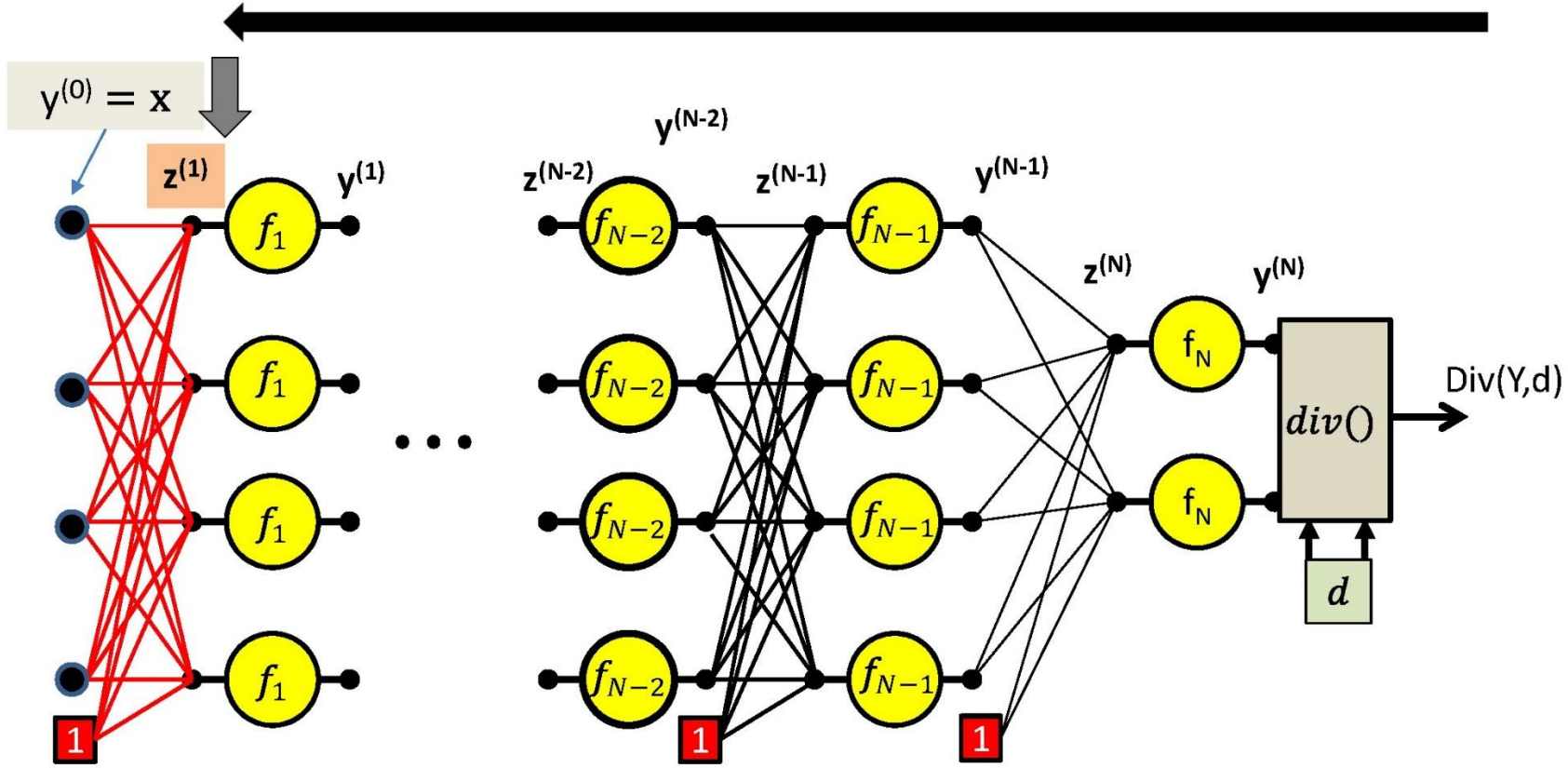
Backward Gradient Computation



We continue our way backwards in the order shown

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

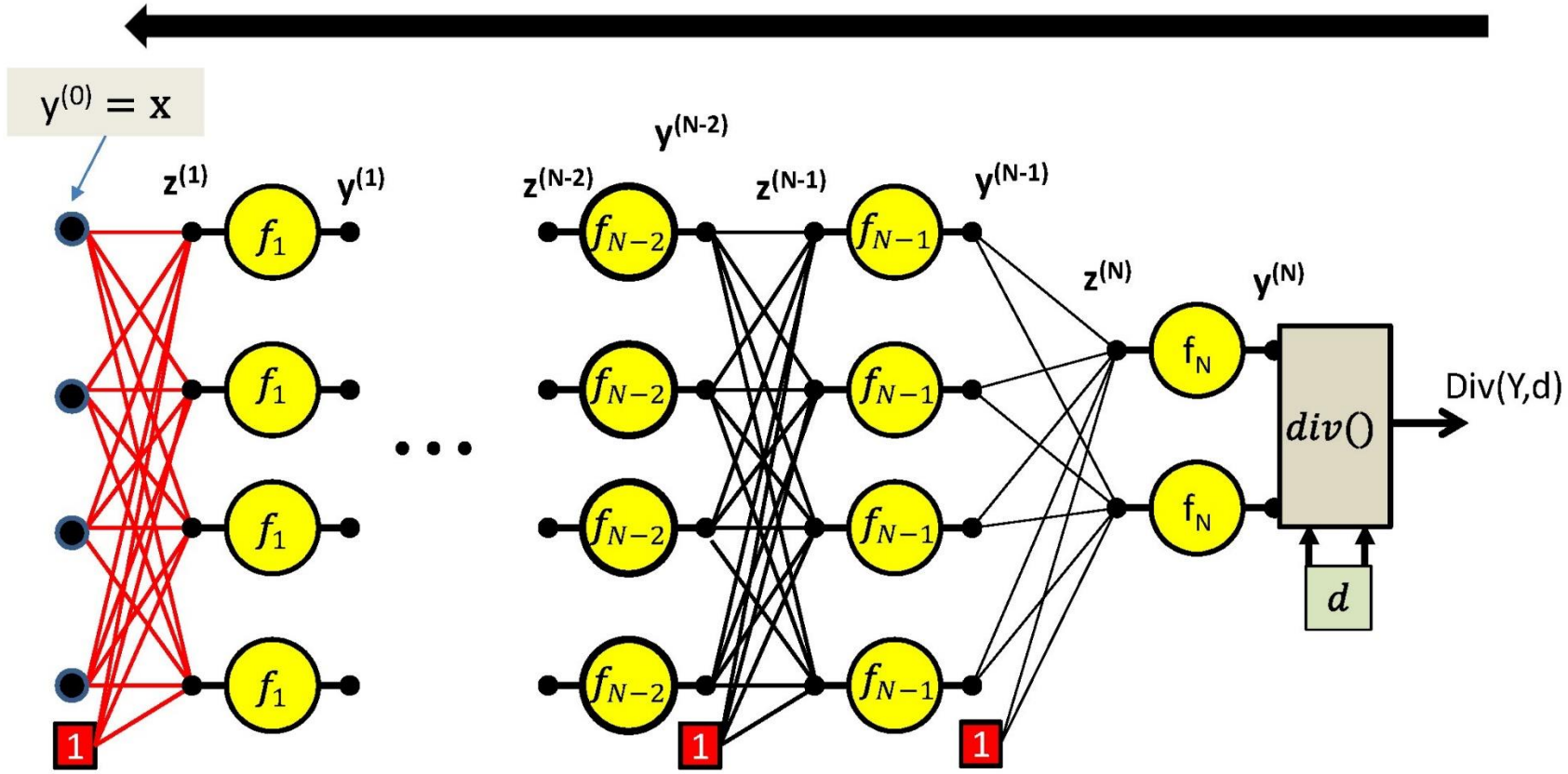
Backward Gradient Computation



We continue our way backwards in the order shown

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

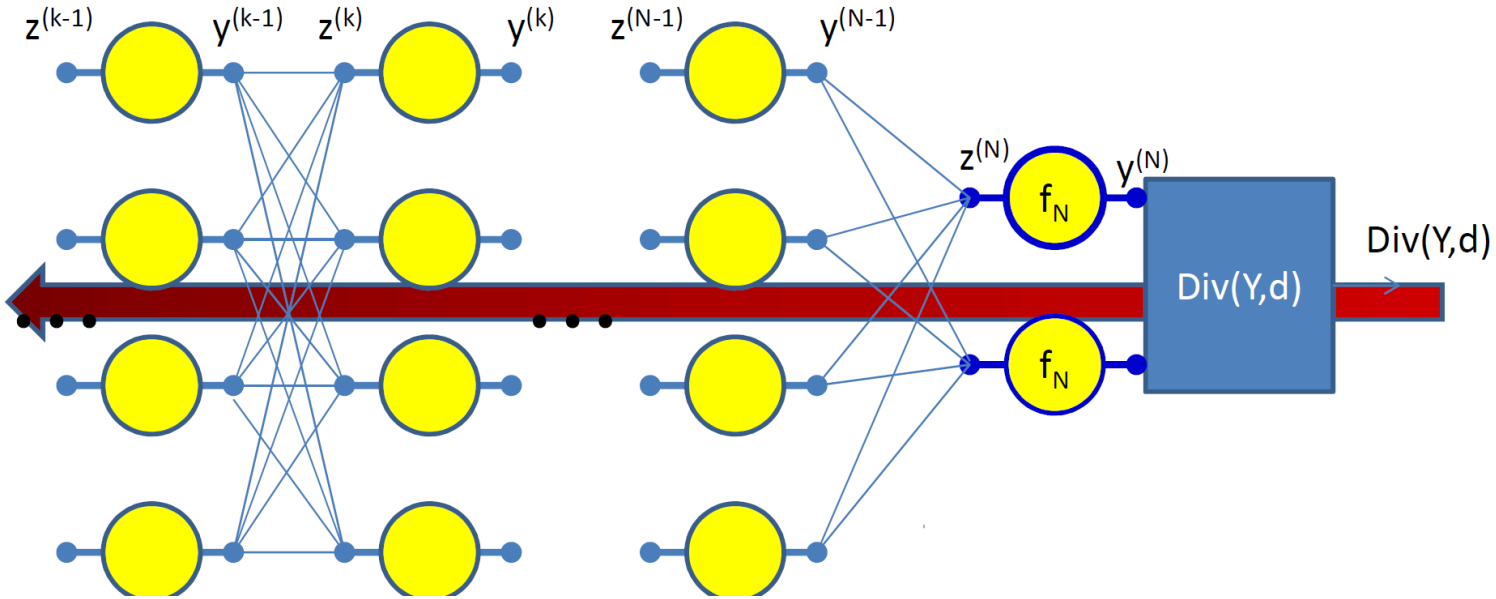
Backward Gradient Computation



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

For k= N-1 … 0

For i= 1 … Layer width

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

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

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

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

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

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 $i = 1 \ldots D_k$

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

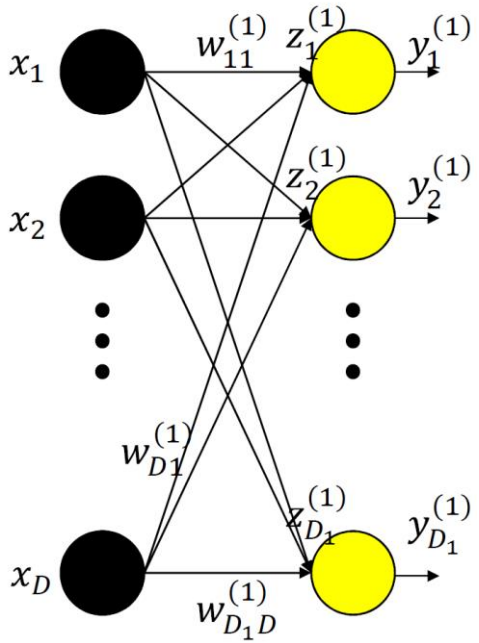
Backward weighted combination of next layer

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

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



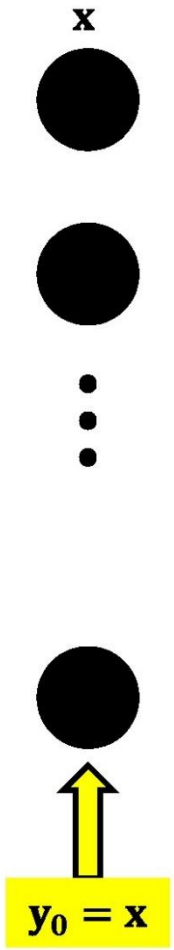
$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix} \quad \mathbf{z_k} = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{D_k}^{(k)} \end{bmatrix} \quad \mathbf{y_k} = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_{D_k}^{(k)} \end{bmatrix}$$

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

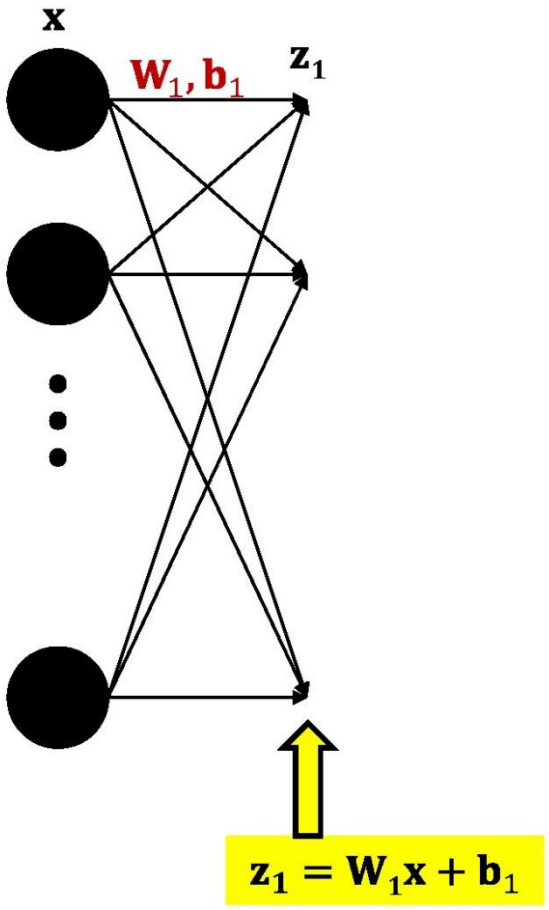
- Arrange all inputs to the network in a vector X
- Arrange the inputs to neurons of the k[th] layer as a vector Z_k
- Arrange the outputs of neurons in the k[th] layer as a vector y_k
- Arrange the weights to any layer as a matrix W_k (Similarly with biases)
- The computation of a single layer is easily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

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

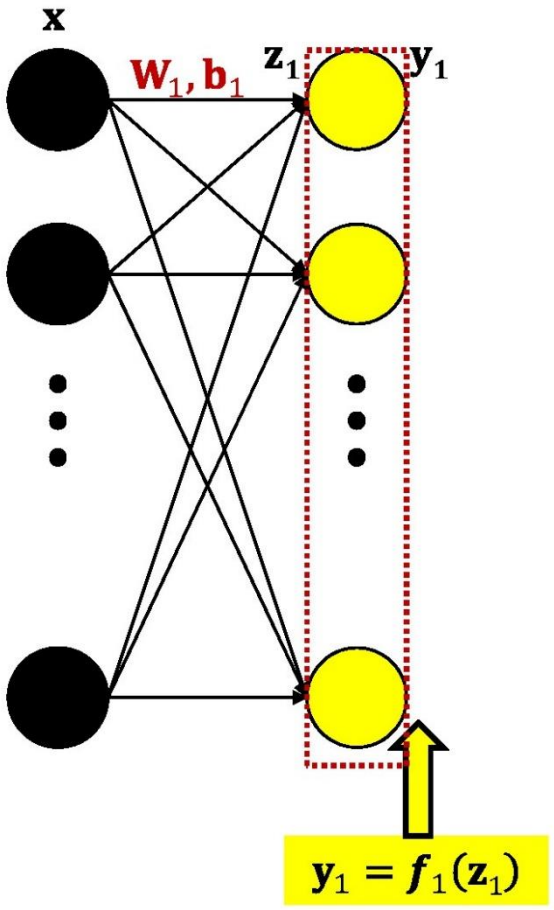
The forward pass: Evaluating the network



The forward pass



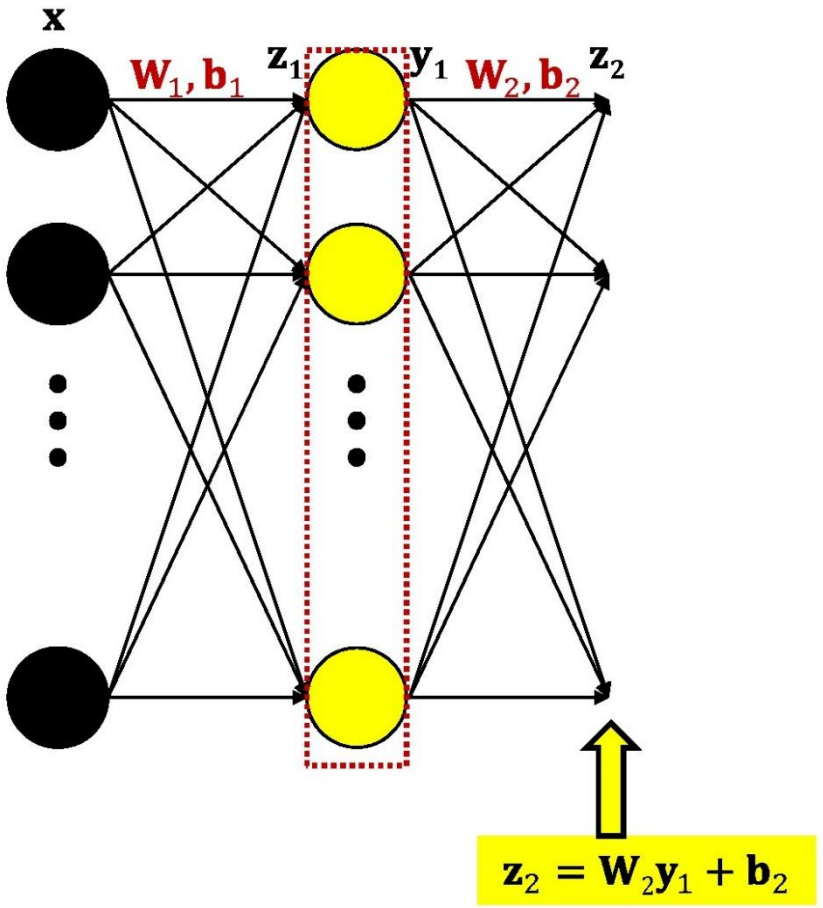
The forward pass



The Complete computation

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

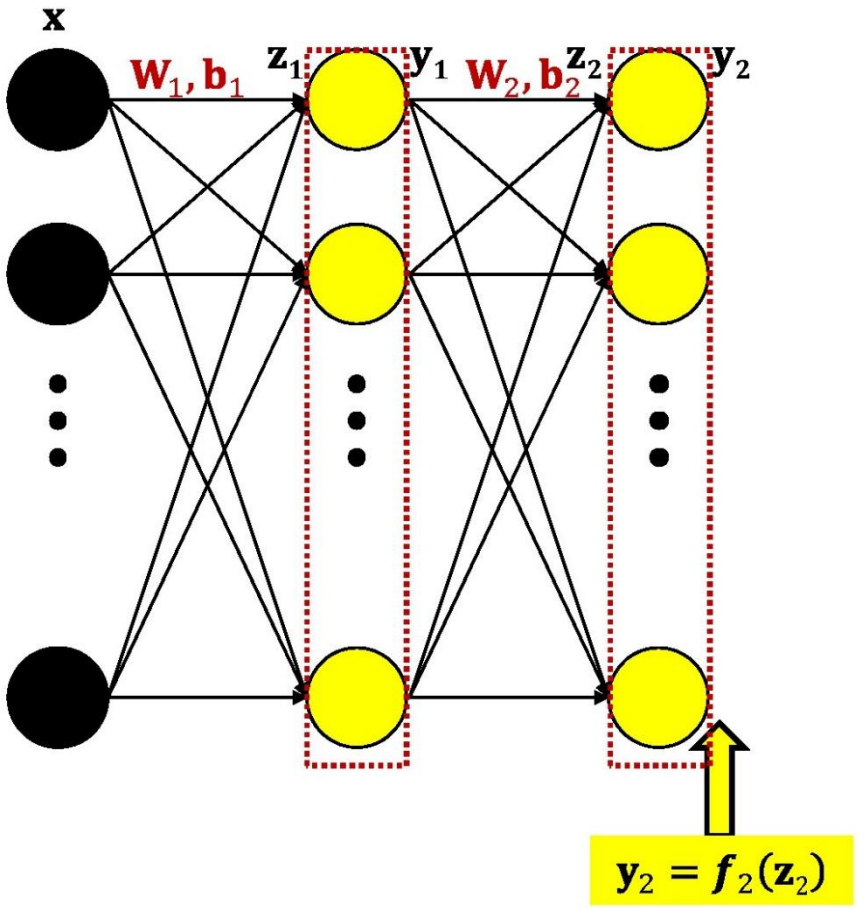
The forward pass



The Complete computation

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

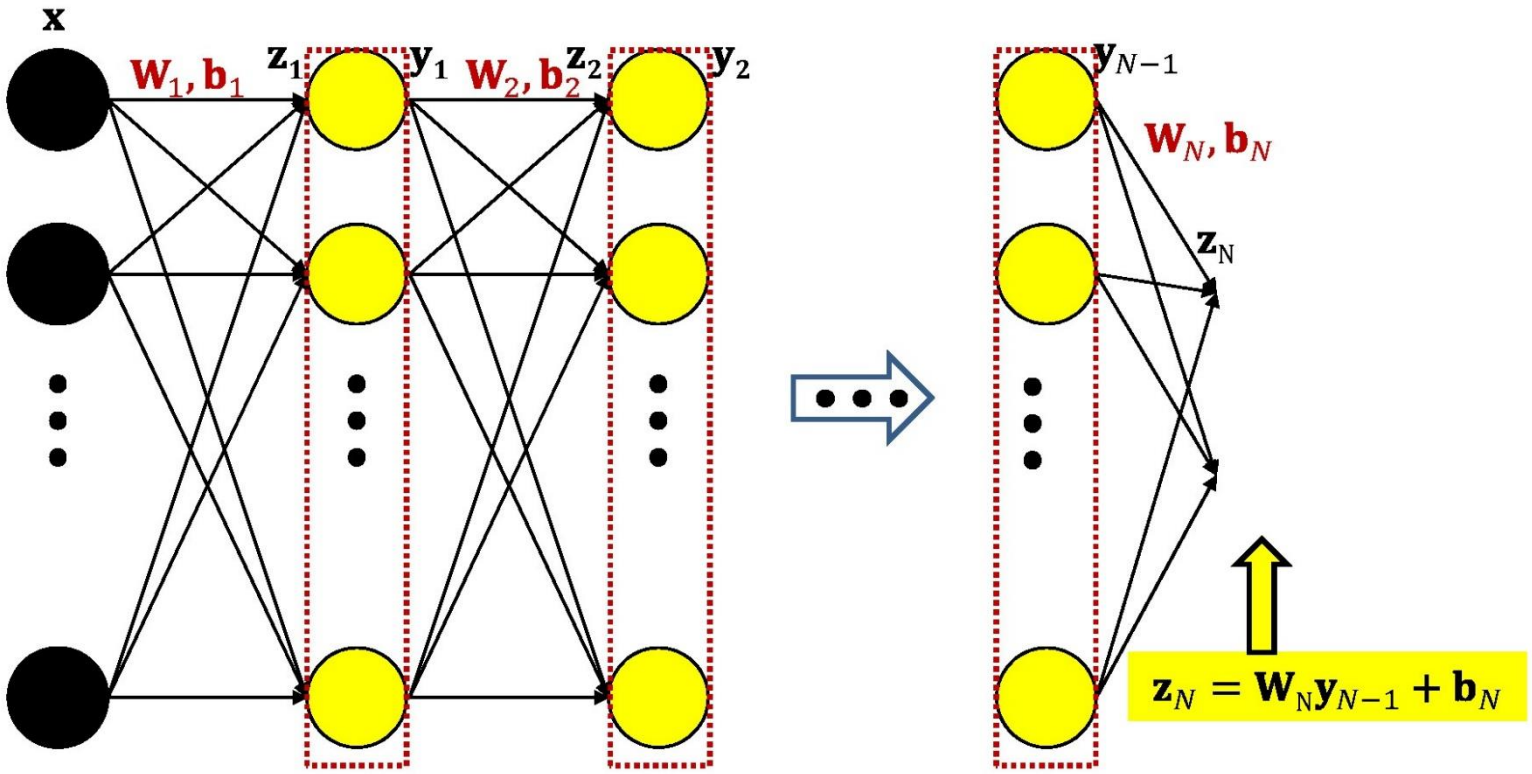
The forward pass



The Complete computation

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

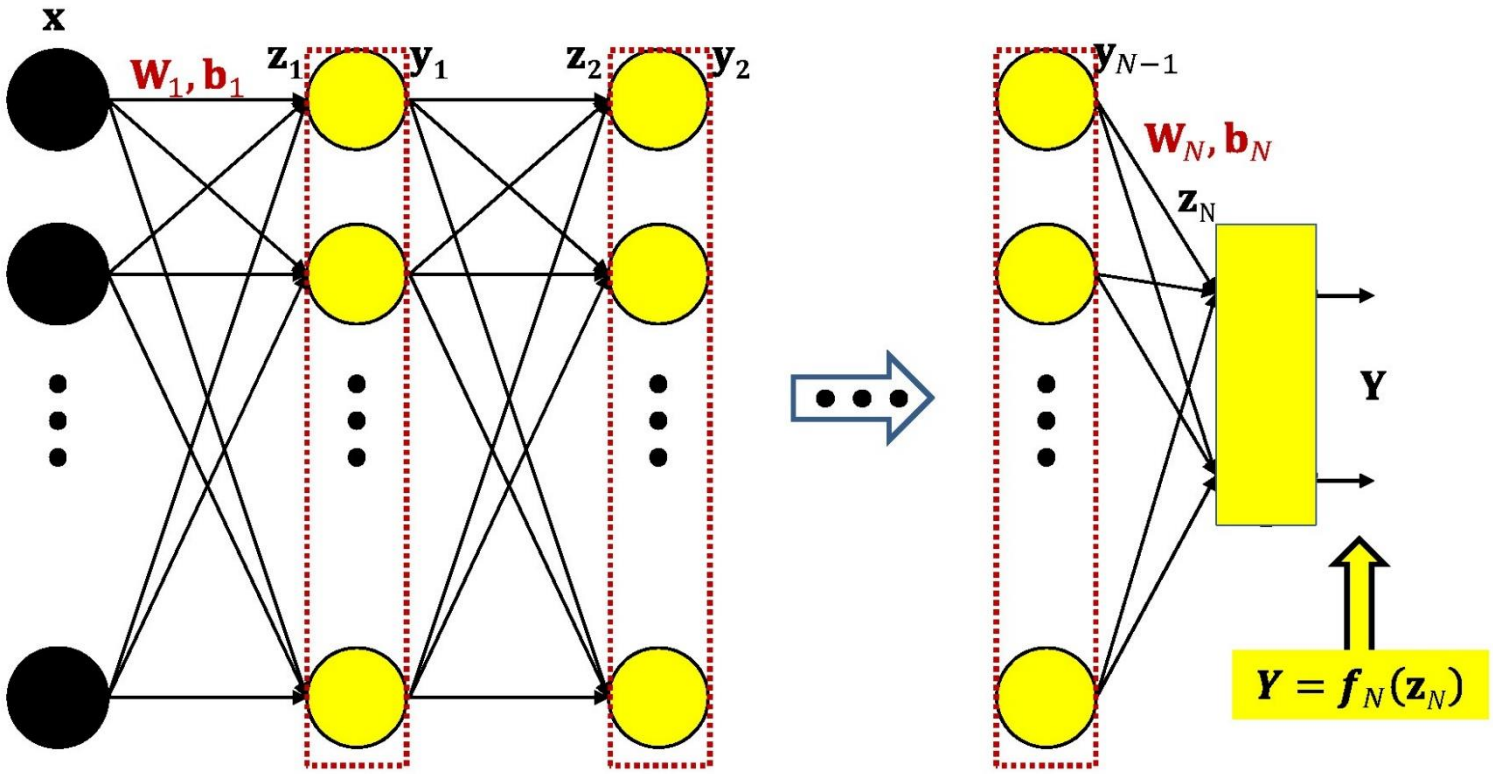
The forward pass



The Complete computation

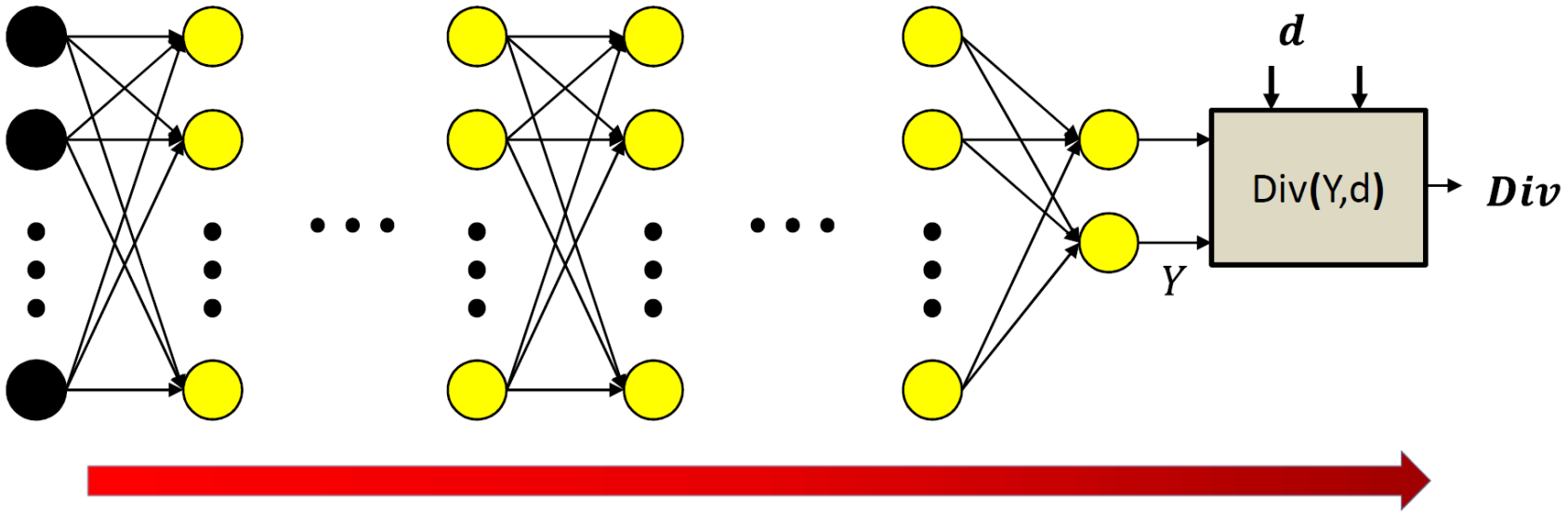
$$\mathbf{y}_2 = f_2(\mathbf{W}_2 f_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)$$

The forward pass



The Complete computation

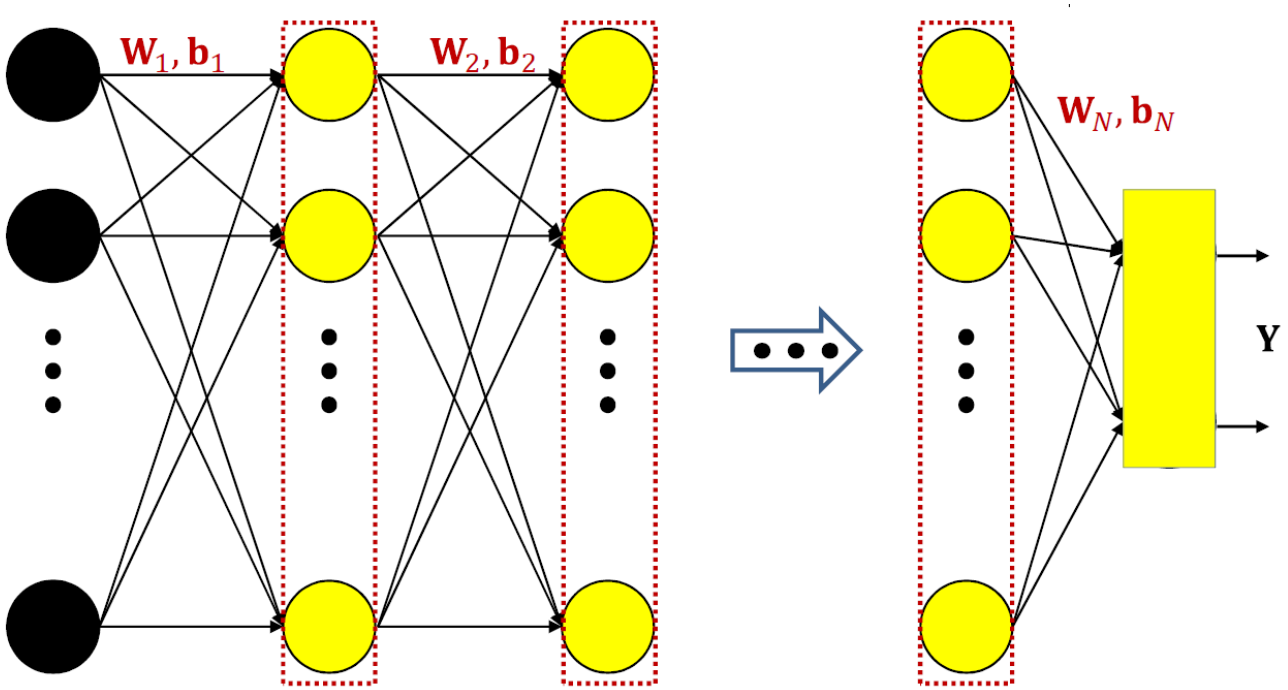
$$\boldsymbol{Y} = f_N(\mathbf{W}_N f_{N-1}(\dots f_2(\mathbf{W}_2 f_1(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)\dots) + \mathbf{b}_N)$$



- 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 network is a nested function

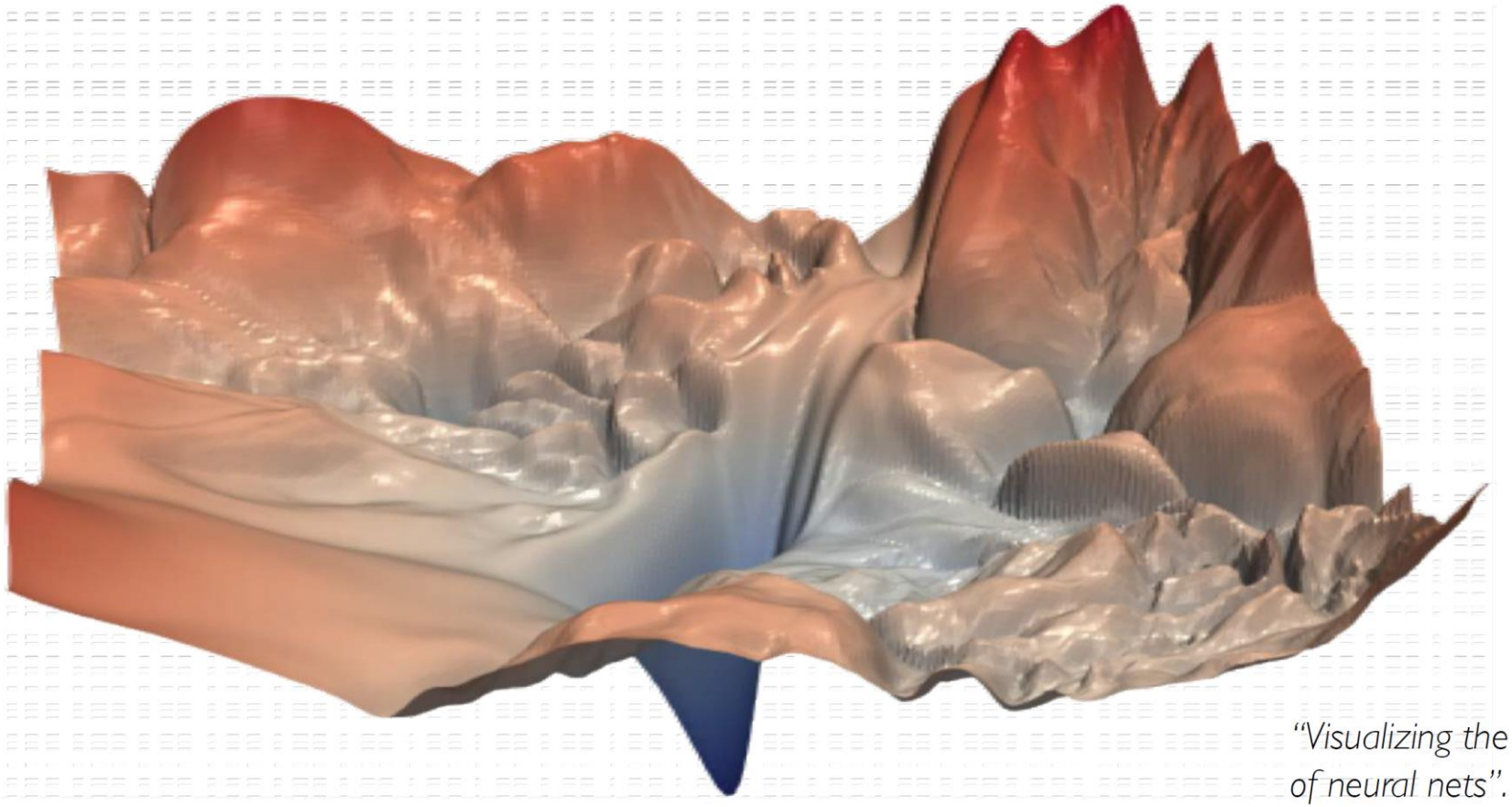
$$\boldsymbol{Y} = f_N(\mathbf{W}_N f_{N-1}(\dots f_2(\mathbf{W}_2 f_1(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2)\dots) + \mathbf{b}_N)$$

- The error for any is also a nested function

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

Neural Networks in Practice: Optimization

Training Neural Networks is Difficult



"Visualizing the loss landscape of neural nets". Dec 2017.

Loss Functions Can Be Difficult to Optimize

Remember:

Optimization through gradient descent

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

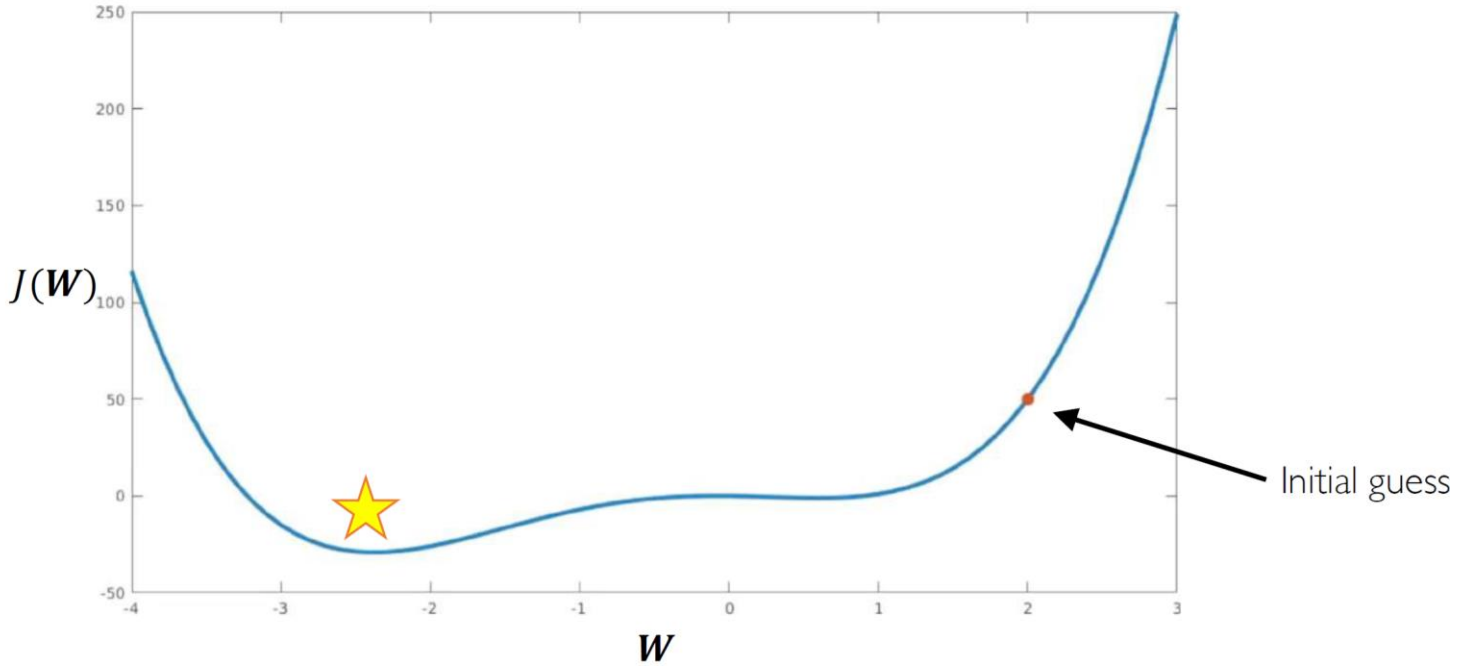
How can we set the learning rate?

Setting the Learning Rate

Small learning rate *converges slowly and gets stuck in false local minima*

Large learning rates *overshoot, become unstable and diverge*

Stable learning rates *converge smoothly and avoid local minima*



How to deal with this?

- Idea 1:

Try lots of different learning rates and see what works "just right"

- Idea 2:

Do something smarter! Design an adaptive learning rate that "adapts" to the landscape

Adaptive Learning Rates

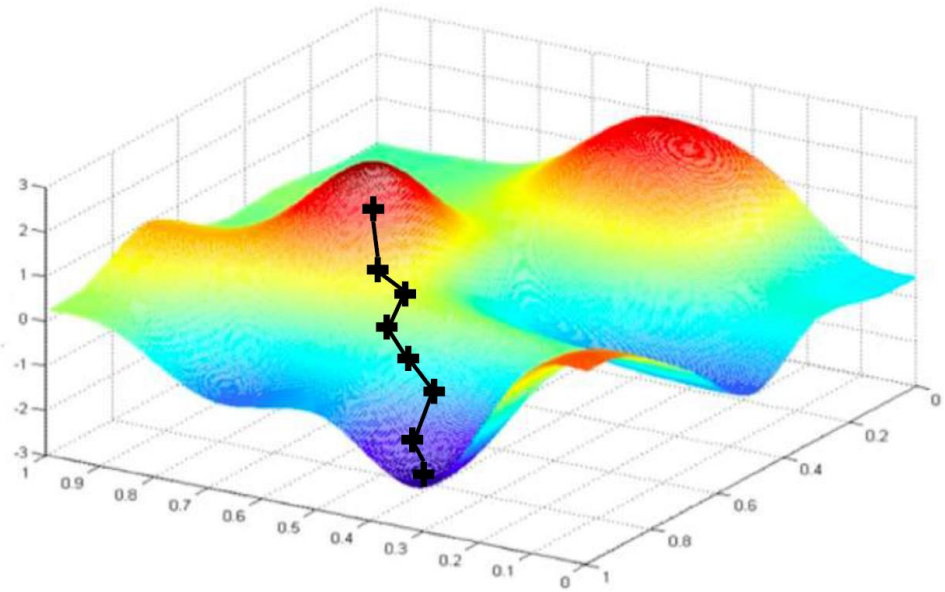
- Learning rates are no longer fixed
  - Can be made larger or smaller depending on:
  - how large gradient is
  - how fast learning is happening
  - size of particular weights
  - etc...

Neural Networks in Practice: Mini-batches

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights



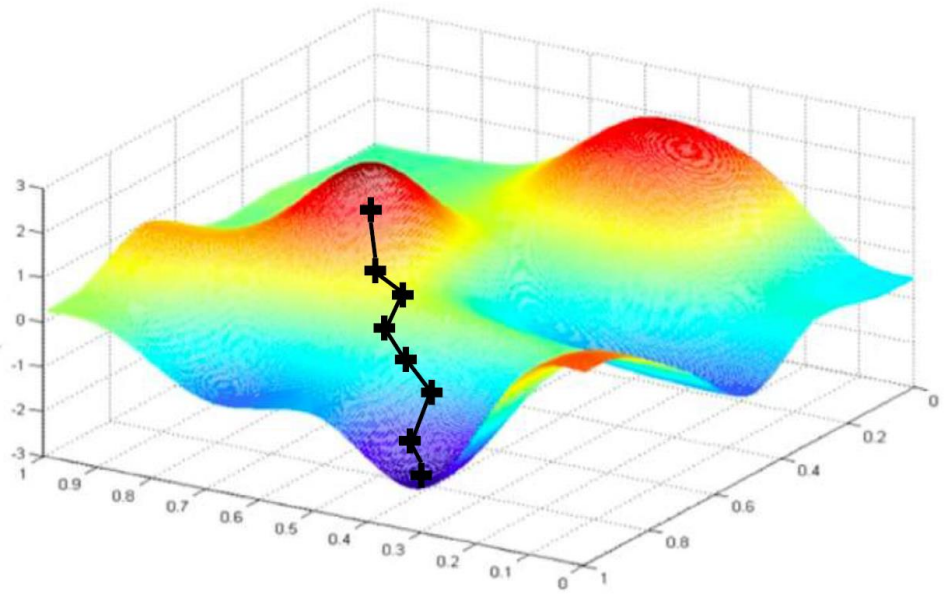
Can be very computational to compute!

Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point $\boldsymbol{i}$
4. Compute gradient, $\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

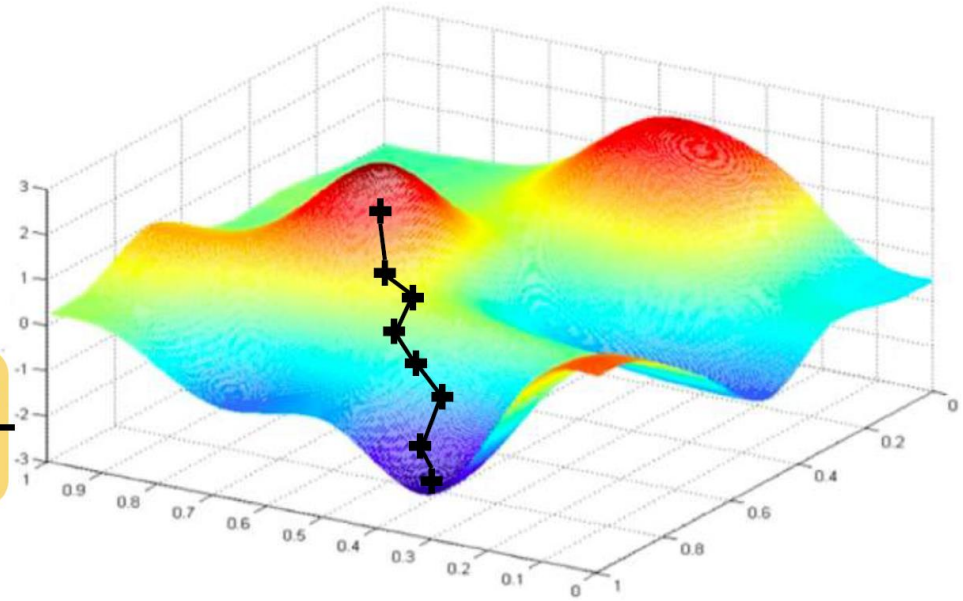
Easy to compute but **very noisy** (stochastic)!



Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



Fast to compute and a much better estimate of the true gradient!

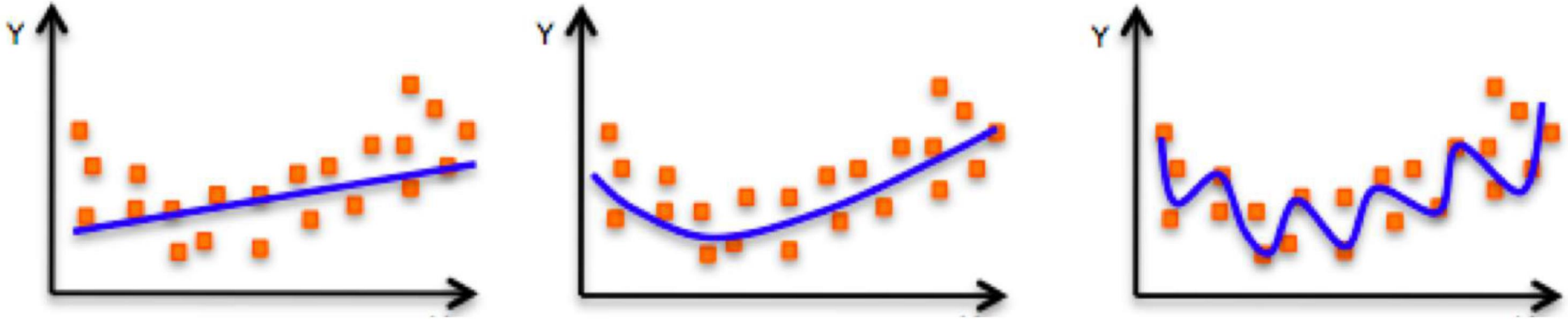
Mini-batches while training

- More accurate estimation of gradient
  - Smoother convergence
  - Allows for larger learning rates

- Mini-batches lead to fast training!
  - Can parallelize computation + achieve significant speed increases on GPU's

Neural Networks in Practice: Overfitting

The Problem of Overfitting



Underfitting
Model does not have capacity to fully learn the data

Ideal fit

Overfitting
Too complex, extra parameters, does not generalize well

Regularization

What is it?

Technique that constrains our optimization problem to discourage complex models

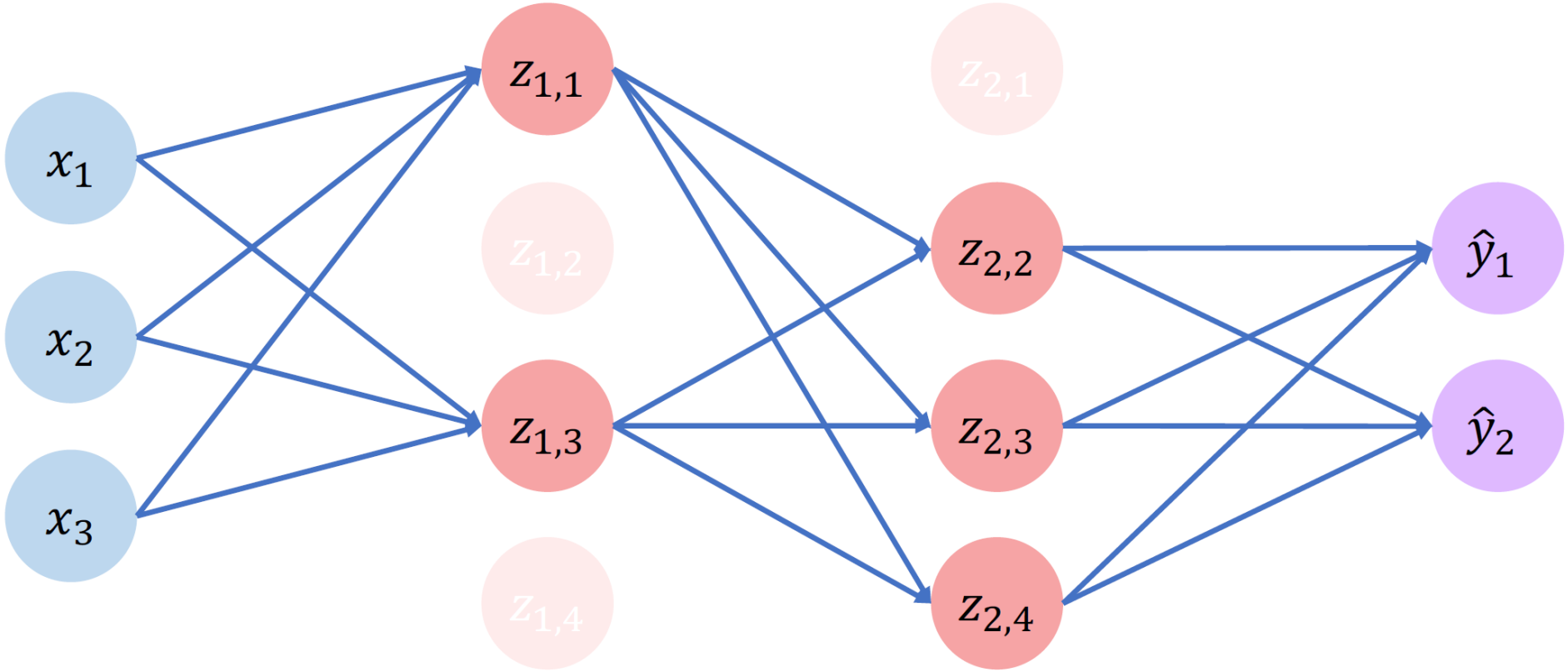
Why do we need it?

Improve generalization of our model on unseen data

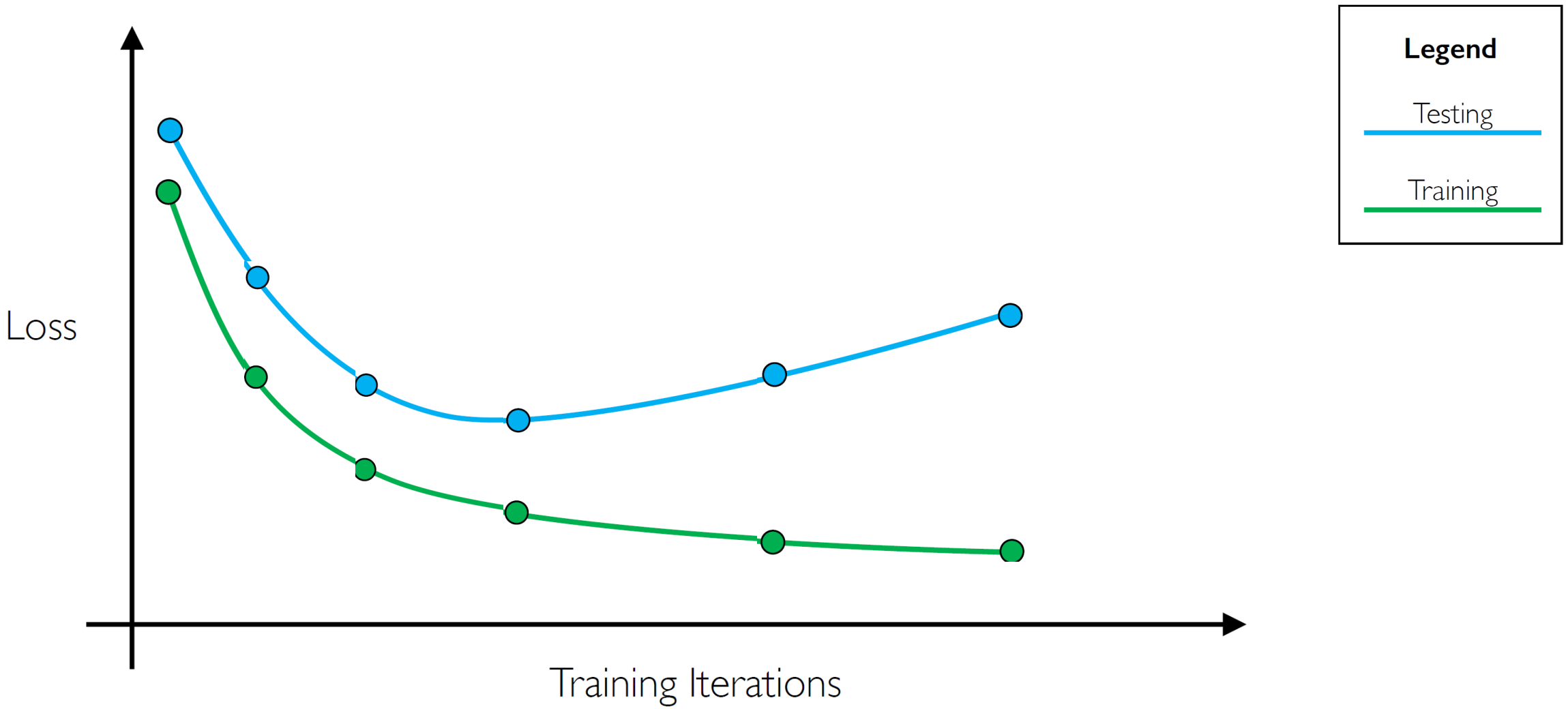
Regularization 1: Dropout

During training, randomly set some activations to 0

Typically 'drop' 50% of activations in layer



Regularization 2: Early Stopping



Regularization 2: Early Stopping

- Stop training before we have a chance to overfit