Neural Networks: Optimization Part 1

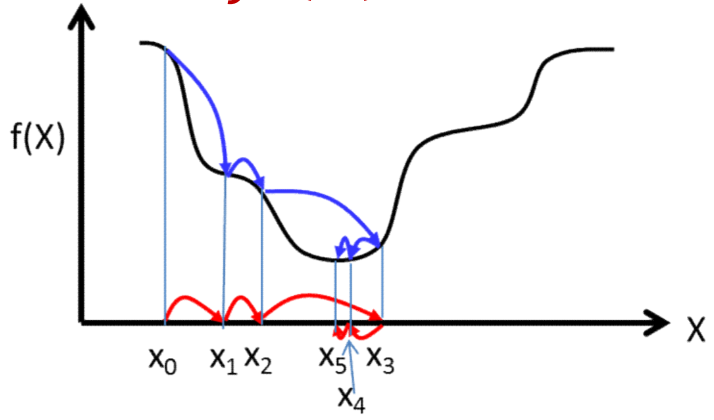
Intro to Deep Learning, Fall 2019

Story so far

- Neural networks are universal approximators
  - Can model any odd thing
  - Provided they have the right architecture
- We must *train* them to approximate any function
  - Specify the architecture
  - Learn their weights and biases
- Networks are trained to minimize total "loss" on a training set
  - We do so through empirical risk minimization
- We use variants of gradient descent to do so
- The gradient of the error with respect to network parameters is computed through backpropagation

Recap: Gradient Descent Algorithm

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



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

Training Neural Nets by Gradient Descent

Total training error:

$$Loss = \frac{1}{T}\sum_{t} Div(\boldsymbol{Y}_t, \boldsymbol{d}_t; \mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_K)$$

- Gradient descent algorithm:
- Initialize weights $\mathbf{W}_k$ for every layer $k = 1 \dots K$
- Do:
  - For every layer $k = 1 \dots K$ compute:
    - $\nabla_{\mathbf{W}_k} Loss = \frac{1}{T}\sum_t \nabla_{\mathbf{W}_k} Div(\boldsymbol{Y}_t, \boldsymbol{d}_t)$
    - $\mathbf{W}_k = \mathbf{W}_k - \eta \nabla_{\mathbf{W}_k} Loss^T$
- Until $Loss$ has converged

Training Neural Nets by Gradient Descent

Total training error:

$$Loss = \frac{1}{T}\sum_{t} Div(\boldsymbol{Y}_t, \boldsymbol{d}_t; \mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K)$$

- Gradient descent algorithm:
- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K$
- Do:
  - For every layer k, compute:
    - $\nabla_{\mathbf{W}_k} Loss = \frac{1}{T}\sum_t \nabla_{\mathbf{W}_k} Div(\boldsymbol{Y}_t, \boldsymbol{d}_t)$
    - $\mathbf{W}_k = \mathbf{W}_k - \eta \nabla_{\mathbf{W}_k} Loss^T$
- Until $Loss$ has converged

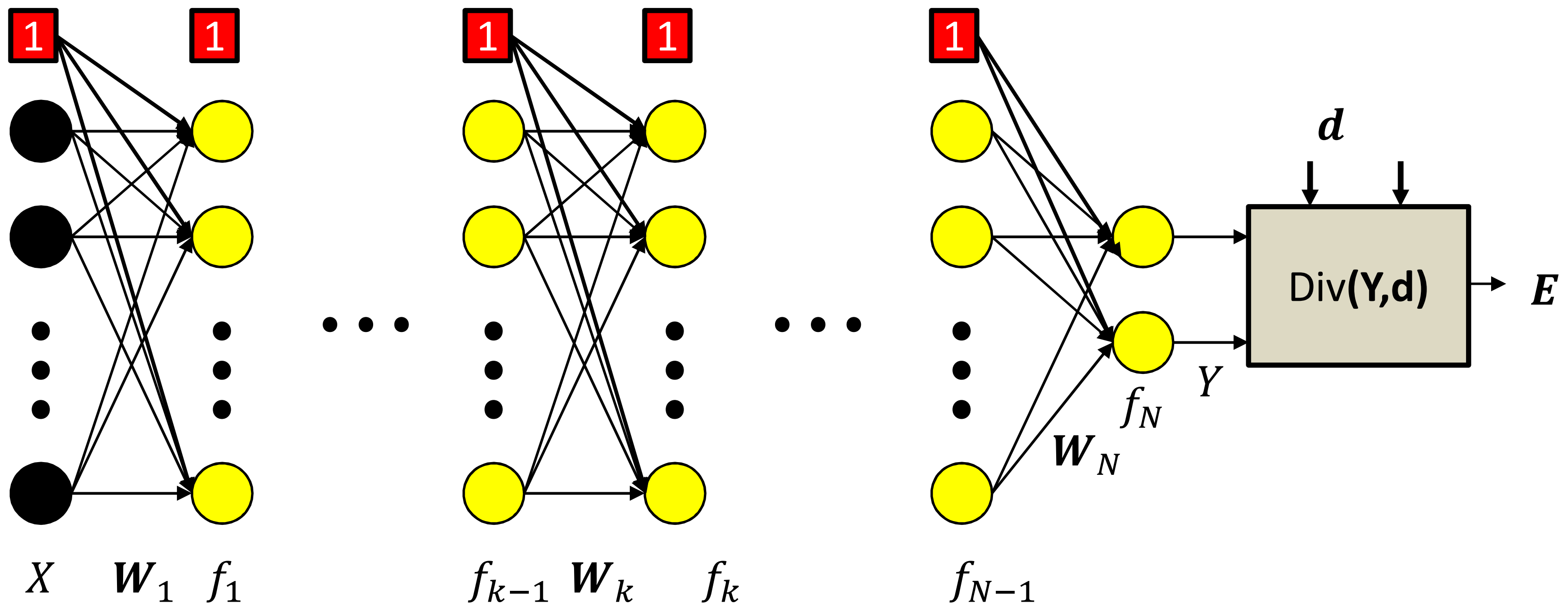
Training by gradient descent

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

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

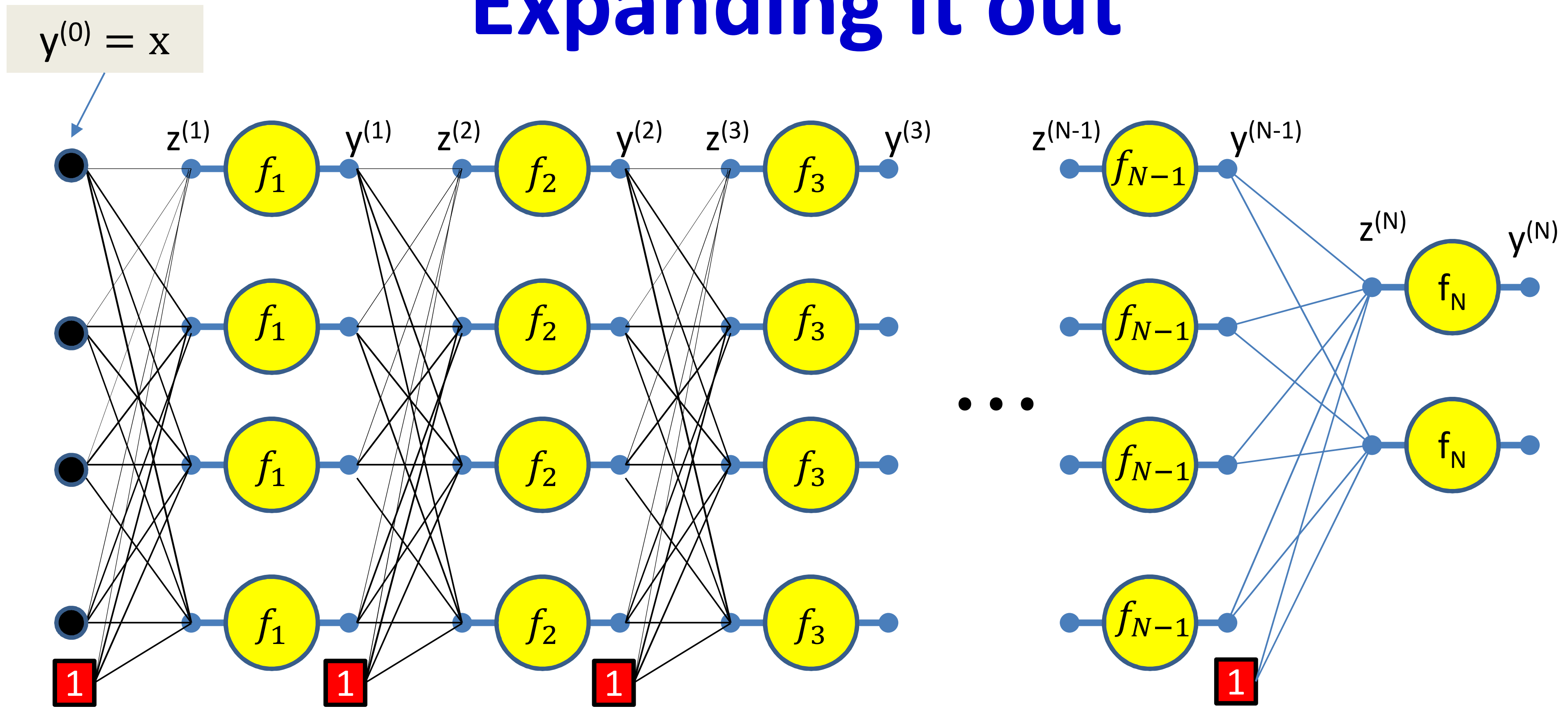
- Until Err has converged

BP: Scalar Formulation



- The network again

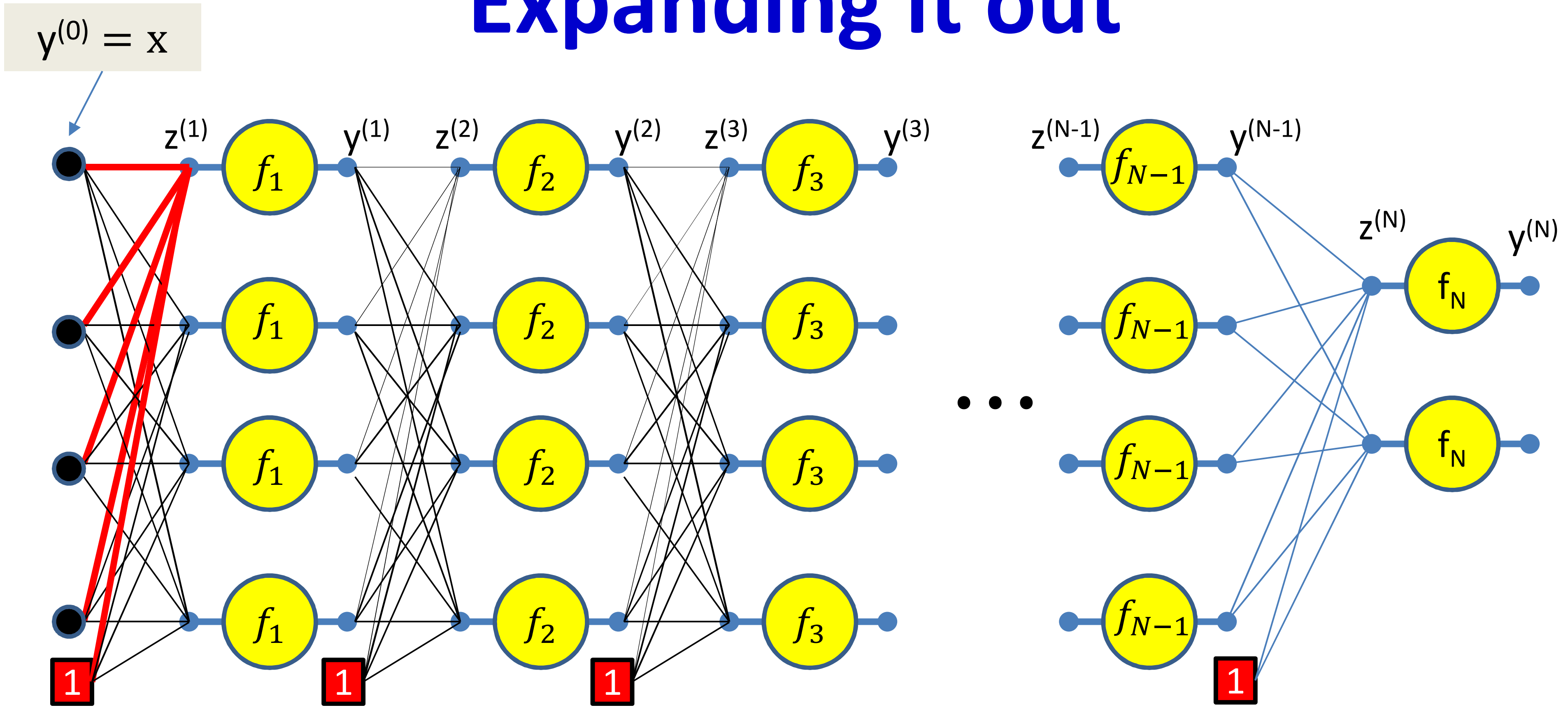
Expanding it out



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

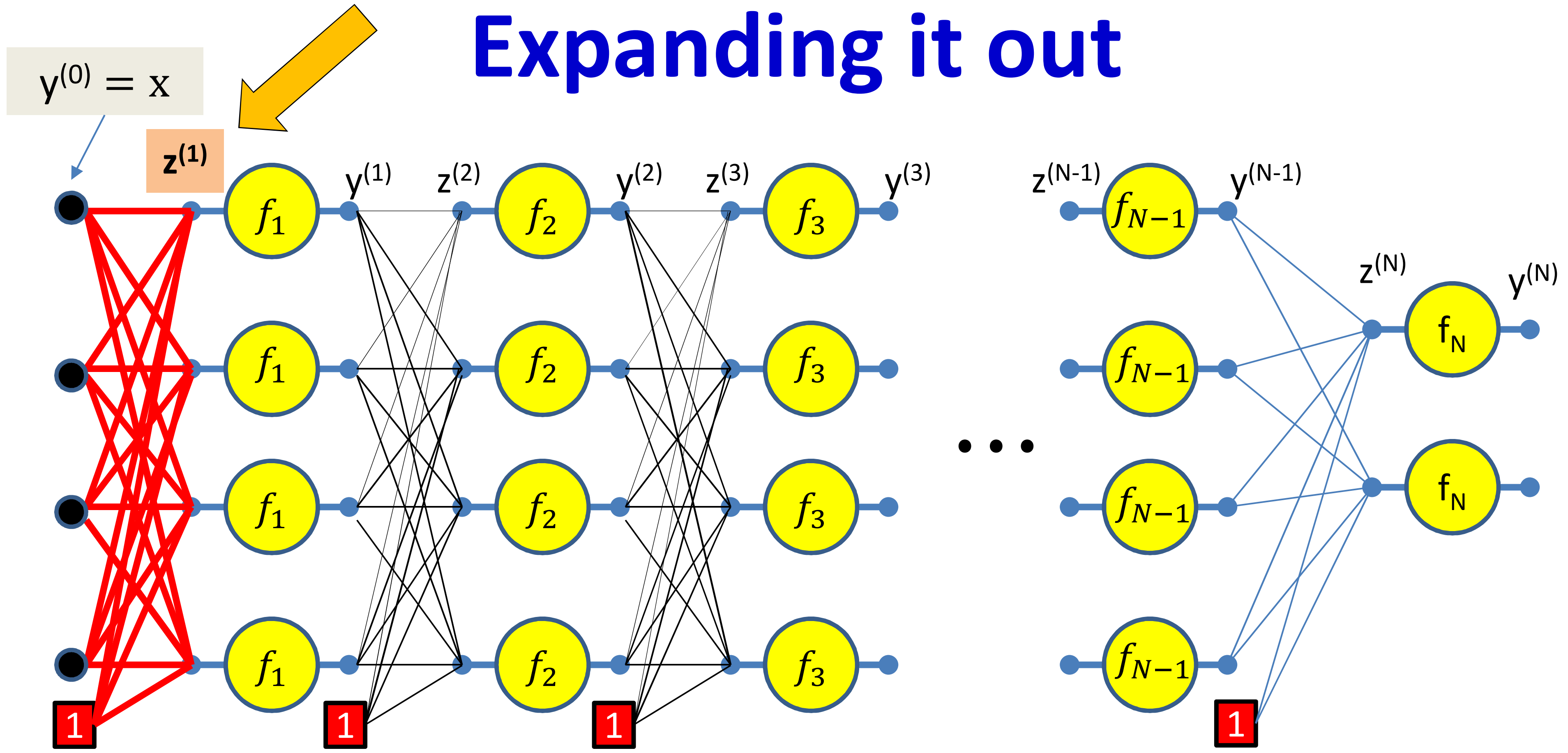
Assuming $w_{0j}^{(k)} = b_j^{(k)}$ and $y_0^{(k)} = 1$ -- assuming the bias is a weight and extending the output of every layer by a constant 1, to account for the biases

Expanding it out

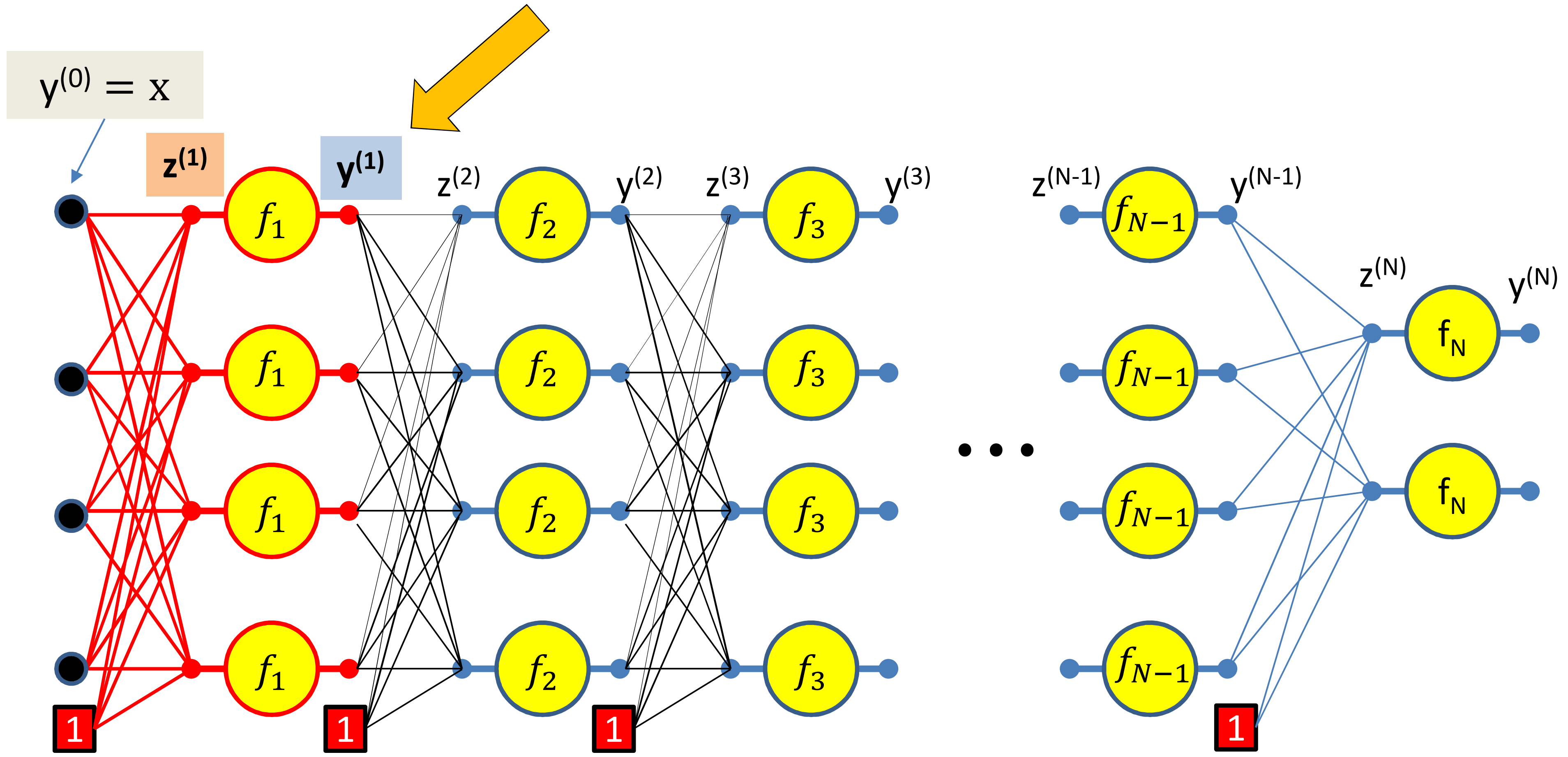


$$\boxed{z_1^{(1)} = \sum_i w_{i1}^{(1)} y_i^{(0)}}$$

Expanding it out

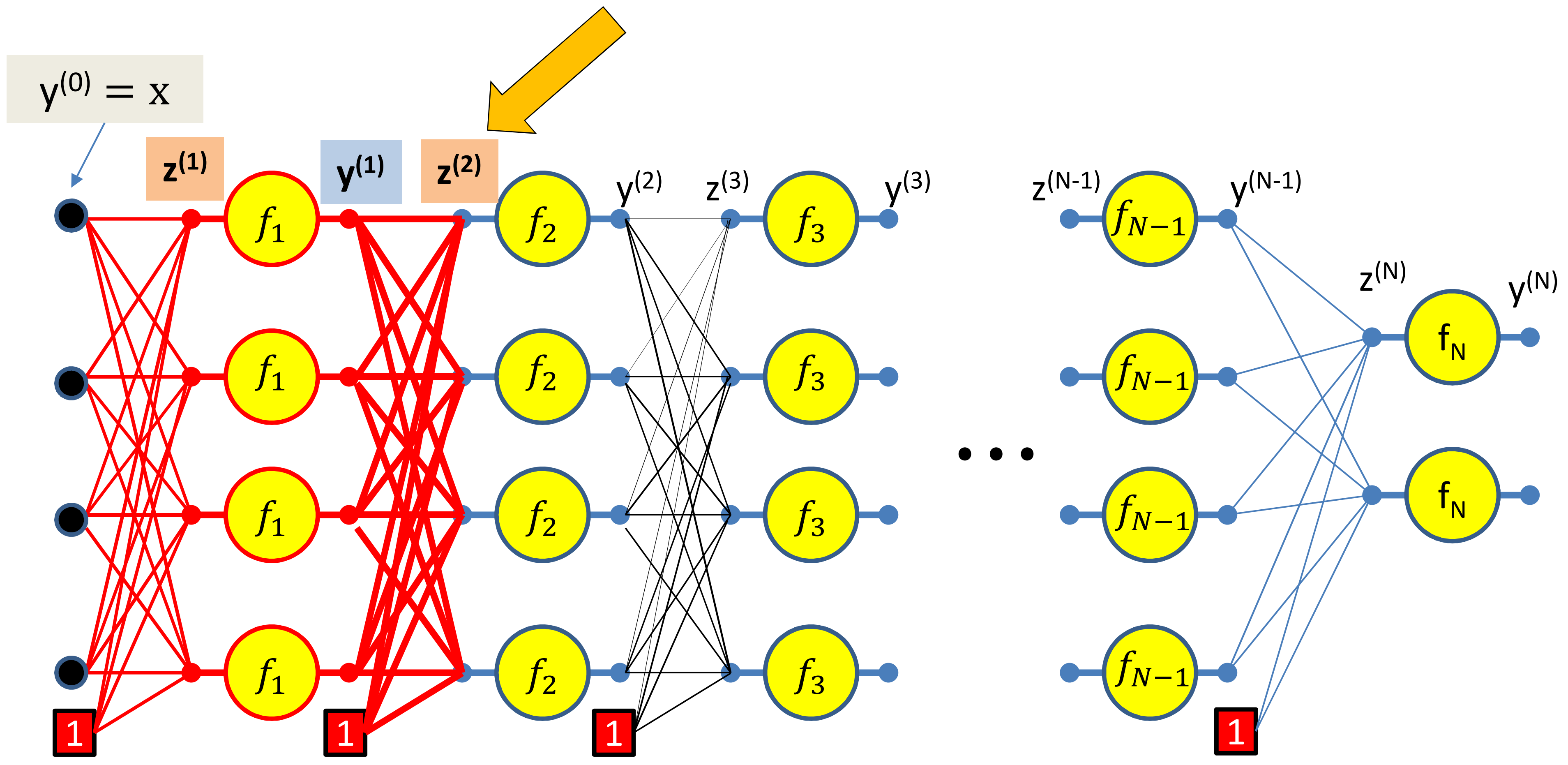


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$



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

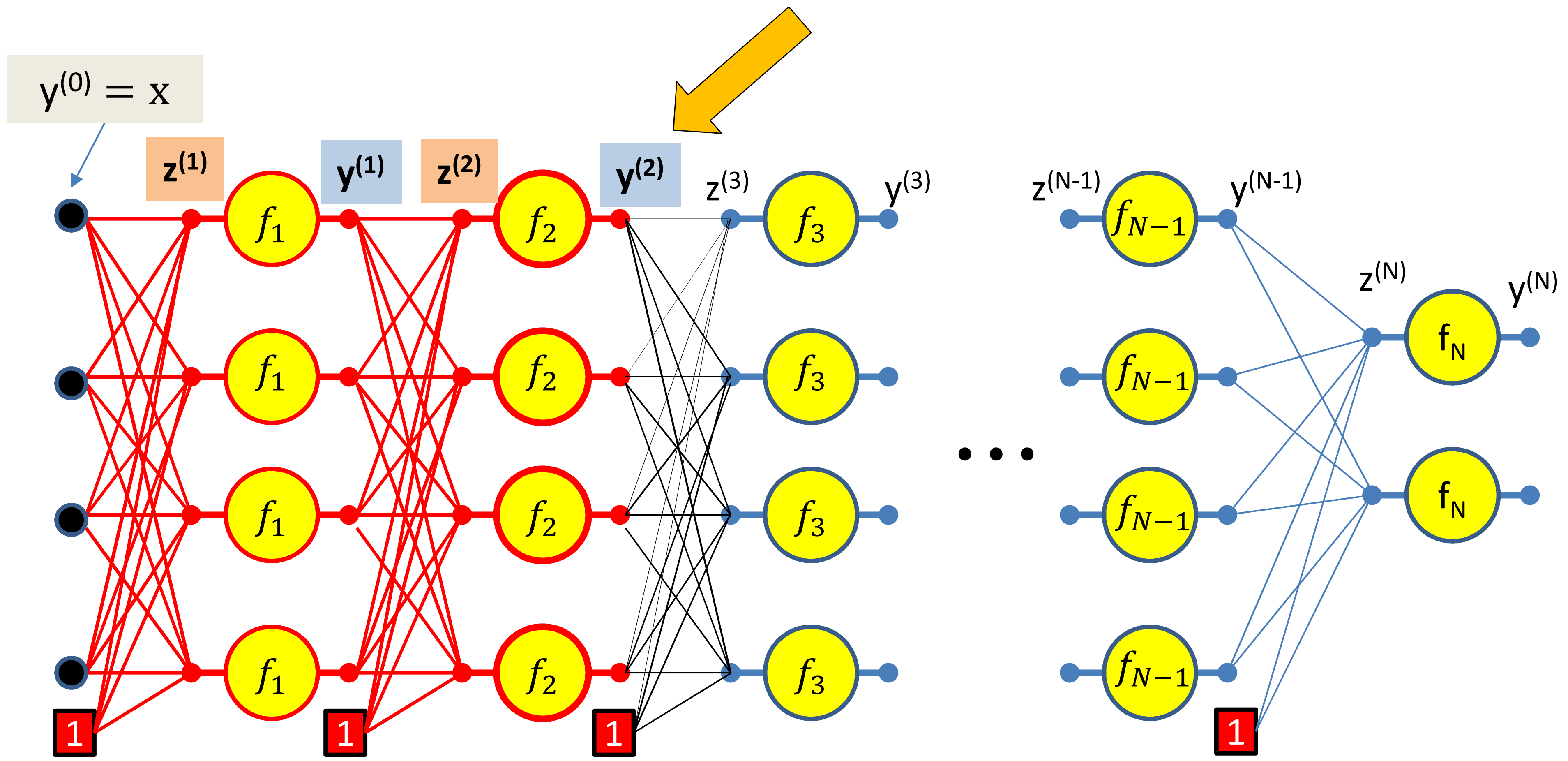
$$\boxed{y_j^{(1)} = f_1\left(z_j^{(1)}\right)}$$



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

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

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

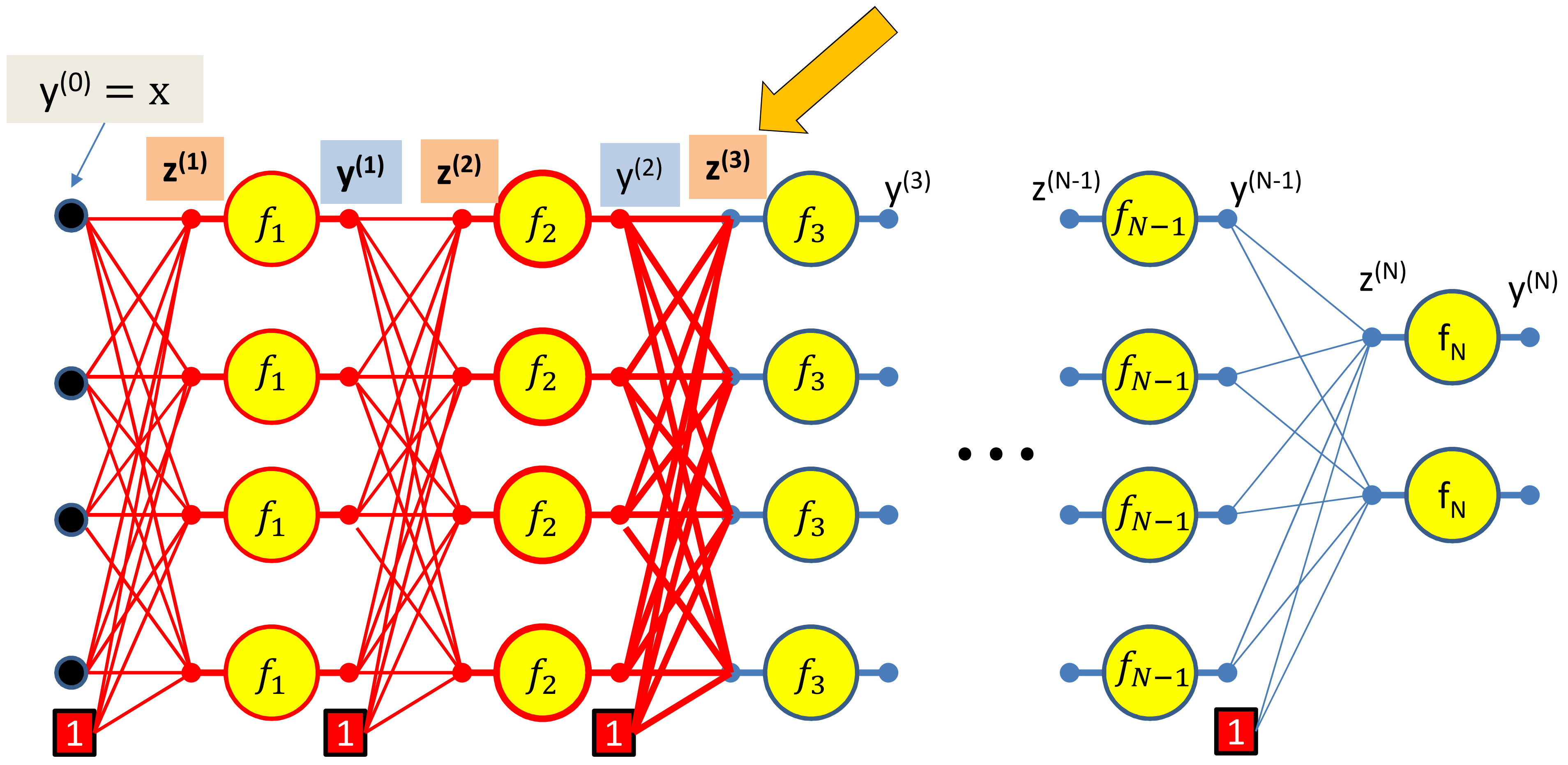


$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$$

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

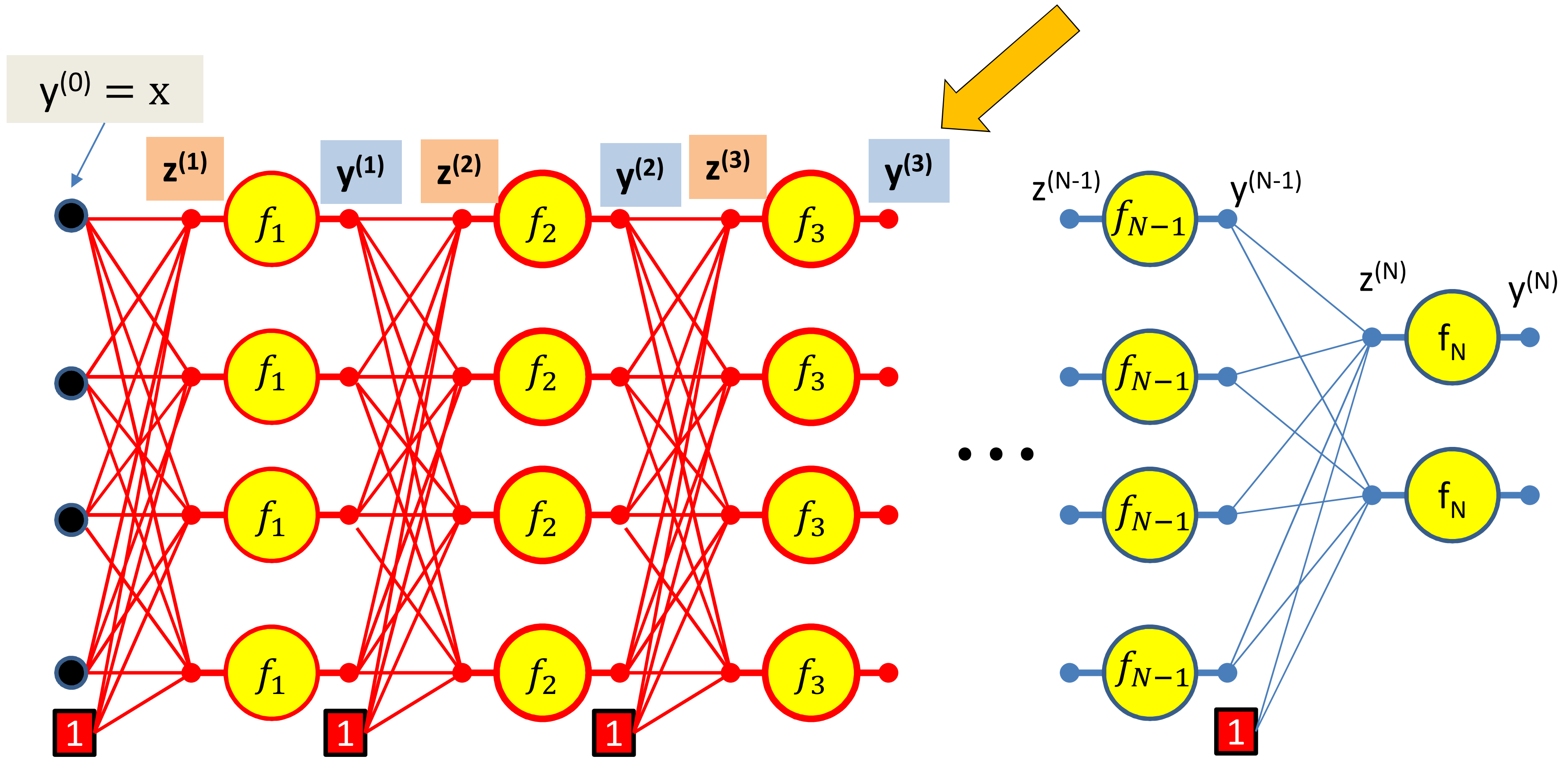
$$z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)}$$

$$\boxed{y_j^{(2)} = f_2\left(z_j^{(2)}\right)}$$



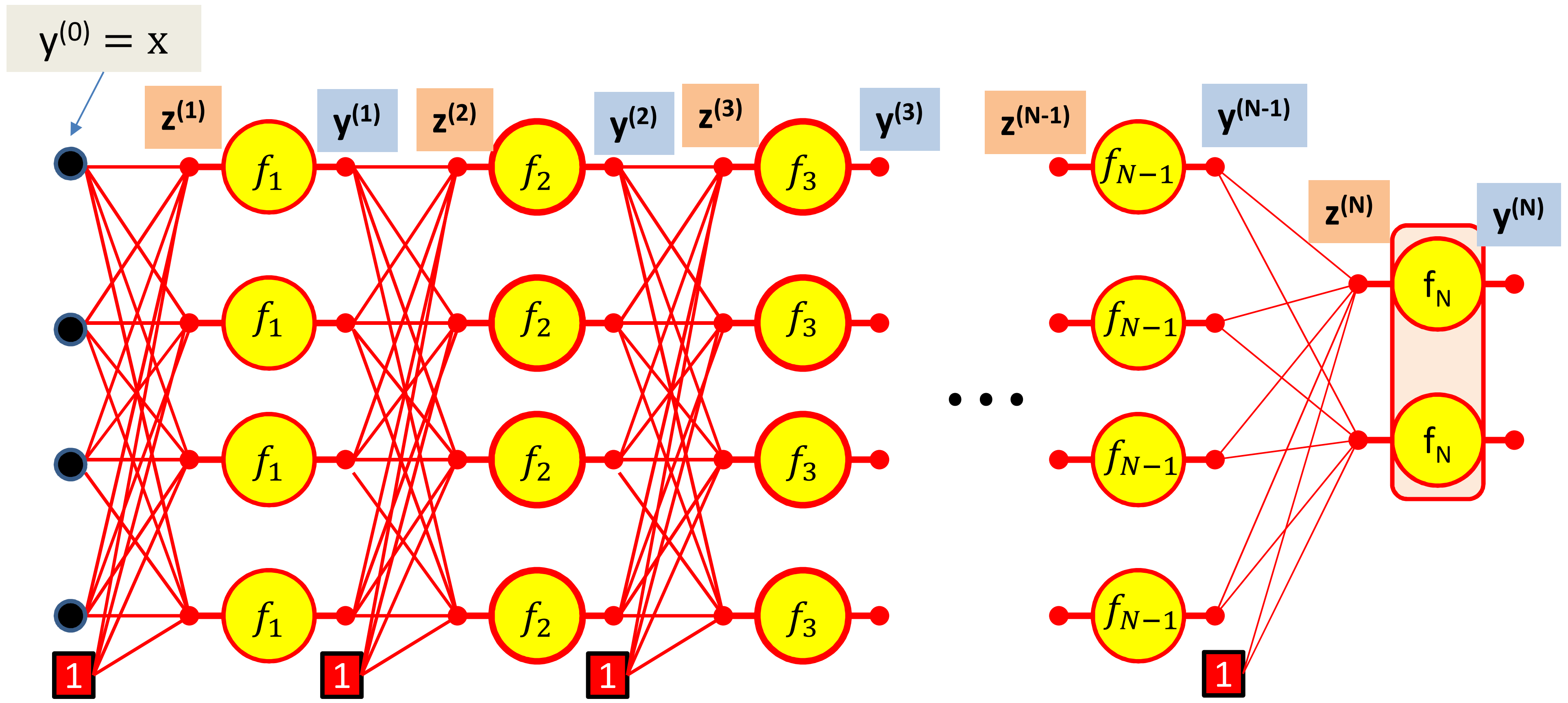
$$z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)} \qquad y_j^{(1)} = f_1\left(z_j^{(1)}\right) \qquad z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)} \qquad y_j^{(2)} = f_2\left(z_j^{(2)}\right)$$

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



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

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

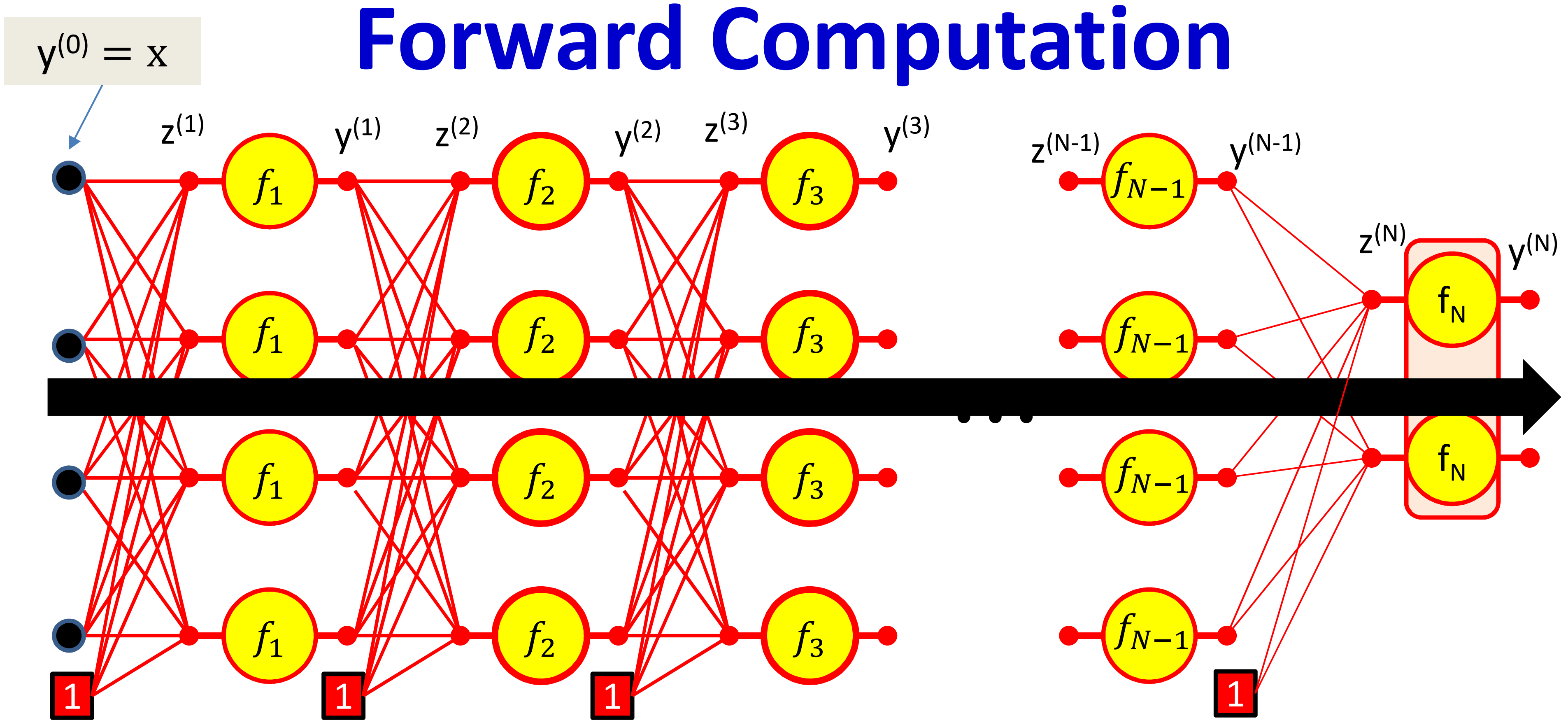


$$y_j^{(N-1)} = f_{N-1}\left(z_j^{(N-1)}\right)$$

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

$$\boldsymbol{y}^{(N)} = f_N\left(\boldsymbol{z}^{(N)}\right)$$

Forward Computation



ITERATE FOR $k = 1{:}N$

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

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

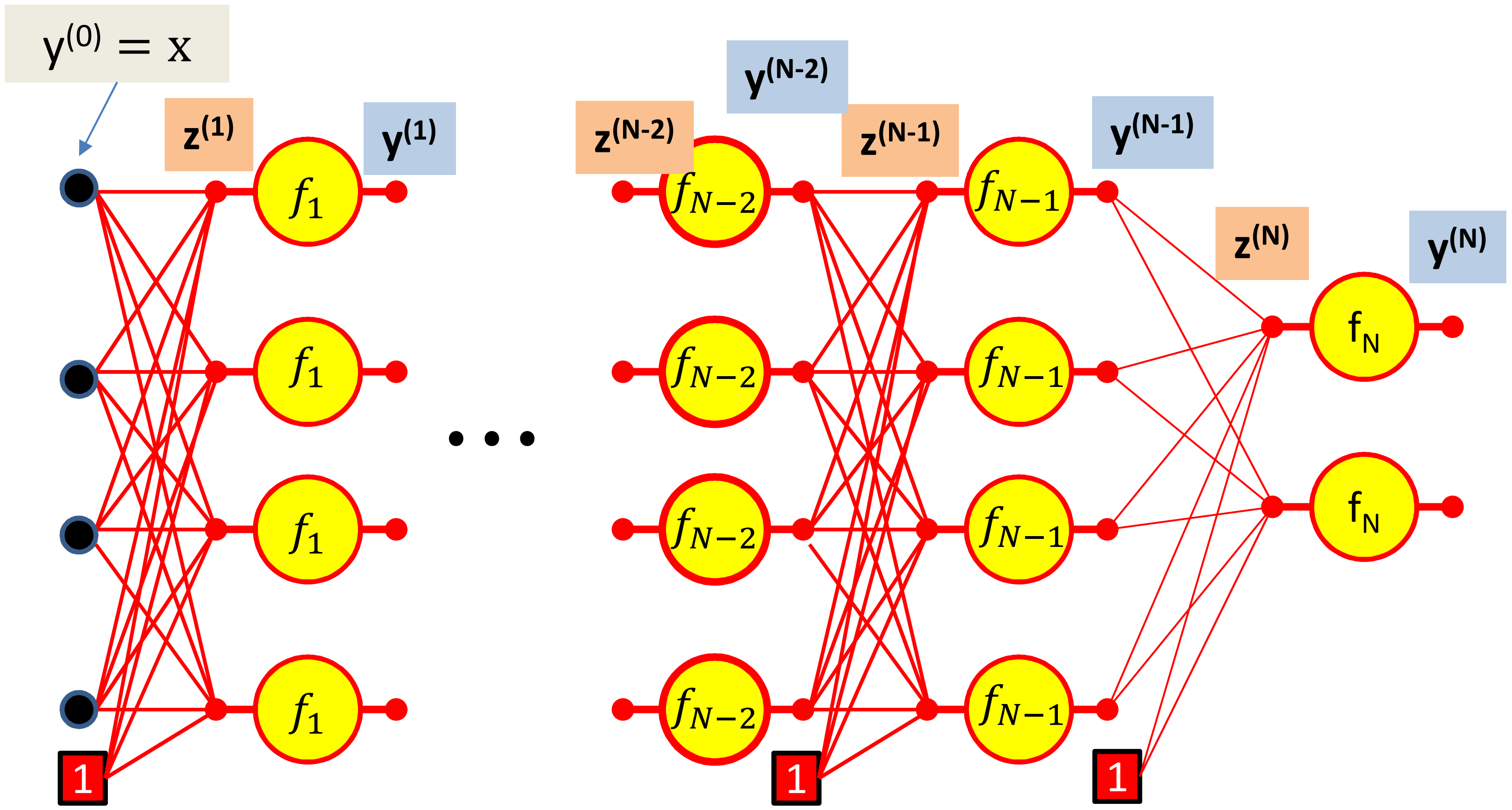
$$z_j^{(k)} = \sum_i w_{ij}^{(k)} y_i^{(k-1)}$$

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

Forward "Pass"

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

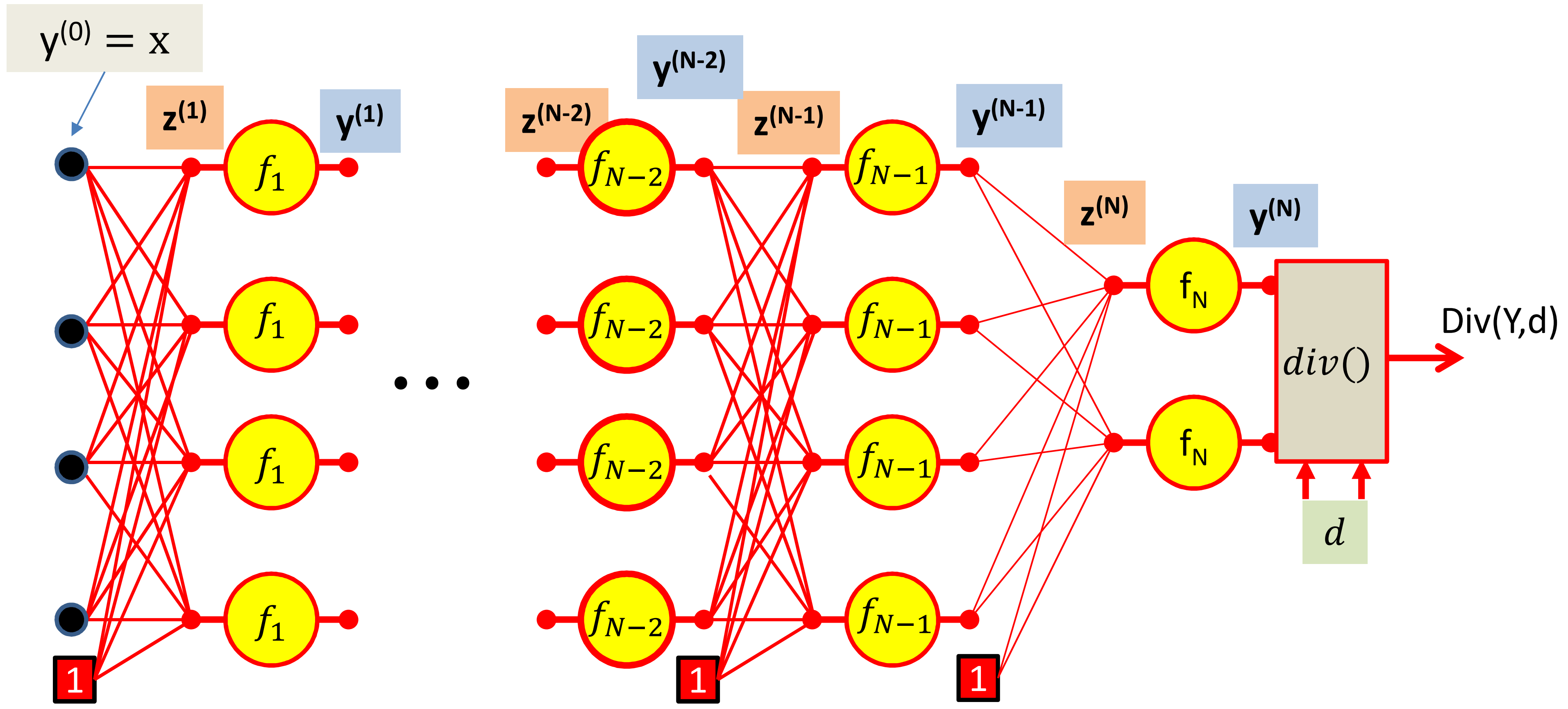
Computing derivatives



We have computed all these intermediate values in the forward computation

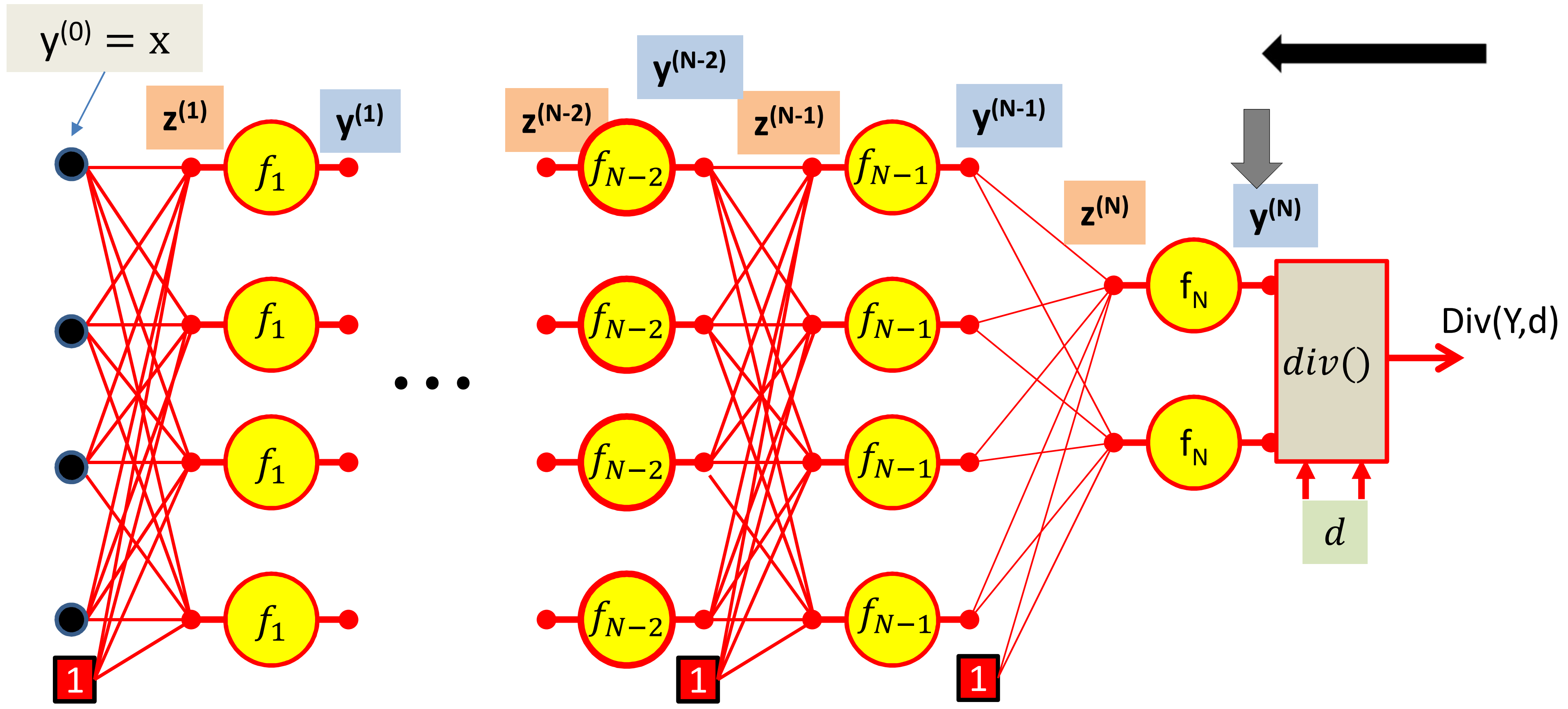
We must remember them – we will need them to compute the derivatives

Computing derivatives



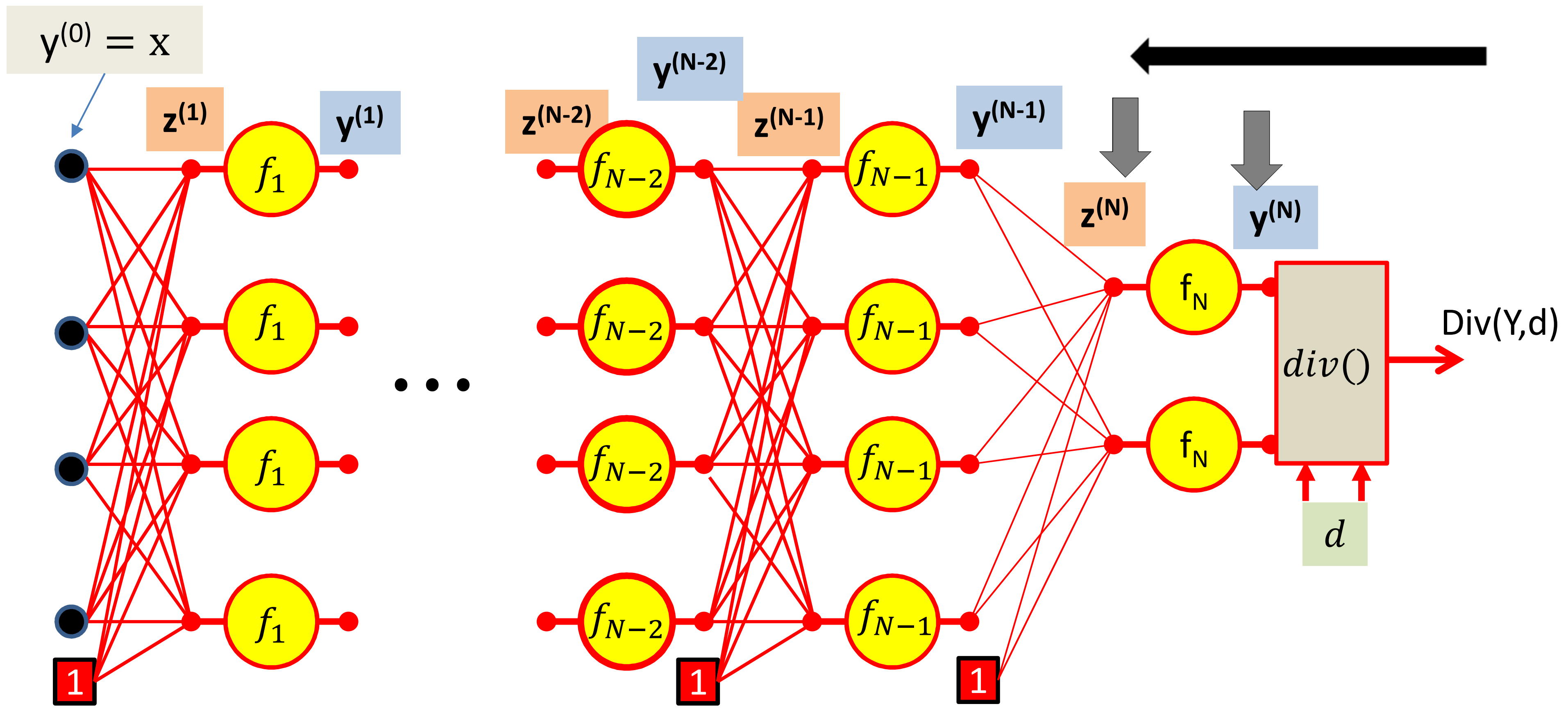
First, we compute the divergence between the output of the net y = $y^{(N)}$ and the desired output d

Computing derivatives



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

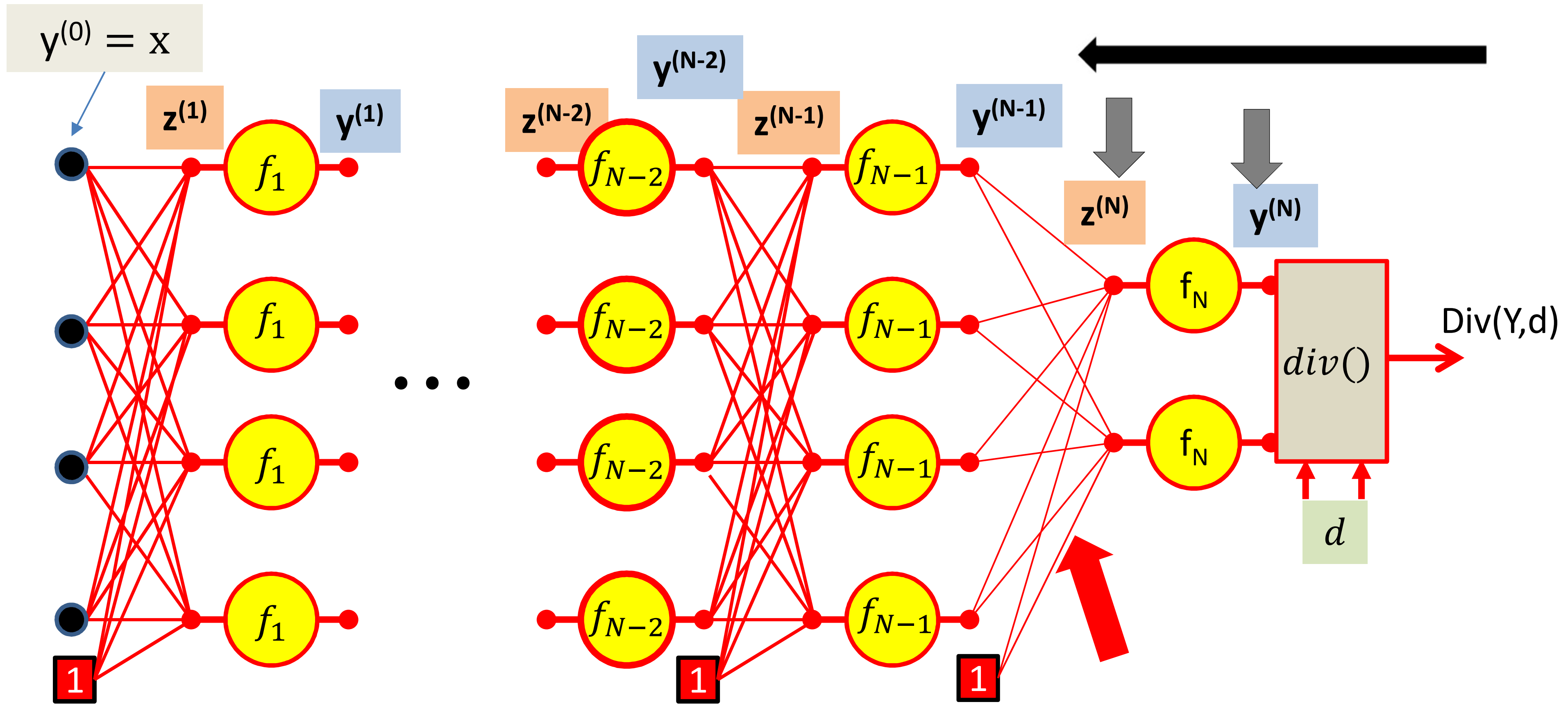
Computing derivatives



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

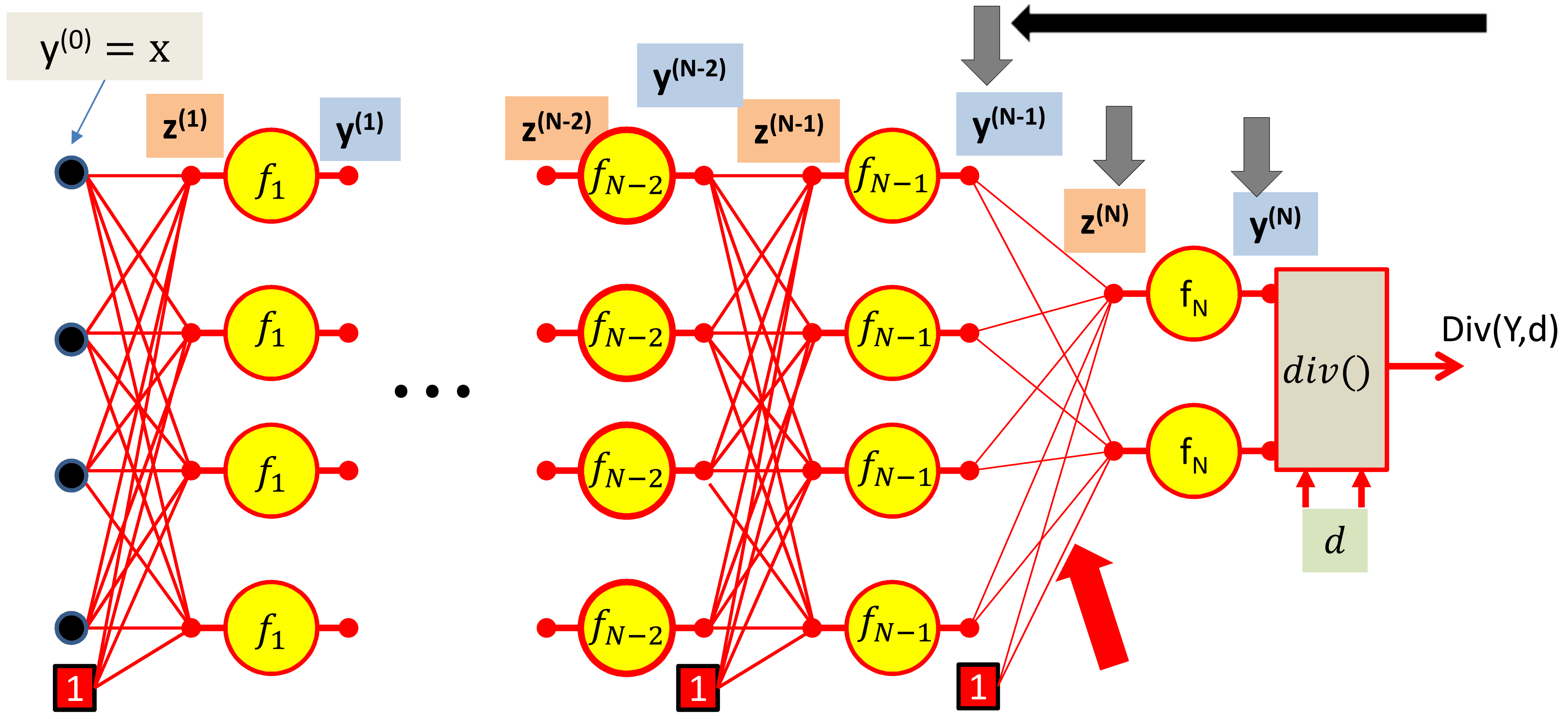
We then compute $\nabla_{z^{(N)}} div(.)$ the derivative of the divergence w.r.t. the *pre-activation* affine combination $z^{(N)}$ using the chain rule

Computing derivatives



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

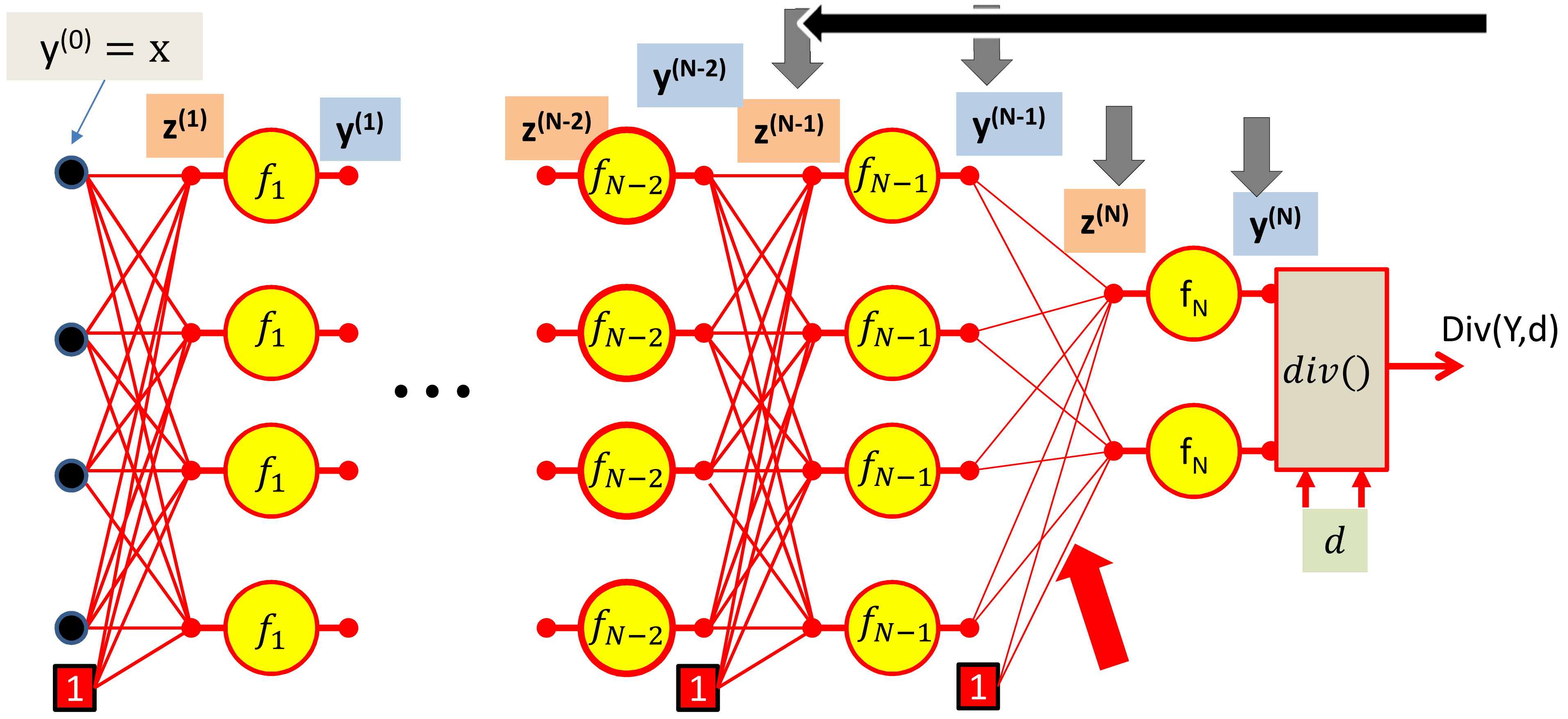
Computing derivatives



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

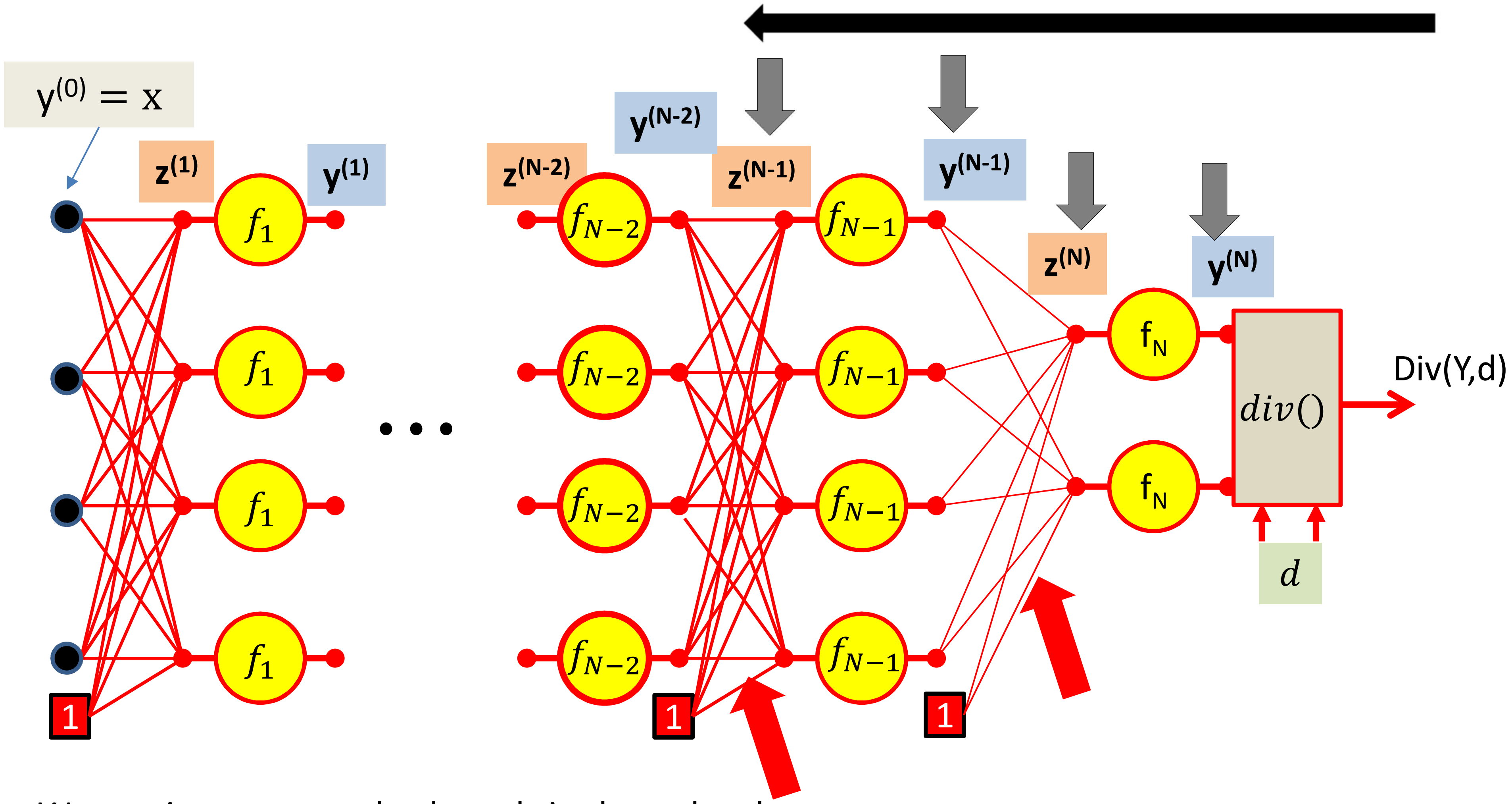
Then continue with the chain rule to compute $\nabla_{Y^{(N-1)}} div(.)$ the derivative of the divergence w.r.t. the output of the N-1th layer

Computing derivatives



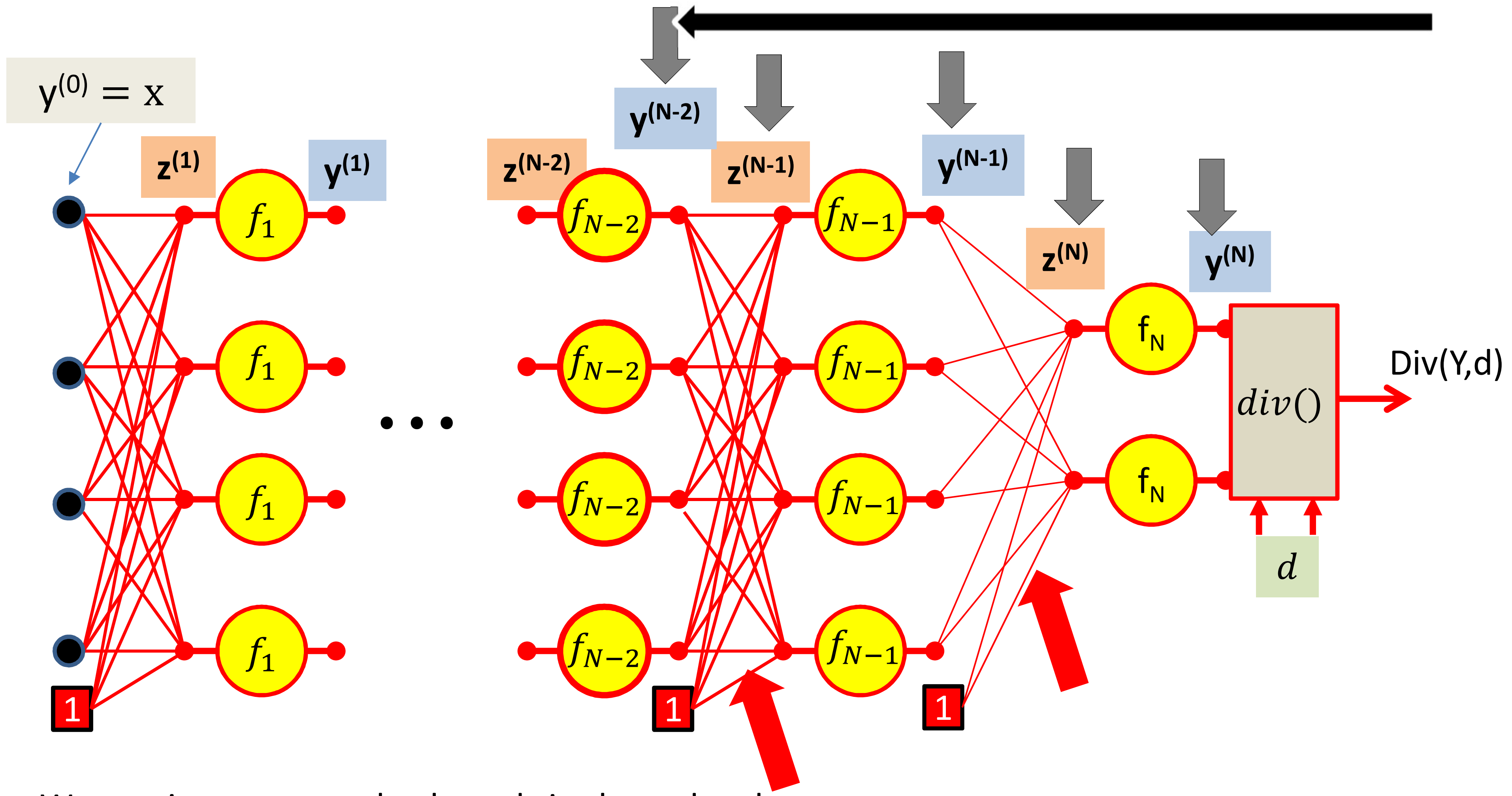
We continue our way backwards in the order shown

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



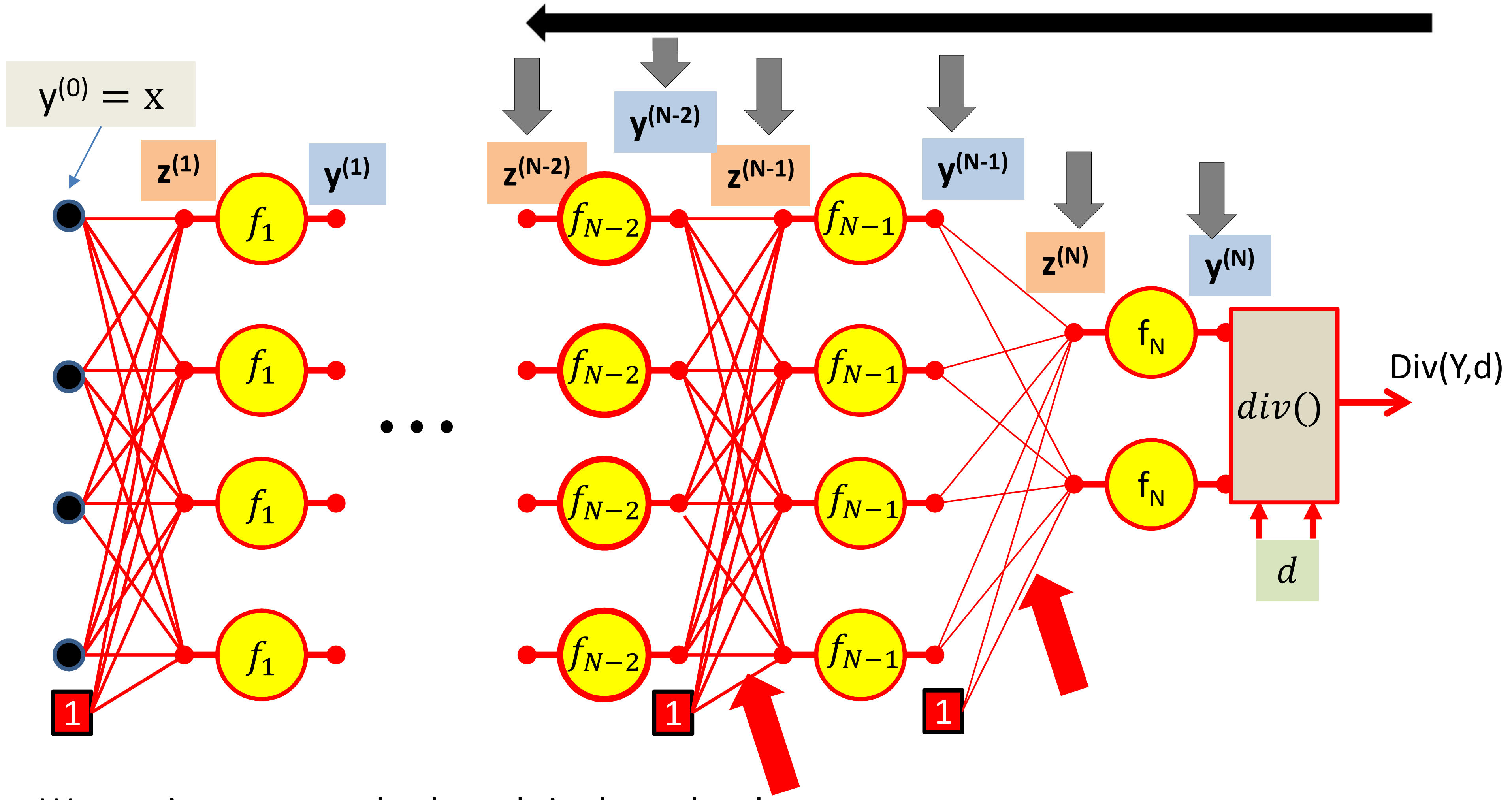
We continue our way backwards in the order shown

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



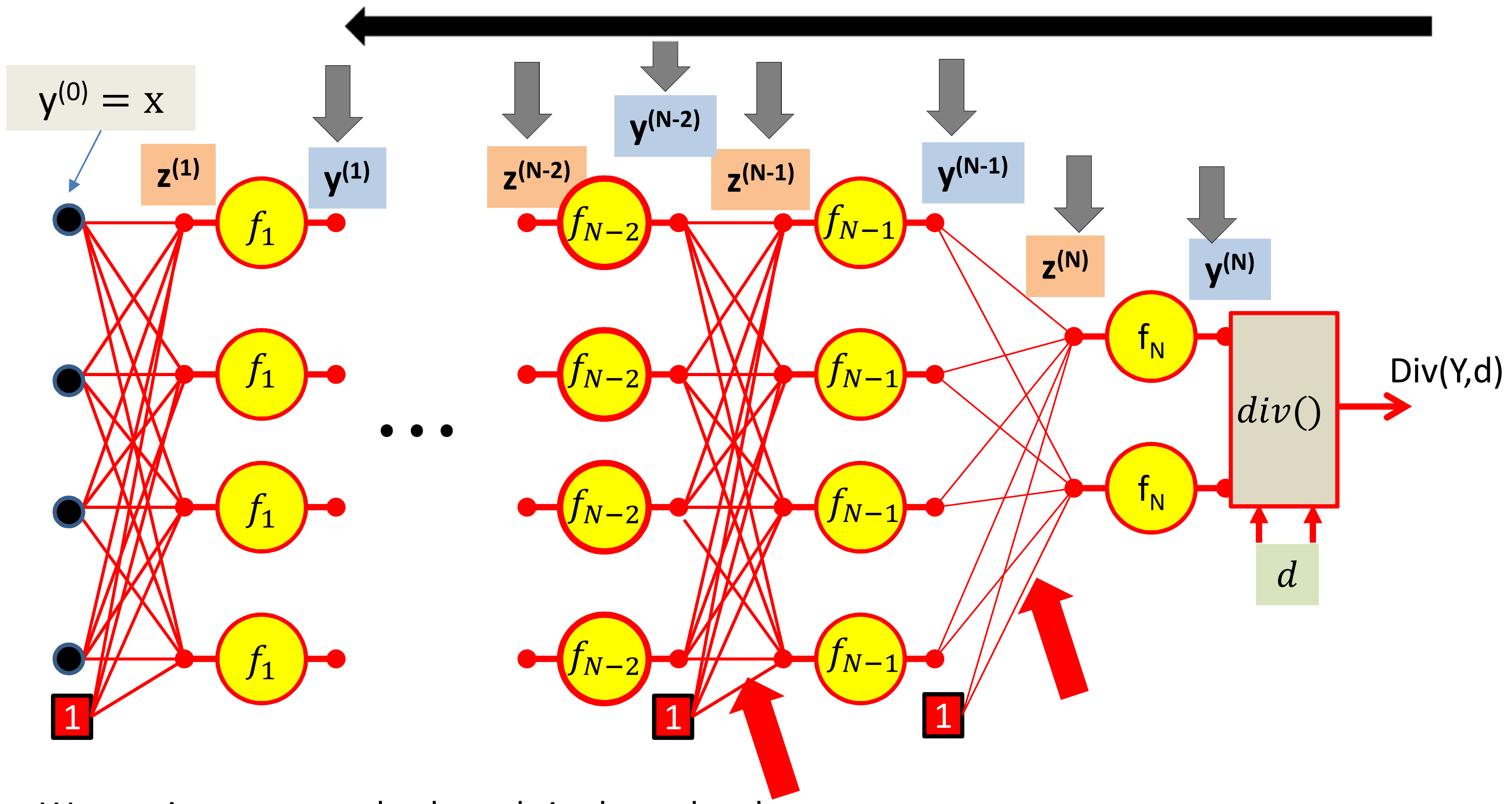
We continue our way backwards in the order shown

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



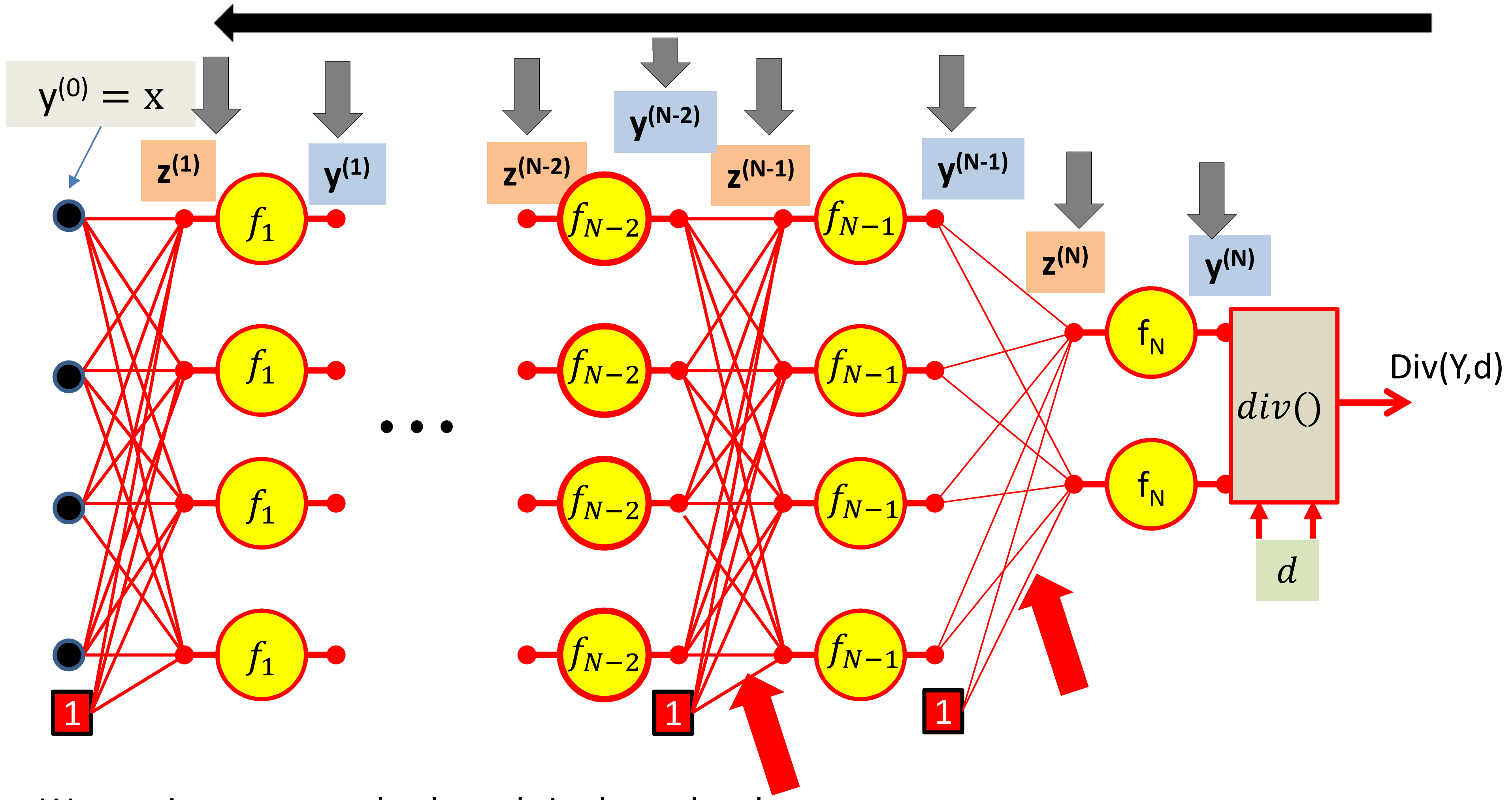
We continue our way backwards in the order shown

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



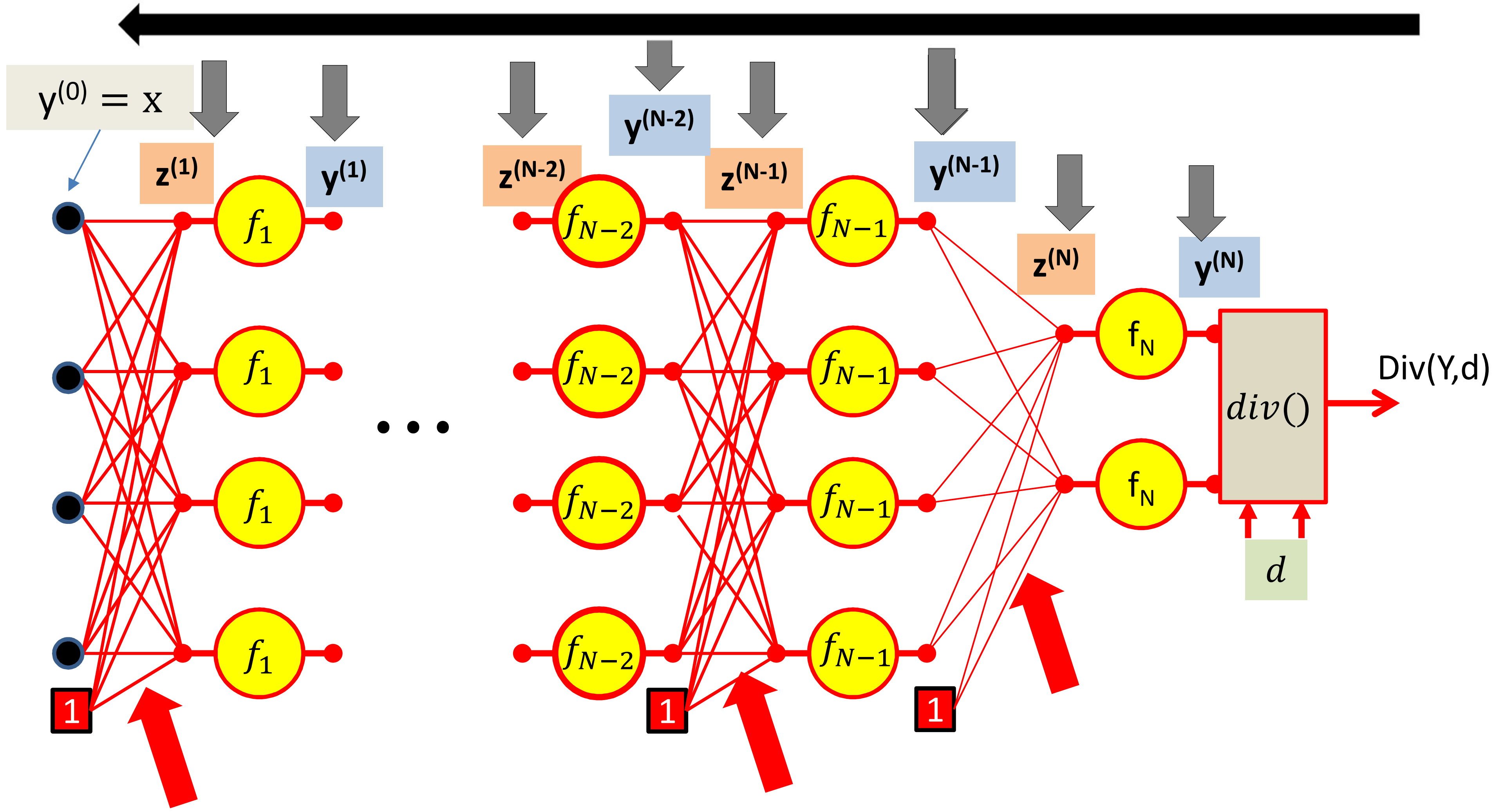
We continue our way backwards in the order shown

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



We continue our way backwards in the order shown

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



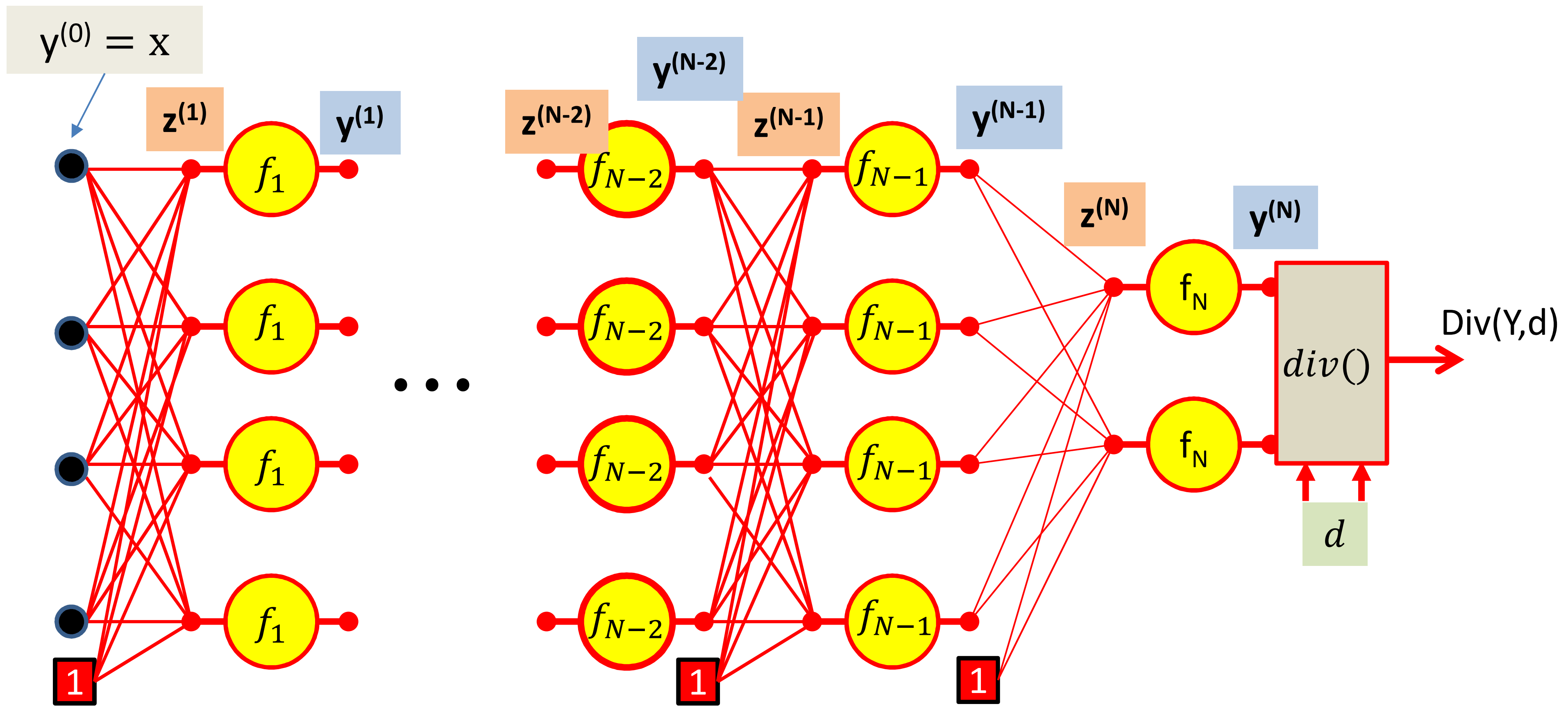
$\nabla_{W^{(1)}} div(.)$

We continue our way backwards in the order shown

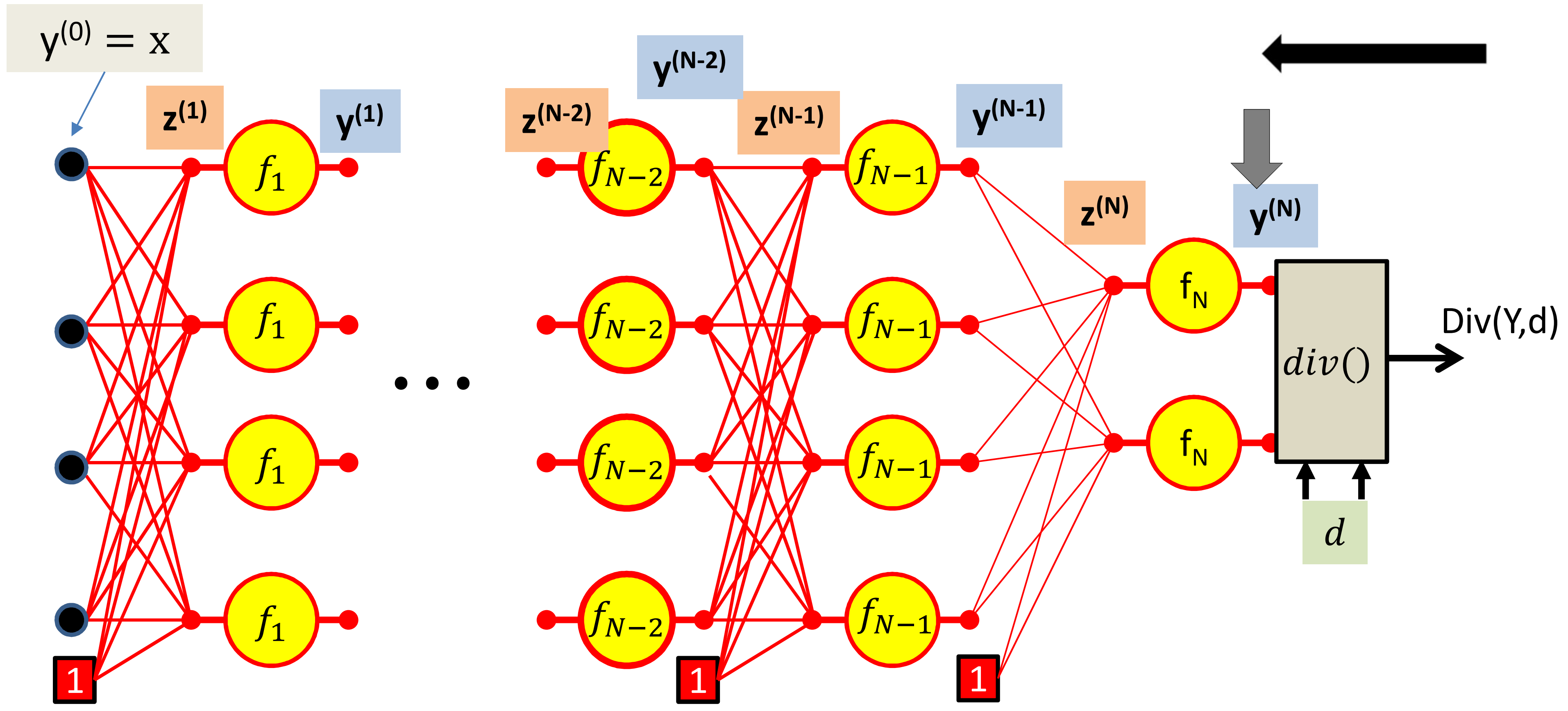
Backward Gradient Computation

- Lets actually see the math..

Computing derivatives



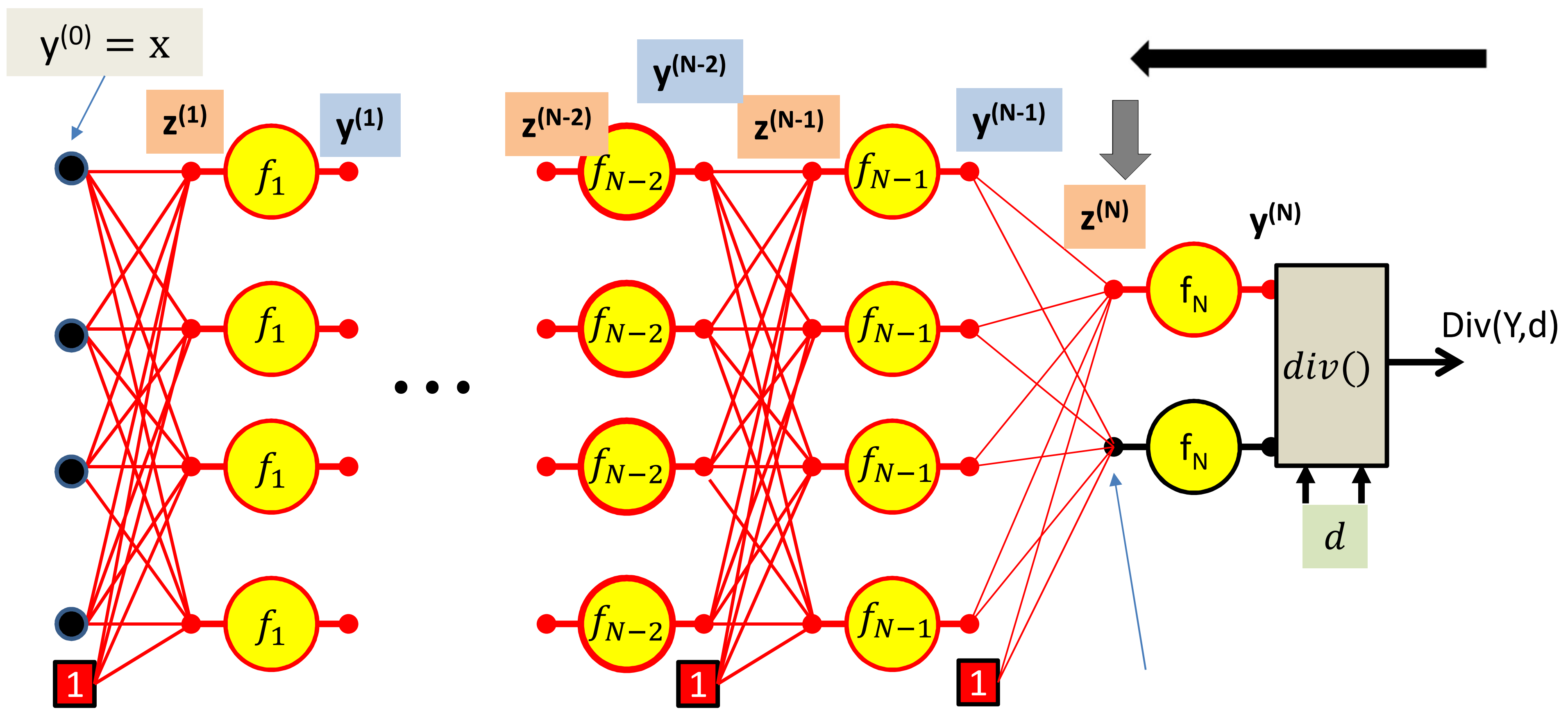
Computing derivatives



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

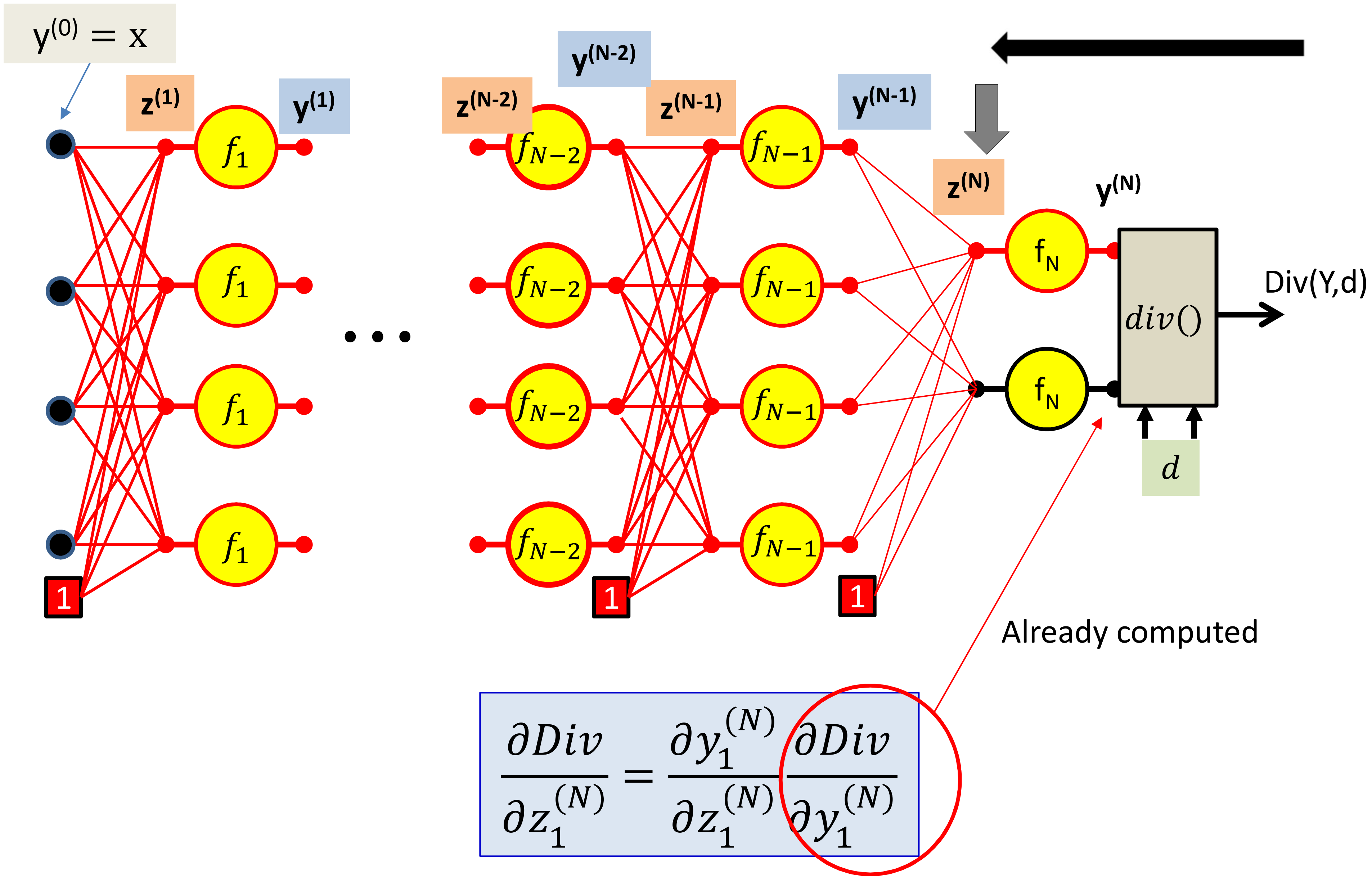
$$\frac{\partial Div(Y,d)}{\partial y_i} = \frac{\partial Div(Y,d)}{\partial y_i^{(N)}}$$

Computing derivatives



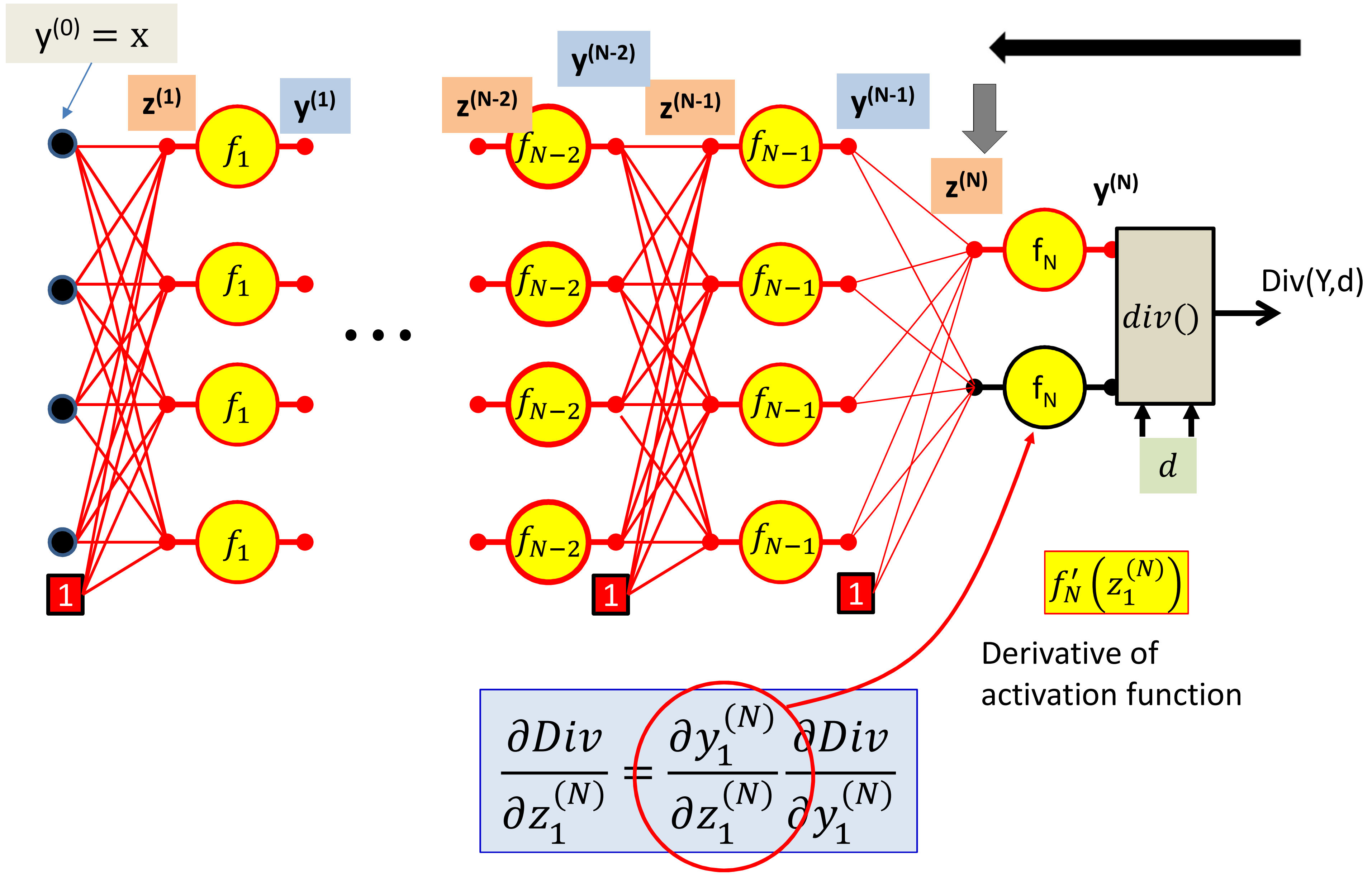
$$\frac{\partial Div}{\partial z_1^{(N)}} = \frac{\partial y_1^{(N)}}{\partial z_1^{(N)}} \frac{\partial Div}{\partial y_1^{(N)}}$$

Computing derivatives



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

Computing derivatives

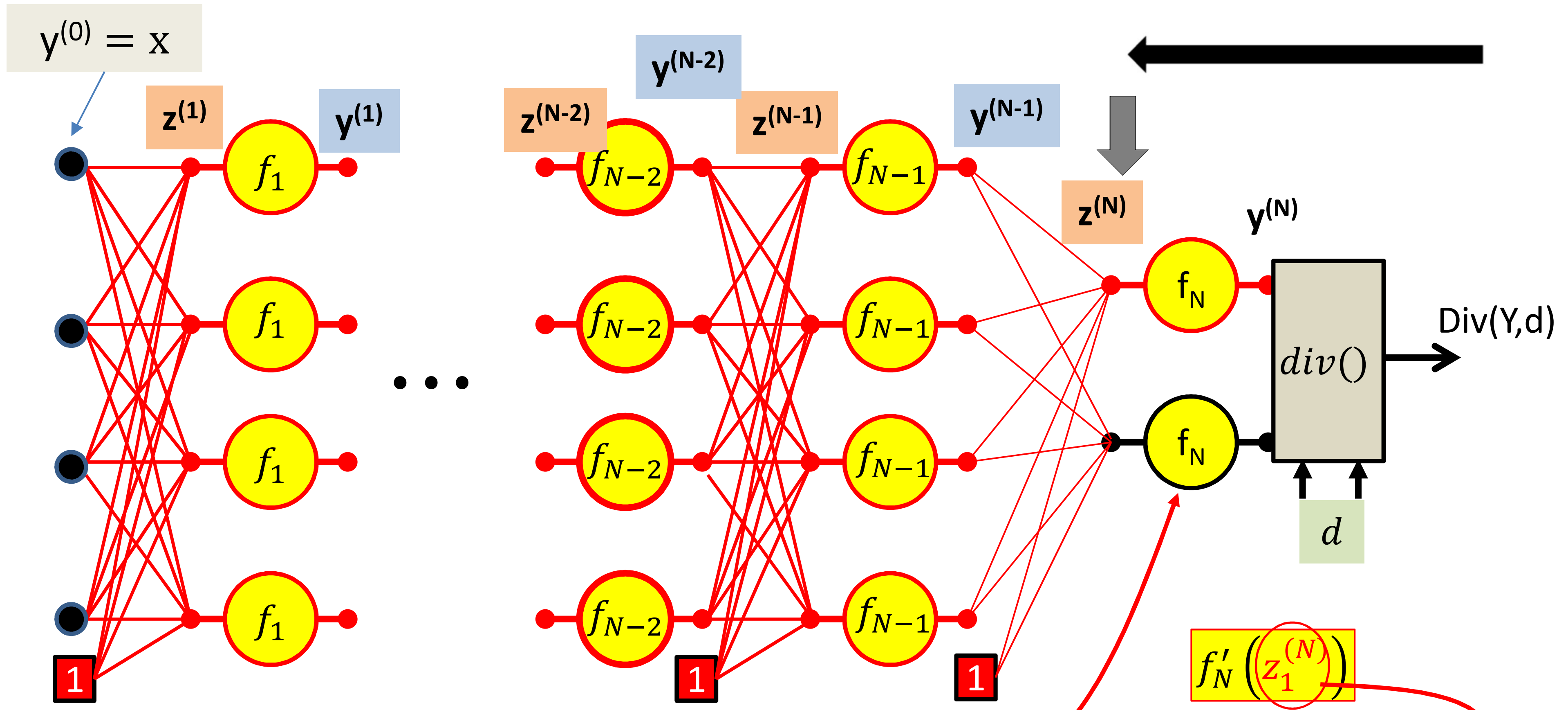


$f_N'\left(z_1^{(N)}\right)$

Derivative of activation function

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

Computing derivatives

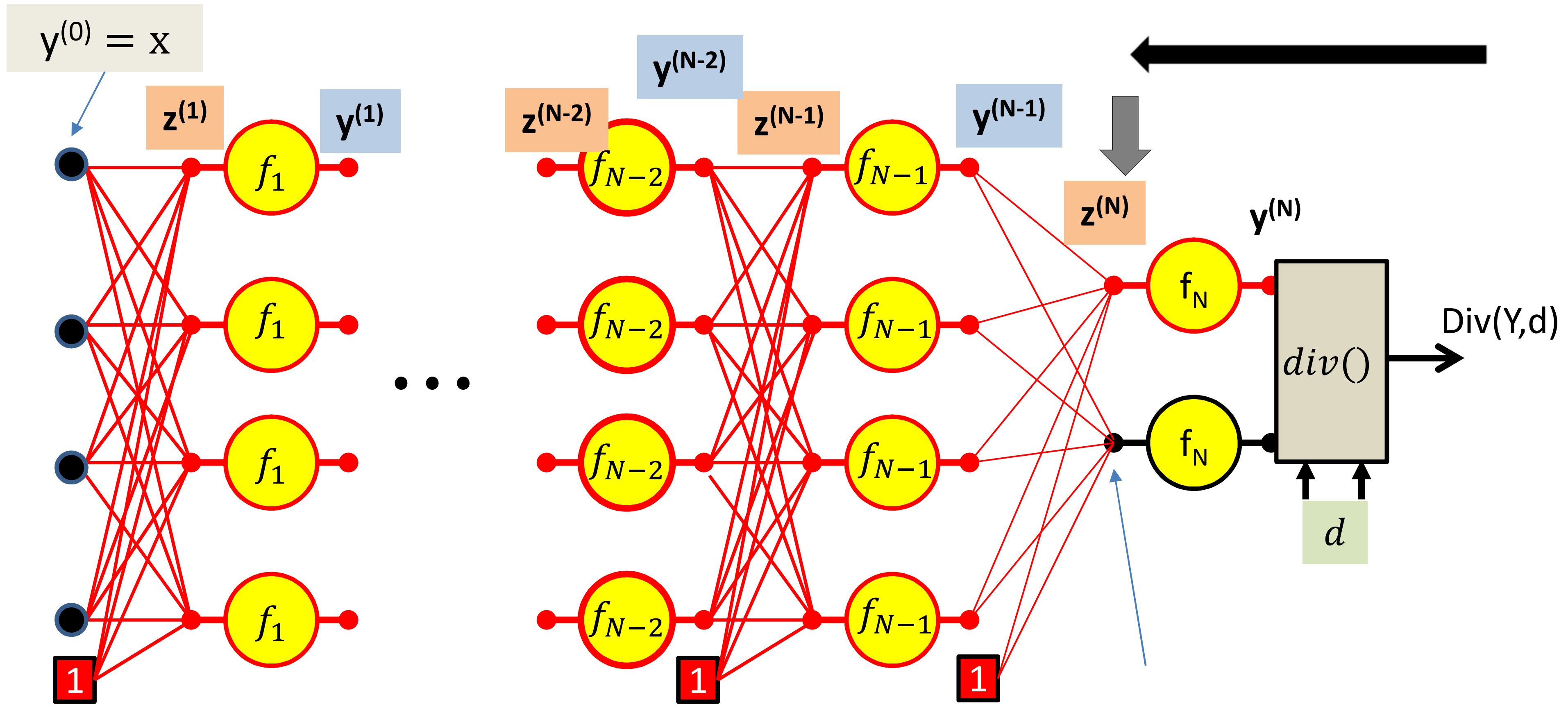


$$\frac{\partial Div}{\partial z_1^{(N)}} = \frac{\partial y_1^{(N)}}{\partial z_1^{(N)}} \frac{\partial Div}{\partial y_1^{(N)}}$$

Derivative of activation function

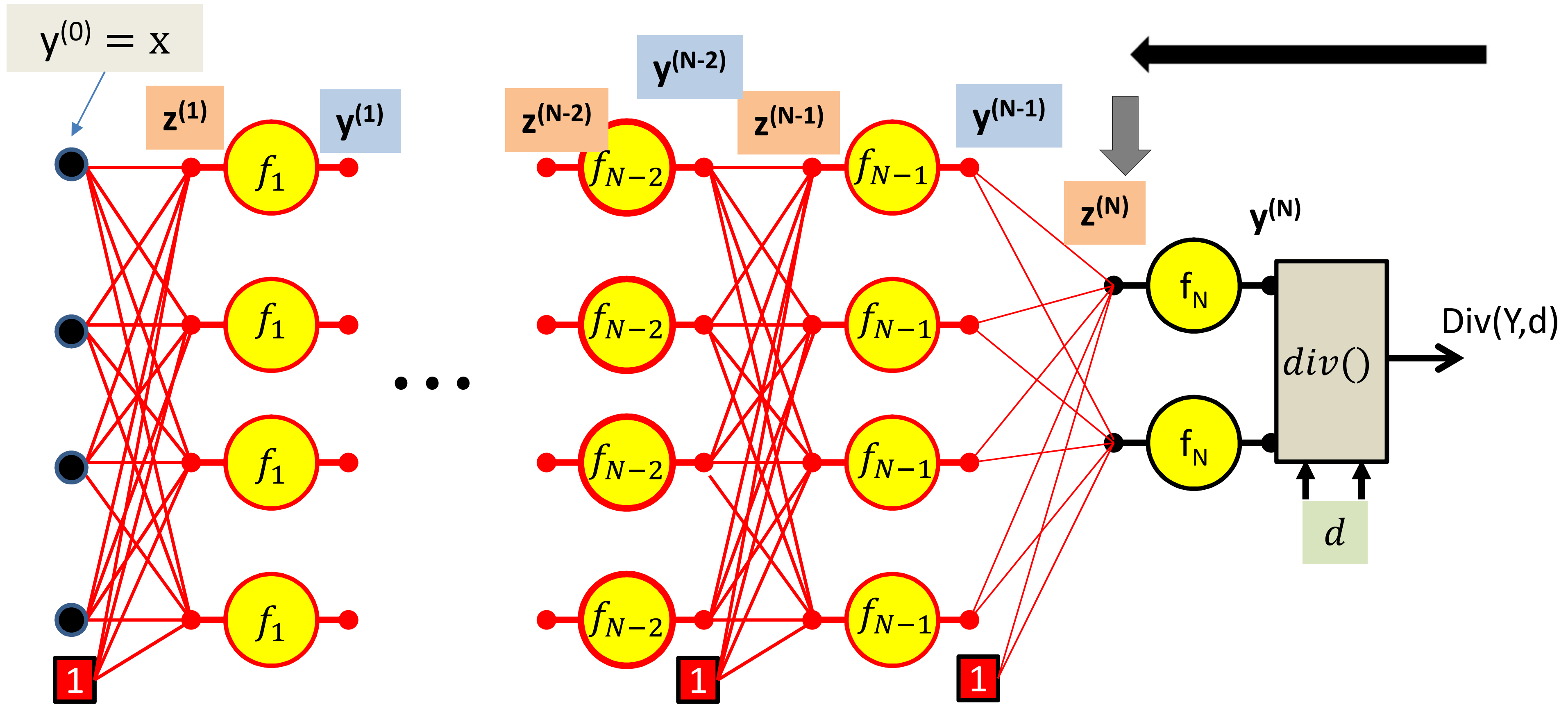
Computed in forward pass

Computing derivatives



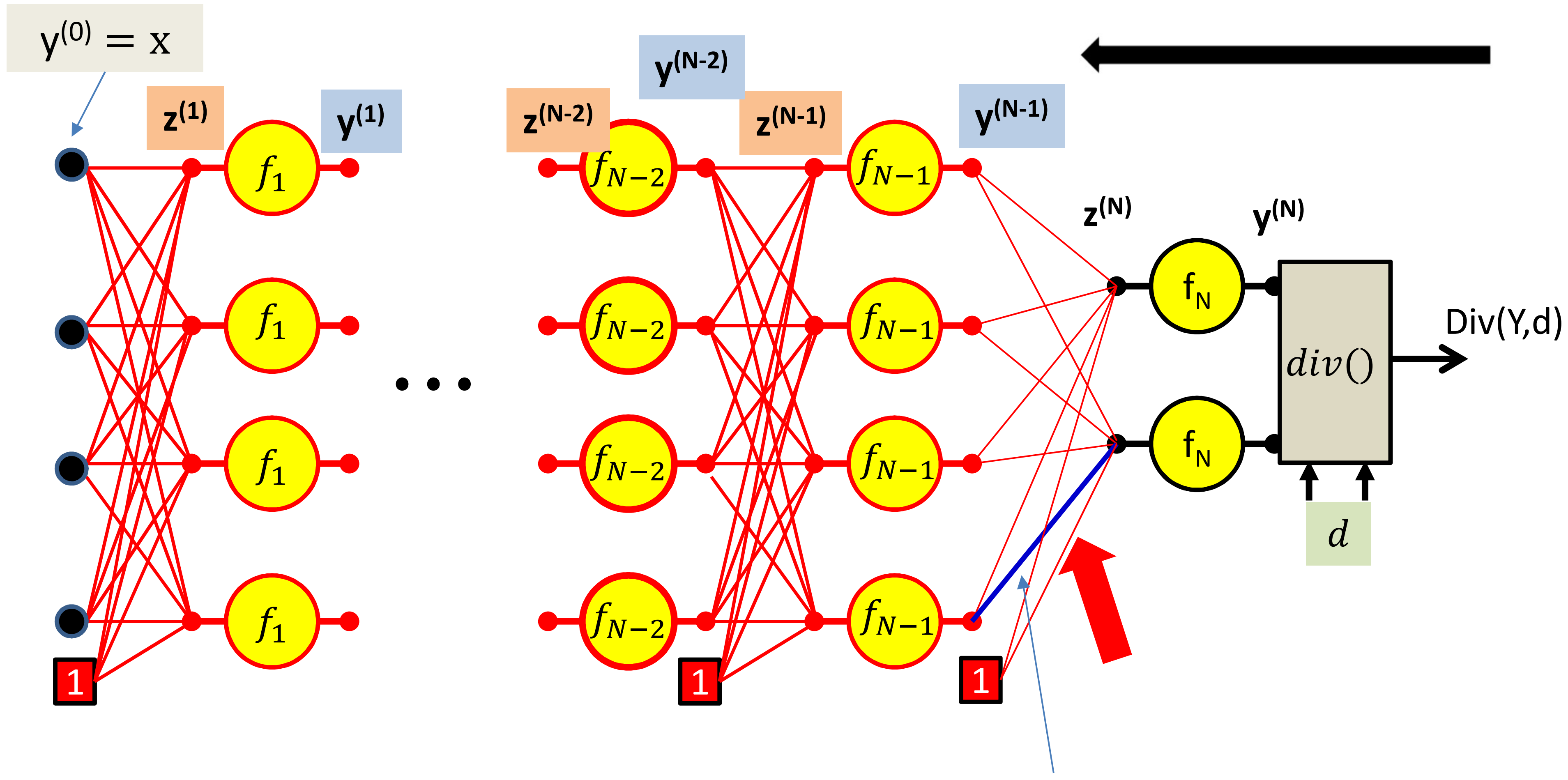
$$\frac{\partial Div}{\partial z_1^{(N)}} = f_N'\left(z_1^{(N)}\right)\frac{\partial Div}{\partial y_1^{(N)}}$$

Computing derivatives



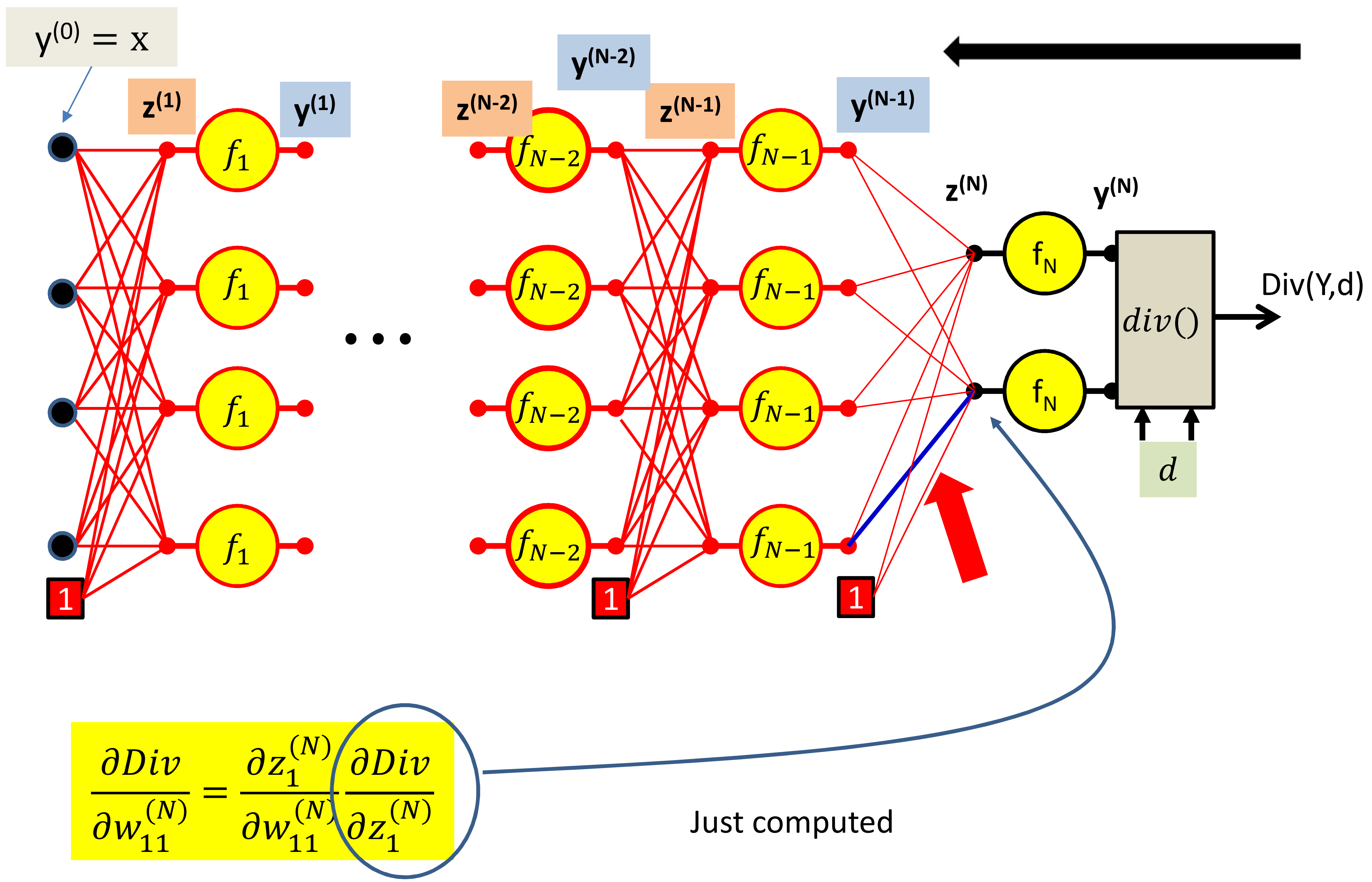
$$\frac{\partial Div}{\partial z_i^{(N)}} = f_N'\left(z_i^{(N)}\right)\frac{\partial Div}{\partial y_i^{(N)}}$$

Computing derivatives



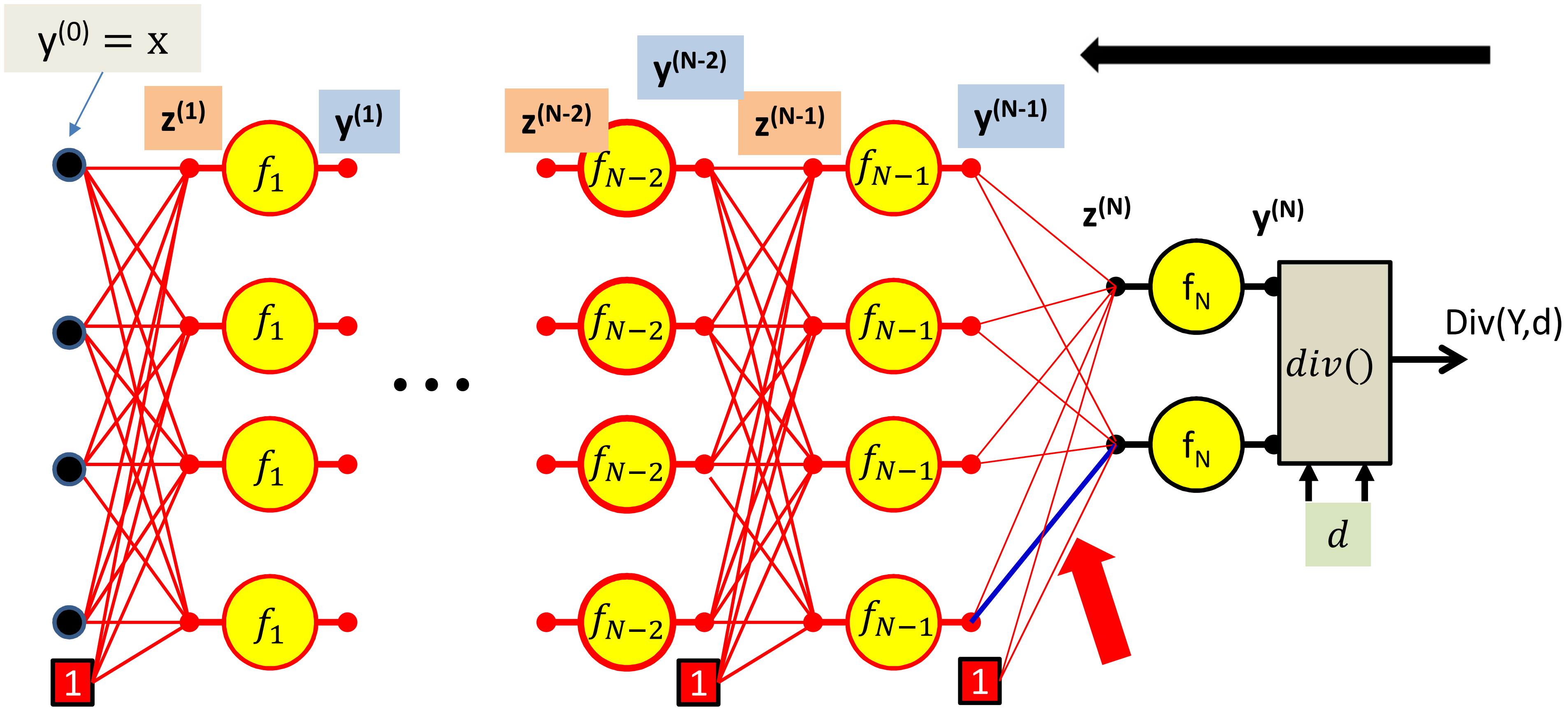
$$\frac{\partial Div}{\partial w_{11}^{(N)}} = \frac{\partial z_1^{(N)}}{\partial w_{11}^{(N)}} \frac{\partial Div}{\partial z_1^{(N)}}$$

Computing derivatives



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

Computing derivatives



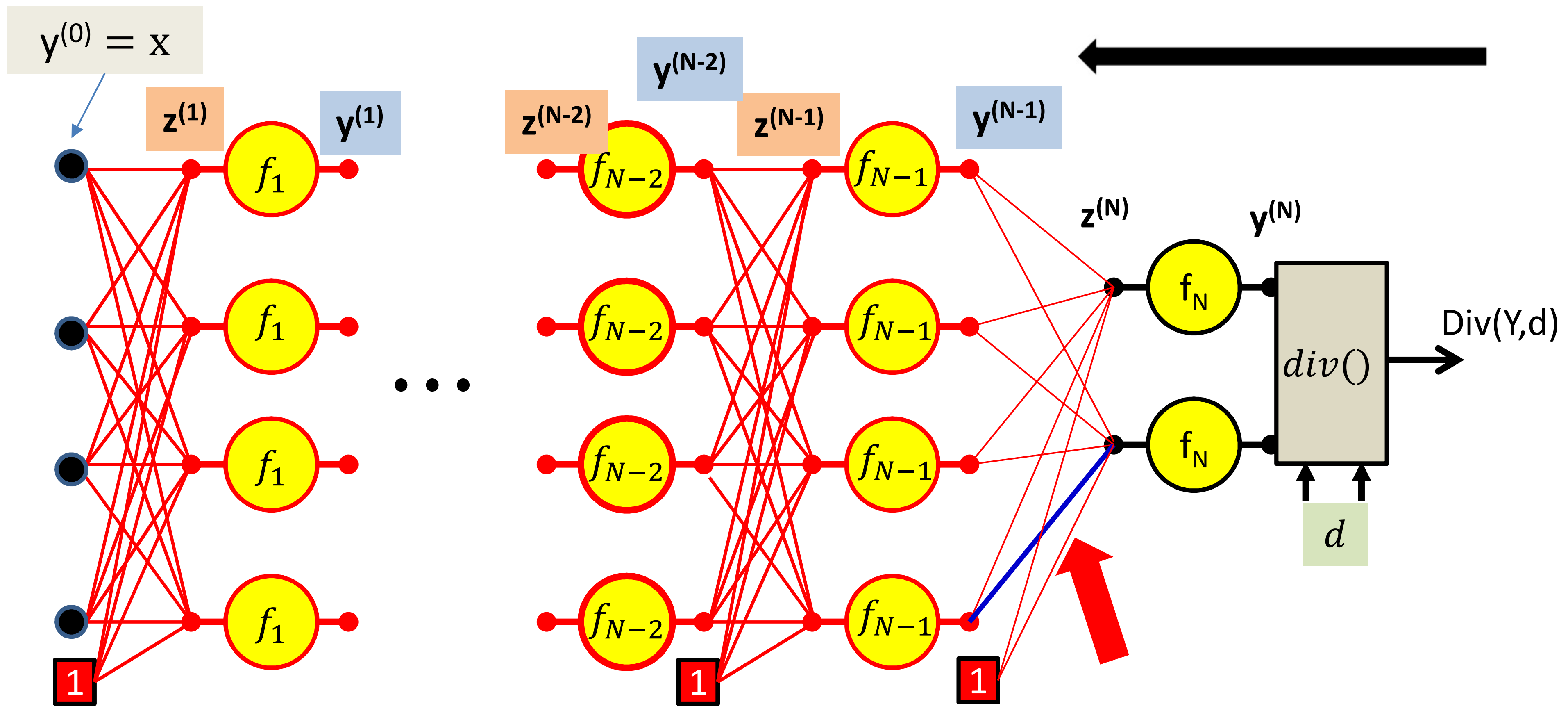
$$\frac{\partial Div}{\partial w_{11}^{(N)}} = \frac{\partial z_1^{(N)}}{\partial w_{11}^{(N)}} \frac{\partial Div}{\partial z_1^{(N)}}$$

$y_1^{(N-1)}$

Because

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

Computing derivatives



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

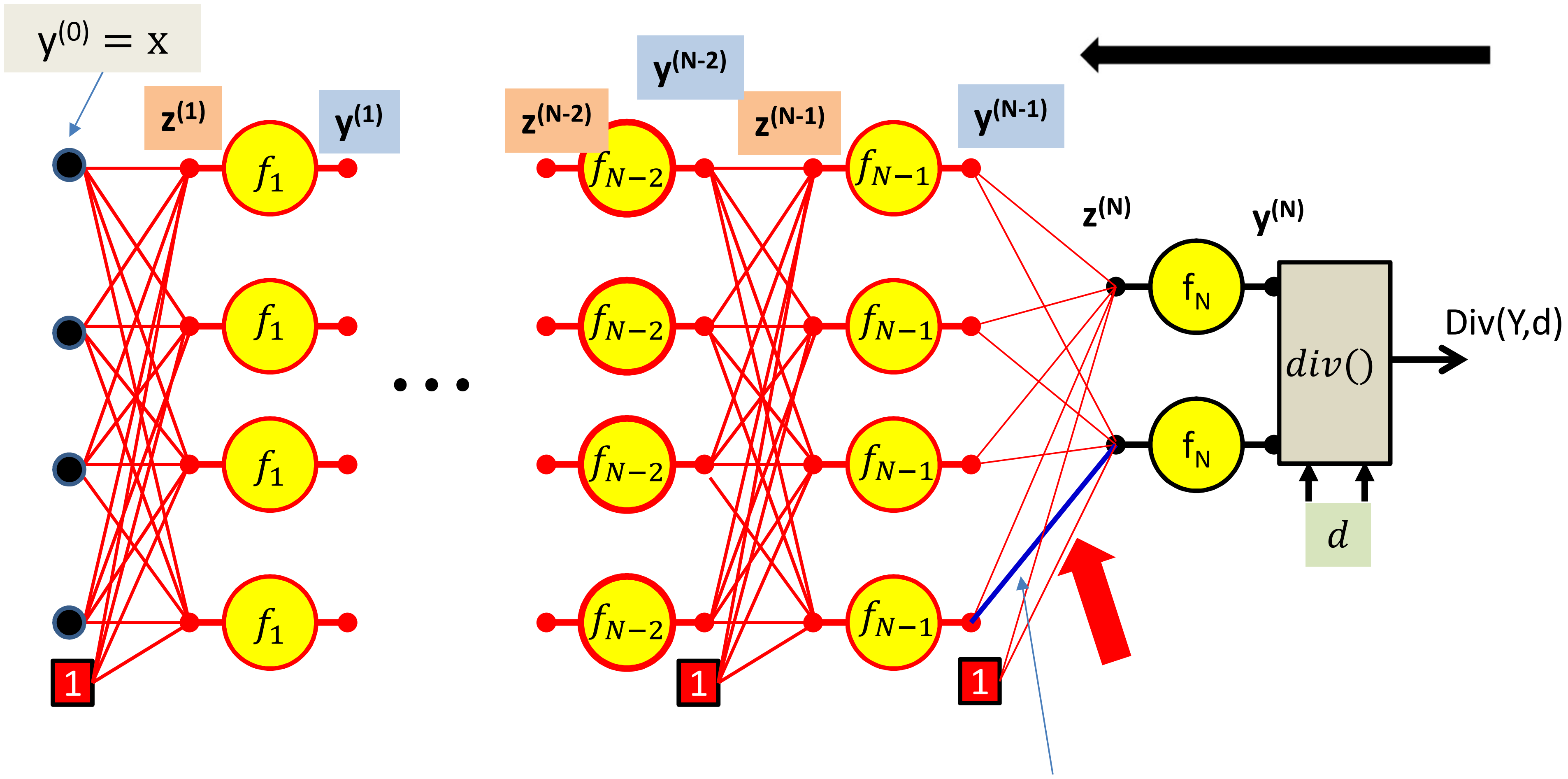
$\boxed{y_1^{(N-1)}}$

Because

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

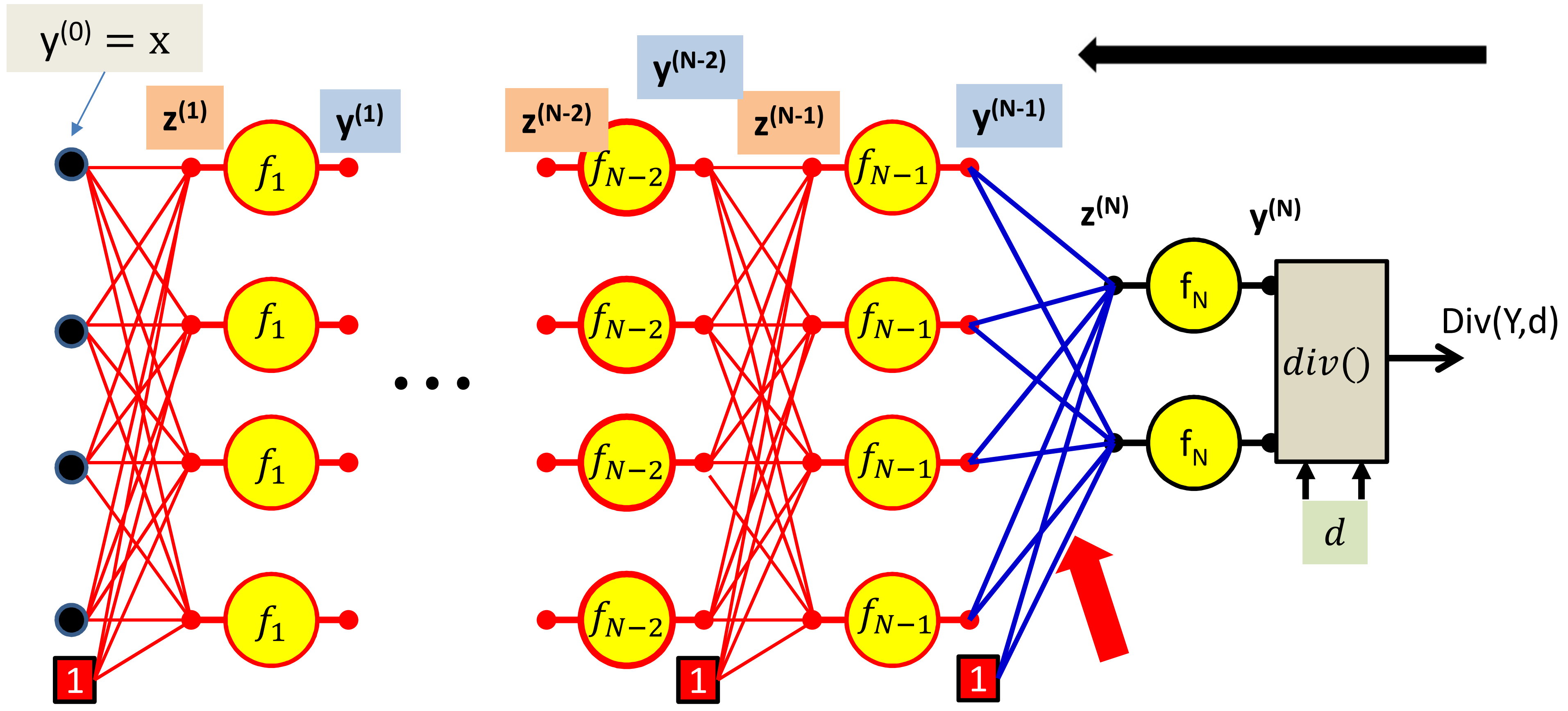
Computed in forward pass

Computing derivatives



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

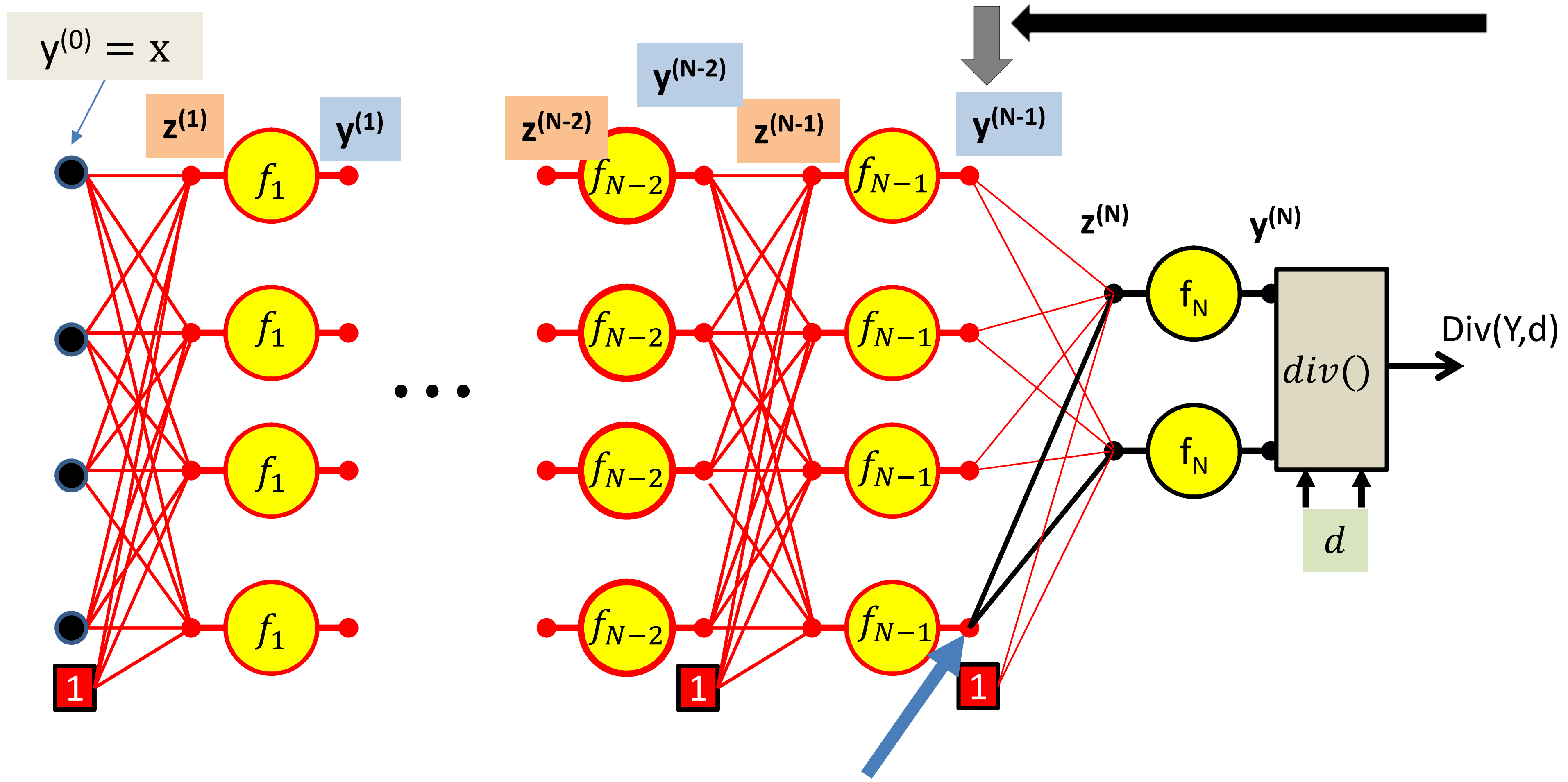
Computing derivatives



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

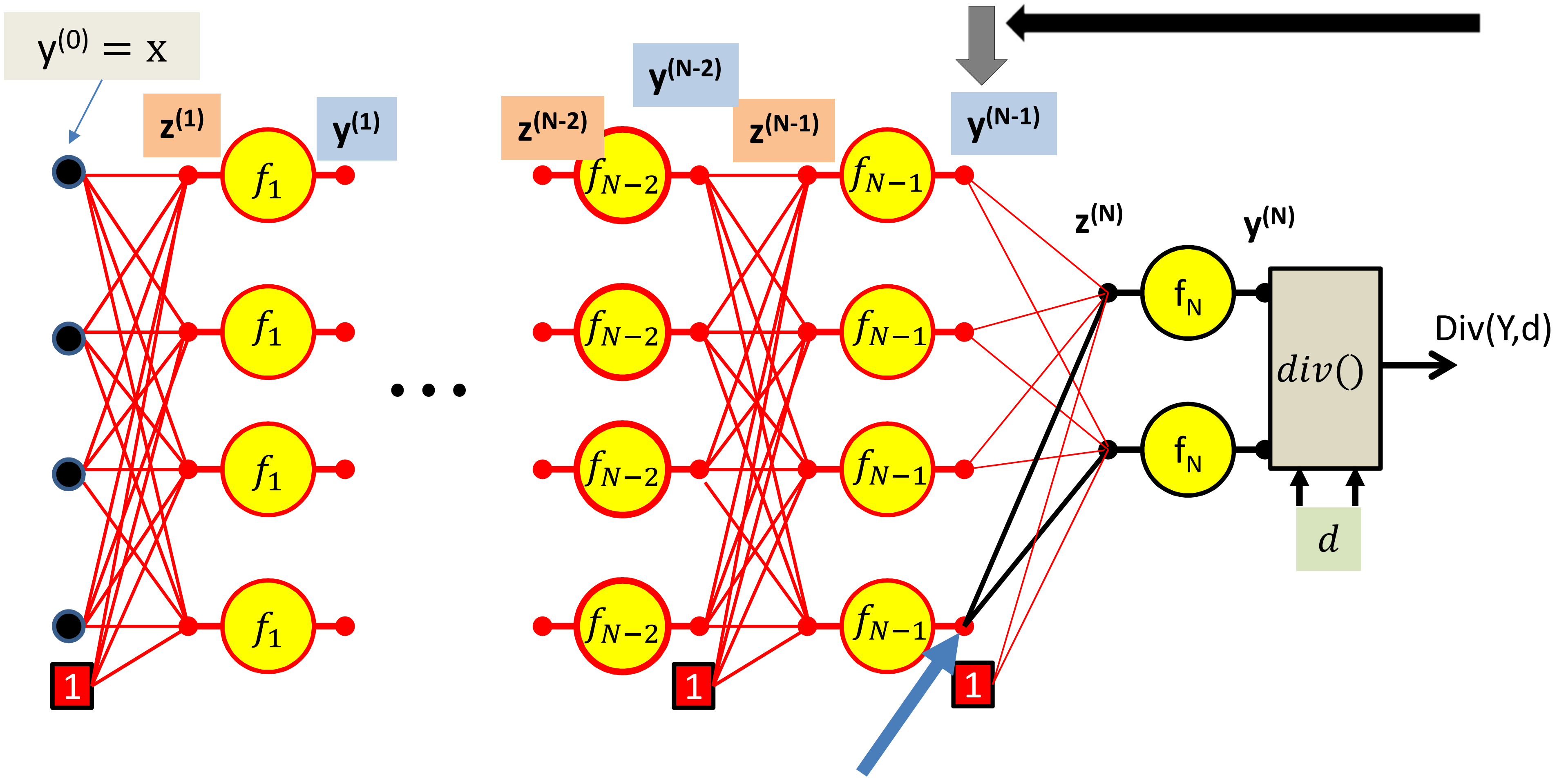
For the bias term $y_0^{(N-1)} = 1$

Computing derivatives



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

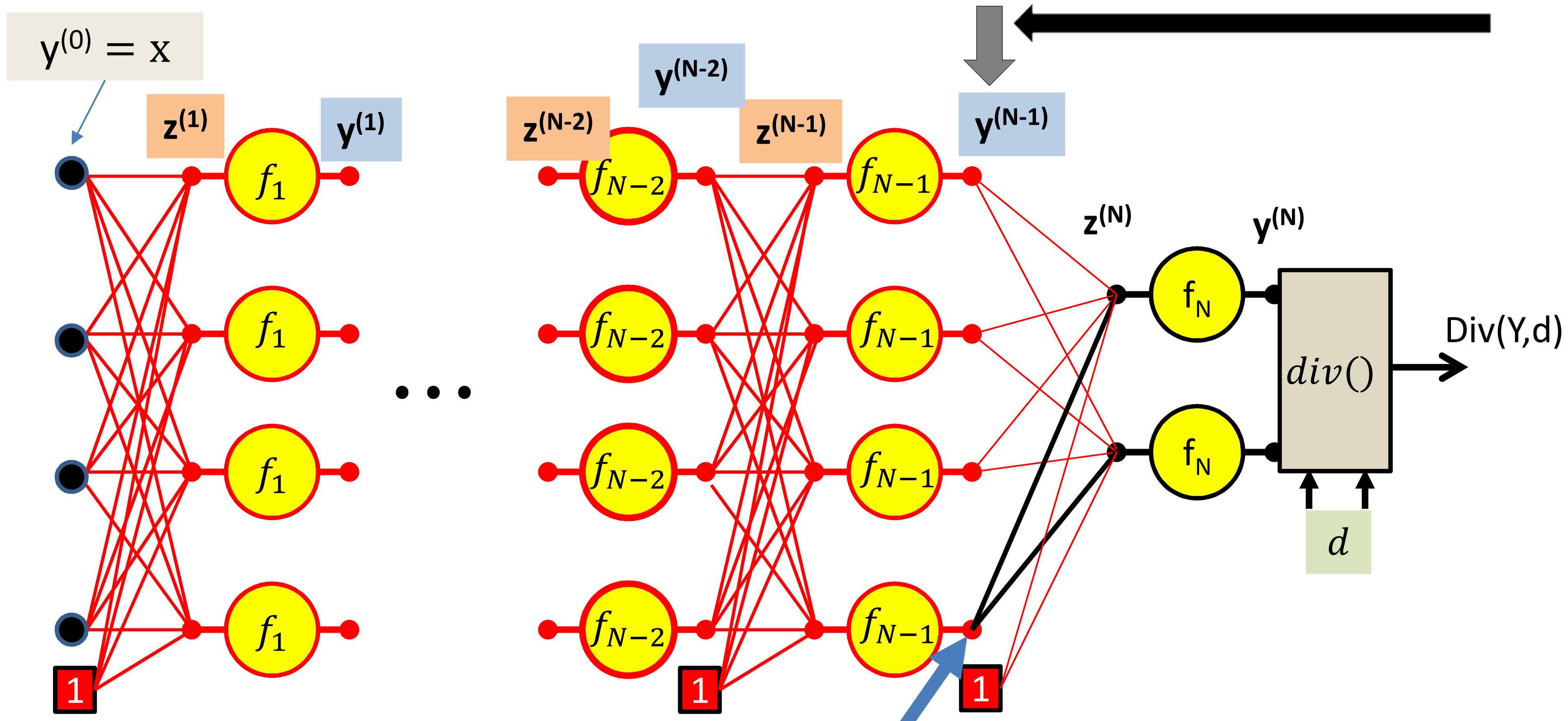
Computing derivatives



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

Already computed

Computing derivatives



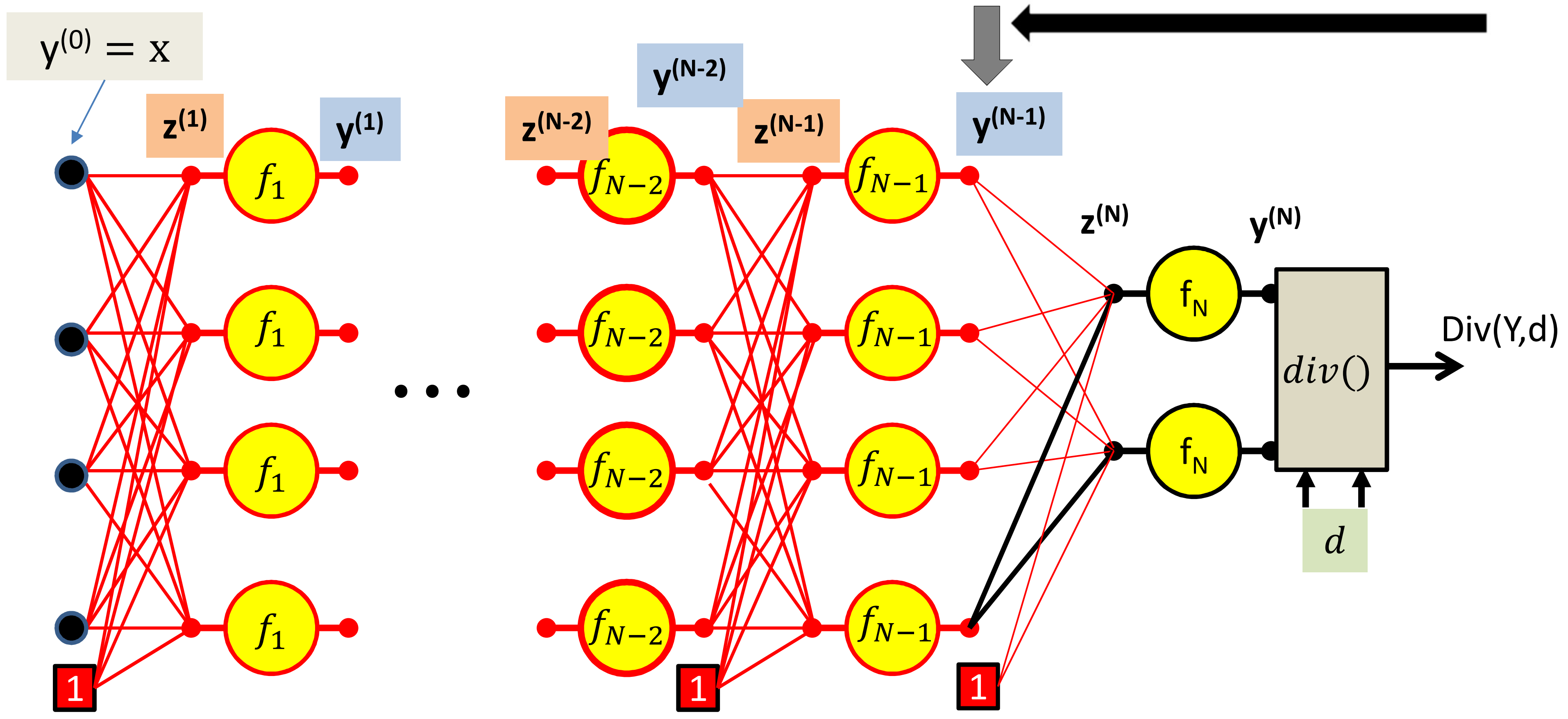
$$\frac{\partial Div}{\partial y_1^{(N-1)}} = \sum_j \frac{\partial z_j^{(N)}}{\partial y_1^{(N-1)}} \frac{\partial Div}{\partial z_j^{(N)}}$$

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

Because

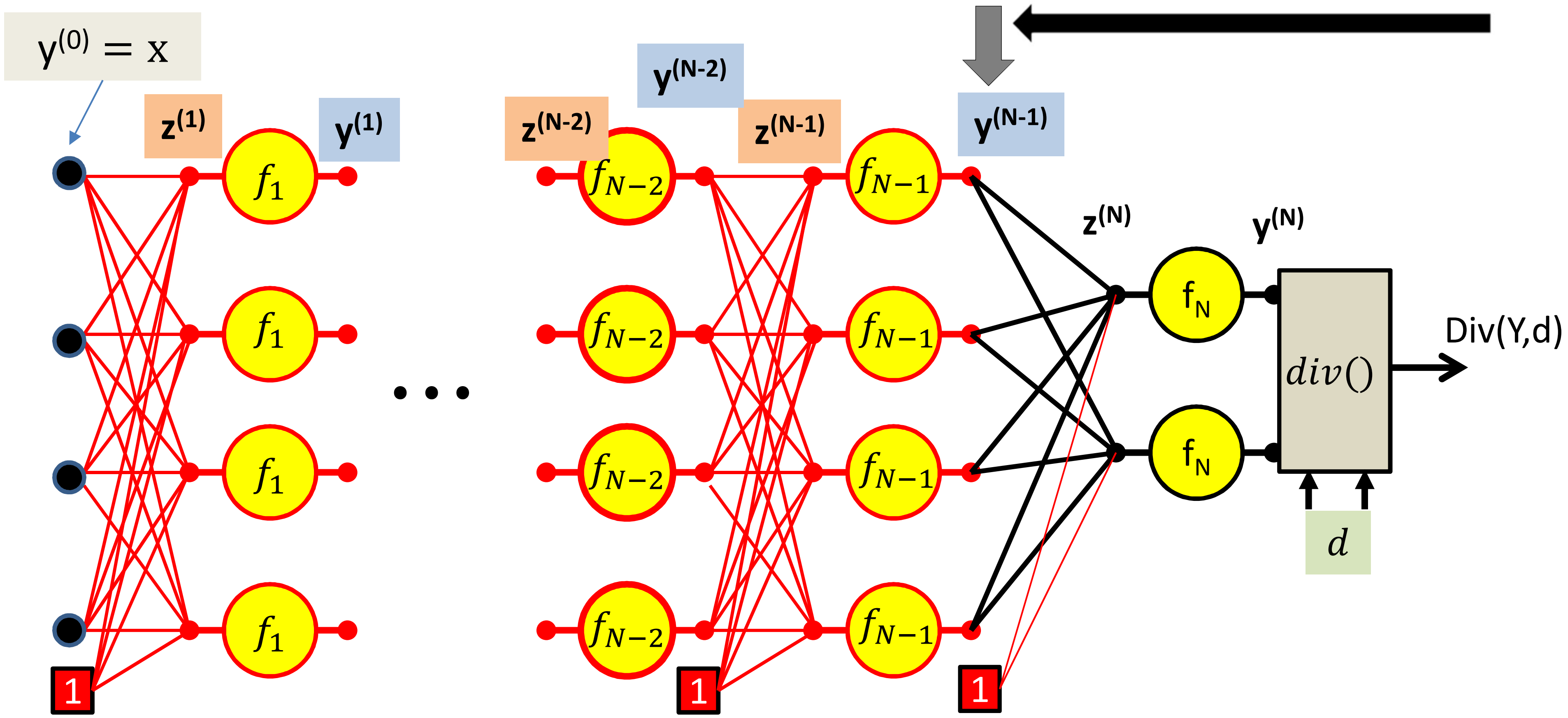
$$z_j^{(N)} = w_{1j}^{(N)} y_1^{(N-1)} + \text{other terms}$$

Computing derivatives



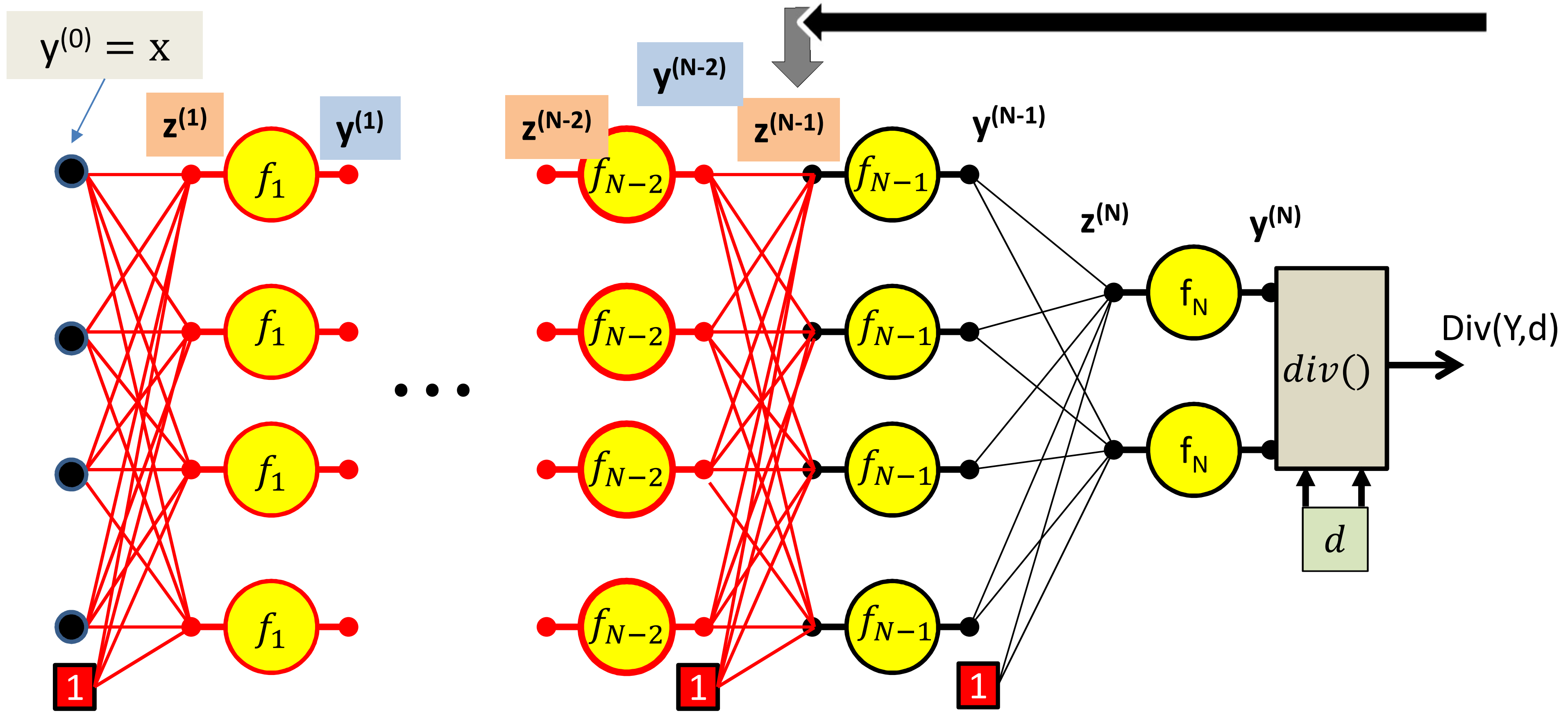
$$\frac{\partial Div}{\partial y_1^{(N-1)}} = \sum_j w_{1j}^{(N)} \frac{\partial Div}{\partial z_j^{(N)}}$$

Computing derivatives



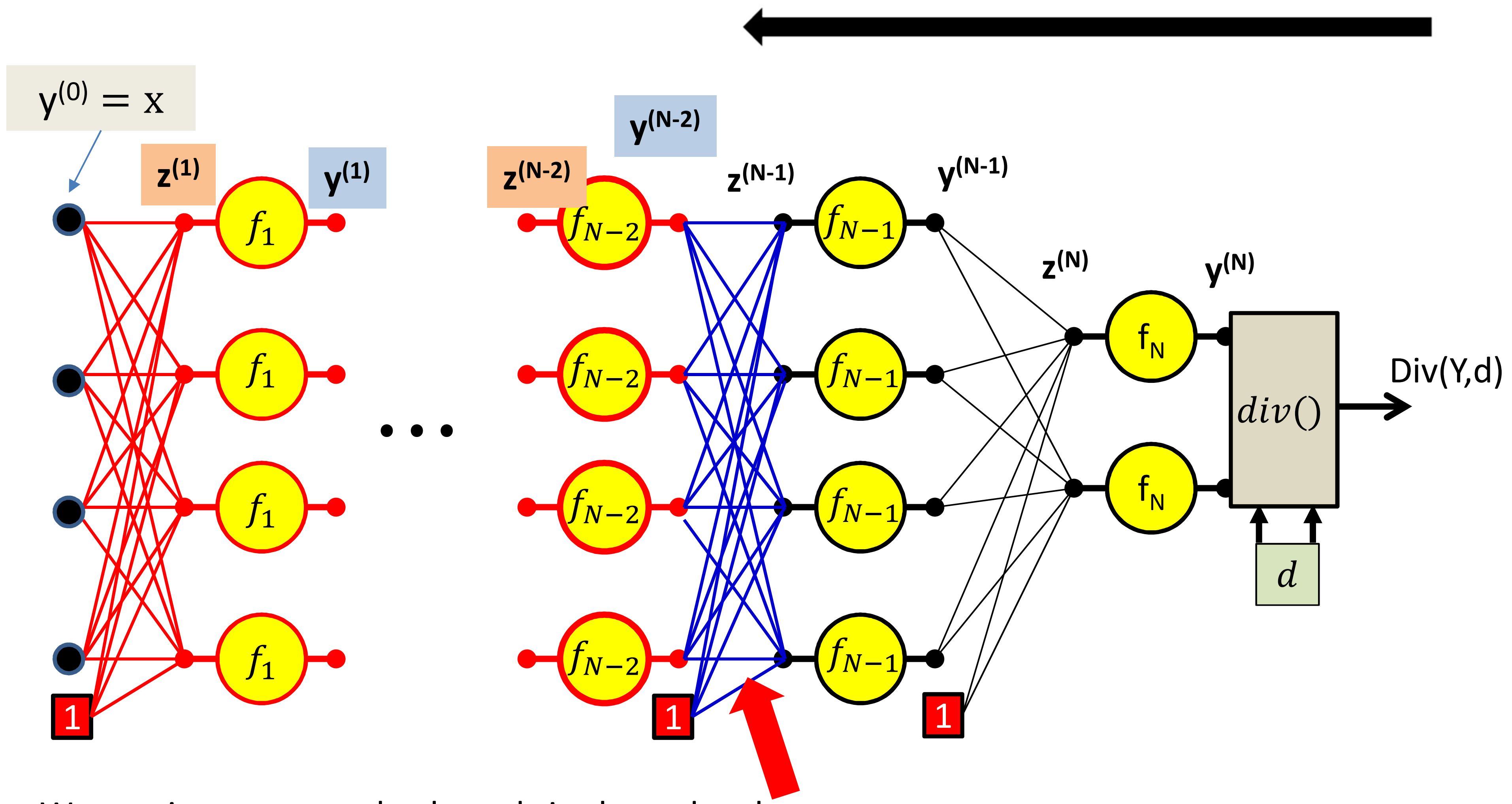
$$\frac{\partial Div}{\partial y_i^{(N-1)}} = \sum_j w_{ij}^{(N)} \frac{\partial Div}{\partial z_j^{(N)}}$$

Computing derivatives



We continue our way backwards in the order shown

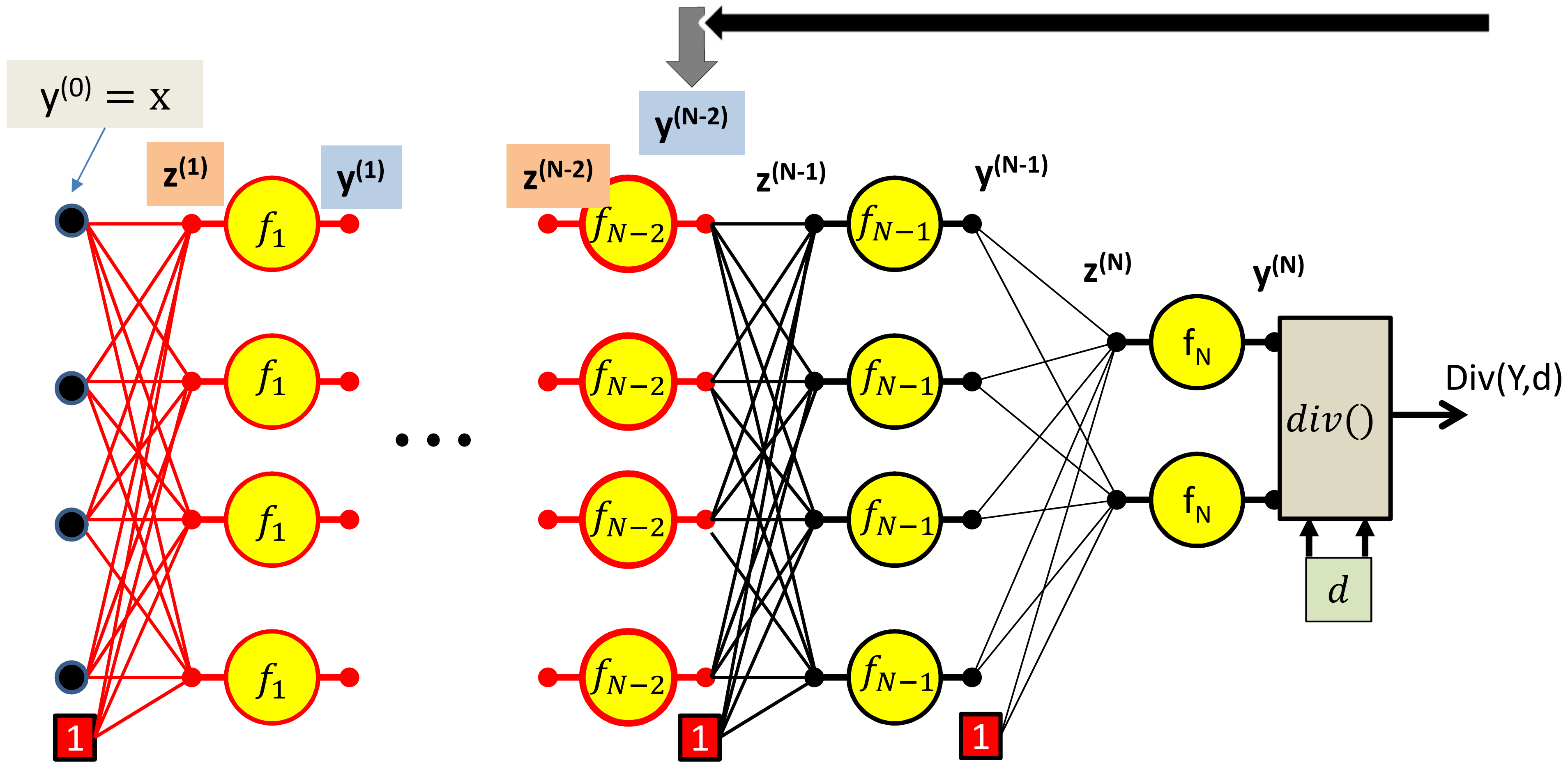
$$\frac{\partial Div}{\partial z_i^{(N-1)}} = f'_{N-1}\left(z_i^{(N-1)}\right)\frac{\partial Div}{\partial y_i^{(N-1)}}$$



We continue our way backwards in the order shown

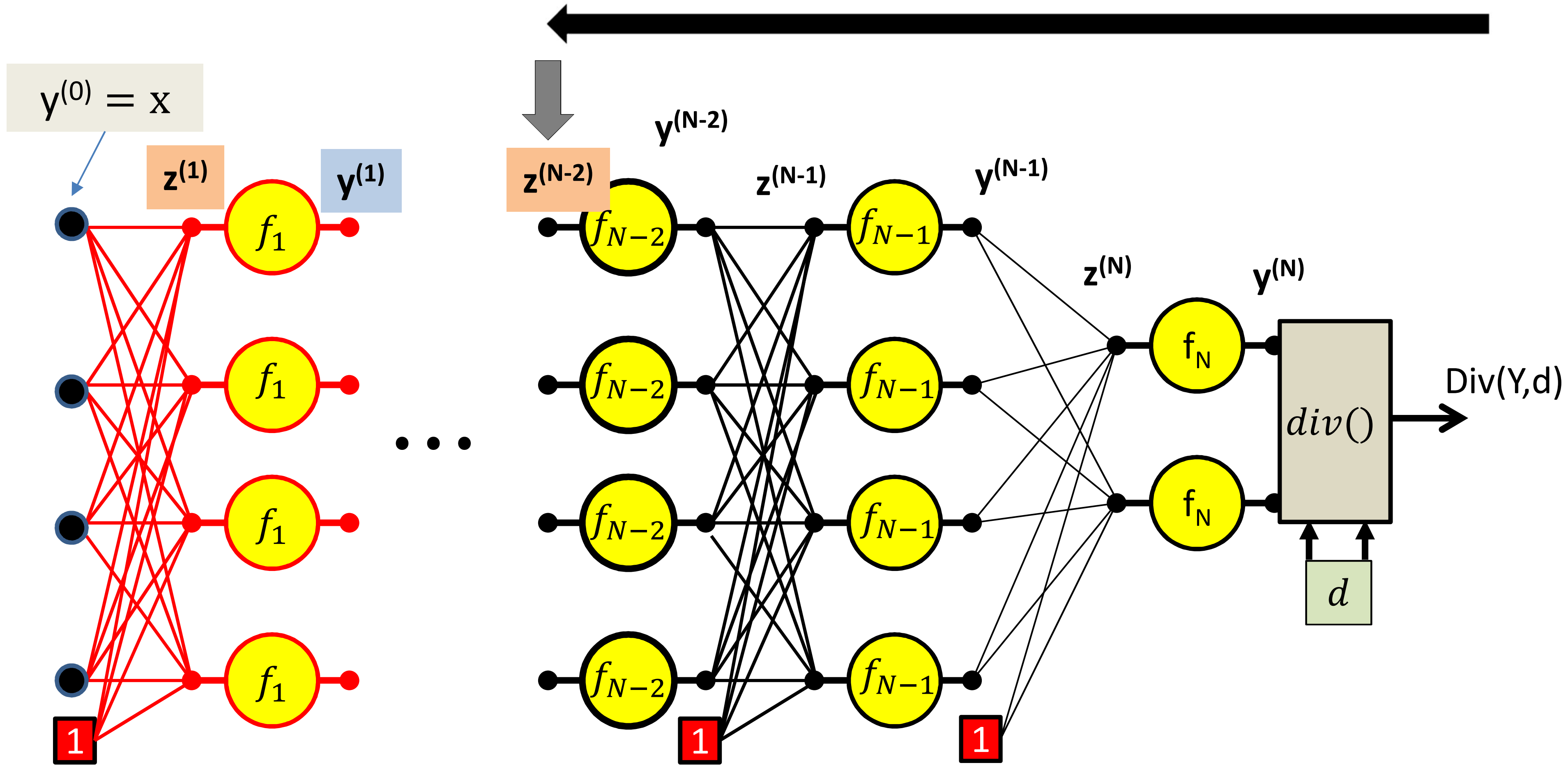
$$\frac{\partial Div}{\partial w_{ij}^{(N-1)}} = y_i^{(N-2)} \frac{\partial Div}{\partial z_j^{(N-1)}}$$

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



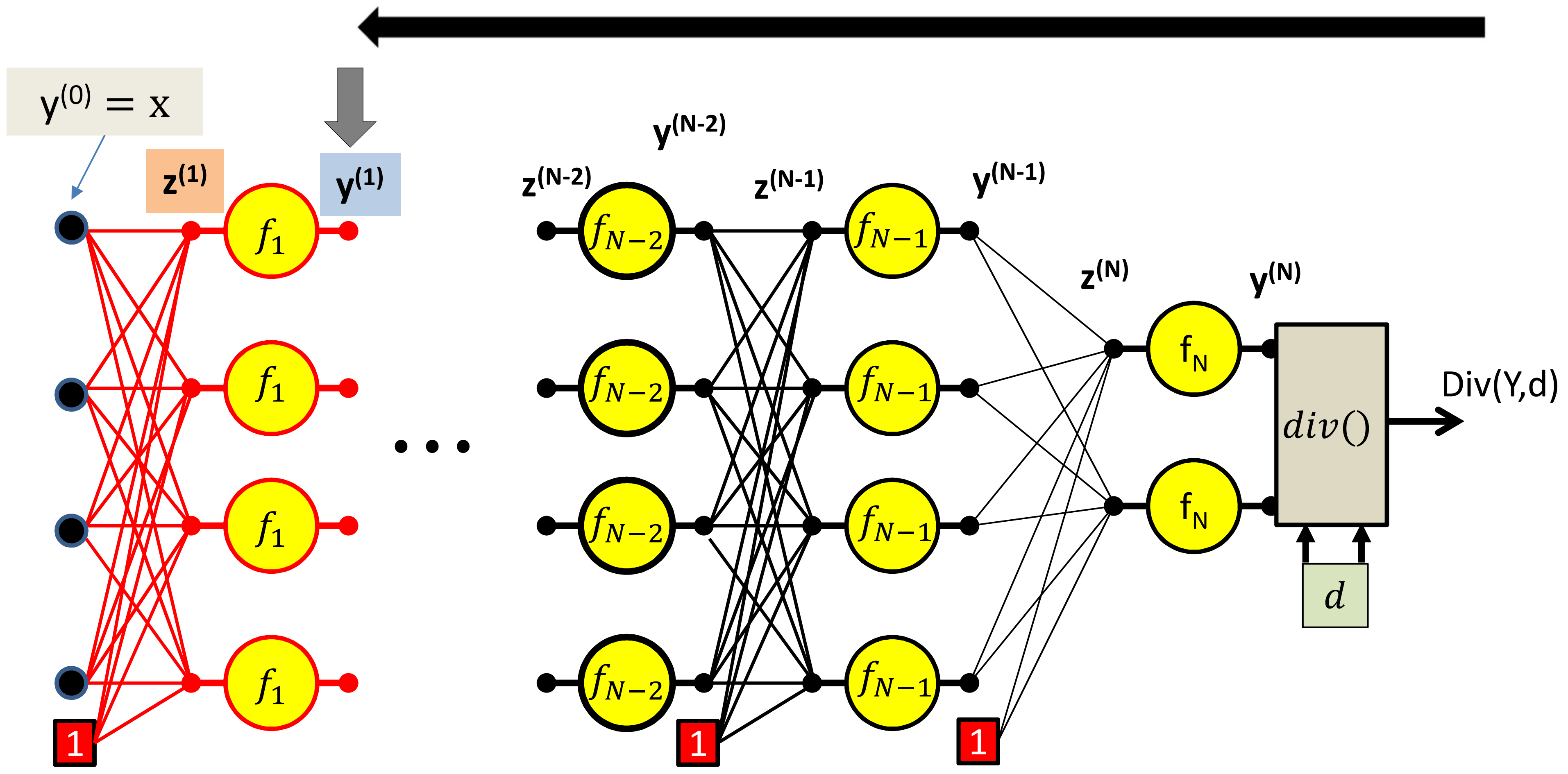
We continue our way backwards in the order shown

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



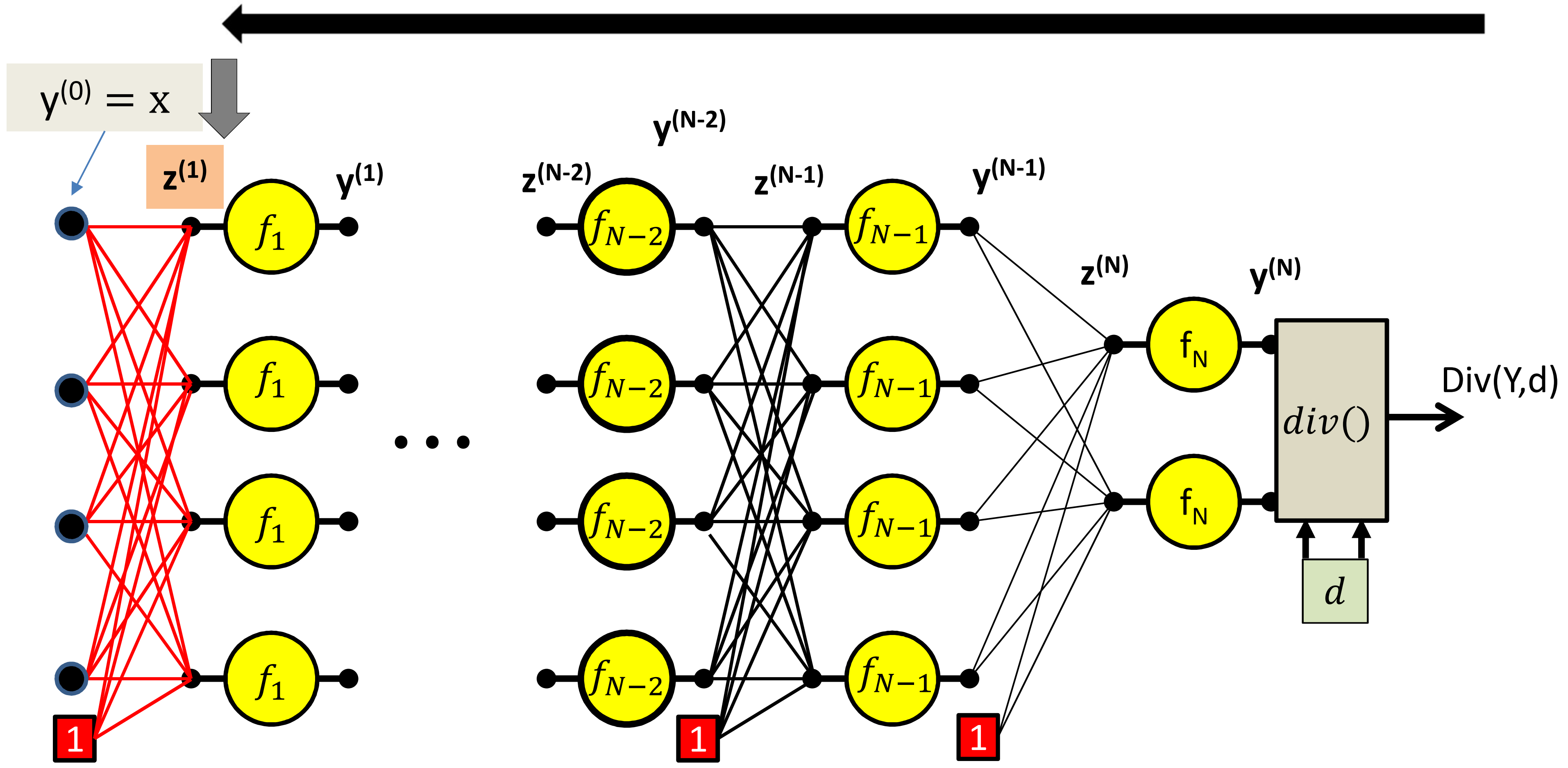
We continue our way backwards in the order shown

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



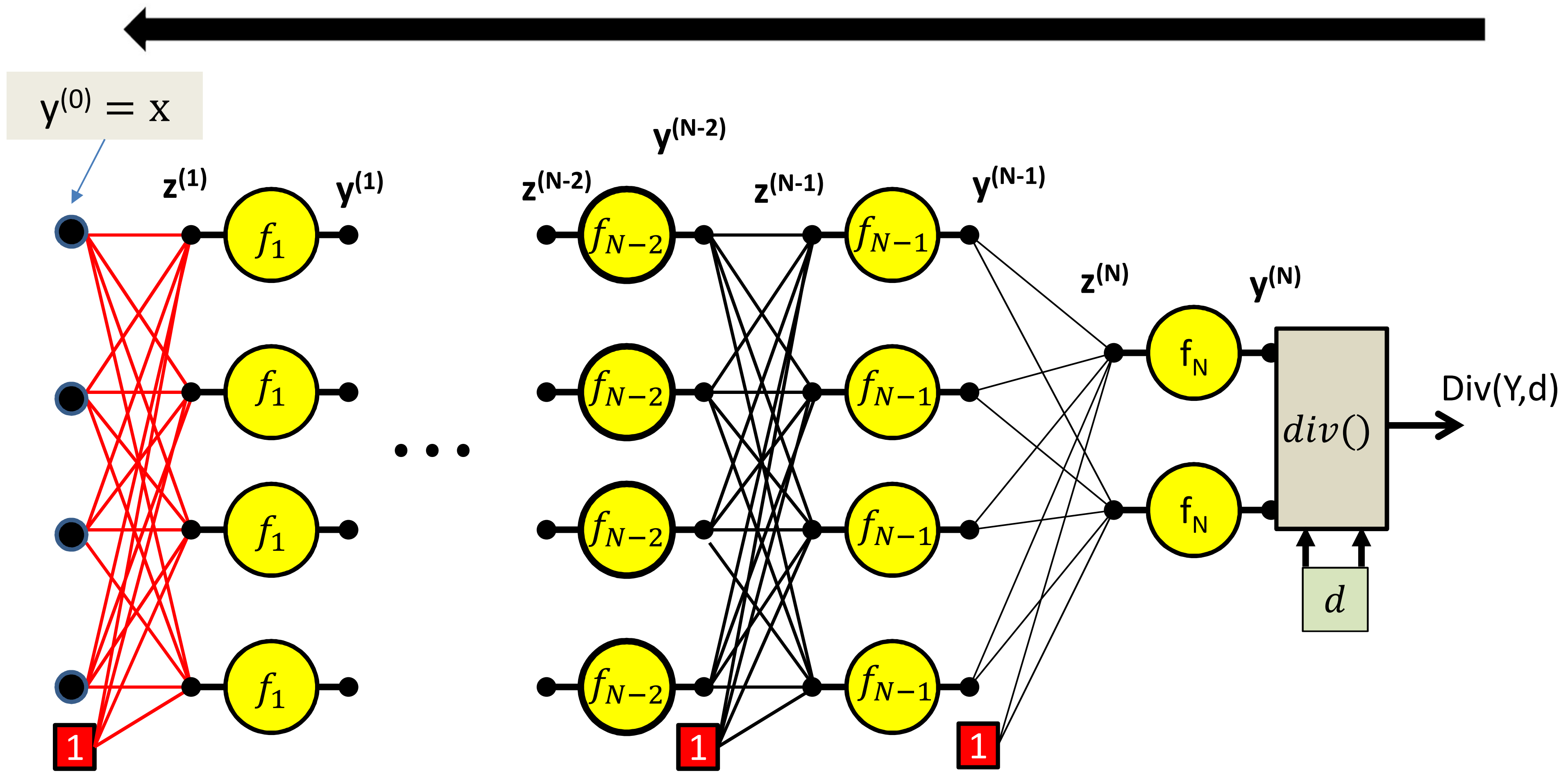
We continue our way backwards in the order shown

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



We continue our way backwards in the order shown

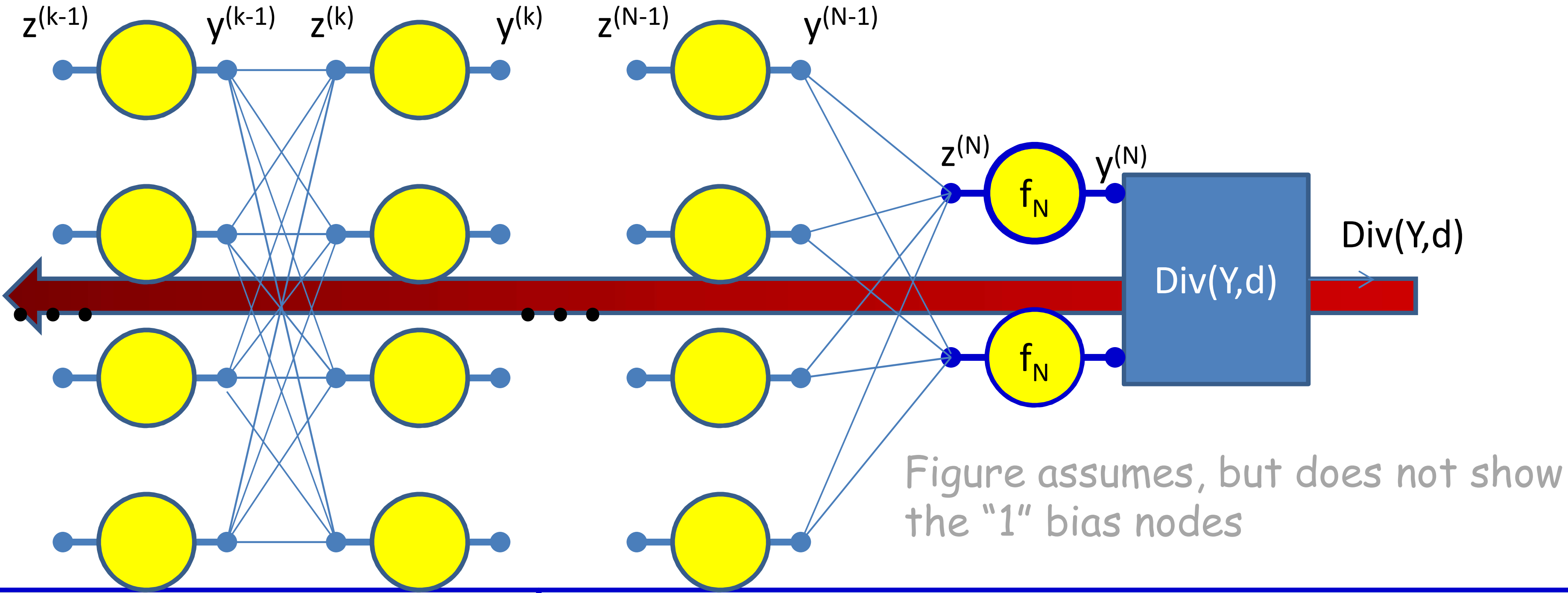
$$\frac{\partial Div}{\partial z_i^{(1)}} = f_1'\left(z_i^{(1)}\right)\frac{\partial Div}{\partial y_i^{(1)}}$$



We continue our way backwards in the order shown

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

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 $i = 1: layer\ width$
```

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

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

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

Backward Pass

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

Backward Pass

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

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

Very analogous to the forward pass:

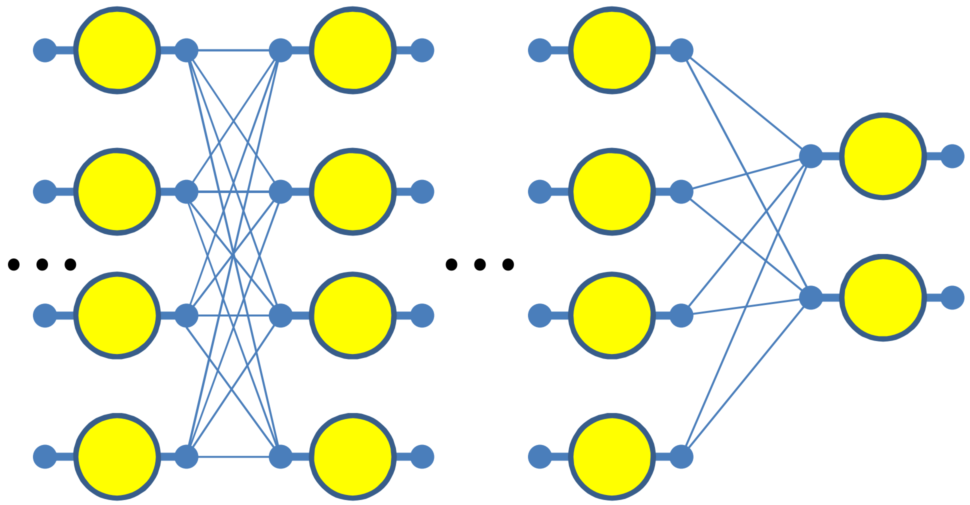
Backward weighted combination of next layer

Backward equivalent of activation

For comparison: the forward pass again

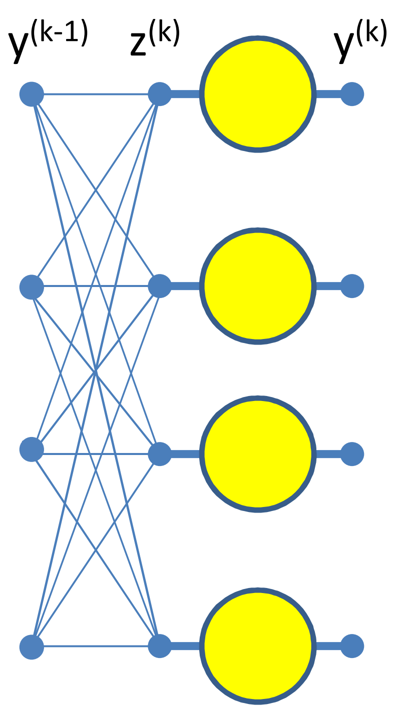
- Input: D dimensional vector $\mathbf{x} = [x_j,\ j = 1 \dots D]$
- Set:
  - $D_0 = D,$ is the width of the 0th (input) layer
  - $y_j^{(0)} = x_j,\ j = 1 \dots D; \qquad y_0^{(k=1\dots N)} = x_0 = 1$
- For layer $k = 1 \dots N$
  - For $j = 1 \dots D_k$
    - $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)}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

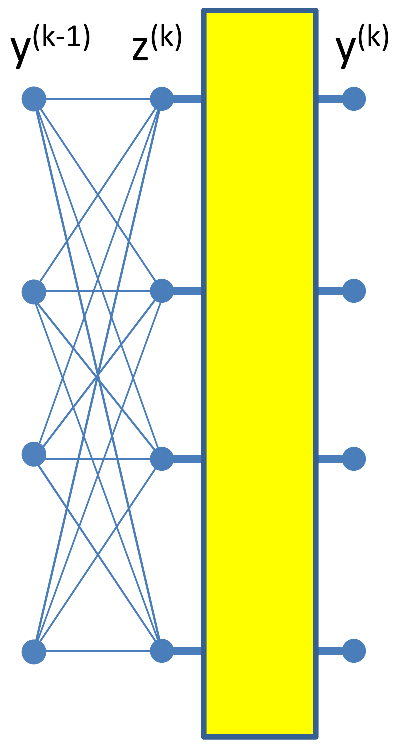
Special cases



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

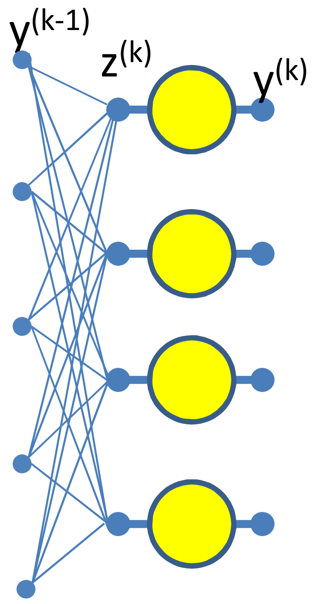
Special Case 1. Vector activations

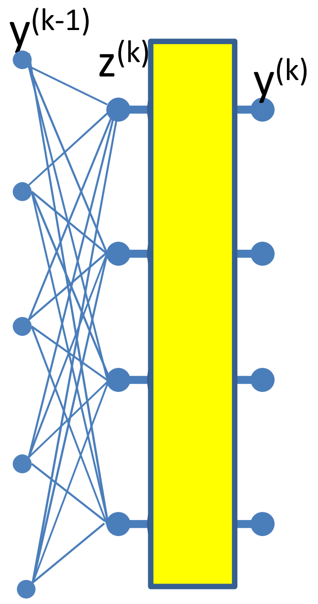




- Vector activations: all outputs are functions of all inputs

Special Case 1. Vector activations





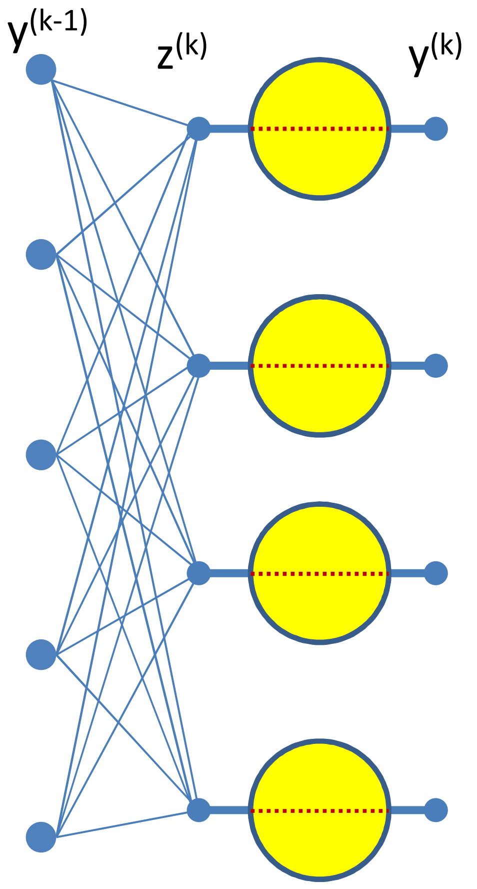
Scalar activation: Modifying a z_i only changes corresponding y_i

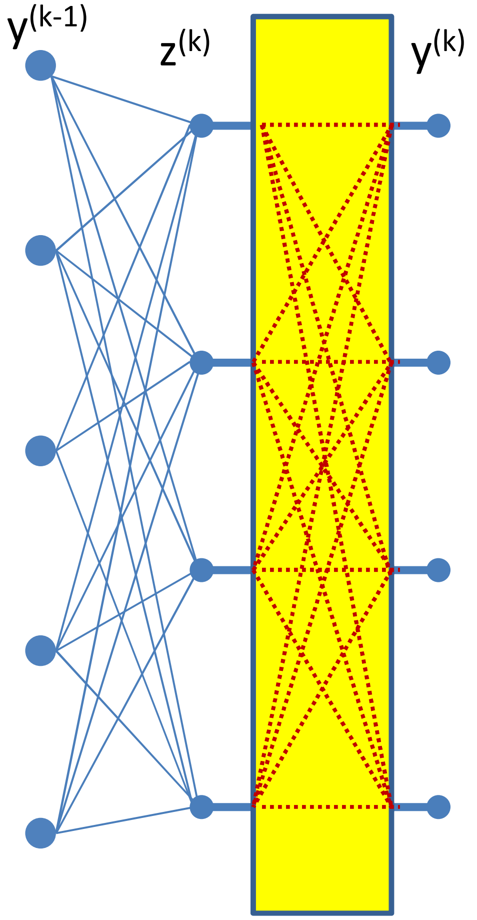
$$y_i^{(k)} = f\left(z_i^{(k)}\right)$$

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

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

"Influence" diagram

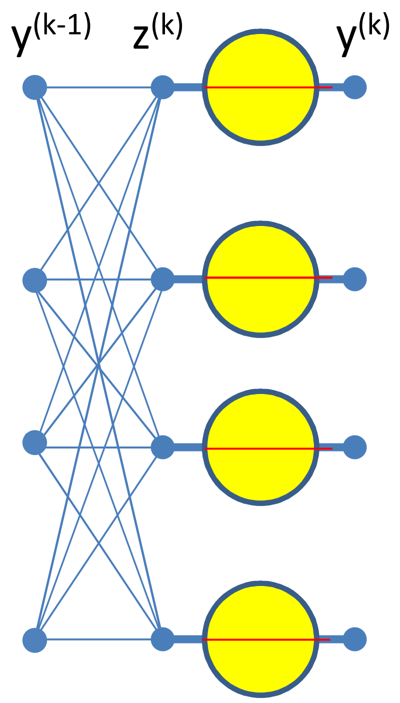




Scalar activation: Each z_i influences *one* y_i

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

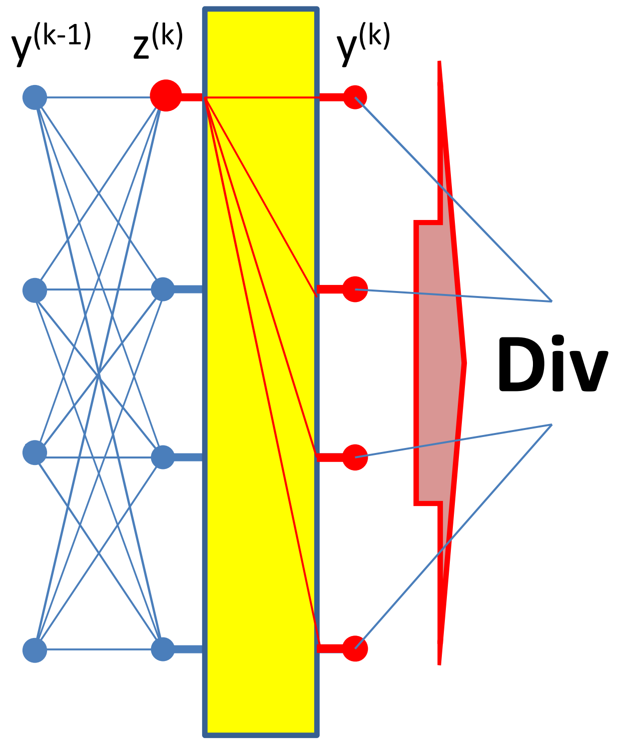
Scalar Activation: Derivative rule



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

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

Derivatives of vector activation



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

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

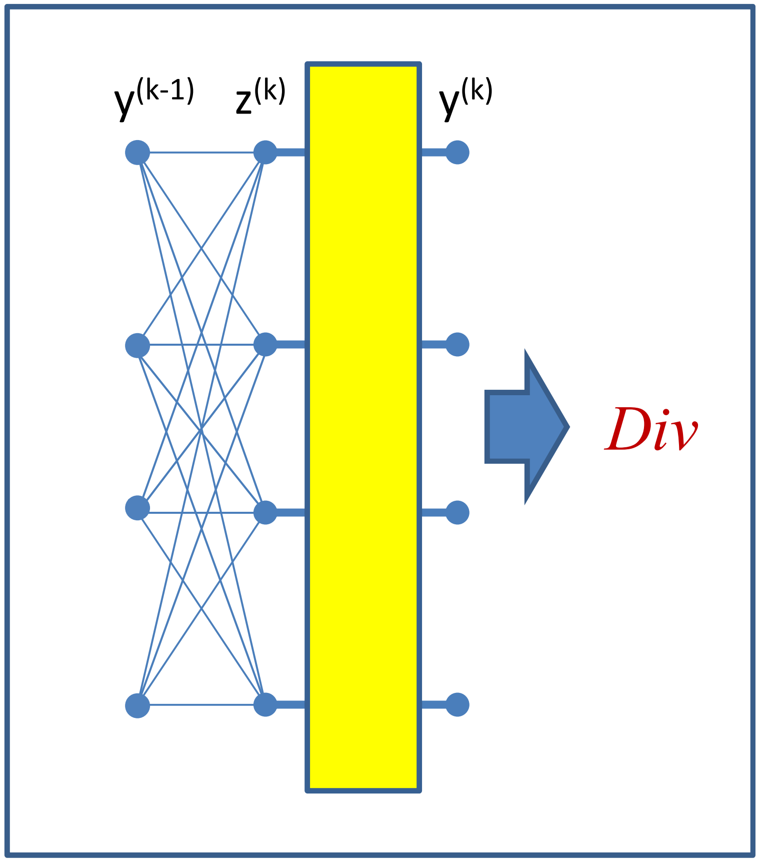
$$\frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} = 0 \;\; for \;\; i \neq j$$

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

Special cases

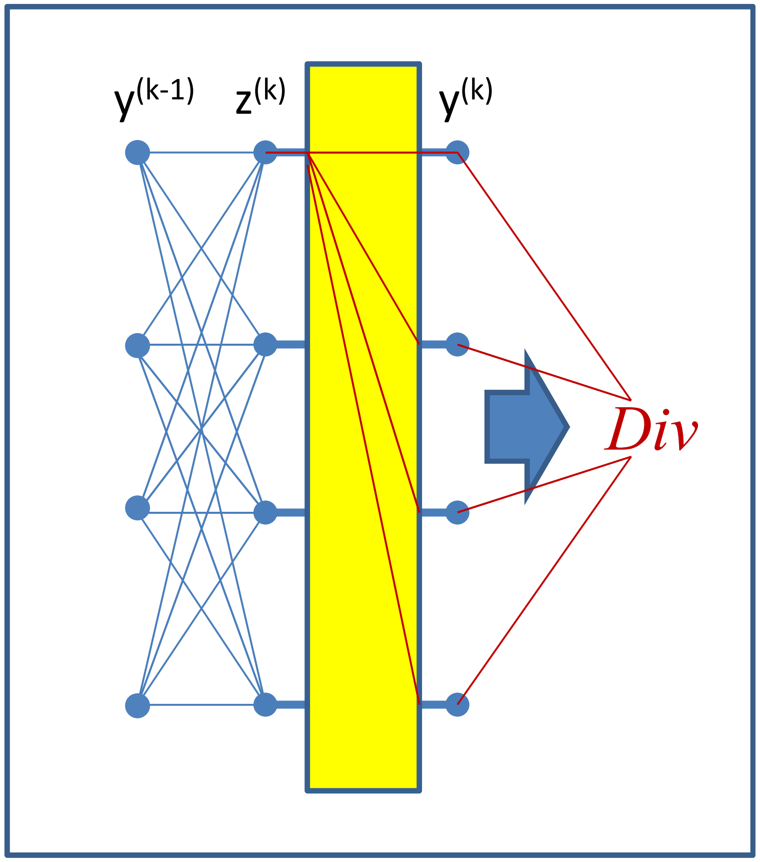
- Examples of vector activations and other special cases on slides
  - Please look up
  - Will appear in quiz!

Example Vector Activation: Softmax



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

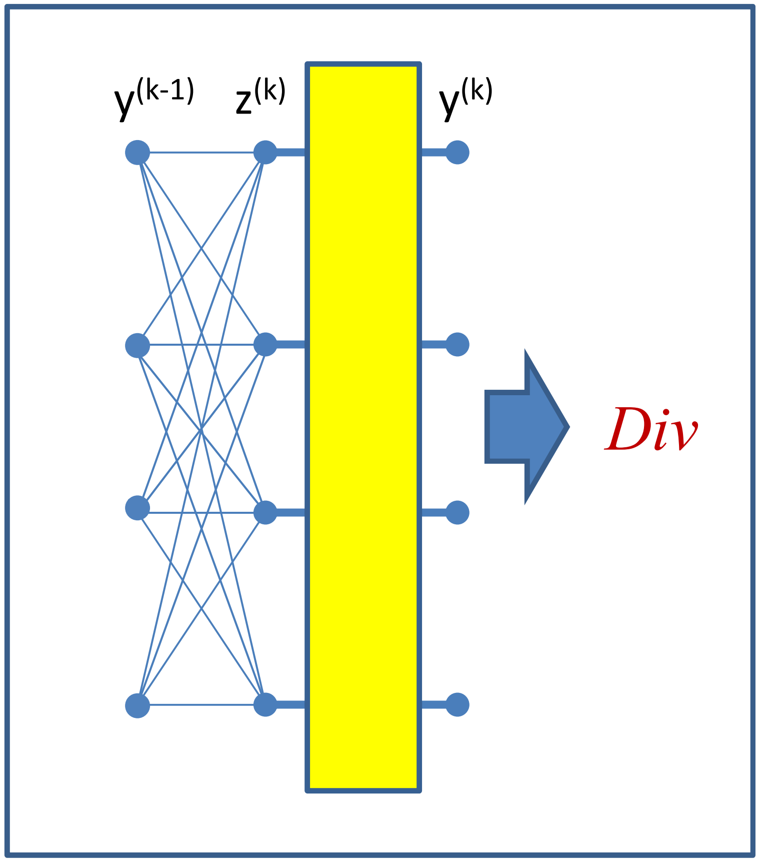
Example Vector Activation: Softmax



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

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

Example Vector Activation: Softmax

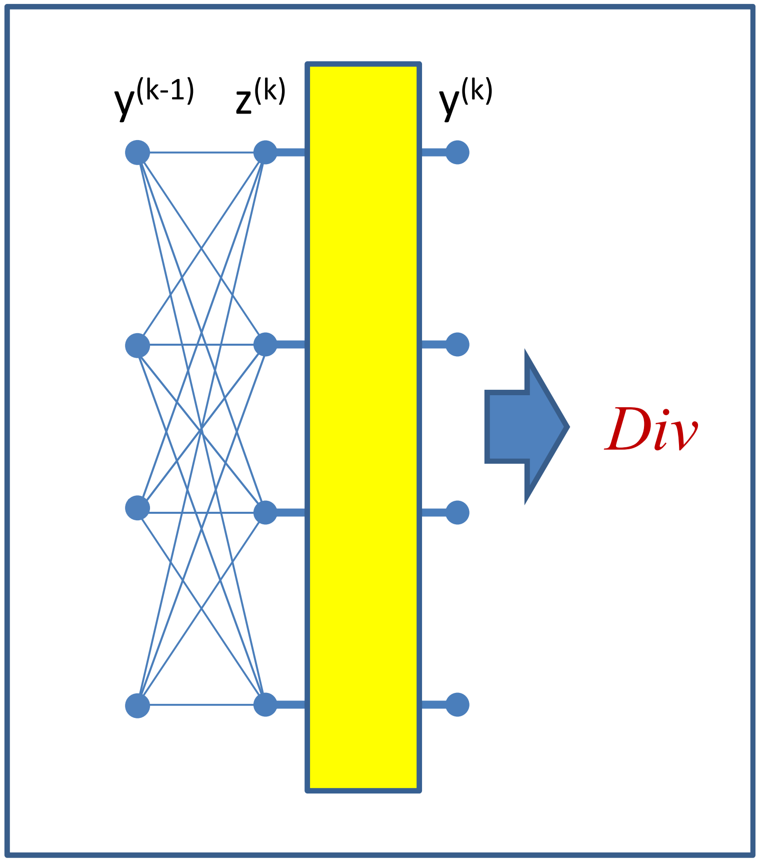


$$y_i^{(k)} = \frac{exp\left(z_i^{(k)}\right)}{\sum_j exp\left(z_j^{(k)}\right)}$$

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

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

Example Vector Activation: Softmax



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

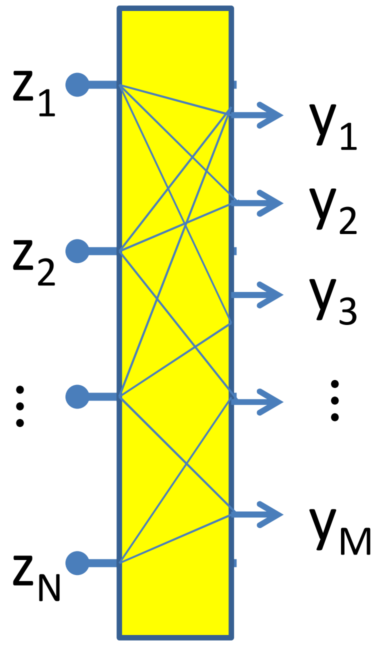
$$\boxed{\frac{\partial Div}{\partial z_i^{(k)}} = \sum_j \frac{\partial Div}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}}}$$

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

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

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

Subgradients and the Max



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

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

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

Backward Pass: Recap

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

Overall Approach

- For each data instance
  - **Forward pass**: Pass instance forward through the net. Store all intermediate outputs of all computation
  - **Backward pass**: Sweep backward through the net, iteratively compute all derivatives w.r.t weights
- Actual loss is the sum of the divergence over all training instances

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

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

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

$$W \leftarrow W - \eta \nabla_W \mathbf{Loss}^{\mathrm{T}}$$

Training by BackProp

- Initialize weights $\boldsymbol{W}^{(k)}$ for all layers $k = 1 \dots K$
- Do:
  - **Initialize** $Err = 0$; For all i, j, k, initialize $\frac{dErr}{dw_{i,j}^{(k)}} = 0$
  - For all $t = 1{:}T$ (Loop over training instances)
    - **Forward pass:** Compute
      - Output $\boldsymbol{Y_t}$
      - $Err += \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$
    - **Backward pass:** For all i, j, k:
      - Compute $\frac{d\boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{dw_{i,j}^{(k)}}$
      - Compute $\frac{dErr}{dw_{i,j}^{(k)}} += \frac{d\boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{dw_{i,j}^{(k)}}$
  - For all i, j, k, update:

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

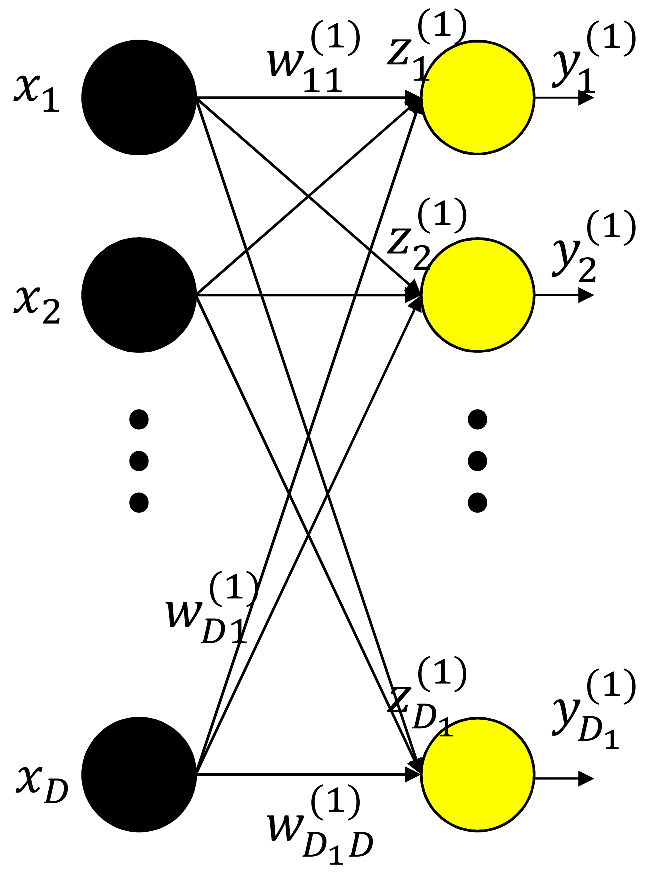
- Until Err has converged

Vector formulation

- For layered networks it is generally simpler to think of the process in terms of vector operations
  - Simpler arithmetic
  - Fast matrix libraries make operations *much* faster

- We can restate the entire process in vector terms
  - This is what is *actually* used in any real system

Vector formulation

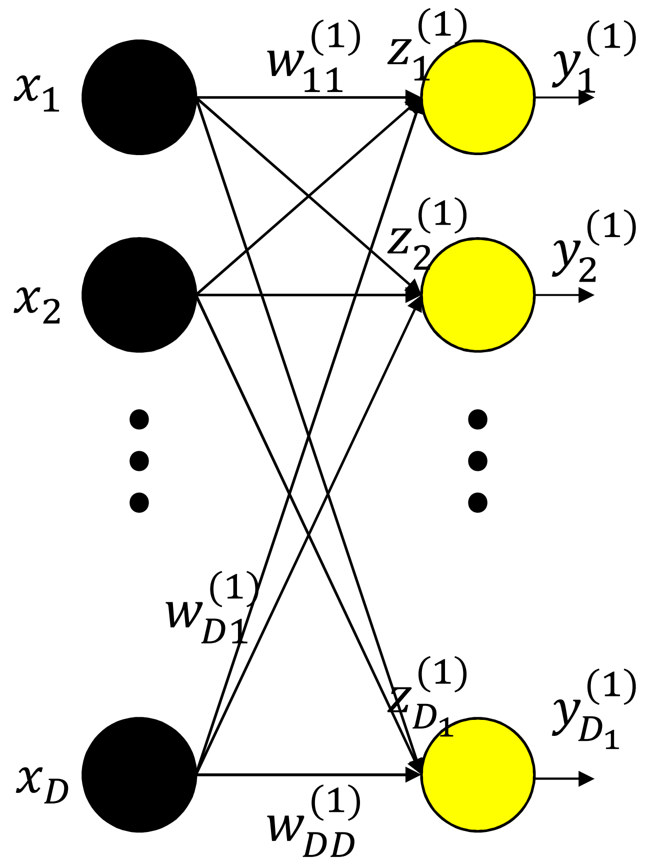


$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix} \qquad \mathbf{z_k} = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{D_k}^{(k)} \end{bmatrix} \qquad \mathbf{y_k} = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_{D_k}^{(k)} \end{bmatrix}$$

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

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

Vector formulation



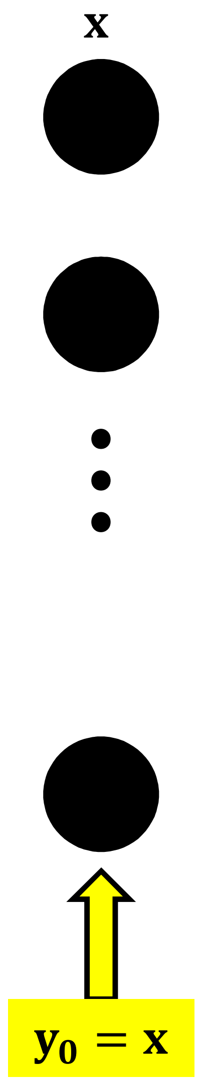
$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix} \qquad \mathbf{z}_k = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{D_k}^{(k)} \end{bmatrix} \qquad \mathbf{y}_k = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_{D_k}^{(k)} \end{bmatrix}$$

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

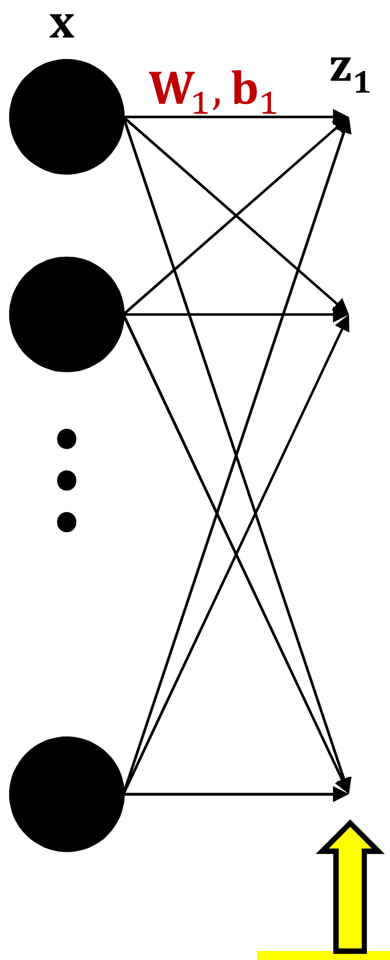
- The computation of a single layer is easily expressed in matrix notation as (setting $\mathbf{y_0} = \mathbf{x}$):

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

The forward pass: Evaluating the network

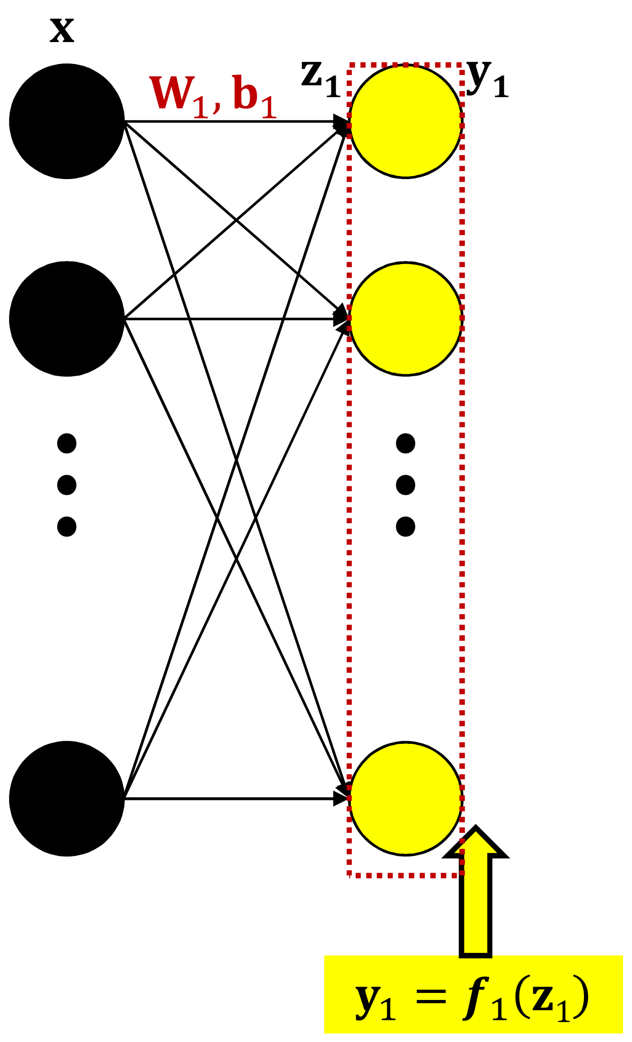


The forward pass



$$\mathbf{z_1 = W_1 x + b_1}$$

The forward pass

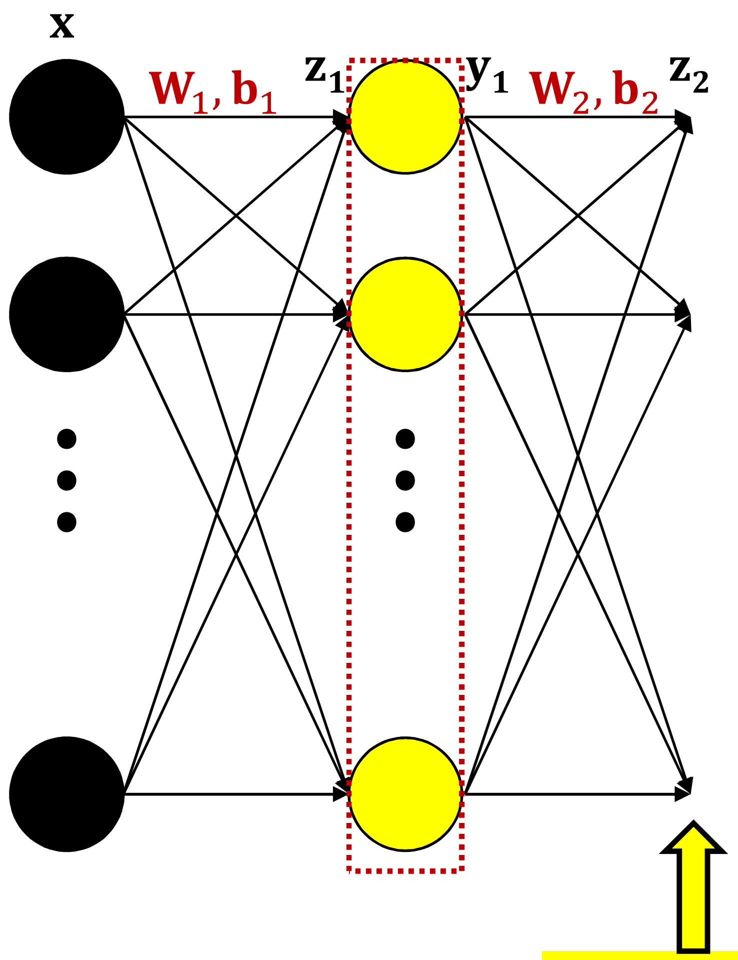


$\mathbf{x}$
$\mathbf{W}_1, \mathbf{b}_1$
$\mathbf{z}_1$
$\mathbf{y}_1$
$\mathbf{y}_1 = \boldsymbol{f}_1(\mathbf{z}_1)$

The Complete computation

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

The forward pass

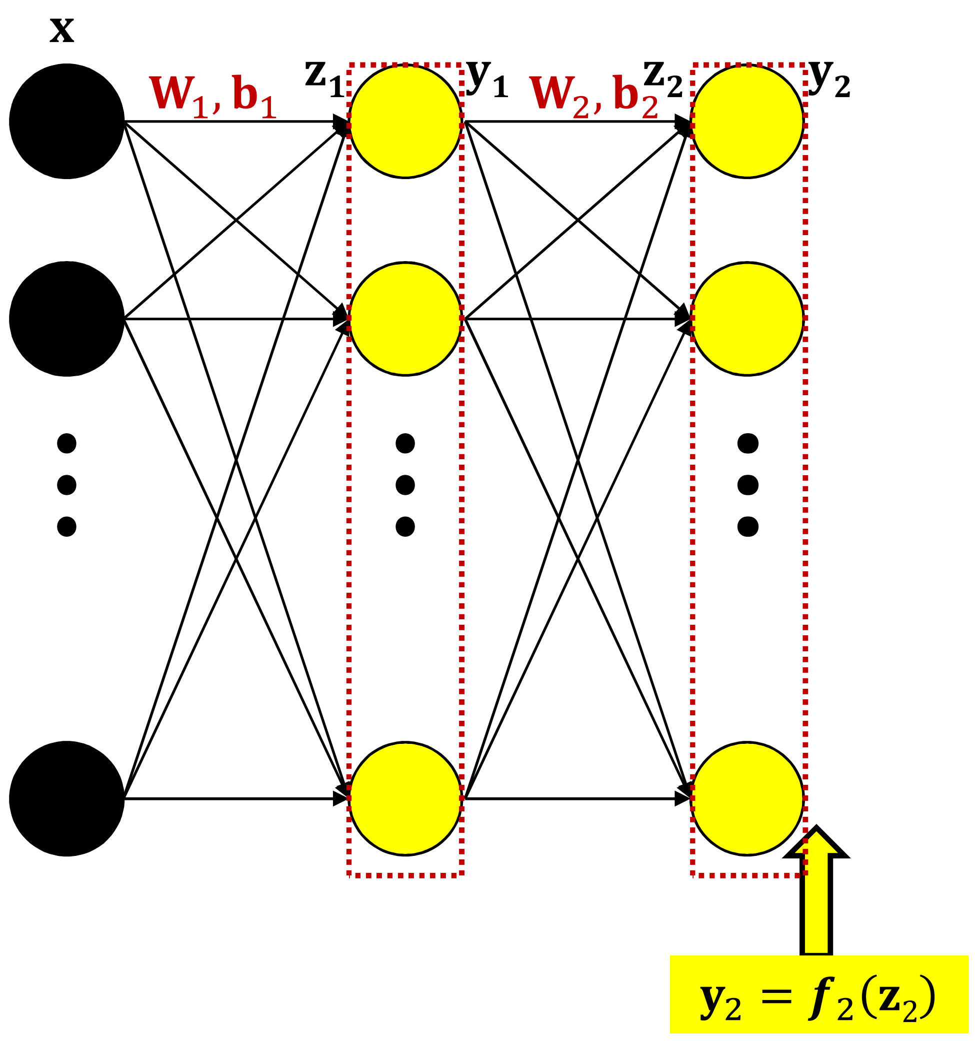


$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{y}_1 + \mathbf{b}_2$$

The Complete computation

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

The forward pass

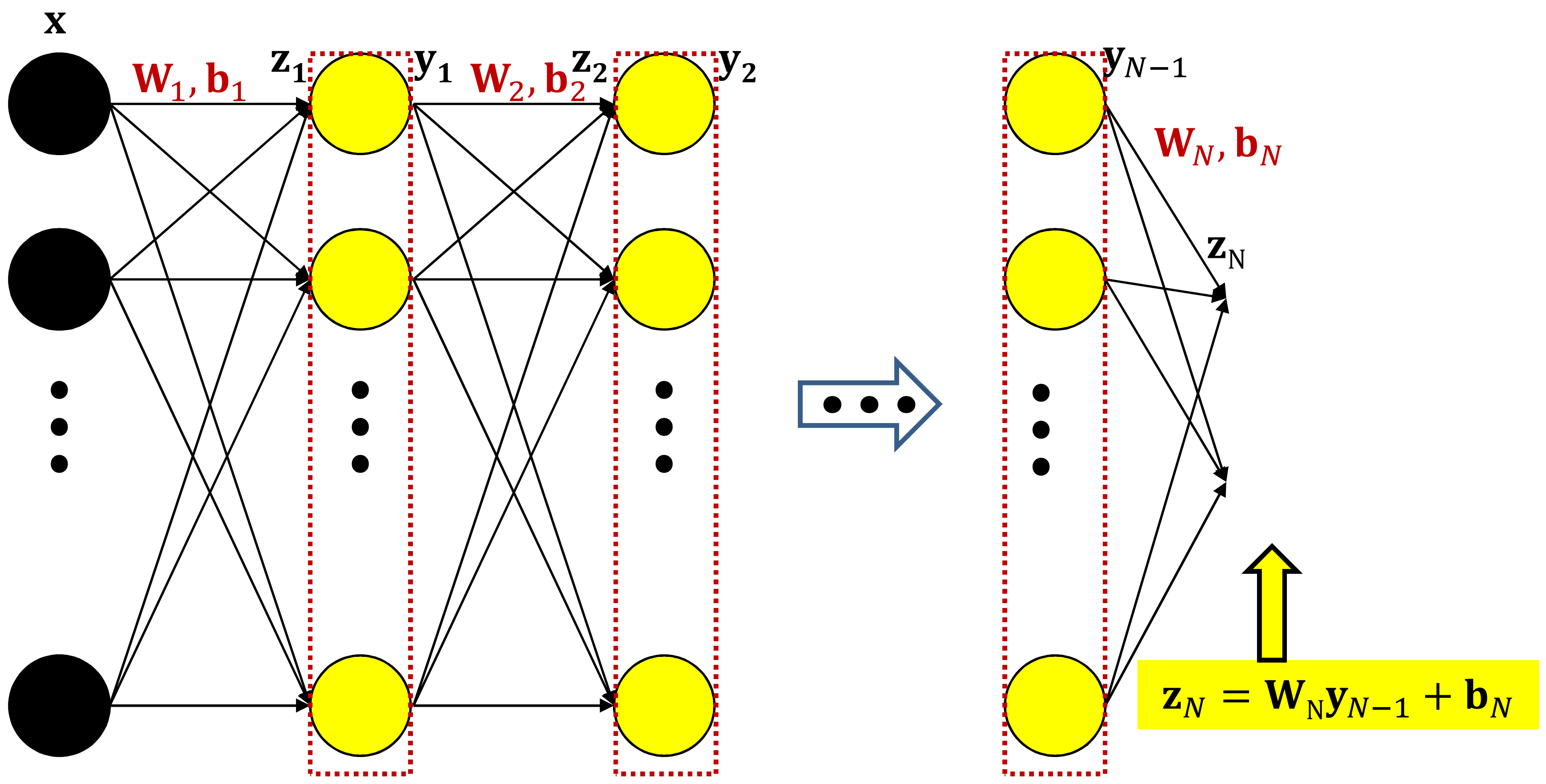


$\mathbf{x}$
$\mathbf{W}_1, \mathbf{b}_1$
$\mathbf{z}_1$
$\mathbf{y}_1$
$\mathbf{W}_2, \mathbf{b}_2$
$\mathbf{z}_2$
$\mathbf{y}_2$
$\mathbf{y}_2 = \boldsymbol{f}_2(\mathbf{z}_2)$

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

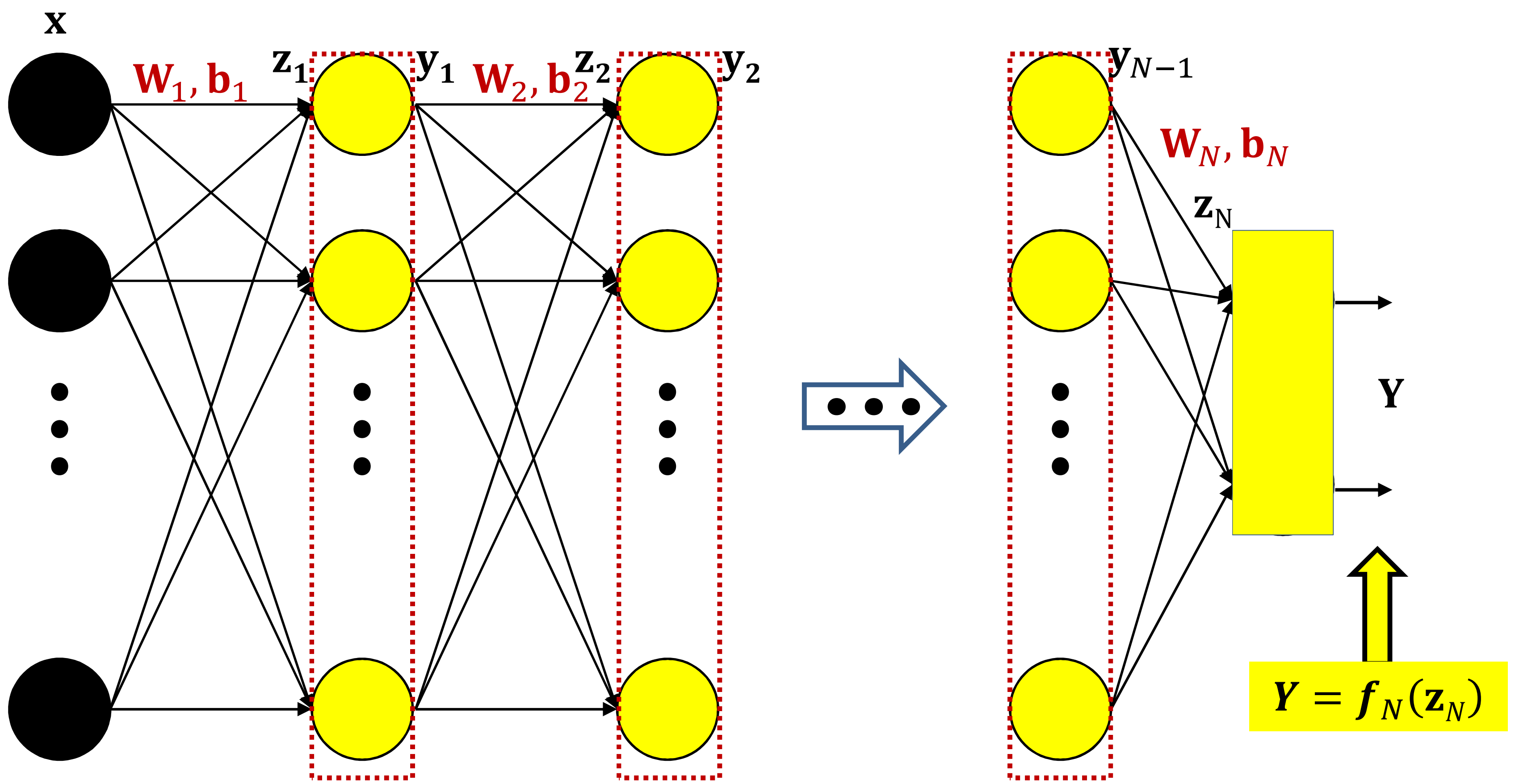


$$\mathbf{z}_N = \mathbf{W}_\mathrm{N}\mathbf{y}_{N-1} + \mathbf{b}_N$$

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

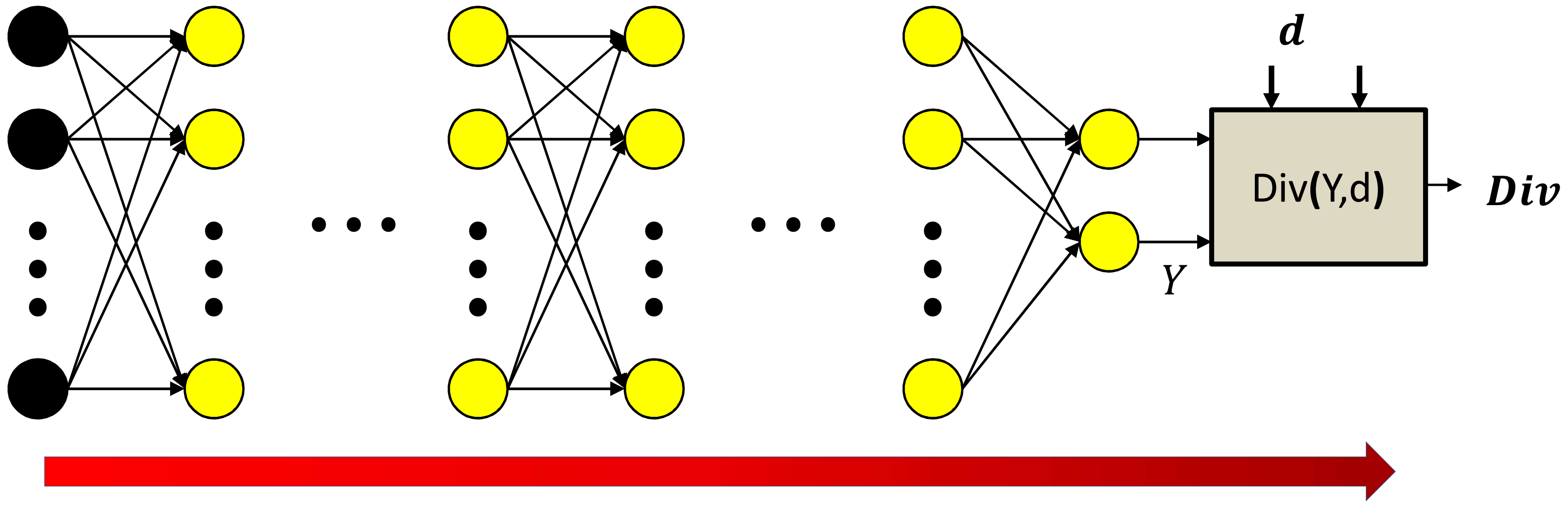


$\mathbf{x}$
$\mathbf{W}_1, \mathbf{b}_1$
$\mathbf{z}_1$
$\mathbf{y}_1$
$\mathbf{W}_2, \mathbf{b}_2$
$\mathbf{z}_2$
$\mathbf{y}_2$
$\mathbf{y}_{N-1}$
$\mathbf{W}_N, \mathbf{b}_N$
$\mathbf{z}_\mathrm{N}$
$\mathbf{Y}$
$\boldsymbol{Y} = \boldsymbol{f}_N(\mathbf{z}_N)$

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



Forward pass:

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

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

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

The Forward Pass

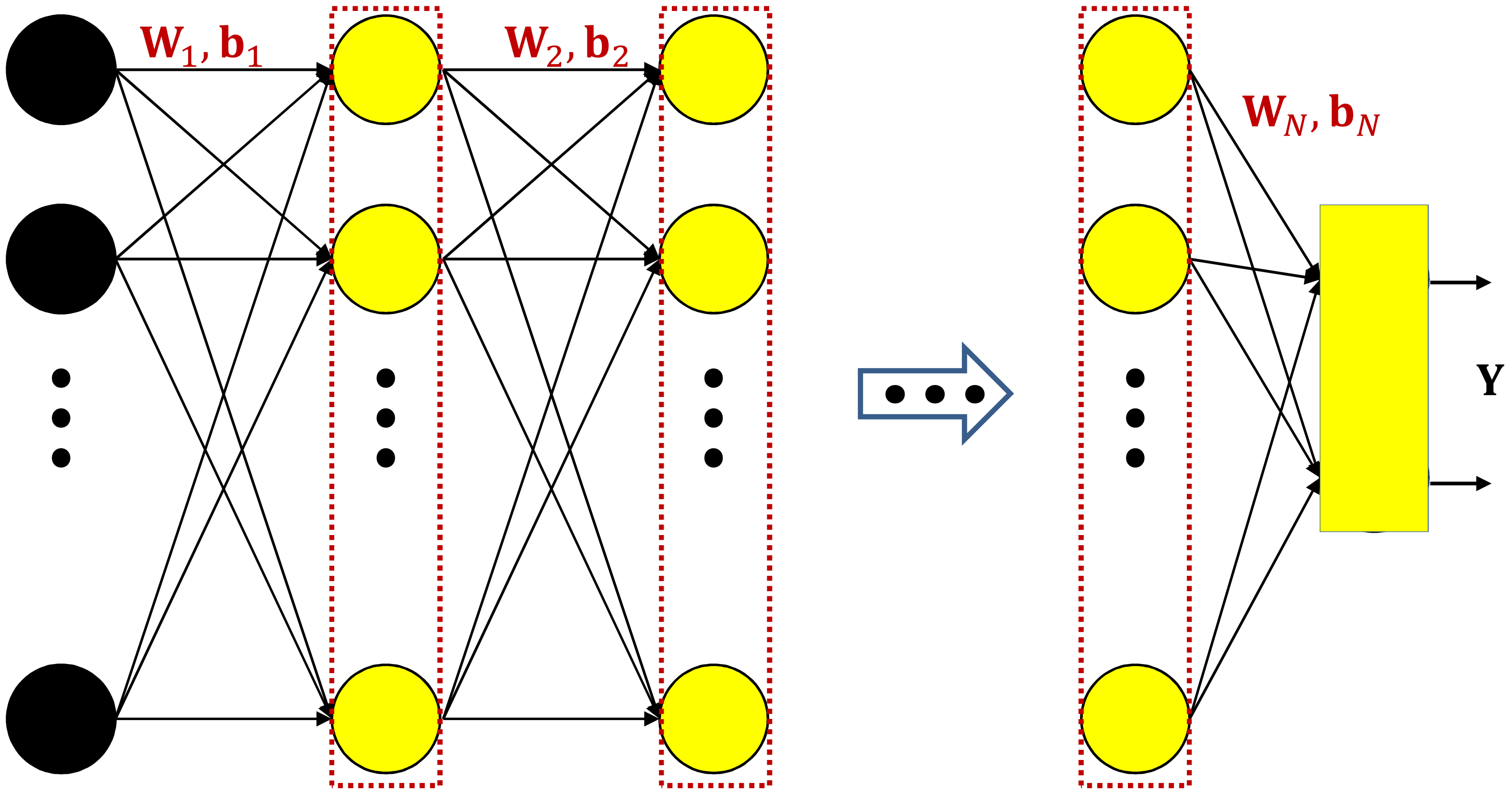
- Set $\mathbf{y}_0 = \mathbf{x}$
- Recursion through layers
  - For layer k = 1 to N:

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

- Output:

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

The backward pass


$\mathbf{W}_1, \mathbf{b}_1$
$\mathbf{W}_2, \mathbf{b}_2$
$\mathbf{W}_N, \mathbf{b}_N$
$\mathbf{Y}$

- The network is a nested function

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

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

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

Calculus recap: 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

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

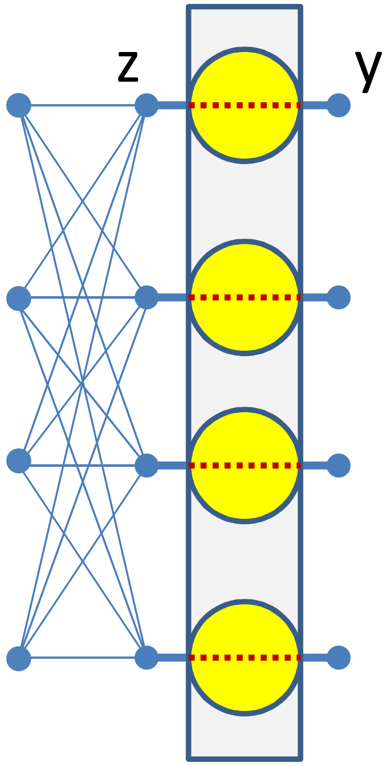
$$J_{\mathbf{y}}(\mathbf{z}) = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \cdots & \frac{\partial y_1}{\partial z_D} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \cdots & \frac{\partial y_2}{\partial z_D} \\ \vdots & \vdots &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

Using vector notation

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

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

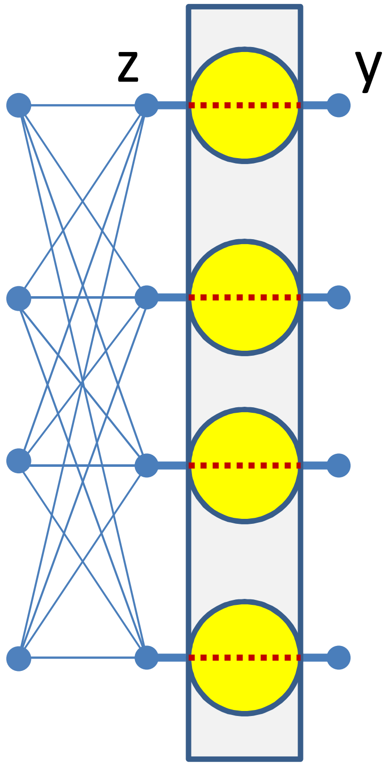
Jacobians can describe the derivatives of neural activations w.r.t their input



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

- **For Scalar activations**
  - Number of outputs is identical to the number of inputs
- Jacobian is a diagonal matrix
  - Diagonal entries are individual derivatives of outputs w.r.t inputs
  - Not showing the superscript “(k)” in equations for brevity

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

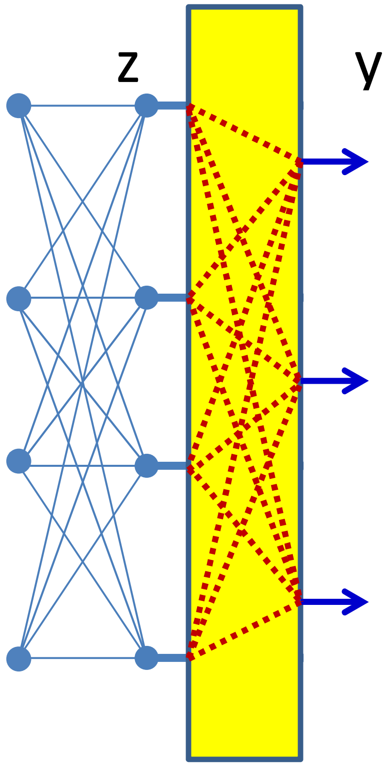


$$\boxed{y_i = f(z_i)}$$

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

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

For *Vector* activations



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

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

Special case: Affine functions

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

$$\Big\Downarrow$$

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

- Matrix $\mathbf{W}$ and bias $\mathbf{b}$ operating on vector $\mathbf{y}$ to produce vector $\mathbf{z}$
- The Jacobian of $\mathbf{z}$ w.r.t $\mathbf{y}$ is simply the matrix $\mathbf{W}$

Vector derivatives: Chain rule

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

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

$$\Downarrow$$

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

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

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

Check

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

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

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

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

Vector derivatives: Chain rule

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

$$\begin{tikzcd}
\boxed{D = f(\boldsymbol{g}(\mathbf{x}))} \arrow[d] & \boxed{\nabla_{\mathbf{x}} D = \nabla_{\mathbf{z}}(D) J_{\mathbf{z}}(\mathbf{x})} \\
\boxed{\mathbf{z} = \boldsymbol{g}(\mathbf{x})} \arrow[ru, bend left] & \\
\boxed{D = f(\mathbf{z})} &
\end{tikzcd}$$

Check

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

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

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

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

Special Case

- Scalar functions of Affine functions

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

$$\boxed{\mathbf{z} = \mathbf{Wy} + \mathbf{b}}$$

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

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

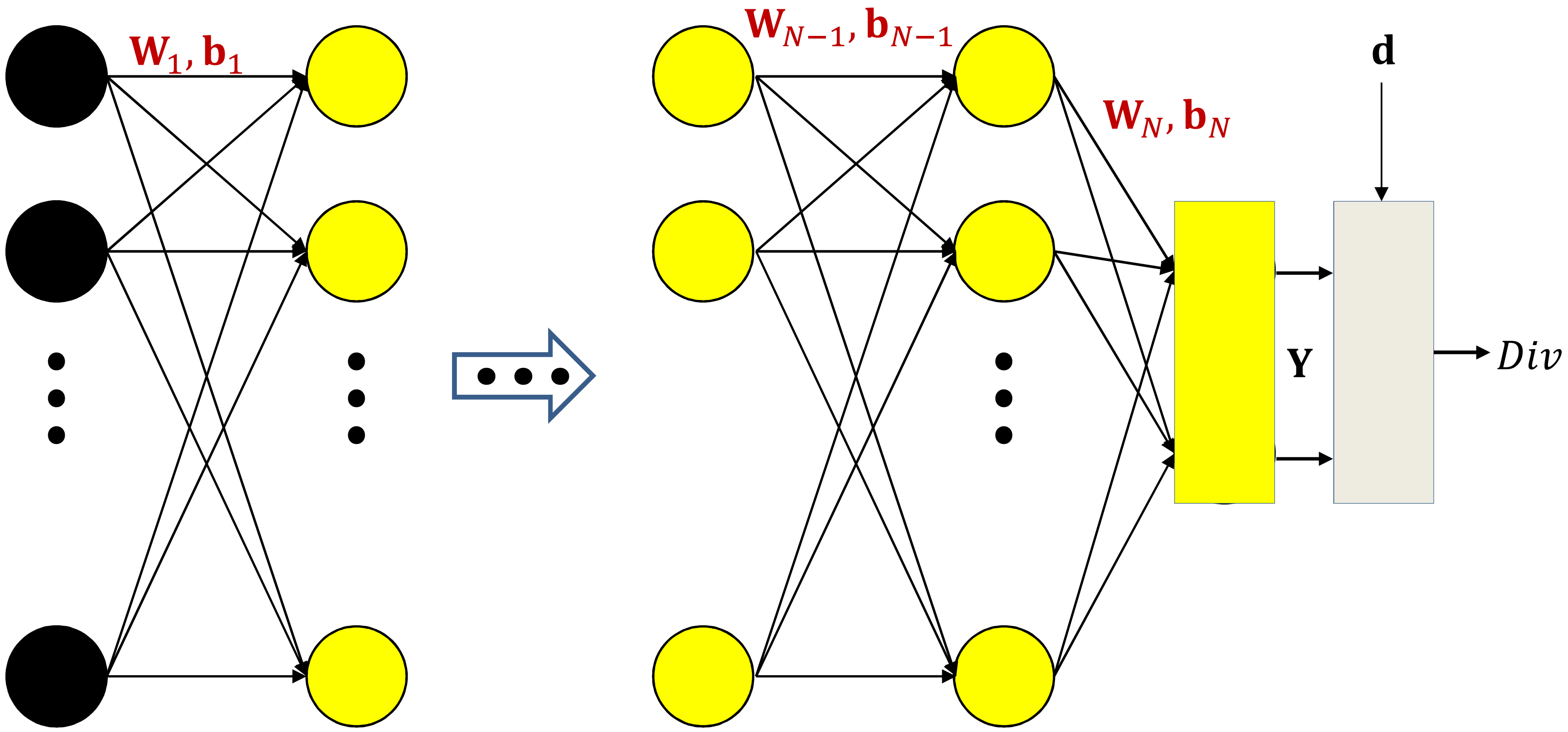
$$\boxed{\nabla_{\mathbf{b}} D = \nabla_{\mathbf{z}}(D)}$$

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

Derivatives w.r.t parameters

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

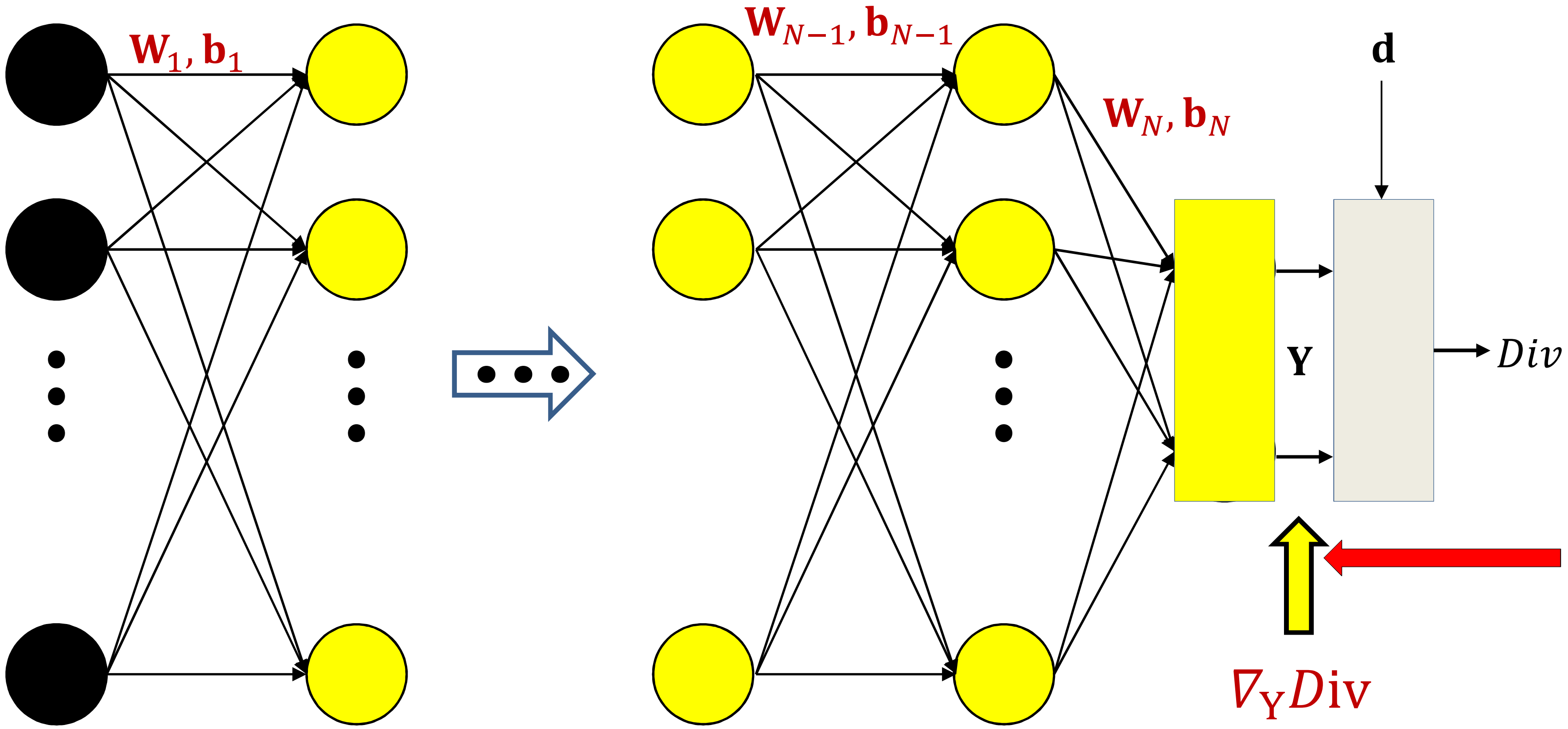
The backward pass


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

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

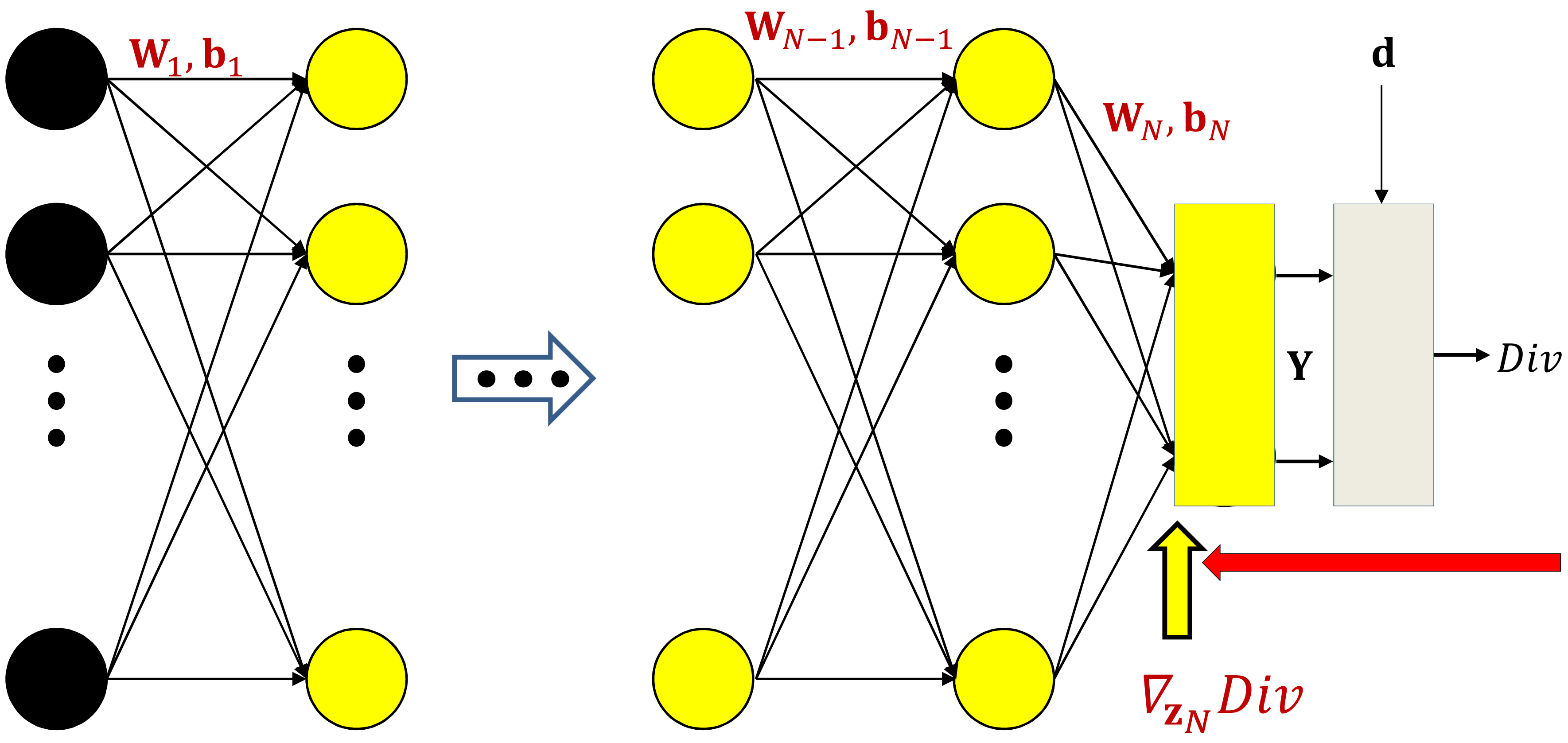
In general $\nabla_{\mathbf{a}}\mathbf{b}$ represents a derivative of $\mathbf{b}$ w.r.t. $\mathbf{a}$ and could be a gradient (for scalar $\mathbf{b}$) Or a Jacobian (for vector $\mathbf{b}$)

The backward pass


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

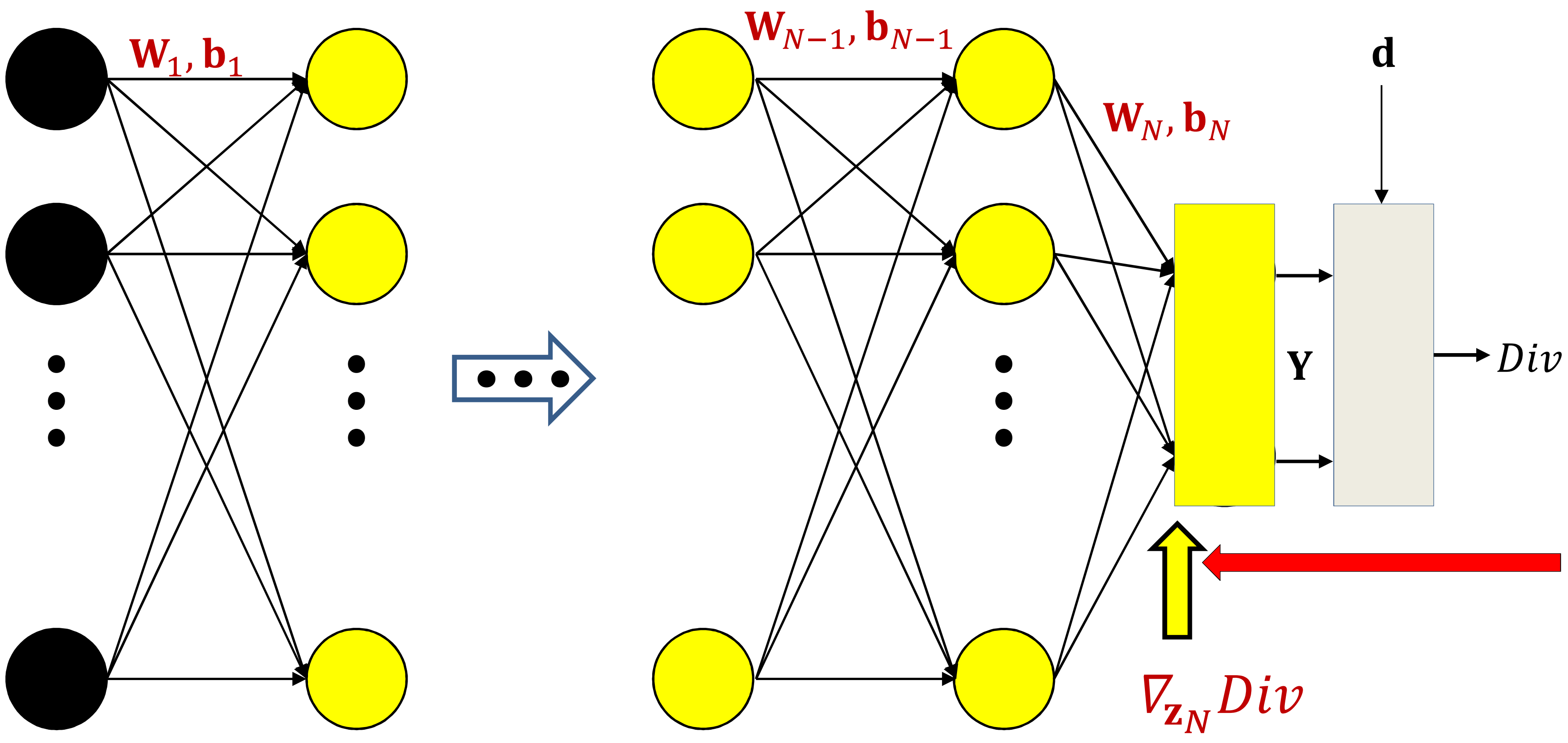
First compute the gradient of the divergence w.r.t. Y.
The actual gradient depends on the divergence function.

The backward pass


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

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

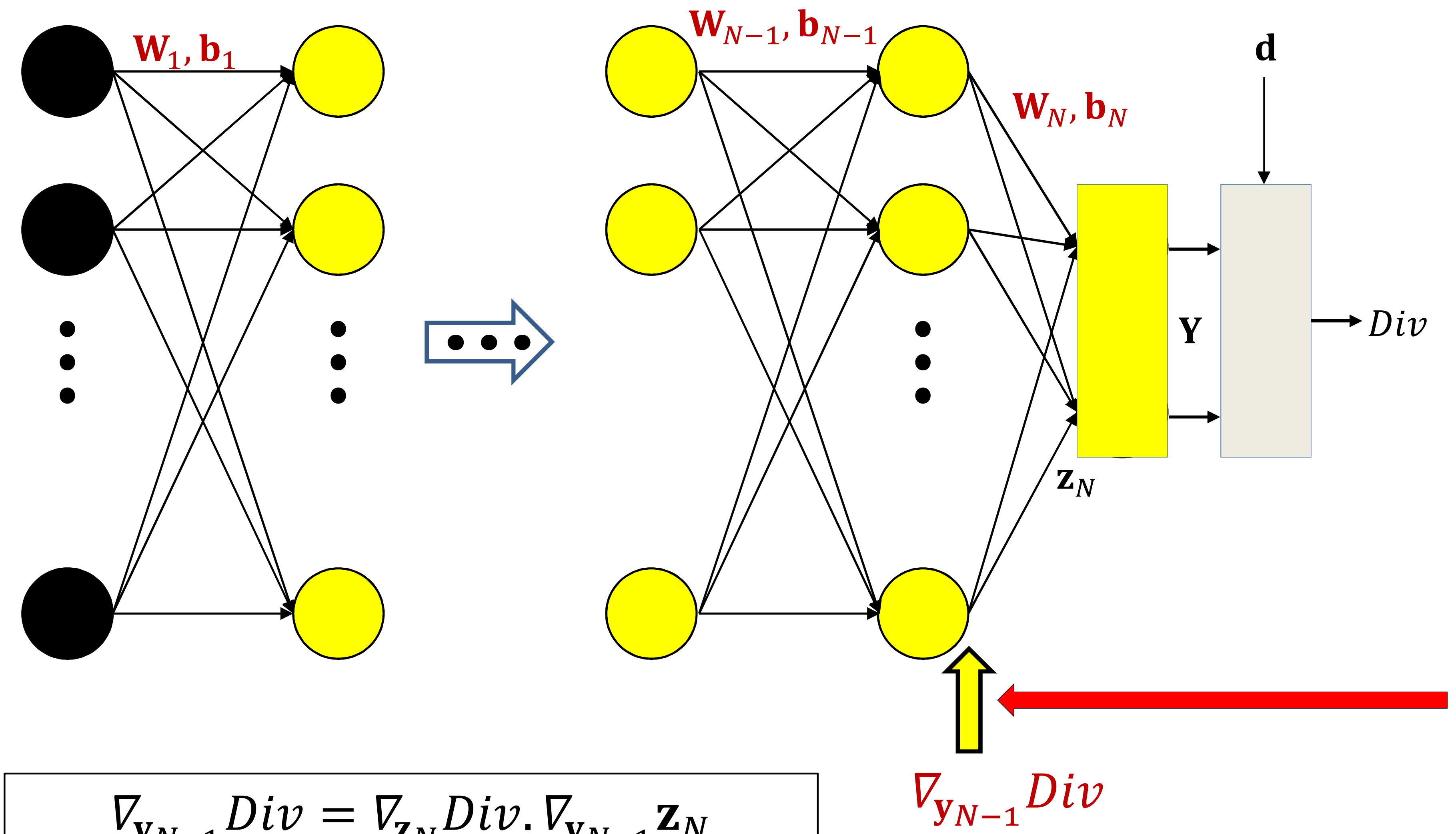
The backward pass



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

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

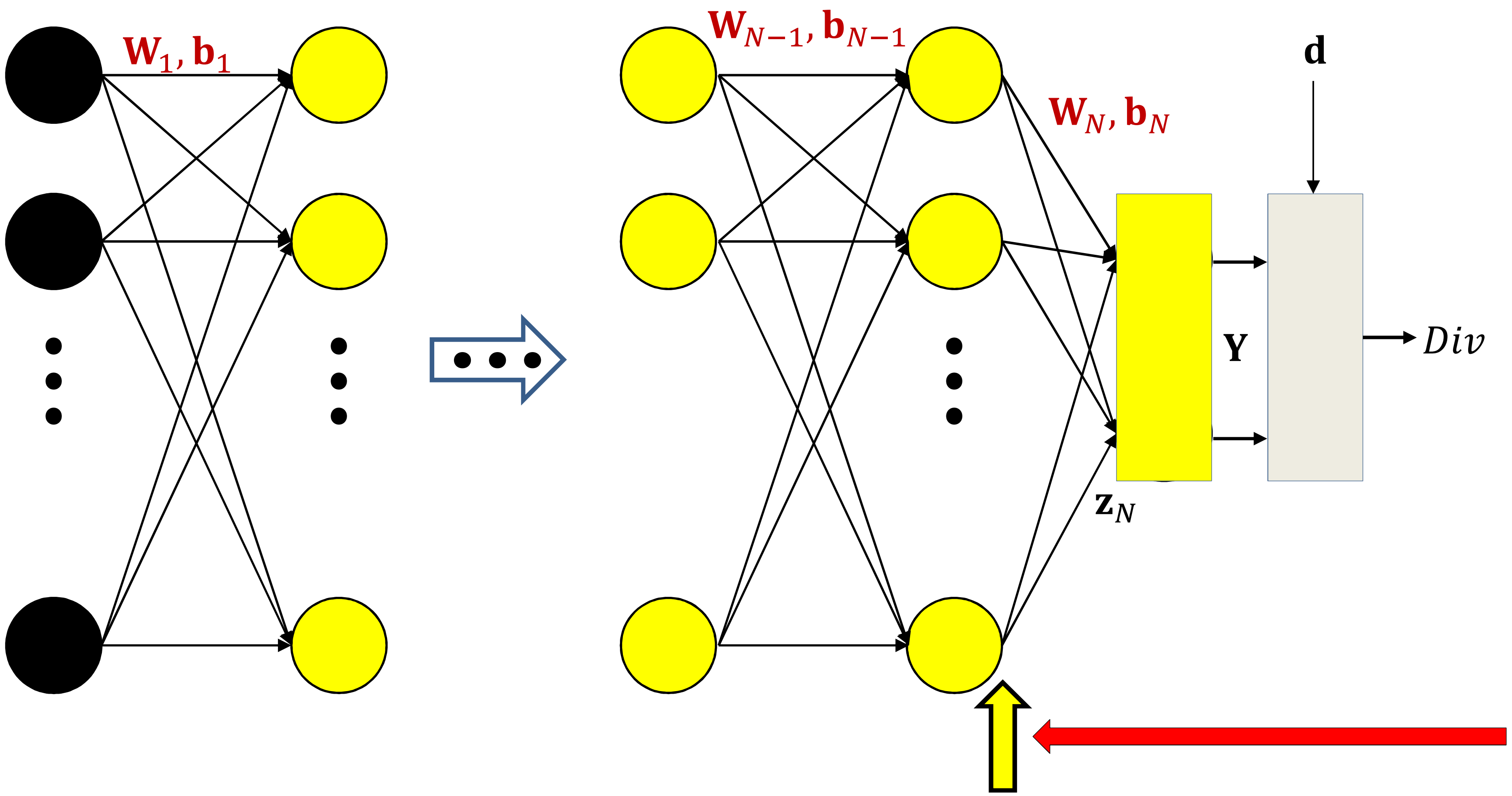
The backward pass



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

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

The backward pass

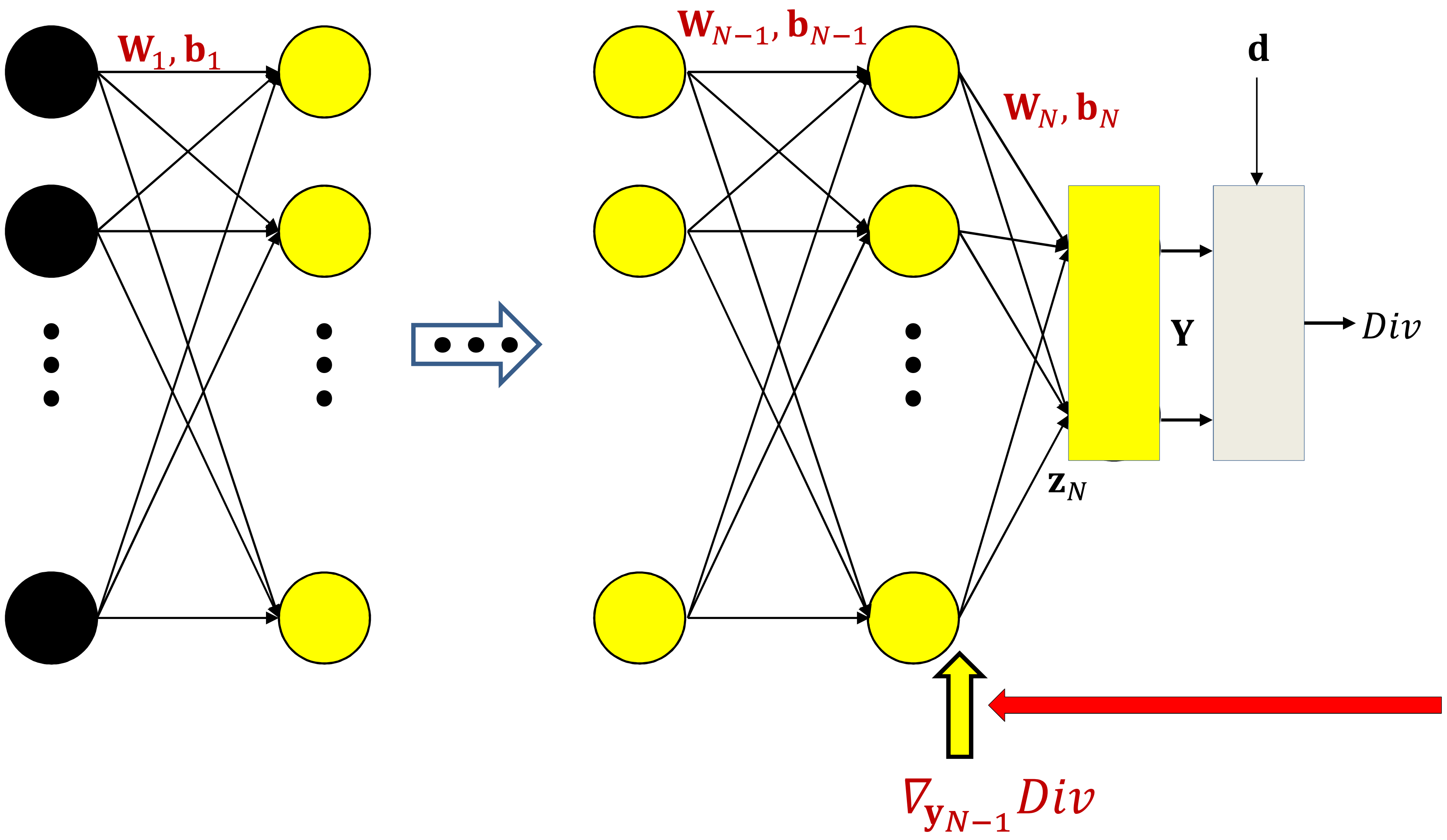


$\mathbf{W}_1, \mathbf{b}_1$
$\mathbf{W}_{N-1}, \mathbf{b}_{N-1}$
$\mathbf{W}_N, \mathbf{b}_N$
$\mathbf{d}$
$\mathbf{Y}$
Div
$\mathbf{z}_N$

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

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

The backward pass



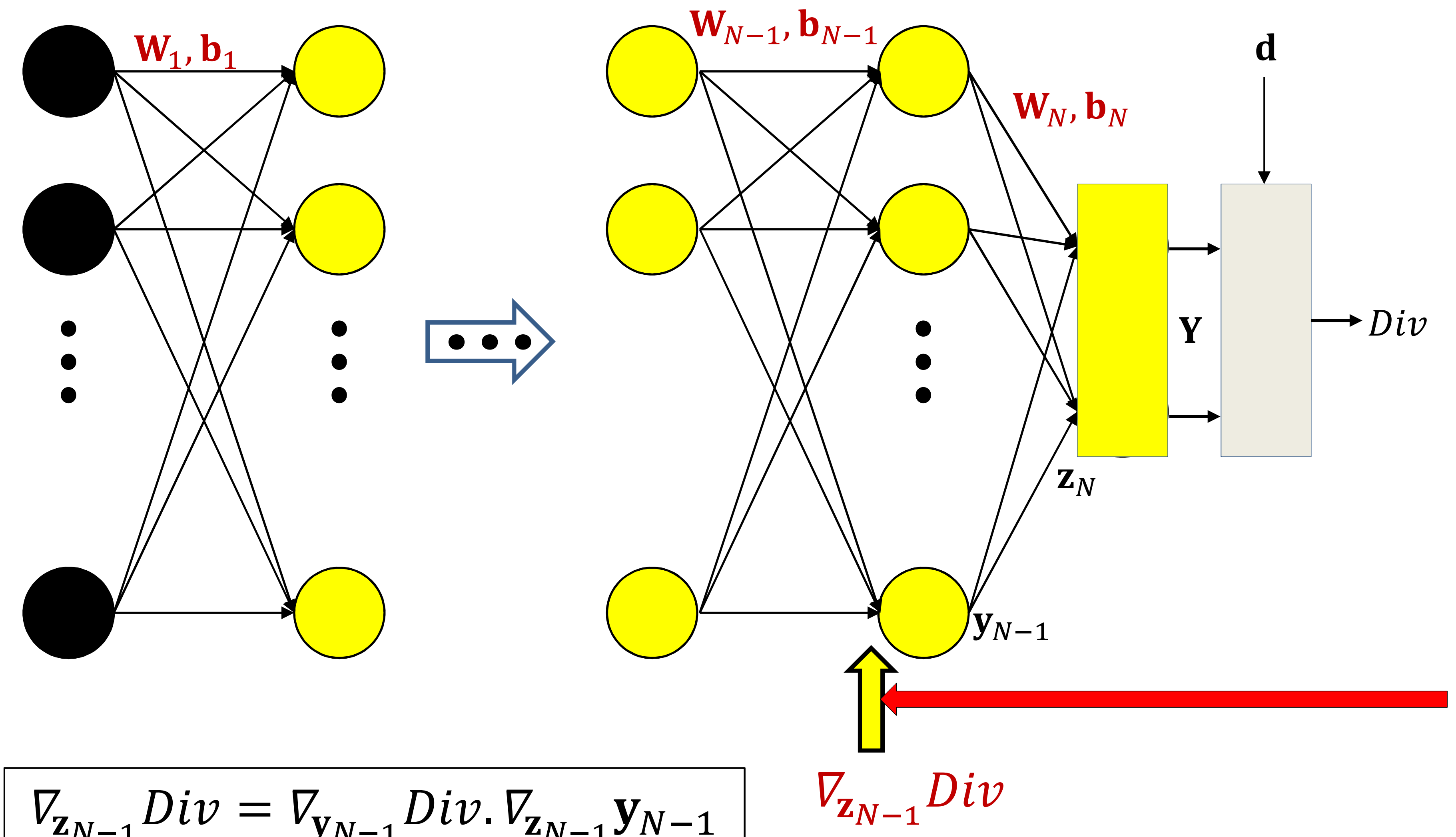
$\mathbf{W}_1, \mathbf{b}_1$
$\mathbf{W}_{N-1}, \mathbf{b}_{N-1}$
$\mathbf{W}_N, \mathbf{b}_N$
$\mathbf{d}$
$\mathbf{Y}$
Div
$\mathbf{z}_N$
$\nabla_{\mathbf{y}_{N-1}} Div$

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

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

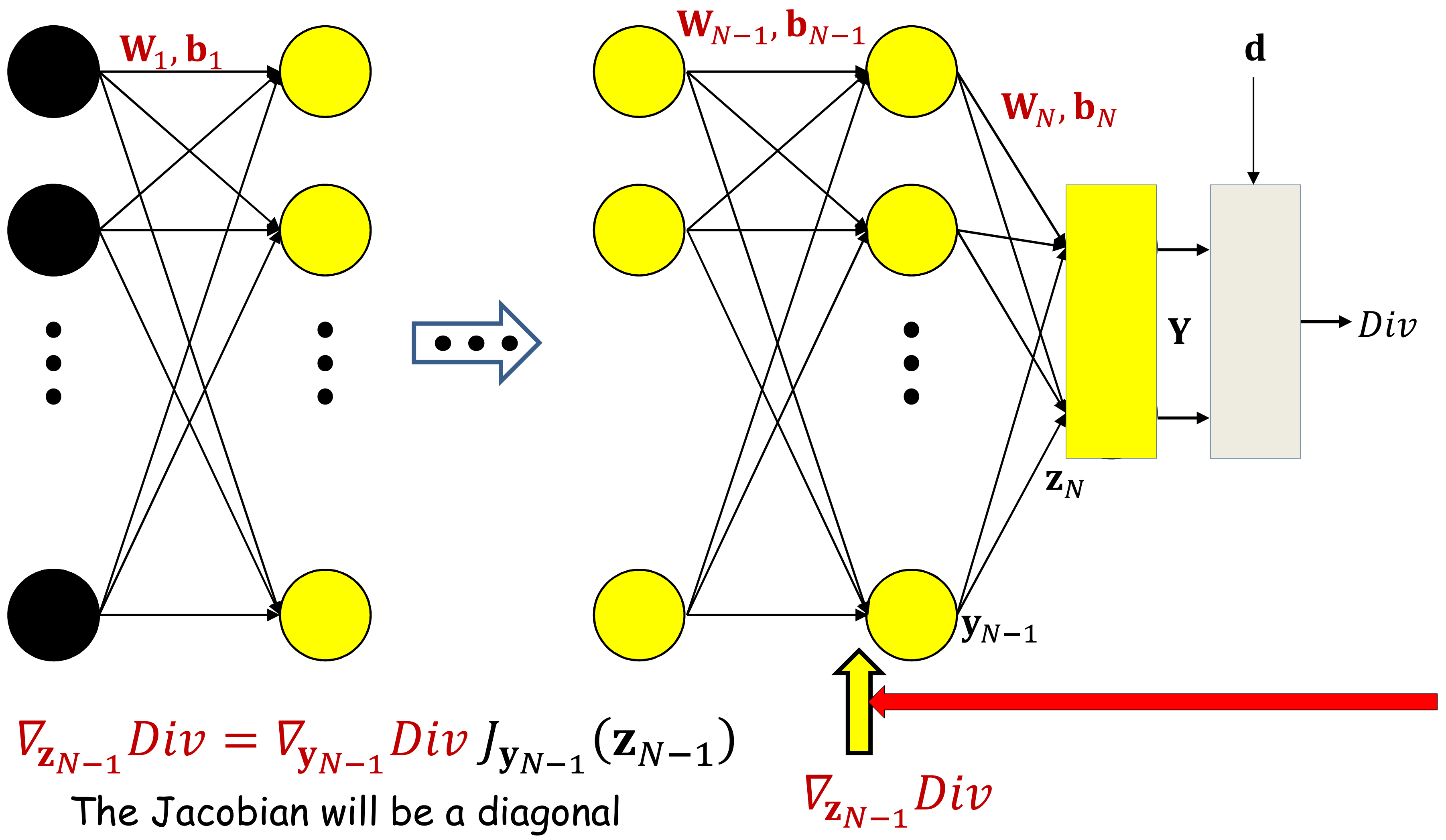
$$\boxed{\nabla_{\mathbf{b}_N} Div = \nabla_{\mathbf{z}_N} Div}$$

The backward pass


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

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

The backward pass

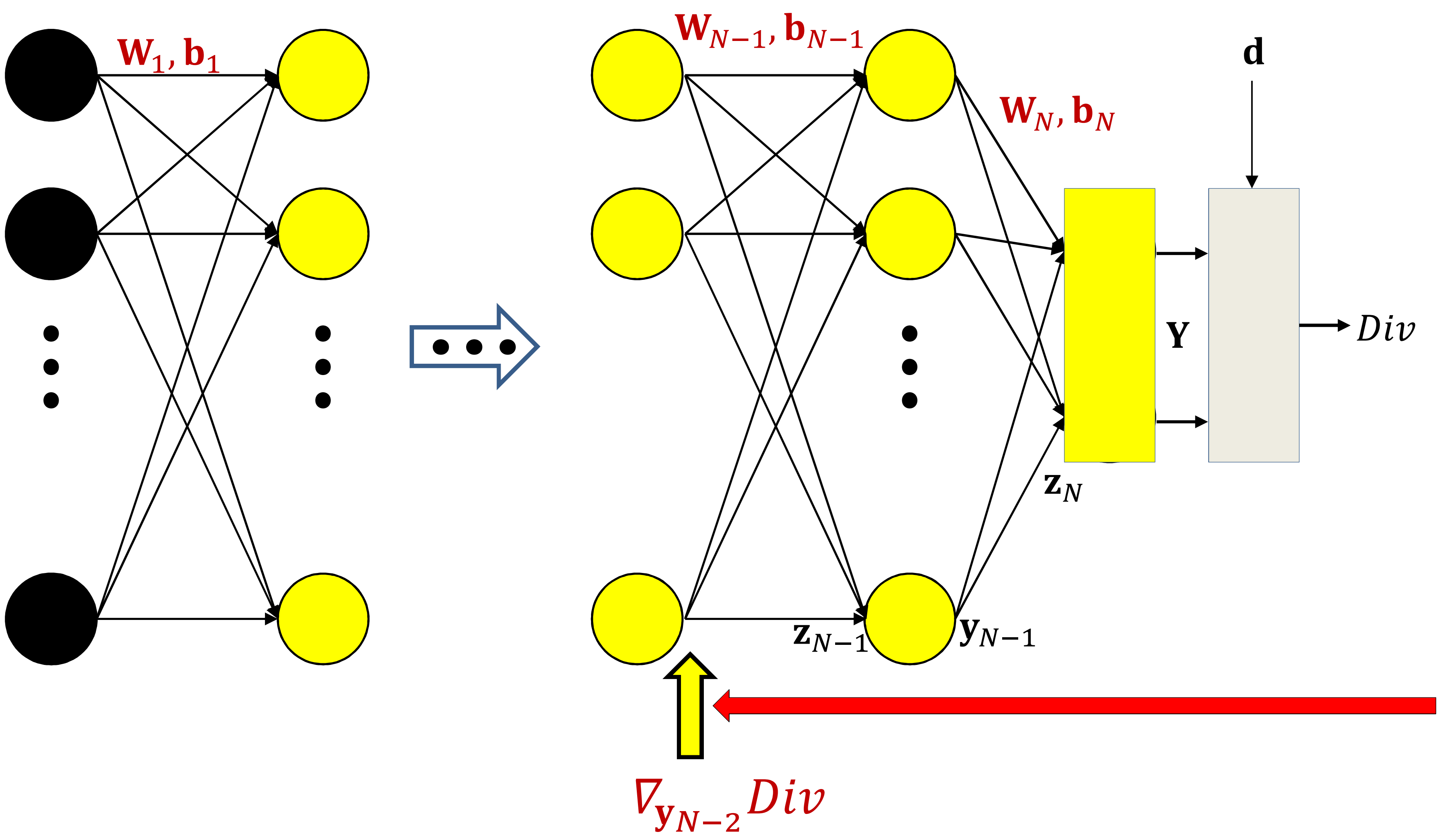


$\mathbf{W}_1, \mathbf{b}_1$
$\mathbf{W}_{N-1}, \mathbf{b}_{N-1}$
$\mathbf{W}_N, \mathbf{b}_N$
$\mathbf{d}$
$\mathbf{Y}$
Div
$\mathbf{z}_N$
$\mathbf{y}_{N-1}$
$\nabla_{\mathbf{z}_{N-1}} Div$

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

The Jacobian will be a diagonal matrix for scalar activations

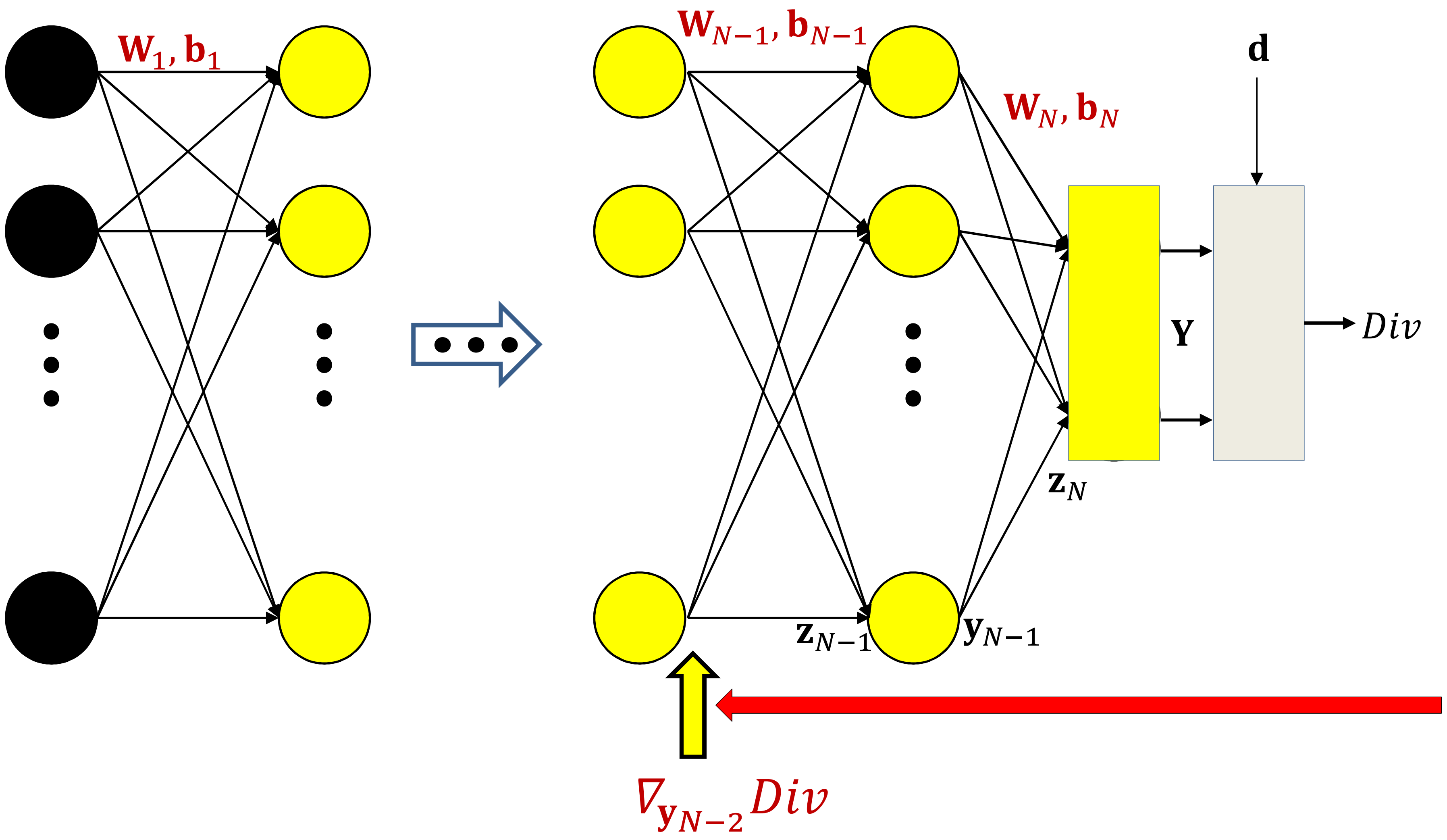
The backward pass



$\mathbf{W}_1, \mathbf{b}_1$
$\mathbf{W}_{N-1}, \mathbf{b}_{N-1}$
$\mathbf{W}_N, \mathbf{b}_N$
$\mathbf{d}$
$\mathbf{Y}$
Div
$\mathbf{z}_N$
$\mathbf{z}_{N-1}$
$\mathbf{y}_{N-1}$
$\nabla_{\mathbf{y}_{N-2}} Div$

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

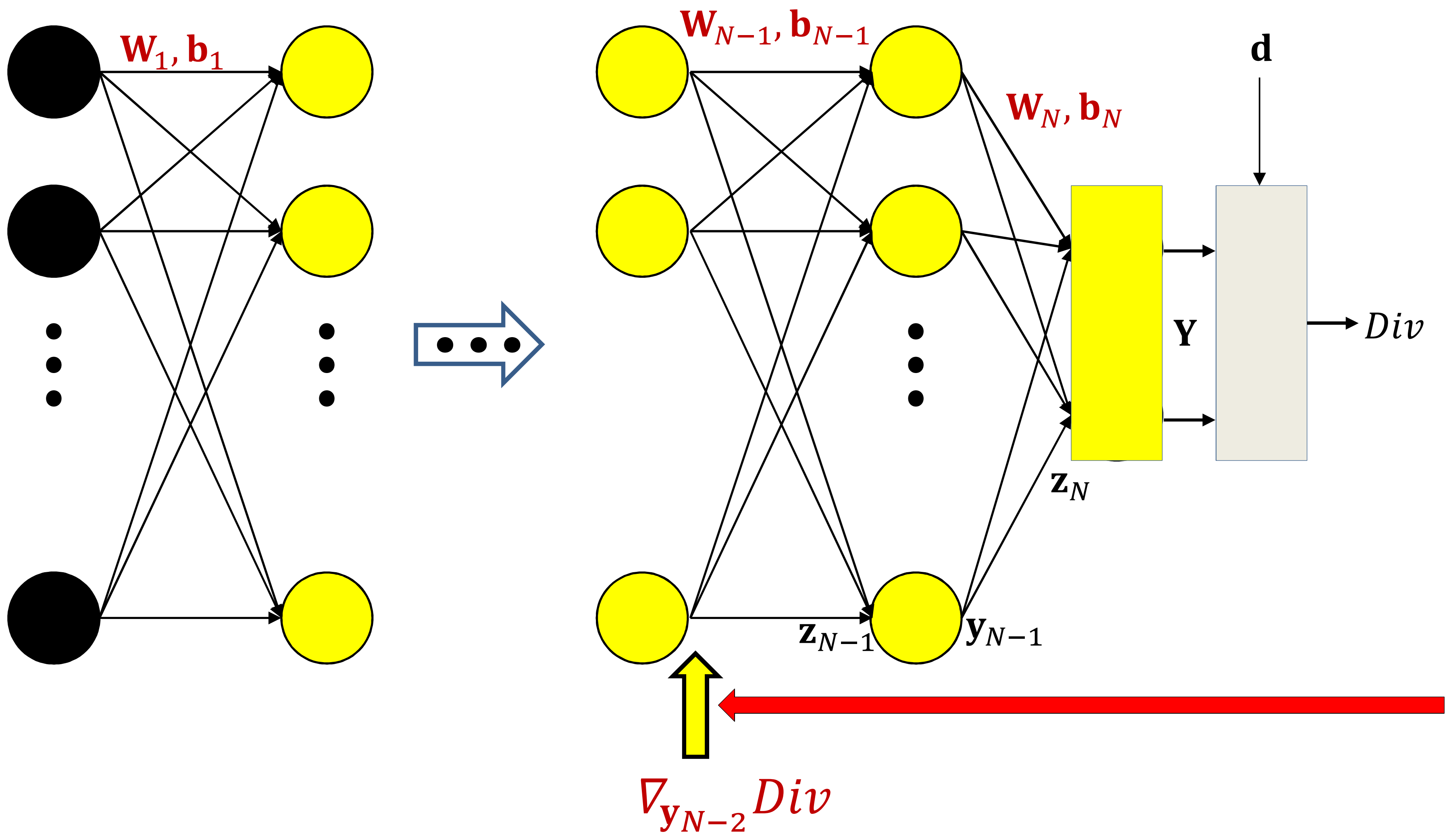
The backward pass



$\mathbf{W}_1, \mathbf{b}_1$
$\mathbf{W}_{N-1}, \mathbf{b}_{N-1}$
$\mathbf{W}_N, \mathbf{b}_N$
$\mathbf{d}$
$\mathbf{Y}$
Div
$\mathbf{z}_N$
$\mathbf{z}_{N-1}$
$\mathbf{y}_{N-1}$
$\nabla_{\mathbf{y}_{N-2}} Div$

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

The backward pass

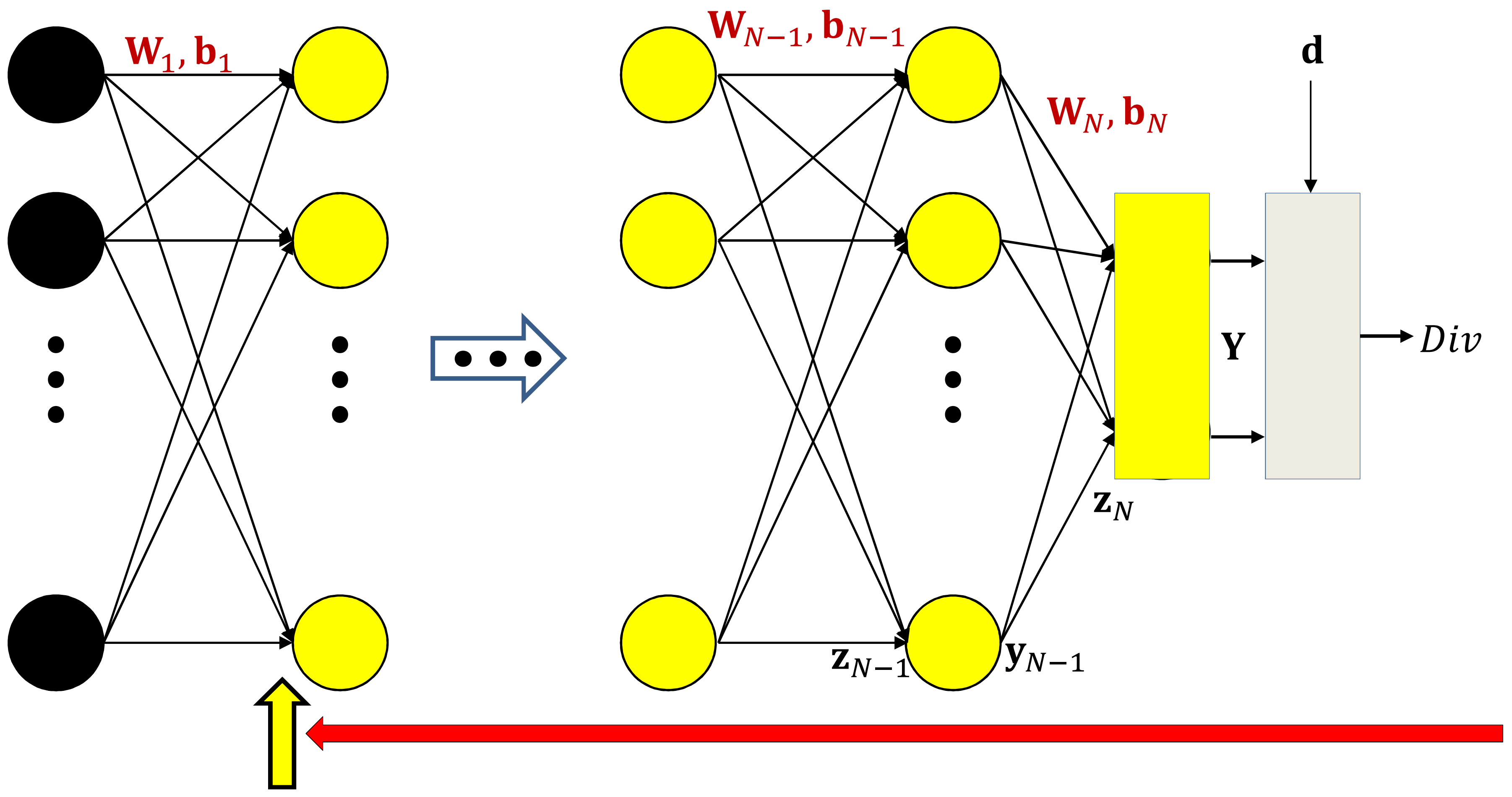

$\mathbf{W}_1, \mathbf{b}_1$
$\mathbf{W}_{N-1}, \mathbf{b}_{N-1}$
$\mathbf{W}_N, \mathbf{b}_N$
$\mathbf{d}$
$\mathbf{Y}$
Div
$\mathbf{z}_N$
$\mathbf{z}_{N-1}$
$\mathbf{y}_{N-1}$
$\nabla_{\mathbf{y}_{N-2}} Div$

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

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

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

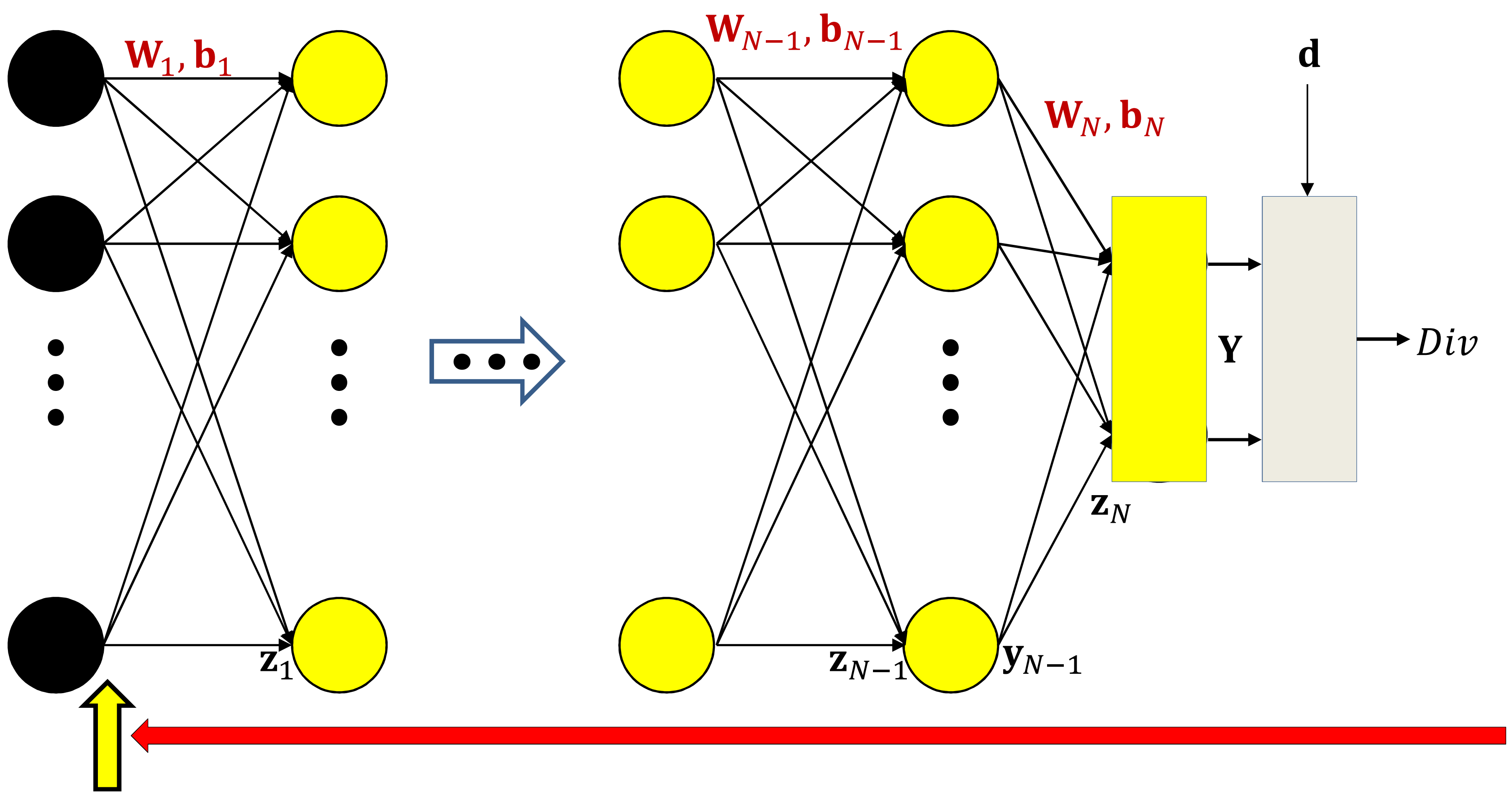
The backward pass



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

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

The backward pass



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

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

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

The Backward Pass

- Set $\mathbf{y}_N = Y, \mathbf{y}_0 = \mathbf{x}$
- Initialize: Compute $\nabla_{\mathbf{y}_N} Div = \nabla_Y Div$

- For layer k = N downto 1:
  - Compute $J_{\mathbf{y}_k}(\mathbf{z}_k)$
    - Will require intermediate values computed in the forward pass
  - Recursion:

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

  - Gradient computation:

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

The Backward Pass

- Set $\mathbf{y}_N = Y, \mathbf{y}_0 = \mathbf{x}$
- Initialize: Compute $\nabla_{\mathbf{y}_N} Div = \nabla_Y Div$

- For layer k = N downto 1:
  - Compute $J_{\mathbf{y}_k}(\mathbf{z}_k)$
    - Will require intermediate values computed in the forward pass
  - Recursion:

Note analogy to forward pass

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

  - Gradient computation:

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

For comparison: The Forward Pass

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

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

- Output:

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

Neural network training algorithm

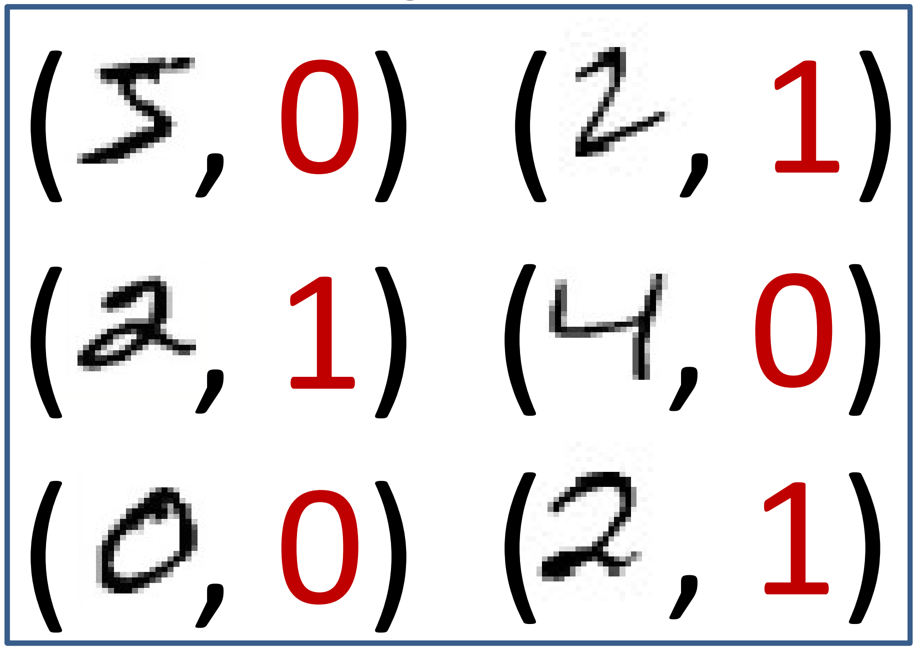
- Initialize all weights and biases $(\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \dots, \mathbf{W}_N, \mathbf{b}_N)$
- Do:
  - $Loss = 0$
  - For all k, initialize $\nabla_{\mathbf{W}_k} Loss = 0$, $\nabla_{\mathbf{b}_k} Loss = 0$
  - For all $t = 1{:}T$
    - Forward pass : Compute
   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

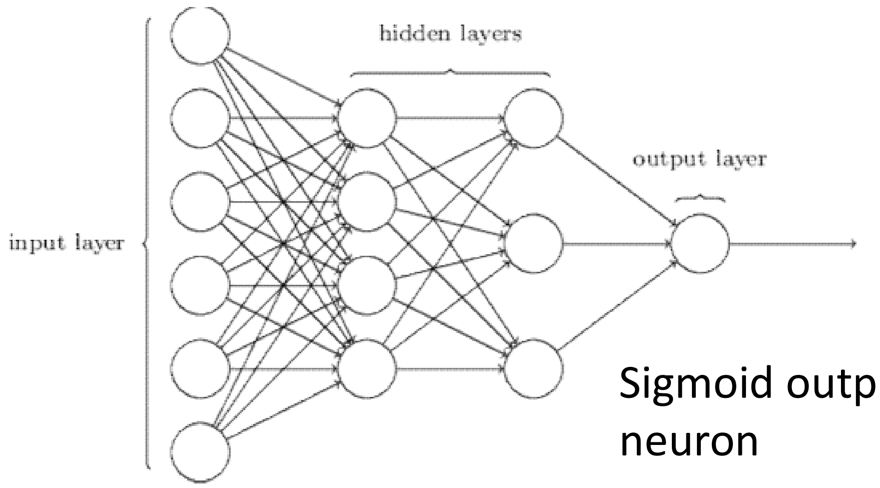
$$\mathbf{W}_k = \mathbf{W}_k - \frac{\eta}{T}\left(\nabla_{\mathbf{W}_k} Loss\right)^T; \qquad \mathbf{b}_k = \mathbf{b}_k - \frac{\eta}{T}\left(\nabla_{\mathbf{W}_k} Err\right)^T$$

- Until Err has converged

Setting up for digit recognition

Training data

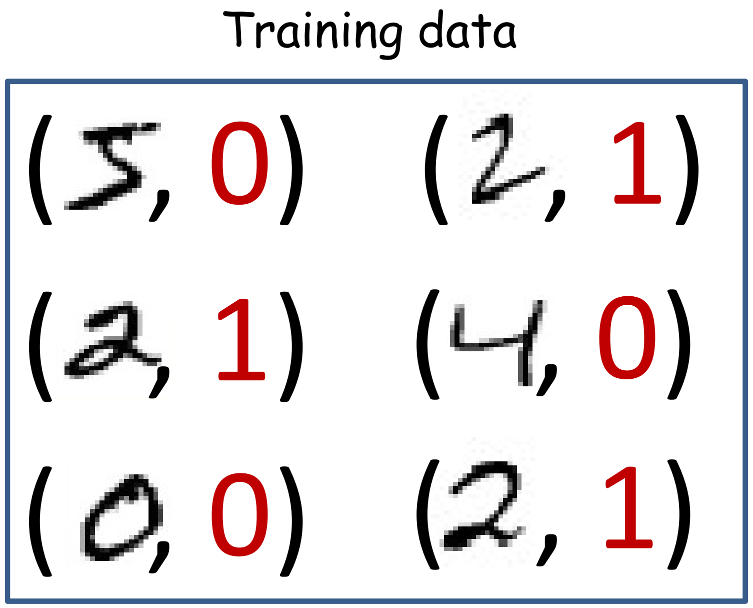


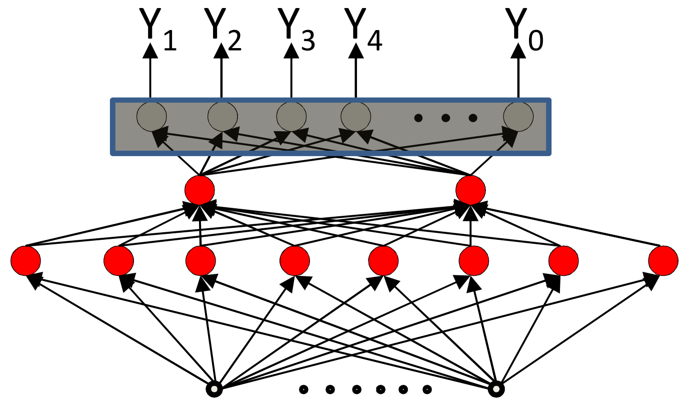


Sigmoid output neuron

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

Recognizing the digit





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

Issues

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

Onward

Onward

- Does backprop always work?
- Convergence of gradient descent
  - Rates, restrictions,
  - Hessians
  - Acceleration and Nestorov
  - Alternate approaches
- Modifying the approach: Stochastic gradients
- Speedup extensions: RMSprop, Adagrad

Does backprop do the right thing?

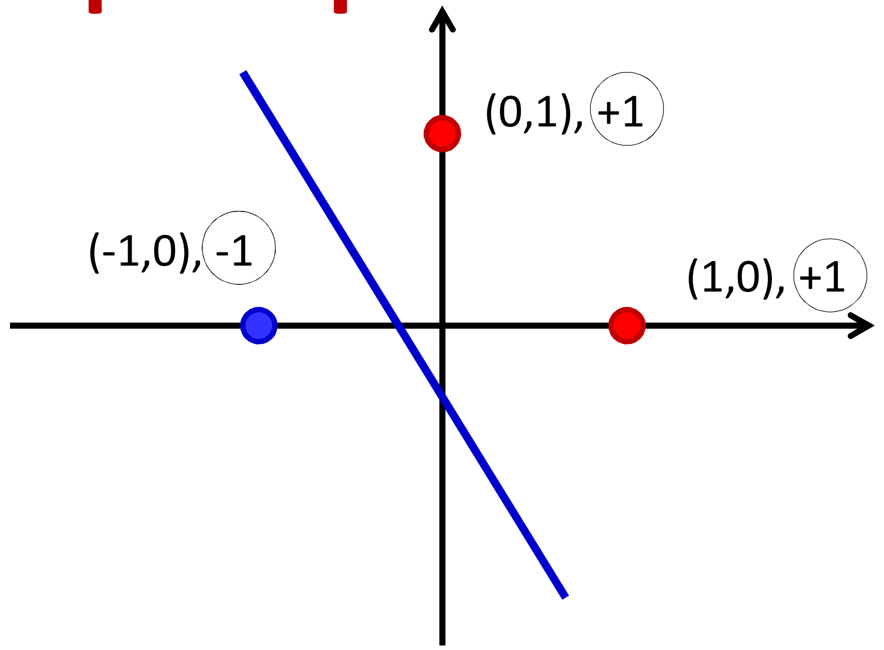
- **Is backprop always right?**
  - Assuming it actually find the global minimum of the divergence function?

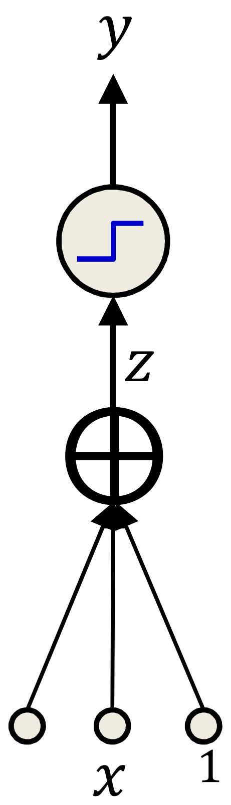
Does backprop do the right thing?

- **Is backprop always right?**
  - Assuming it actually find the global minimum of the divergence function?

- In classification problems, the classification error is a non-differentiable function of weights
- The divergence function minimized is only a *proxy* for classification error
- Minimizing divergence may not minimize classification error

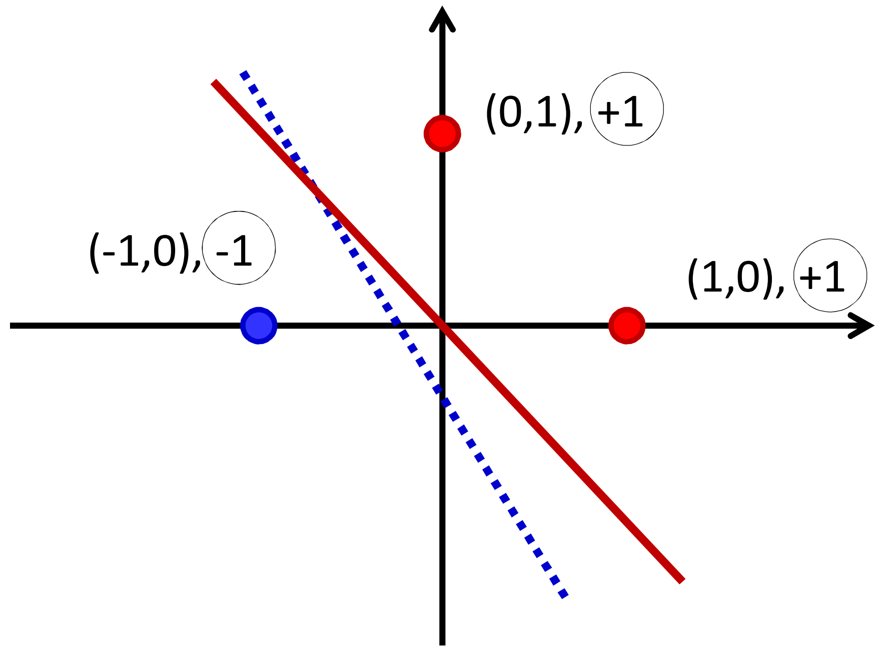
Backprop fails to separate where perceptron succeeds

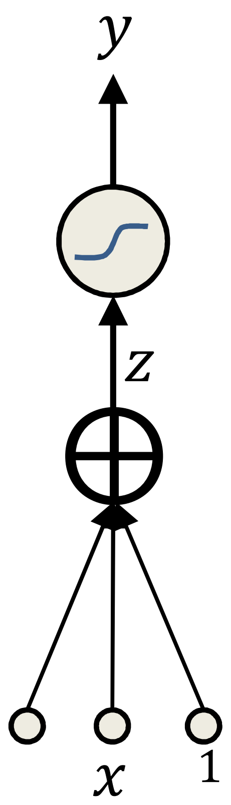




- Brady, Raghavan, Slawny, ’89
- Simple problem, 3 training instances, single neuron
- Perceptron training rule trivially find a perfect solution

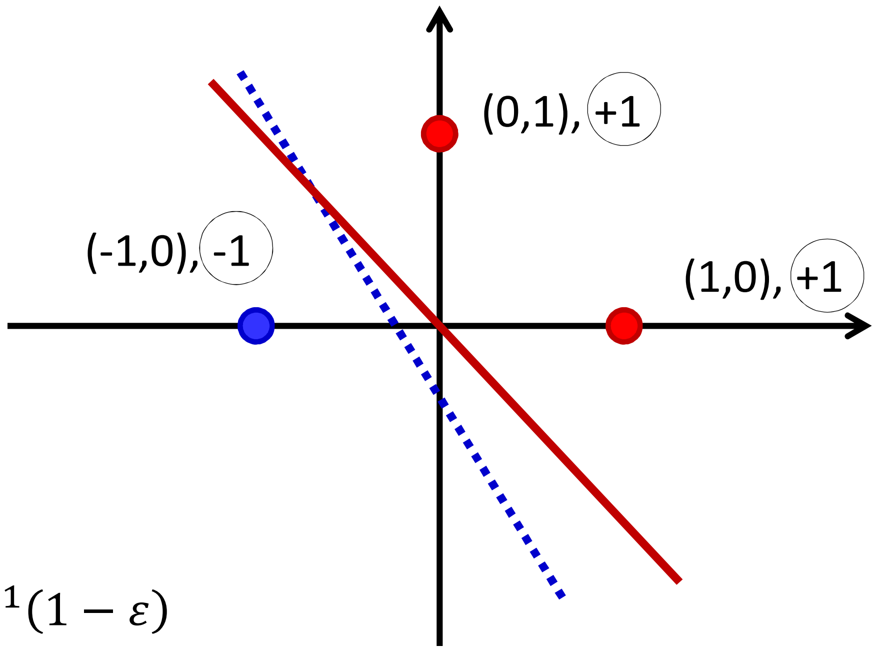
Backprop vs. Perceptron

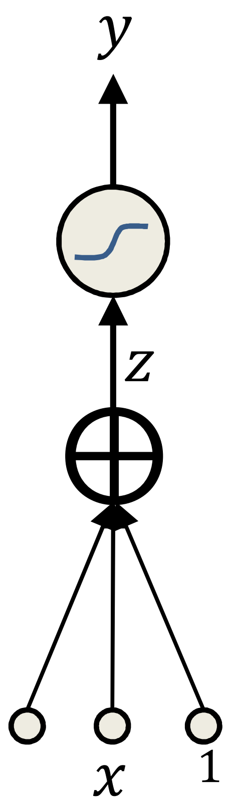




- Back propagation using logistic function and L_2 divergence $(Div = (y-d)^2)$
- Unique minimum trivially proved to exist, Preceptron rule finds it

Unique solution exists



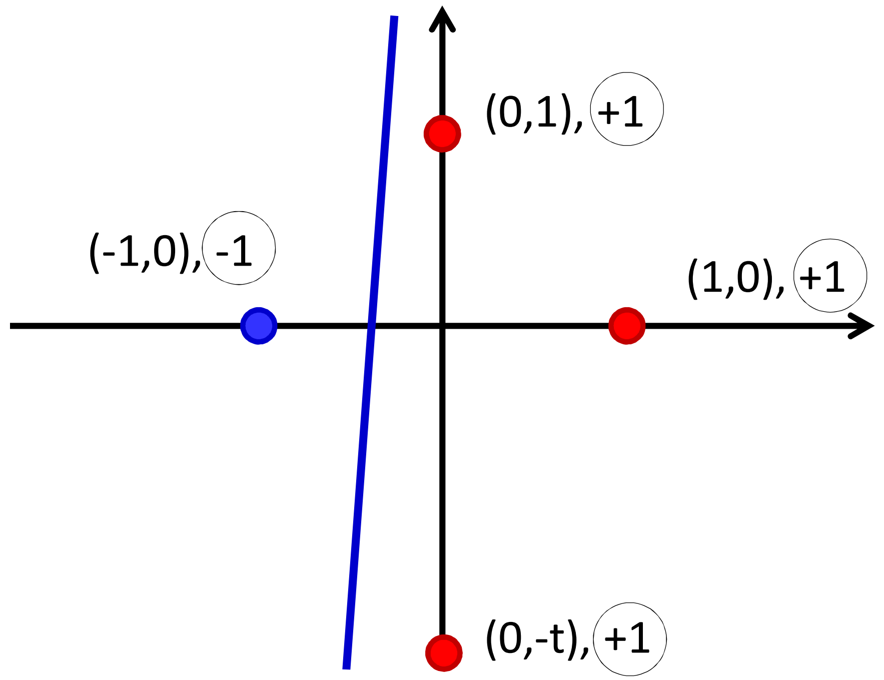


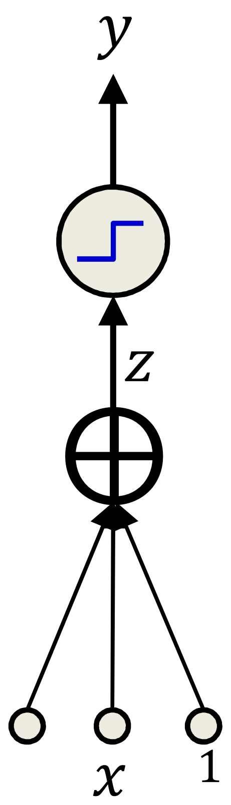
- Let $u = f^{-1}(1-\varepsilon)$
  - E.g. $u = f^{-1}(0.99)$ representing a 99% confidence in the class
- From the three points we get three independent equations:

$$w_x.1 + w_y.0 + b = u$$
$$w_x.0 + w_y.1 + b = u$$
$$w_x.-1 + w_y.0 + b = -u$$

- Unique solution $(w_x = u, w_x = u, b = 0)$ exists
  - represents a unique line regardless of the value of u

Backprop vs. Perceptron

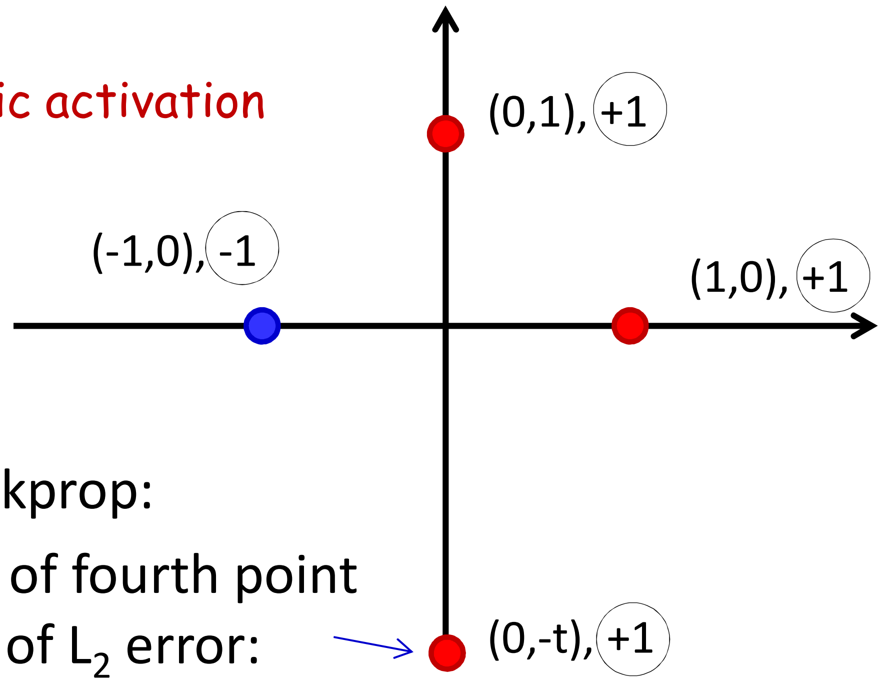


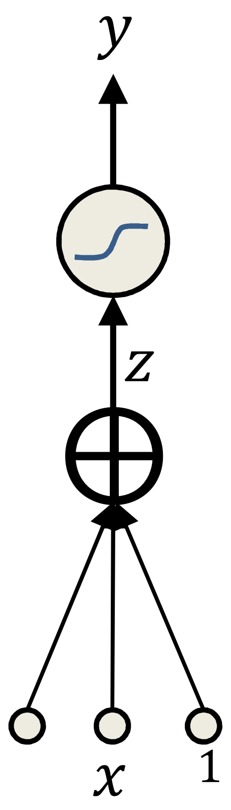


- Now add a fourth point
- t is very large (point near $-\infty$)
- Perceptron trivially finds a solution (may take t^2 iterations)

Backprop

Notation:
$y = \sigma(z)$ = logistic activation





- Consider backprop:
- Contribution of fourth point to derivative of L_2 error:

$$div_4 = \left(1 - \varepsilon - \sigma(-w_y t + b)\right)^2$$

$$\frac{d\,div_4}{dw_y} = 2\left(1 - \varepsilon - \sigma(-w_y t + b)\right)\sigma'(-w_y t + b)t$$

$$\frac{d\,div_4}{db} = -2\left(1 - \varepsilon - \sigma(-w_y t + b)\right)\sigma'(-w_y t + b)$$

1-ε is the actual *achievable* value

Backprop

Notation:
$y = \sigma(z)$ = logistic activation

$$div_4 = \left(1 - \varepsilon - \sigma(-w_y t + b)\right)^2$$

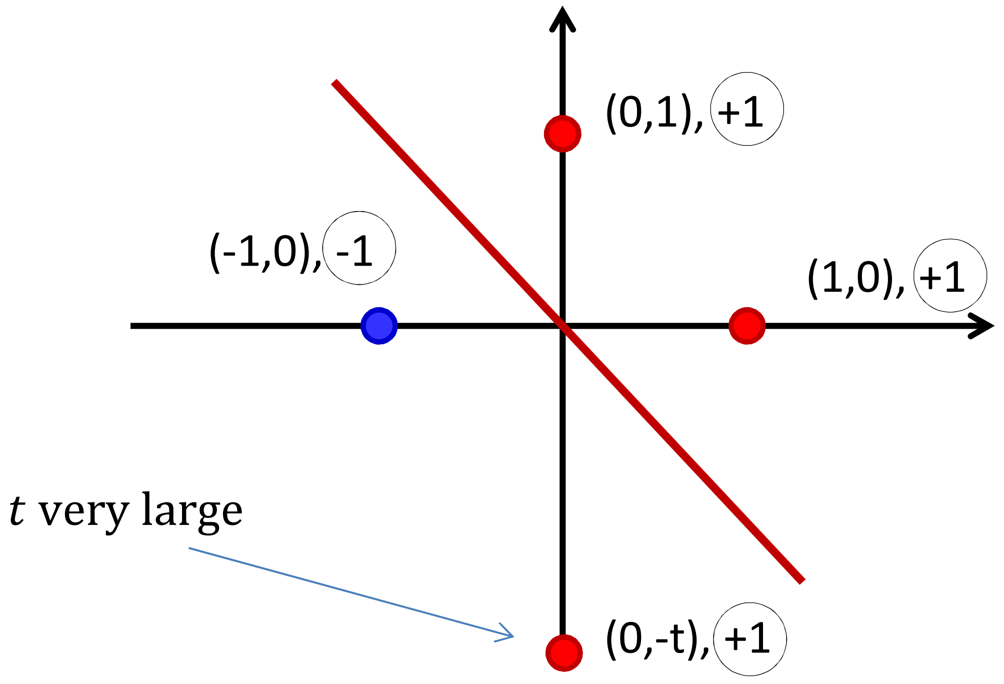
$$\frac{d\, div_4}{dw_y} = 2\left(1 - \varepsilon - \sigma(-w_y t + b)\right)\sigma'(-w_y t + b)t$$

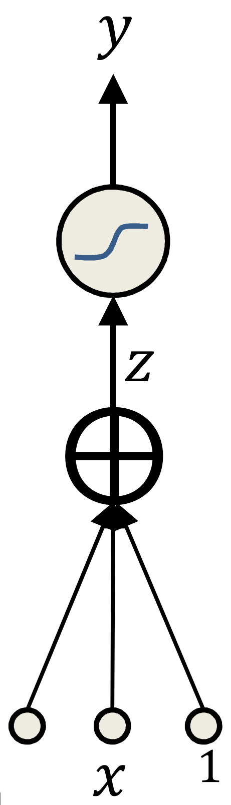
$$\frac{d\, div_4}{db} = 2\left(1 - \sigma(-w_y t + b)\right)\sigma'(-w_y t + b)t$$

- For very large positive t, $|w_y| > \epsilon$ (where $\mathbf{w} = [w_x, w_y, b]$)
- $\left(1 - \varepsilon - \sigma(-w_y t + b)\right) \to 1$ as $t \to \infty$
- $\sigma'(-w_y t + b) \to 0$ *exponentially* as $t \to \infty$
- Therefore, for very large positive t

$$\frac{d\, div_4}{dw_y} = \frac{d\, div_4}{db} = 0$$

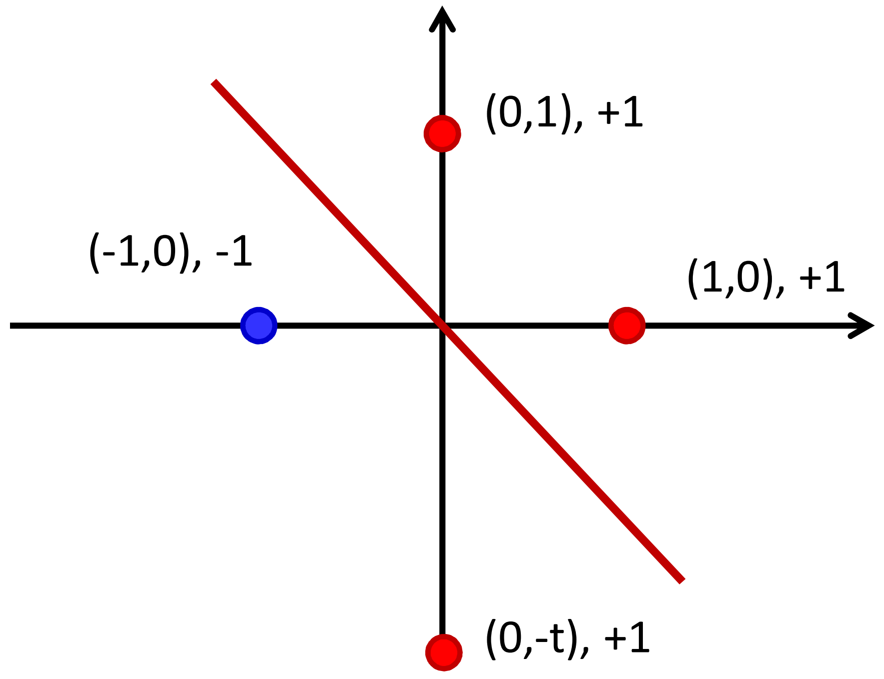
Backprop

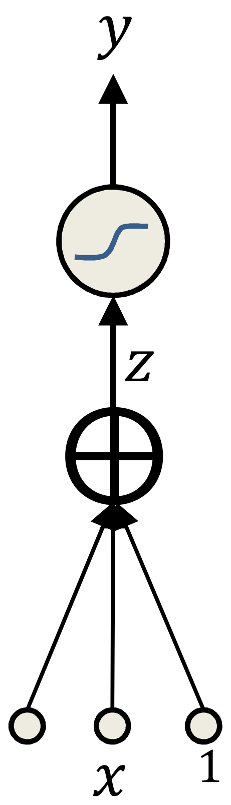




- The fourth point at $(0, -t)$ does not change the gradient of the L_2 divergence near the optimal solution for 3 points
- The optimum solution for 3 points is also a broad *local* minimum (0 gradient) for the 4-point problem!
  - Will be trivially found by backprop nearly all the time
    - Although the global minimum will separate for unbounded weights

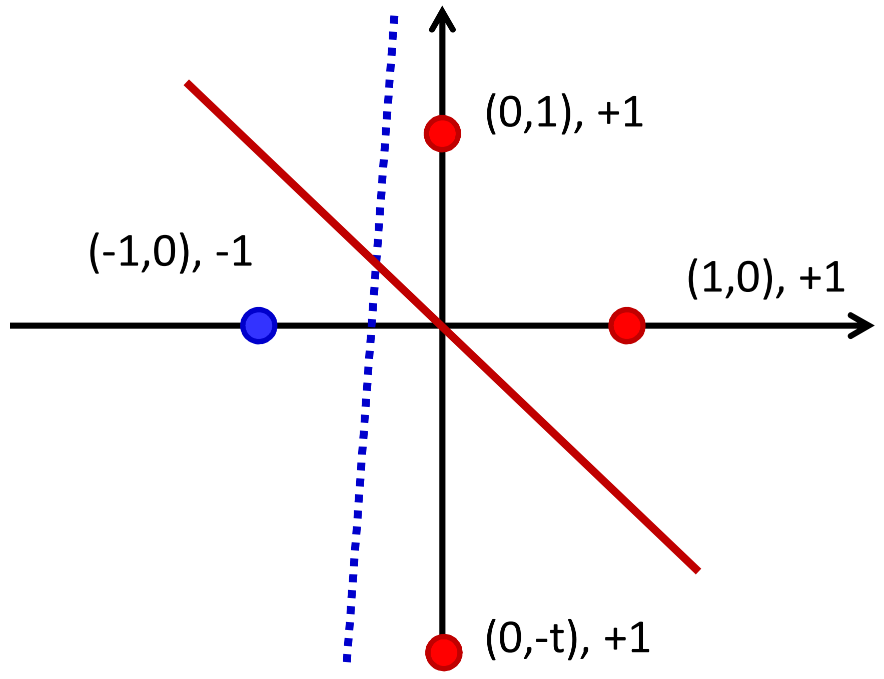
Backprop

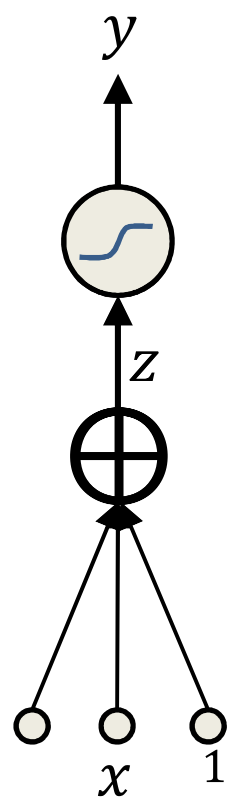




- Local optimum solution found by backprop
- Does not separate the points *even though the points are linearly separable!*

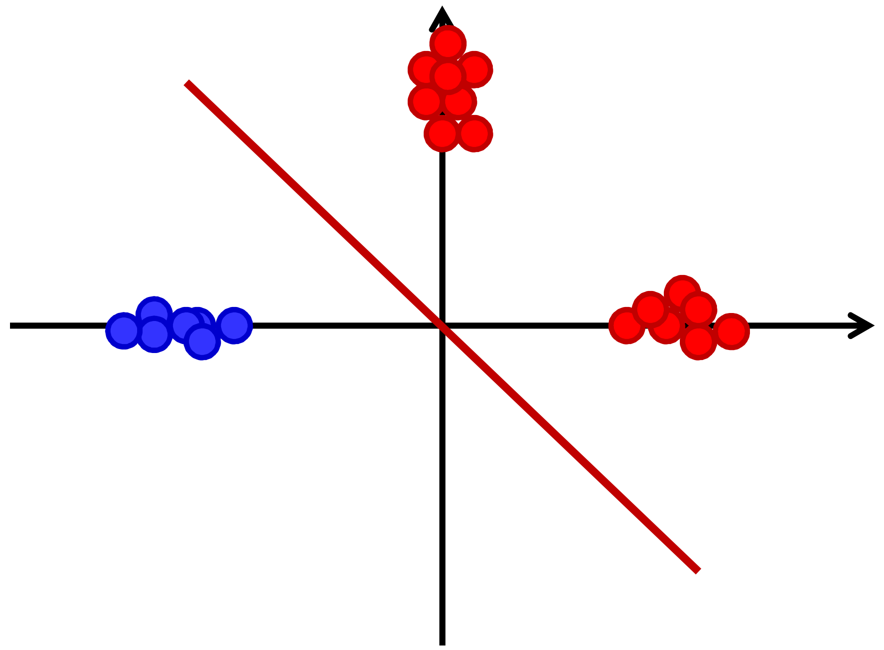
Backprop

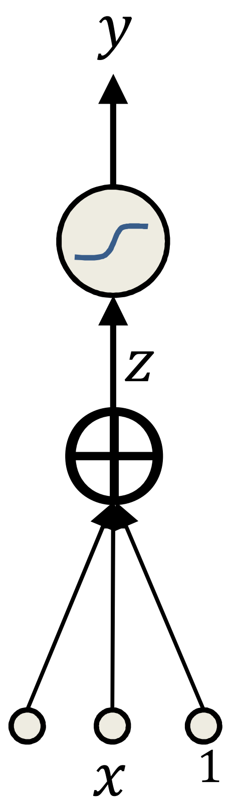




- Solution found by backprop
- Does not separate the points *even though the points are linearly separable!*
- Compare to the perceptron: *Backpropagation fails to separate where the perceptron succeeds*

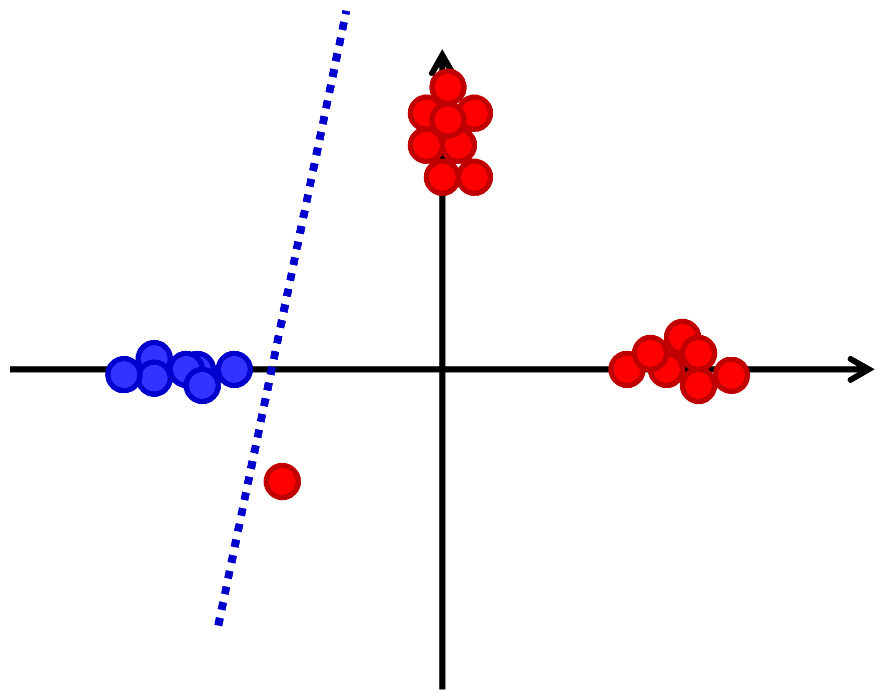
Backprop fails to separate where perceptron succeeds

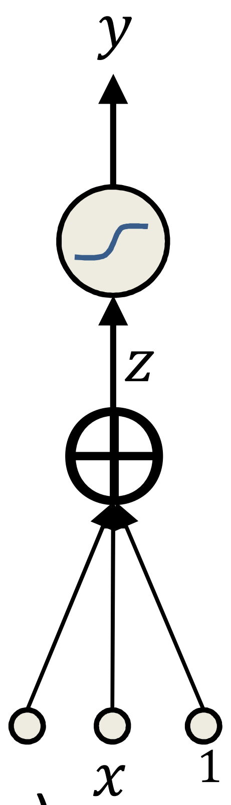




- Brady, Raghavan, Slawny, ’89
- *Several* linearly separable training examples
- Simple setup: both backprop and perceptron algorithms find solutions

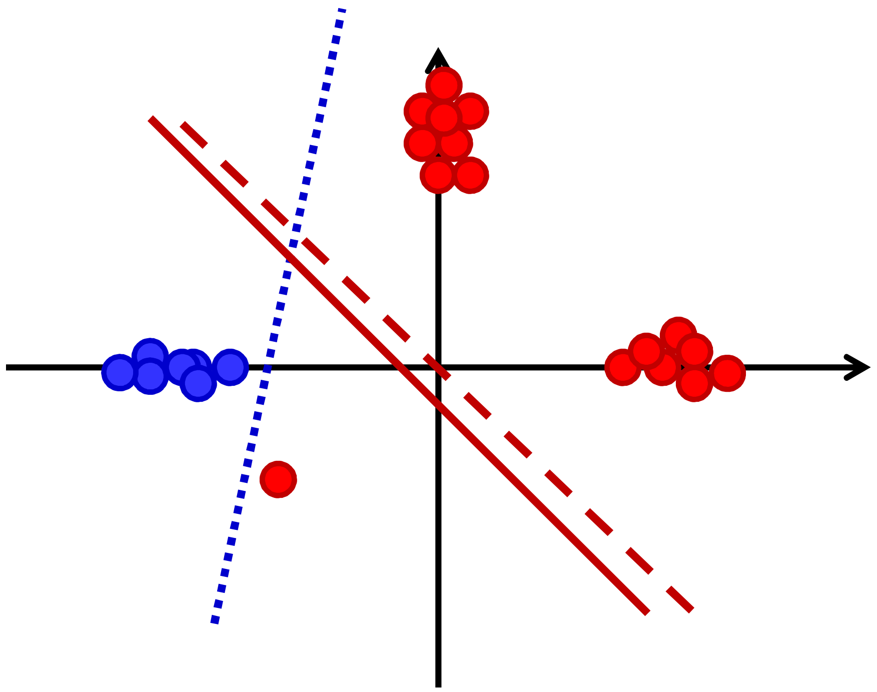
A more complex problem

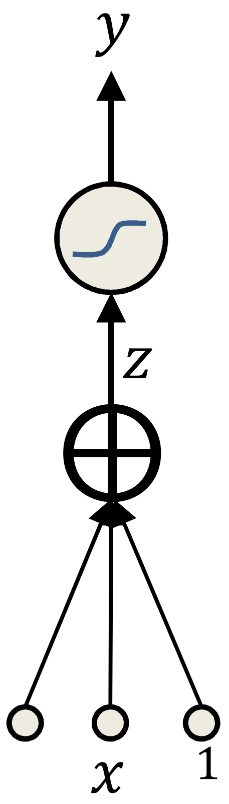




- Adding a “spoiler” (or a small number of spoilers)
  – Perceptron finds the linear separator,

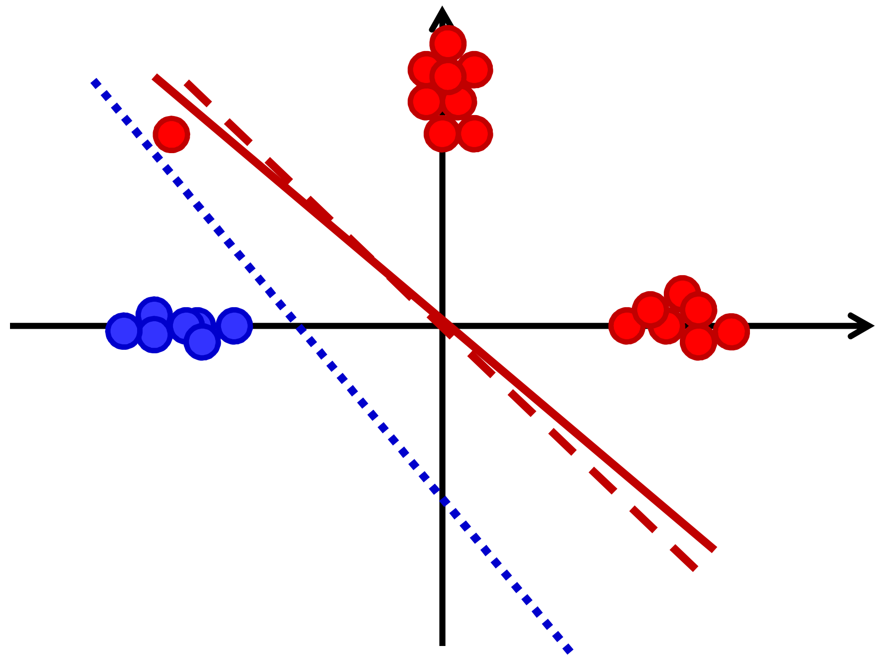
A more complex problem

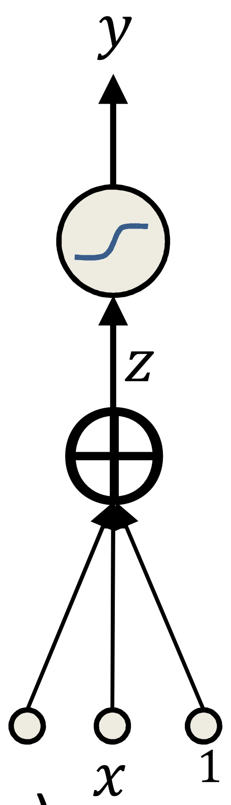




- Adding a “spoiler” (or a small number of spoilers)
  - Perceptron finds the linear separator,
  - Backprop does not find a separator
    - A single additional input does not change the loss function significantly
      - **Assuming weights are constrained to be bounded**

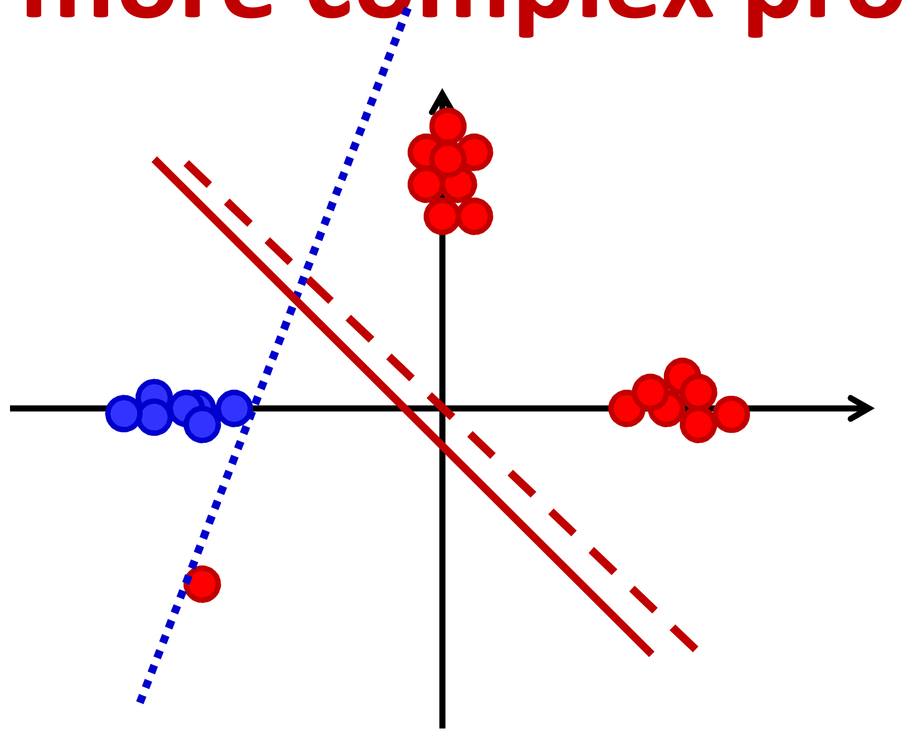
A more complex problem

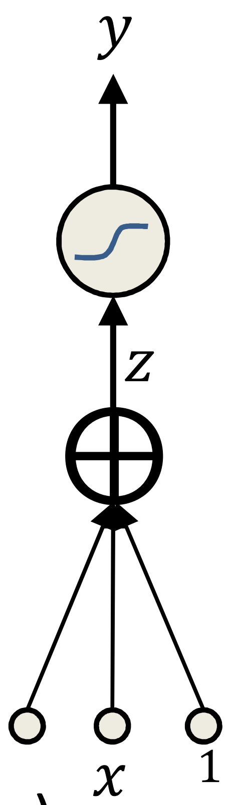




- Adding a “spoiler” (or a small number of spoilers)
  - Perceptron finds the linear separator,
  - For bounded w, backprop does not find a separator
    - A single additional input does not change the loss function significantly

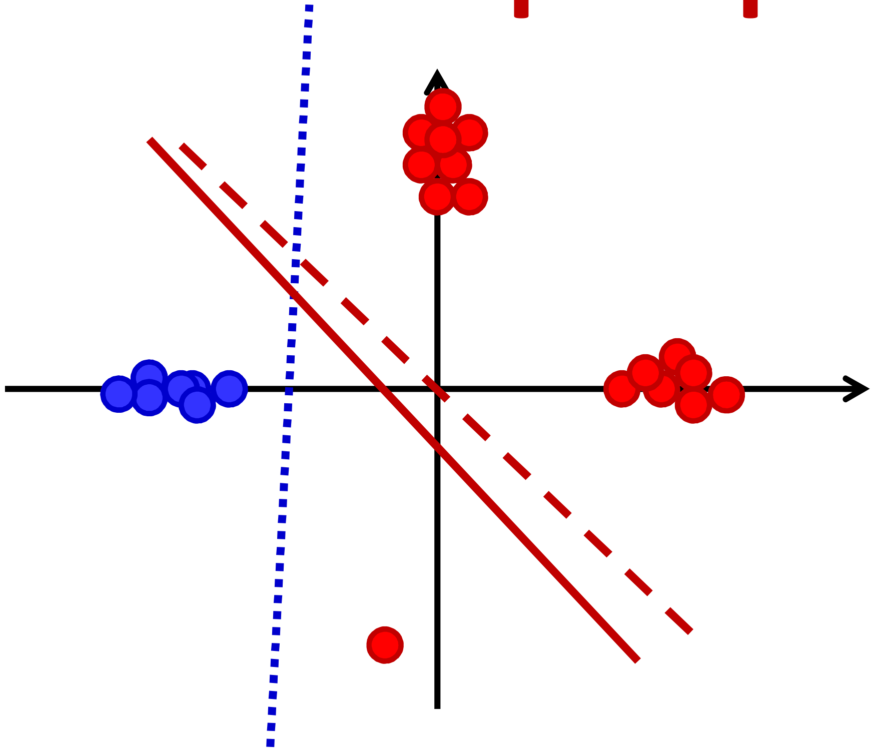
A more complex problem

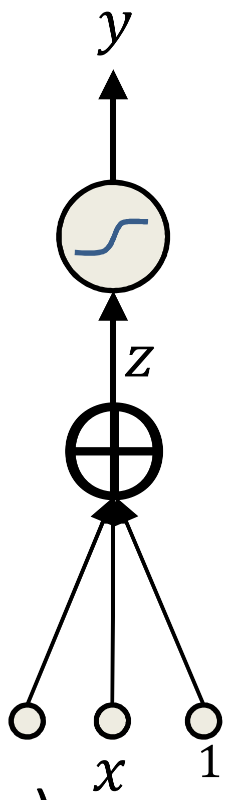




- Adding a "spoiler" (or a small number of spoilers)
  - Perceptron finds the linear separator,
  - For bounded w, backprop does not find a separator
    - A single additional input does not change the loss function significantly

A more complex problem



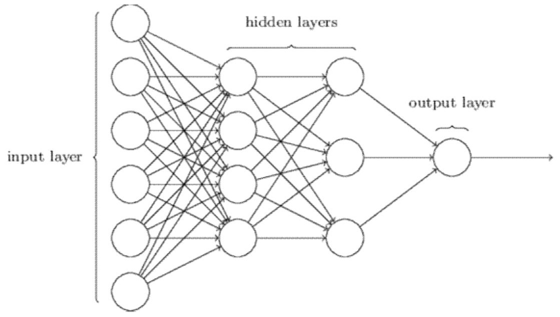


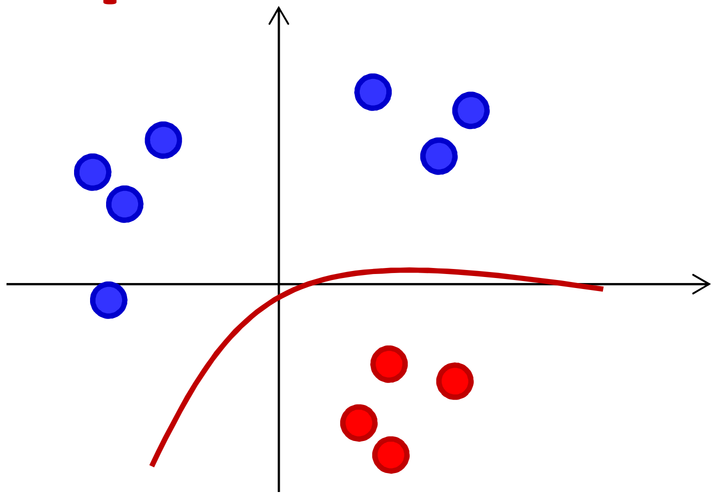
- Adding a “spoiler” (or a small number of spoilers)
  - Perceptron finds the linear separator,
  - For bounded w, Backprop does not find a separator
    - A single additional input does not change the loss function significantly

So what is happening here?

- The perceptron may change greatly upon adding just a single new training instance
  - But it fits the training data well
  - The perceptron rule has *low bias*
    - Makes no errors if possible
  - But high variance
    - Swings wildly in response to small changes to input
- Backprop is minimally changed by new training instances
  - Prefers consistency over perfection
  - It is a *low-variance* estimator, at the potential cost of bias

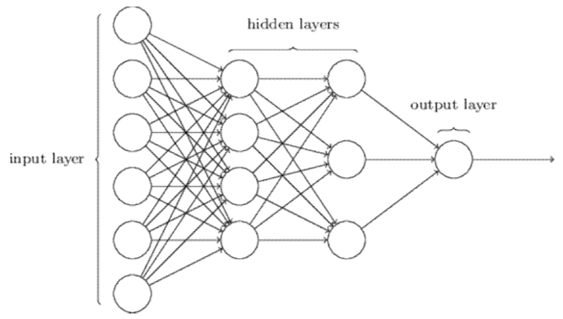
Backprop fails to separate even when possible

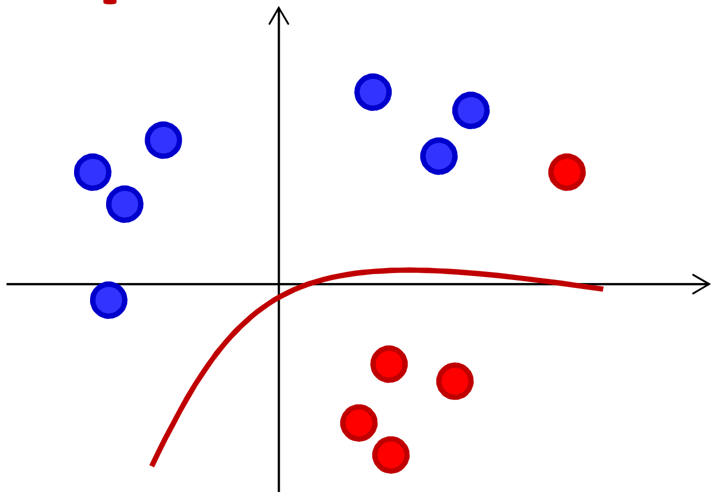




- This is not restricted to single perceptrons
- In an MLP the lower layers "learn a representation" that enables linear separation by higher layers
  - More on this later
- Adding a few "spoilers" will not change their behavior

Backprop fails to separate even when possible





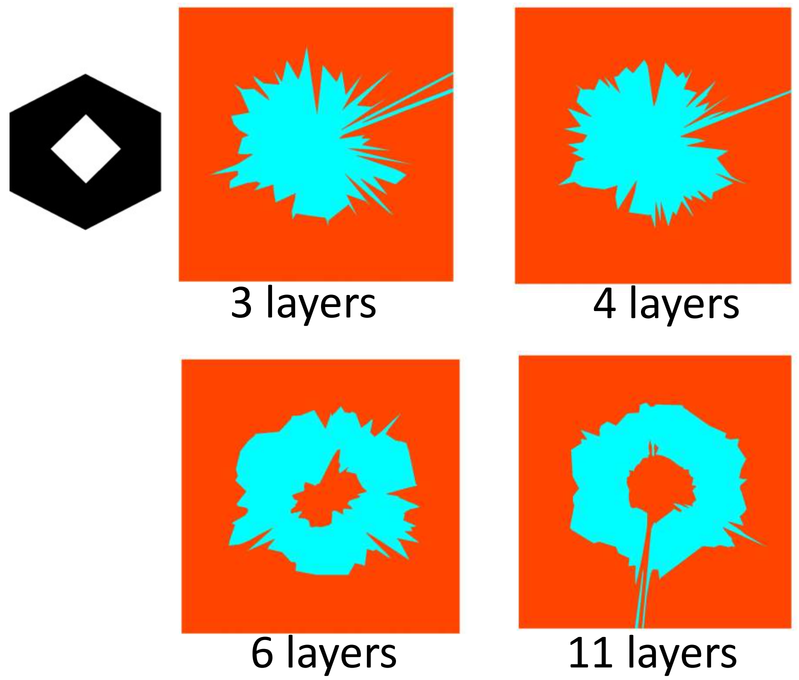
- This is not restricted to single perceptrons
- In an MLP the lower layers "learn a representation" that enables linear separation by higher layers
  - More on this later
- Adding a few "spoilers" will not change their behavior

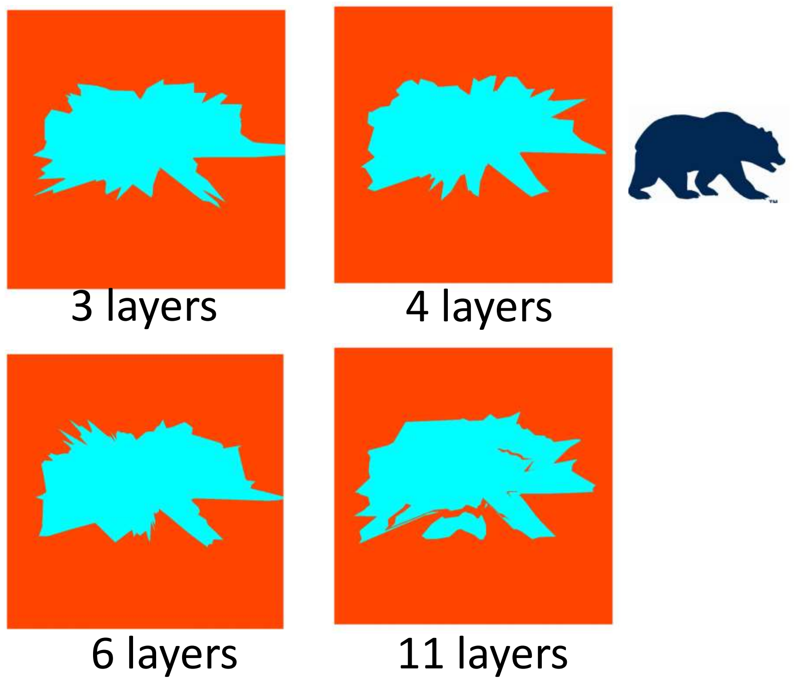
Backpropagation

- Backpropagation will often not find a separating solution ***even though the solution is within the class of functions learnable by the network***

- This is because the separating solution is not a feasible optimum for the loss function
- One resulting benefit is that a backprop-trained neural network classifier has lower variance than an optimal classifier for the training data

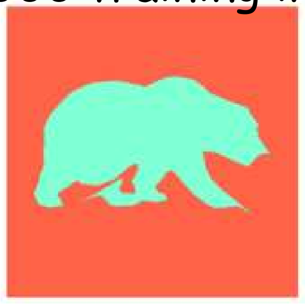
Variance and Depth





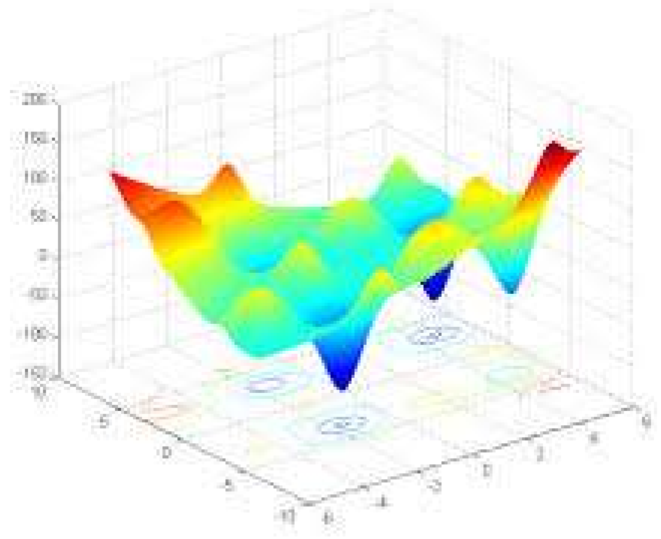
- Dark figures show desired decision boundary (2D)
  - 1000 training points, 660 hidden neurons
  - Network heavily overdesigned even for shallow nets
- **Anecdotal: Variance decreases with**
  - Depth
  - Data

10000 training instances



The Loss Surface

- The example (and statements) earlier assumed the loss objective had a single global optimum that could be found
  - Statement about variance is assuming global optimum

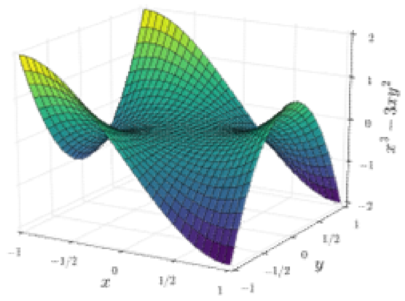


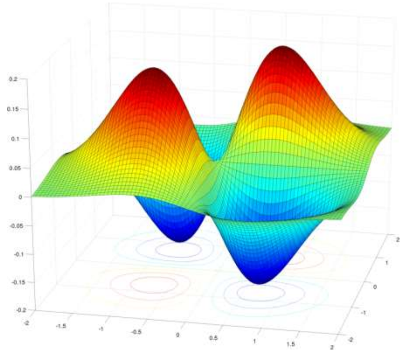
- What about local optima

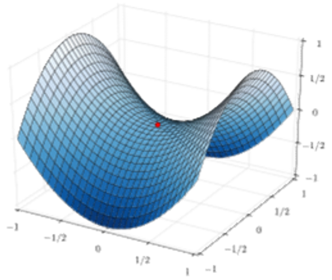
The Loss Surface

- **Popular hypothesis**:
  - In large networks, saddle points are far more common than local minima
    - Frequency exponential in network size
  - Most local minima are equivalent
    - And close to global minimum
  - This is not true for small networks

- **Saddle point:** A point where
  - The slope is zero
  - The surface increases in some directions, but decreases in others
    - Some of the Eigenvalues of the Hessian are positive; others are negative
  - Gradient descent algorithms often get "stuck" in saddle points







The Controversial Loss Surface

- **Baldi and Hornik (89), "***Neural Networks and Principal Component Analysis: Learning from Examples Without Local Minima*" : An MLP with a *single* hidden layer has only saddle points and no local Minima

- **Dauphin et. al (2015),** "*Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*" : An exponential number of saddle points in large networks

- **Chomoranksa et. al (2015)**, "*The loss surface of multilayer networks*" : For large networks, most local minima lie in a band and are equivalent
  - Based on analysis of spin glass models

- **Swirscz et. al. (2016)**, "Local minima in training of deep networks", In networks of finite size, trained on finite data, you *can* have horrible local minima
- Watch this space…

Story so far

- Neural nets can be trained via gradient descent that minimizes a loss function
- Backpropagation can be used to derive the derivatives of the loss
- Backprop *is not guaranteed* to find a “true” solution, even if it exists, and lies within the capacity of the network to model
  - The optimum for the loss function may not be the “true” solution
- For large networks, the loss function may have a large number of unpleasant saddle points
  - Which backpropagation may find

Convergence

- In the discussion so far we have assumed the training arrives at a local minimum

- Does it always converge?
- How long does it take?

- Hard to analyze for an MLP, but we can look at the problem through the lens of convex optimization

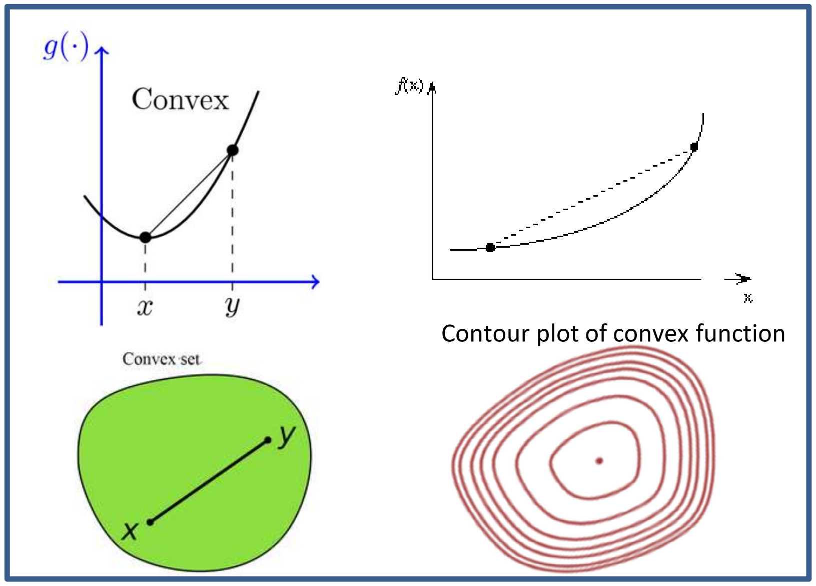
A quick tour of (convex) optimization

The streetlight effect is a type of observational bias where people only look for whatever they are searching by looking where it is easiest

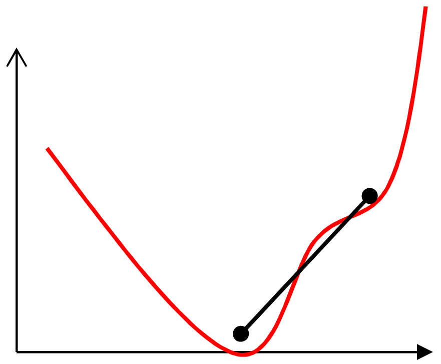
"I'm searching for my keys."

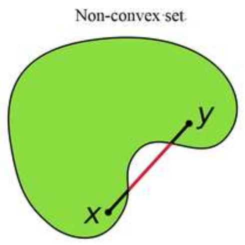
Convex Loss Functions

- A surface is "convex" if it is continuously curving upward
  - We can connect any two points above the surface without intersecting it
  - Many mathematical definitions that are equivalent



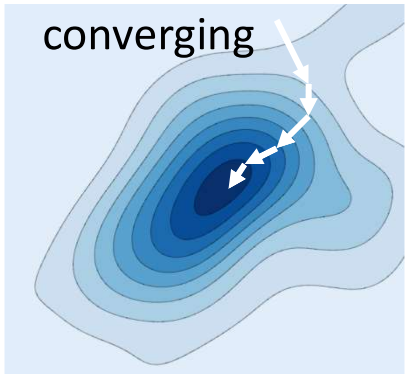
- Caveat: Neural network loss surface is generally not convex
  - Streetlight effect

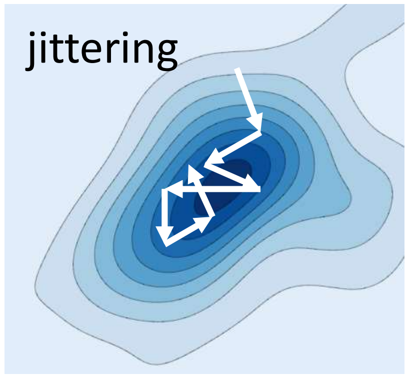


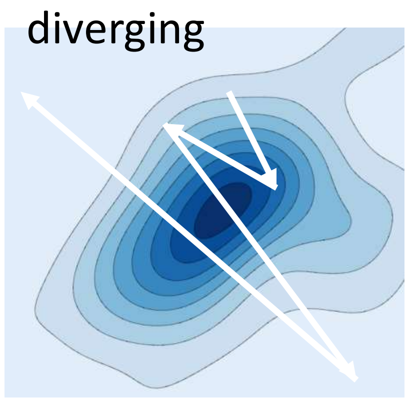


Convergence of gradient descent

- An iterative algorithm is said to *converge* to a solution if the value updates arrive at a fixed point
  - Where the gradient is 0 and further updates do not change the estimate
- The algorithm may not actually converge
  - It may jitter around the local minimum
  - It may even diverge
- Conditions for convergence?





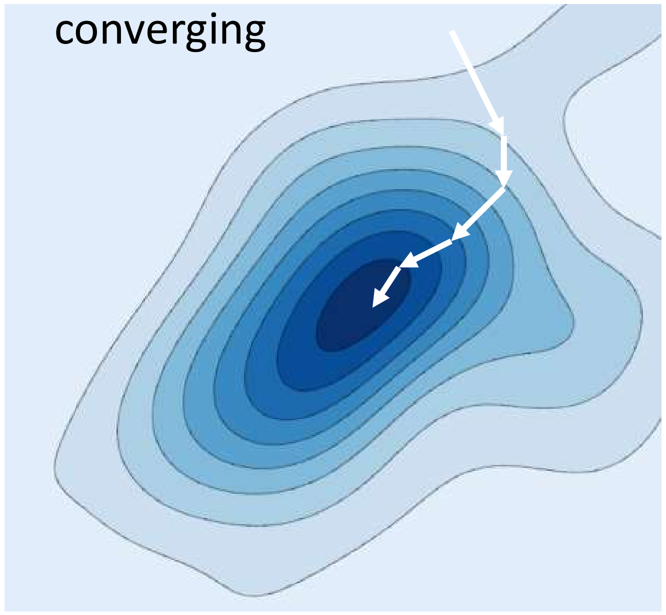


Convergence and convergence rate

- Convergence rate: How fast the iterations arrive at the solution
- Generally quantified as

$$R = \frac{\left|f\left(x^{(k+1)}\right) - f(x^*)\right|}{\left|f\left(x^{(k)}\right) - f(x^*)\right|}$$

  - $x^{(k+1)}$is the k-th iteration
  - $x^*$is the optimal value of x



- If R is a constant (or upper bounded), the convergence is *linear*
  - In reality, its arriving at the solution exponentially fast

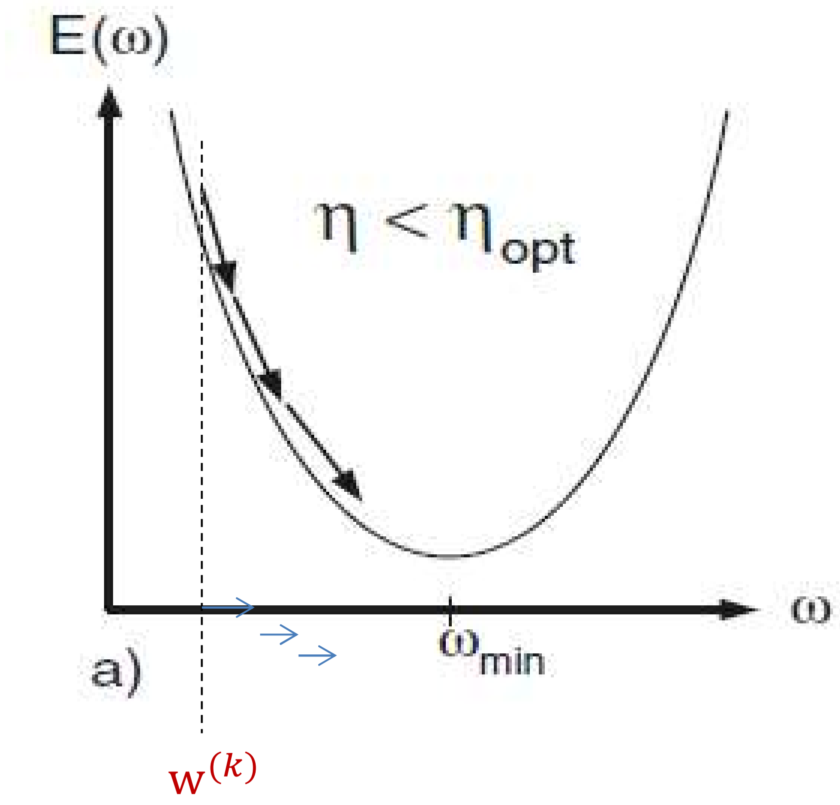
$$\left|f\left(x^{(k)}\right) - f(x^*)\right| = c^k \left|f\left(x^{(0)}\right) - f(x^*)\right|$$

Convergence for quadratic surfaces

$$Minimize\ E = \frac{1}{2}aw^2 + bw + c$$

$$\mathrm{w}^{(k+1)} = \mathrm{w}^{(k)} - \eta \frac{dE\left(\mathrm{w}^{(k)}\right)}{d\mathrm{w}}$$

Gradient descent with fixed step size η to estimate *scalar* parameter w



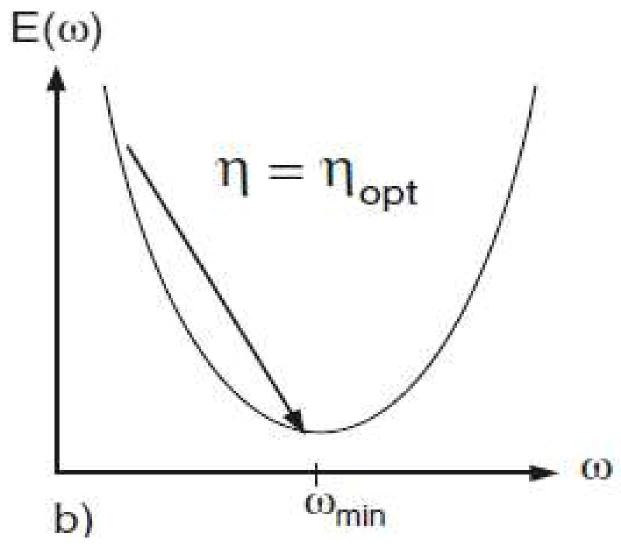
- Gradient descent to find the optimum of a quadratic, starting from $\mathrm{w}^{(k)}$

- Assuming fixed step size η
- What is the optimal step size η to get there fastest?

Convergence for quadratic surfaces

$$\boxed{E = \frac{1}{2}aw^2 + bw + c}$$

$$\boxed{\mathrm{w}^{(k+1)} = \mathrm{w}^{(k)} - \eta \frac{dE\left(\mathrm{w}^{(k)}\right)}{d\mathrm{w}}}$$



- Any quadratic objective can be written as

$$E(w) = E\left(\mathrm{w}^{(k)}\right) + E'\left(\mathrm{w}^{(k)}\right)\left(w - \mathrm{w}^{(k)}\right) + \tfrac{1}{2}E''\left(\mathrm{w}^{(k)}\right)\left(w - \mathrm{w}^{(k)}\right)^2$$

  – Taylor expansion

- Minimizing w.r.t w, we get (Newton's method)

$$w_{min} = \mathrm{w}^{(k)} - E''\left(\mathrm{w}^{(k)}\right)^{-1} E'\left(\mathrm{w}^{(k)}\right)$$

- Note:

$$\frac{dE\left(\mathrm{w}^{(k)}\right)}{d\mathrm{w}} = E'\left(\mathrm{w}^{(k)}\right)$$

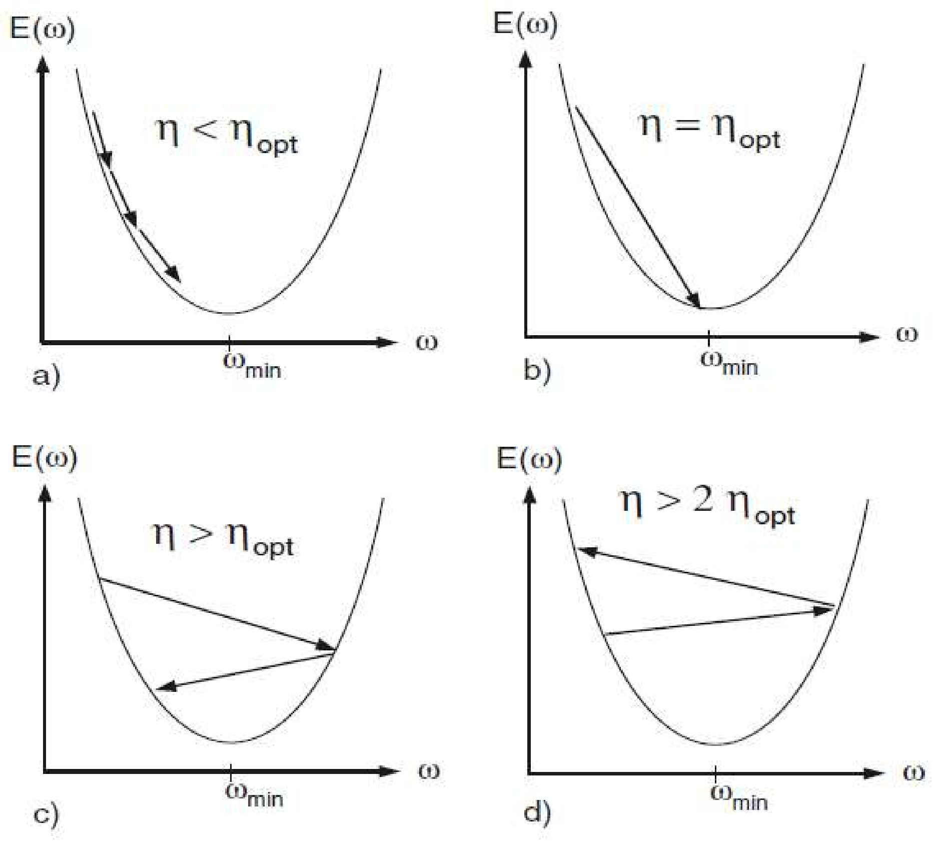
- Comparing to the gradient descent rule, we see that we can arrive at the optimum in a single step using the optimum step size

$$\boxed{\eta_{opt} = E''\left(\mathrm{w}^{(k)}\right)^{-1} = \boldsymbol{a^{-1}}}$$

With non-optimal step size

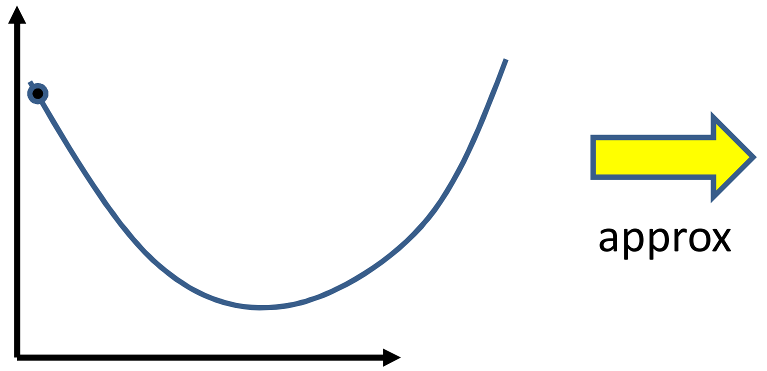
$$\mathrm{w}^{(k+1)} = \mathrm{w}^{(k)} - \eta \frac{dE\left(\mathrm{w}^{(k)}\right)}{d\mathrm{w}}$$

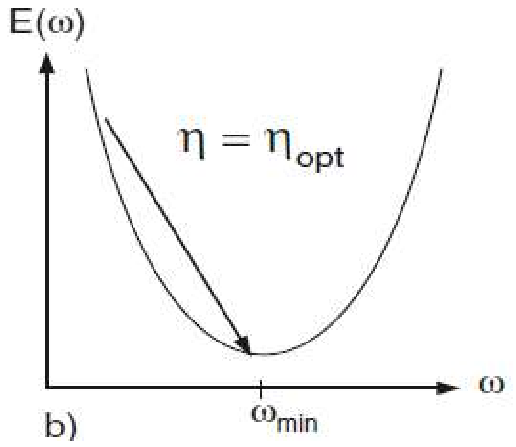
Gradient descent with fixed step size η to estimate scalar parameter w



- For $\eta < \eta_{opt}$ the algorithm will converge monotonically
- For $2\eta_{opt} > \eta > \eta_{opt}$ we have oscillating convergence
- For $\eta > 2\eta_{opt}$ we get divergence

For generic differentiable convex objectives





- Any differentiable convex objective $E(w)$ can be approximated as

$$E \approx E\left(\mathrm{w}^{(k)}\right) + \left(w - \mathrm{w}^{(k)}\right)\frac{dE\left(\mathrm{w}^{(k)}\right)}{dw} + \frac{1}{2}\left(w - \mathrm{w}^{(k)}\right)^2 \frac{d^2E\left(\mathrm{w}^{(k)}\right)}{dw^2} + \cdots$$

  - Taylor expansion

- Using the same logic as before, we get (Newton's method)

$$\eta_{opt} = \left(\frac{d^2E\left(\mathrm{w}^{(k)}\right)}{dw^2}\right)^{-1}$$

- We can get divergence if $\eta \geq 2\eta_{opt}$

For functions of *multivariate* inputs

$E = g(\mathbf{w})$, $\mathbf{w}$ is a vector $\mathbf{w} = [w_1, w_2, \ldots, w_N]$

- Consider a simple quadratic convex (paraboloid) function

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c$$

  - Since $E^T = E$ (E is scalar), $\mathbf{A}$ can always be made symmetric
    - For **convex** E, $\mathbf{A}$ is always positive definite, and has positive eigenvalues

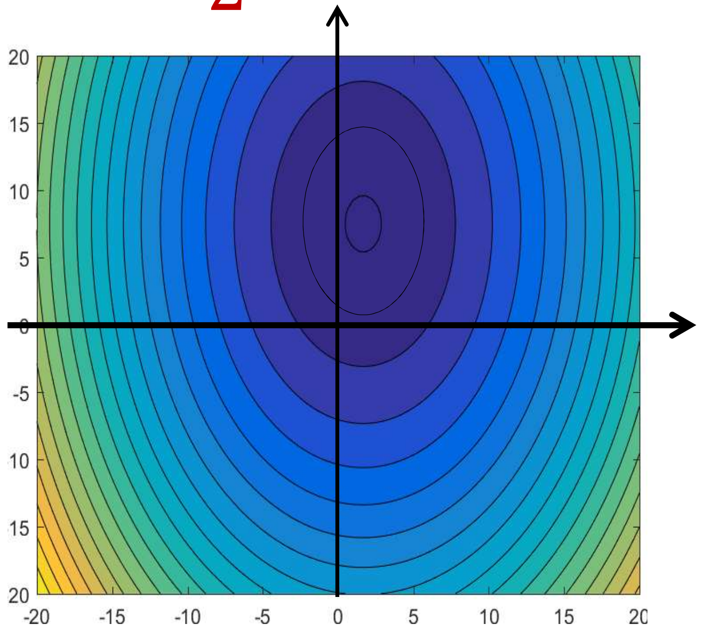
- When $\mathbf{A}$ is diagonal:

$$E = \frac{1}{2}\sum_i \left(a_{ii}w_i^2 + b_i w_i\right) + c$$

  - The w_is are *uncoupled*
  - For *convex* (paraboloid) E, the a_{ii} values are all positive
  - Just an sum of N independent quadratic functions

Multivariate Quadratic with Diagonal $\mathbf{A}$

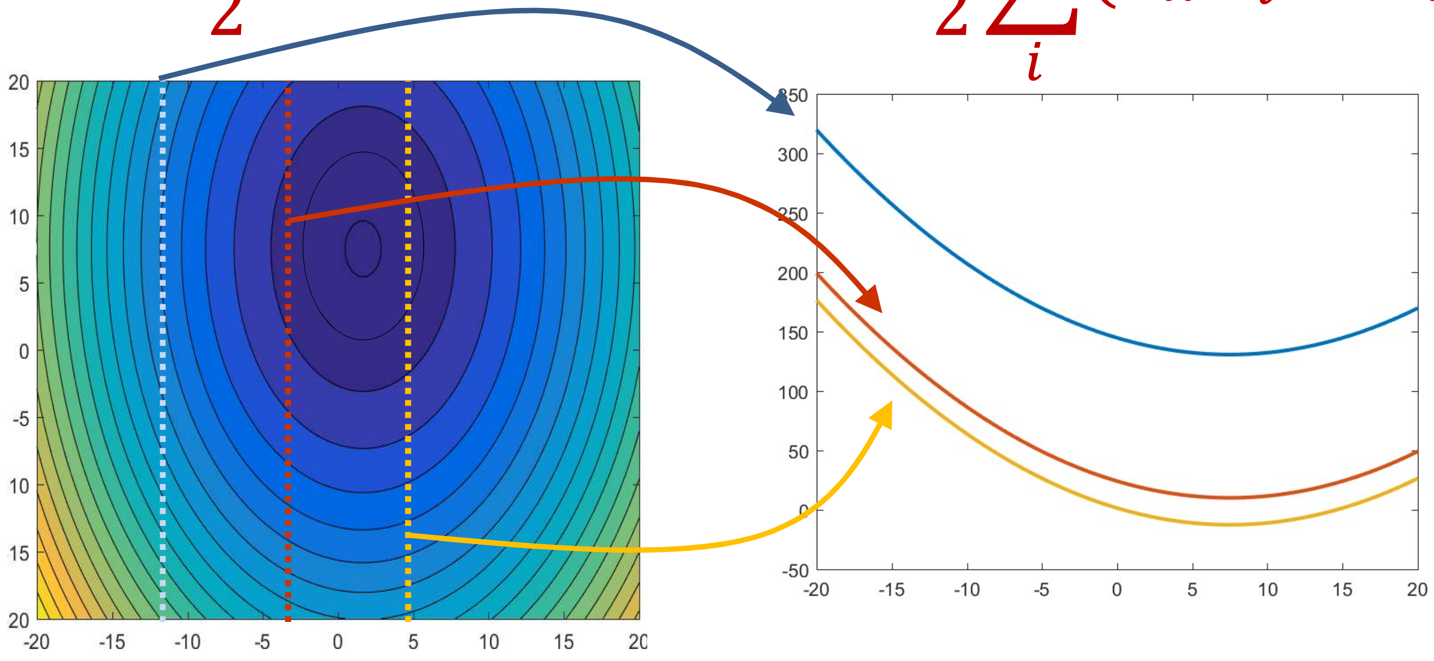
$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c = \frac{1}{2}\sum_i \left(a_{ii}w_i^2 + b_i w_i\right) + c$$



- Equal-value contours will be parallel to the axis

Multivariate Quadratic with Diagonal A

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c = \frac{1}{2}\sum_i \left(a_{ii}w_i^2 + b_i w_i\right) + c$$

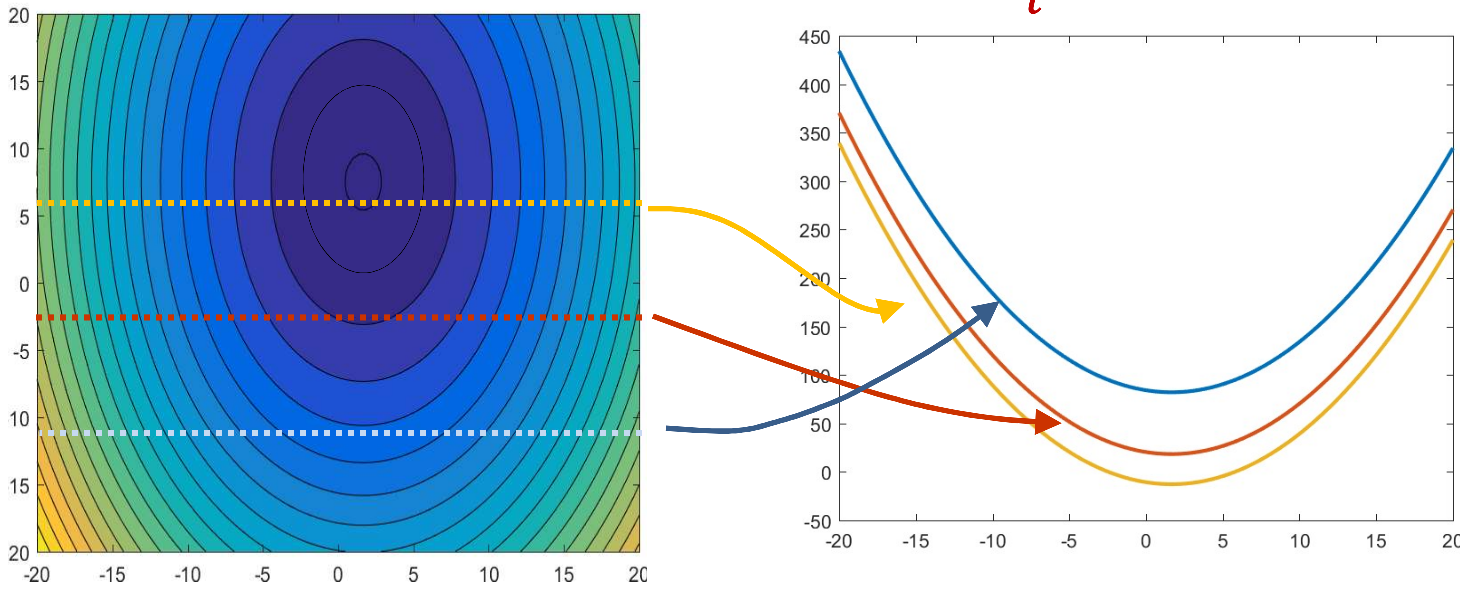


- Equal-value contours will be parallel to the axis
  - All “slices” parallel to an axis are shifted versions of one another

$$E = \frac{1}{2}a_{ii}w_i^2 + b_i w_i + c + C(\neg w_i)$$

Multivariate Quadratic with Diagonal A

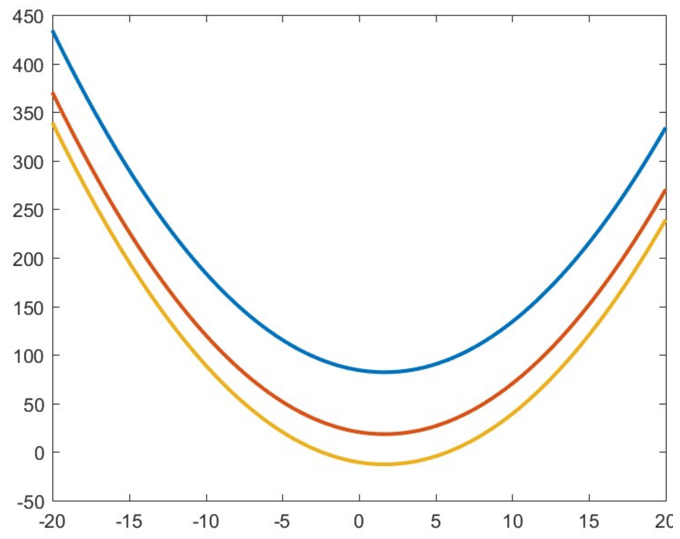
$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c = \frac{1}{2}\sum_i \left(a_{ii}w_i^2 + b_iw_i\right) + c$$

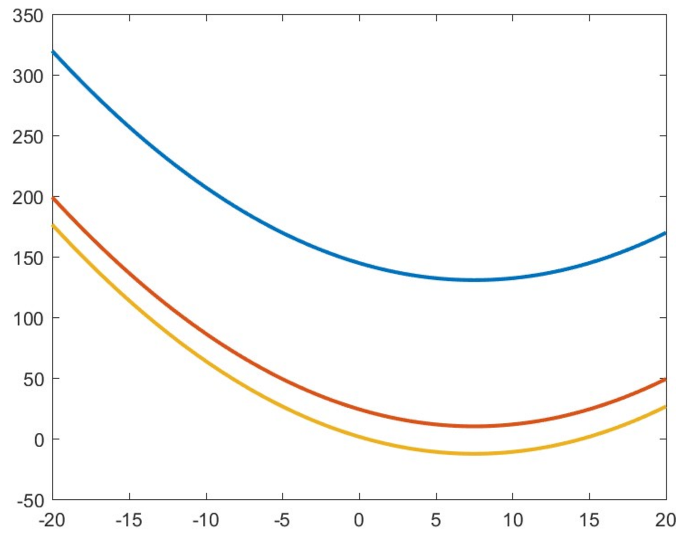


- Equal-value contours will be parallel to the axis
  - All "slices" parallel to an axis are shifted versions of one another

$$E = \frac{1}{2}a_{ii}w_i^2 + b_iw_i + c + C(\neg w_i)$$

"Descents" are uncoupled



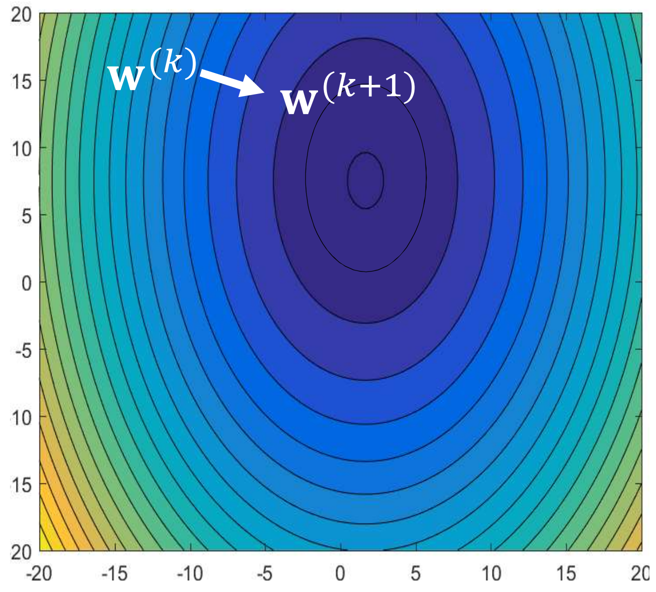


$$E = \frac{1}{2}a_{11}w_1^2 + b_1w_1 + c + C(\neg w_1) \qquad E = \frac{1}{2}a_{22}w_2^2 + b_2w_2 + c + C(\neg w_2)$$

$$\eta_{1,opt} = a_{11}^{-1} \qquad \eta_{2,opt} = a_{22}^{-1}$$

- The optimum of each coordinate is not affected by the other coordinates
  - I.e. we could optimize each coordinate independently
- **Note: Optimal learning rate is different for the different coordinates**

Vector update rule



$$\boxed{\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta \nabla_{\mathbf{w}} E}$$

$$\boxed{w_i^{(k+1)} = w_i^{(k)} - \eta \frac{dE\left(w_i^{(k)}\right)}{d\mathrm{w}}}$$

- Conventional vector update rules for gradient descent: update entire vector against direction of gradient
  - Note : Gradient is perpendicular to equal value contour
  - The same learning rate is applied to all components

Problem with vector update rule

$$\boxed{\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta \nabla_{\mathbf{w}} E^T}$$

$$\boxed{w_i^{(k+1)} = w_i^{(k)} - \eta \frac{dE\left(w_i^{(k)}\right)}{d\mathrm{w}}}$$

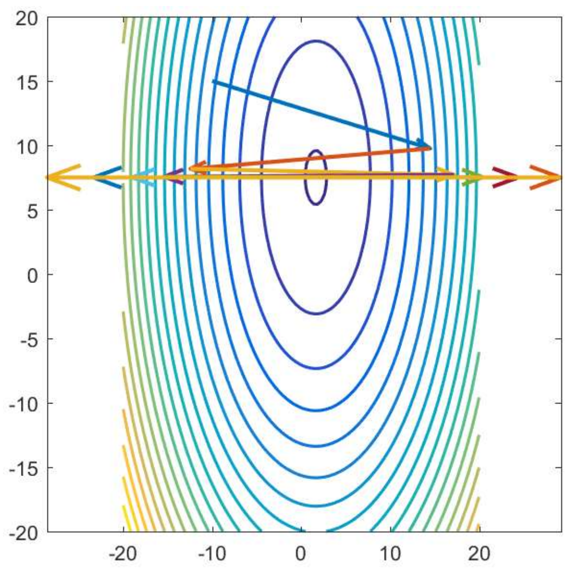
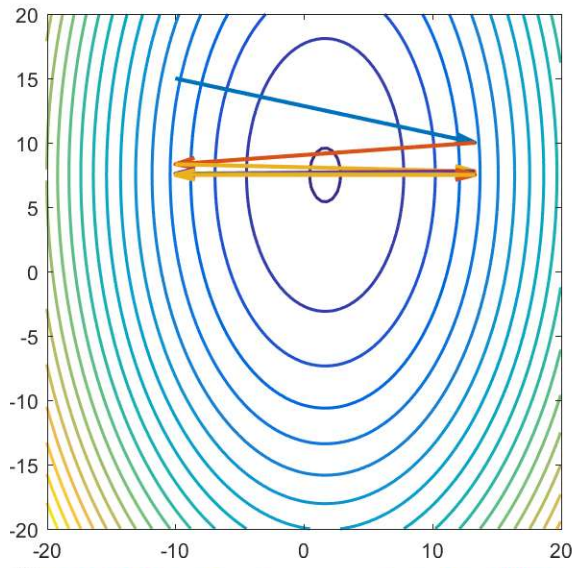
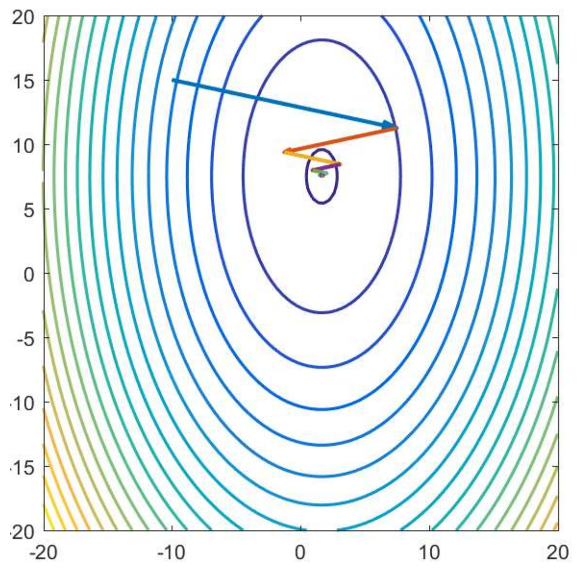
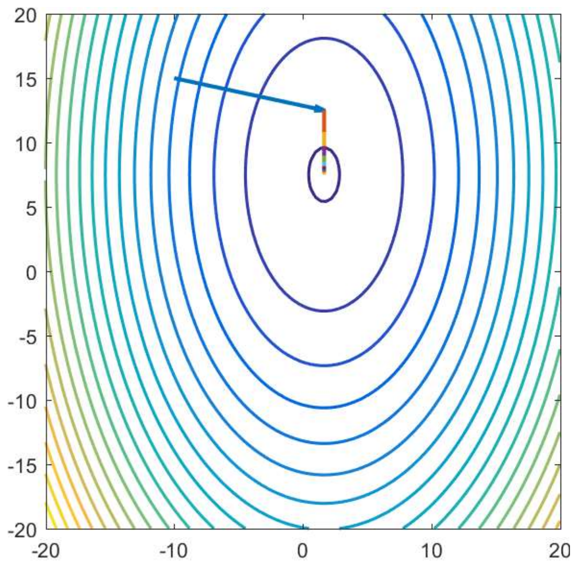
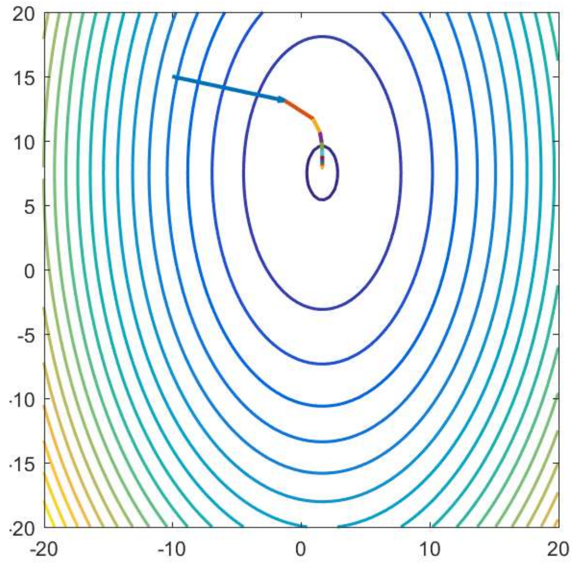
$$\boxed{\eta_{i,opt} = \left(\frac{d^2E\left(w_i^{(k)}\right)}{dw_i^2}\right)^{-1} = a_{ii}^{-1}}$$

- The learning rate must be lower than twice the *smallest* optimal learning rate for any component

$$\eta < 2 \min_i \eta_{i,opt}$$

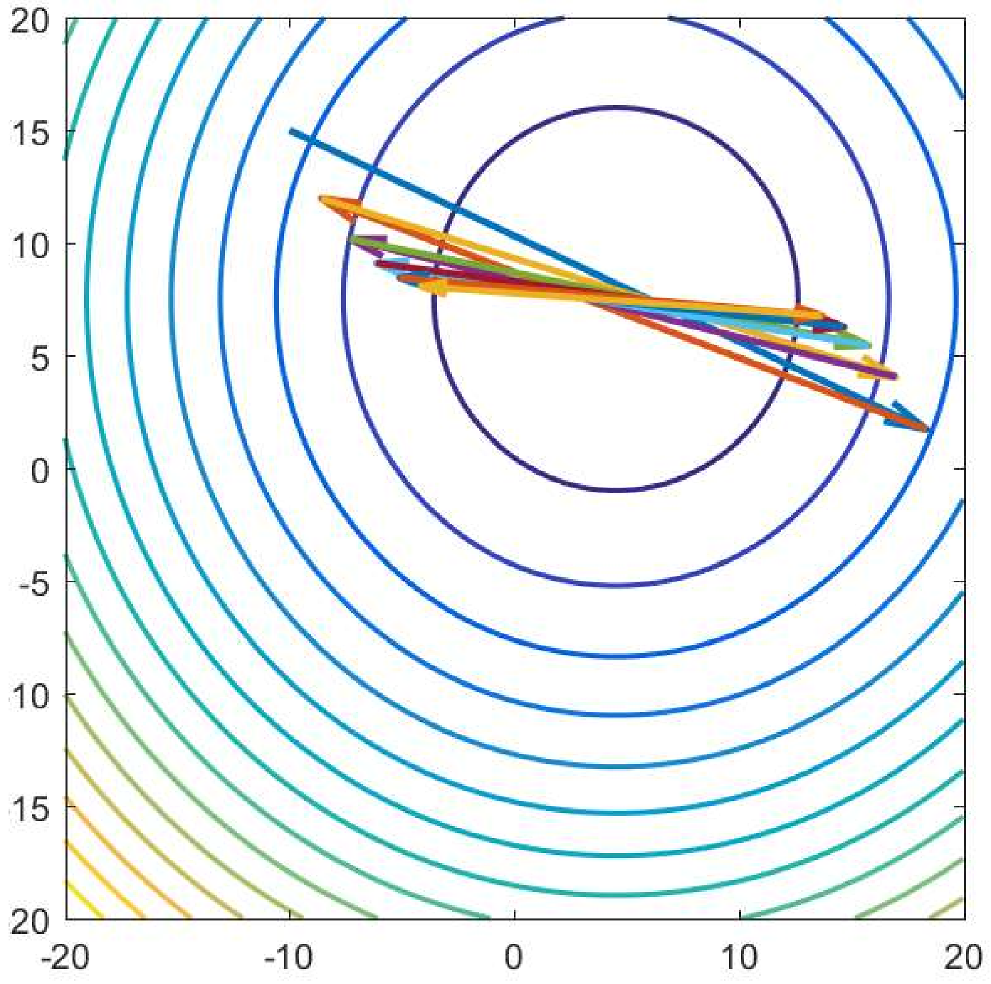
  – Otherwise the learning will diverge
- This, however, makes the learning very slow
  – And will oscillate in all directions where $\eta_{i,opt} \le \eta < 2\eta_{i,opt}$

Dependence on learning rate

- $\eta_{1,opt} = 1;\ \eta_{2,opt} = 0.33$
- $\eta = 2.1\eta_{2,opt}$
- $\eta = 2\eta_{2,opt}$
- $\eta = 1.5\eta_{2,opt}$
- $\eta = \eta_{2,opt}$
- $\eta = 0.75\eta_{2,opt}$

Dependence on learning rate



- $\eta_{1,opt} = 1; \eta_{2,opt} = 0.91;$ $\eta = 1.9\, \eta_{2,opt}$

Convergence

- Convergence behaviors become increasingly unpredictable as dimensions increase
- For the fastest convergence, ideally, the learning rate η must be close to both, the largest $\eta_{i,opt}$ and the smallest $\eta_{i,opt}$
  - To ensure convergence in every direction
  - Generally infeasible
- Convergence is particularly slow if $\frac{\max_i \eta_{i,opt}}{\min_i \eta_{i,opt}}$ is large
  - The "condition" number is small

Comments on the quadratic

- Why are we talking about quadratics?
  - Quadratic functions form some kind of benchmark
  - Convergence of gradient descent is linear
    - Meaning it converges to solution exponentially fast

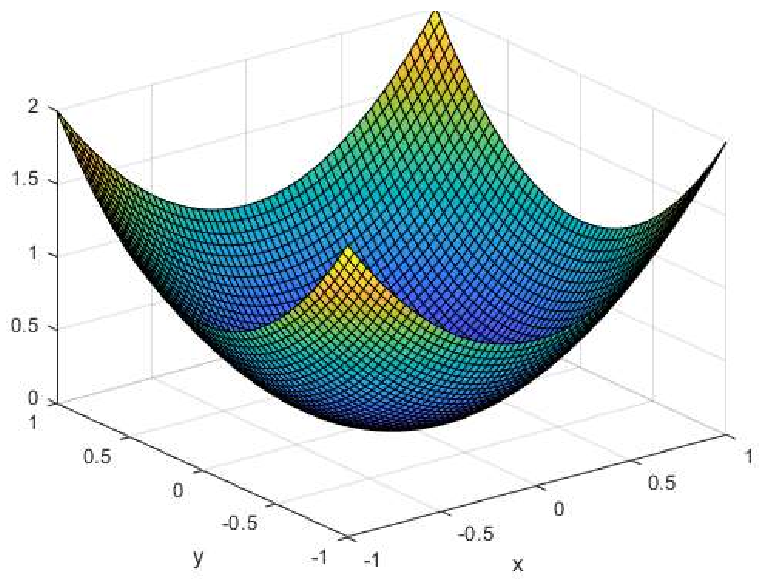
- The convergence for other kinds of functions can be viewed against this benchmark

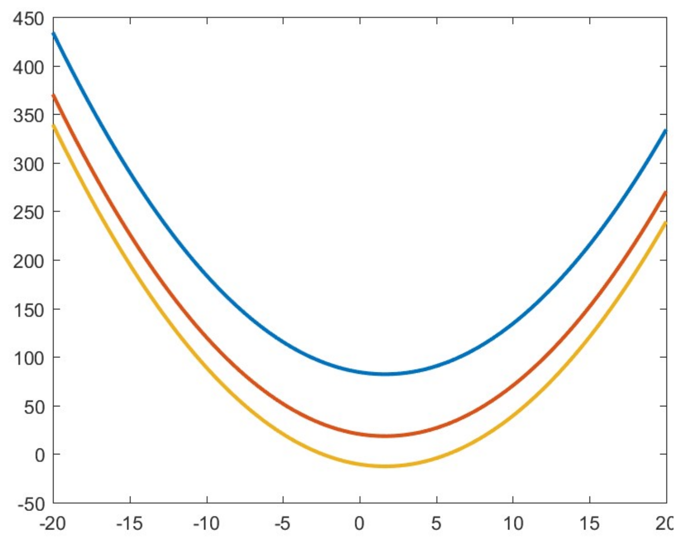
- Actual losses will not be quadratic, but may locally have other structure
  - Local between current location and nearest local minimum

- Some examples in the following slides..
  - Strong convexity
  - Lifschitz continuity
  - Lifschitz smoothness

  - ..and how they affect convergence of gradient descent

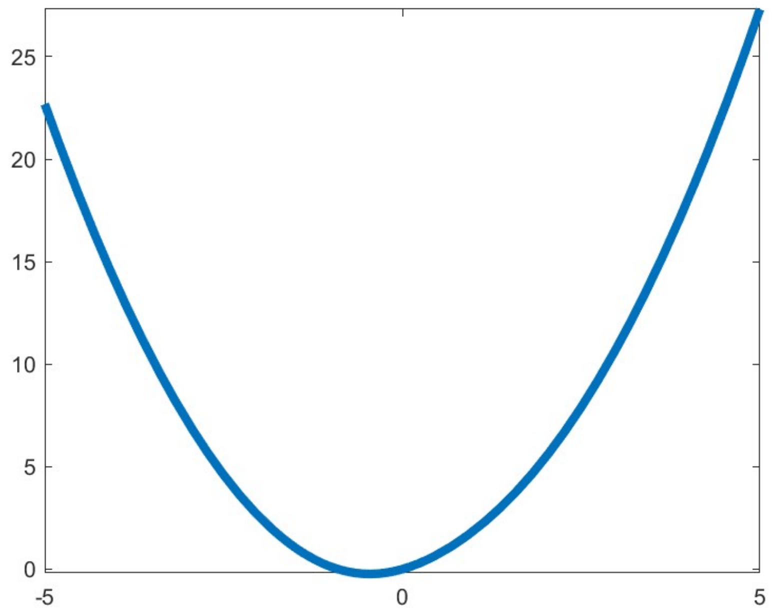
Quadratic convexity

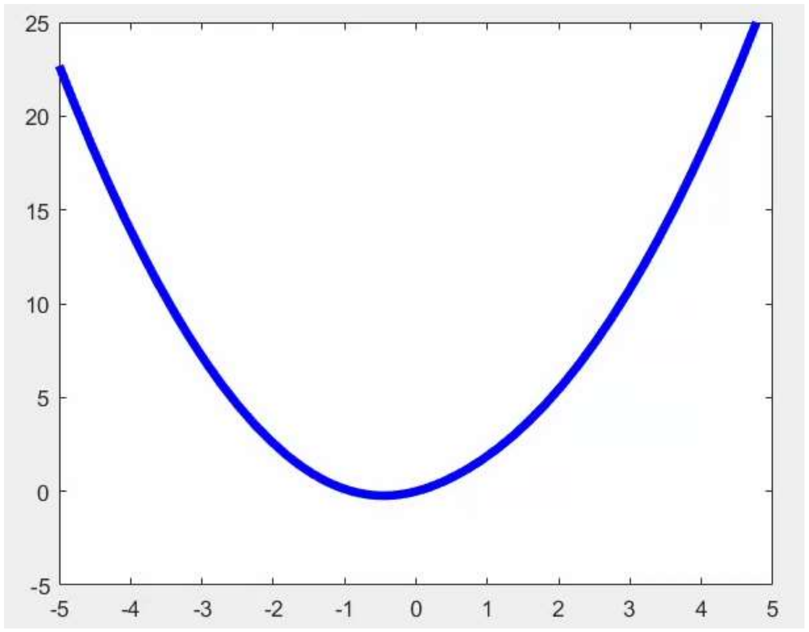




- A quadratic function has the form $\frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c$
  - Every “slice” is a quadratic bowl
- In some sense, the “standard” for gradient-descent based optimization
  - Others convex functions will be steeper in some regions, but flatter in others
- Gradient descent solution will have linear convergence
  - Take $O(\log 1/\varepsilon)$ steps to get within ε of the optimal solution

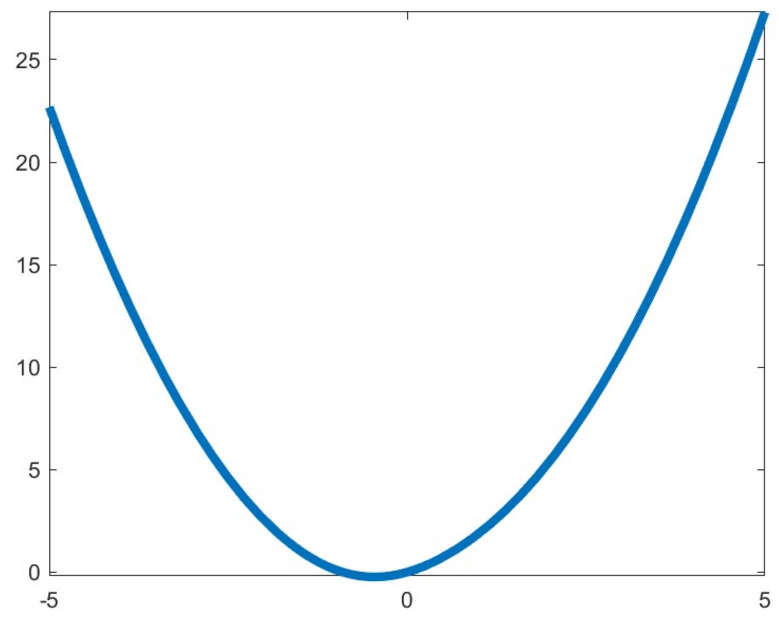
Strong convexity

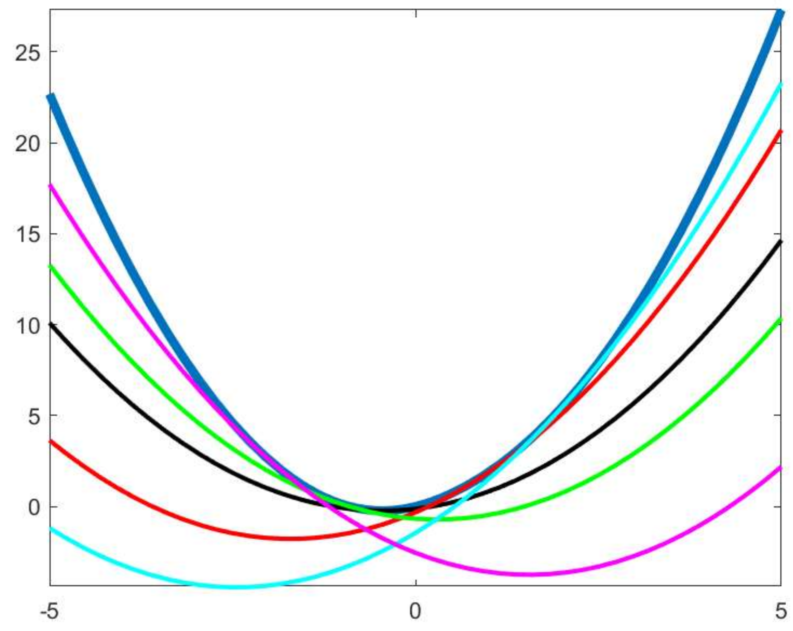




- A strongly convex function is *at least* quadratic in its convexity
  - Has a lower bound to its second derivative
- The function sits within a quadratic bowl
  - At any location, you can draw a quadratic bowl of fixed convexity (quadratic constant equal to lower bound of 2nd derivative) touching the function at that point, which contains it
- Convergence of gradient descent algorithms at least as good as that of the enclosing quadratic

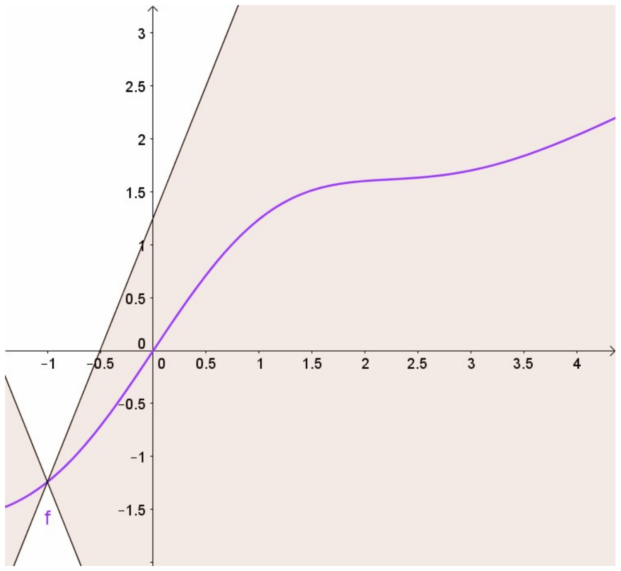
Strong convexity





- A strongly convex function is *at least* quadratic in its convexity
  - Has a lower bound to its second derivative
- The function sits within a quadratic bowl
  - At any location, you can draw a quadratic bowl of fixed convexity (quadratic constant equal to lower bound of 2nd derivative) touching the function at that point, which contains it
- Convergence of gradient descent algorithms at least as good as that of the enclosing quadratic

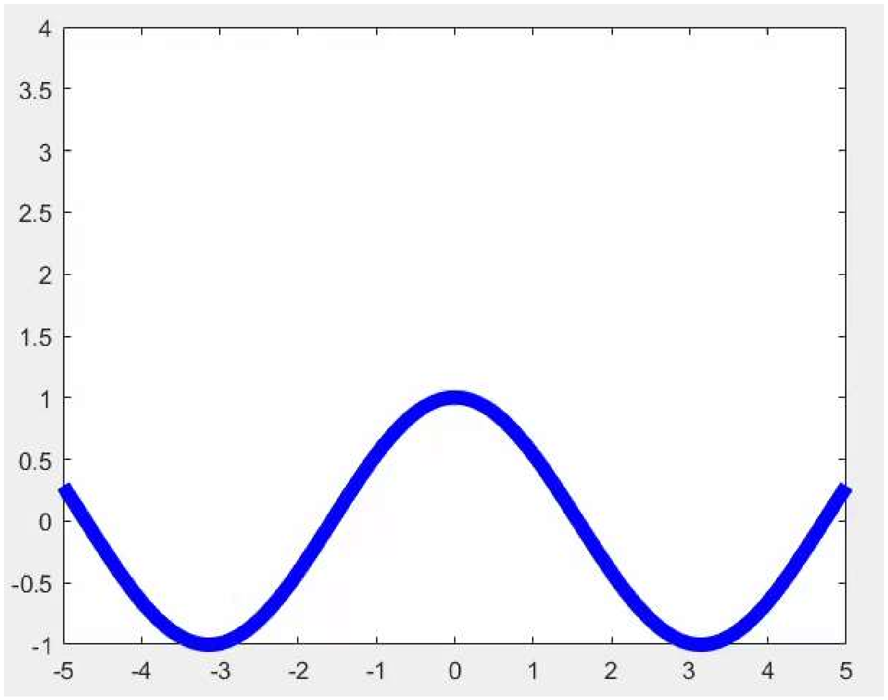
Types of continuity

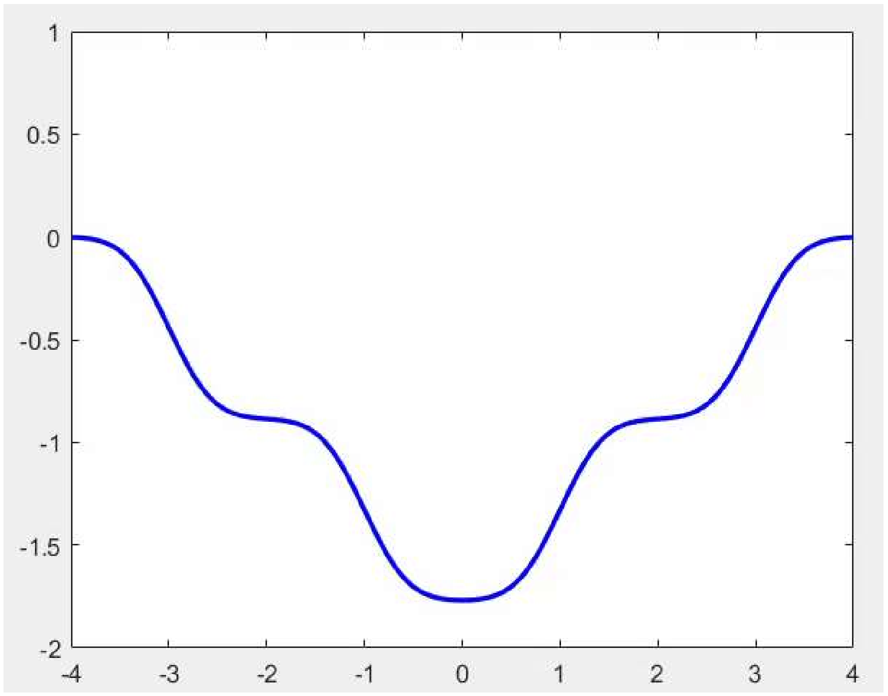


From wikipedia

- Most functions are not strongly convex (if they are convex)
- Instead we will talk in terms of Lifschitz smoothness
- But first : a definition
- ***Lifschitz continuous***: The function always lies outside a cone
  - The slope of the outer surface is the Lifschitz constant
  - $|f(x) - f(y)| \leq L|x - y|$

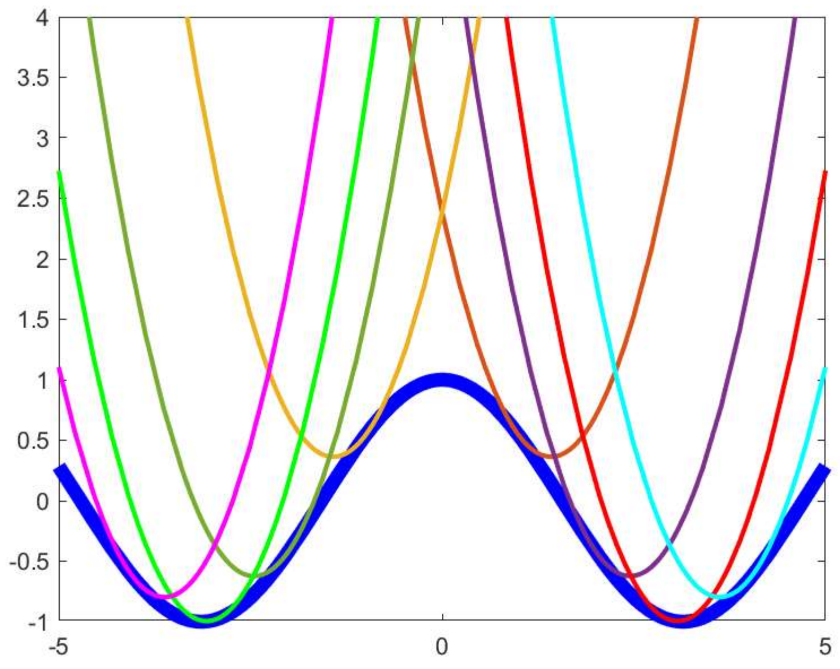
Lifschitz *smoothness*

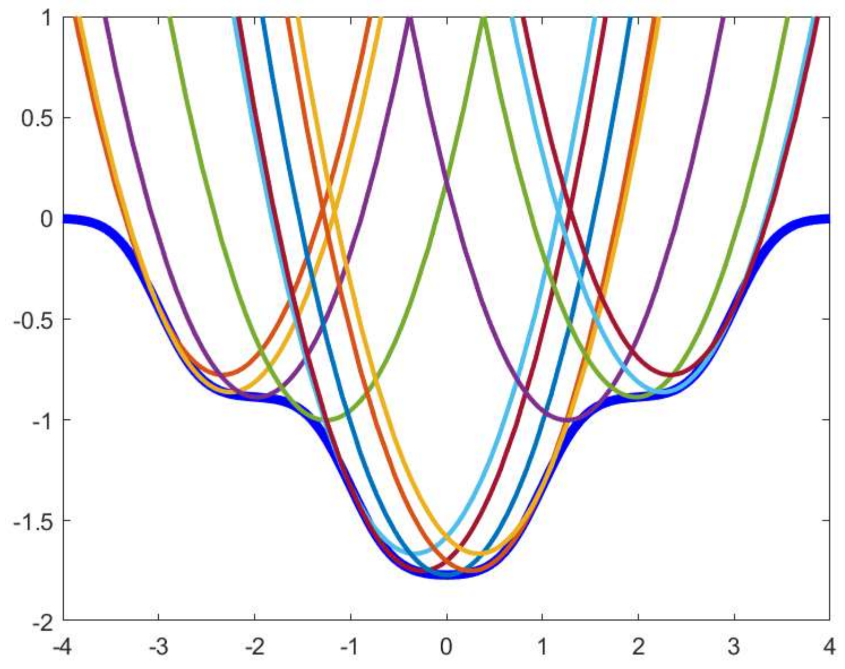




- Lifschitz smooth: The function’s *derivative* is Lifschitz continuous
  - Need not be convex (or even differentiable)
  - Has an *upper bound* on second derivative (if it exists)
- Can always place a quadratic bowl of a fixed curvature within the function
  - Minimum curvature of quadratic must be >= upper bound of second derivative of function (if it exists)

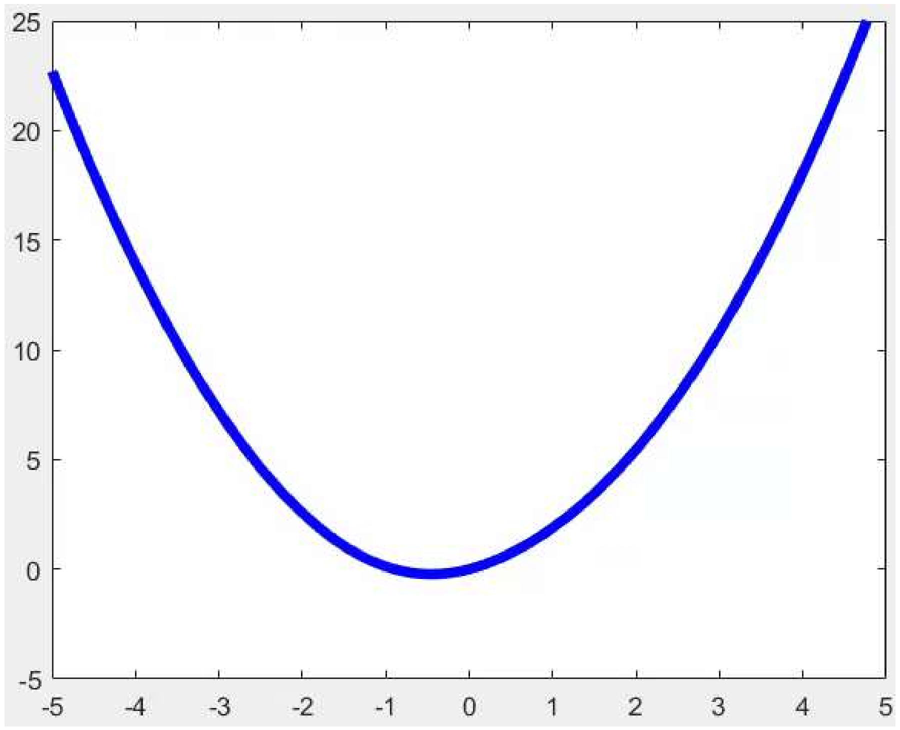
Lifschitz *smoothness*

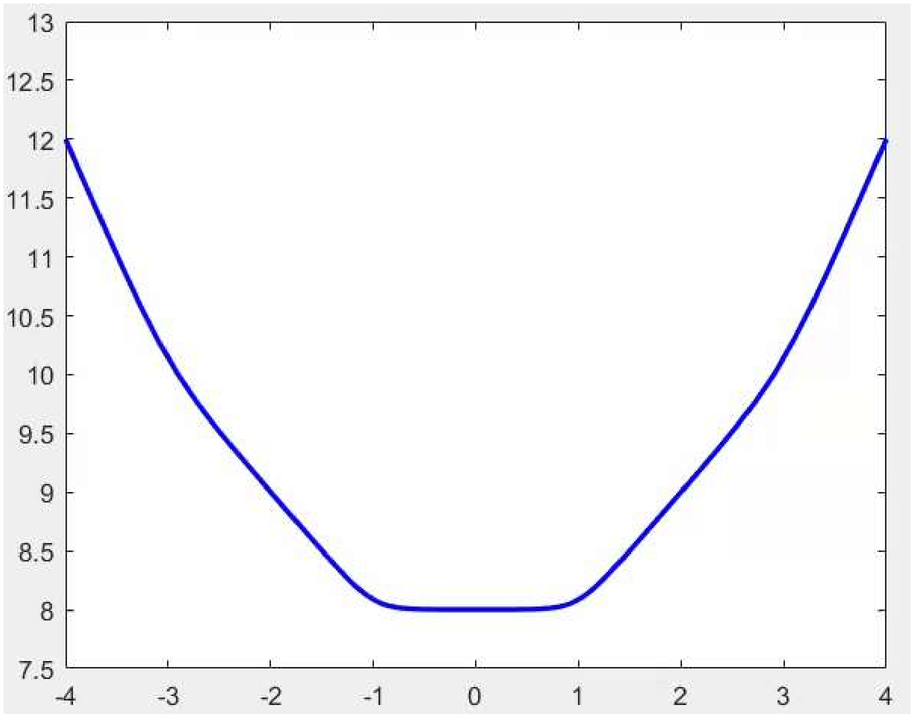




- Lifschitz smooth: The function's *derivative* is Lifschitz continuous
  - Need not be convex (or even differentiable)
  - Has an *upper bound* on second derivative (if it exists)
- Can always place a quadratic bowl of a fixed curvature within the function
  - Minimum curvature of quadratic must be >= upper bound of second derivative of function (if it exists)

Types of smoothness

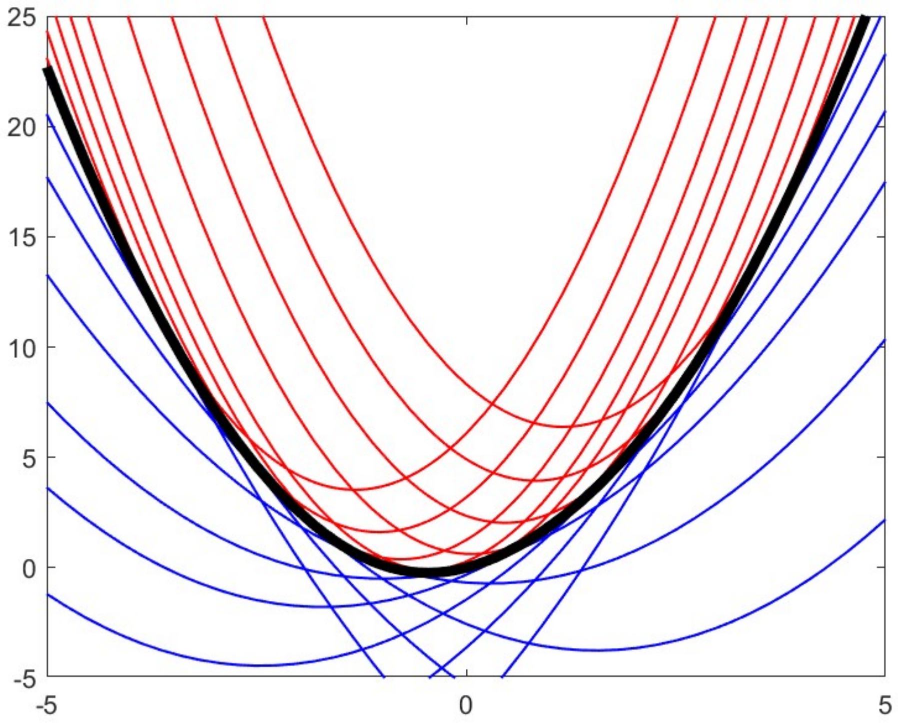


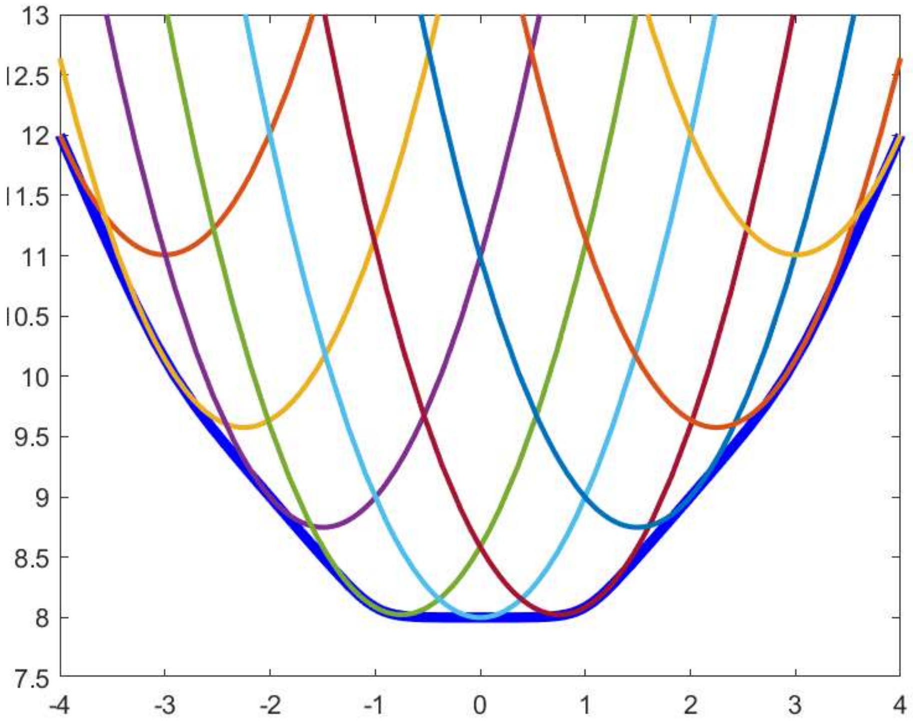


- A function can be both strongly convex and Lipschitz smooth
  - Second derivative has upper *and* lower bounds
  - Convergence depends on curvature of strong convexity (at least linear)

- A function can be convex and Lifschitz smooth, but not strongly convex
  - Convex, but upper bound on second derivative
  - Weaker convergence guarantees, if any (at best linear)
  - This is often a reasonable assumption for the local structure of your loss function

Types of smoothness





- A function can be both strongly convex and Lipschitz smooth
  - Second derivative has upper *and* lower bounds
  - Convergence depends on curvature of strong convexity (at least linear)

- A function can be convex and Lifschitz smooth, but not strongly convex
  - Convex, but upper bound on second derivative
  - Weaker convergence guarantees, if any (at best linear)
  - This is often a reasonable assumption for the local structure of your loss function

Convergence Problems

- For quadratic (strongly) convex functions, gradient descent is exponentially fast
  - Linear convergence
    - Assuming learning rate is non-divergent

- For generic (Lifschitz Smooth) convex functions however, it is very slow

$$\left|f\left(w^{(k)}\right) - f(w^*)\right| \propto \frac{1}{k}\left|f\left(w^{(0)}\right) - f(w^*)\right|$$

  - And inversely proportional to learning rate

$$\left|f\left(w^{(k)}\right) - f(w^*)\right| \leq \frac{1}{2\eta k}\left|w^{(0)} - w^*\right|$$

  - Takes $O(1/\epsilon)$ iterations to get to within ϵ of the solution

  - An inappropriate learning rate will destroy your happiness

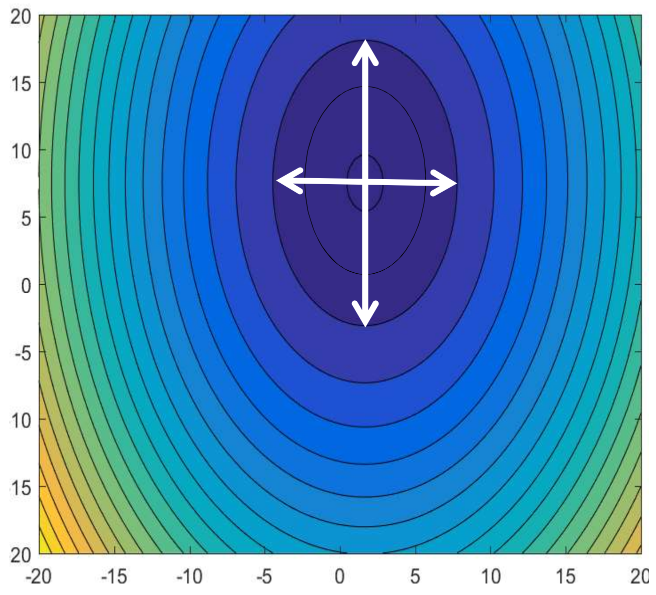
- Second order methods will *locally* convert the loss function to quadratic
  - Convergence behavior will still depend on the nature of the original function

- ***Continuing with the quadratic-based explanation…***

Convergence

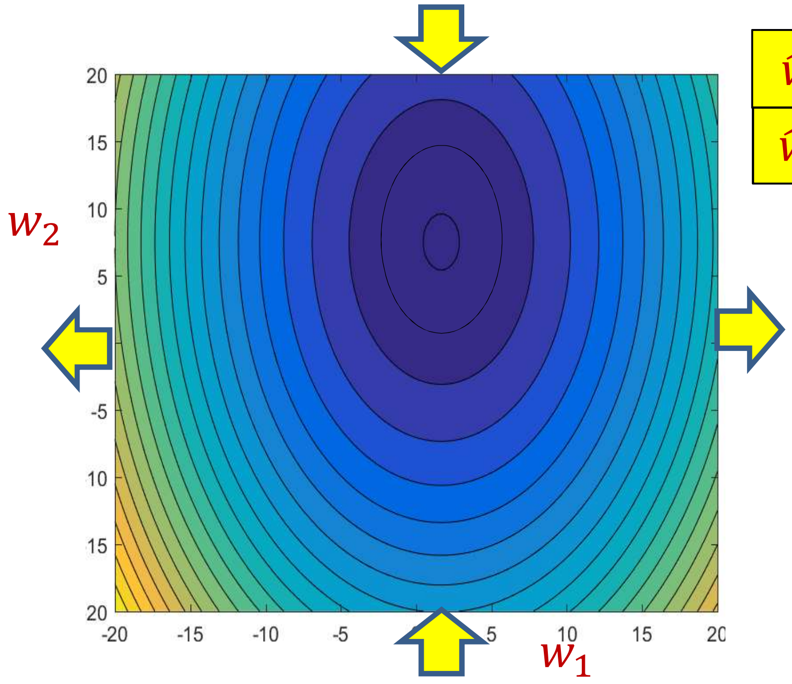
- Convergence behaviors become increasingly unpredictable as dimensions increase
- For the fastest convergence, ideally, the learning rate η must be close to both, the largest $\eta_{i,opt}$ and the smallest $\eta_{i,opt}$
  - To ensure convergence in every direction
  - Generally infeasible
- Convergence is particularly slow if $\frac{\max_i \eta_{i,opt}}{\min_i \eta_{i,opt}}$ is large
  - The "condition" number is small

One reason for the problem



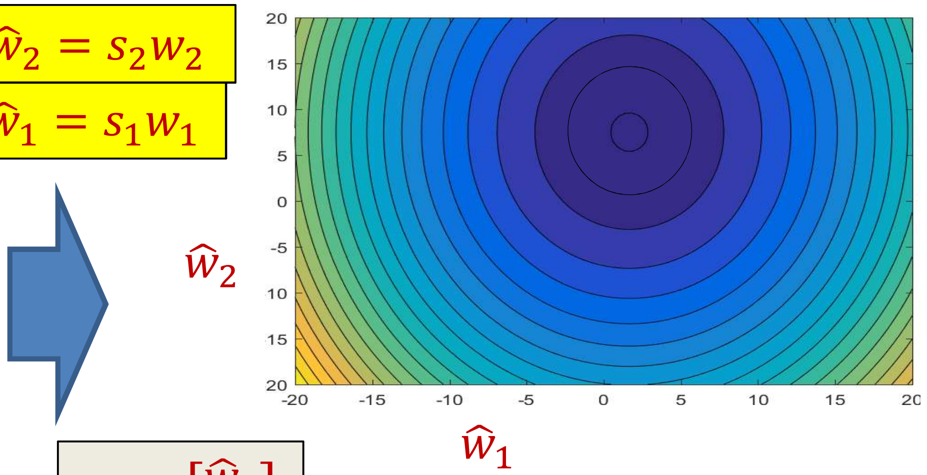
- The objective function has different eccentricities in different directions
  - Resulting in different optimal learning rates for different directions
  - The problem is more difficult when the ellipsoid is not axis aligned: the steps along the two directions are coupled! Moving in one direction changes the gradient along the other
- Solution: *Normalize* the objective to have identical eccentricity in all directions
  - Then all of them will have identical optimal learning rates
  - Easier to find a working learning rate

Solution: Scale the axes



$\widehat{w}_2 = s_2 w_2$

$\widehat{w}_1 = s_1 w_1$



$\widehat{\mathbf{w}} = \begin{bmatrix} \widehat{w}_1 \\ \widehat{w}_2 \end{bmatrix}$

$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$

$S = \begin{bmatrix} s_1 & 0 \\ 0 & s_2 \end{bmatrix}$

$\widehat{\mathbf{w}} = S\mathbf{w}$

- Scale (and rotate) the axes, such that all of them have identical (identity) "spread"
  - Equal-value contours are circular
  - Movement along the coordinate axes become independent
- **Note:** equation of a quadratic surface with circular equal-value contours can be written as

$$E = \frac{1}{2}\widehat{\mathbf{w}}^T\widehat{\mathbf{w}} + \hat{\mathbf{b}}^T\widehat{\mathbf{w}} + c$$

Scaling the axes

- Original equation:

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{b}^T\mathbf{w} + c$$

- We want to find a (diagonal) scaling matrix S such that

$$\mathrm{S} = \begin{bmatrix} s_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & s_N \end{bmatrix}, \qquad \widehat{\mathbf{w}} = \mathrm{S}\mathbf{w}$$

- And

$$E = \frac{1}{2}\widehat{\mathbf{w}}^T\widehat{\mathbf{w}} + \hat{\mathbf{b}}^T\widehat{\mathbf{w}} + \mathrm{c}$$

Scaling the axes

- Original equation:

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{b}^T\mathbf{w} + c$$

- We want to find a (diagonal) scaling matrix S such that

$$\mathrm{S} = \begin{bmatrix} s_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & s_N \end{bmatrix}, \qquad \widehat{\mathbf{w}} = \mathrm{S}\mathbf{w}$$

- And

$$E = \frac{1}{2}\widehat{\mathbf{w}}^T\widehat{\mathbf{w}} + \hat{\mathbf{b}}^T\widehat{\mathbf{w}} + \mathrm{c}$$

By inspection:
$\mathrm{S} = \mathbf{A}^{0.5}$

Scaling the axes

- We have

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{b}^T\mathbf{w} + c$$

$$\widehat{\mathbf{w}} = \mathrm{S}\mathbf{w}$$

$$E = \frac{1}{2}\widehat{\mathbf{w}}^T\widehat{\mathbf{w}} + \hat{\mathbf{b}}^T\widehat{\mathbf{w}} + \mathrm{c}$$

$$= \frac{1}{2}\mathbf{w}^T\mathrm{S}^T\mathrm{S}\mathbf{w} + \hat{\mathbf{b}}^T\mathrm{S}\mathbf{w} + c$$

- Equating linear and quadratic coefficients, we get

$$\mathrm{S}^T\mathrm{S} = \mathbf{A}, \qquad \hat{\mathbf{b}}^T\mathrm{S} = \mathbf{b}^T$$

- Solving: $\boxed{\mathrm{S} = \mathbf{A}^{0.5}, \qquad \hat{\mathbf{b}} = \mathbf{A}^{-0.5}\mathbf{b}}$

Scaling the axes

- We have

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{b}^T\mathbf{w} + c$$

$$\widehat{\mathbf{w}} = S\mathbf{w}$$

$$E = \frac{1}{2}\widehat{\mathbf{w}}^T\widehat{\mathbf{w}} + \hat{\mathbf{b}}^T\widehat{\mathbf{w}} + \mathrm{c}$$

- Solving for S we get

$$\widehat{\mathbf{w}} = \mathbf{A}^{0.5}\mathbf{w}, \qquad \hat{\mathbf{b}} = \mathbf{A}^{-0.5}\mathbf{b}$$

Scaling the axes

- We have

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{b}^T\mathbf{w} + c$$

$$\widehat{\mathbf{w}} = S\mathbf{w}$$

$$E = \frac{1}{2}\widehat{\mathbf{w}}^T\widehat{\mathbf{w}} + \hat{\mathbf{b}}^T\widehat{\mathbf{w}} + \mathrm{c}$$

- Solving for S we get

$$\widehat{\mathbf{w}} = \boxed{\mathbf{A}^{0.5}}\mathbf{w}, \qquad \hat{\mathbf{b}} = \boxed{\mathbf{A}^{-0.5}}\mathbf{b}$$

The Inverse Square Root of A

- For *any* positive definite $\mathbf{A}$, we can write

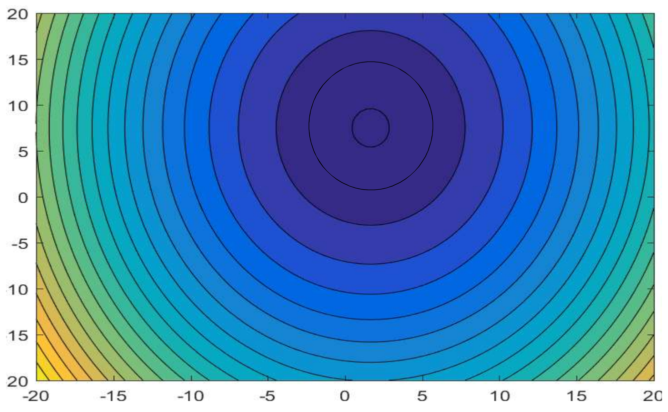
$$\mathbf{A} = \mathbf{E}\boldsymbol{\Lambda}\mathbf{E}^{\mathrm{T}}$$

  - Eigen decomposition
  - $\mathbf{E}$ is an orthogonal matrix
  - $\boldsymbol{\Lambda}$ is a diagonal matrix of non-zero diagonal entries

- Defining $\mathbf{A}^{0.5} = \mathbf{E}\boldsymbol{\Lambda}^{0.5}\mathbf{E}^{\mathrm{T}}$
  - Check $(\mathbf{A}^{0.5})^{\mathrm{T}}\mathbf{A}^{0.5} = \mathbf{E}\boldsymbol{\Lambda}\mathbf{E}^{\mathrm{T}} = \mathbf{A}$

- Defining $\mathbf{A}^{-0.5} = \mathbf{E}\boldsymbol{\Lambda}^{-0.5}\mathbf{E}^{\mathrm{T}}$
  - Check: $(\mathbf{A}^{-0.5})^{\mathrm{T}}\mathbf{A}^{-0.5} = \mathbf{E}\boldsymbol{\Lambda}^{-1}\mathbf{E}^{\mathrm{T}} = \mathbf{A}^{-1}$

Returning to our problem

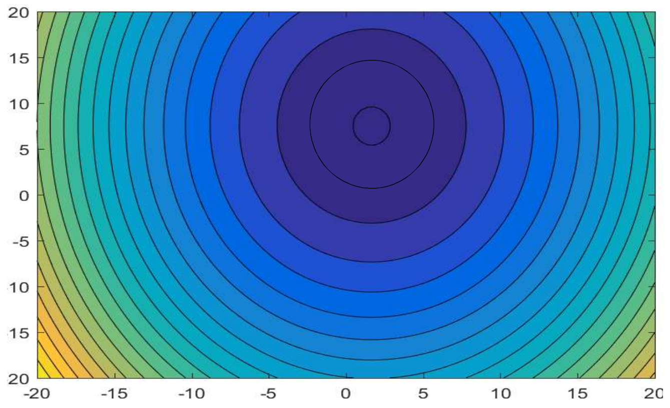


$$E = \frac{1}{2}\hat{\mathbf{w}}^T\hat{\mathbf{w}} + \hat{\mathbf{b}}^T\hat{\mathbf{w}} + c$$

- Computing the gradient, and noting that $\mathbf{A}^{0.5}$is symmetric, we can relate $\nabla_{\hat{\mathbf{w}}}E$ and $\nabla_{\mathbf{w}}E$:

$$\nabla_{\hat{\mathbf{w}}}E = \hat{\mathbf{w}}^T + \hat{\mathbf{b}}^T$$
$$= \mathbf{w}^T\mathbf{A}^{0.5} + \mathbf{b}^T\mathbf{A}^{-0.5}$$
$$= (\mathbf{w}^T\mathbf{A} + \mathbf{b}^T)\mathbf{A}^{-0.5}$$
$$= \nabla_{\mathbf{w}}E.\mathbf{A}^{-0.5}$$

Returning to our problem

- 

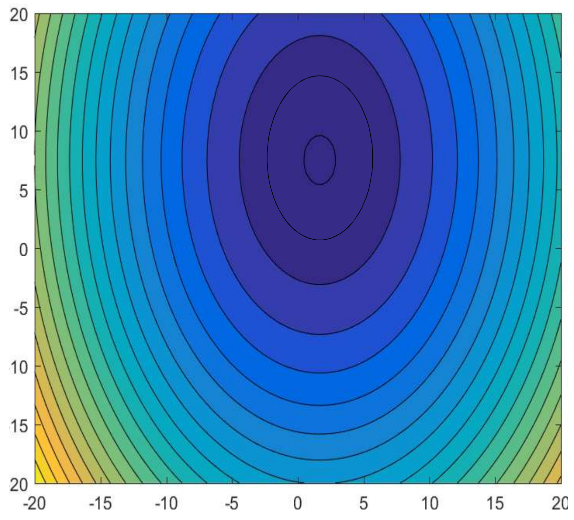
$$E = \frac{1}{2}\hat{\mathbf{w}}^T\hat{\mathbf{w}} + \hat{\mathbf{b}}^T\hat{\mathbf{w}} + c$$

- Gradient descent rule:
  - $\hat{\mathbf{w}}^{(k+1)} = \hat{\mathbf{w}}^{(k)} - \eta \nabla_{\hat{\mathbf{w}}} E\left(\hat{\mathbf{w}}^{(k)}\right)^T$
  - Learning rate is now independent of direction

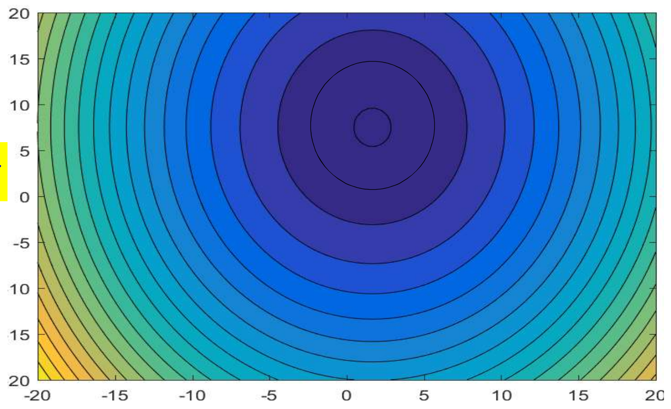
- Using $\hat{\mathbf{w}} = \mathbf{A}^{0.5}\mathbf{w}$, and $\nabla_{\hat{\mathbf{w}}} E(\hat{\mathbf{w}})^T = \mathbf{A}^{-0.5}\nabla_{\mathbf{w}} E(\mathbf{w})^T$

$$\boxed{\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \mathbf{A}^{-1}\nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T}$$

Modified update rule



$\widehat{\mathbf{w}} = \mathbf{A}^{0.5}\mathbf{w}$



$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{b}^T\mathbf{w} + c$

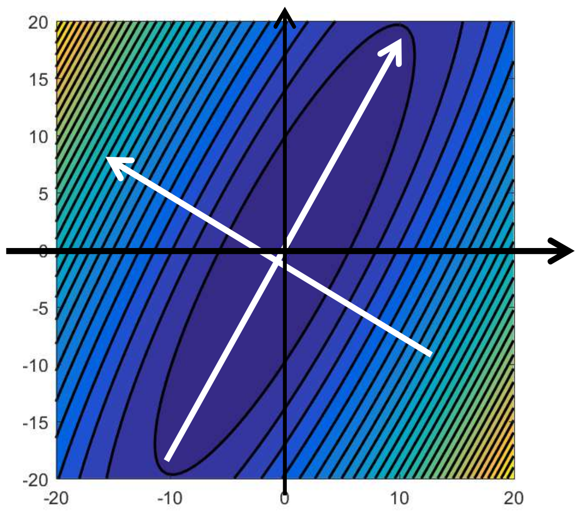
$E = \frac{1}{2}\widehat{\mathbf{w}}^T\widehat{\mathbf{w}} + \hat{\mathbf{b}}^T\widehat{\mathbf{w}} + c$

- $\widehat{\mathbf{w}}^{(k+1)} = \widehat{\mathbf{w}}^{(k)} - \eta\nabla_{\widehat{\mathbf{w}}}E\left(\widehat{\mathbf{w}}^{(k)}\right)^T$

- Leads to the modified gradient descent rule

$$\boxed{\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta\mathbf{A}^{-1}\nabla_{\mathbf{w}}E\left(\mathbf{w}^{(k)}\right)^T}$$

For non-axis-aligned quadratics..



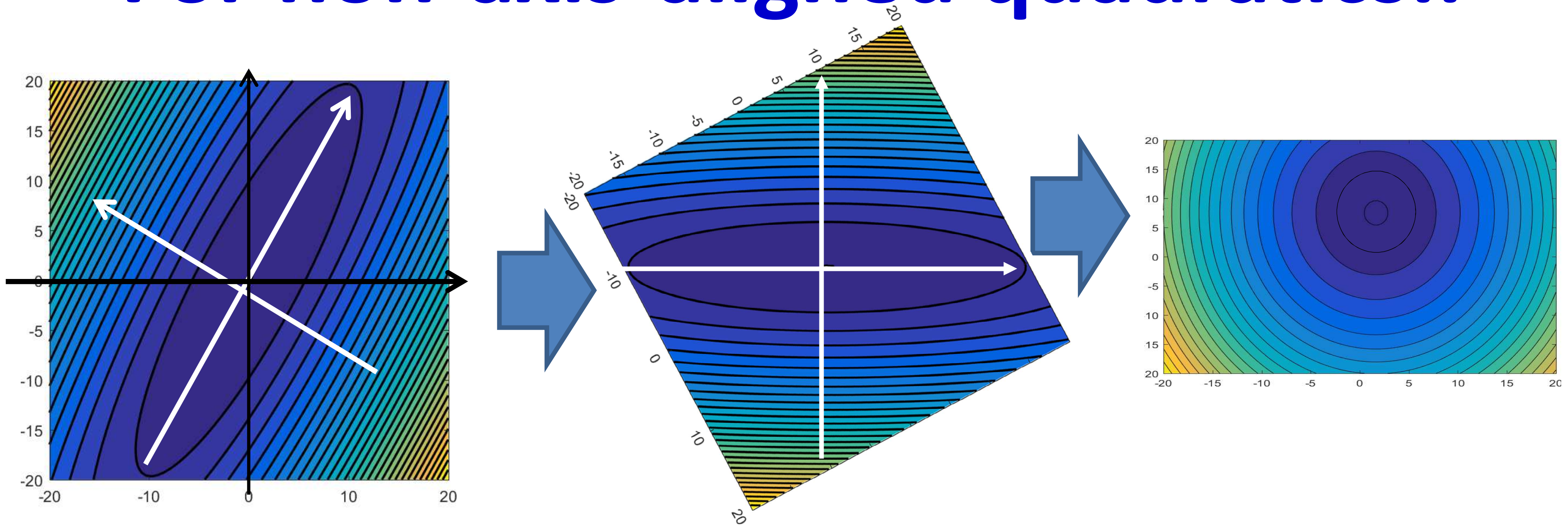
$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c$$

$$E = \frac{1}{2}\sum_i a_{ii}w_i^2 + \sum_{i \neq j} a_{ij}w_iw_j + \sum_i b_iw_i + c$$

- If $\mathbf{A}$ is not diagonal, the contours are not axis-aligned
  - Because of the cross-terms $a_{ij}w_iw_j$
  - The major axes of the ellipsoids are the *Eigenvectors* of $\mathbf{A}$, and their diameters are proportional to the Eigen values of $\mathbf{A}$

- But this does not affect the discussion
  - This is merely a rotation of the space from the axis-aligned case
  - The component-wise optimal learning rates along the major and minor axes of the equal-contour ellipsoids will be different, causing problems
    - The optimal rates along the axes are Inversely proportional to the *eigenvalues* of $\mathbf{A}$

For non-axis-aligned quadratics..

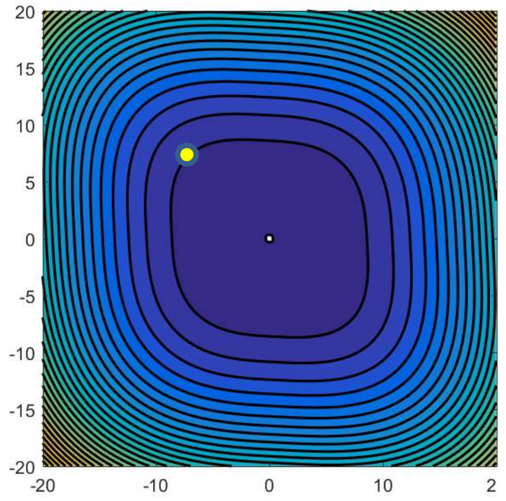
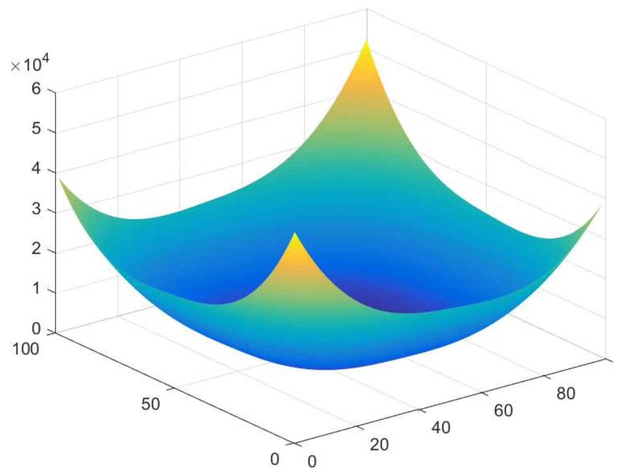


- The component-wise optimal learning rates along the major and minor axes of the contour ellipsoids will differ, causing problems
  - Inversely proportional to the *eigenvalues* of $\mathbf{A}$

- This can be fixed as before by rotating and resizing the different directions to obtain the same *normalized* update rule as before:

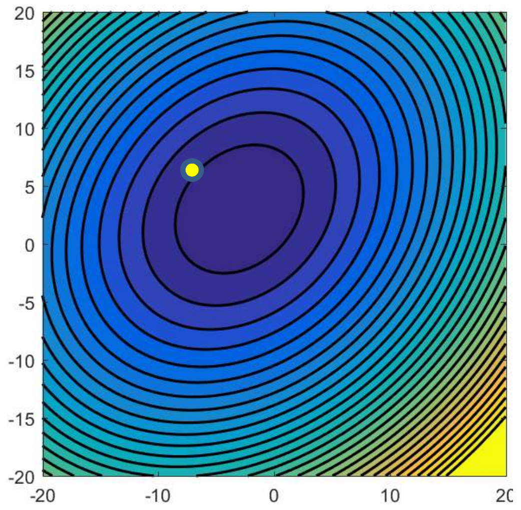
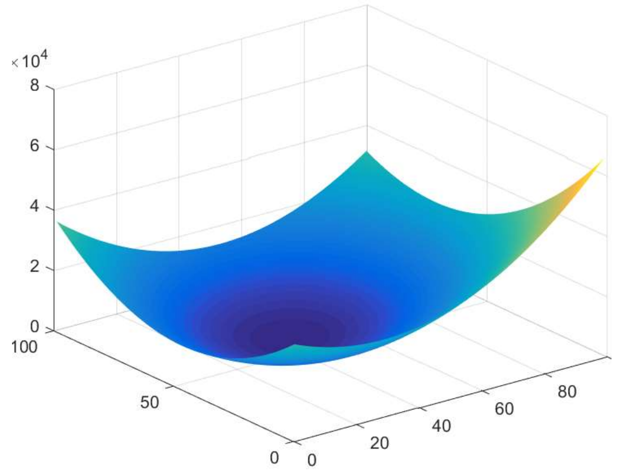
$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \mathbf{A}^{-1}\mathbf{b}$$

Generic differentiable *multivariate* convex functions

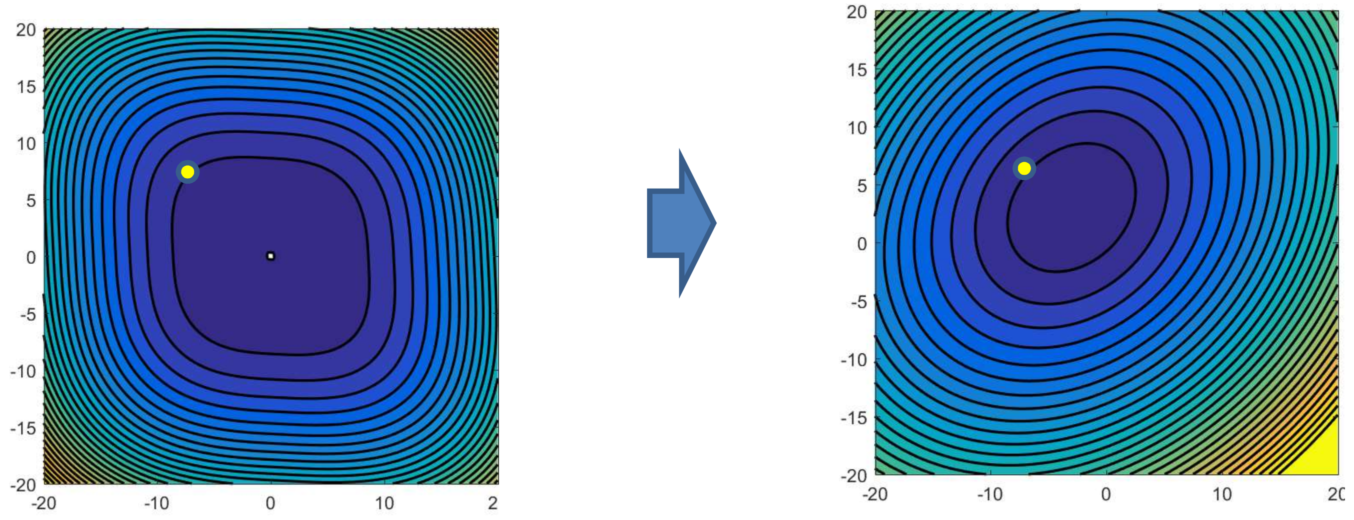



- Taylor expansion

$$E(\mathbf{w}) \approx E\left(\mathbf{w}^{(k)}\right) + \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)\left(\mathbf{w} - w^{(k)}\right) + \frac{1}{2}\left(\mathbf{w} - w^{(k)}\right)^T H_E\left(w^{(k)}\right)\left(\mathbf{w} - w^{(k)}\right) + \cdots$$

Generic differentiable *multivariate* convex functions



- Taylor expansion

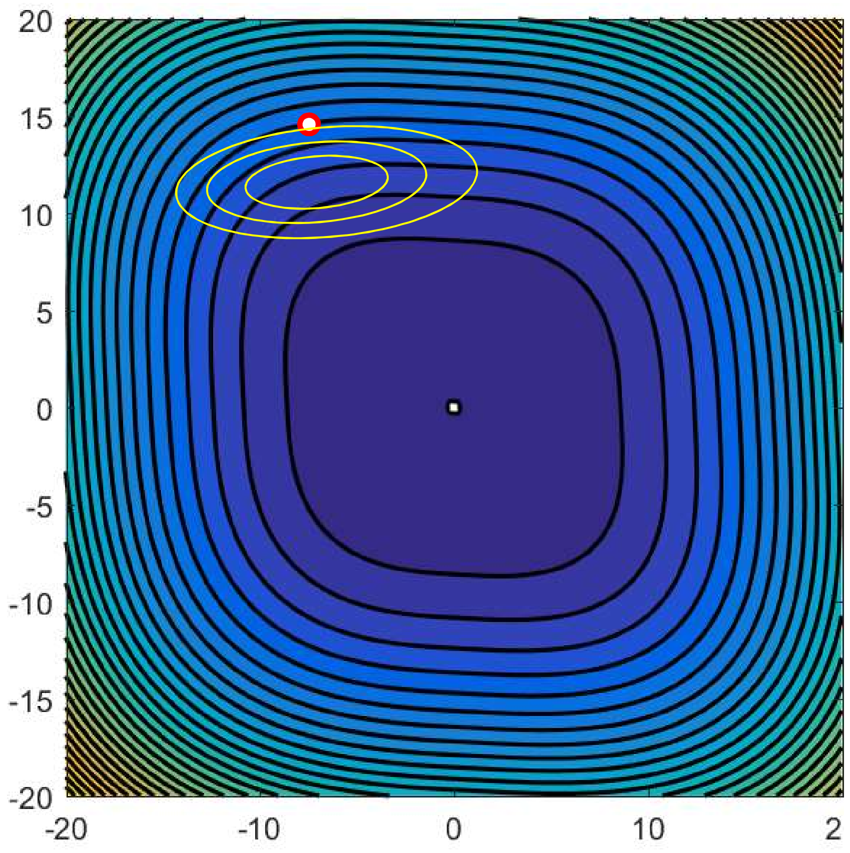
$$\boldsymbol{E}(\mathbf{w}) \approx \boldsymbol{E}\left(\mathbf{w}^{(k)}\right) + \nabla_{\mathbf{w}}\boldsymbol{E}\left(\mathbf{w}^{(k)}\right)\left(\mathbf{w} - \boldsymbol{w}^{(k)}\right) + \frac{\mathbf{1}}{\mathbf{2}}\left(\mathbf{w} - \boldsymbol{w}^{(k)}\right)^{T}\boldsymbol{H}_{E}\left(\boldsymbol{w}^{(k)}\right)\left(\mathbf{w} - \boldsymbol{w}^{(k)}\right) + \cdots$$

- Note that this has the form $\frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c$
- Using the same logic as before, we get the normalized update rule

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1}\nabla_{\mathbf{w}}E\left(\mathbf{w}^{(k)}\right)^{T}$$

- **For a quadratic function, the optimal η is 1 (which is exactly Newton's method)**
  - **And should not be greater than 2!**

Minimization by Newton's method $(\eta = 1)$



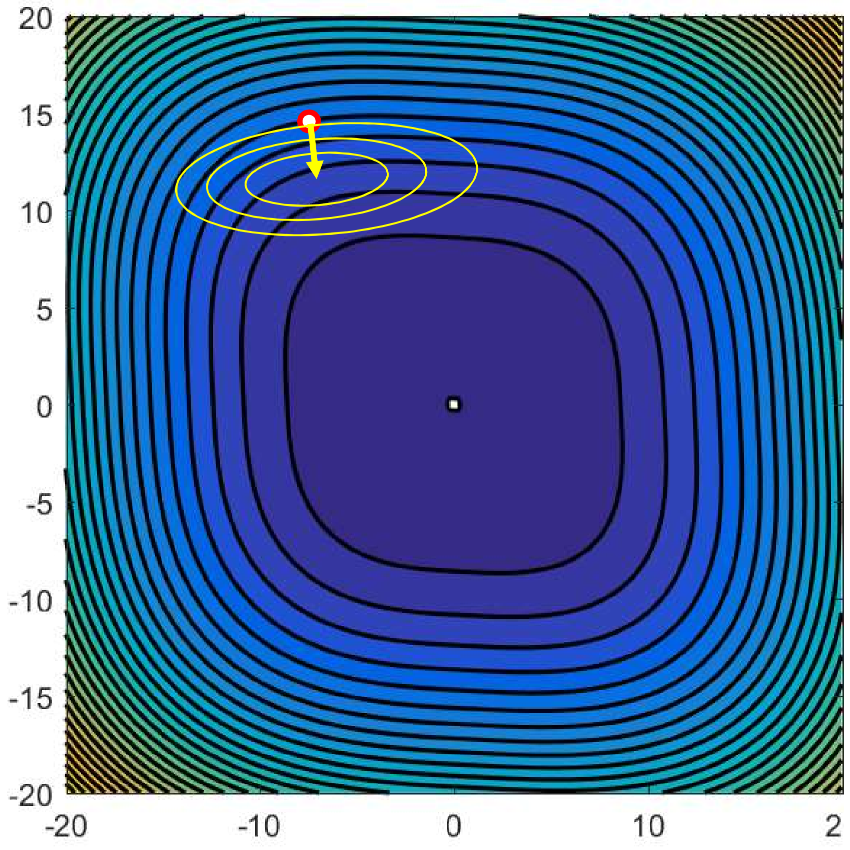
Fit a quadratic at each point and find the minimum of that quadratic

- Iterated localized optimization with quadratic approximations

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

– $\eta = 1$

Minimization by Newton's method $(\eta = 1)$

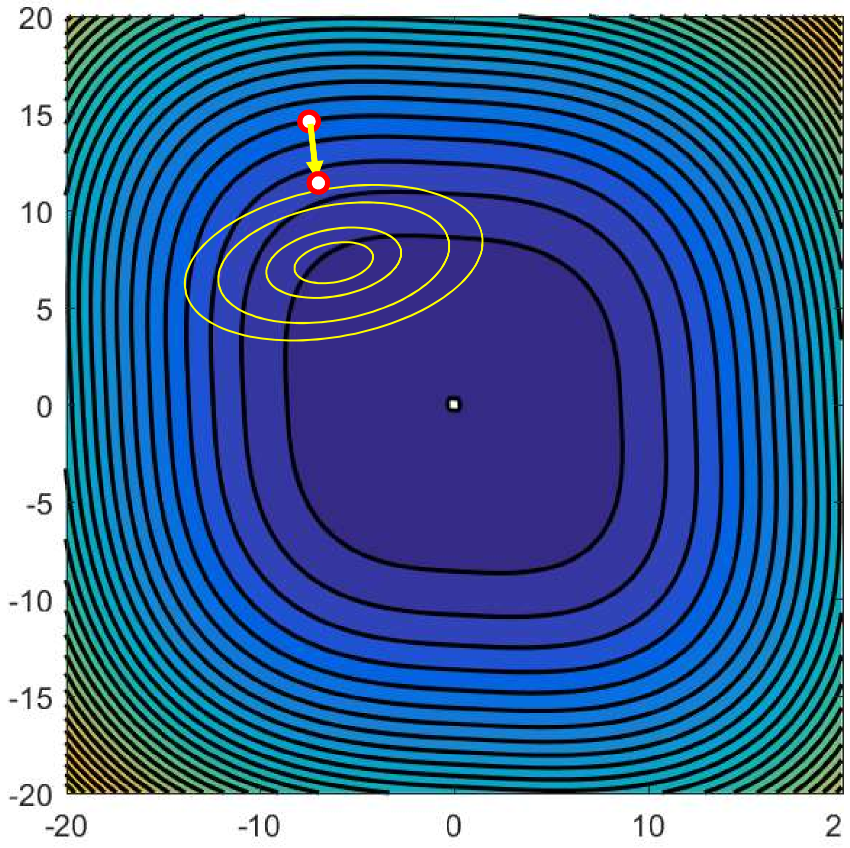


- Iterated localized optimization with quadratic approximations

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

  - $\eta = 1$

Minimization by Newton's method $(\eta = 1)$

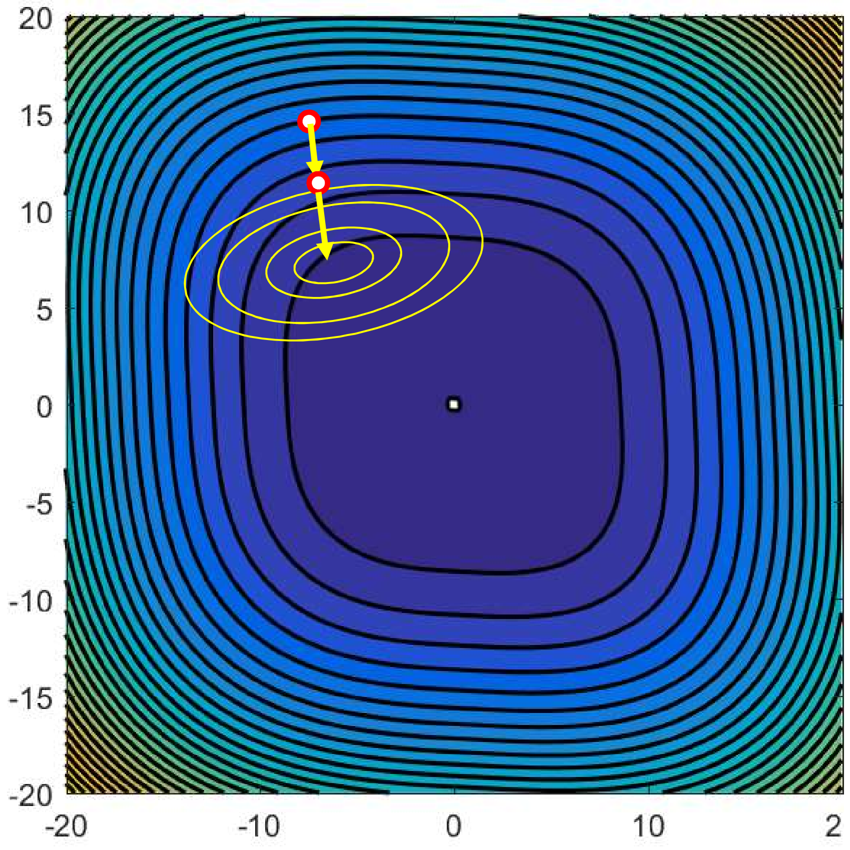


- Iterated localized optimization with quadratic approximations

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

– $\eta = 1$

Minimization by Newton's method $(\eta = 1)$

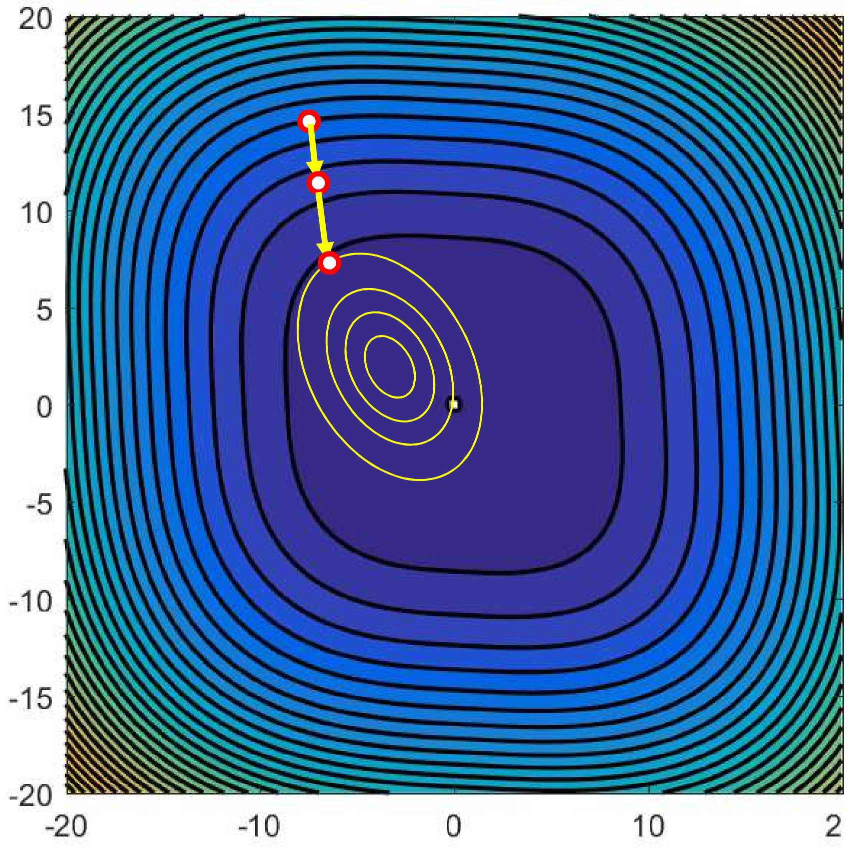


- Iterated localized optimization with quadratic approximations

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

– $\eta = 1$

Minimization by Newton’s method

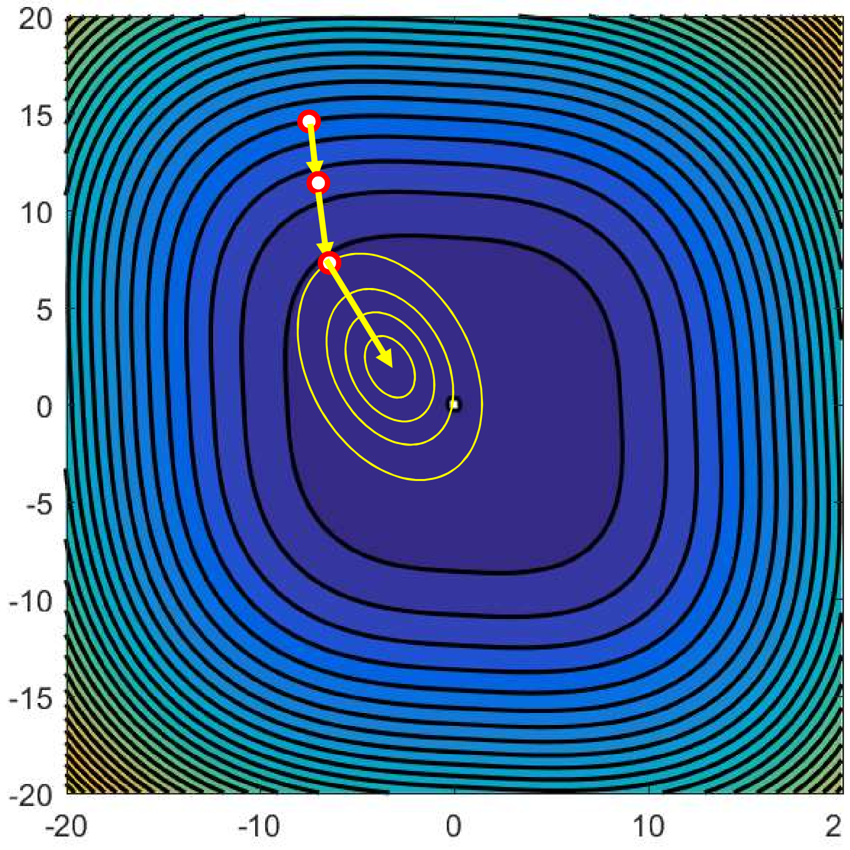


- Iterated localized optimization with quadratic approximations

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

– $\eta = 1$

Minimization by Newton's method

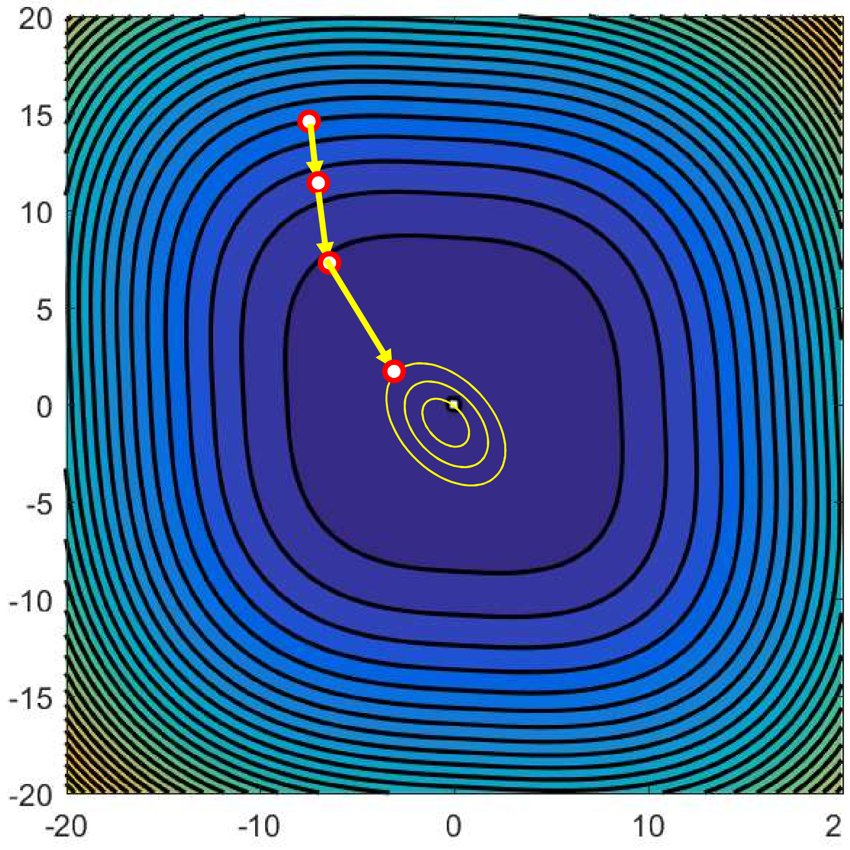


- Iterated localized optimization with quadratic approximations

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

  - $\eta = 1$

Minimization by Newton’s method

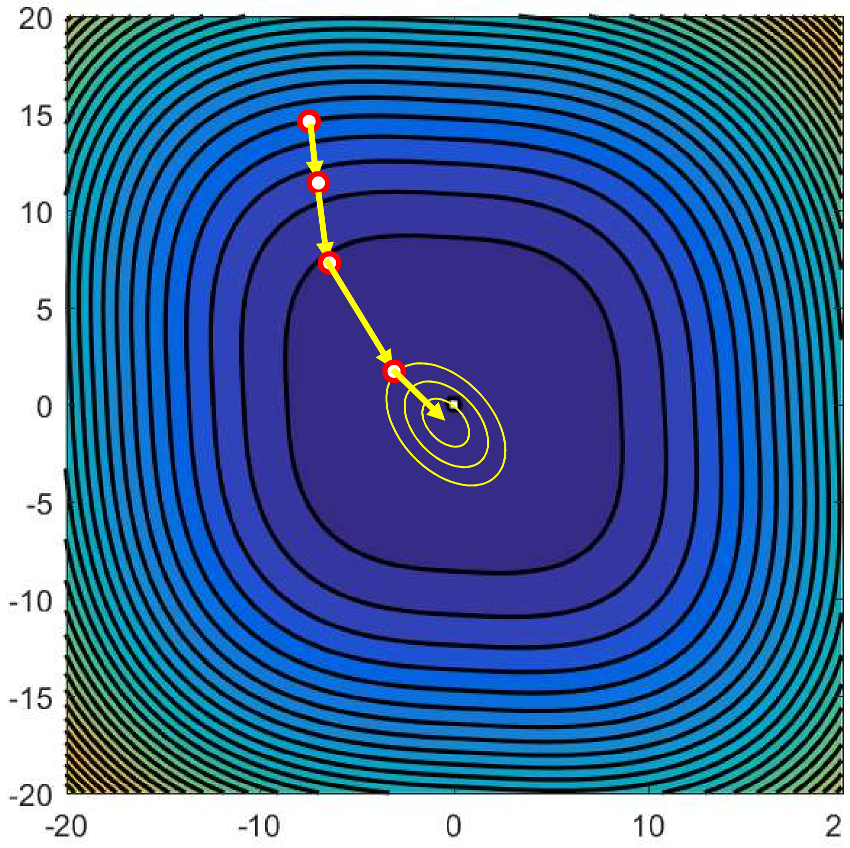


- Iterated localized optimization with quadratic approximations

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

– $\eta = 1$

Minimization by Newton's method

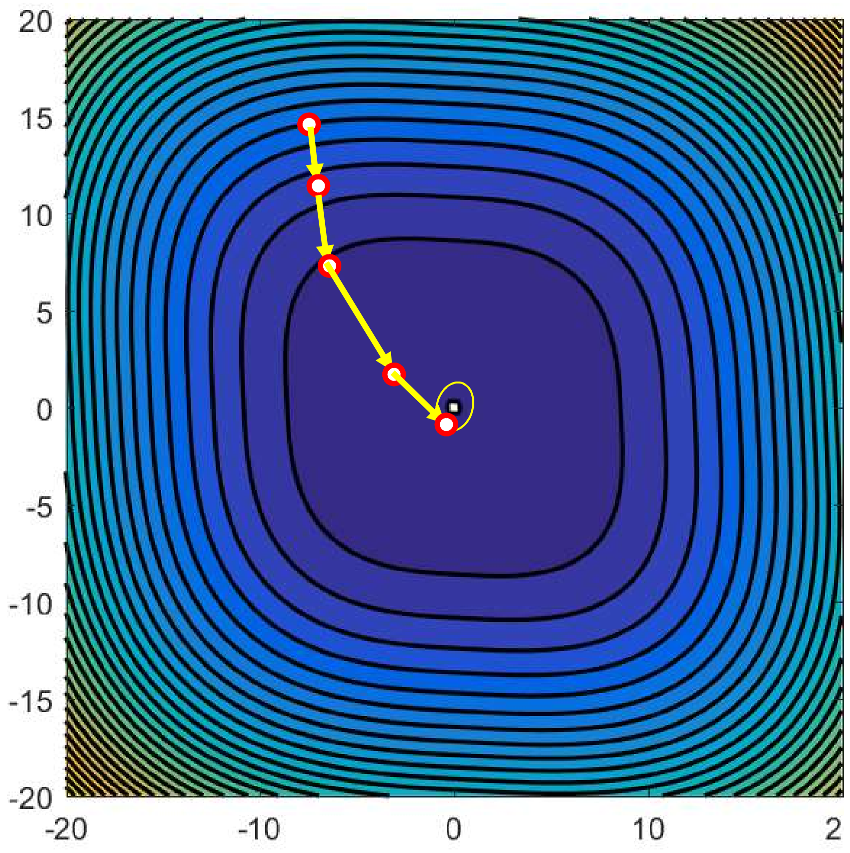


- Iterated localized optimization with quadratic approximations

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

  - $\eta = 1$

Minimization by Newton's method

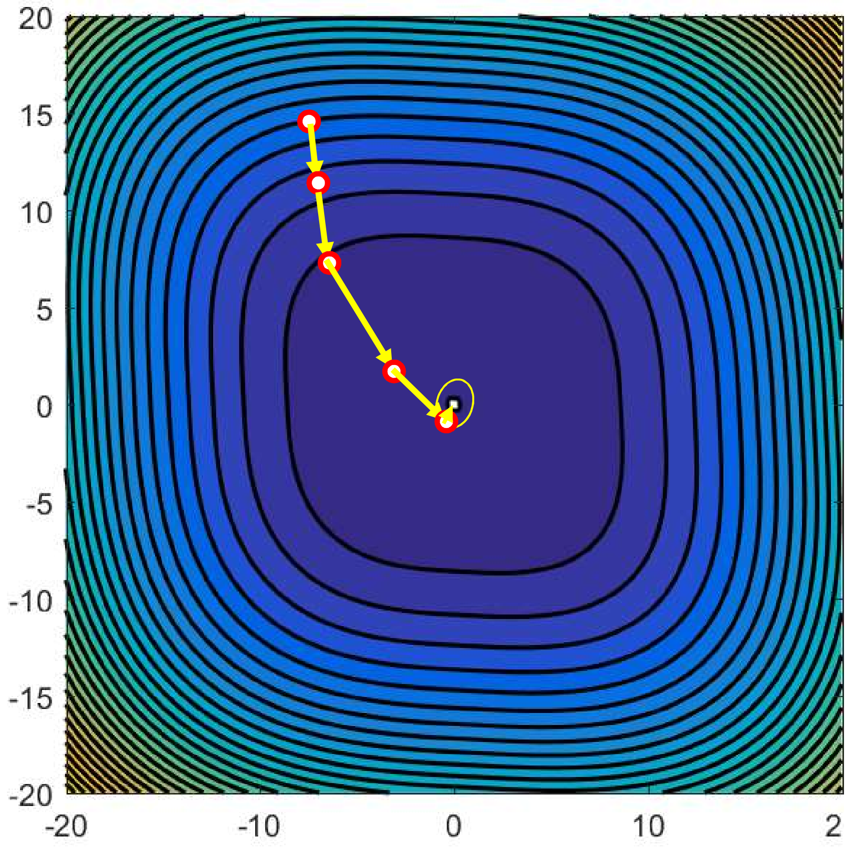


- Iterated localized optimization with quadratic approximations

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

– $\eta = 1$

Minimization by Newton’s method

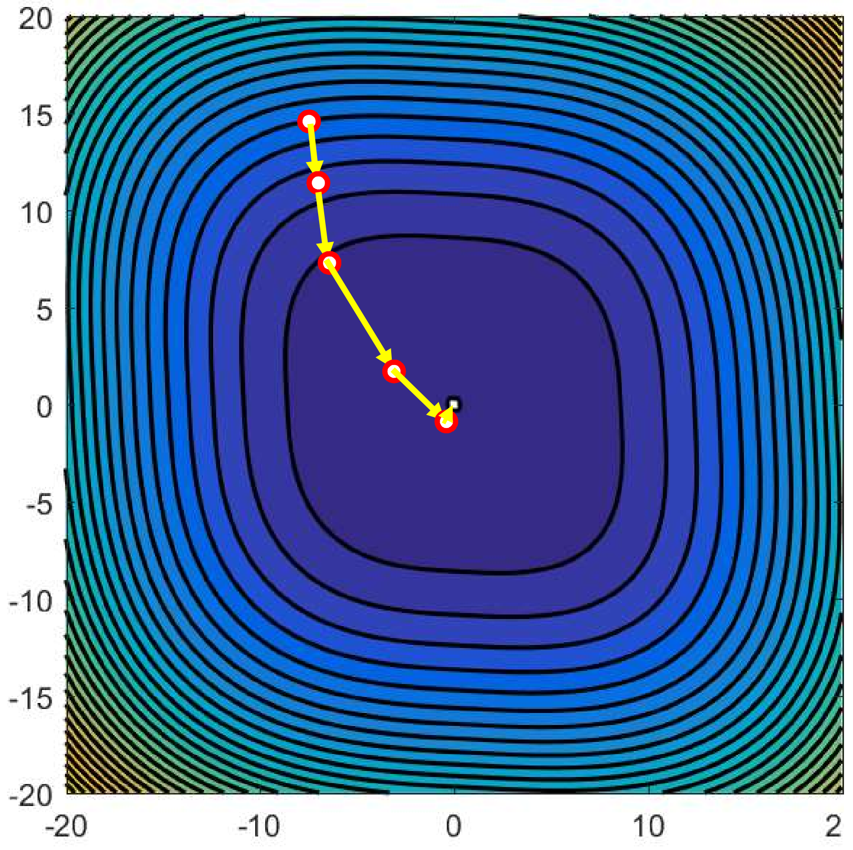


- Iterated localized optimization with quadratic approximations

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

  - $\eta = 1$

Minimization by Newton's method



- Iterated localized optimization with quadratic approximations

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

  - $\eta = 1$

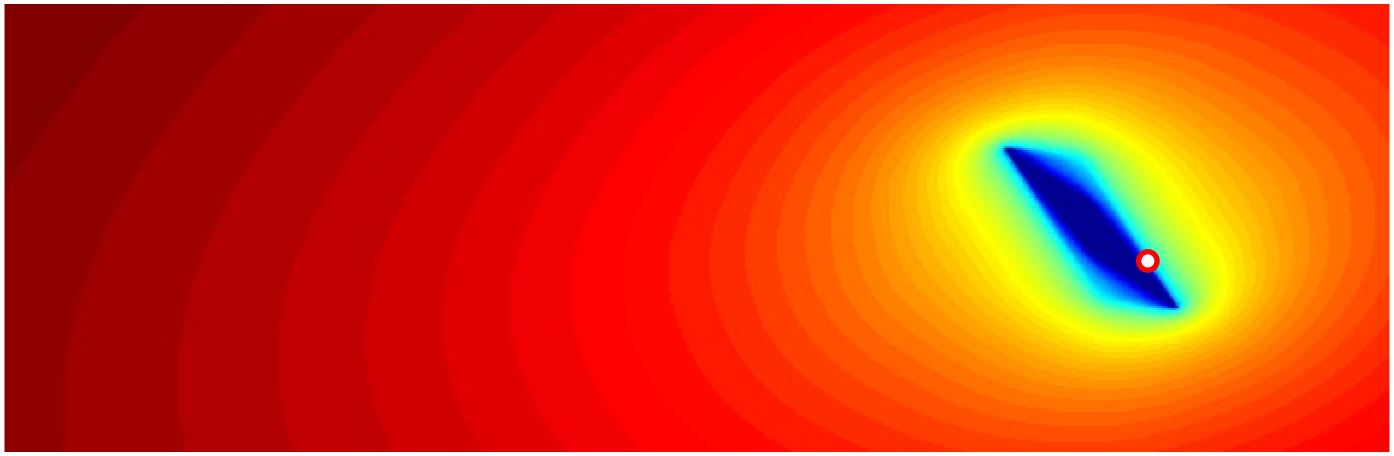
Issues: 1. The Hessian

- Normalized update rule

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\boldsymbol{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

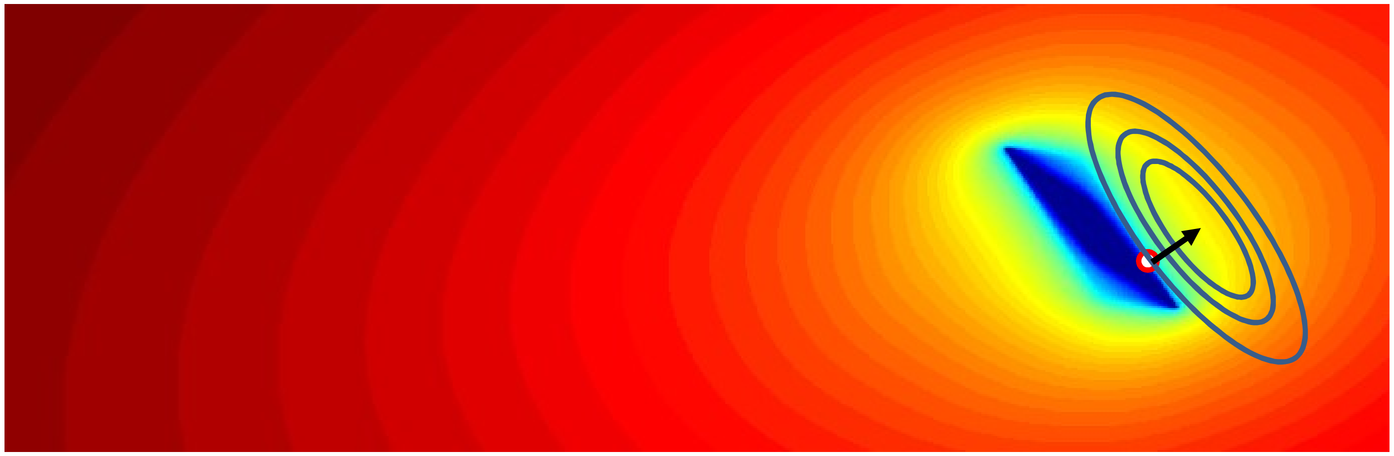
- For complex models such as neural networks, with a very large number of parameters, the Hessian $H_E\left(\boldsymbol{w}^{(k)}\right)$ is extremely difficult to compute
  - For a network with only 100,000 parameters, the Hessian will have 10^{10} cross-derivative terms
  - And its even harder to invert, since it will be enormous

Issues: 1. The Hessian



- For non-convex functions, the Hessian may not be positive semi-definite, in which case the algorithm can *diverge*
  - Goes away from, rather than towards the minimum

Issues: 1. The Hessian

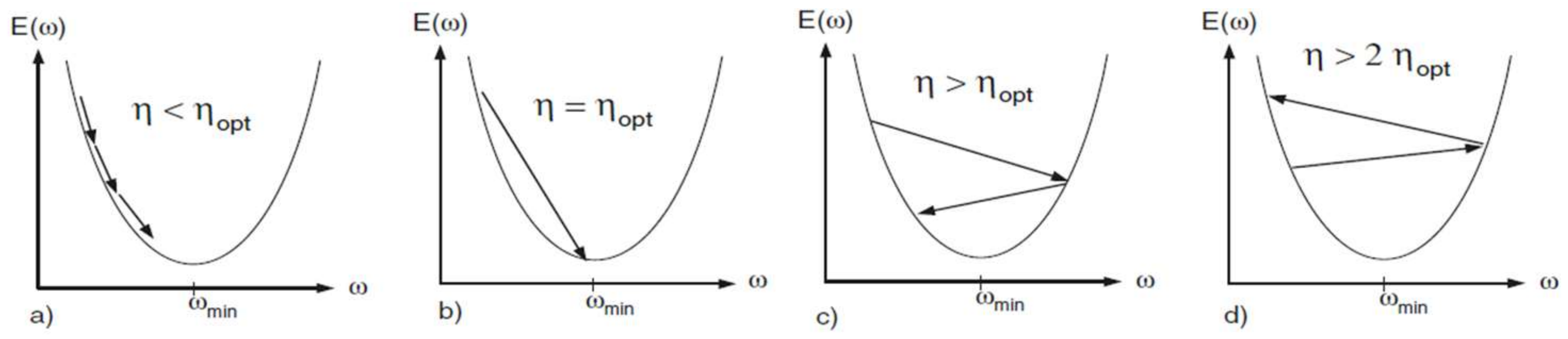


- For non-convex functions, the Hessian may not be positive semi-definite, in which case the algorithm can *diverge*
  - Goes away from, rather than towards the minimum
  - Now requires additional checks to avoid movement in directions corresponding to –ve Eigenvalues of the Hessian

Issues: 1 – contd.

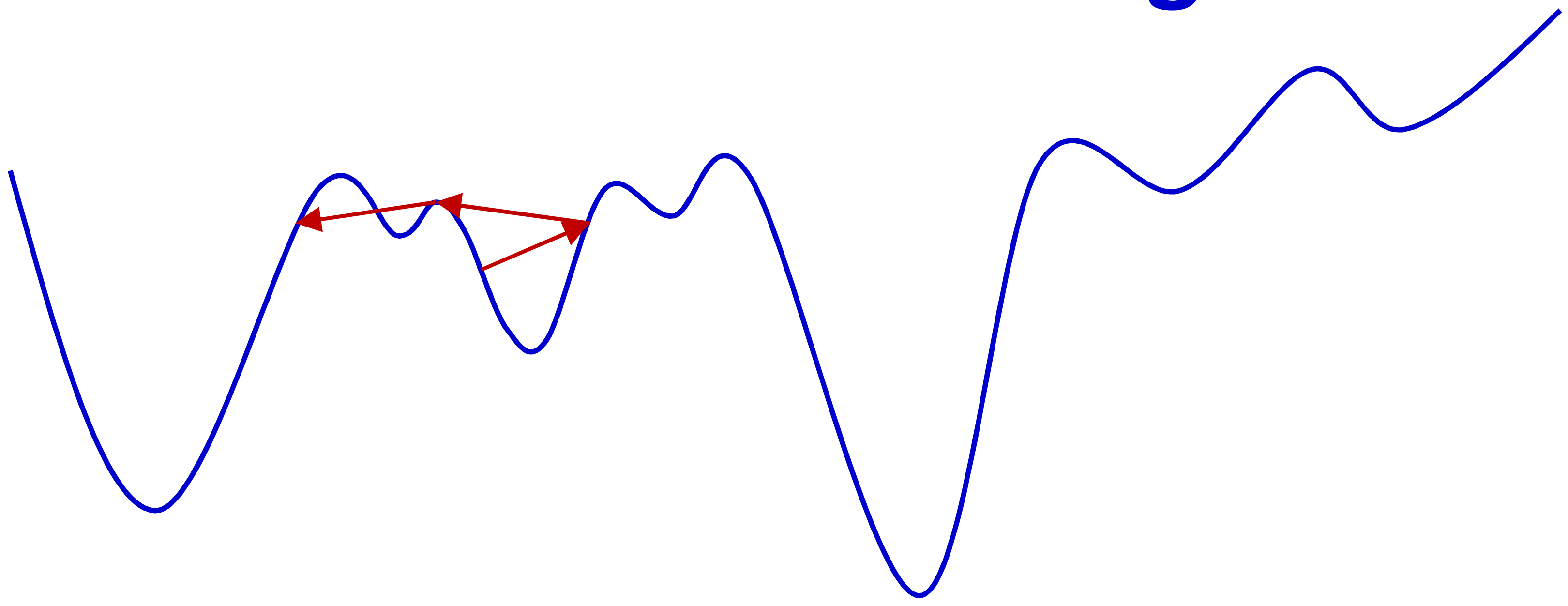
- A great many approaches have been proposed in the literature to *approximate* the Hessian in a number of ways and improve its positive definiteness
  - Boyden-Fletcher-Goldfarb-Shanno (BFGS)
    - And “low-memory” BFGS (L-BFGS)
    - Estimate Hessian from finite differences
  - Levenberg-Marquardt
    - Estimate Hessian from Jacobians
    - Diagonal load it to ensure positive definiteness
  - Other “Quasi-newton” methods
- Hessian estimates may even be *local* to a set of variables
- Not particularly popular anymore for large neural networks..

Issues: 2. The learning rate



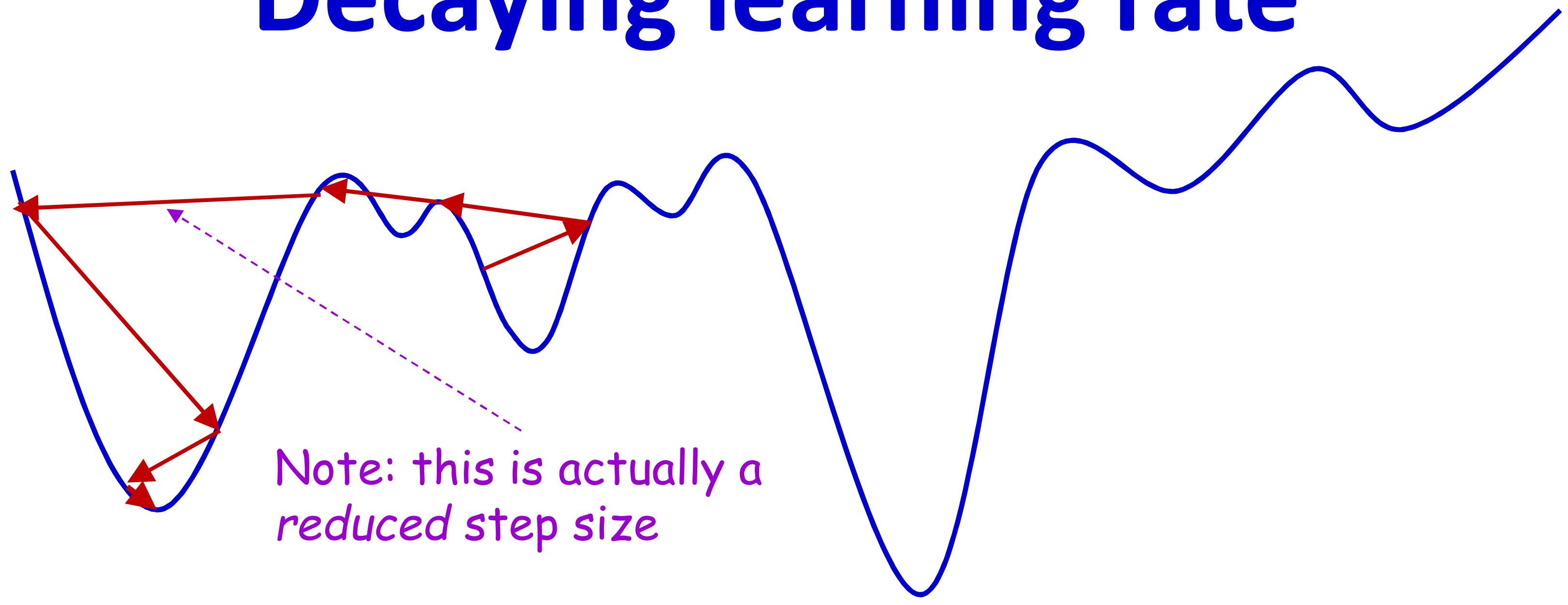
- Much of the analysis we just saw was based on trying to ensure that the step size was not so large as to cause divergence within a convex region
  - $\eta < 2\eta_{opt}$

Issues: 2. The learning rate



- For complex models such as neural networks the loss function is often not convex
  - Having $\eta > 2\eta_{opt}$ can actually help escape local optima
- However *always* having $\eta > 2\eta_{opt}$ will ensure that you never ever actually find a solution

Decaying learning rate



- Start with a large learning rate
  - Greater than 2 (assuming Hessian normalization)
  - Gradually reduce it with iterations

Decaying learning rate

- Typical decay schedules
  - Linear decay: $\eta_k = \frac{\eta_0}{k+1}$
  - Quadratic decay: $\eta_k = \frac{\eta_0}{(k+1)^2}$
  - Exponential decay: $\eta_k = \eta_0 e^{-\beta k}$, where $\beta > 0$

- A common approach (for nnets):
  1. Train with a fixed learning rate η until loss (or performance on a held-out data set) stagnates
  2. $\eta \leftarrow \alpha\eta$, where $\alpha < 1$ (typically 0.1)
  3. Return to step 1 and continue training from where we left off

Story so far : Convergence

- Gradient descent can miss obvious answers
  - And this may be a *good* thing

- Convergence issues abound
  - The loss surface has many saddle points
    - Although, perhaps, not so many bad local minima
    - Gradient descent can stagnate on saddle points
  - Vanilla gradient descent may not converge, or may converge toooooo slowly
    - The optimal learning rate for one component may be too high or too low for others

Story so far : Second-order methods

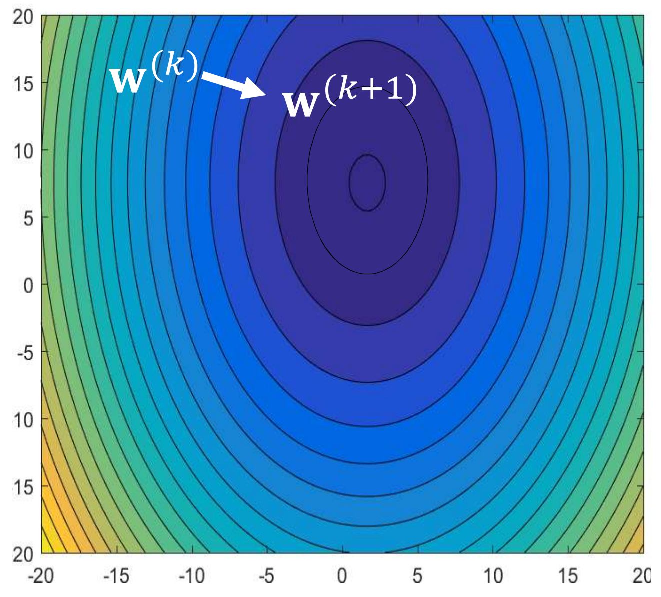
- Second-order methods "normalize" the variation along the components to mitigate the problem of different optimal learning rates for different components
  - But this requires computation of inverses of second-order derivative matrices
  - Computationally infeasible
  - Not stable in non-convex regions of the loss surface
  - Approximate methods address these issues, but simpler solutions may be better

Story so far : Learning rate

- Divergence-causing learning rates may not be a bad thing
  - Particularly for ugly loss functions
- *Decaying* learning rates provide good compromise between escaping poor local minima and convergence

- *Many of the convergence issues arise because we force the same learning rate on all parameters*

Lets take a step back



$$\boxed{\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta(\nabla_{\mathbf{w}} E)^T}$$

$$\boxed{w_i^{(k+1)} = w_i^{(k)} - \eta \frac{dE\left(w_i^{(k)}\right)}{dw}}$$

- Problems arise because of requiring a fixed step size across all dimensions
  - Because step are “tied” to the gradient
- Lets try releasing this requirement

Derivative-*inspired* algorithms

- Algorithms that use derivative information for trends, but do not follow them absolutely

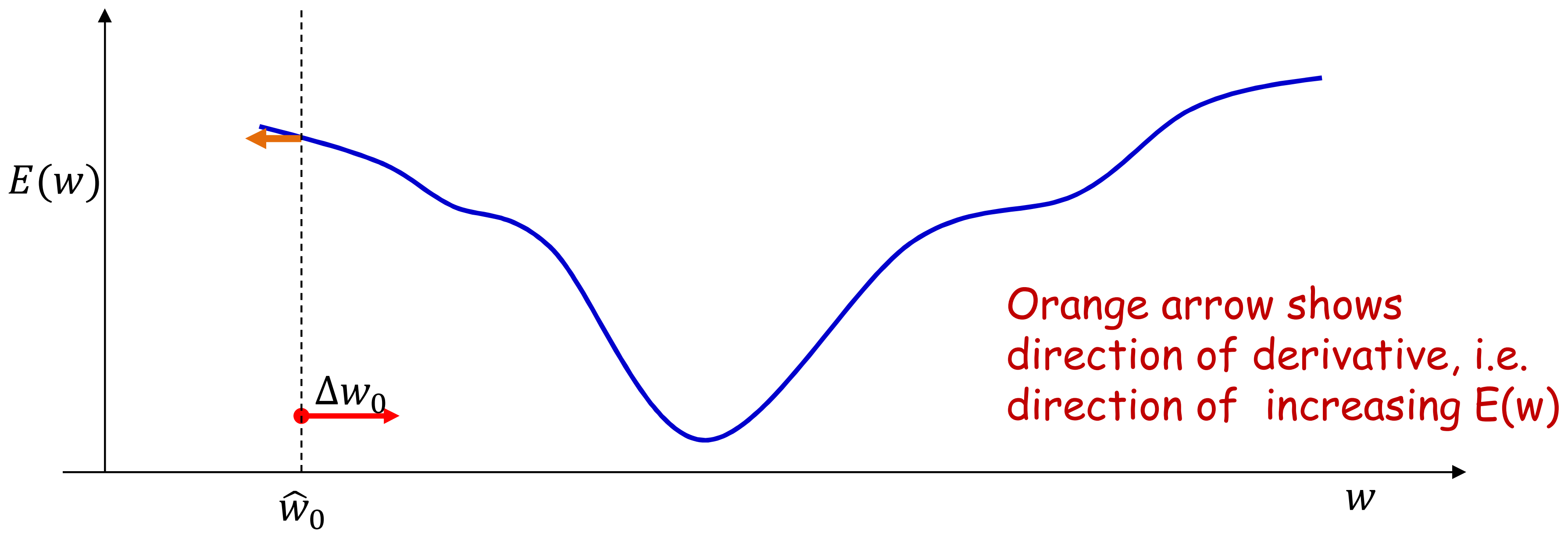
- Rprop
- Quick prop

RProp

- *Resilient* propagation
- Simple algorithm, to be followed *independently* for each component
  - I.e. steps in different directions are not coupled

- At each time
  - If the derivative at the current location recommends continuing in the same direction as before (i.e. has not changed sign from earlier):
    - *increase* the step, and continue in the same direction
  - If the derivative has changed sign (i.e. we've overshot a minimum)
    - *reduce* the step and reverse direction

Rprop



$E(w)$

Δw_0

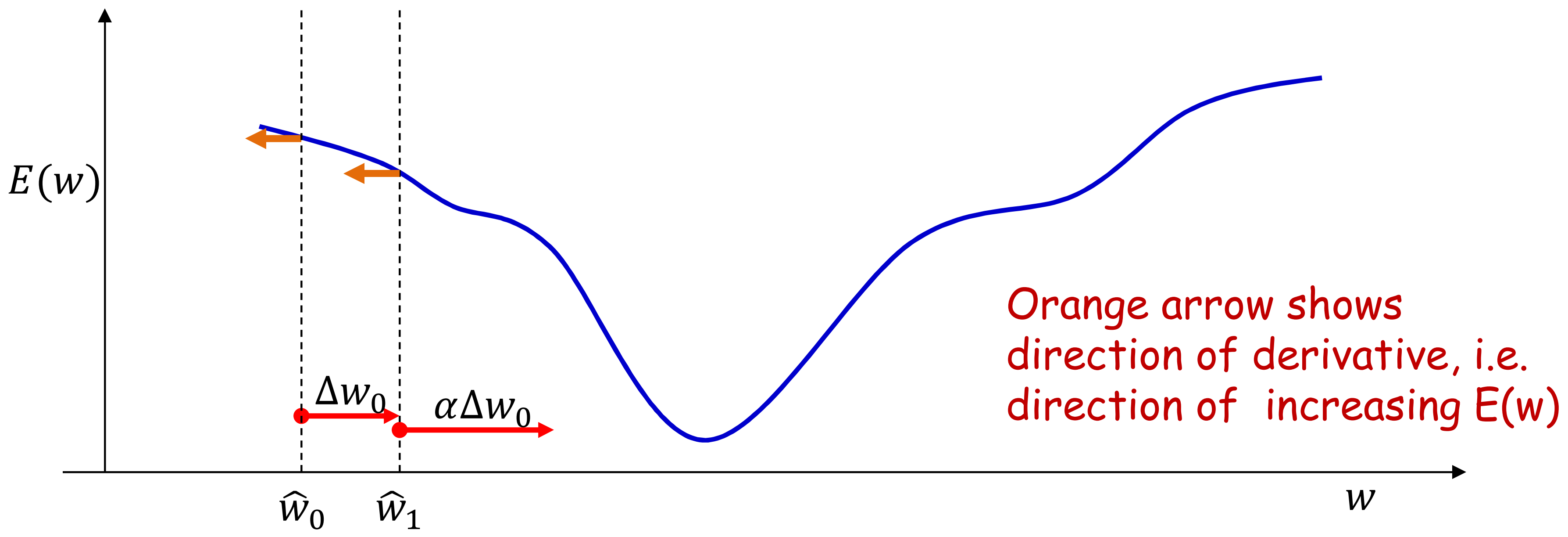
$\widehat{w}_0$

w

Orange arrow shows direction of derivative, i.e. direction of increasing E(w)

- Select an initial value $\widehat{w}$ and compute the derivative
  - Take an initial step Δw against the derivative
    - In the direction that reduces the function
  - $\Delta w = sign\left(\frac{dE(\widehat{w})}{dw}\right)\Delta w$
  - $\widehat{w} = \widehat{w} - \Delta w$

Rprop



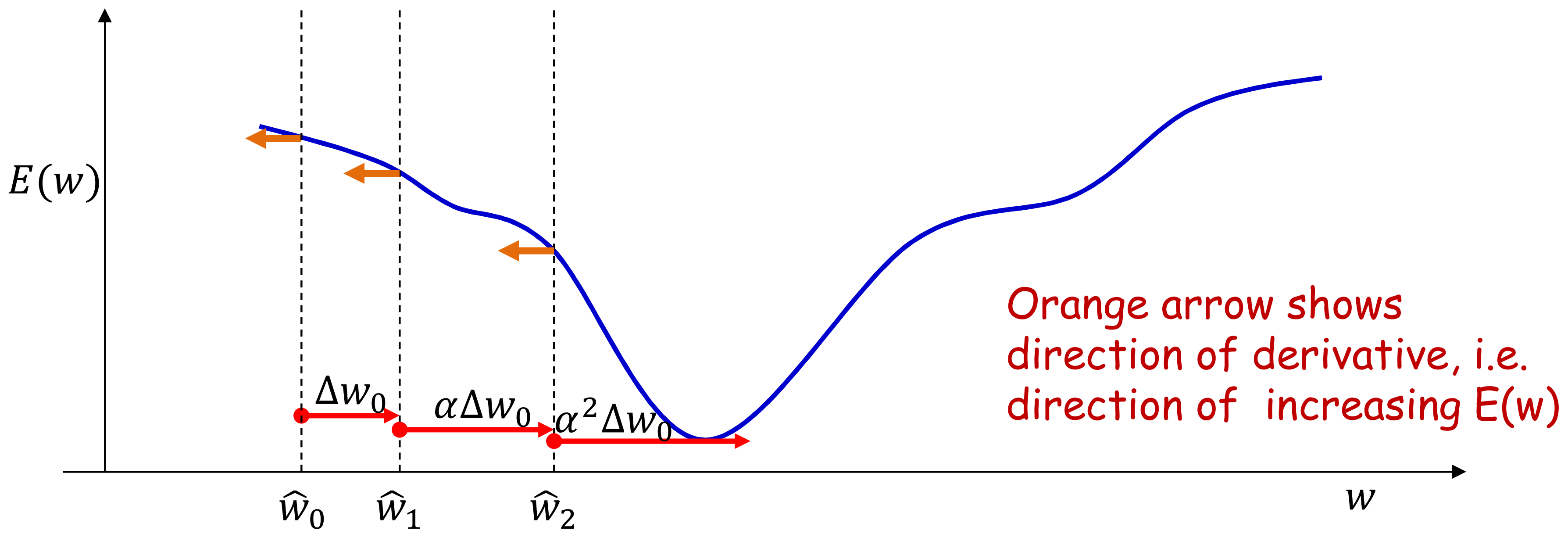
$E(w)$
Δw_0
$\alpha \Delta w_0$
$\hat{w}_0$
$\hat{w}_1$
w
Orange arrow shows direction of derivative, i.e. direction of increasing E(w)

- Compute the derivative in the new location
  - If the derivative has not changed sign from the previous location, increase the step size and take a longer step

$\boxed{\alpha > 1}$

- $\Delta w = \alpha \Delta w$
- $\hat{w} = \hat{w} - \Delta w$

Rprop

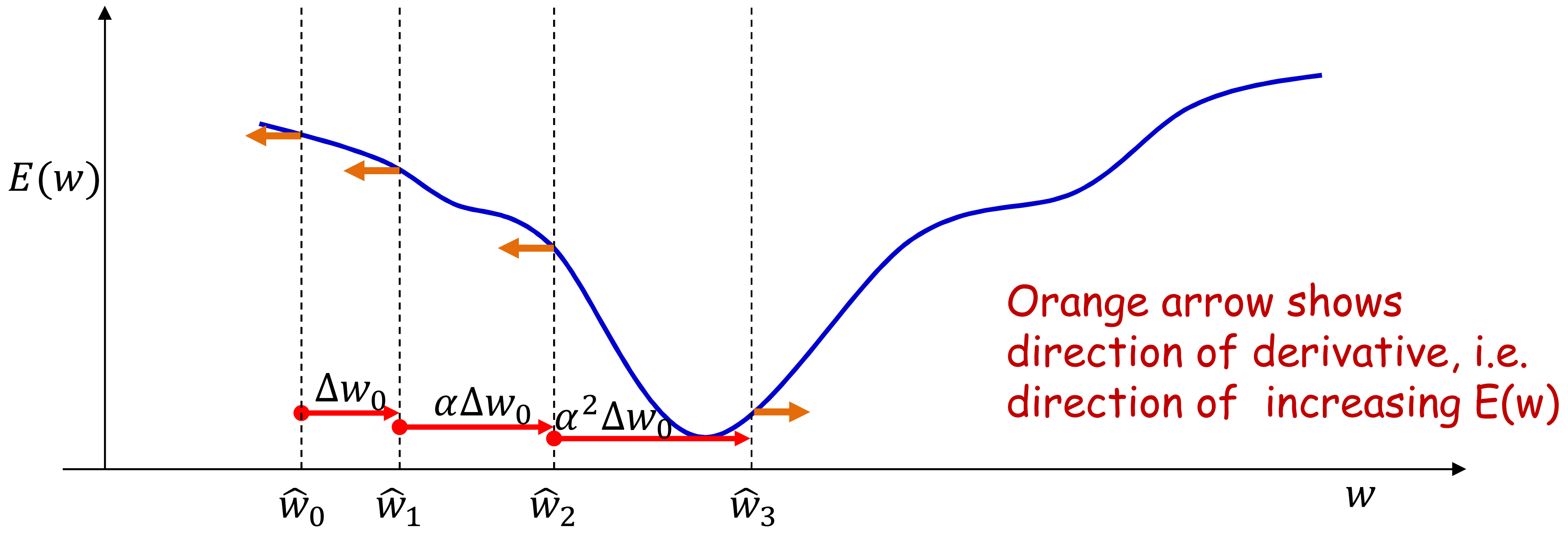


- Compute the derivative in the new location
  - If the derivative has not changed sign from the previous location, increase the step size and take a step

$\boxed{\alpha > 1}$

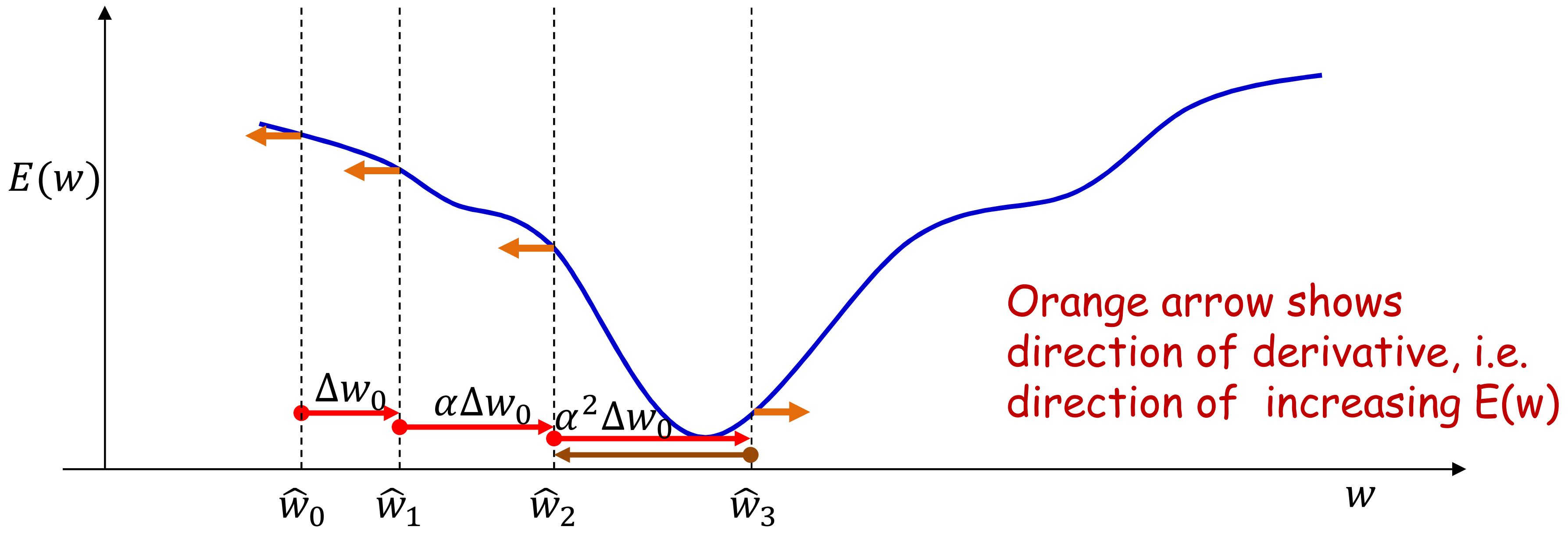
- $\Delta w = \alpha \Delta w$
- $\hat{w} = \hat{w} - \Delta w$

Rprop



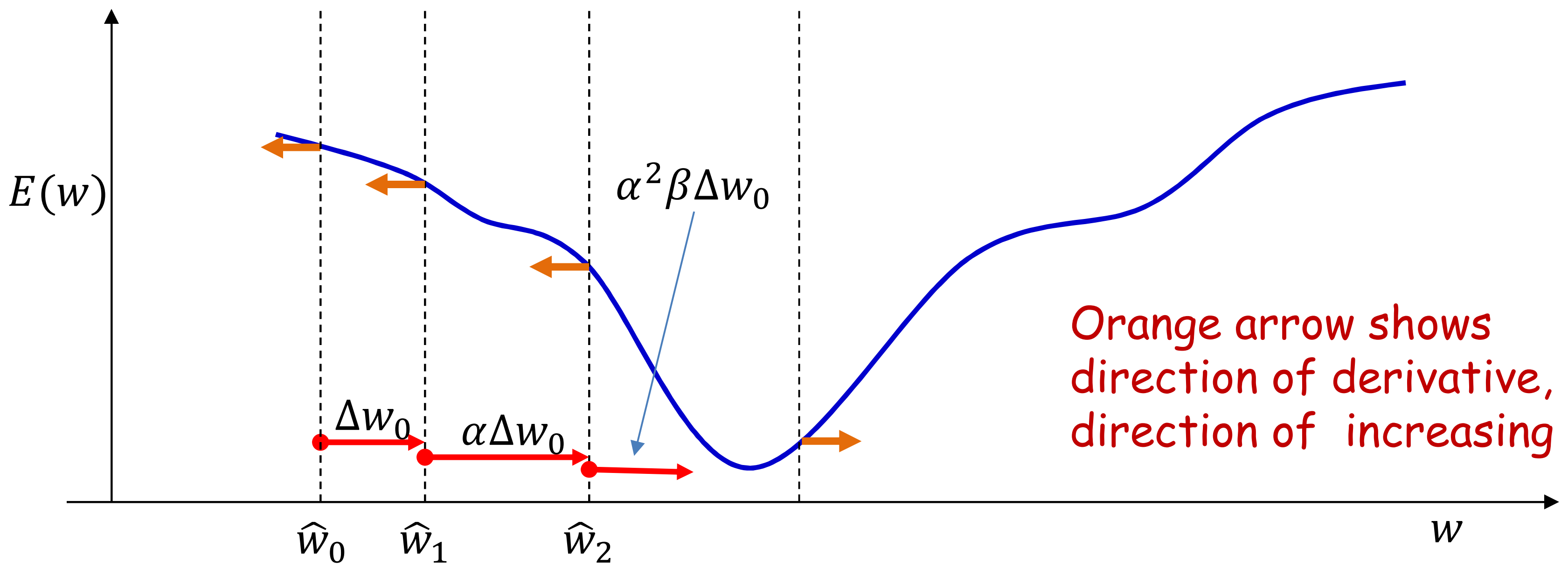
- Compute the derivative in the new location
  - If the derivative has changed sign

Rprop



- Compute the derivative in the new location
  - If the derivative has changed sign
  - Return to the previous location
    - $\hat{w} = \hat{w} + \Delta w$

Rprop

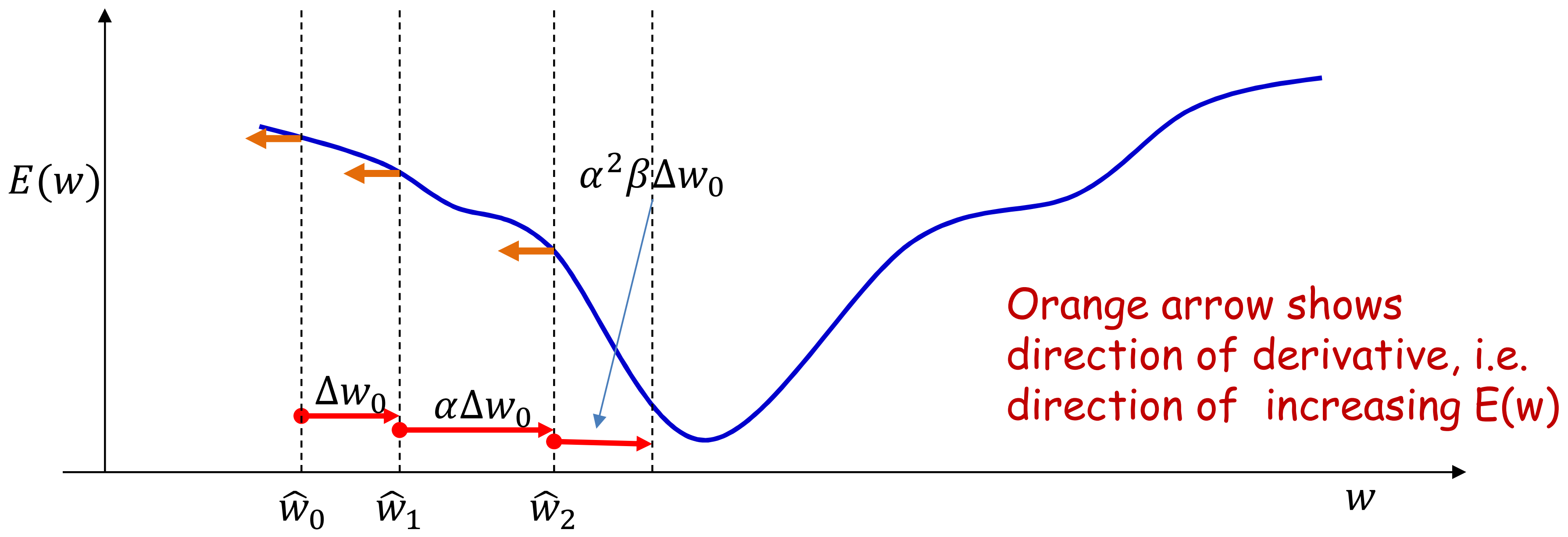


Orange arrow shows direction of derivative, i.e. direction of increasing E(w)

- Compute the derivative in the new location
  - If the derivative has changed sign
  - Return to the previous location
    - $\hat{w} = \hat{w} + \Delta w$
  - Shrink the step
    - $\Delta w = \beta \Delta w$

$\boxed{\beta < 1}$

Rprop



$E(w)$
$\alpha^2\beta\Delta w_0$
Δw_0
$\alpha\Delta w_0$
$\hat{w}_0$
$\hat{w}_1$
$\hat{w}_2$
w
Orange arrow shows direction of derivative, i.e. direction of increasing E(w)

- Compute the derivative in the new location
  - If the derivative has changed sign
  - Return to the previous location
    - $\hat{w} = \hat{w} + \Delta w$
  - Shrink the step
    - $\Delta w = \beta \Delta w$
  - Take the smaller step forward
    - $\hat{w} = \hat{w} - \Delta w$

$\boxed{\beta < 1}$

Rprop (simplified)

- Set $\alpha = 1.2, \beta = 0.5$
- For each layer l, for each i, j:
  - Initialize $w_{l,i,j}, \Delta w_{l,i,j} > 0,$
  - $prevD(l,i,j) = \frac{dErr(w_{l,i,j})}{dw_{l,i,j}}$
  - $\Delta w_{l,i,j} = \text{sign}\big(prevD(l,i,j)\big)\Delta w_{l,i,j}$
  - While not converged:
    - $w_{l,i,j} = w_{l,i,j} - \Delta w_{l,i,j}$
    - $D(l,i,j) = \frac{dErr(w_{l,i,j})}{dw_{l,i,j}}$
    - If $\text{sign}\big(prevD(l,i,j)\big) == \text{sign}\big(D(l,i,j)\big)$:
      - $\Delta w_{l,i,j} = \min(\alpha \Delta w_{l,i,j}, \Delta_{max})$
      - $prevD(l,i,j) = D(l,i,j)$
    - else:
      - $w_{l,i,j} = w_{l,i,j} + \Delta w_{l,i,j}$
      - $\Delta w_{l,i,j} = \max(\beta \Delta w_{l,i,j}, \Delta_{min})$

Ceiling and floor on step

Rprop (simplified)

- Set $\alpha = 1.2, \beta = 0.5$
- For each layer l, for each i, j:
  - Initialize $w_{l,i,j}$, $\Delta w_{l,i,j} > 0$,
  - $prevD(l,i,j) = \frac{dErr(w_{l,i,j})}{dw_{l,i,j}}$
  - $\Delta w_{l,i,j} = \text{sign}\big(prevD(l,i,j)\big)\Delta w_{l,i,j}$
  - While not converged:
    - $w_{l,i,j} = w_{l,i,j} - \Delta w_{l,i,j}$
    - $D(l,i,j) = \frac{dErr(w_{l,i,j})}{dw_{l,i,j}}$
    - If $\text{sign}\big(prevD(l,i,j)\big) == \text{sign}\big(D(l,i,j)\big)$:
      - $\Delta w_{l,i,j} = \alpha \Delta w_{l,i,j}$
      - $prevD(l,i,j) = D(l,i,j)$
    - else:
      - $w_{l,i,j} = w_{l,i,j} + \Delta w_{l,i,j}$
      - $\Delta w_{l,i,j} = \beta \Delta w_{l,i,j}$

Obtained via backprop

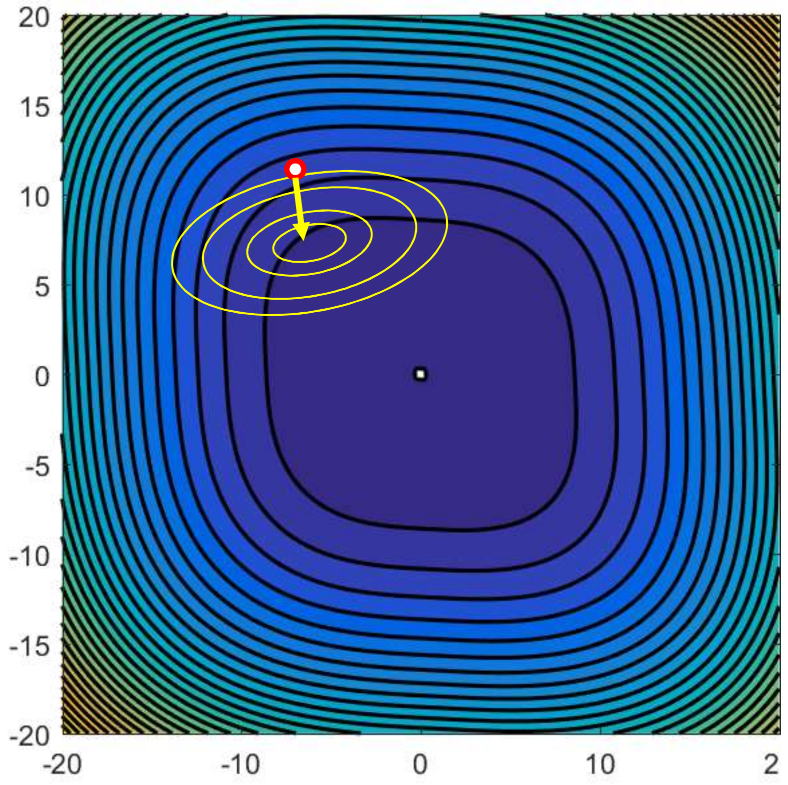
Note: Different parameters updated independently

RProp

- A remarkably simple first-order algorithm, that is frequently much more efficient than gradient descent.
  - And can even be competitive against some of the more advanced second-order methods

- Only makes minimal assumptions about the loss function
  - No convexity assumption

QuickProp

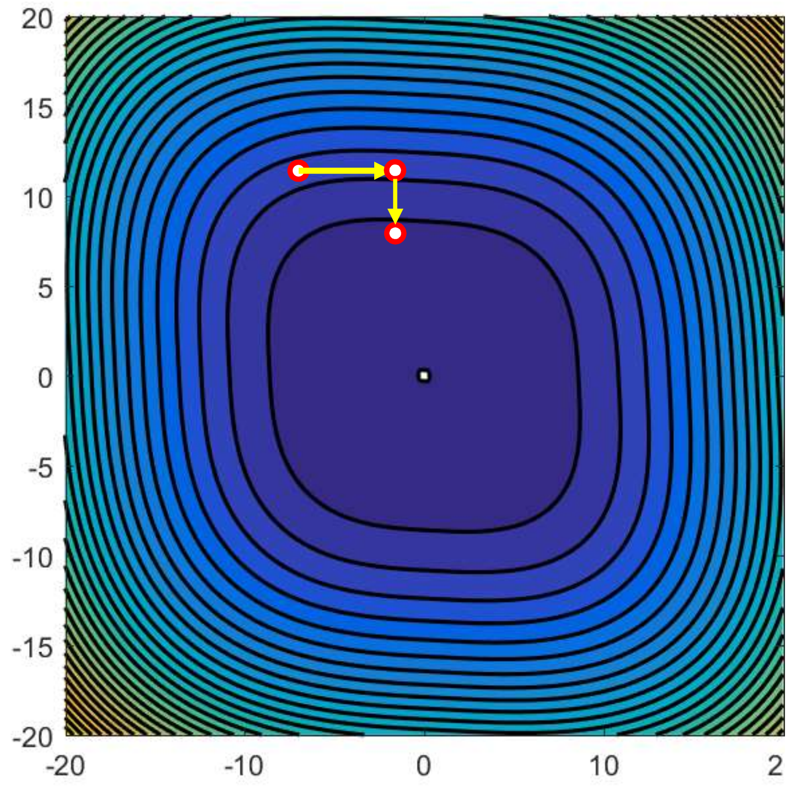


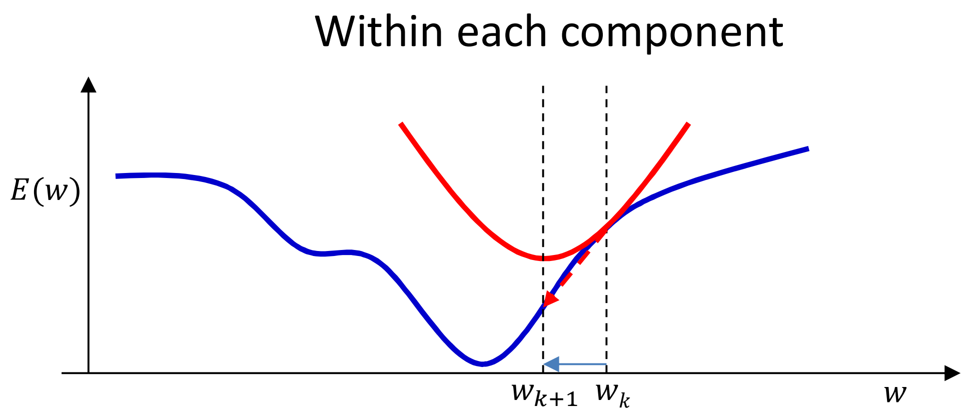
- Quickprop employs the Newton updates with two modifications

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta H_E\left(\mathbf{w}^{(k)}\right)^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)^T$$

- But with two modifications

QuickProp: Modification 1



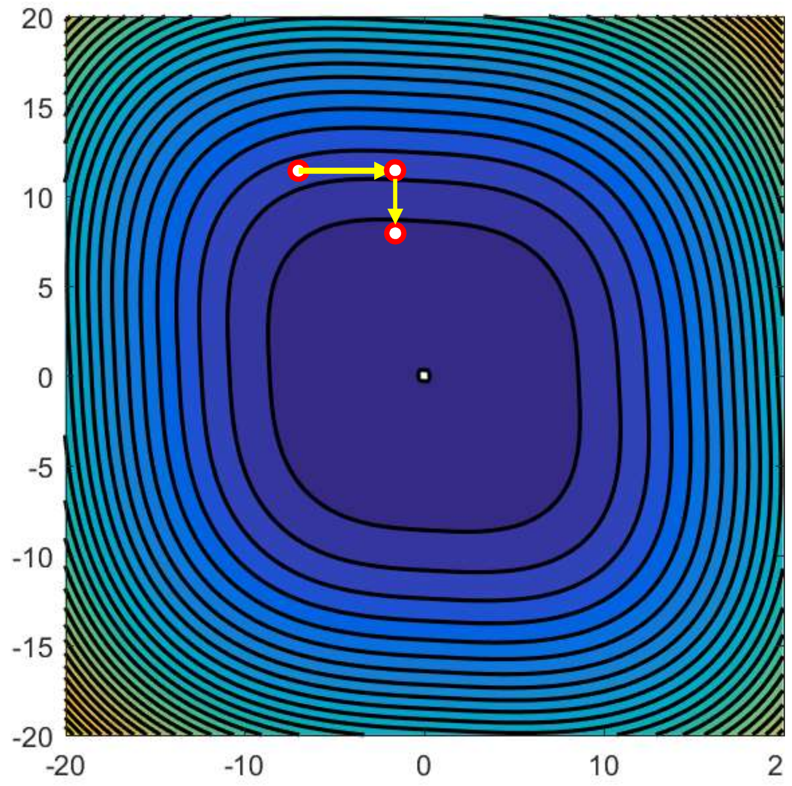


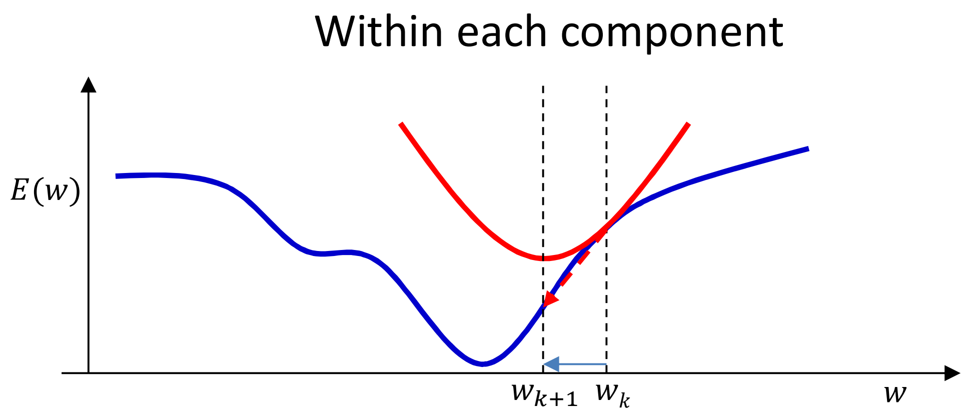
- It treats each dimension independently
- For $i = 1:N$

$$w_i^{k+1} = w_i^k - E''\left(w_i^k | w_j^k, j \neq i\right)^{-1} E'\left(w_i^k | w_j^k, j \neq i\right)$$

- This eliminates the need to compute and invert expensive Hessians

QuickProp: Modification 2





- **It approximates the second derivative through finite differences**
- For $i = 1 : N$

$$w_i^{k+1} = w_i^k - D\left(w_i^k, w_i^{k-1}\right)^{-1} E'\left(w_i^k | w_j^k, j \neq i\right)$$

- This eliminates the need to compute expensive double derivatives

QuickProp

$$w^{(k+1)} = w^{(k)} - \underbrace{\left(\frac{E'\left(w^{(k)}\right) - E'\left(w^{(k-1)}\right)}{\Delta w^{(k-1)}}\right)}_{\text{Finite-difference approximation to double derivative obtained assuming a quadratic } E()}^{-1} E'\left(w^{(k)}\right)$$

- Updates are independent for every parameter
- For every layer l, for every connection from node i in the $(l-1)$th layer to node j in the lth layer:

$$\Delta w_{l,ij}^{(k)} = \frac{\Delta w_{l,ij}^{(k-1)}}{Err'\left(w_{l,ij}^{(k)}\right) - Err'\left(w_{l,ij}^{(k-1)}\right)} Err'\left(w_{l,ij}^{(k)}\right)$$

$$w_{l,ij}^{(k+1)} = w_{l,ij}^{(k)} - \Delta w_{l,ij}^{(k)}$$

QuickProp

$$w^{(k+1)} = w^{(k)} - \underbrace{\left(\frac{E'\left(w^{(k)}\right) - E'\left(w^{(k-1)}\right)}{\Delta w^{(k-1)}}\right)}_{\text{Finite-difference approximation to double derivative obtained assuming a quadratic } E()}^{-1} E'\left(w^{(k)}\right)$$

- Updates are independent for every parameter
- For every layer l, for every connection from node i in the $(l-1)^{\text{th}}$ layer to node j in the l^{th} layer:

$$\Delta w_{l,ij}^{(k)} = \frac{\Delta w_{l,ij}^{(k-1)}}{Err'\left(w_{l,ij}^{(k)}\right) - Err'\left(w_{l,ij}^{(k-1)}\right)} \underbrace{Err'\left(w_{l,ij}^{(k)}\right)}_{\text{Computed using backprop}}$$

$$w_{l,ij}^{(k+1)} = w_{l,ij}^{(k)} - \Delta w_{l,ij}^{(k)}$$

Quickprop

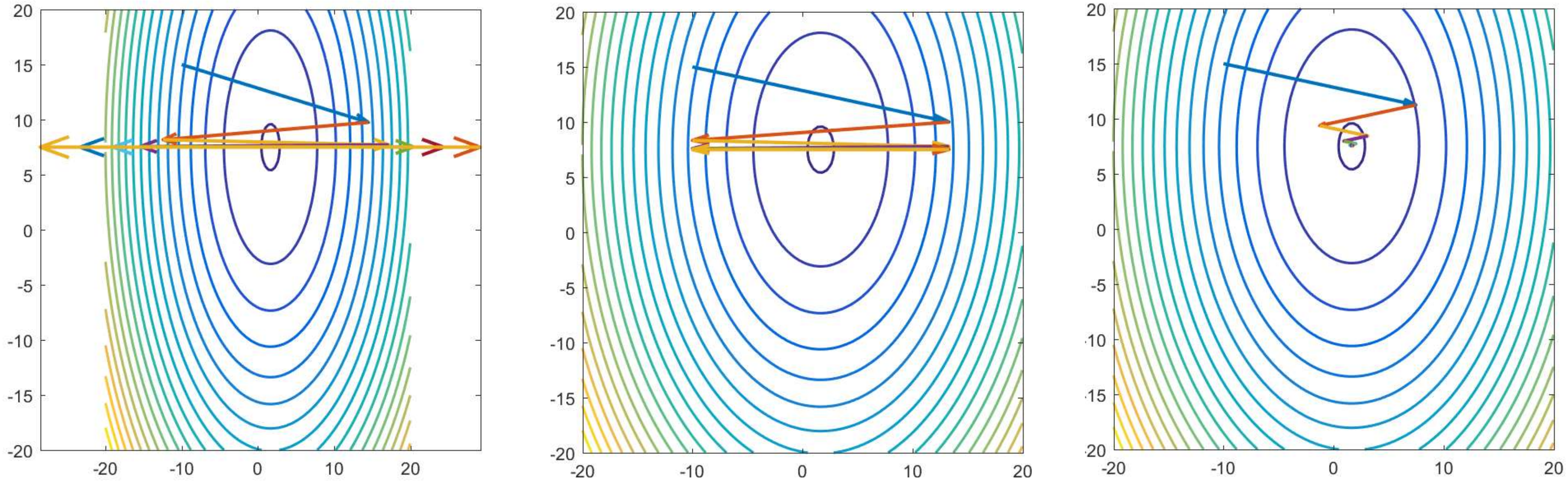
- Prone to some instability for non-convex objective functions

- But is still one of the fastest training algorithms for many problems

Story so far : Convergence

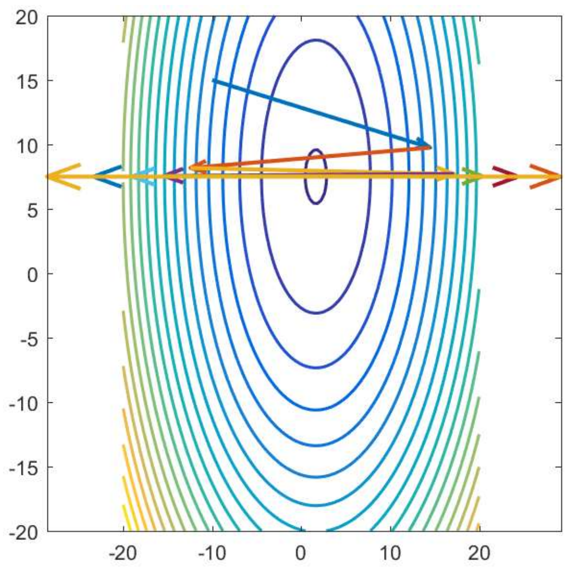
- Gradient descent can miss obvious answers
  - And this may be a *good* thing
- Vanilla gradient descent may be too slow or unstable due to the differences between the dimensions
- Second order methods can normalize the variation across dimensions, but are complex
- Adaptive or decaying learning rates can improve convergence
- Methods that decouple the dimensions can improve convergence

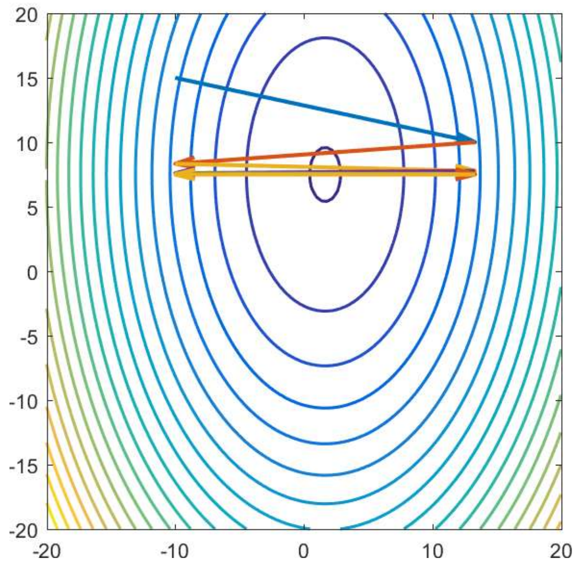
A closer look at the convergence problem

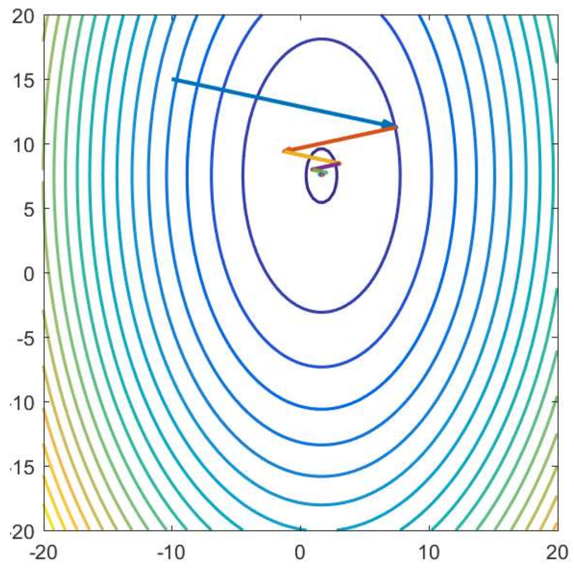


- With dimension-independent learning rates, the solution will converge smoothly in some directions, but oscillate or diverge in others

A closer look at the convergence problem







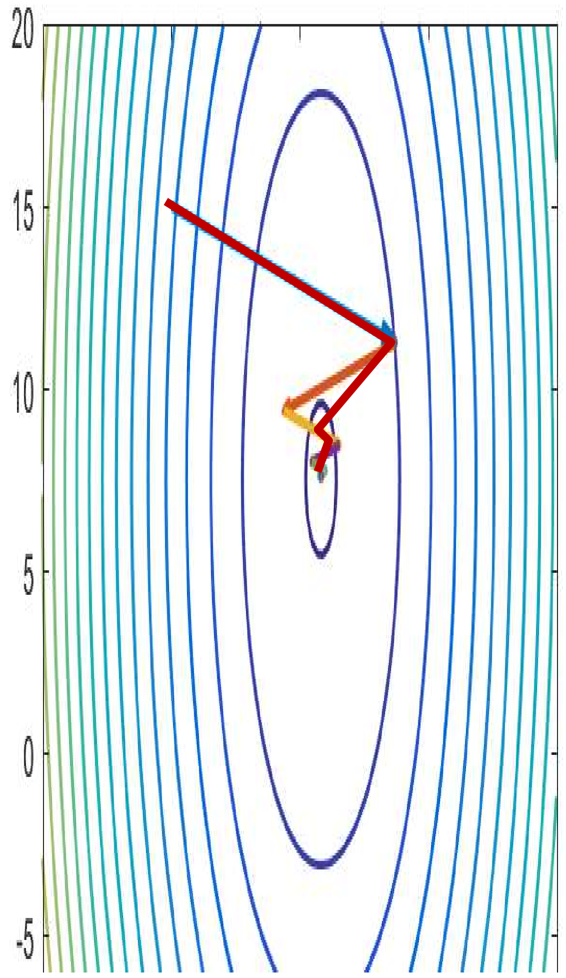
- With dimension-independent learning rates, the solution will converge smoothly in some directions, but oscillate or diverge in others

- **Proposal:**
  - Keep track of oscillations
  - Emphasize steps in directions that converge smoothly
  - Shrink steps in directions that bounce around..

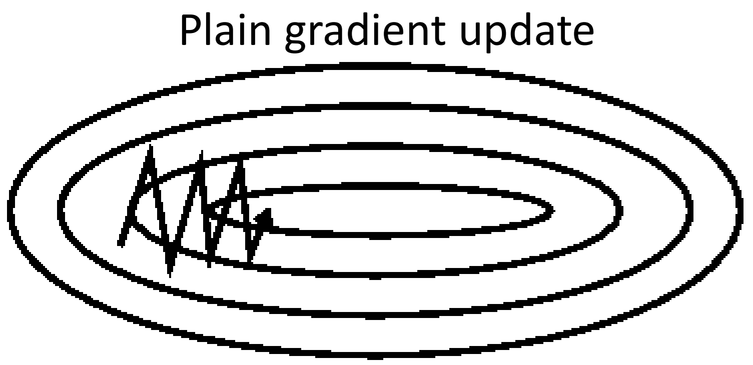
The momentum methods

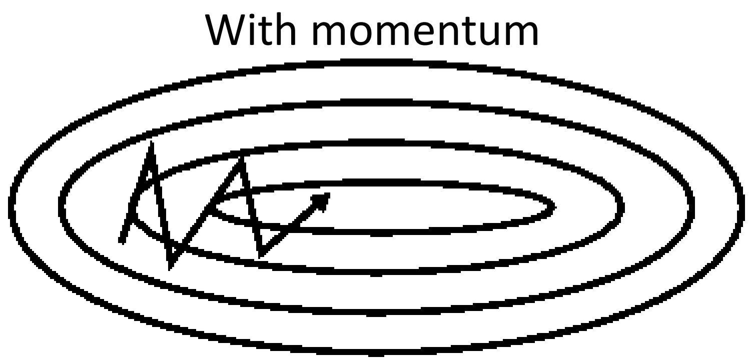
- Maintain a running average of all past steps
  - In directions in which the convergence is smooth, the average will have a large value
  - In directions in which the estimate swings, the positive and negative swings will cancel out in the average

- Update with the running average, rather than the current gradient



Momentum Update





- The momentum method maintains a running average of all gradients until the *current* step

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Loss\left(W^{(k-1)}\right)^T$$

$$W^{(k)} = W^{(k-1)} + \Delta W^{(k)}$$

  - Typical β value is 0.9
- The running average steps
  - Get longer in directions where gradient stays in the same sign
  - Become shorter in directions where the sign keeps flipping

Training by gradient descent

- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_K$
- Do:
  - For all $i, j, k,$ initialize $\nabla_{W_k} Loss = 0$
  - For all $t = 1{:}T$
    - For every layer k:
      - Compute $\nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
      - Compute $\nabla_{W_k} Loss \mathrel{+}= \frac{1}{T} \nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
  - For every layer k:

$$W_k = W_k - \eta (\nabla_{W_k} Loss)^T$$

- Until $Loss$ has converged

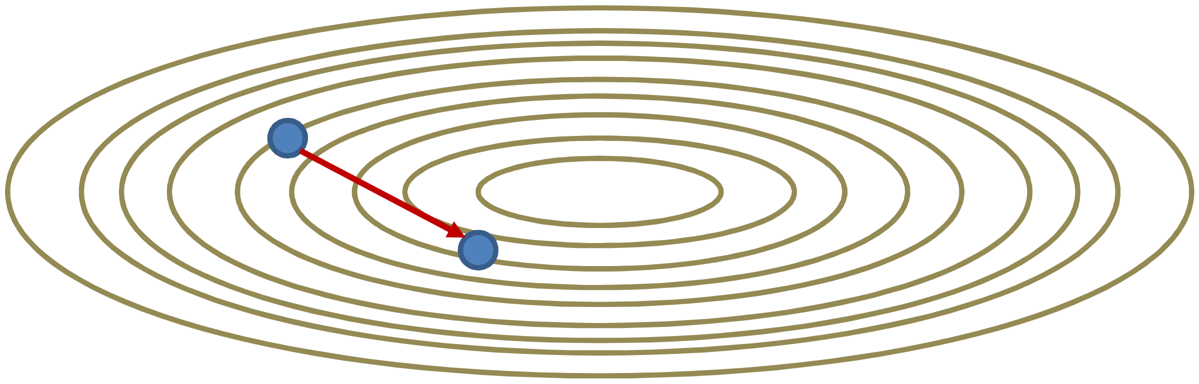
Training with momentum

- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_K$
- Do:
  - For all layers k, initialize $\nabla_{W_k} Loss = 0, \Delta W_k = 0$
  - For all $t = 1{:}T$
    - For every layer k:
      - Compute gradient $\nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
      - $\nabla_{W_k} Loss \mathrel{+}= \frac{1}{T} \nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
  - For every layer k

$$\Delta W_k = \beta \Delta W_k - \eta \left(\nabla_{W_k} Loss\right)^T$$
$$W_k = W_k + \Delta W_k$$

- Until $Loss$ has converged

Momentum Update

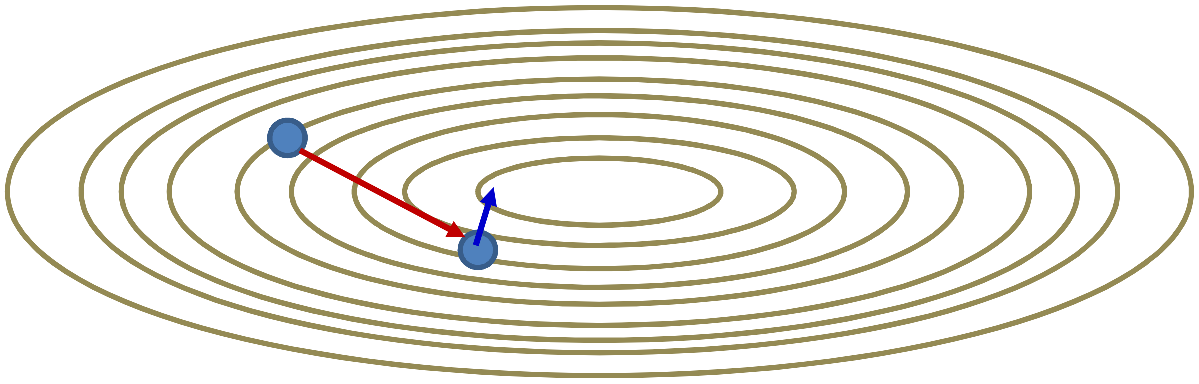


- The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Loss\left(W^{(k-1)}\right)^T$$

- At any iteration, to compute the current step:

Momentum Update

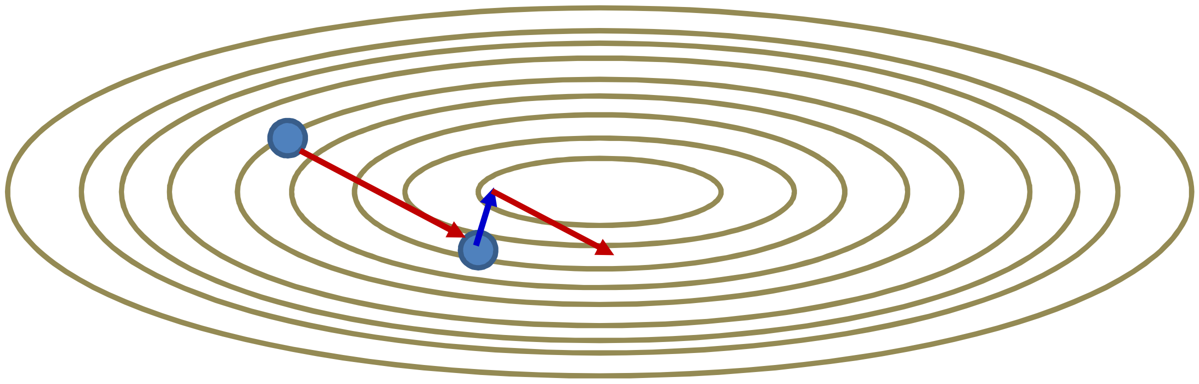


- The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Loss\left(W^{(k-1)}\right)^T$$

- At any iteration, to compute the current step:
  - First computes the gradient step at the current location

Momentum Update

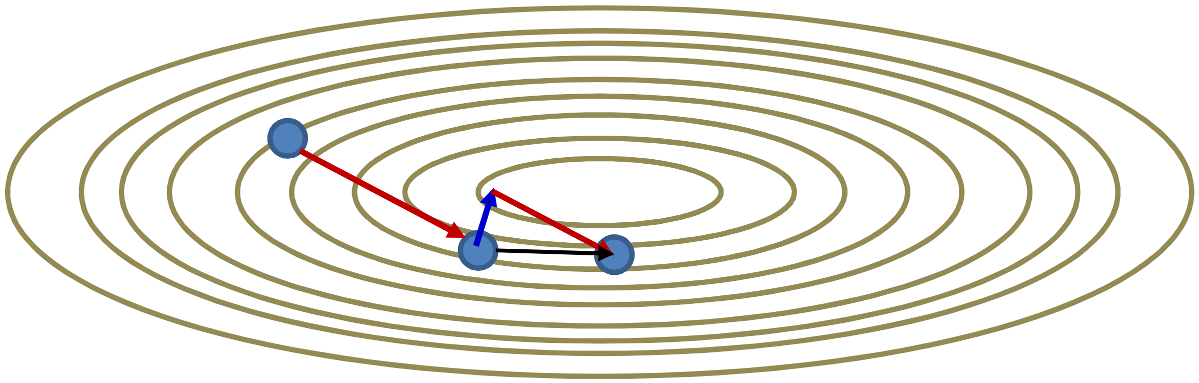


- The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Loss\left(W^{(k-1)}\right)^T$$

- At any iteration, to compute the current step:
  - First computes the gradient step at the current location
  - Then adds in the scaled *previous* step
    - Which is actually a running average

Momentum Update



- The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Loss\left(W^{(k-1)}\right)^T$$

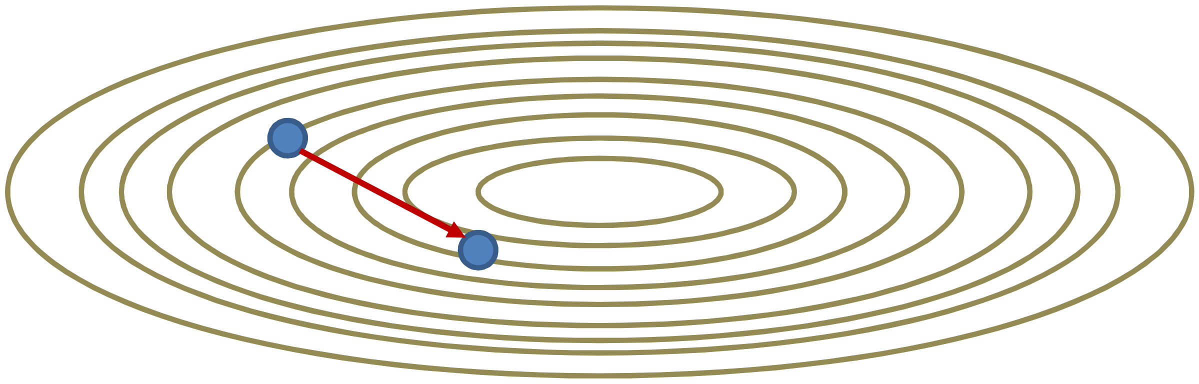
- At any iteration, to compute the current step:
  - First computes the gradient step at the current location
  - Then adds in the scaled *previous* step
    - Which is actually a running average
  - To get the final step

Momentum update

- Takes a step along the past running average *after* walking along the gradient

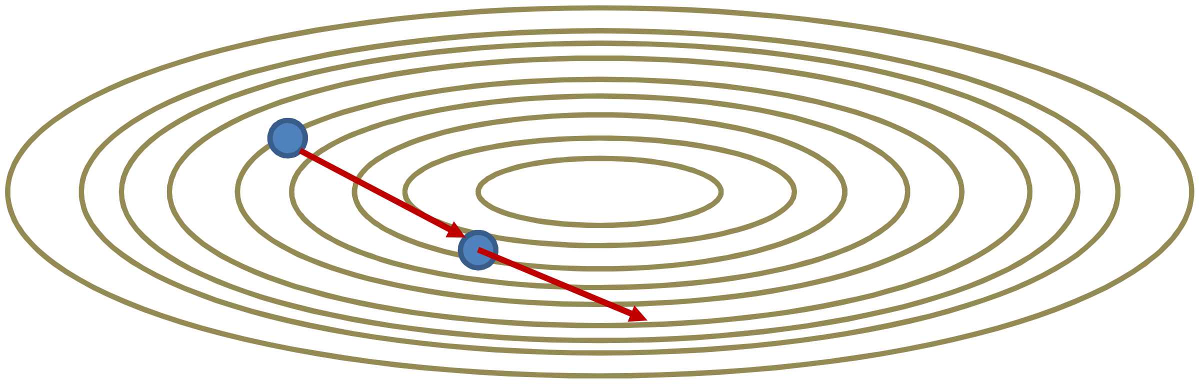
- The procedure can be made more optimal by reversing the order of operations..

Nestorov’s Accelerated Gradient



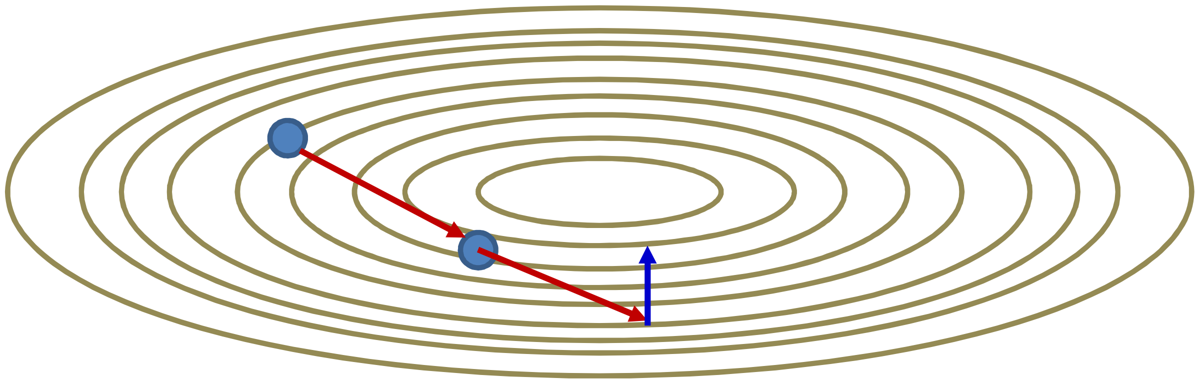
- Change the order of operations
- At any iteration, to compute the current step:

Nestorov's Accelerated Gradient



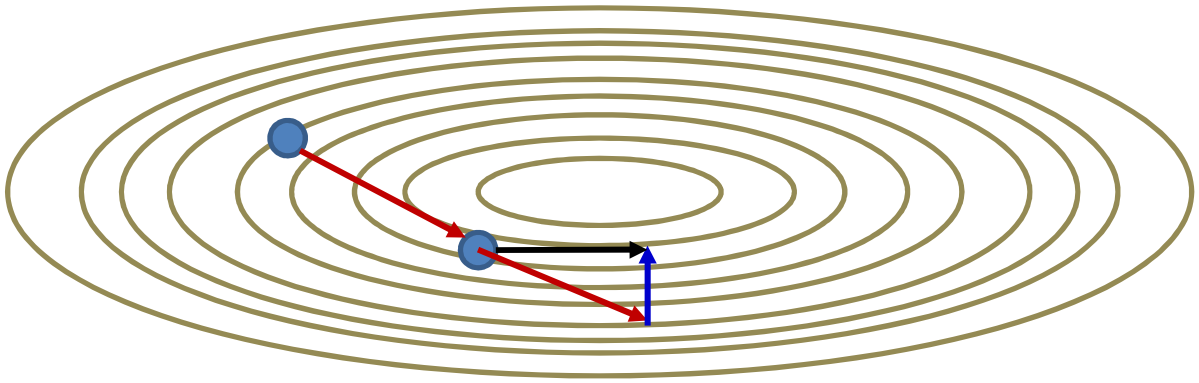
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step

Nestorov's Accelerated Gradient



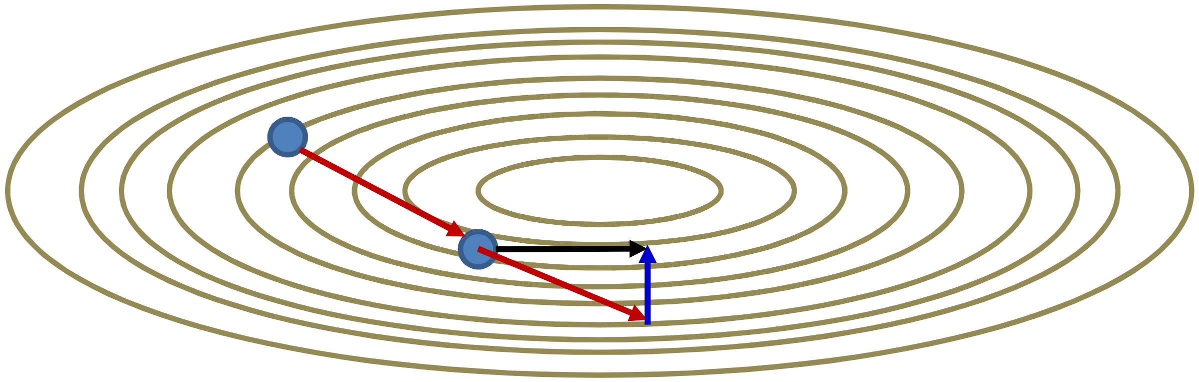
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step
  - Then compute the gradient step at the resultant position

Nestorov's Accelerated Gradient



- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step
  - Then compute the gradient step at the resultant position
  - Add the two to obtain the final step

Nestorov's Accelerated Gradient

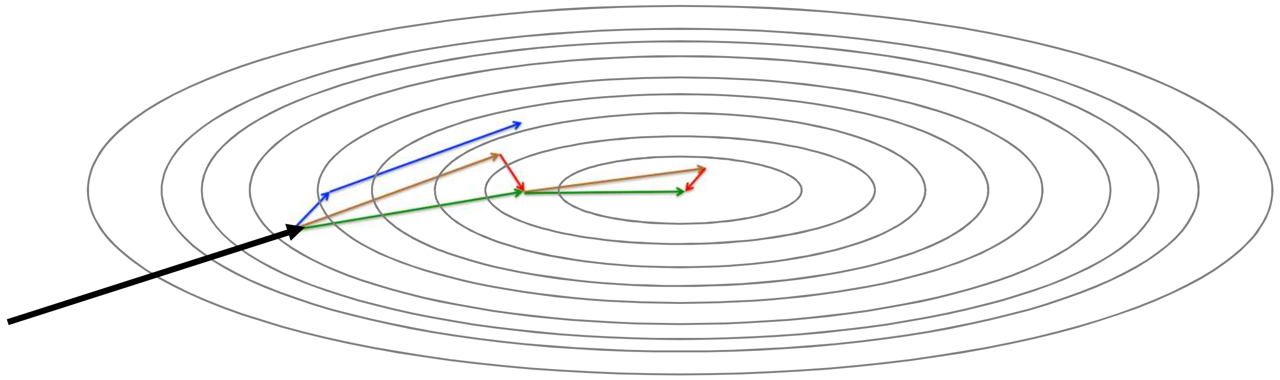


- Nestorov's method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Loss\left(W^{(k-1)} + \beta \Delta W^{(k-1)}\right)^T$$

$$W^{(k)} = W^{(k-1)} + \Delta W^{(k)}$$

Nestorov's Accelerated Gradient



- Comparison with momentum (example from Hinton)
- Converges much faster

Training with Nestorov

- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_K$
- Do:
  - For all layers k, initialize $\nabla_{W_k} Loss = 0$, $\Delta W_k = 0$
  - For every layer k

$$W_k = W_k + \beta \Delta W_k$$

  - For all $t = 1{:}T$
    - For every layer k:
      - Compute gradient $\nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
      - $\nabla_{W_k} Loss \mathrel{+}= \frac{1}{T} \nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
  - For every layer k

$$W_k = W_k - \eta (\nabla_{W_k} Loss)^T$$

$$\Delta W_k = \beta \Delta W_k - \eta (\nabla_{W_k} Loss)^T$$

- Until $Loss$ has converged

Momentum and trend-based methods..

- We will return to this topic again, very soon..

Story so far : Convergence

- Gradient descent can miss obvious answers
  - And this may be a *good* thing

- Vanilla gradient descent may be too slow or unstable due to the differences between the dimensions

- Second order methods can normalize the variation across dimensions, but are complex

- Adaptive or decaying learning rates can improve convergence

- Methods that decouple the dimensions can improve convergence

- Momentum methods which emphasize directions of steady improvement are demonstrably superior to other methods

Coming up

- Incremental updates
- Revisiting “trend” algorithms
- Generalization
- Tricks of the trade
  - Divergences..
  - Activations
  - Normalizations