



Neural Networks: Optimization Part 1

Intro to Deep Learning, Spring 2019

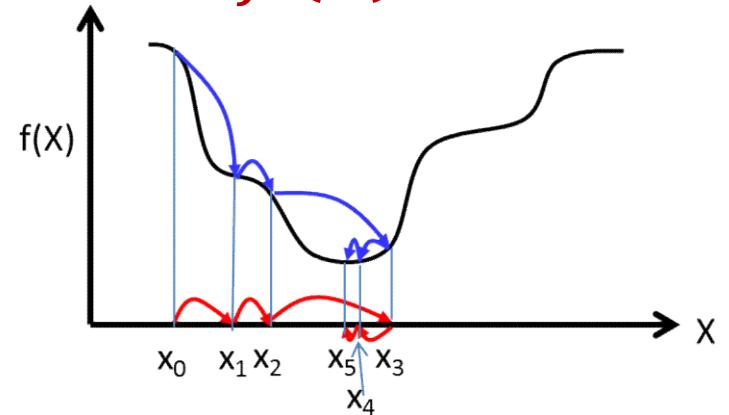
Story so far

- 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 “error” 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
 - $k = 0$
- While $|f(x^{k+1}) - f(x^k)| > \varepsilon$
 - $x^{k+1} = x^k - \eta \nabla_x f^T$
 - $k = k + 1$



Training Neural Nets by Gradient Descent

Total training error:

$$Err = \frac{1}{T} \sum_t Div(Y_t, d_t; W_1, W_2, \dots, W_K)$$

- Gradient descent algorithm:
- Initialize all weights W_1, W_2, \dots, W_K
- Do:
 - For every layer k , compute:
 - $\nabla_{W_k} Err = \frac{1}{T} \sum_t \nabla_{W_k} Div(Y_t, d_t)$
 - $W_k = W_k - \eta \nabla_{W_k} Err^T$
- Until Err has converged

Training Neural Nets by Gradient Descent

Total training error:

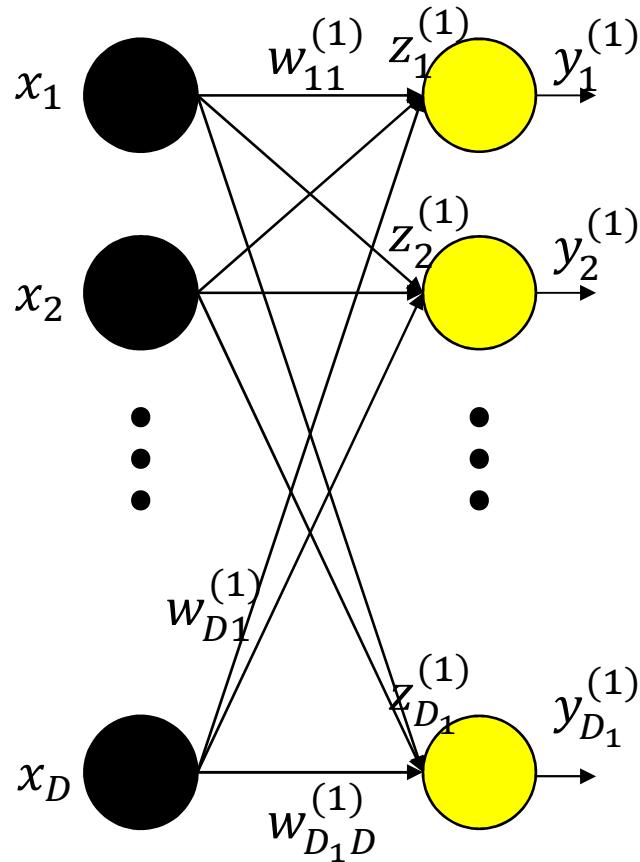
$$Err = \frac{1}{T} \sum_t Div(Y_t, d_t; W_1, W_2, \dots, W_K)$$

- Gradient descent algorithm:
- Initialize all weights W_1, W_2, \dots, W_K
- Do:
 - For every layer k , compute:
 - $\nabla_{W_k} Err = \frac{1}{T} \sum_t \nabla_{W_k} Div(Y_t, d_t)$
 - $W_k = W_k - \eta \nabla_{W_k} Err$
 - 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



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

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

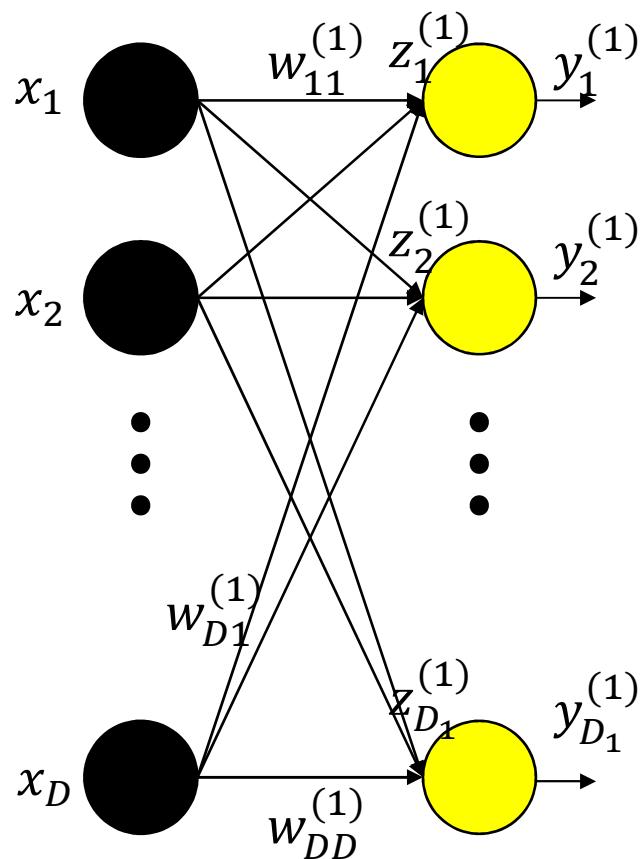
$$\mathbf{y}_k = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_{D_k}^{(k)} \end{bmatrix}$$

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

$$\mathbf{b}_k = \begin{bmatrix} b_1^{(k)} \\ b_2^{(k)} \\ \vdots \\ b_{D_{k+1}}^{(k)} \end{bmatrix}$$

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

Vector formulation



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

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

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

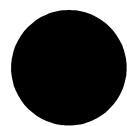
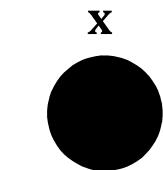
$$\mathbf{W}_k = \begin{bmatrix} w_{11}^{(k)} & w_{21}^{(k)} & \vdots & w_{D_{k-1}1}^{(k)} \\ w_{12}^{(k)} & w_{22}^{(k)} & \vdots & w_{D_{k-1}2}^{(k)} \\ \dots & \dots & \ddots & \vdots \\ w_{1D_k}^{(k)} & w_{2D_k}^{(k)} & \dots & w_{D_{k-1}D_k}^{(k)} \end{bmatrix} \quad \mathbf{b}_k = \begin{bmatrix}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

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

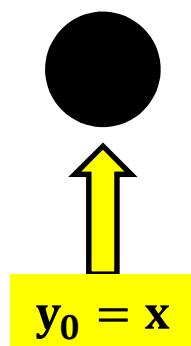
$$\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k$$

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

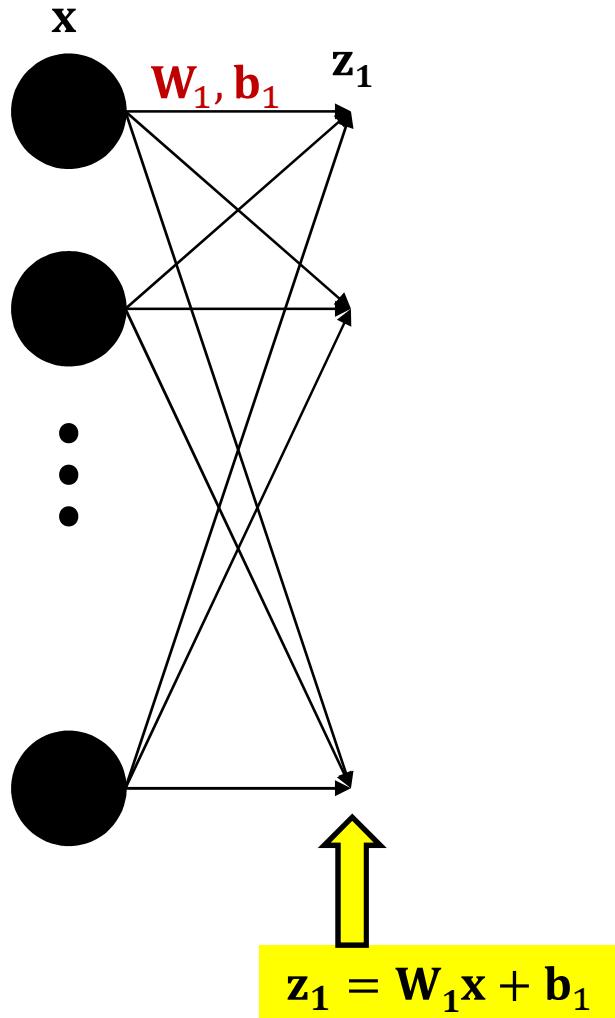
The forward pass: Evaluating the network



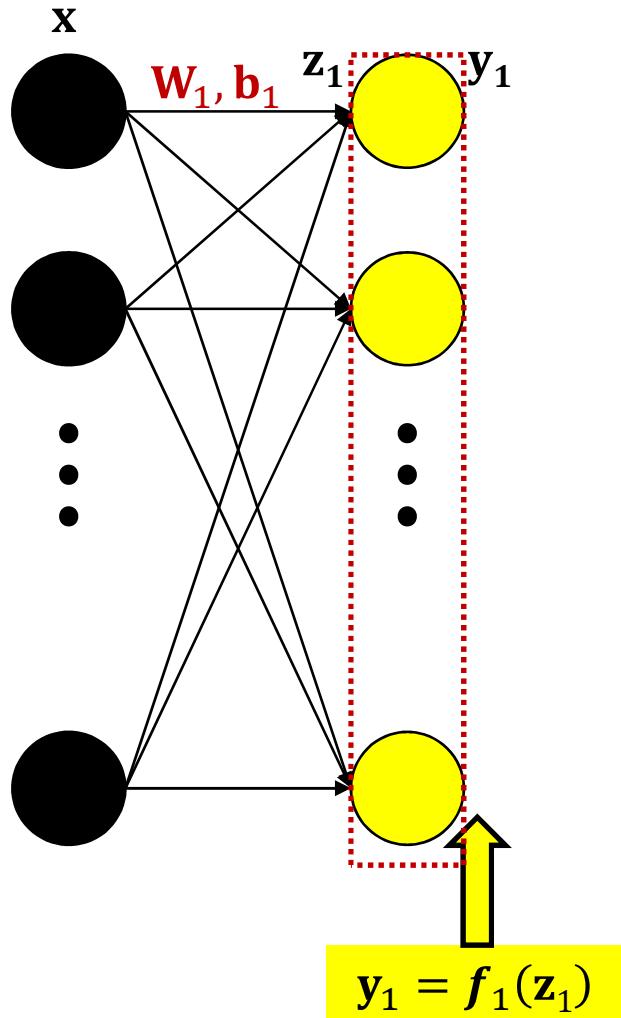
⋮



The forward pass



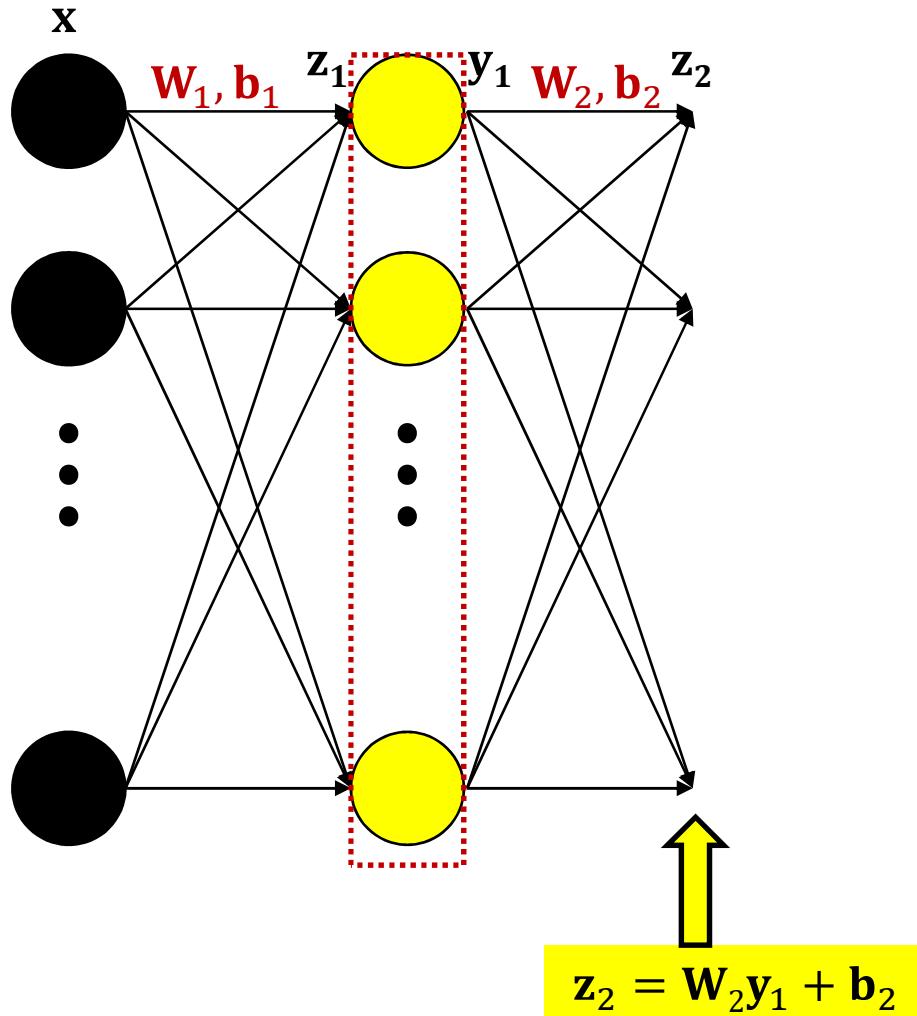
The forward pass



The Complete computation

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

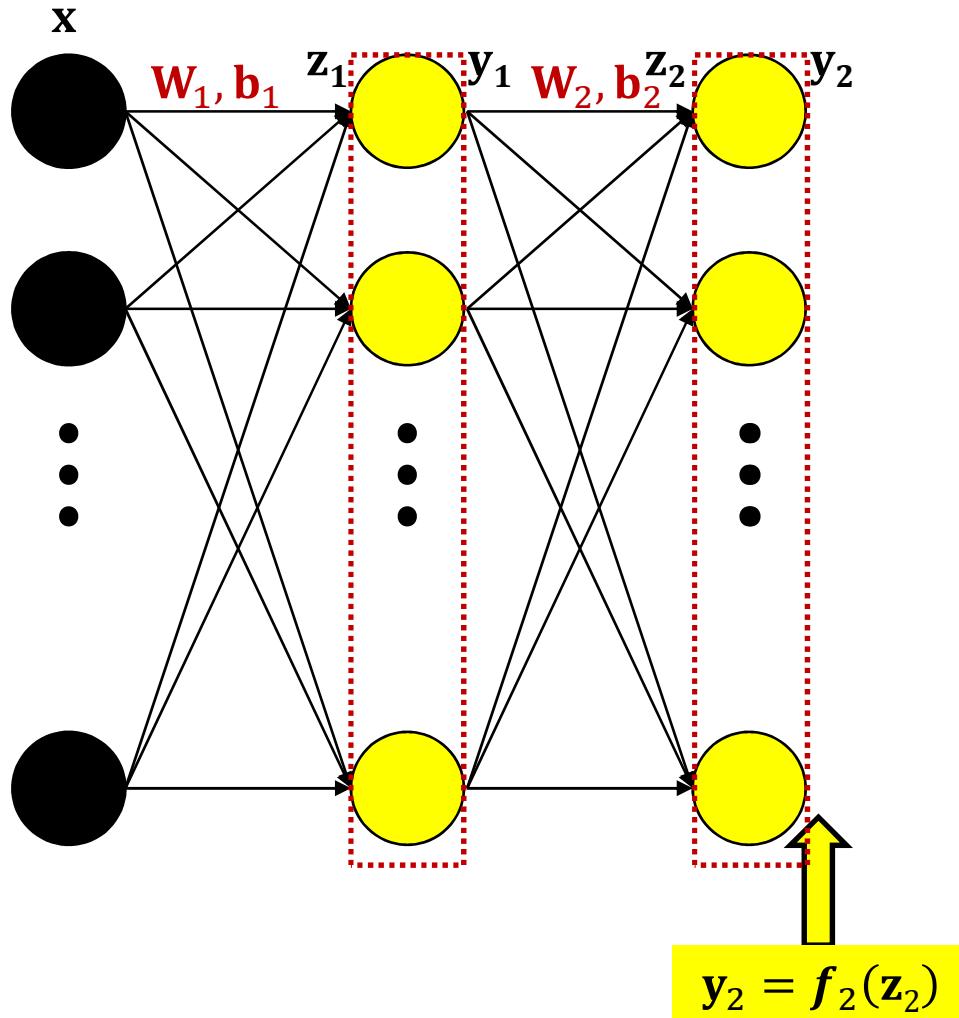
The forward pass



The Complete computation

$$y_1 = f_1(W_1x + b_1)$$

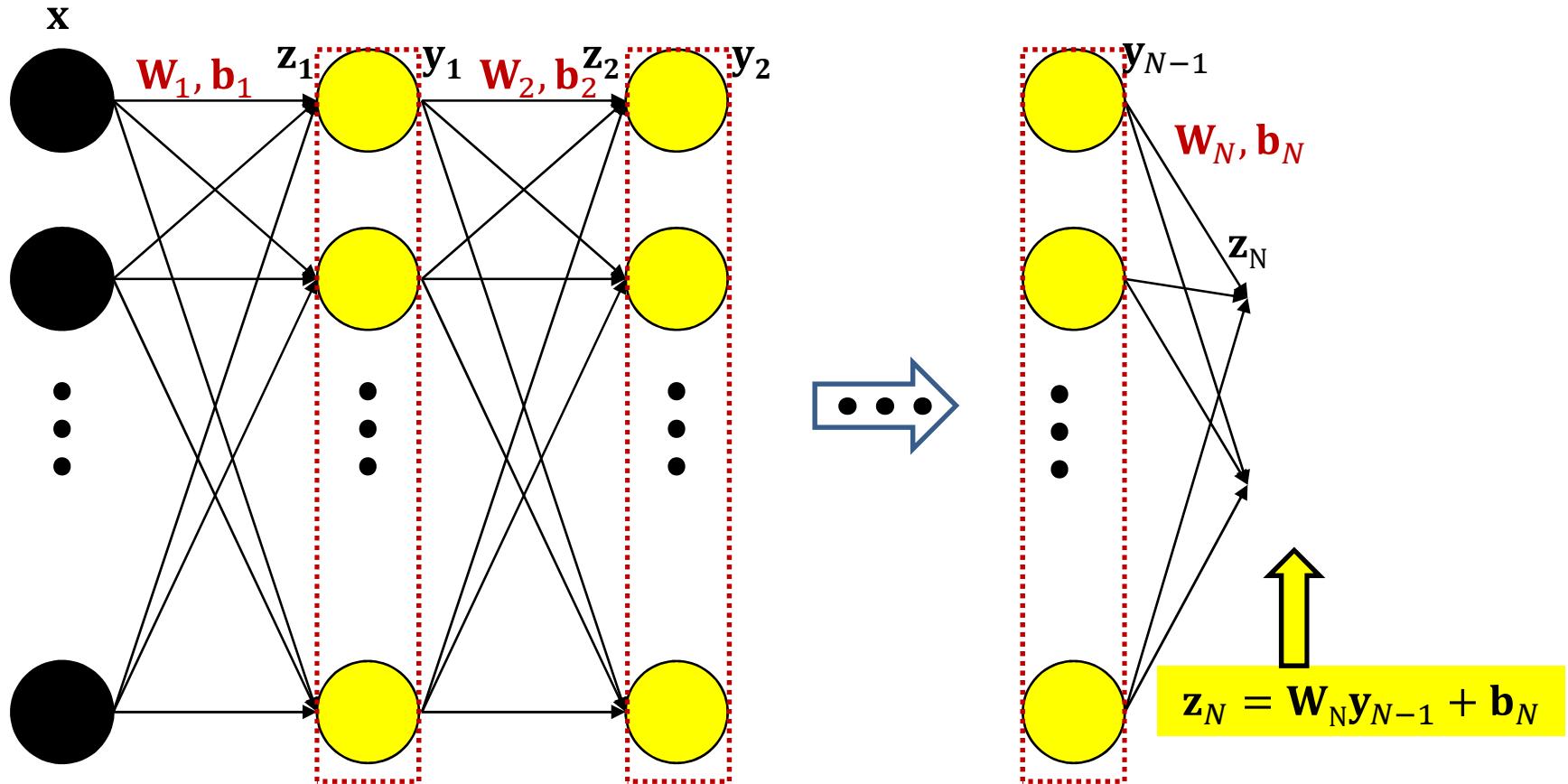
The forward pass



The Complete computation

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

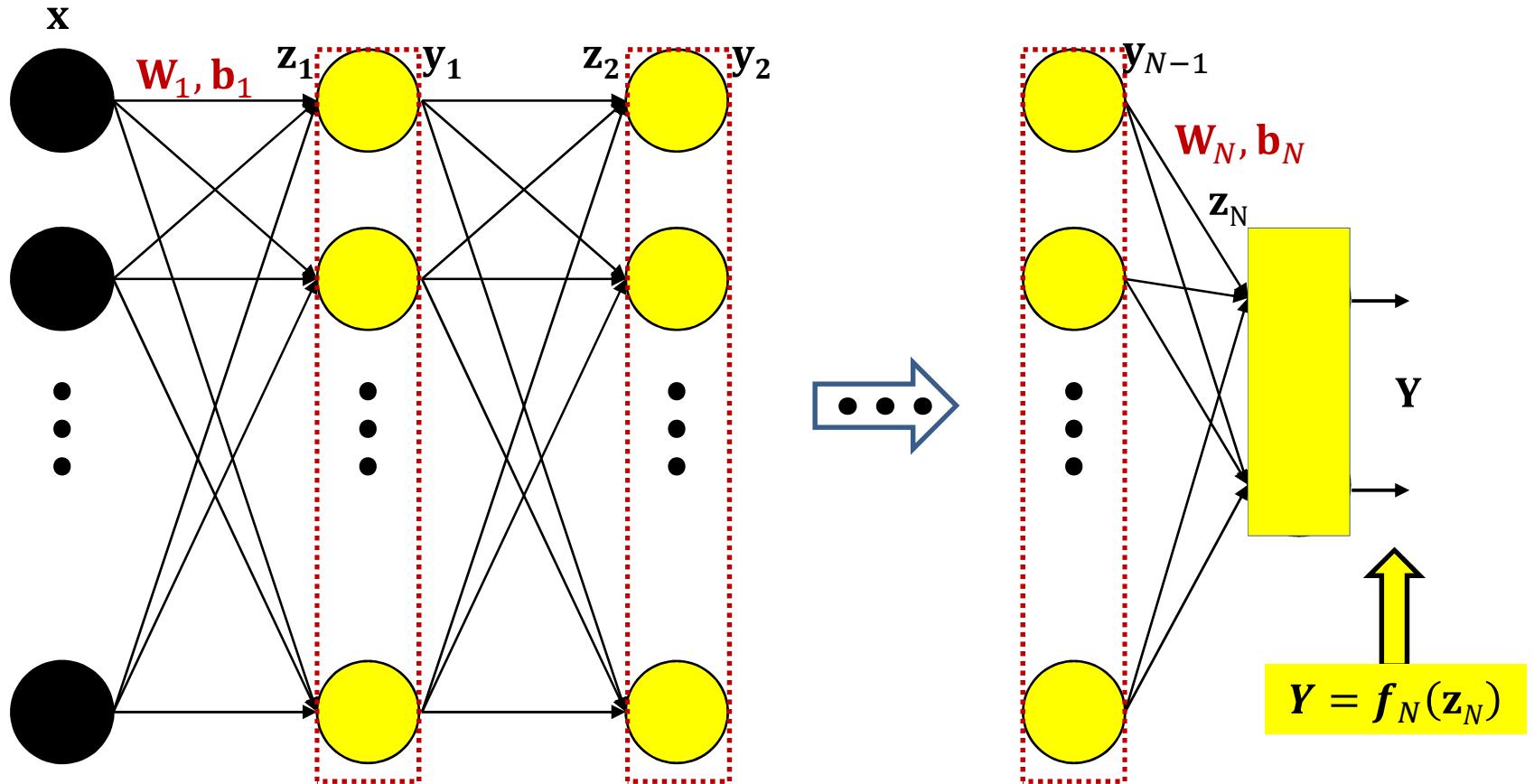
The forward pass



The Complete computation

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

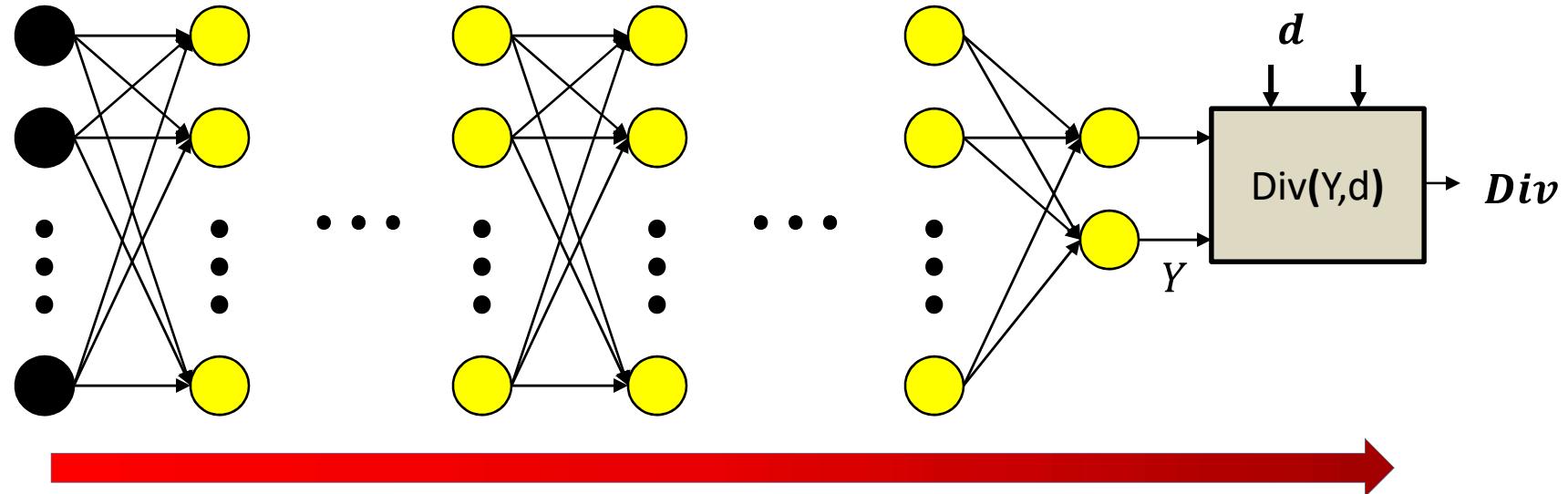
The forward pass



The Complete computation

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

Forward pass



Forward pass:

Initialize

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

For $k = 1$ to N :

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

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

Output

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

The Forward Pass

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

– Recursion:

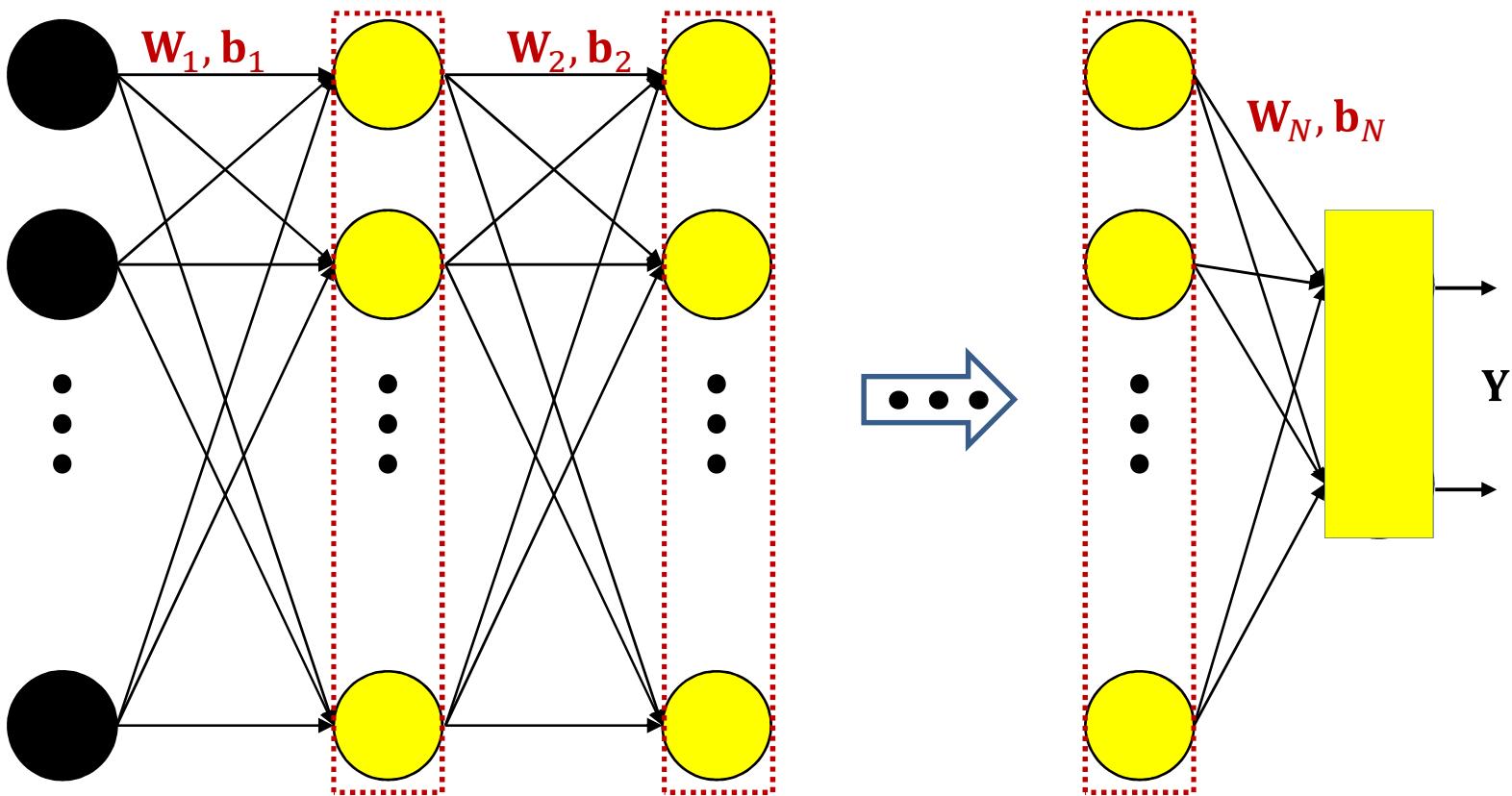
$$\mathbf{z}_k = \mathbf{W}_k \mathbf{y}_{k-1} + \mathbf{b}_k$$

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

- Output:

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

The backward pass



- The network is a nested function

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

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

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

Calculus recap 2: The Jacobian

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

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

Using vector notation

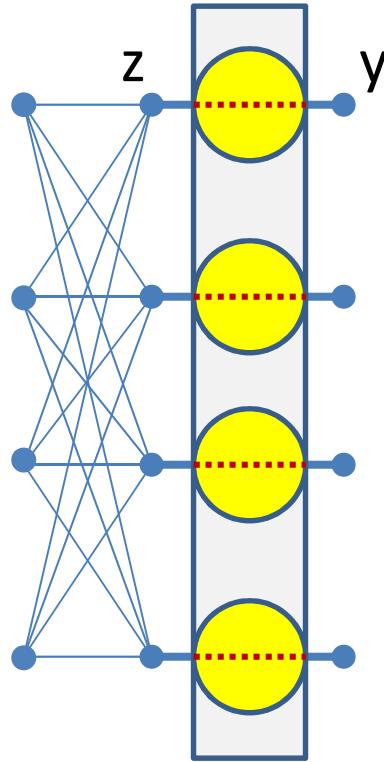
$$\mathbf{y} = f(\mathbf{z})$$

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

Check:

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

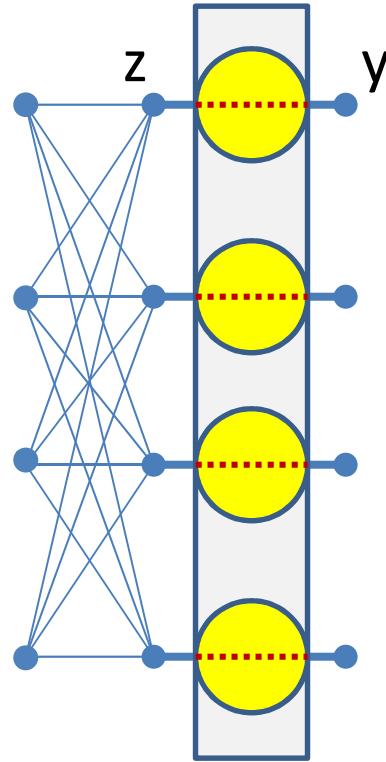
Jacobians can describe the derivatives of neural activations w.r.t their input



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

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

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

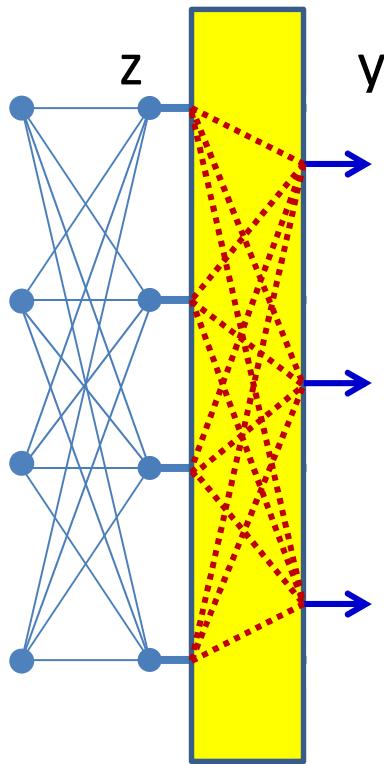


$$y_i = f(z_i)$$

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

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

For *Vector* activations

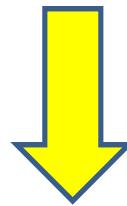


$$J_y(\mathbf{z}) = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \dots & \frac{\partial y_1}{\partial z_D} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \dots & \frac{\partial y_2}{\partial z_D} \\ \dots & \dots & \ddots & \dots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

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

Special case: Affine functions

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

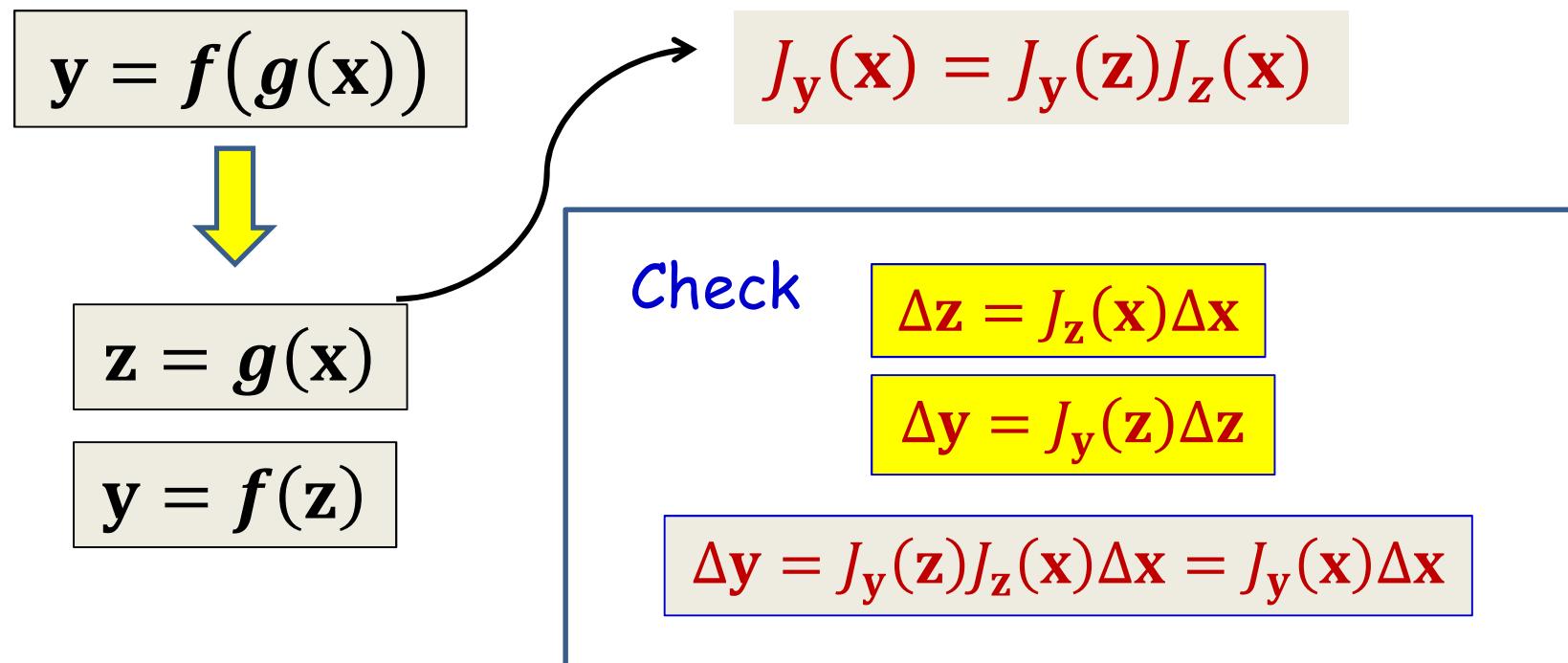


$$J_{\mathbf{z}}(\mathbf{y}) = \mathbf{W}$$

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

Vector derivatives: Chain rule

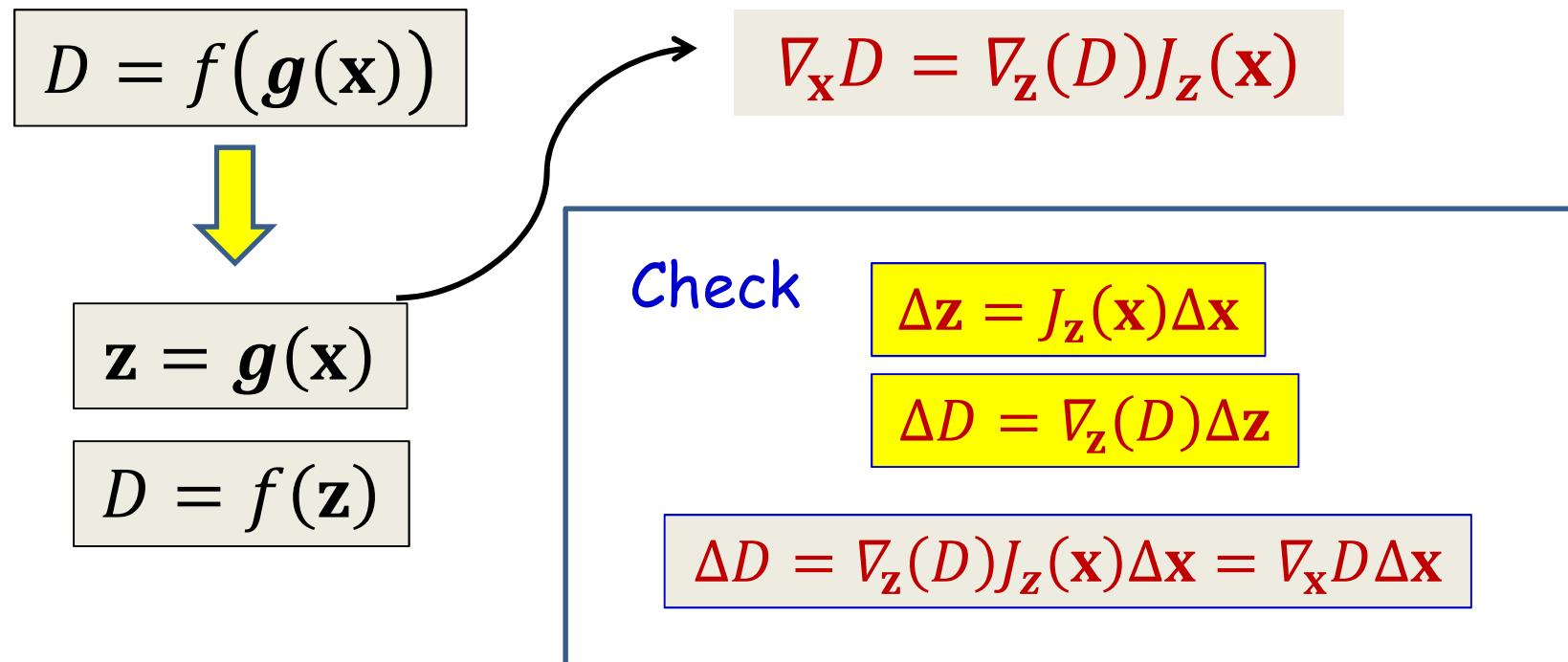
- We can define a chain rule for Jacobians
- **For vector functions of vector inputs:**



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

Vector derivatives: Chain rule

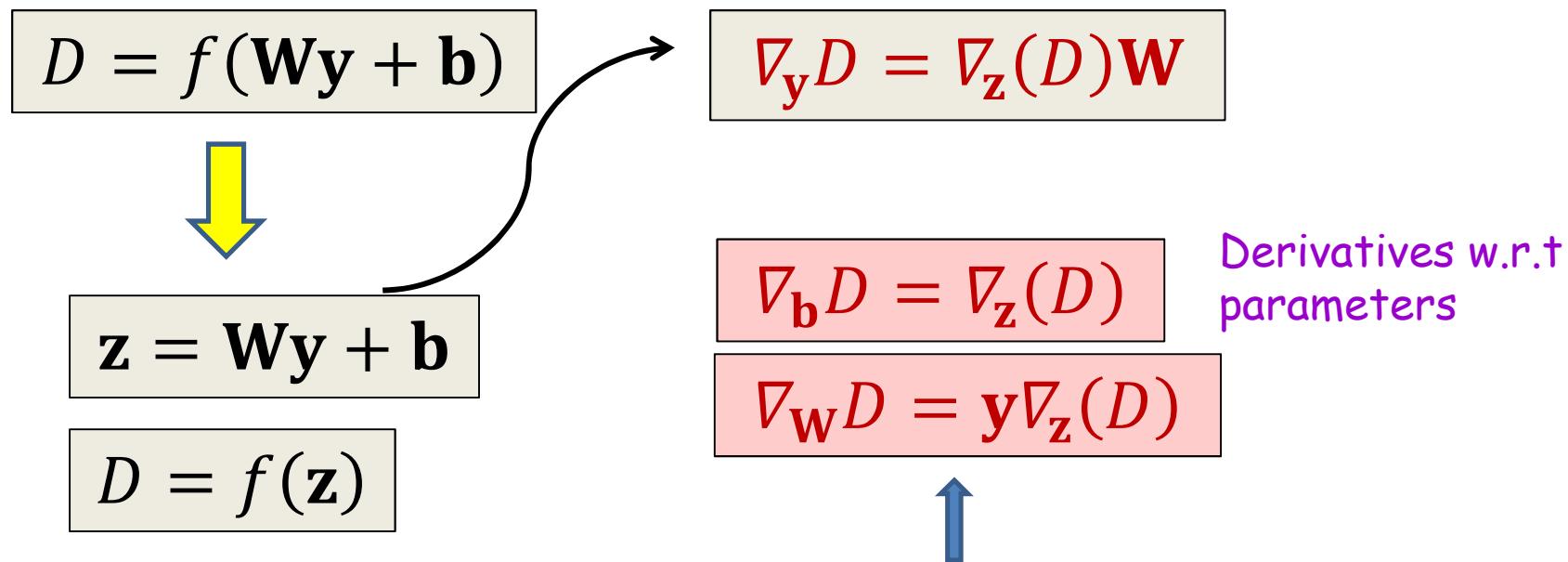
- *The chain rule can combine Jacobians and Gradients*
- **For scalar functions of vector inputs ($g()$ is vector):**



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

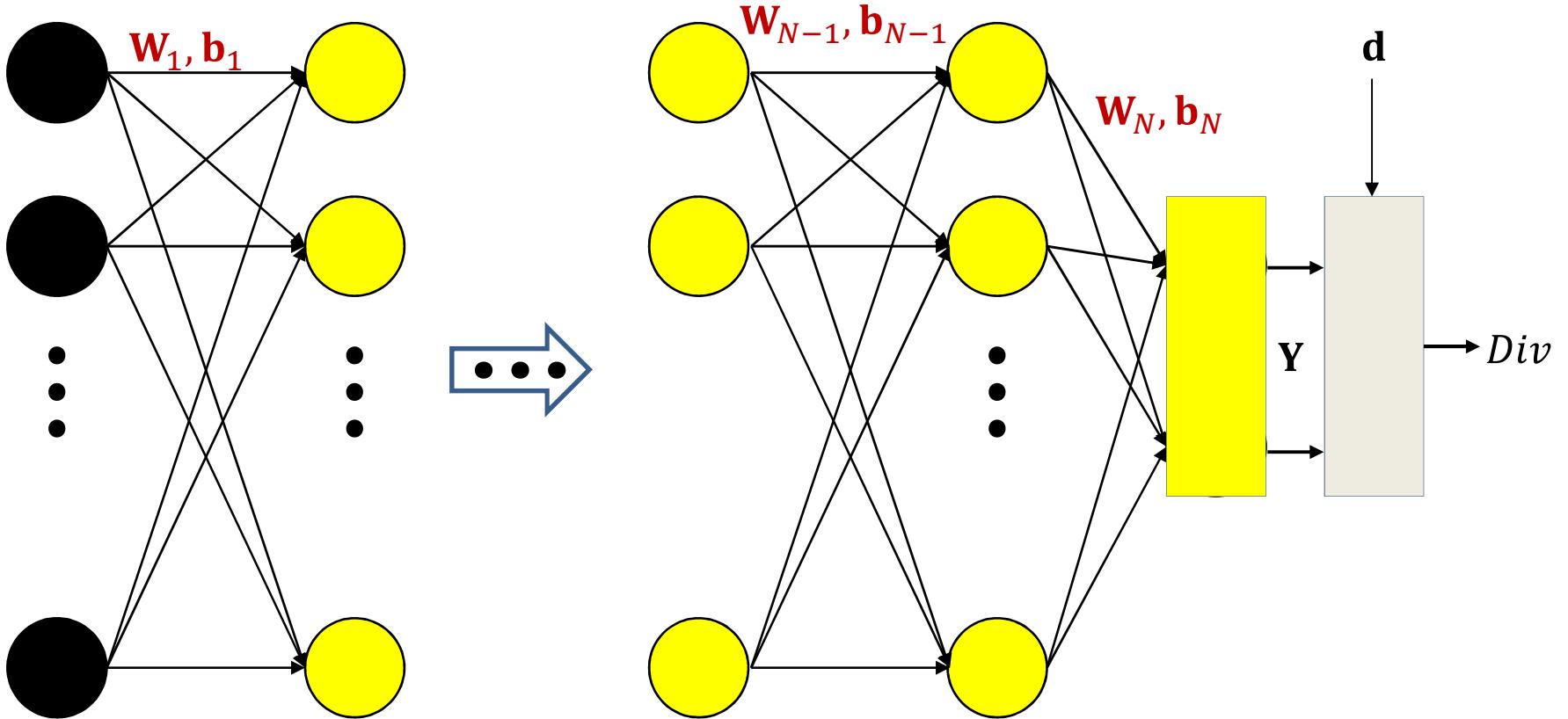
Special Case

- Scalar functions of Affine functions



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

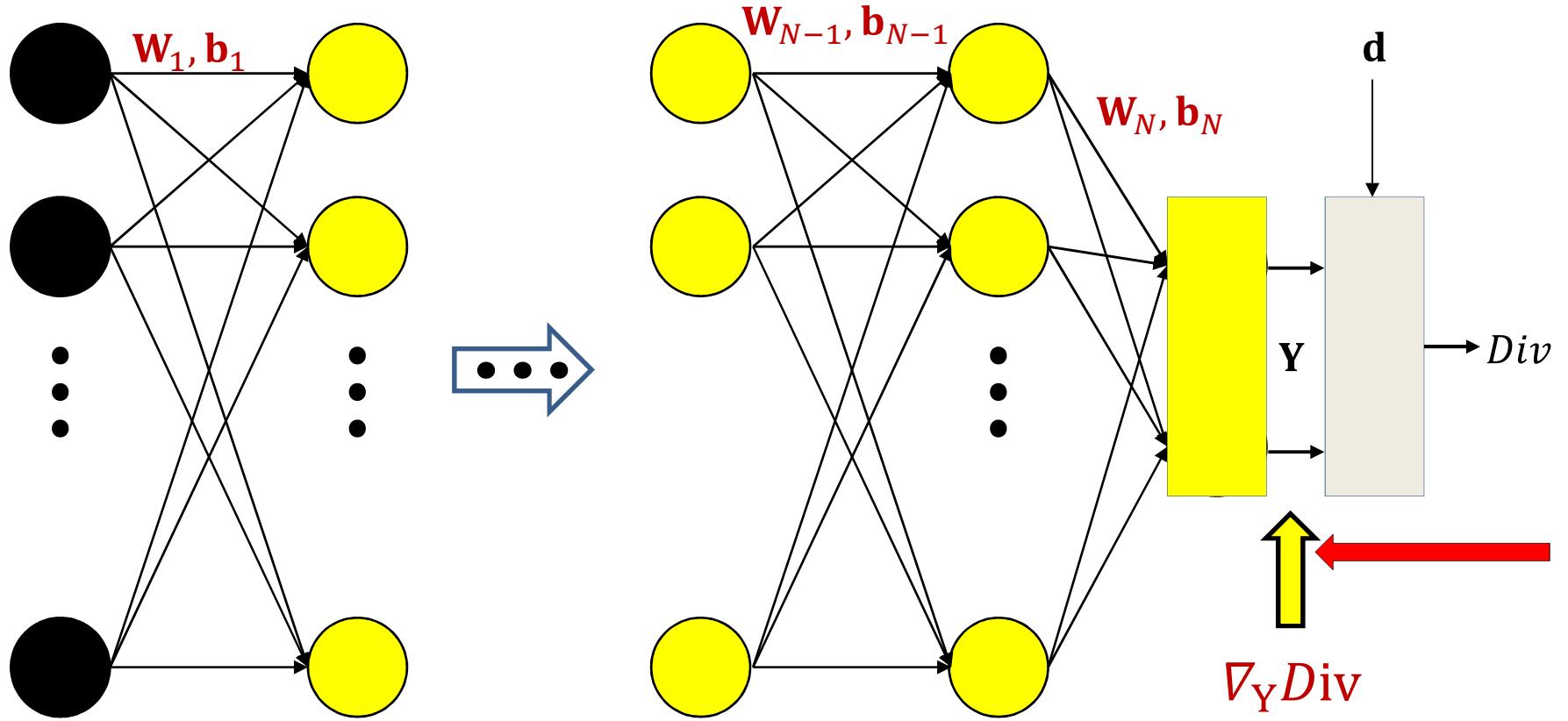
The backward pass



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

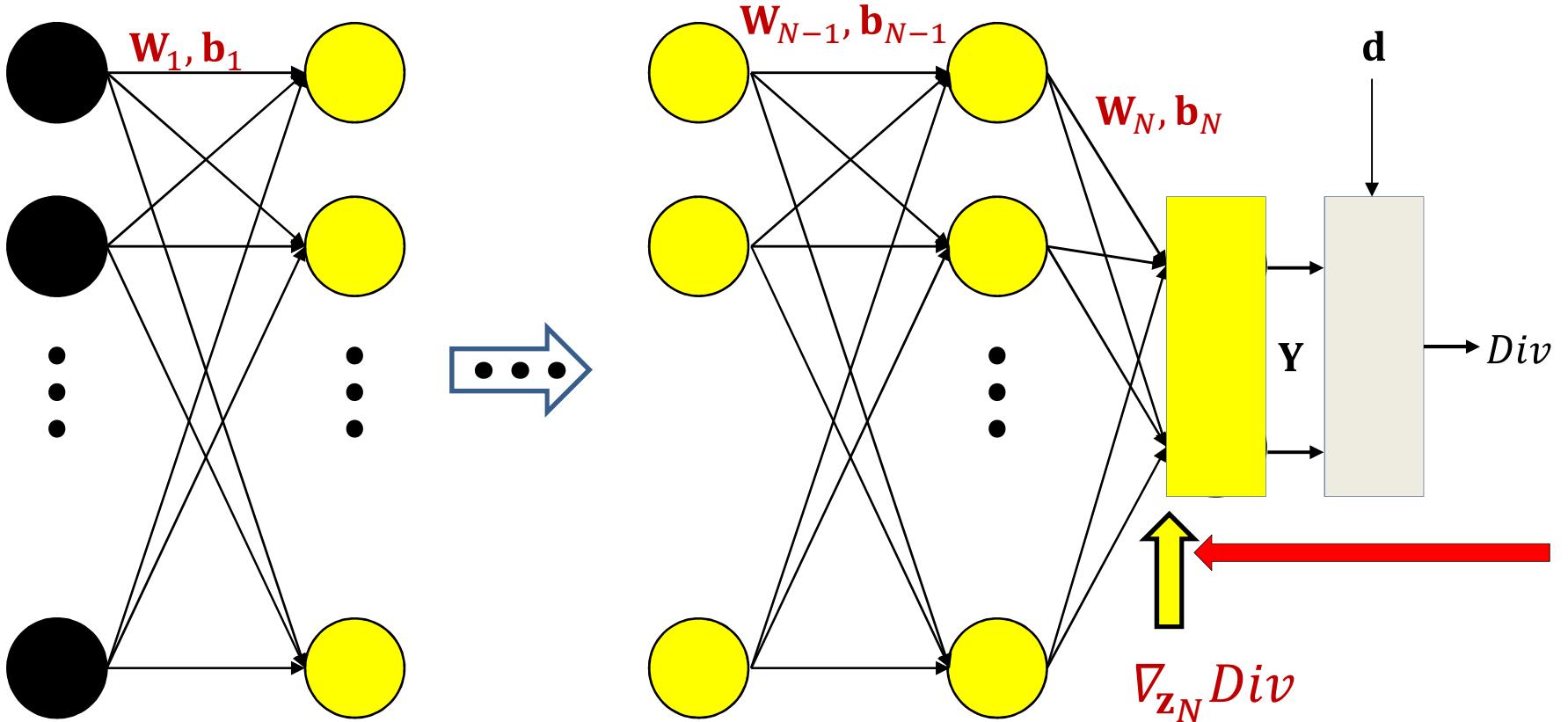
In general $\nabla_{\mathbf{a}} \mathbf{b}$ represents a derivative of \mathbf{b} w.r.t. \mathbf{a} and could be a gradient (for scalar \mathbf{b}) Or a Jacobian (for vector \mathbf{b})

The backward pass



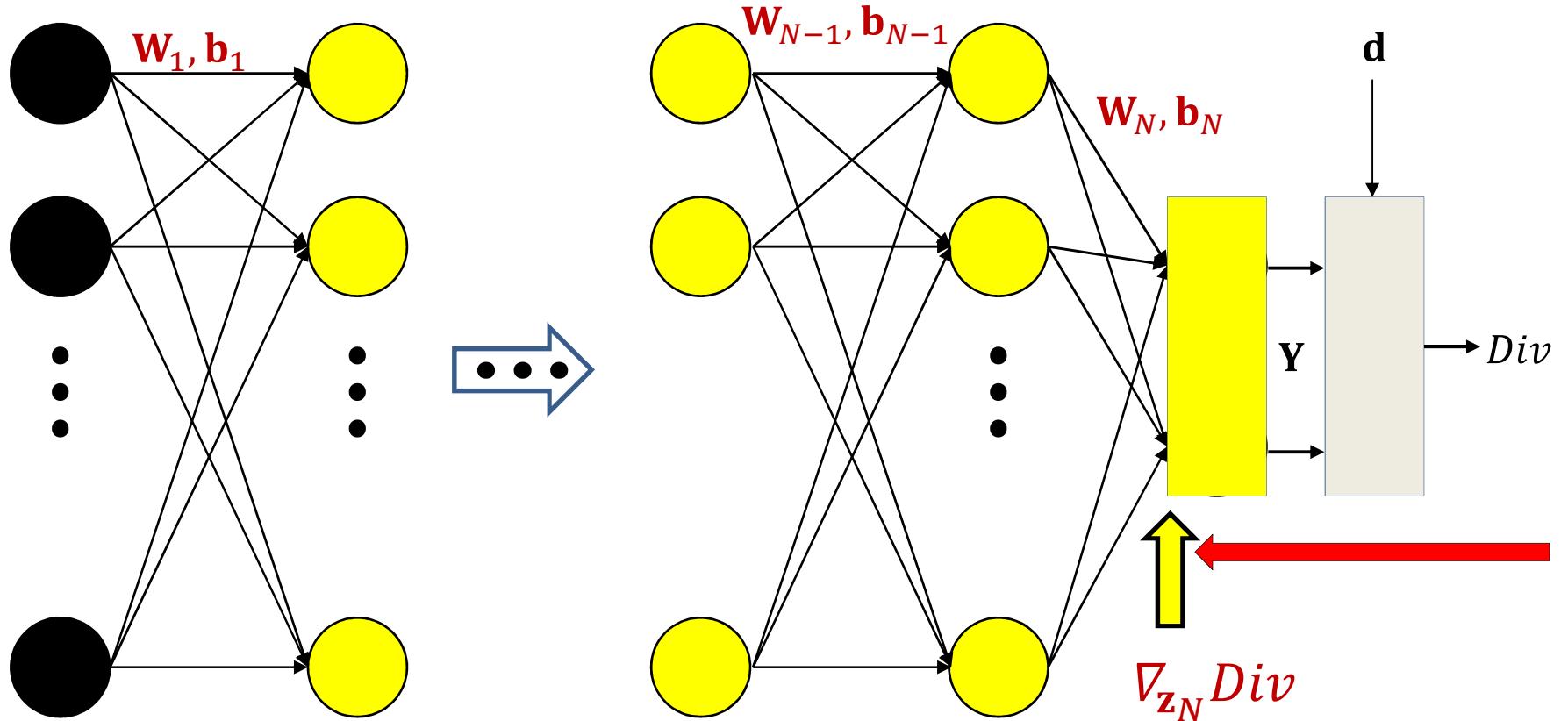
First compute the gradient of the divergence w.r.t. Y .
The actual gradient depends on the divergence function.

The backward pass



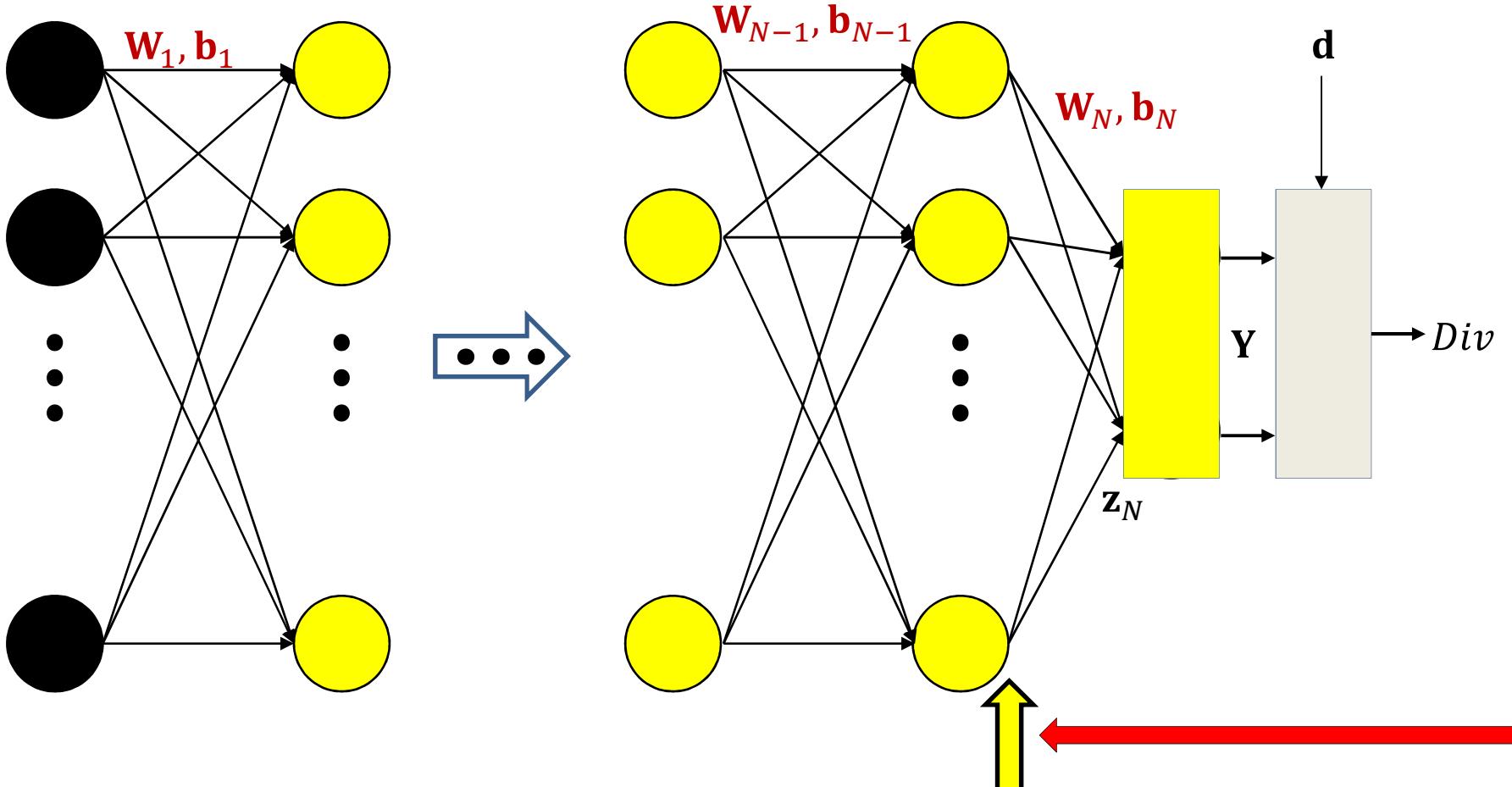
$$\nabla_{\mathbf{z}_N} \text{Div} = \nabla_{\mathbf{Y}} \text{Div} \cdot \nabla_{\mathbf{z}_N} \mathbf{Y}$$

The backward pass



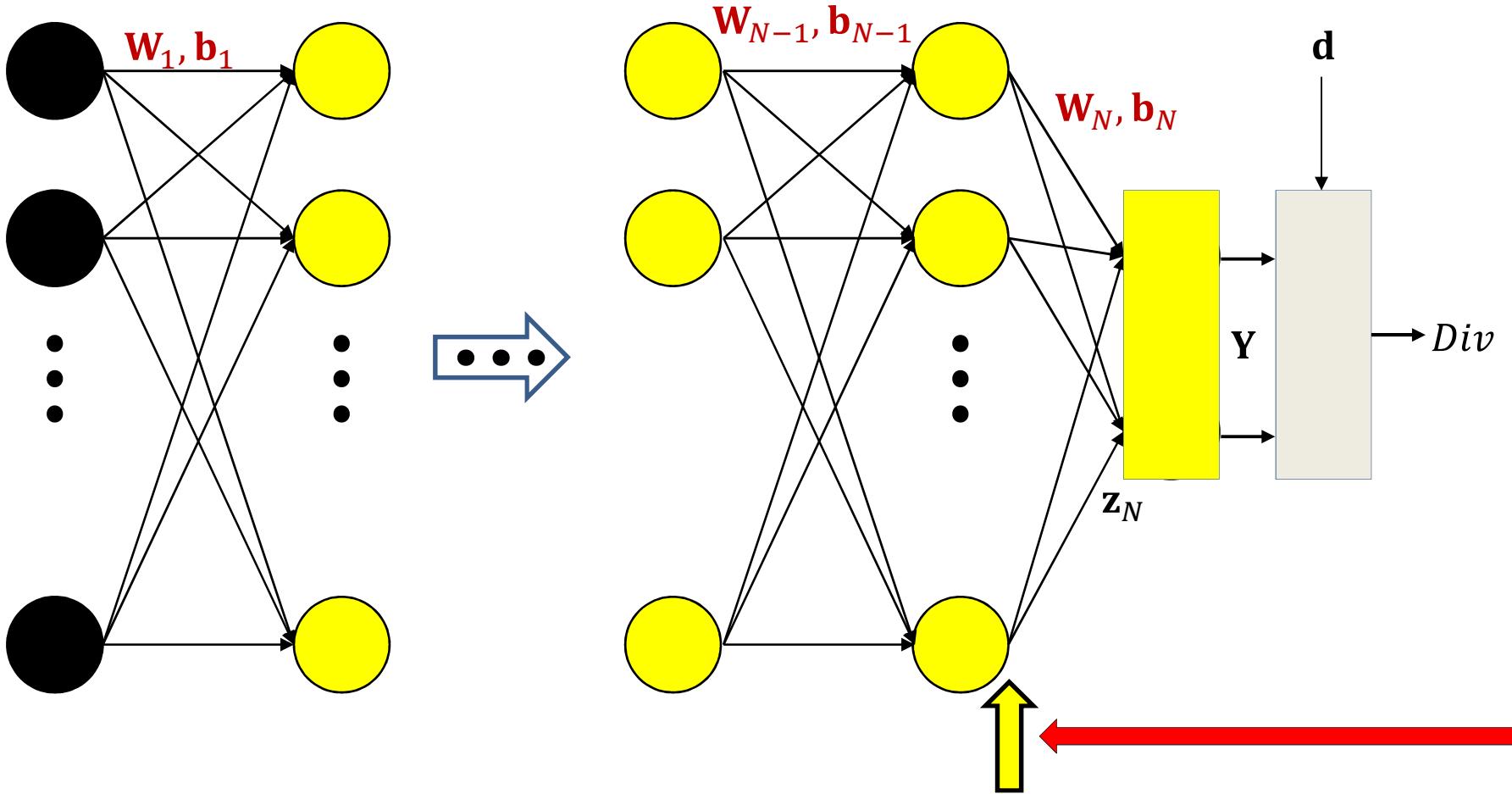
$$\nabla_{\mathbf{z}_N} \text{Div} = \nabla_{\mathbf{Y}} \text{Div} J_{\mathbf{Y}}(\mathbf{z}_N)$$

The backward pass



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

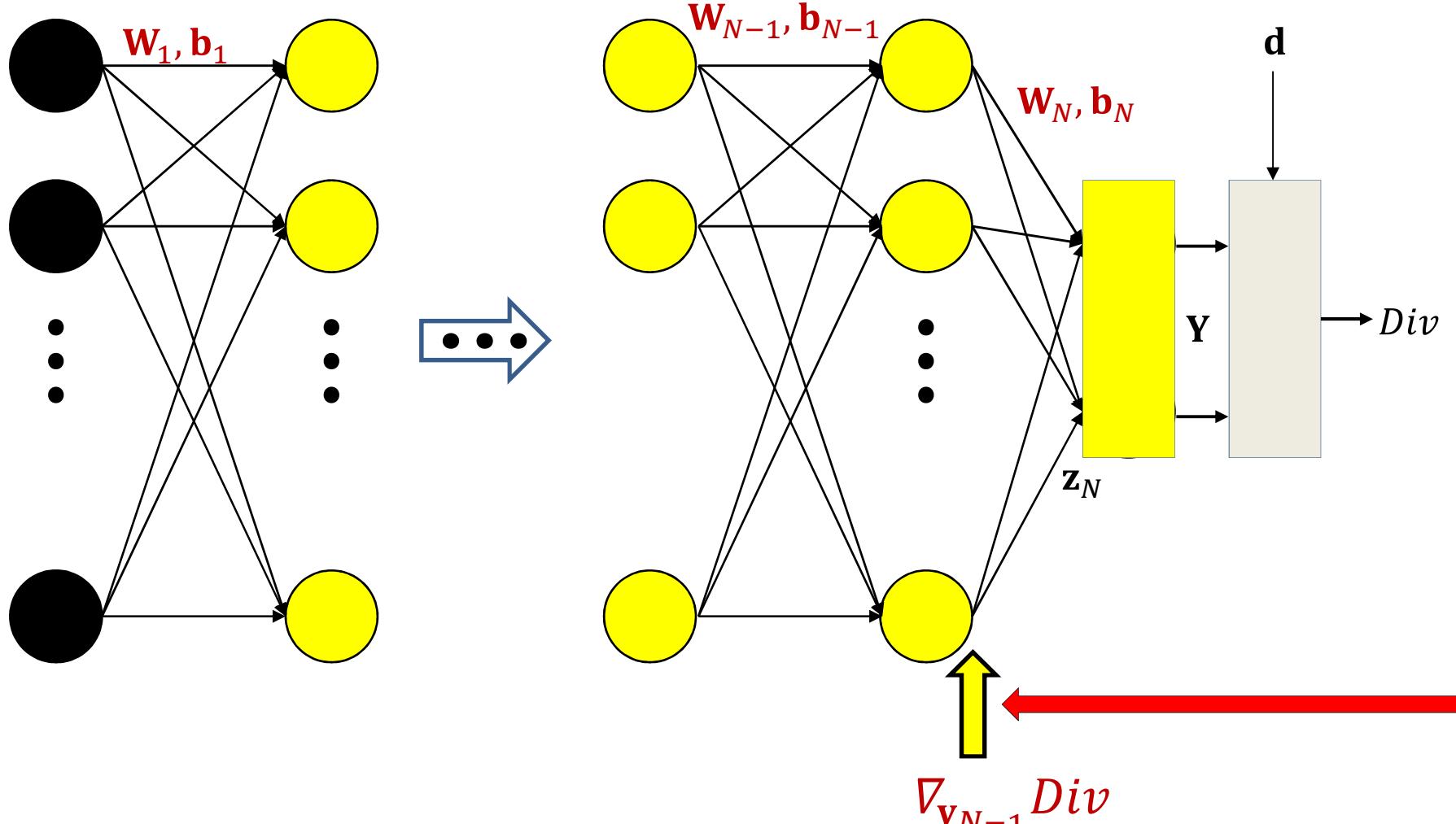
The backward pass



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

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

The backward pass

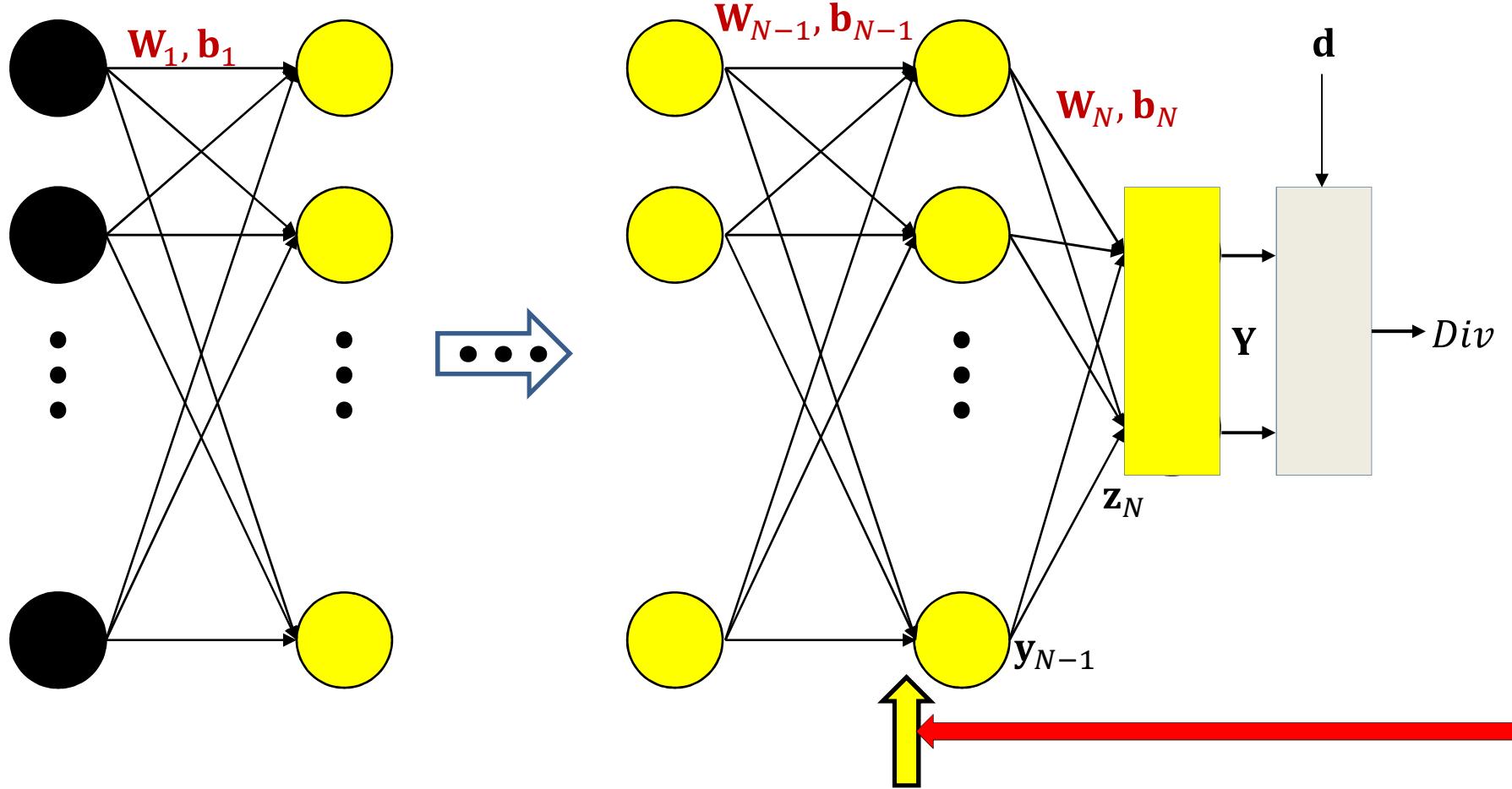


$$\nabla_{y_{N-1}} Div = \nabla_{z_N} Div \mathbf{W}_N$$

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

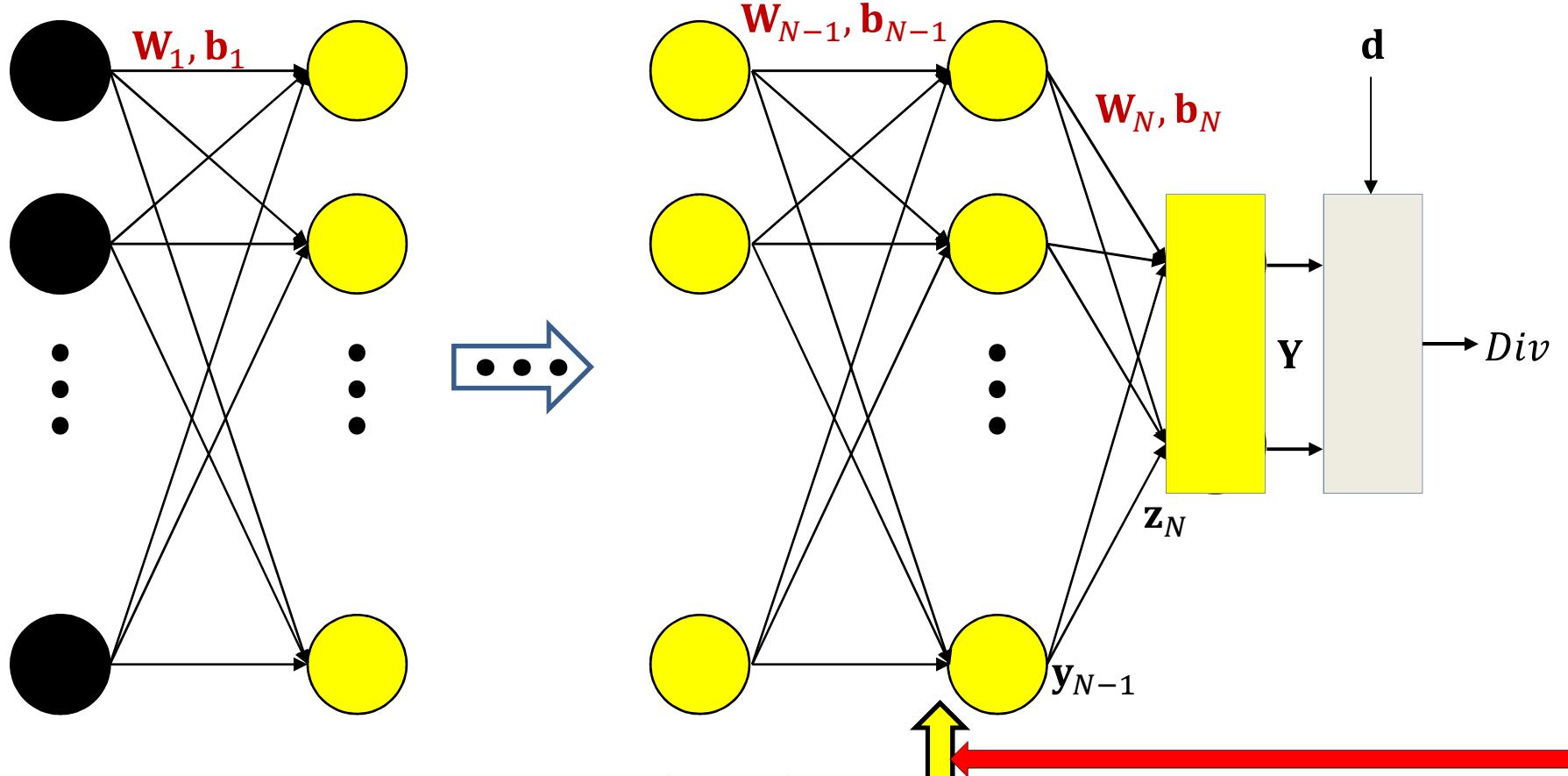
$$\nabla_{\mathbf{b}_N} Div = \nabla_{z_N} Div$$

The backward pass



$$\nabla_{z_{N-1}} Div = \nabla_{y_{N-1}} Div \cdot \nabla_{z_{N-1}} y_{N-1}$$

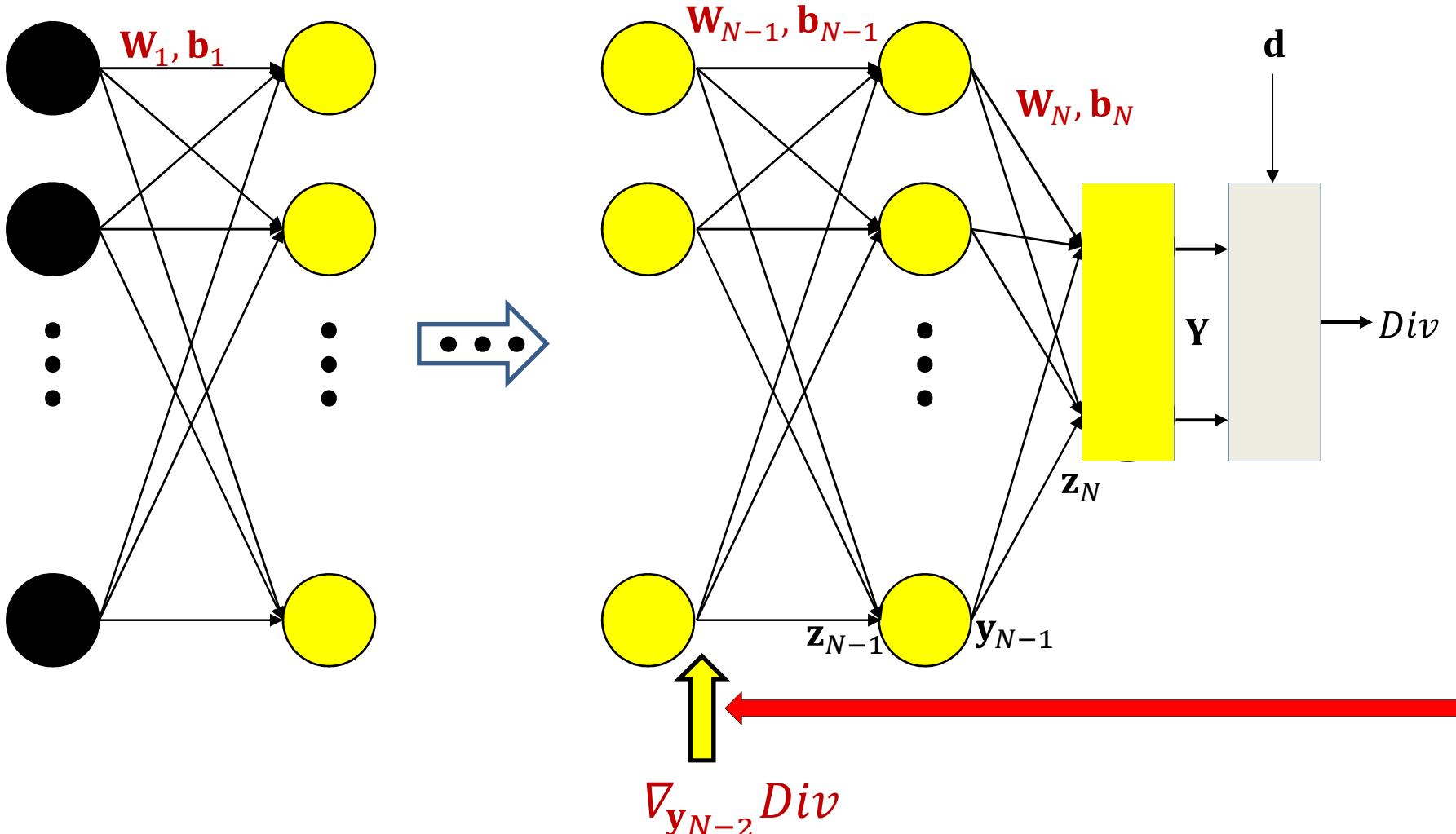
The backward pass



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

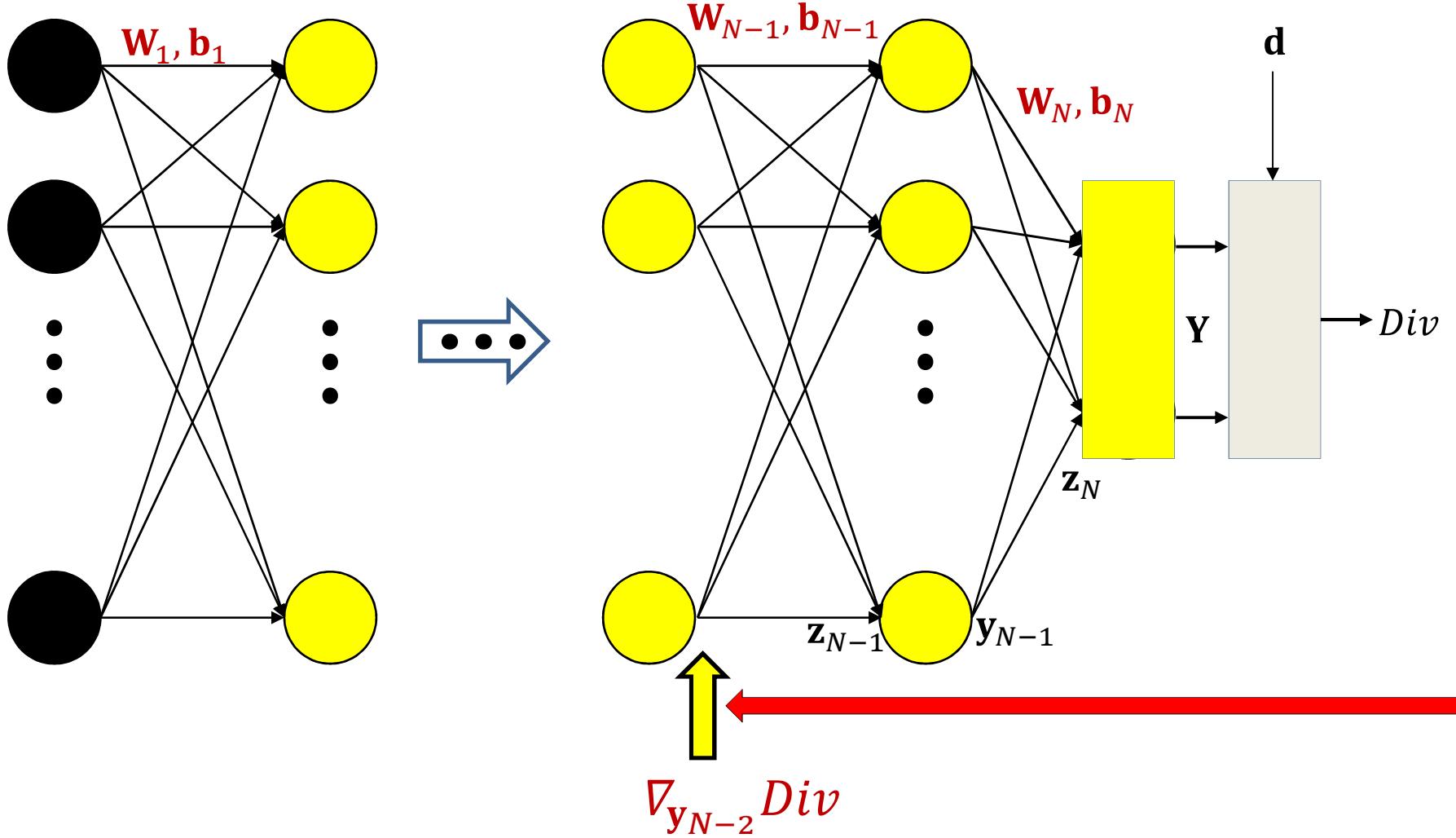
The Jacobian will be a diagonal matrix for scalar activations

The backward pass



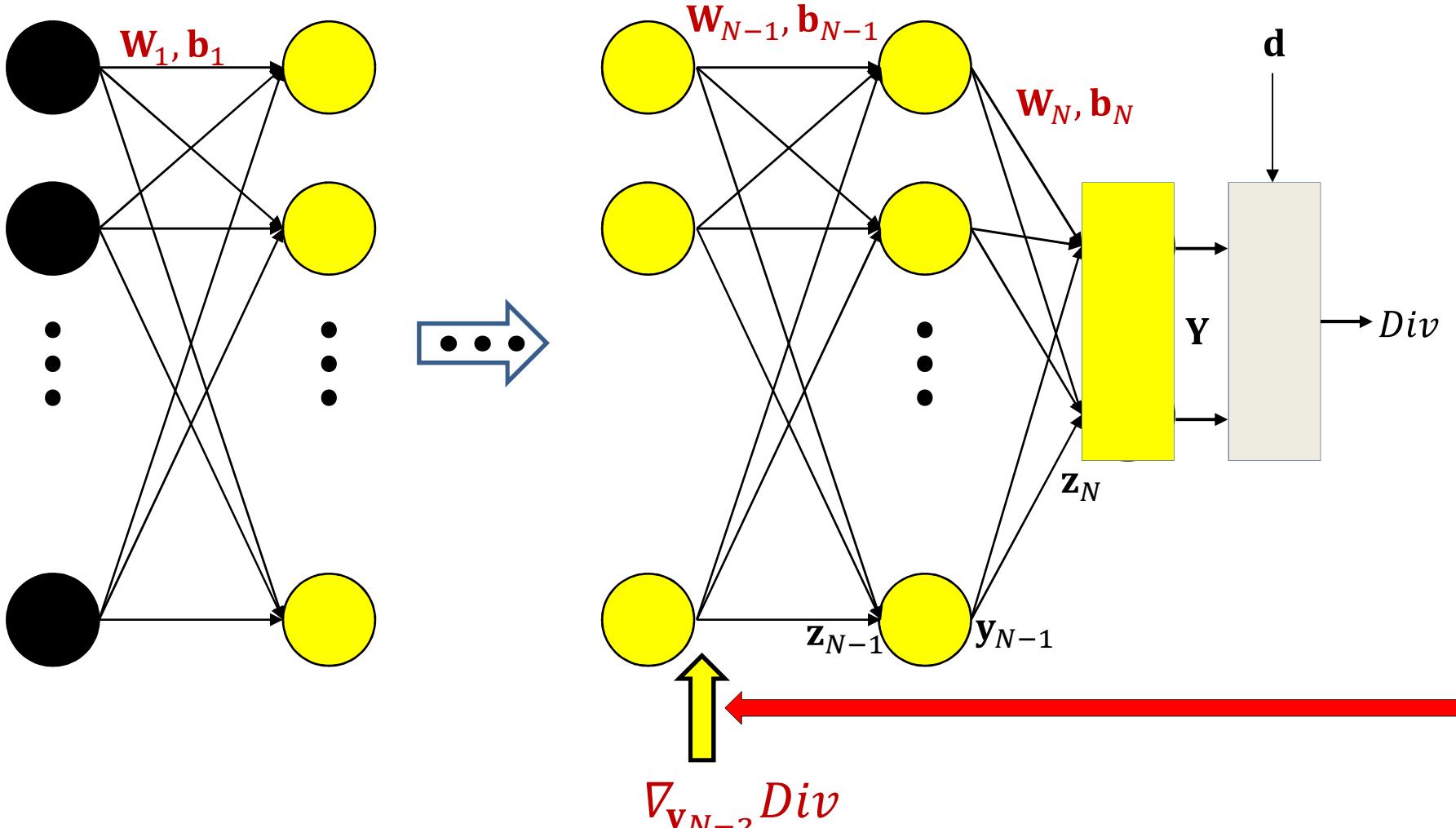
$$\nabla_{\mathbf{y}_{N-2}} Div = \nabla_{\mathbf{z}_{N-1}} Div \cdot \nabla_{\mathbf{y}_{N-2}} \mathbf{z}_{N-1}$$

The backward pass



$$\nabla_{y_{N-2}} Div = \nabla_{z_{N-1}} Div \mathbf{W}_{N-1}$$

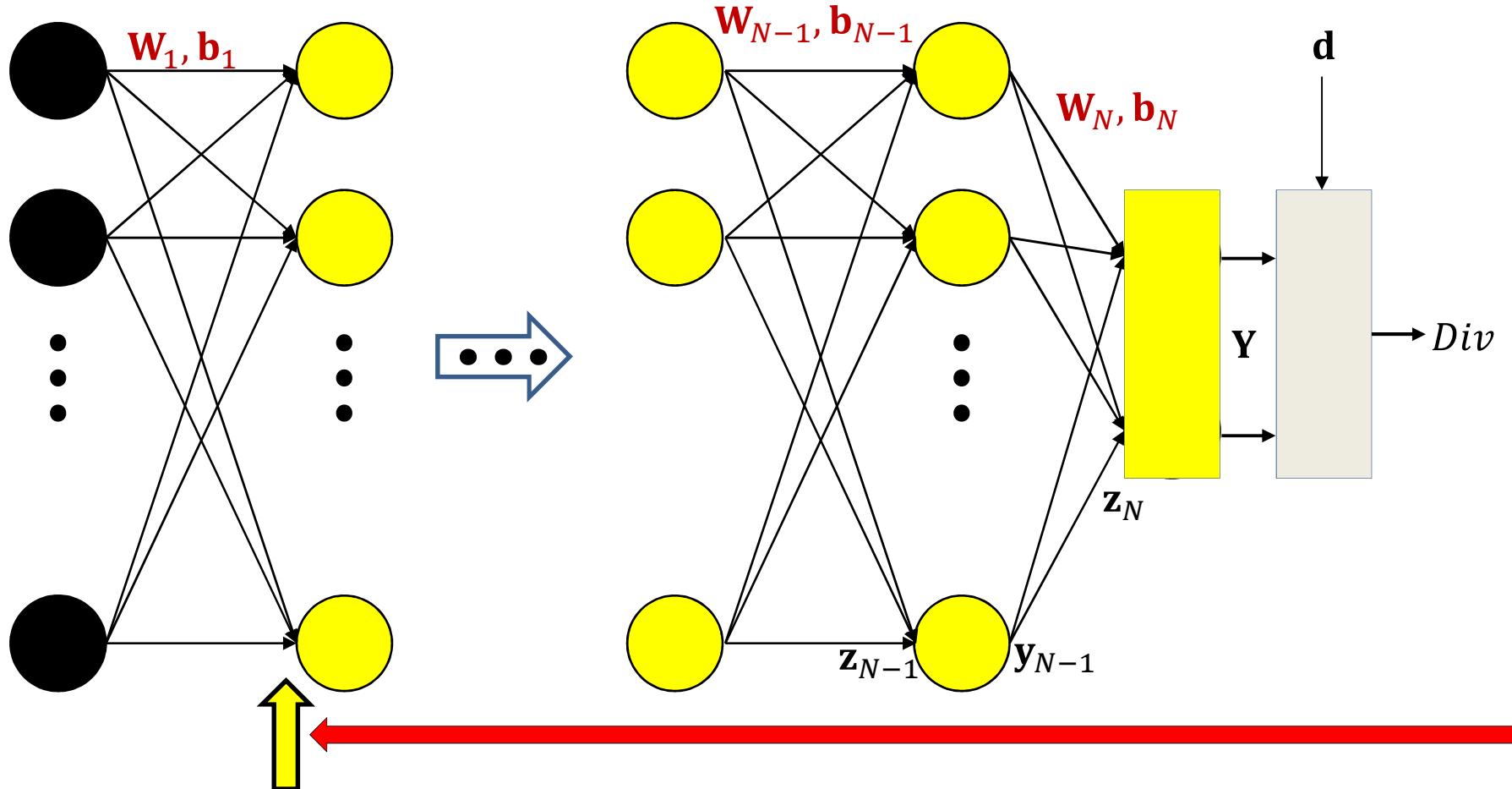
The backward pass



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

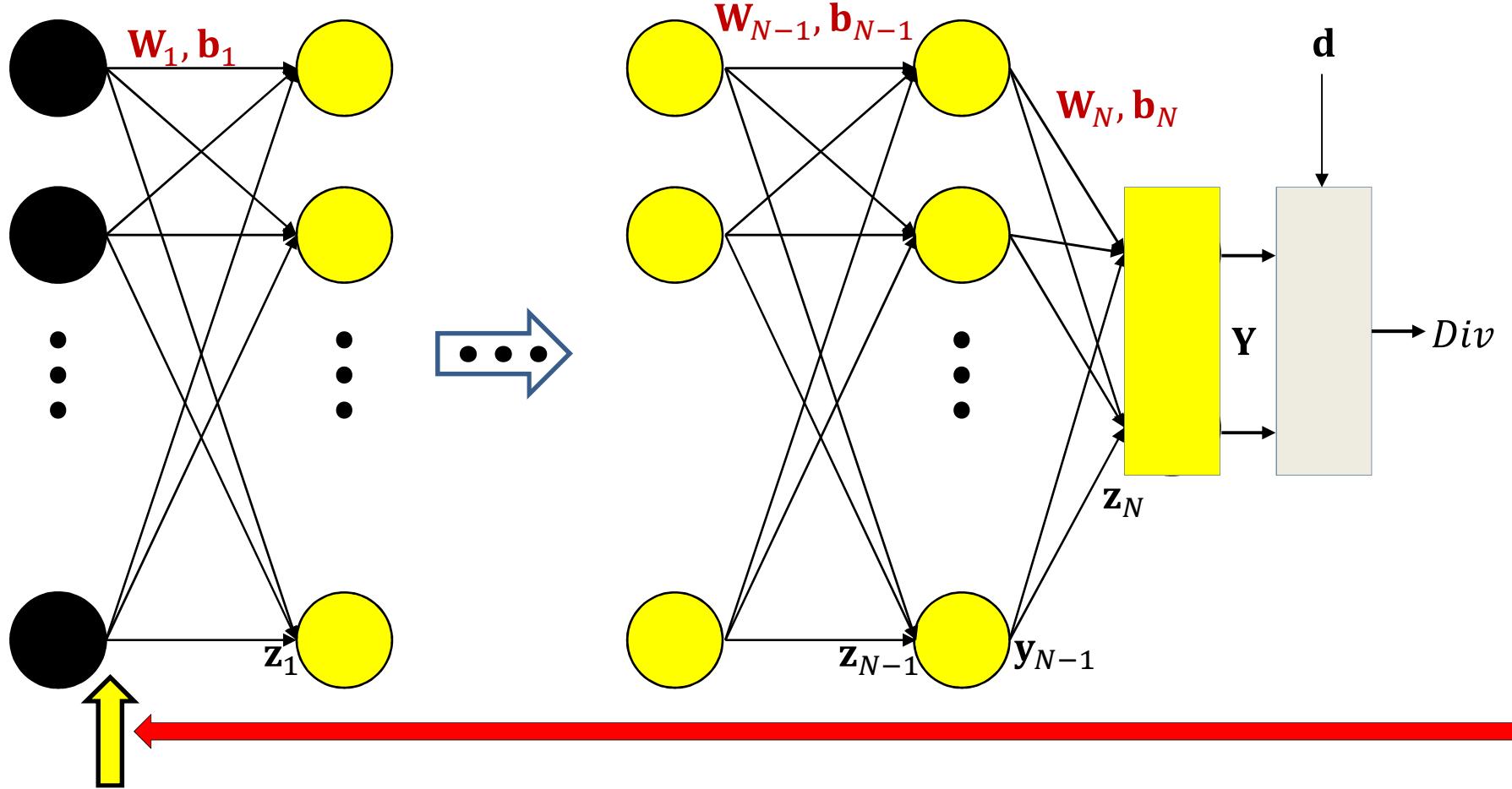
$\nabla_{W_{N-1}} Div = y_{N-2} \nabla_{z_{N-1}} Div$
$\nabla_{b_{N-1}} Div = \nabla_{z_{N-1}} Div$

The backward pass



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

The backward pass



$$\nabla_{W_1} Div = \mathbf{x} \nabla_{z_1} Div$$

$$\nabla_{b_1} Div = \nabla_{z_1} Div$$

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

The Backward Pass

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

The Backward Pass

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

Note analogy to forward pass

$$\nabla_{\mathbf{z}_k} Div = \nabla_{\mathbf{y}_k} Div J_{\mathbf{y}_k}(\mathbf{z}_k)$$
$$\nabla_{\mathbf{y}_{k-1}} Div = \nabla_{\mathbf{z}_k} Div \mathbf{W}_k$$
 - Gradient computation:
$$\nabla_{\mathbf{W}_k} Div = \mathbf{y}_{k-1} \nabla_{\mathbf{z}_k} Div$$
$$\nabla_{\mathbf{b}_k} Div = \nabla_{\mathbf{z}_k} Div$$

For comparison: The Forward Pass

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

– Recursion:

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

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

- Output:

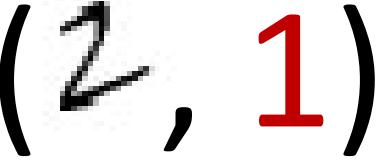
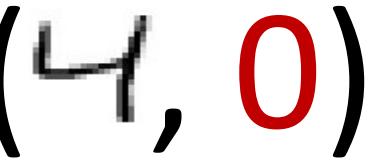
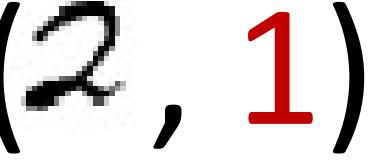
$$\mathbf{Y} = \mathbf{y}_N$$

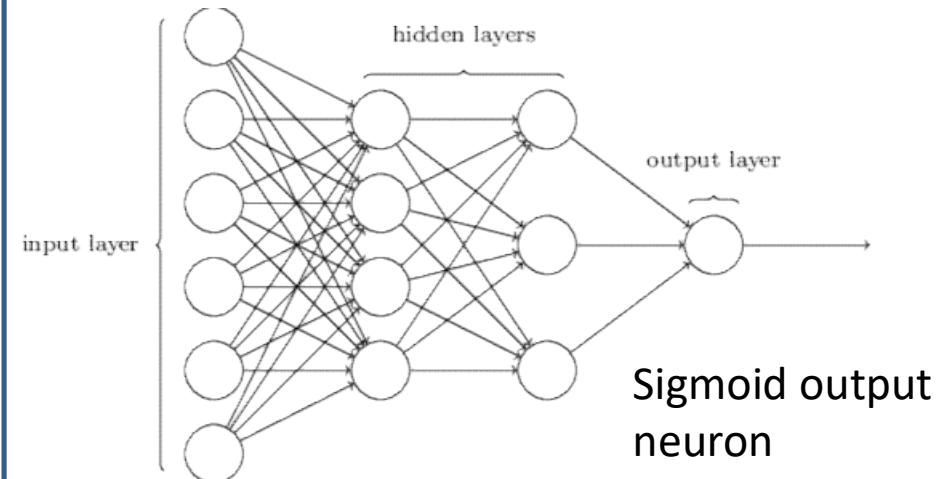
Neural network training algorithm

- Initialize all weights and biases ($\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \dots, \mathbf{W}_N, \mathbf{b}_N$)
- Do:
 - $Err = 0$
 - For all k , initialize $\nabla_{\mathbf{W}_k} Err = 0, \nabla_{\mathbf{b}_k} Err = 0$
 - For all $t = 1:T$
 - Forward pass : Compute
 - Output $\mathbf{Y}(X_t)$
 - Divergence $\text{Div}(\mathbf{Y}_t, \mathbf{d}_t)$
 - $Err += \text{Div}(\mathbf{Y}_t, \mathbf{d}_t)$
 - Backward pass: For all k compute:
 - $\nabla_{\mathbf{y}_k} \text{Div} = \nabla_{\mathbf{z}_{k+1}} \text{Div} \mathbf{W}_k$
 - $\nabla_{\mathbf{z}_k} \text{Div} = \nabla_{\mathbf{y}_k} \text{Div} J_{\mathbf{y}_k}(\mathbf{z}_k)$
 - $\nabla_{\mathbf{W}_k} \text{Div}(\mathbf{Y}_t, \mathbf{d}_t); \nabla_{\mathbf{b}_k} \text{Div}(\mathbf{Y}_t, \mathbf{d}_t)$
 - $\nabla_{\mathbf{W}_k} Err += \nabla_{\mathbf{W}_k} \text{Div}(\mathbf{Y}_t, \mathbf{d}_t); \nabla_{\mathbf{b}_k} Err += \nabla_{\mathbf{b}_k} \text{Div}(\mathbf{Y}_t, \mathbf{d}_t)$
 - For all k , update:
$$\mathbf{W}_k = \mathbf{W}_k - \frac{\eta}{T} (\nabla_{\mathbf{W}_k} Err)^T; \quad \mathbf{b}_k = \mathbf{b}_k - \frac{\eta}{T} (\nabla_{\mathbf{b}_k} Err)^T$$
- Until Err has converged

Setting up for digit recognition

Training data

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

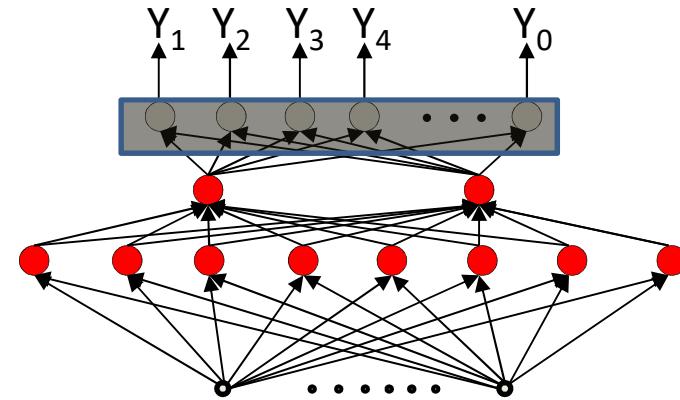


- Simple Problem: Recognizing “2” or “not 2”
- Single output with sigmoid activation
 - $Y \in (0,1)$
 - d is either 0 or 1
- Use KL divergence
- Backpropagation to learn network parameters

Recognizing the digit

Training data

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



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

Issues

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

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

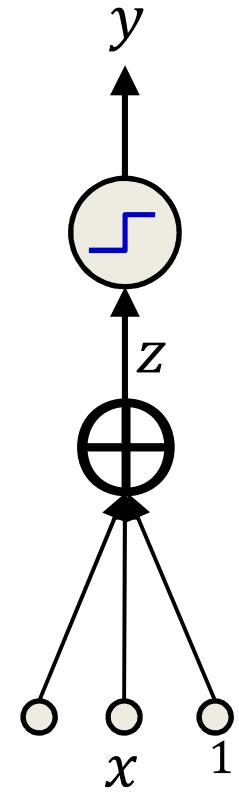
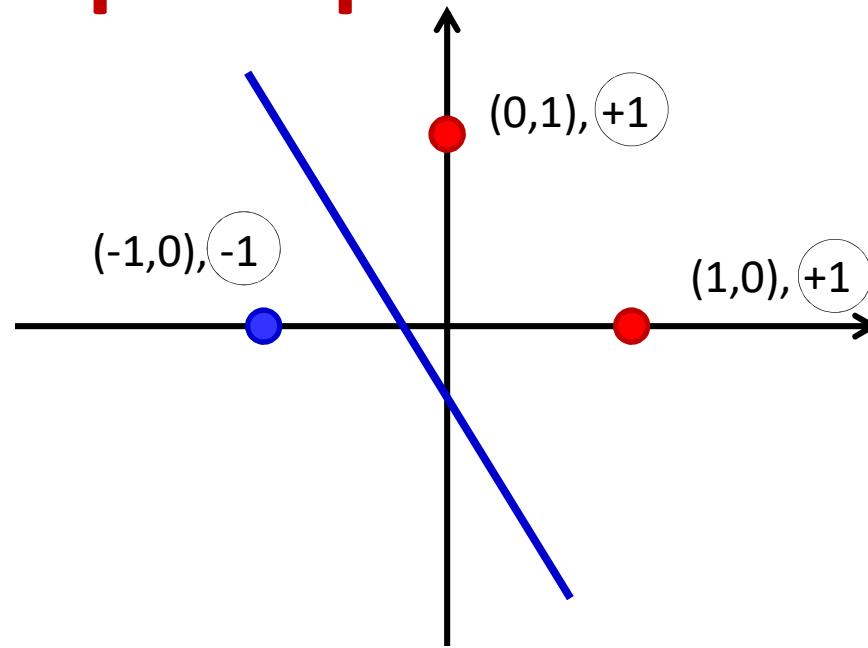
Does backprop do the right thing?

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

Does backprop do the right thing?

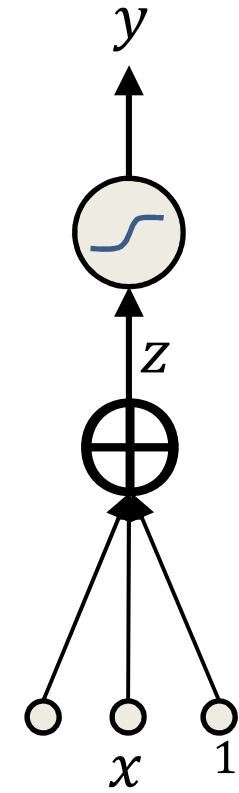
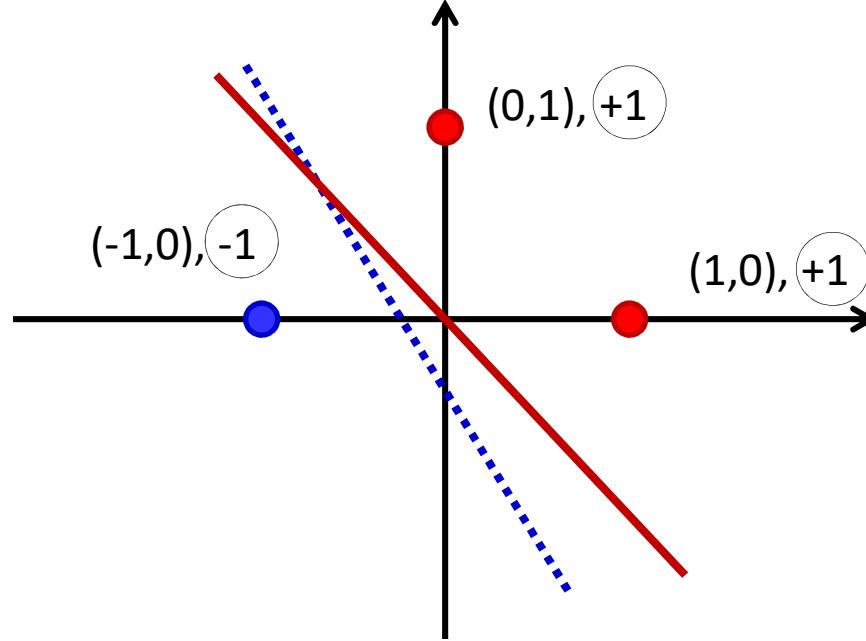
- **Is backprop always right?**
 - Assuming it actually find the global minimum of the divergence function?
- In classification problems, the classification error is a non-differentiable function of weights
- The divergence function minimized is only a *proxy* for classification error
- Minimizing divergence may not minimize classification error

Backprop fails to separate where perceptron succeeds



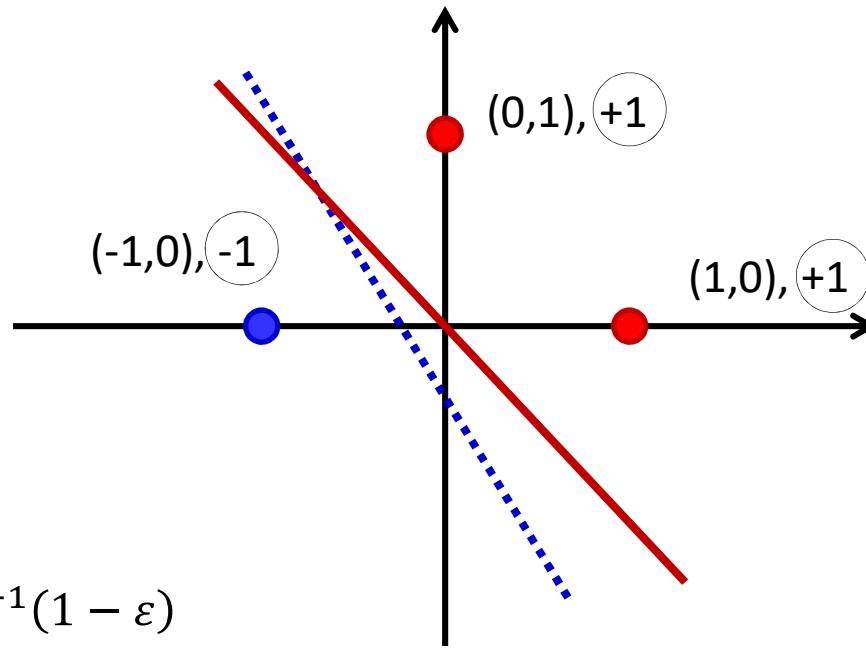
- Brady, Raghavan, Slawny, '89
- Simple problem, 3 training instances, single neuron
- Perceptron training rule trivially find a perfect solution

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

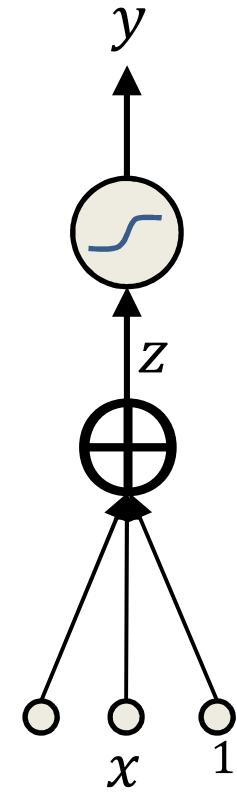
Unique solution exists



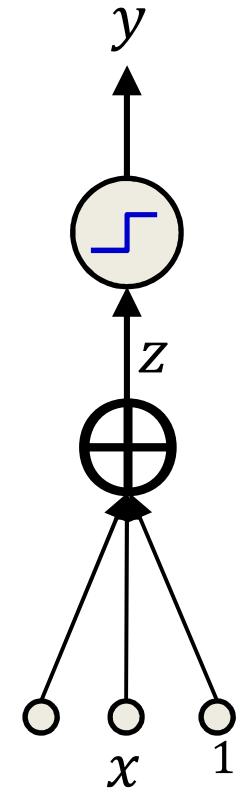
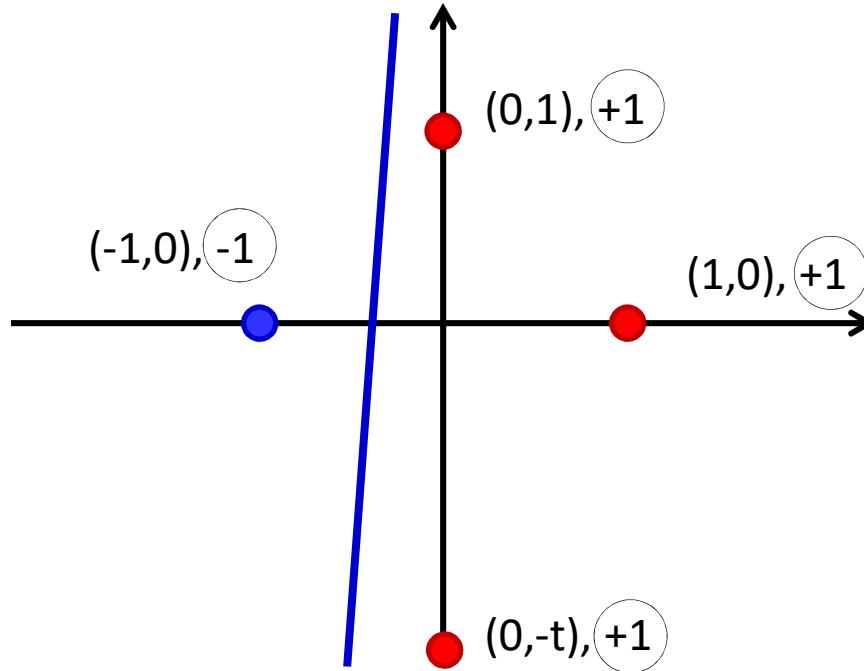
- Let $u = f^{-1}(1 - \varepsilon)$
 - E.g. $u = f^{-1}(0.99)$ representing a 99% confidence in the class
- From the three points we get three independent equations:

$$\begin{aligned} w_x \cdot 1 + w_y \cdot 0 + b &= u \\ w_x \cdot 0 + w_y \cdot 1 + b &= u \\ w_x \cdot -1 + w_y \cdot 0 + b &= -u \end{aligned}$$

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



Backprop vs. Perceptron

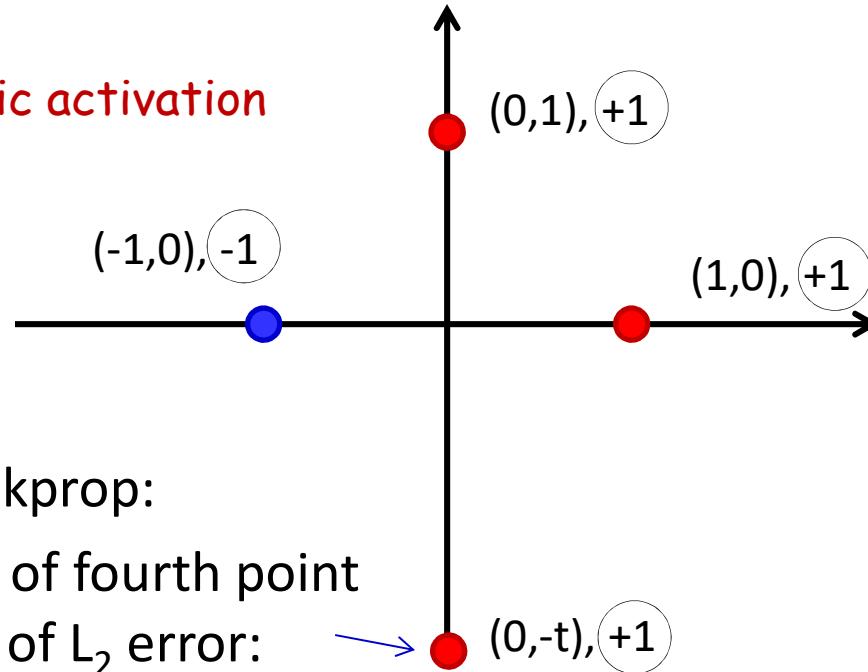


- Now add a fourth point
- t is very large (point near $-\infty$)
- Perceptron trivially finds a solution (may take t^2 iterations)

Backprop

Notation:

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



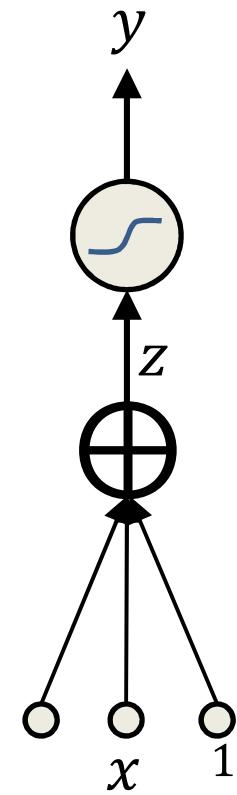
- Consider backprop:
- Contribution of fourth point to derivative of L_2 error:

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

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

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

1- ε is the actual achievable value



Backprop

Notation:

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

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

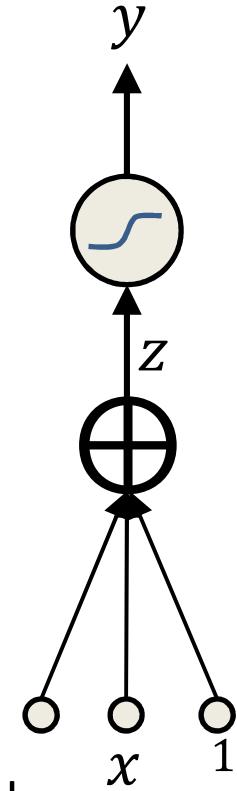
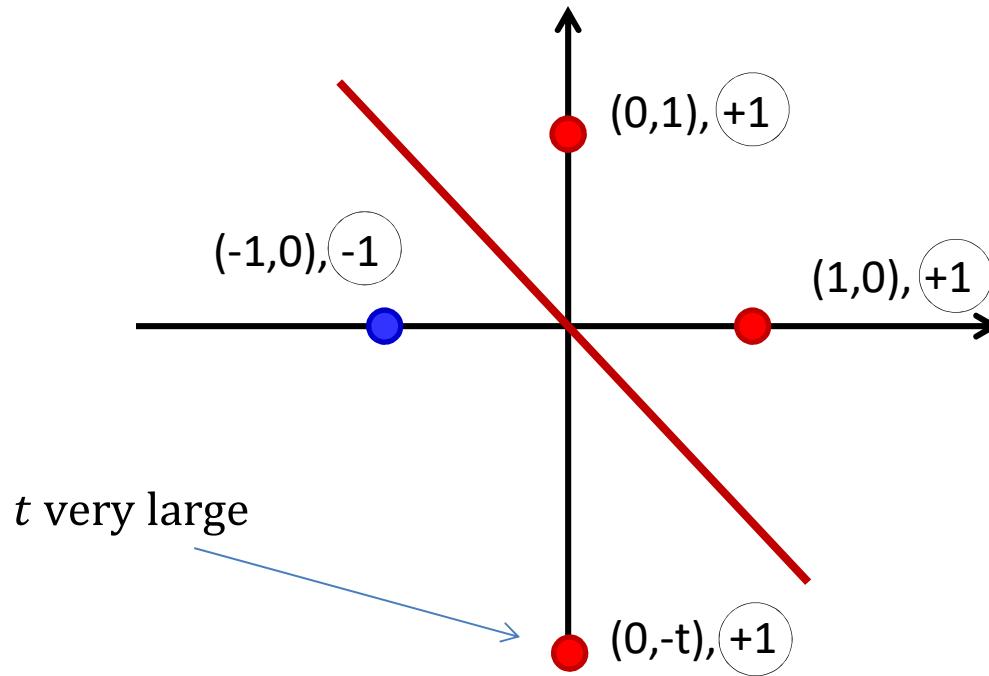
$$\frac{d div_4}{dw_y} = 2(1 - \varepsilon - \sigma(-w_y t + b))\sigma'(-w_y t + b)t$$

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

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

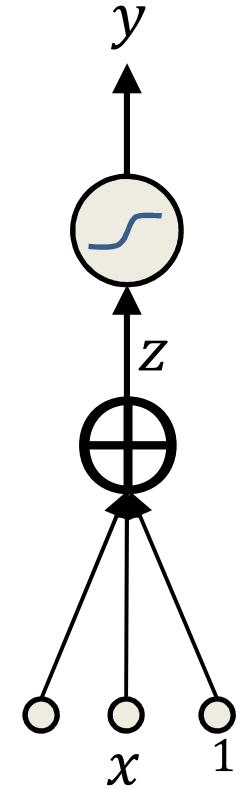
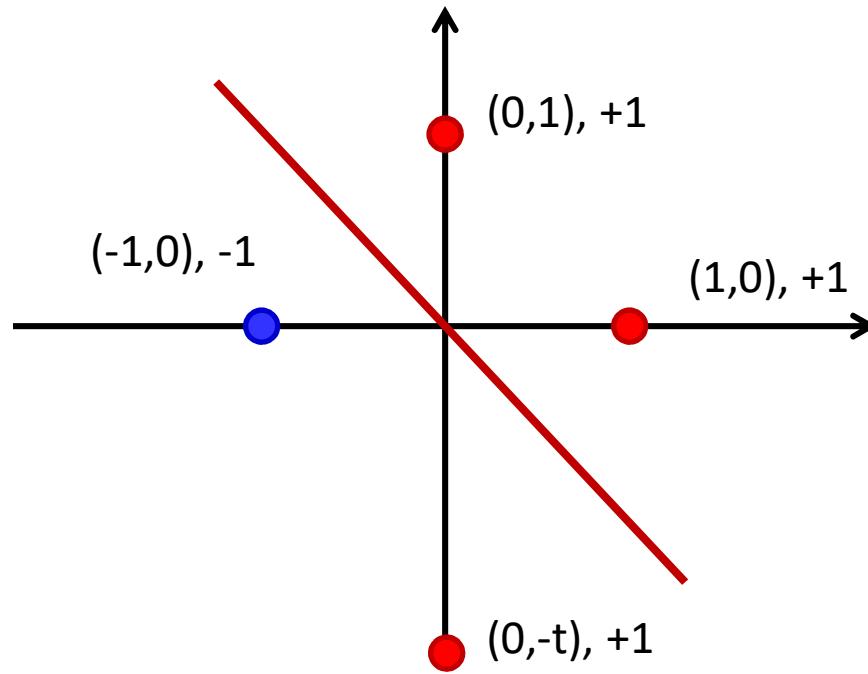
$$\frac{d div_4}{dw_y} = \frac{d div_4}{db} = 0$$

Backprop



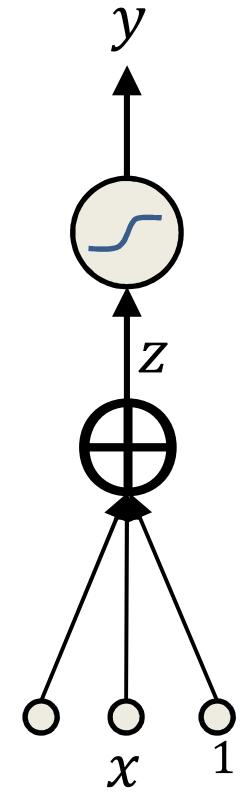
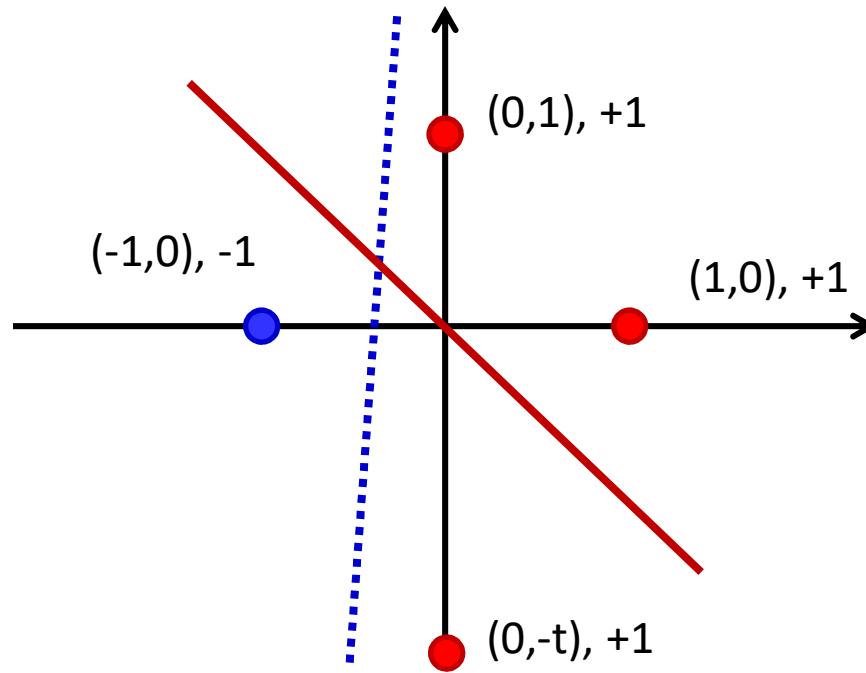
- The fourth point at $(0, -t)$ does not change the gradient of the L_2 divergence near the optimal solution for 3 points
- The optimum solution for 3 points is also a broad *local* minimum (0 gradient) for the 4-point problem!
 - Will be trivially found by backprop nearly all the time
 - Although the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

Backprop



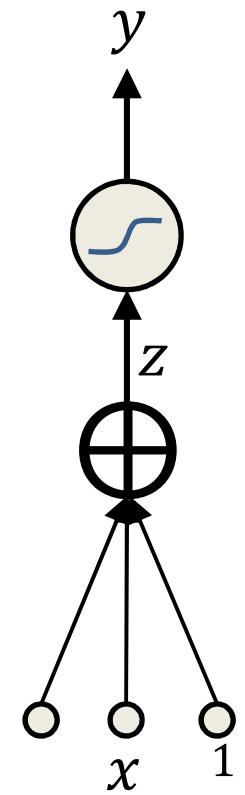
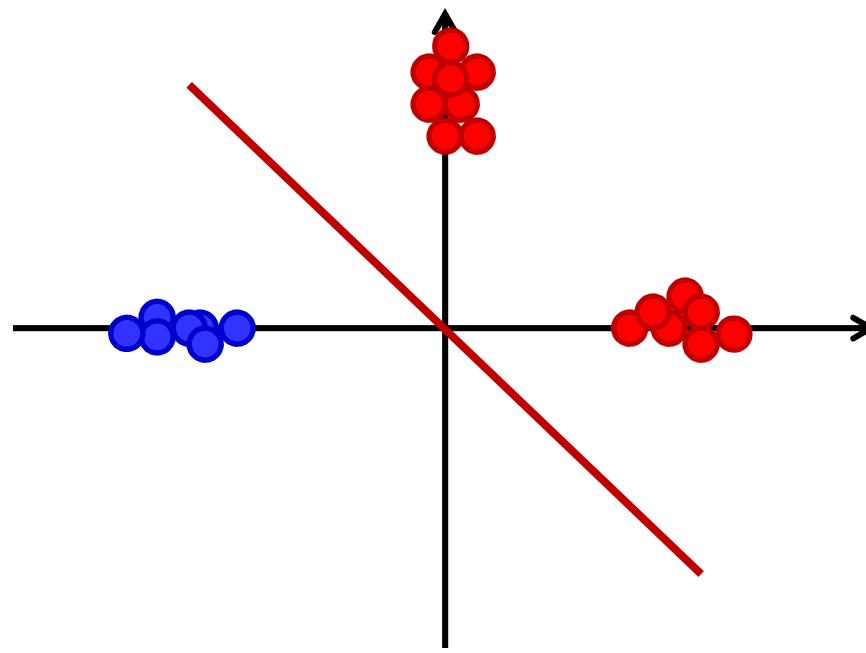
- Local optimum solution found by backprop
- Does not separate the points *even though the points are linearly separable!*

Backprop



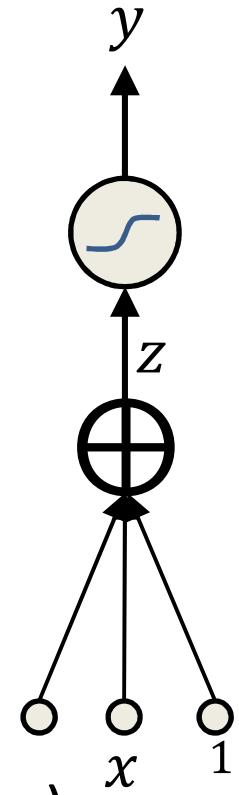
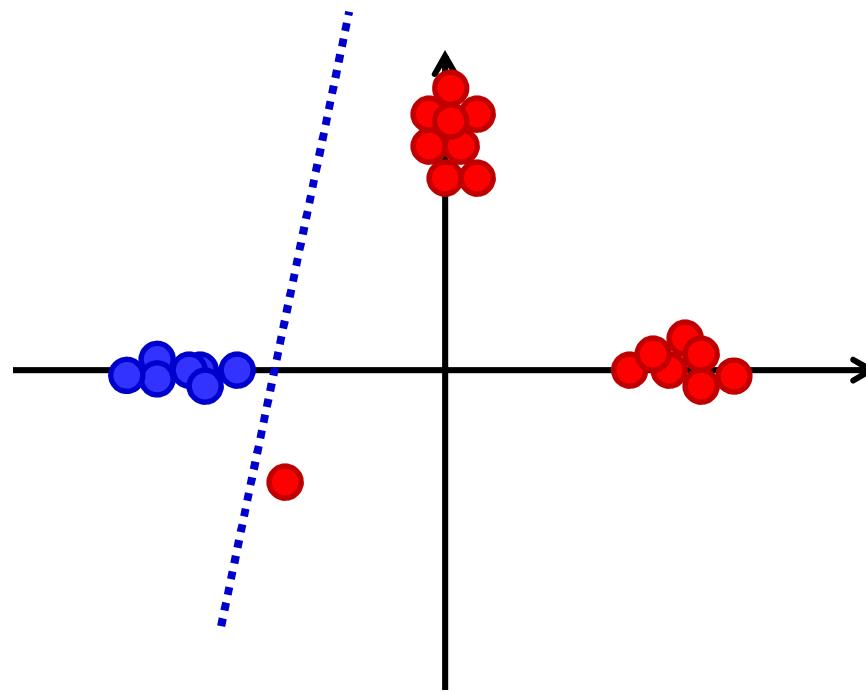
- Solution found by backprop
- Does not separate the points *even though the points are linearly separable!*
- Compare to the perceptron: *Backpropagation fails to separate where the perceptron succeeds*

Backprop fails to separate where perceptron succeeds



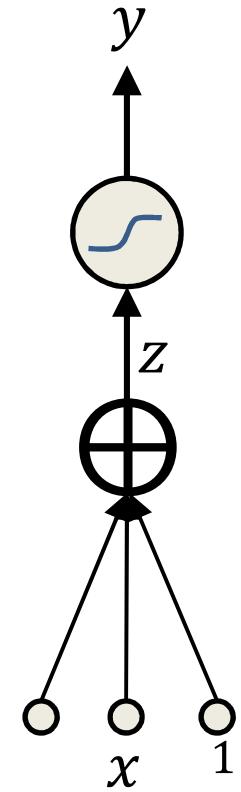
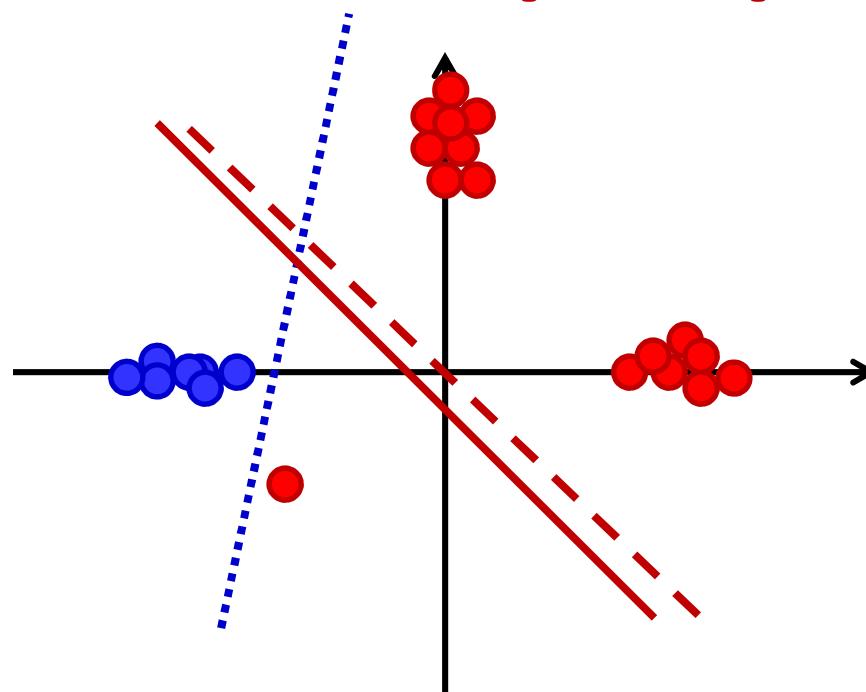
- Brady, Raghavan, Slawny, '89
- *Several* linearly separable training examples
- Simple setup: both backprop and perceptron algorithms find solutions

A more complex problem



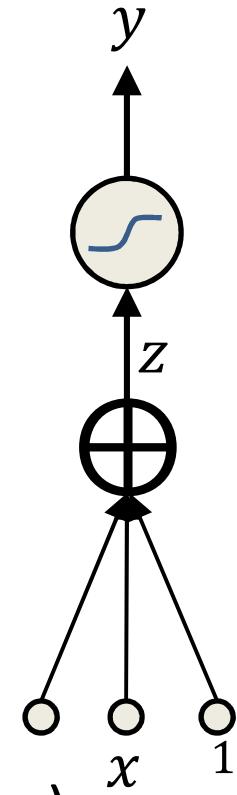
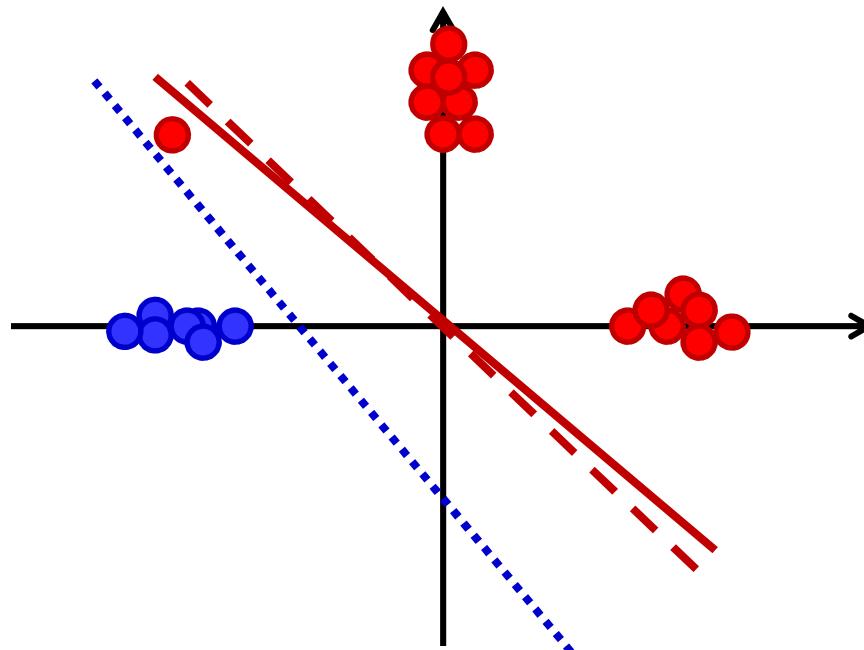
- Adding a “spoiler” (or a small number of spoilers)
 - Perceptron finds the linear separator,

A more complex problem



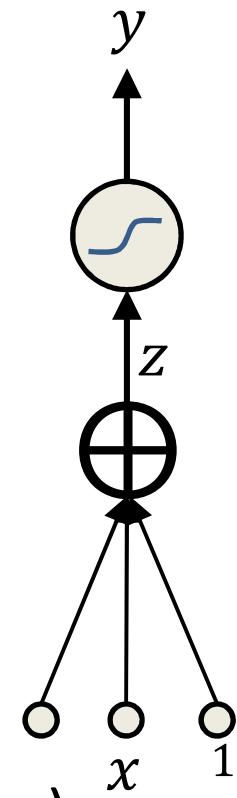
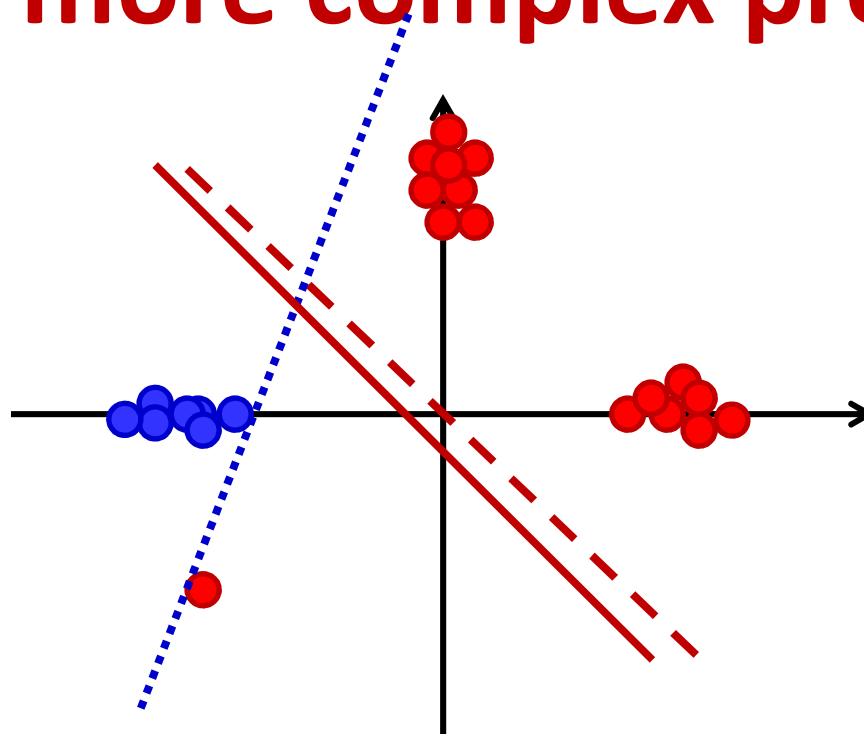
- Adding a “spoiler” (or a small number of spoilers)
 - Perceptron finds the linear separator,
 - Backprop does not find a separator
 - A single additional input does not change the loss function significantly
 - Assuming weights are constrained to be bounded

A more complex problem



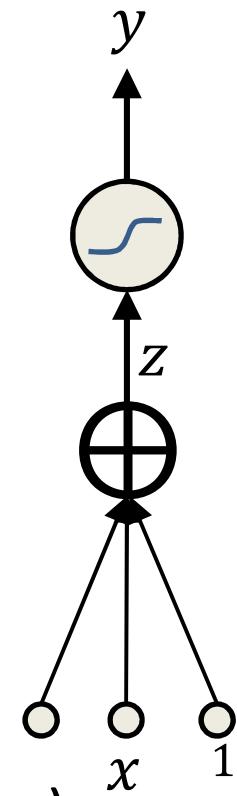
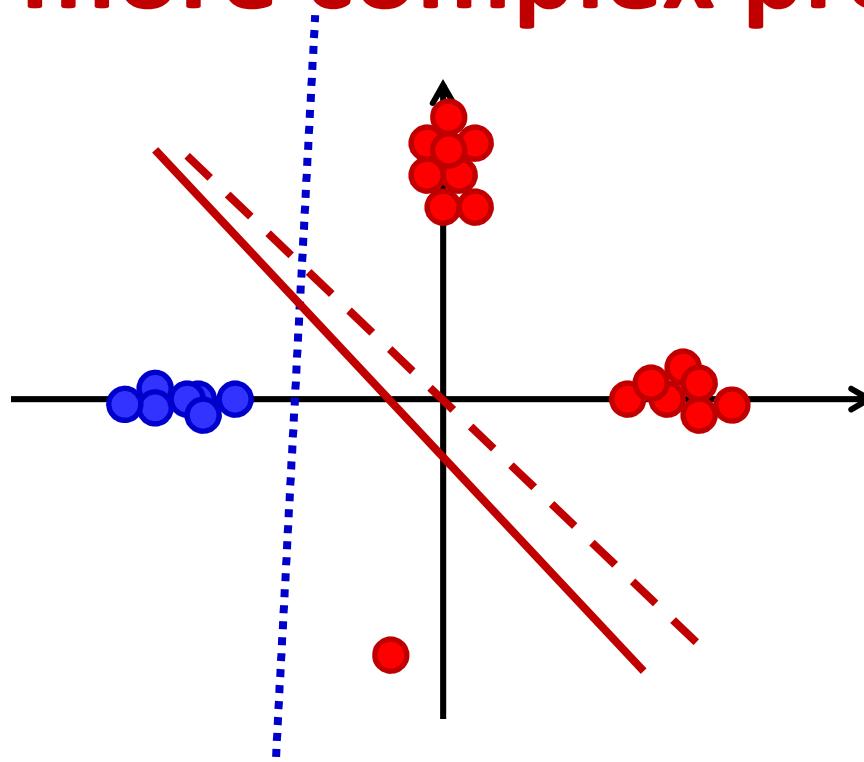
- Adding a “spoiler” (or a small number of spoilers)
 - Perceptron finds the linear separator,
 - For bounded w , backprop does not find a separator
 - A single additional input does not change the loss function significantly

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

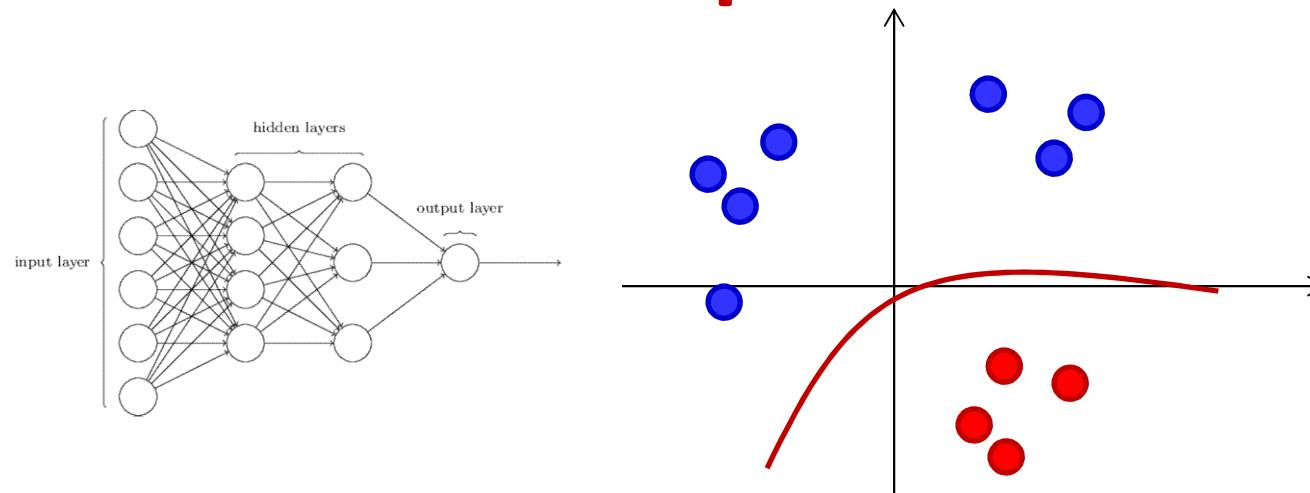


- Adding a “spoiler” (or a small number of spoilers)
 - Perceptron finds the linear separator,
 - For bounded w , Backprop does not find a separator
 - A single additional input does not change the loss function significantly

So what is happening here?

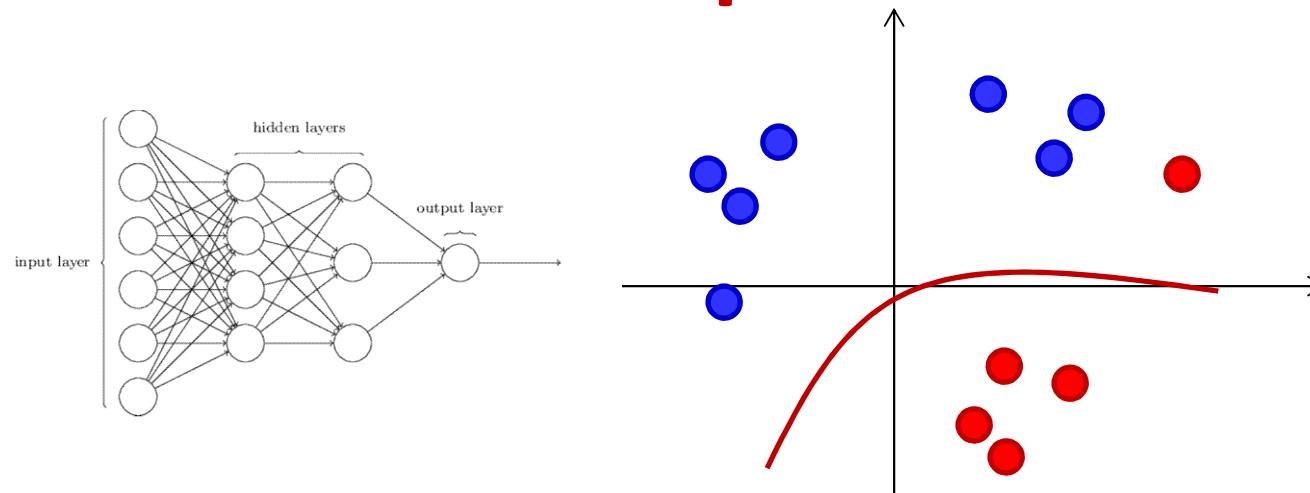
- The perceptron may change greatly upon adding just a single new training instance
 - But it fits the training data well
 - The perceptron rule has *low bias*
 - Makes no errors if possible
 - But high variance
 - Swings wildly in response to small changes to input
- Backprop is minimally changed by 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

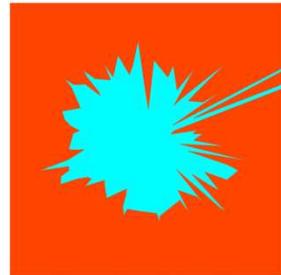


- This is not restricted to single perceptrons
- In an MLP the lower layers “learn a representation” that enables linear separation by higher layers
 - More on this later
- Adding a few “spoilers” will not change their behavior

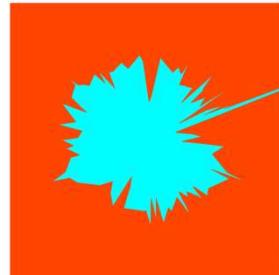
Backpropagation

- 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



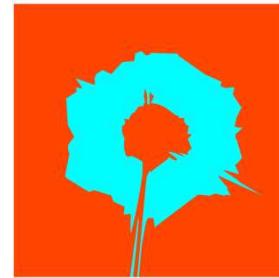
3 layers



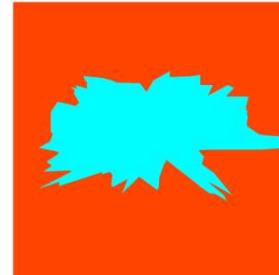
4 layers



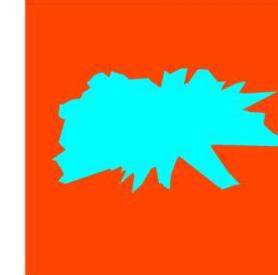
6 layers



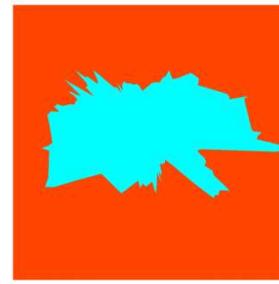
11 layers



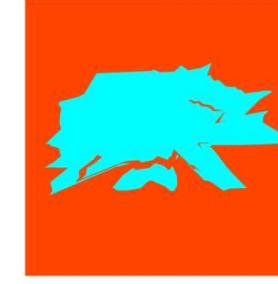
3 layers



4 layers



6 layers



11 layers



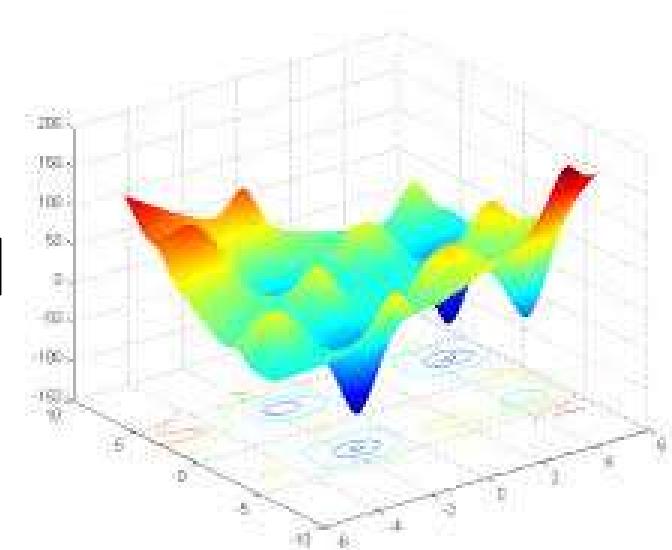
- Dark figures show desired decision boundary (2D)
 - 1000 training points, 660 hidden neurons
 - Network heavily overdesigned even for shallow nets
- **Anecdotal: Variance decreases with**
 - Depth
 - Data

10000 training instances



The Error 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



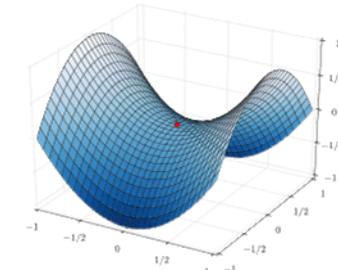
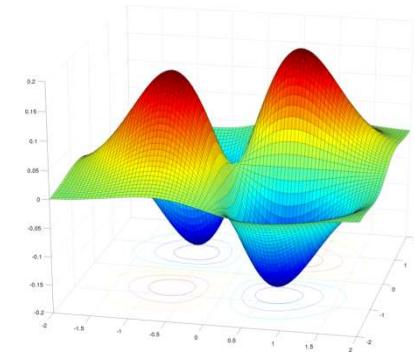
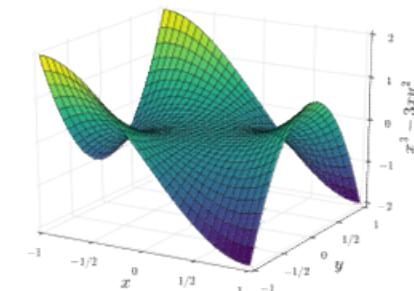
The Error Surface

- **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 Error 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
- **Swirszcz 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

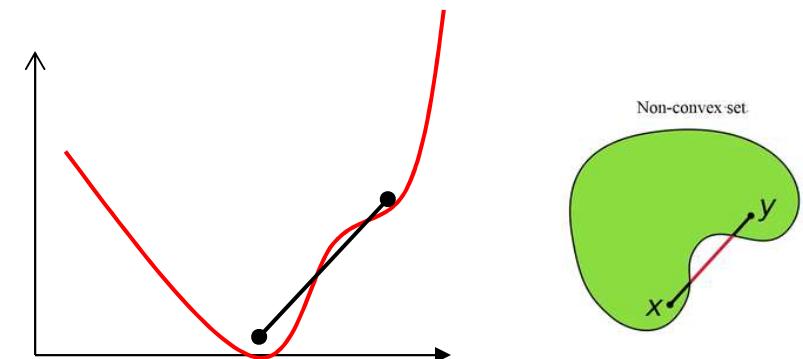
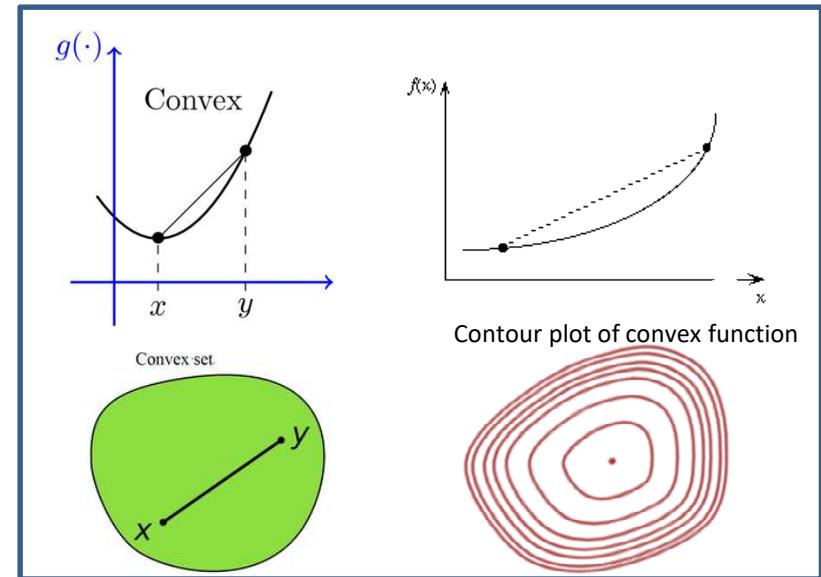


The streetlight effect is a type of observational bias where people only look for whatever they are searching by looking where it is easiest

"I'm searching for my keys."

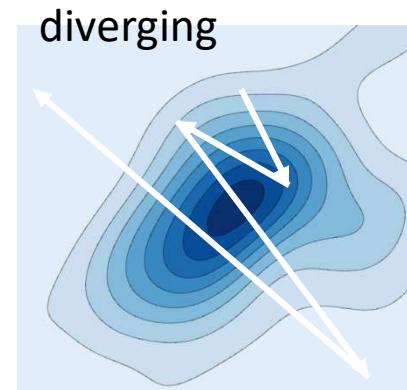
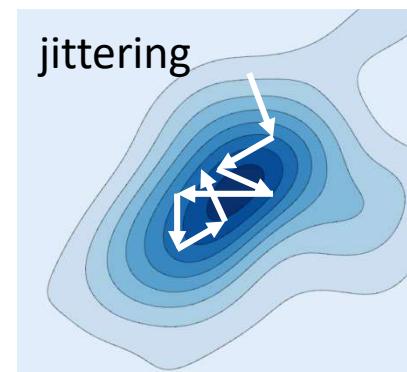
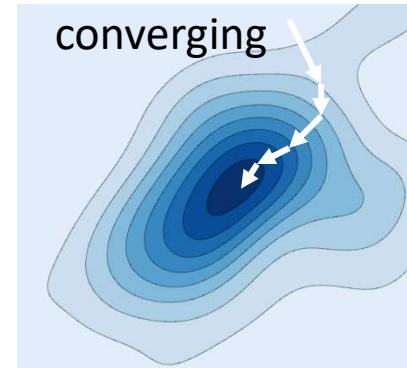
Convex Loss Functions

- 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 error 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

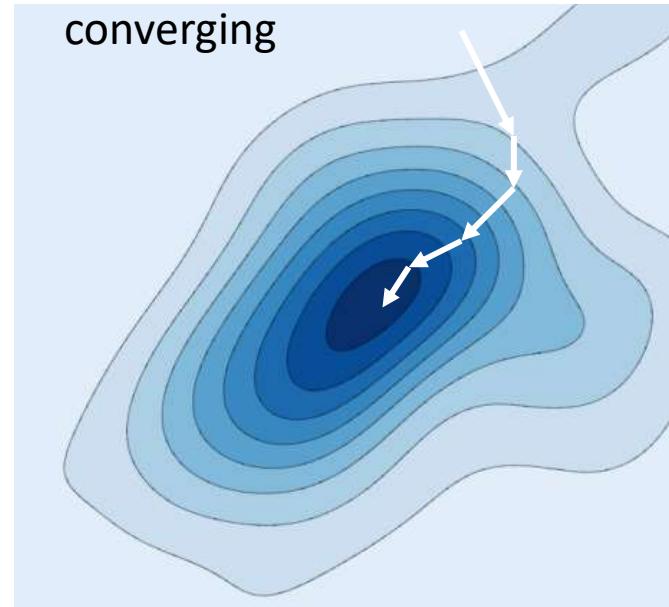
$$R = \frac{|f(x^{(k+1)}) - f(x^*)|}{|f(x^{(k)}) - f(x^*)|}$$

- $x^{(k+1)}$ 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

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

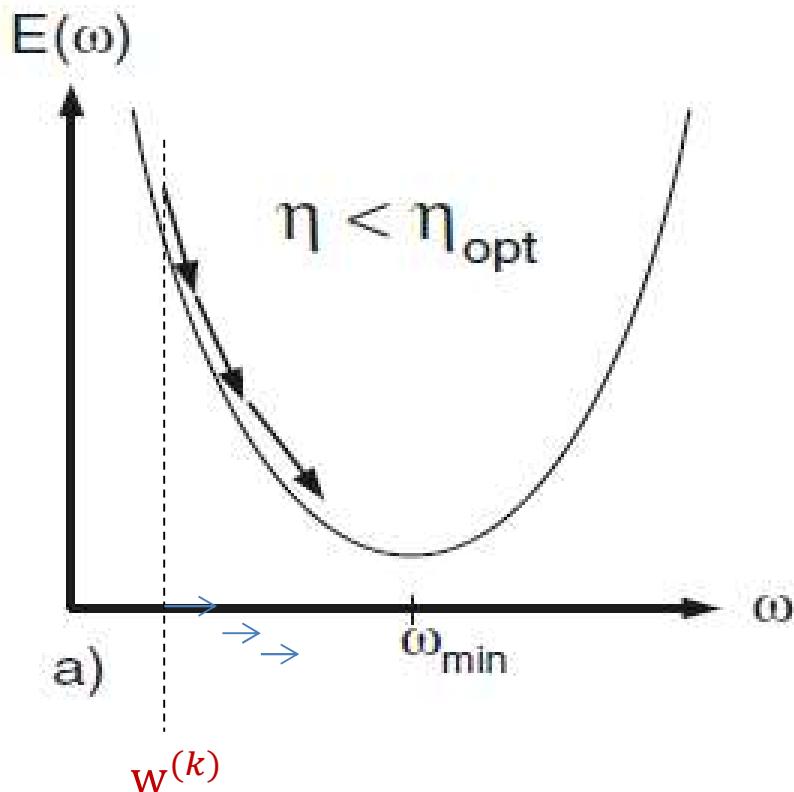


Convergence for quadratic surfaces

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

$$w^{(k+1)} = w^{(k)} - \eta \frac{dE(w^{(k)})}{dw}$$

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

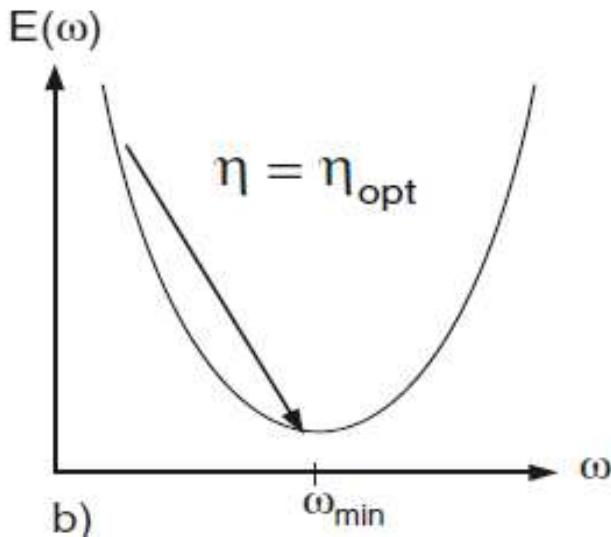


- Gradient descent to find the optimum of a quadratic, starting from $w^{(k)}$
- Assuming fixed step size η
- What is the optimal step size η to get there fastest?

Convergence for quadratic surfaces

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

$$w^{(k+1)} = w^{(k)} - \eta \frac{dE(w^{(k)})}{dw}$$



- Any quadratic objective can be written as

$$\begin{aligned} E(w) &= E(w^{(k)}) + E'(w^{(k)})(w - w^{(k)}) \\ &\quad + \frac{1}{2}E''(w^{(k)})(w - w^{(k)})^2 \end{aligned}$$

– Taylor expansion

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

$$w_{min} = w^{(k)} - E''(w^{(k)})^{-1}E'(w^{(k)})$$

- Note:

$$\frac{dE(w^{(k)})}{dw} = E'(w^{(k)})$$

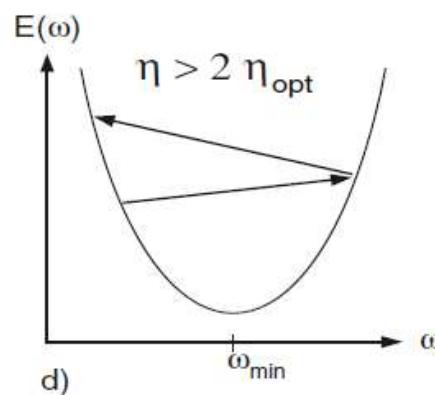
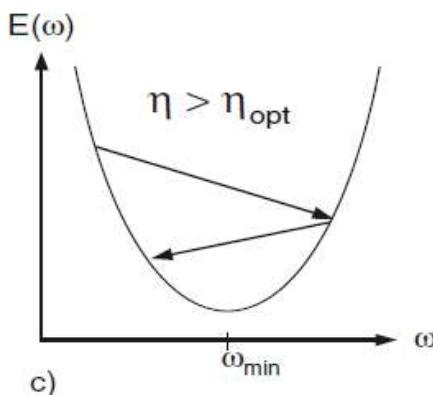
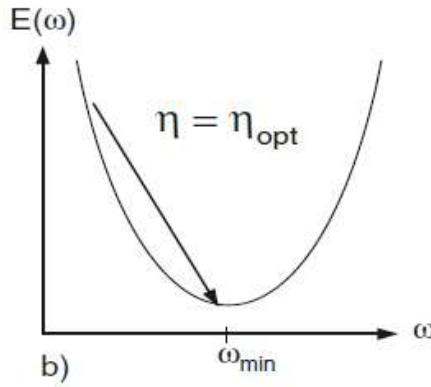
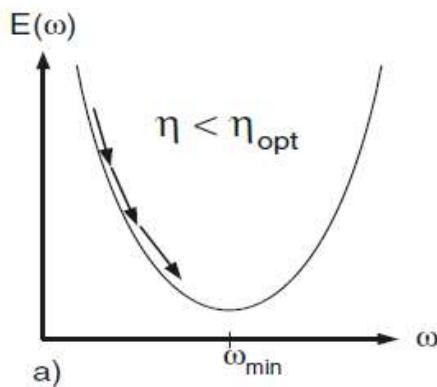
- Comparing to the gradient descent rule, we see that we can arrive at the optimum in a single step using the optimum step size

$$\eta_{opt} = E''(w^{(k)})^{-1} = a^{-1}$$

With non-optimal step size

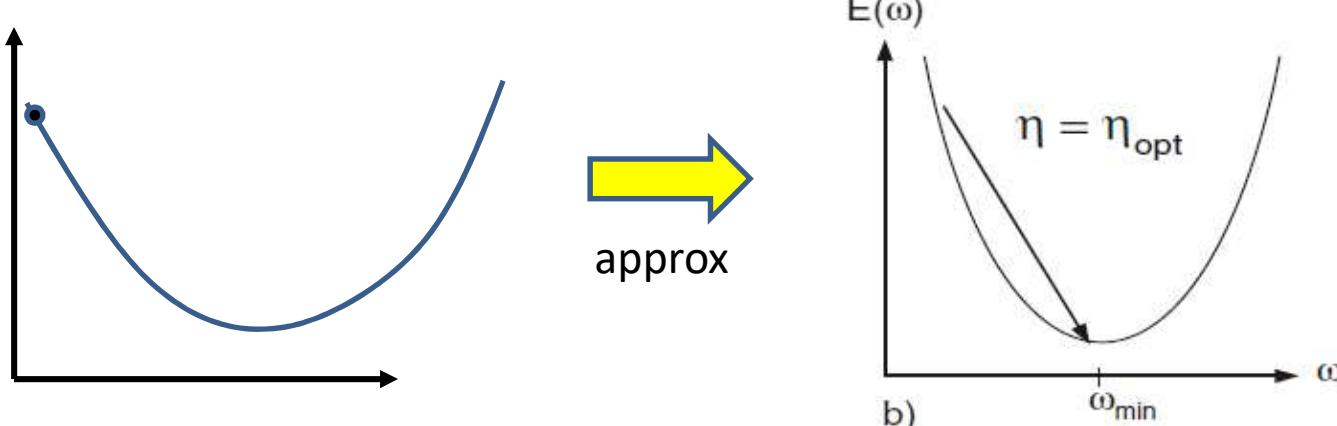
$$w^{(k+1)} = w^{(k)} - \eta \frac{dE(w^{(k)})}{dw}$$

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

$$E \approx E(w^{(k)}) + (w - w^{(k)}) \frac{dE(w^{(k)})}{dw} + \frac{1}{2} (w - w^{(k)})^2 \frac{d^2 E(w^{(k)})}{dw^2} + \dots$$

- Taylor expansion

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

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

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

For functions of *multivariate* inputs

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

- Consider a simple quadratic convex (paraboloid) function

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

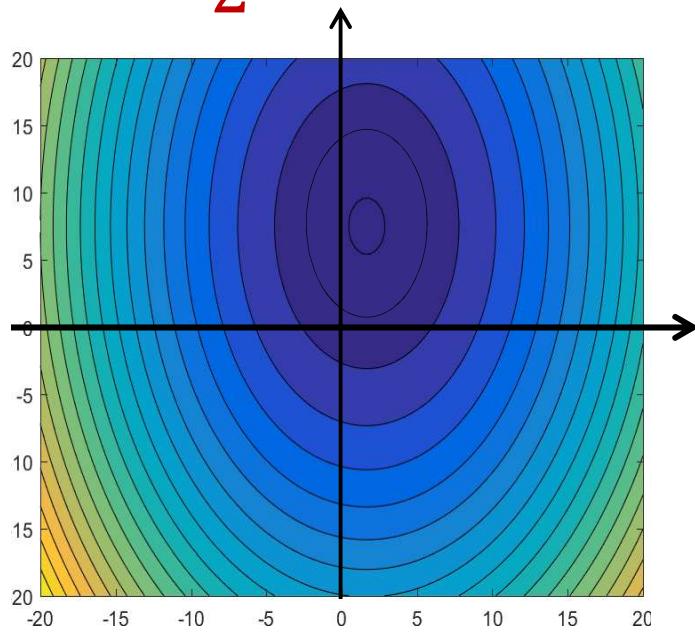
- Since $E^T = E$ (E is scalar), \mathbf{A} can always be made symmetric
 - For **convex** E , \mathbf{A} is always positive definite, and has positive eigenvalues
- When \mathbf{A} is diagonal:

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

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

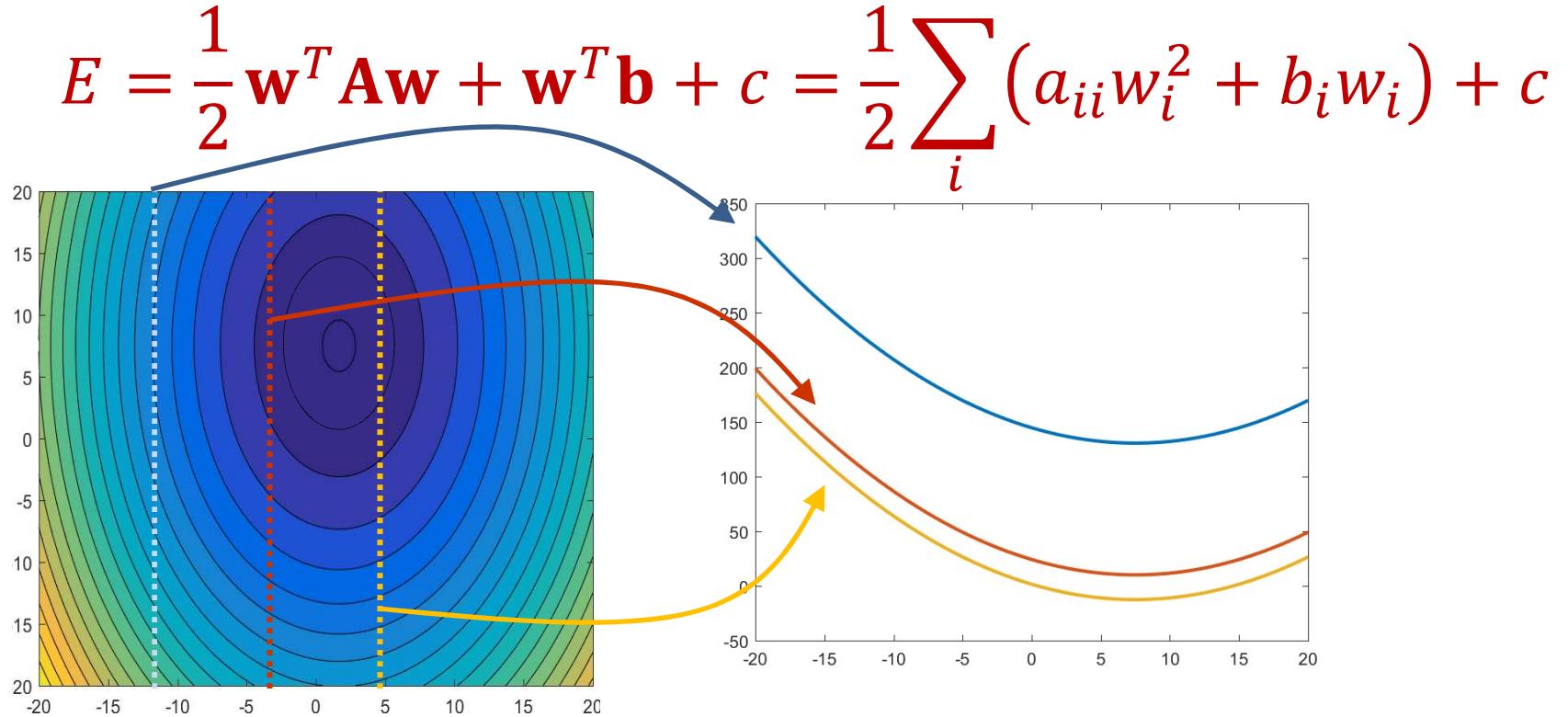
Multivariate Quadratic with Diagonal A

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



- Equal-value contours will be parallel to the axis

Multivariate Quadratic with Diagonal A

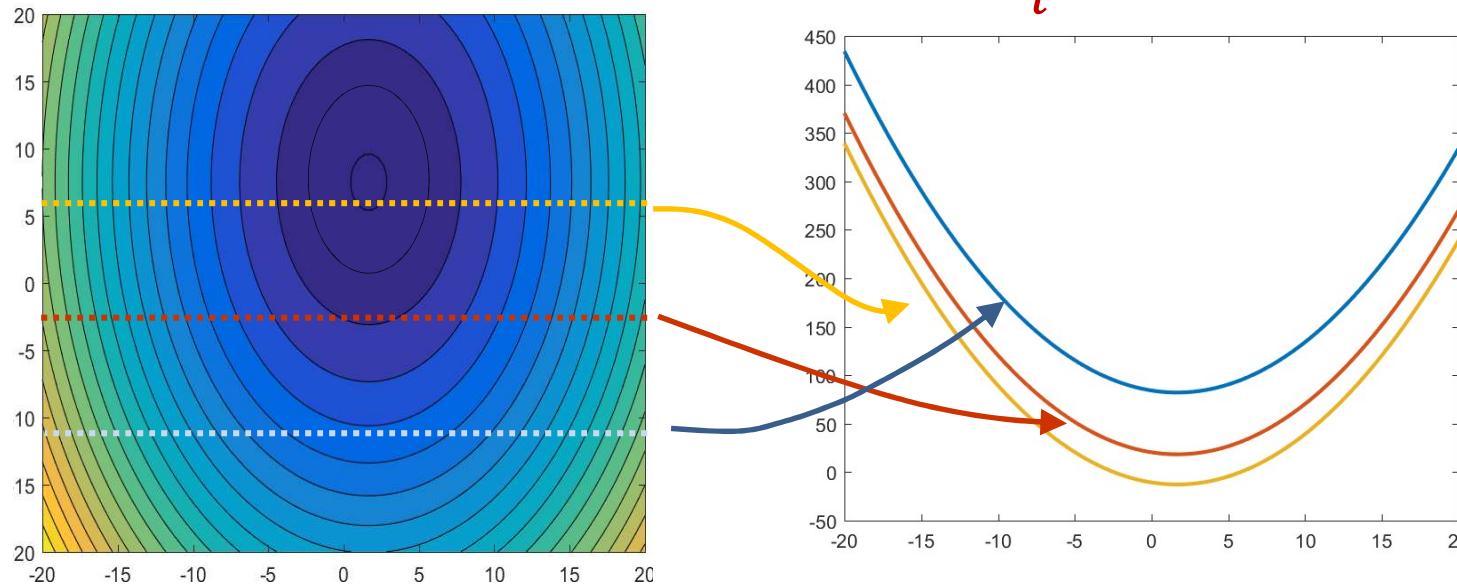


- Equal-value contours will be parallel to the axis
 - All “slices” parallel to an axis are shifted versions of one another

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

Multivariate Quadratic with Diagonal A

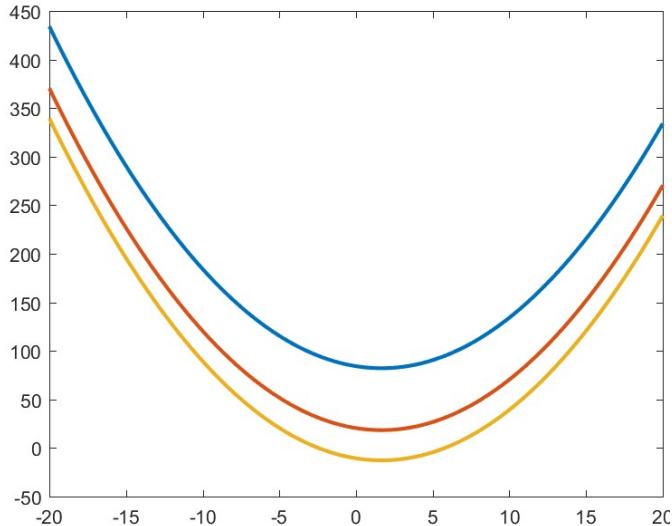
$$E = \frac{1}{2} \mathbf{w}^T \mathbf{A} \mathbf{w} + \mathbf{w}^T \mathbf{b} + c = \frac{1}{2} \sum_i (a_{ii} w_i^2 + b_i w_i) + c$$



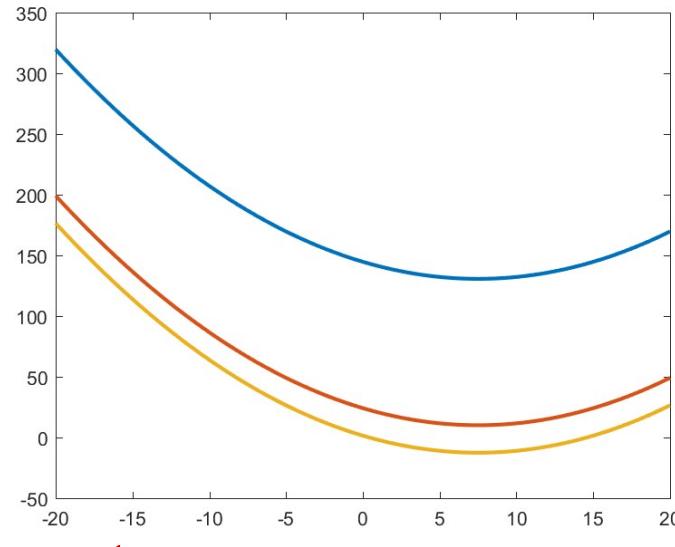
- Equal-value contours will be parallel to the axis
 - All “slices” parallel to an axis are shifted versions of one another

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

“Descents” are uncoupled



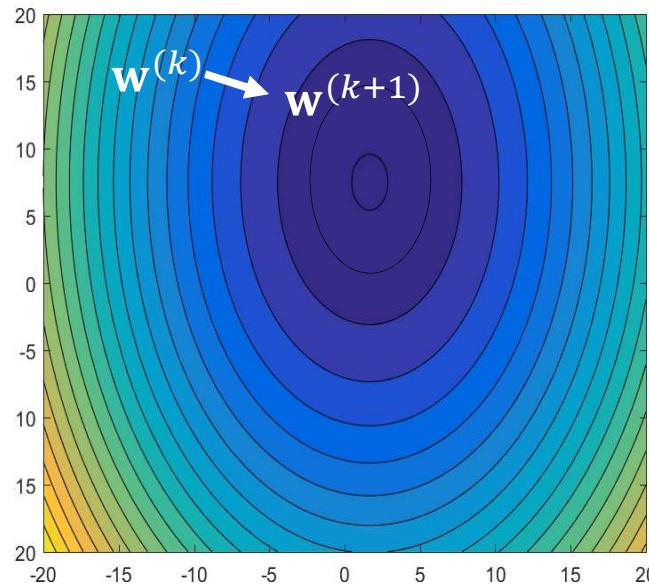
$$E = \frac{1}{2} a_{11} w_1^2 + b_1 w_1 + c + C(\neg w_1)$$
$$\eta_{1,opt} = a_{11}^{-1}$$



$$E = \frac{1}{2} a_{22} w_2^2 + b_2 w_2 + c + C(\neg w_2)$$
$$\eta_{2,opt} = a_{22}^{-1}$$

- 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



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

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

- 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

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

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

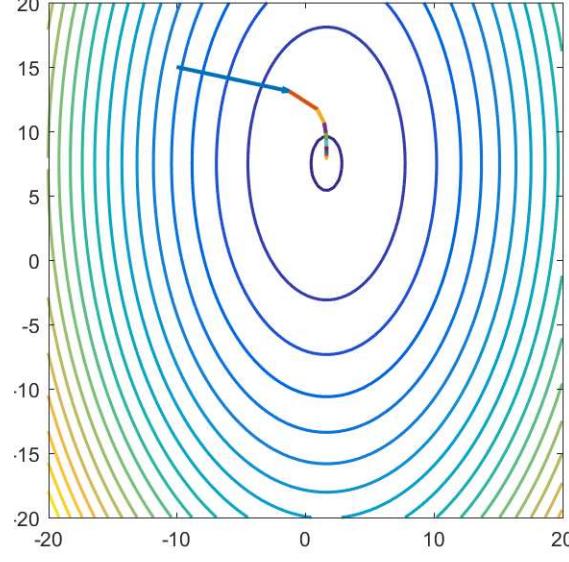
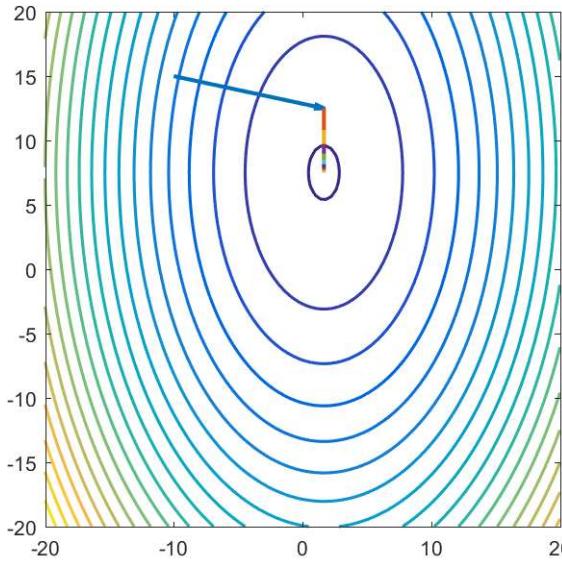
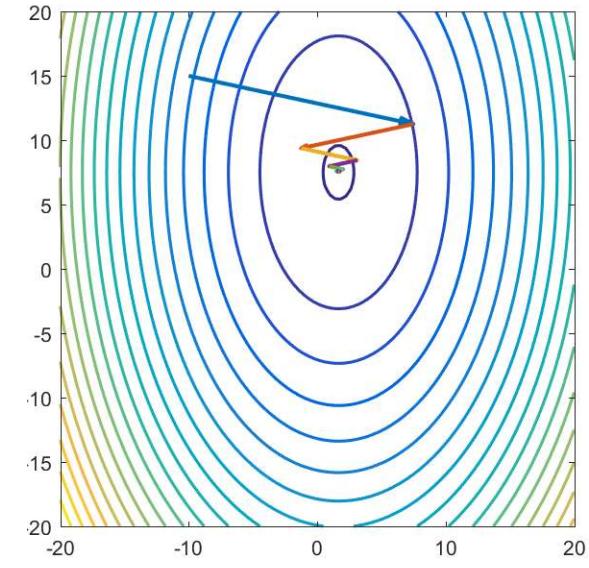
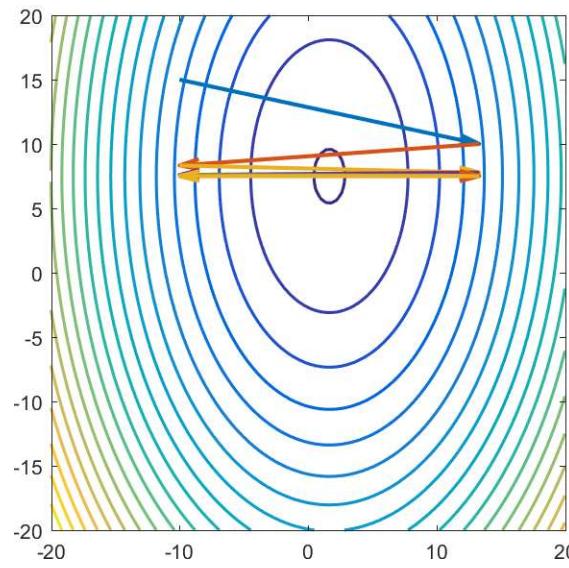
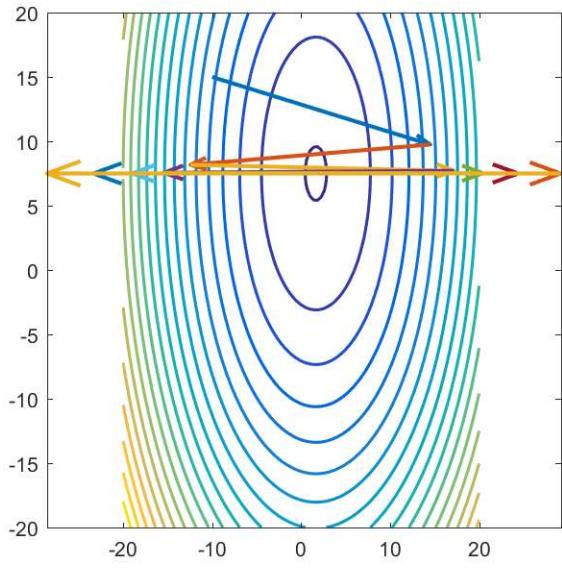
$$\eta_{i,opt} = \left(\frac{d^2 E(w_i^{(k)})}{dw_i^2} \right)^{-1} = a_{ii}^{-1}$$

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

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

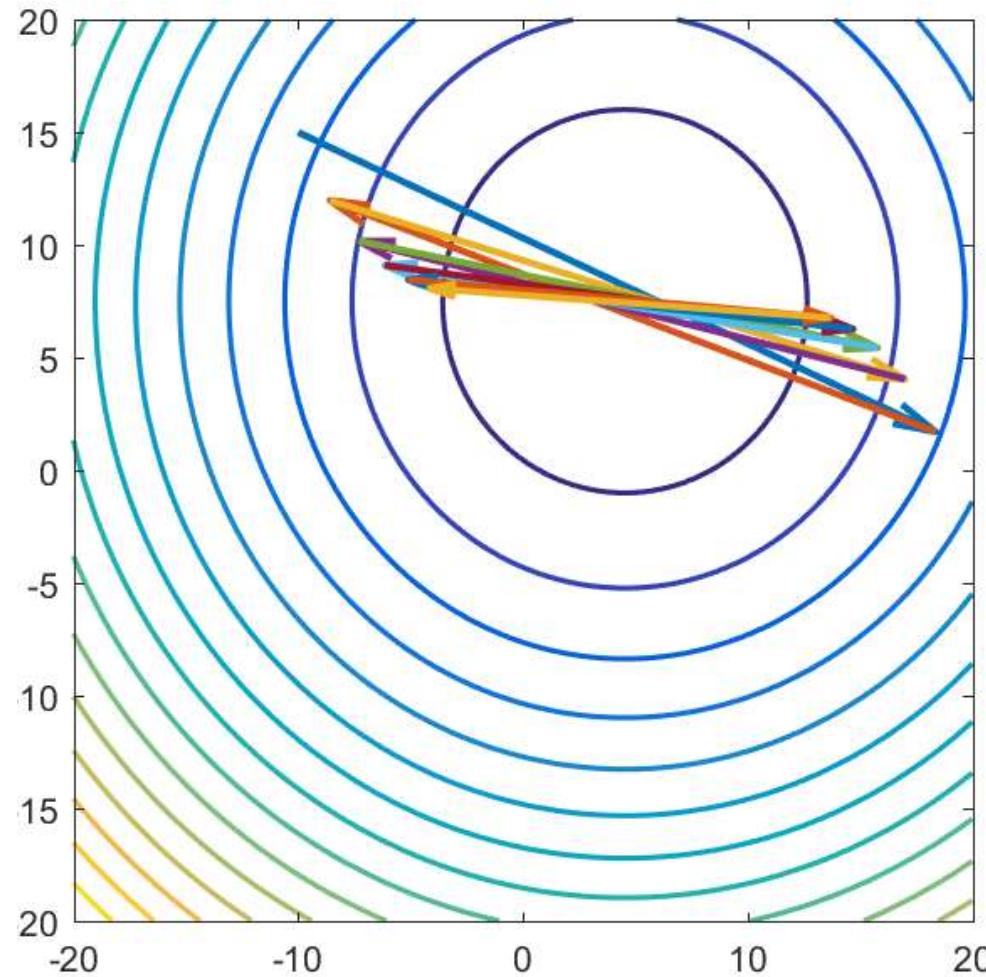
- Otherwise the learning will diverge
- This, however, makes the learning very slow
 - And will oscillate in all directions where $\eta_{i,opt} \leq \eta < 2\eta_{i,opt}$

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



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

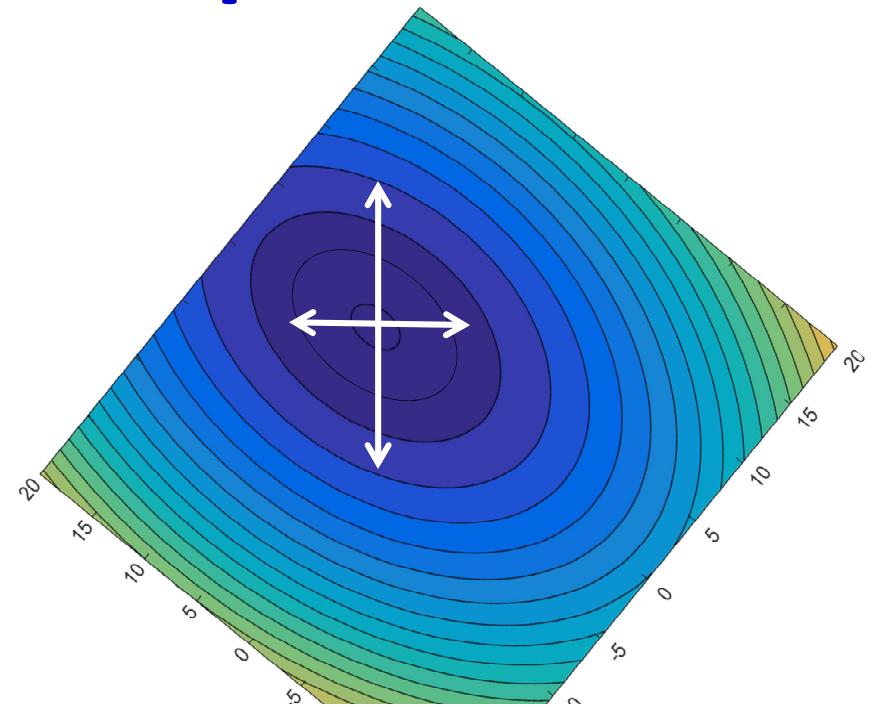
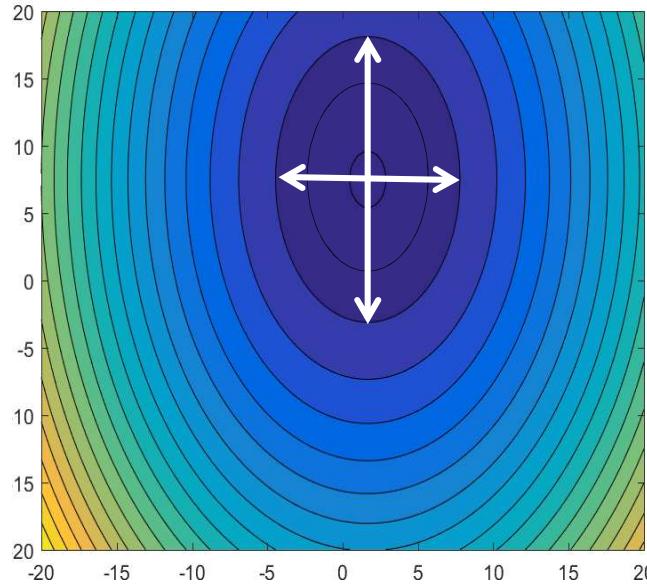
Convergence

- 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

More Problems

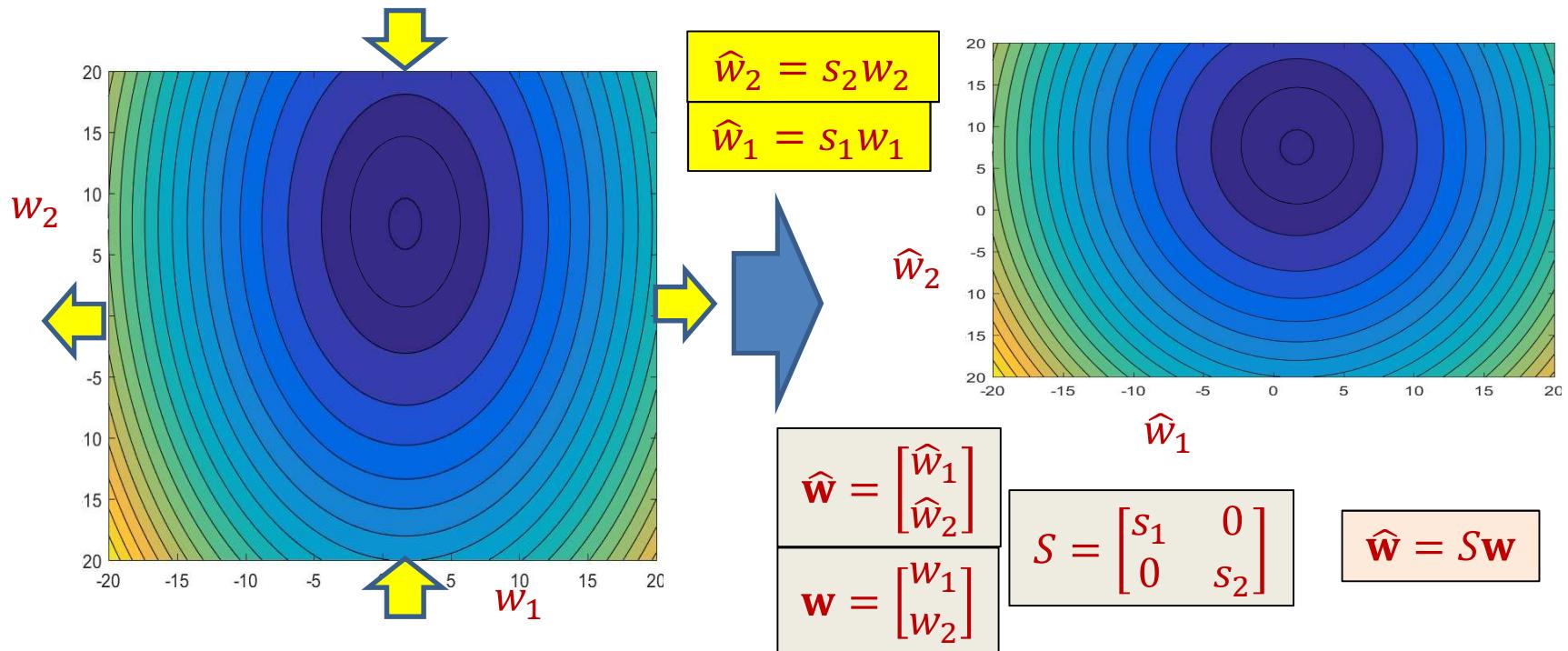
- For quadratic (strongly) convex functions, gradient descent is exponentially fast
 - Linear convergence
 - Assuming learning rate is non-divergent
- For generic (Lipschitz Smooth) convex functions however, it is very slow
$$|f(w^{(k)}) - f(w^*)| \propto \frac{1}{k} |f(w^{(0)}) - f(w^*)|$$
 - And inversely proportional to learning rate
$$|f(w^{(k)}) - f(w^*)| \leq \frac{1}{2\eta k} |w^{(0)} - w^*|$$
 - Takes $O(1/\epsilon)$ iterations to get to within ϵ of the solution
- An inappropriate learning rate will destroy your happiness

The 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



- 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

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

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

$$S = \begin{bmatrix} s_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & s_N \end{bmatrix}, \quad \hat{\mathbf{w}} = S\mathbf{w}$$

- And

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

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

$$S = \begin{bmatrix} s_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & s_N \end{bmatrix}, \quad \hat{\mathbf{w}} = S\mathbf{w}$$

- And

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

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

Scaling the axes

- We have

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

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

$$\begin{aligned} E &= \frac{1}{2} \hat{\mathbf{w}}^T \hat{\mathbf{w}} + \hat{\mathbf{b}}^T \hat{\mathbf{w}} + c \\ &= \frac{1}{2} \mathbf{w}^T S^T S \mathbf{w} + \hat{\mathbf{b}}^T S \mathbf{w} + c \end{aligned}$$

- Equating linear and quadratic coefficients, we get

$$S^T S = \mathbf{A}, \quad \hat{\mathbf{b}}^T S = -\mathbf{b}^T$$

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

Scaling the axes

- We have

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

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

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

- Solving for S we get

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

Scaling the axes

- We have

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

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

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

- Solving for S we get

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

The Inverse Square Root of A

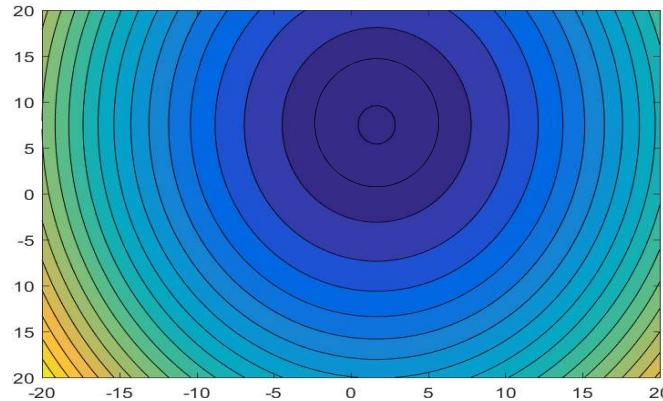
- For *any* positive definite \mathbf{A} , we can write

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

- Eigen decomposition
 - \mathbf{E} is an orthogonal matrix
 - $\boldsymbol{\Lambda}$ is a diagonal matrix of non-zero diagonal entries
-
- Defining $\mathbf{A}^{0.5} = \mathbf{E}\boldsymbol{\Lambda}^{0.5}\mathbf{E}^T$
 - Check $(\mathbf{A}^{0.5})^T\mathbf{A}^{0.5} = \mathbf{E}\boldsymbol{\Lambda}\mathbf{E}^T = \mathbf{A}$

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

Returning to our problem

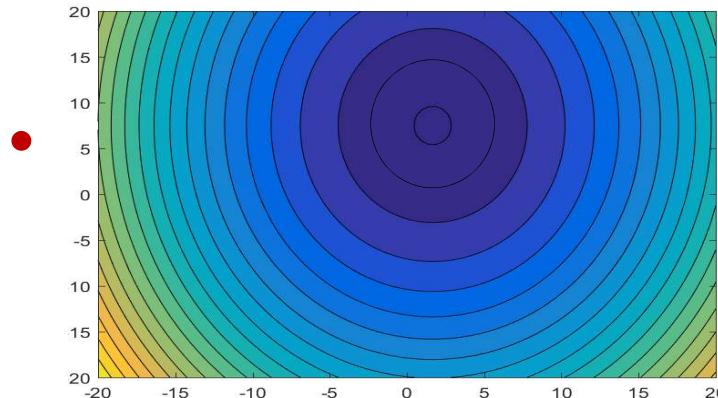


$$E = \frac{1}{2} \hat{\mathbf{w}}^T \hat{\mathbf{w}} + \hat{\mathbf{b}}^T \hat{\mathbf{w}} + c$$

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

$$\begin{aligned}\nabla_{\hat{\mathbf{w}}} E &= \hat{\mathbf{w}}^T + \hat{\mathbf{b}}^T \\ &= \mathbf{w}^T \mathbf{A}^{0.5} + \mathbf{b}^T \mathbf{A}^{-0.5} \\ &= (\mathbf{w}^T \mathbf{A} + \mathbf{b}^T) \mathbf{A}^{-0.5} \\ &= \nabla_{\mathbf{w}} E \cdot \mathbf{A}^{-0.5}\end{aligned}$$

Returning to our problem

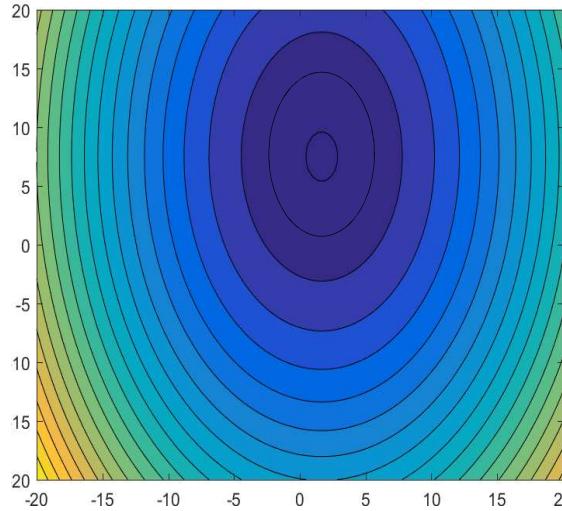


$$E = \frac{1}{2} \hat{\mathbf{w}}^T \hat{\mathbf{w}} + \hat{\mathbf{b}}^T \hat{\mathbf{w}} + c$$

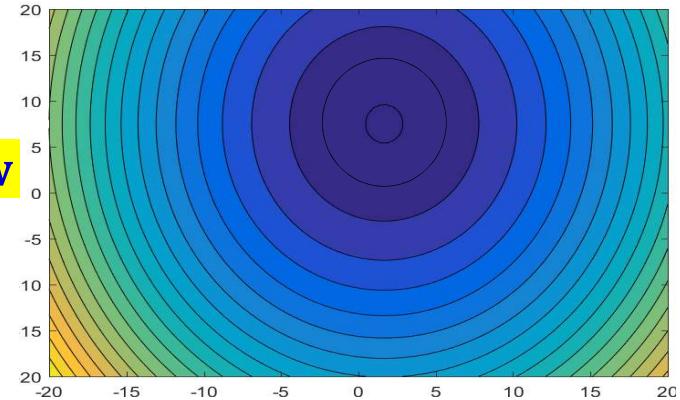
- Gradient descent rule:
 - $\hat{\mathbf{w}}^{(k+1)} = \hat{\mathbf{w}}^{(k)} - \eta \nabla_{\hat{\mathbf{w}}} E(\hat{\mathbf{w}}^{(k)})^T$
 - Learning rate is now independent of direction
- Using $\hat{\mathbf{w}} = \mathbf{A}^{0.5} \mathbf{w}$, and $\nabla_{\hat{\mathbf{w}}} E(\hat{\mathbf{w}})^T = \mathbf{A}^{-0.5} \nabla_{\mathbf{w}} E(\mathbf{w})^T$

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

Modified update rule



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



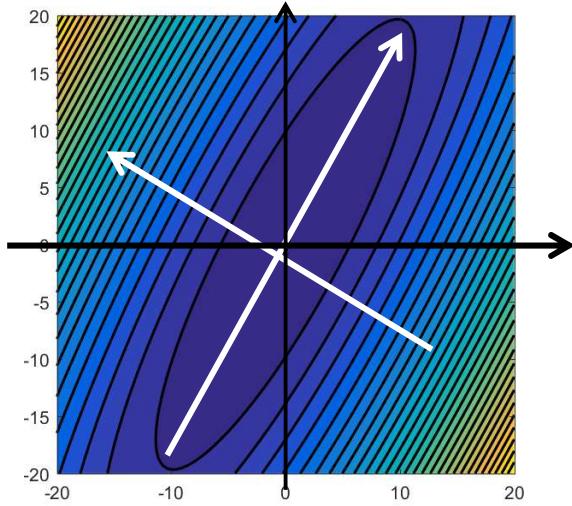
$$E = \frac{1}{2} \mathbf{w}^T \mathbf{A} \mathbf{w} + \mathbf{b}^T \mathbf{w} + c$$

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

- $\hat{\mathbf{w}}^{(k+1)} = \hat{\mathbf{w}}^{(k)} - \eta \nabla_{\hat{\mathbf{w}}} E(\hat{\mathbf{w}}^{(k)})^T$
- Leads to the modified gradient descent rule

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

For non-axis-aligned quadratics..

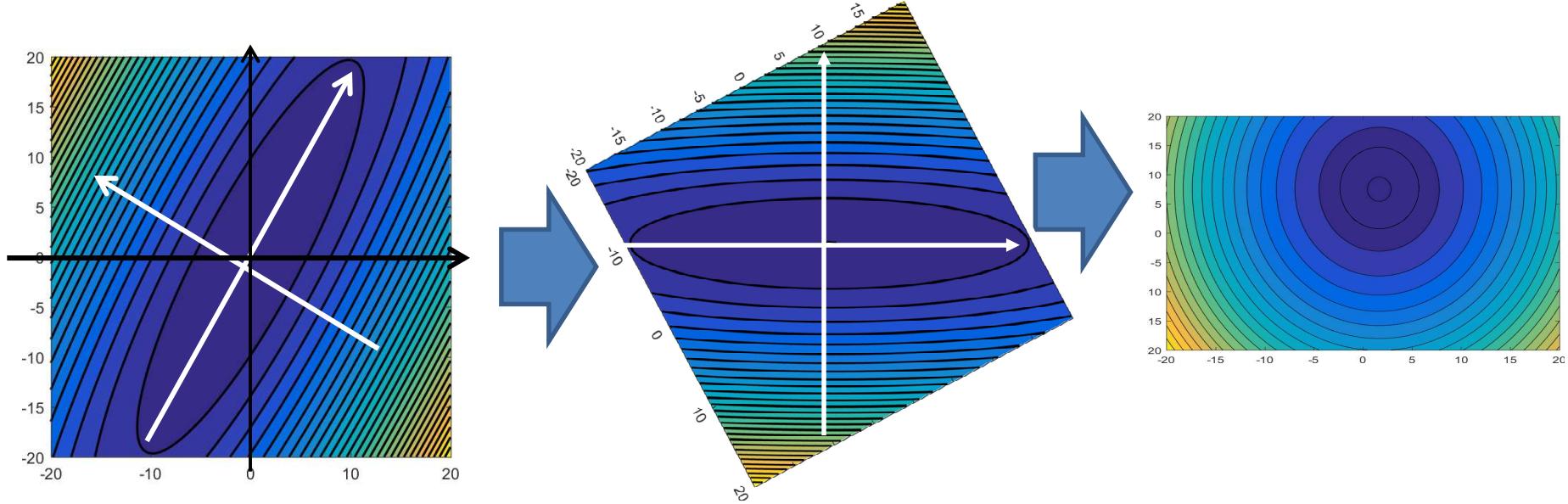


$$E = \frac{1}{2} \mathbf{w}^T \mathbf{A} \mathbf{w} + \mathbf{w}^T \mathbf{b} + c$$

$$\begin{aligned} E = & \frac{1}{2} \sum_i a_{ii} w_i^2 + \sum_{i \neq j} a_{ij} w_i w_j \\ & + \sum_i b_i w_i + c \end{aligned}$$

- If \mathbf{A} is not diagonal, the contours are not axis-aligned
 - Because of the cross-terms $a_{ij} w_i w_j$
 - 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}

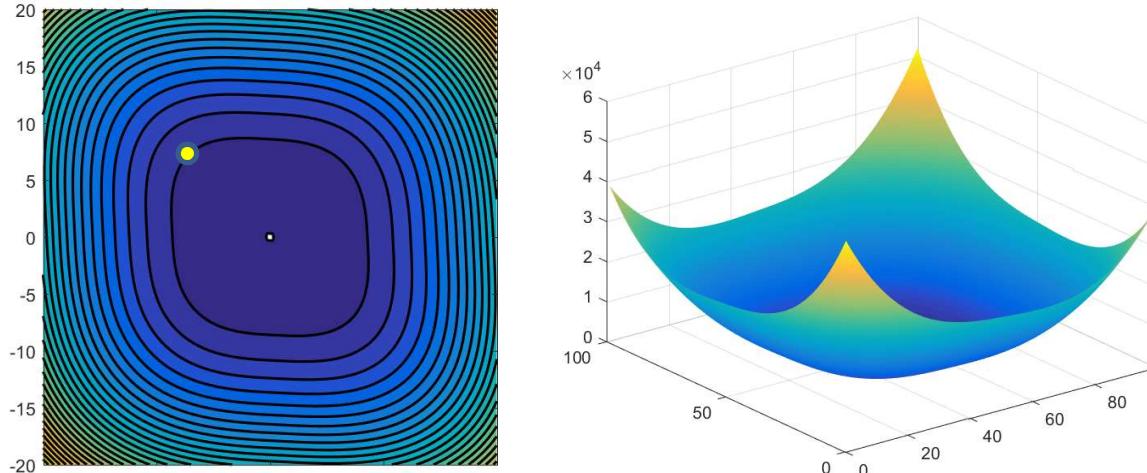
For non-axis-aligned quadratics..



- 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:

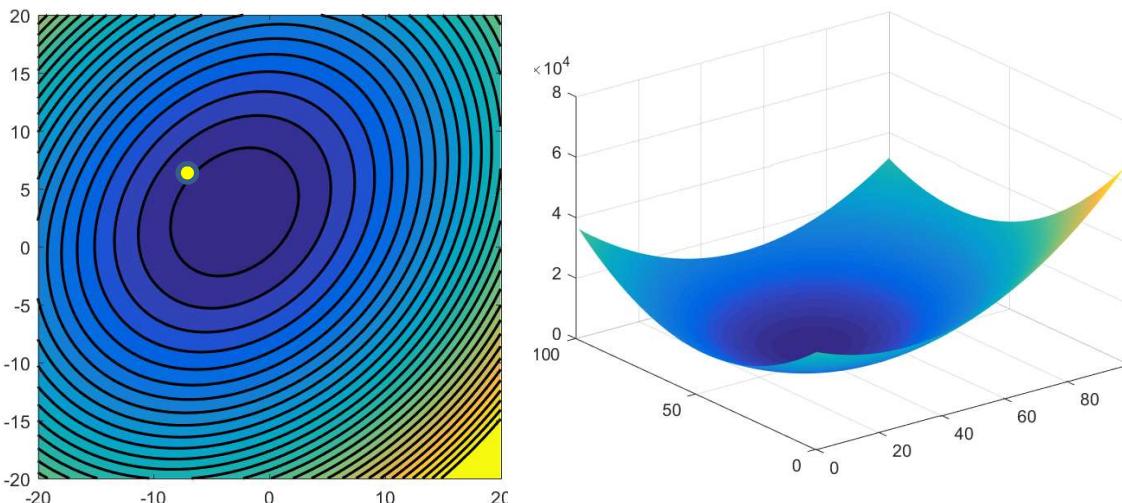
$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \mathbf{A}^{-1} \mathbf{b}$$

Generic differentiable *multivariate convex functions*

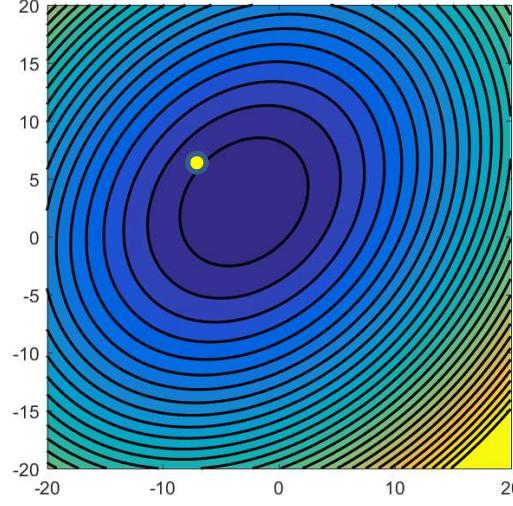
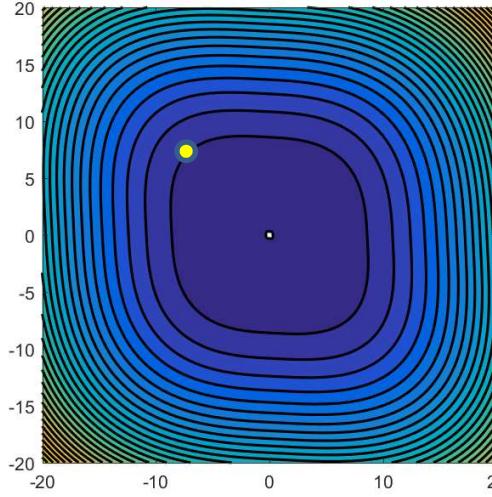


- Taylor expansion

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



Generic differentiable *multivariate convex functions*



- Taylor expansion

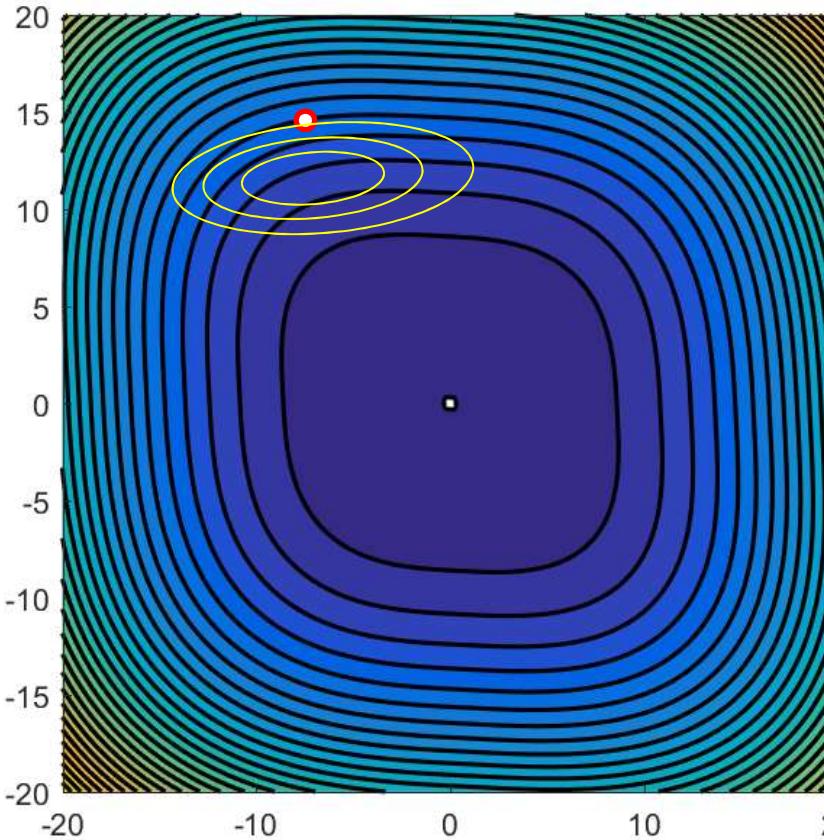
$$E(\mathbf{w}) \approx E(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} E(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2} (\mathbf{w} - \mathbf{w}^{(k)})^T H_E(\mathbf{w}^{(k)}) (\mathbf{w} - \mathbf{w}^{(k)}) + \dots$$

- 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

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

- **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 ($\eta = 1$)



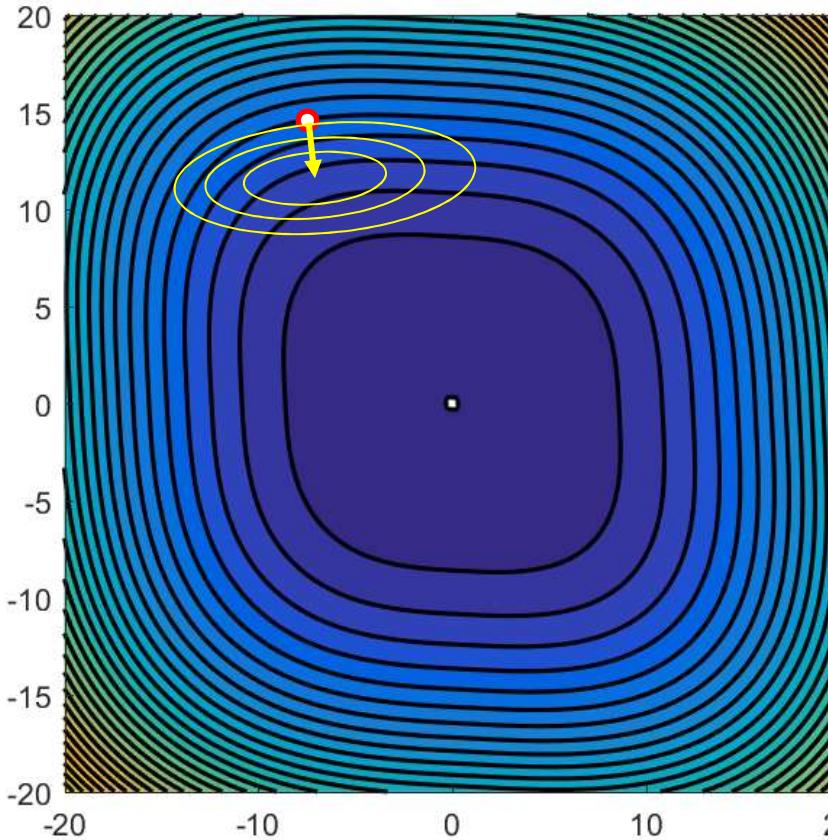
Fit a quadratic at each point and find the minimum of that quadratic

- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

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

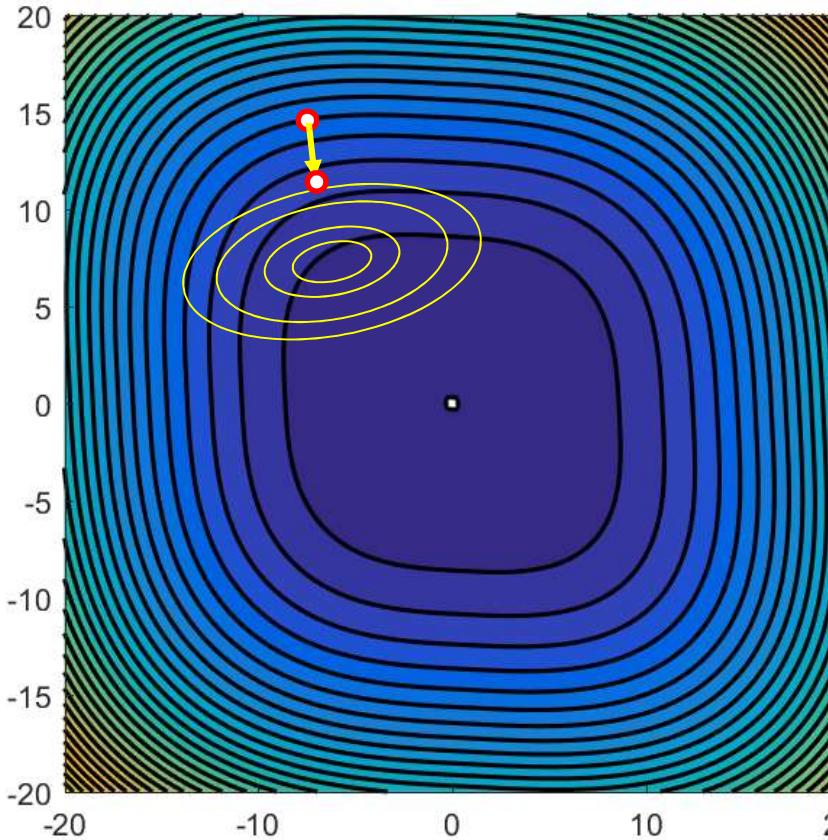


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

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

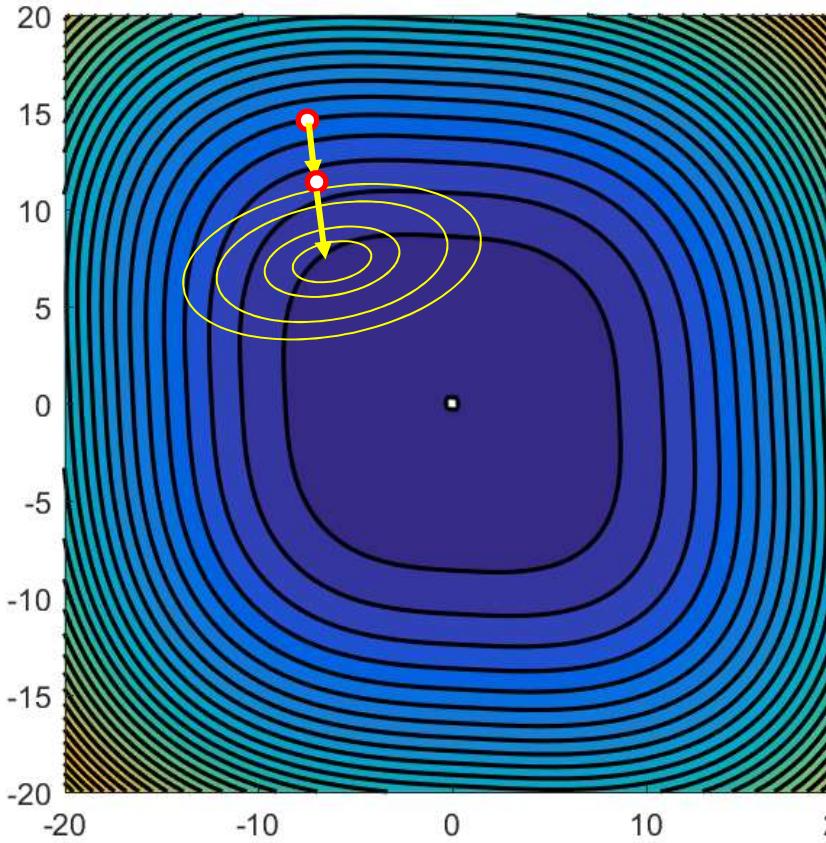


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

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

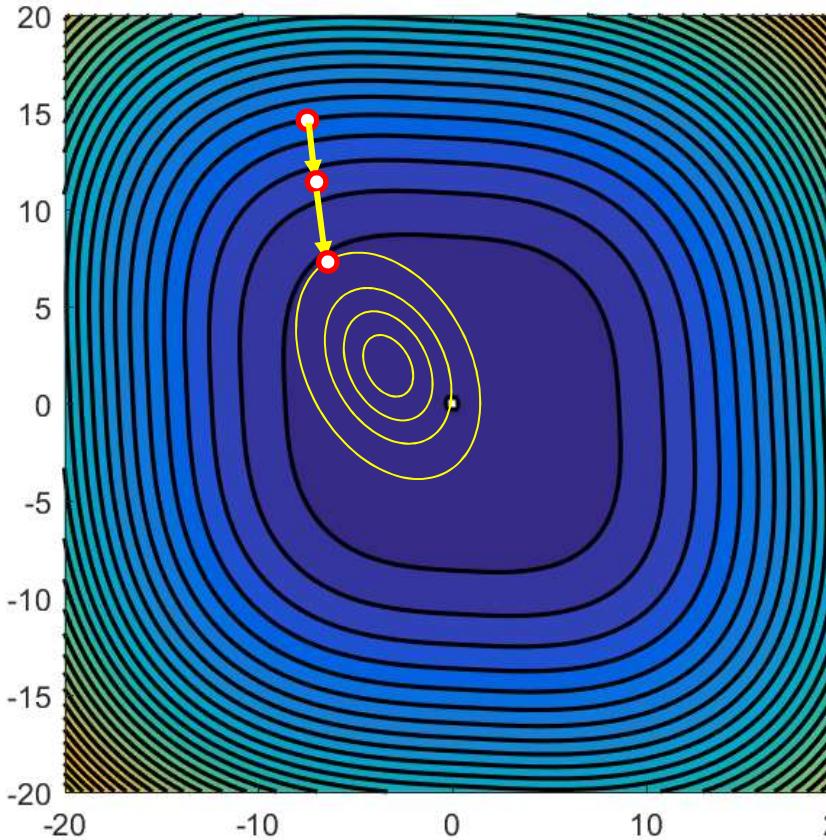


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method

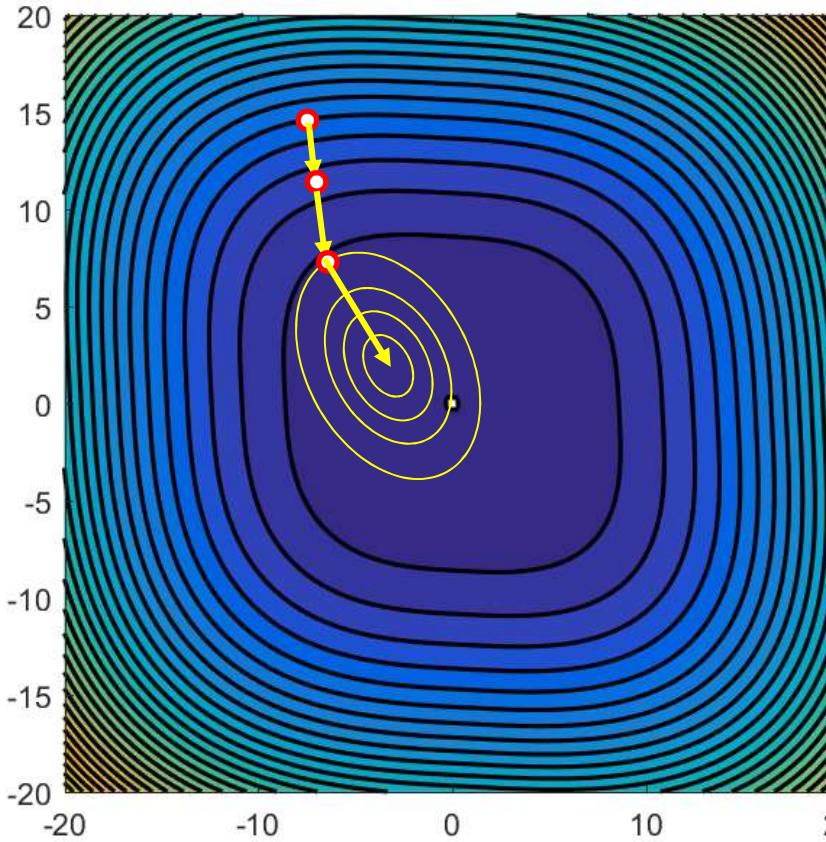


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method

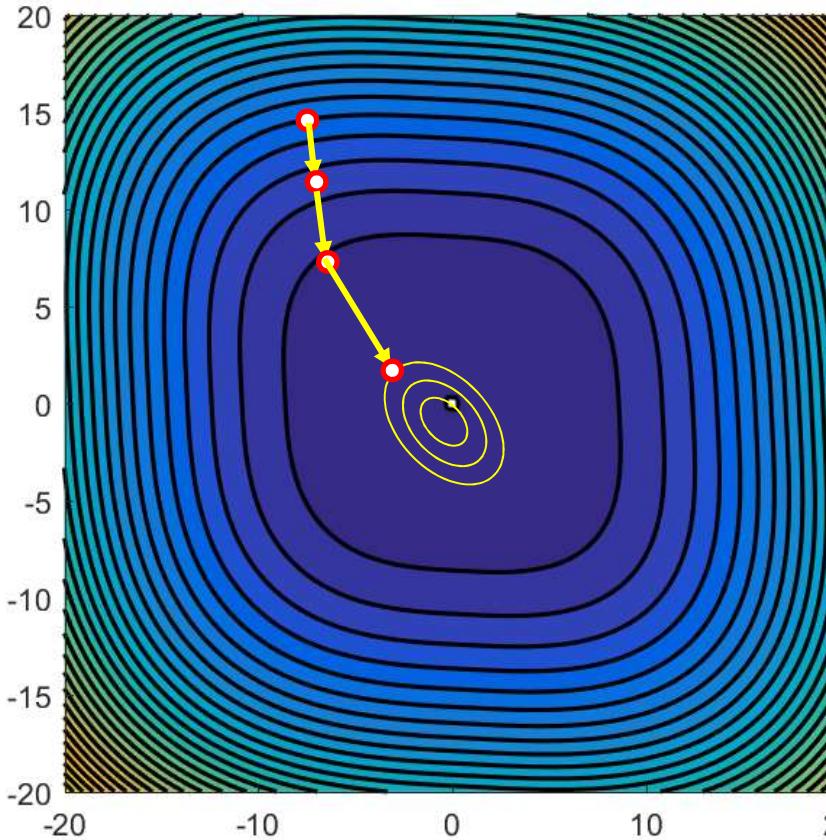


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method

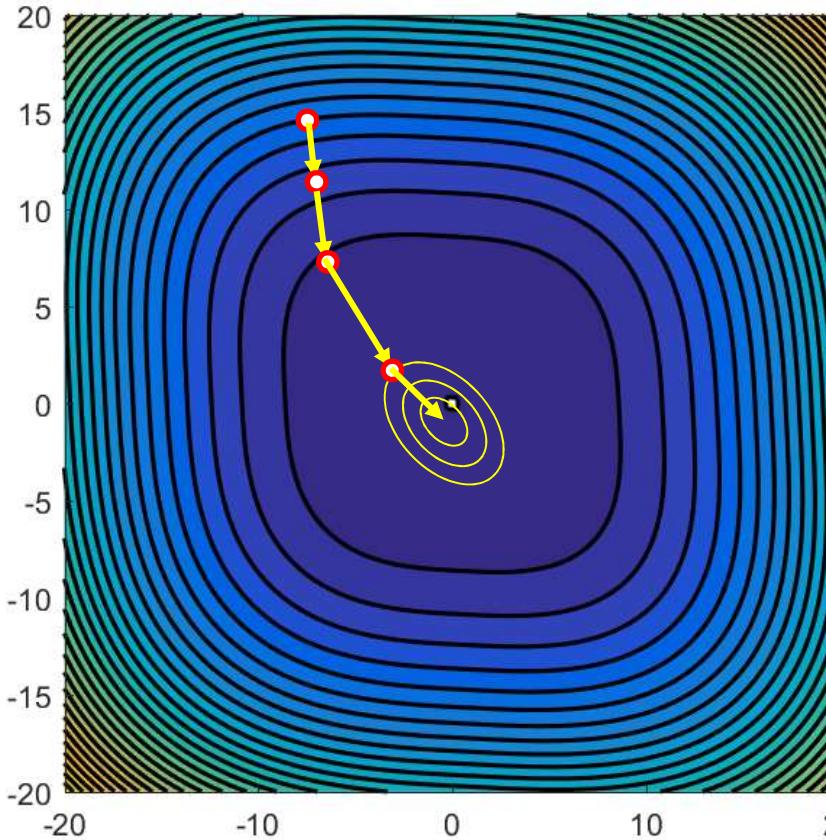


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method

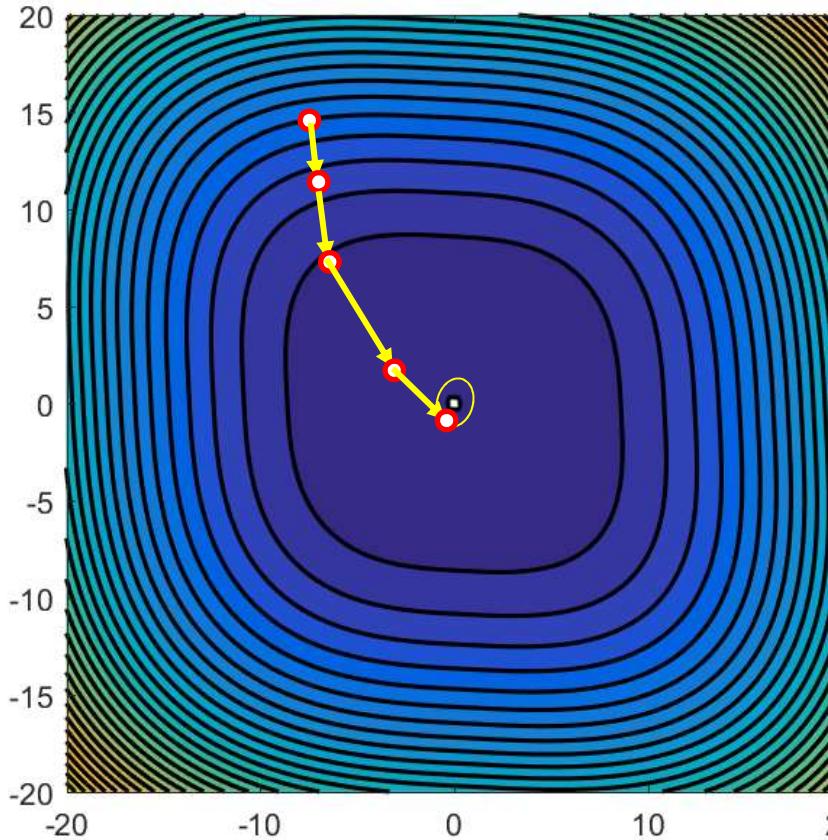


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method

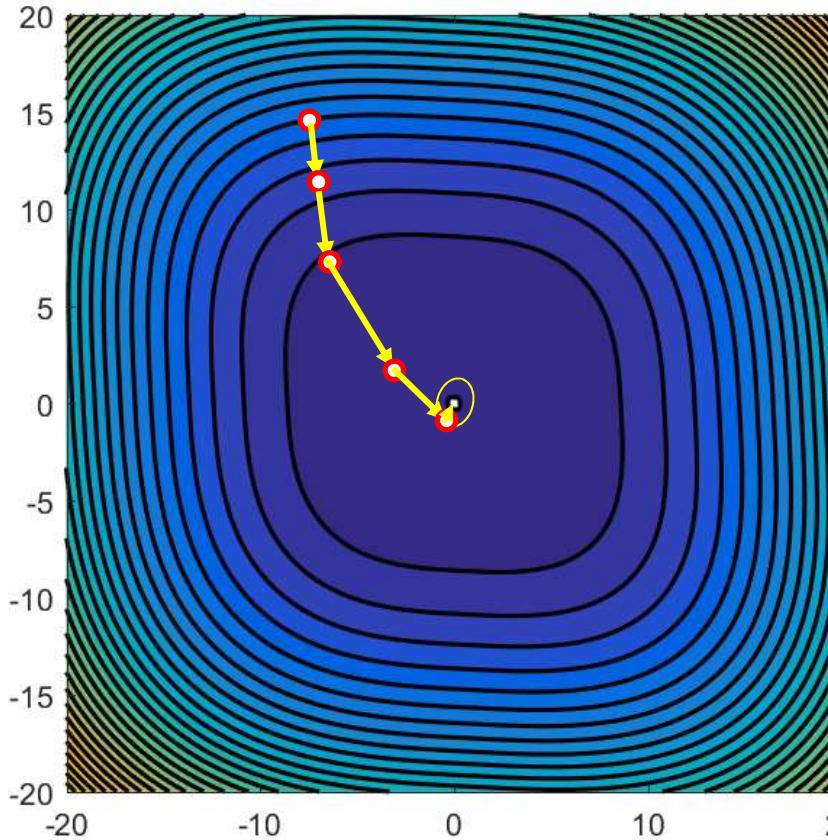


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method

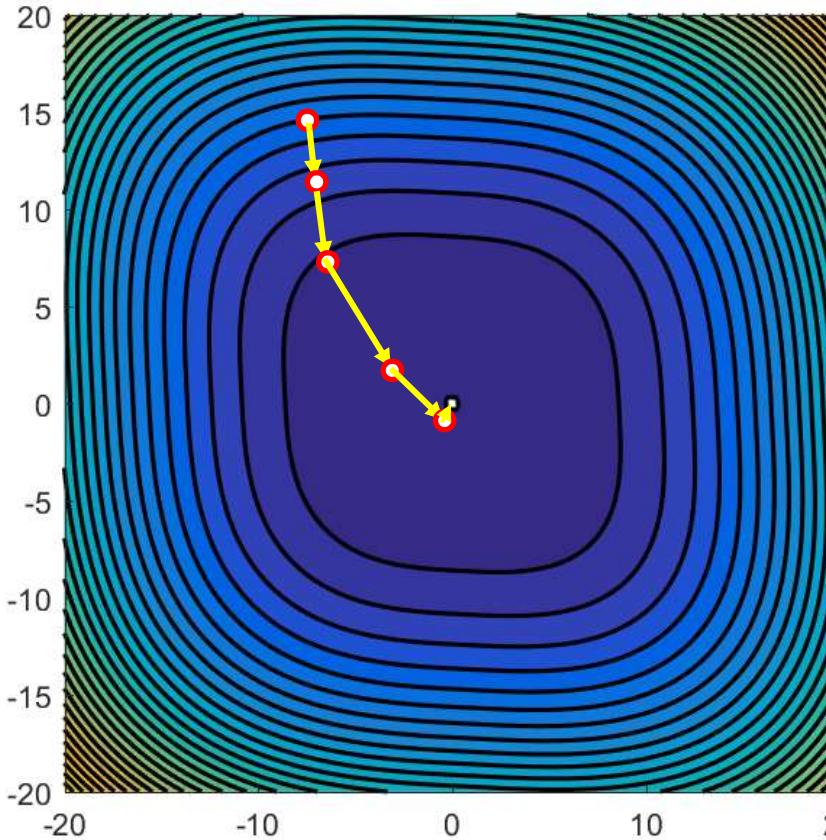


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method



- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Issues: 1. The Hessian

- Normalized update rule

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

- For complex models such as neural networks, with a very large number of parameters, the Hessian $H_E(\mathbf{w}^{(k)})$ is extremely difficult to compute
 - For a network with only 100,000 parameters, the Hessian will have 10^{10} cross-derivative terms
 - And it's 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

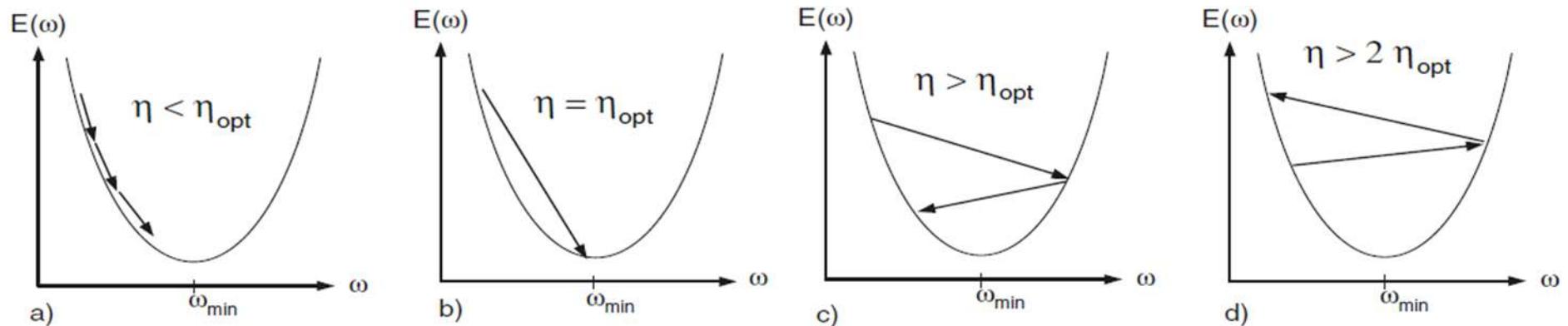


- For non-convex functions, the Hessian may not be positive semi-definite, in which case the algorithm can *diverge*
 - Goes away from, rather than towards the minimum
 - Now requires additional checks to avoid movement in directions corresponding to –ve Eigenvalues of the Hessian

Issues: 1 – contd.

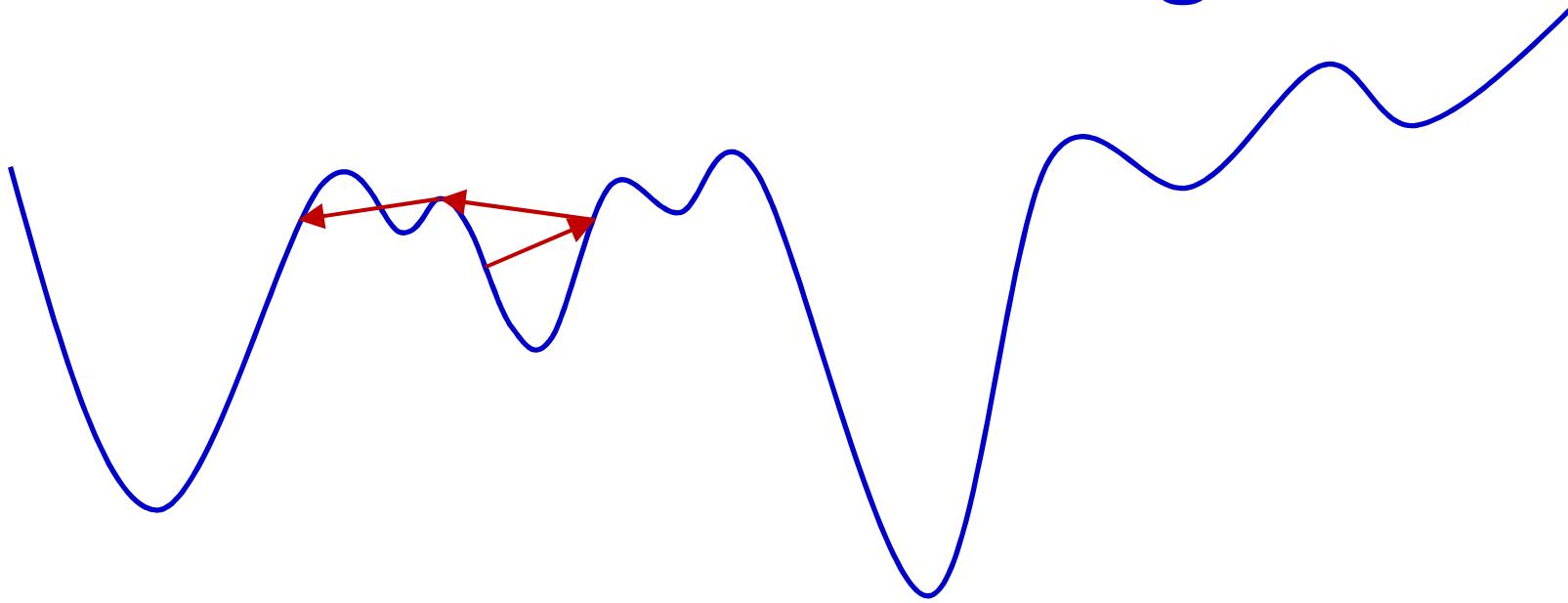
- A great many approaches have been proposed in the literature to *approximate* the Hessian in a number of ways and improve its positive definiteness
 - Boyden-Fletcher-Goldfarb-Shanno (BFGS)
 - And “low-memory” BFGS (L-BFGS)
 - Estimate Hessian from finite differences
 - Levenberg-Marquardt
 - Estimate 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



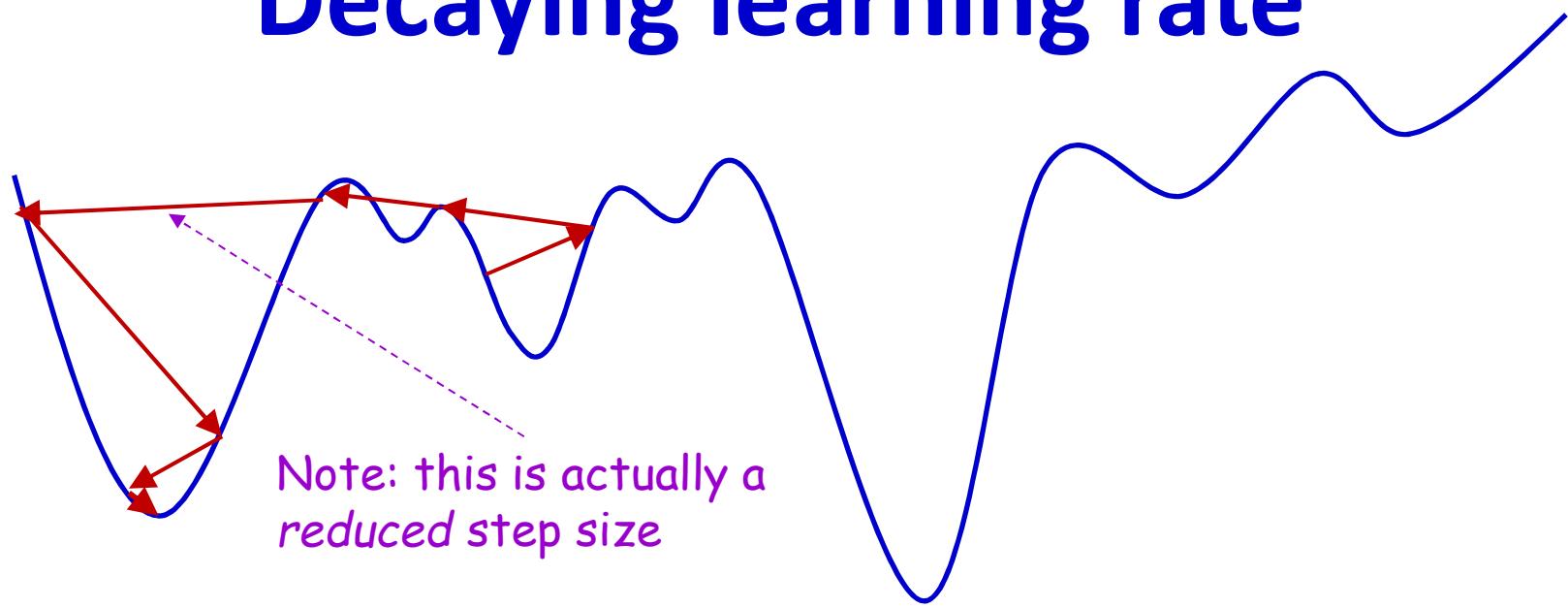
- Much of the analysis we just saw was based on trying to ensure that the step size was not so large as to cause divergence within a convex region
 - $\eta < 2\eta_{\text{opt}}$

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 error 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 tooooooo slowly
 - The optimal learning rate for one component may be too high or too low for others

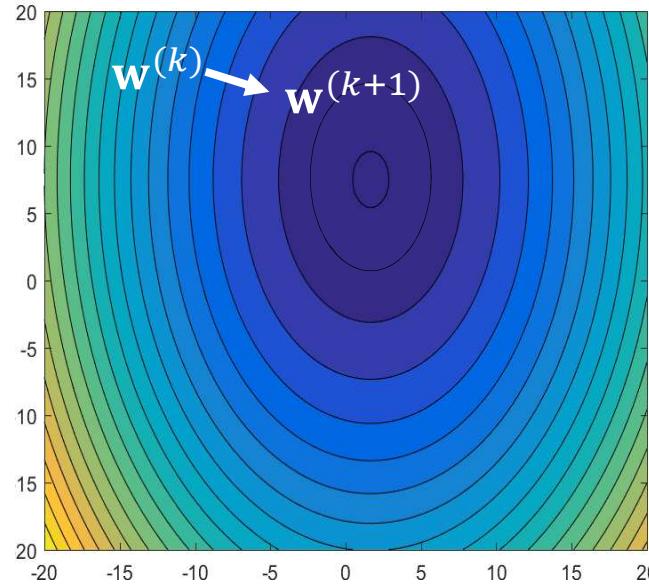
Story so far : Second-order methods

- 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 error 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



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

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

- Problems arise because of requiring a fixed step size across all dimensions
 - Because steps are “tied” to the gradient
- Let's try releasing these requirements

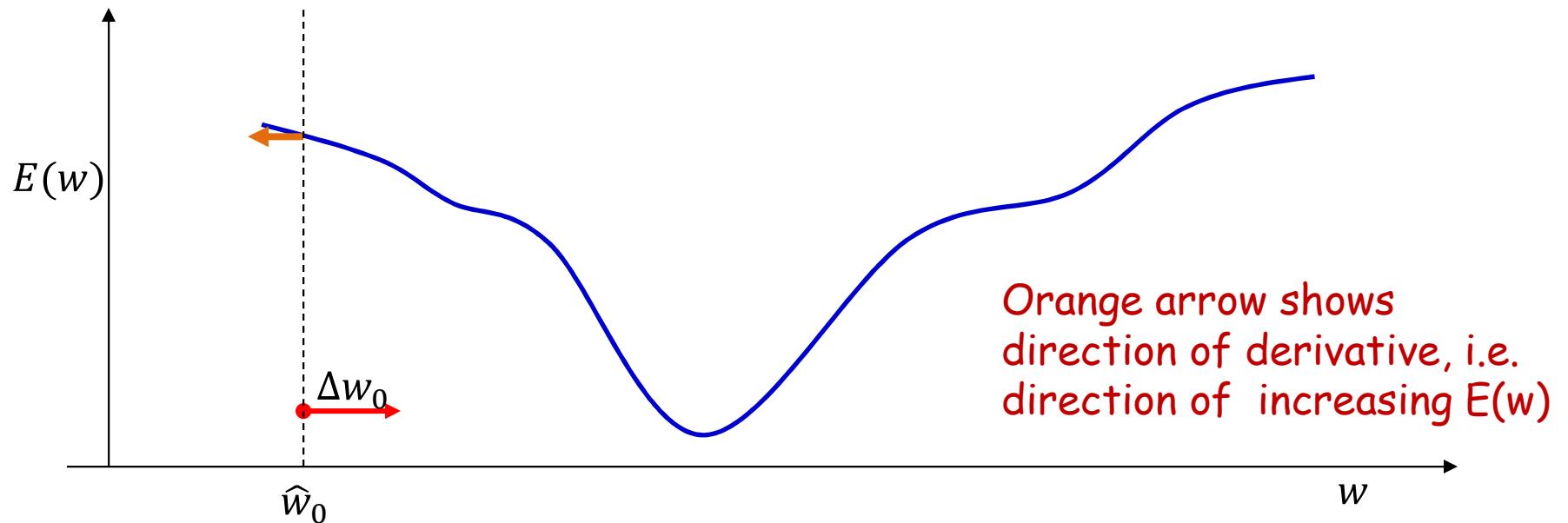
Derivative-*inspired* algorithms

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

RProp

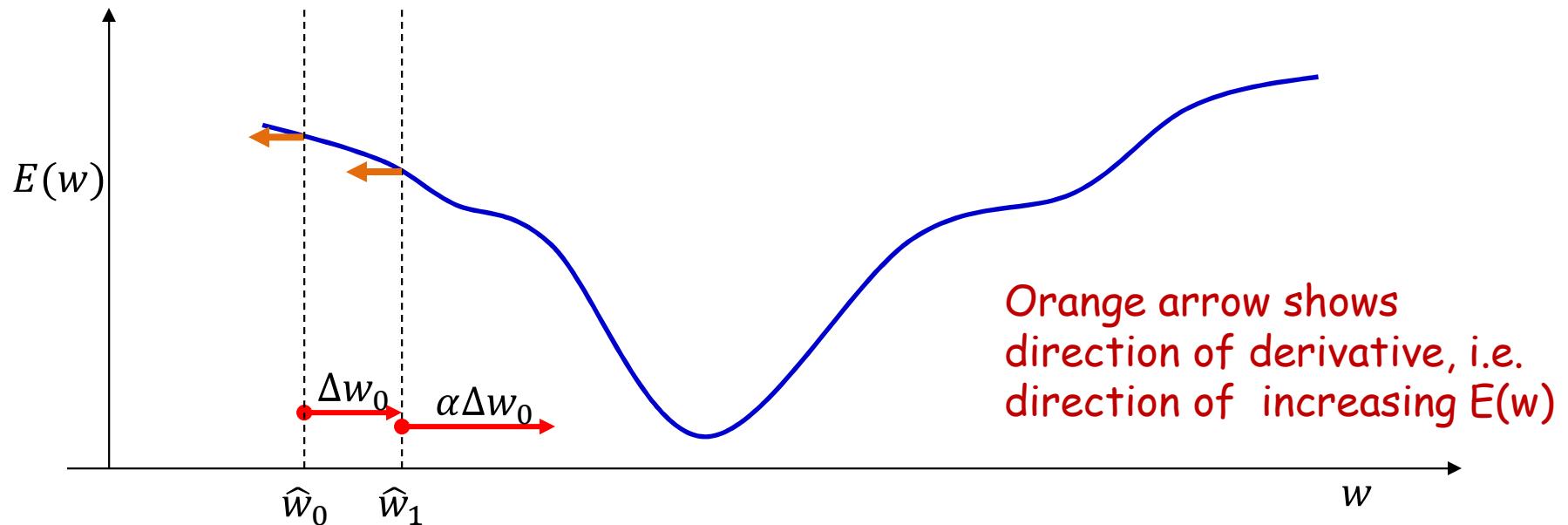
- *Resilient* propagation
- Simple algorithm, to be followed *independently* for each component
 - I.e. steps in different directions are not coupled
- At each time
 - If the derivative at the current location recommends continuing in the same direction as before (i.e. has not changed sign from earlier):
 - *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



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

Rprop

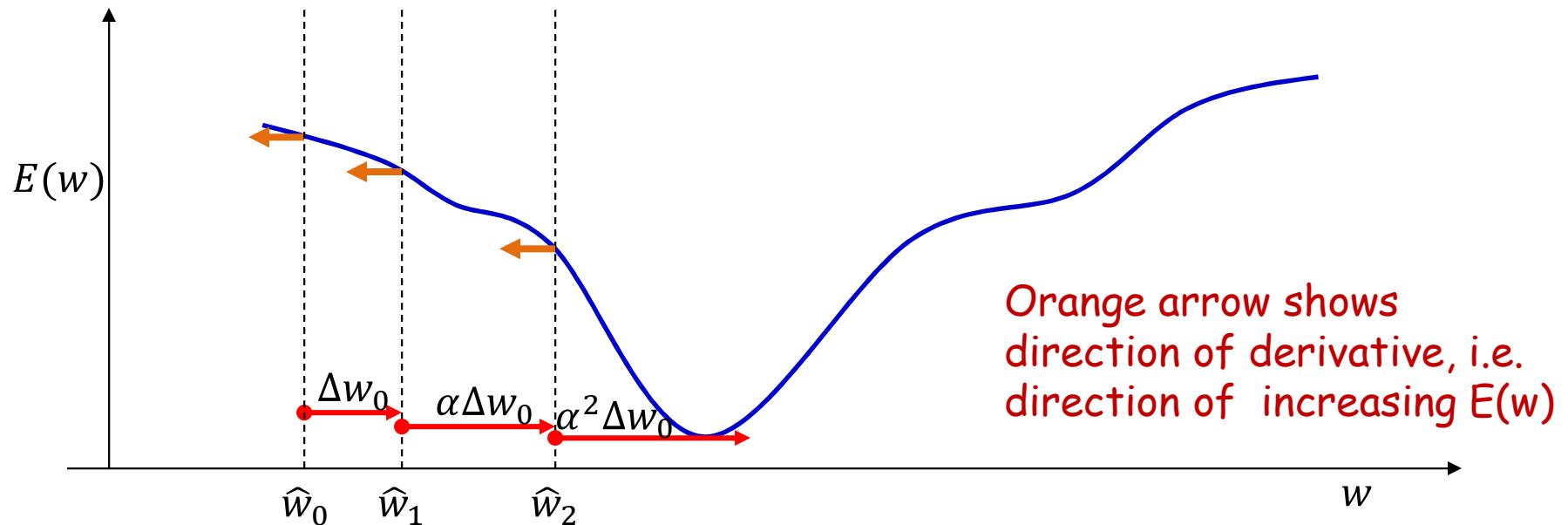


- Compute the derivative in the new location
 - If the derivative has not changed sign from the previous location, increase the step size and take a longer step

$$\alpha > 1$$

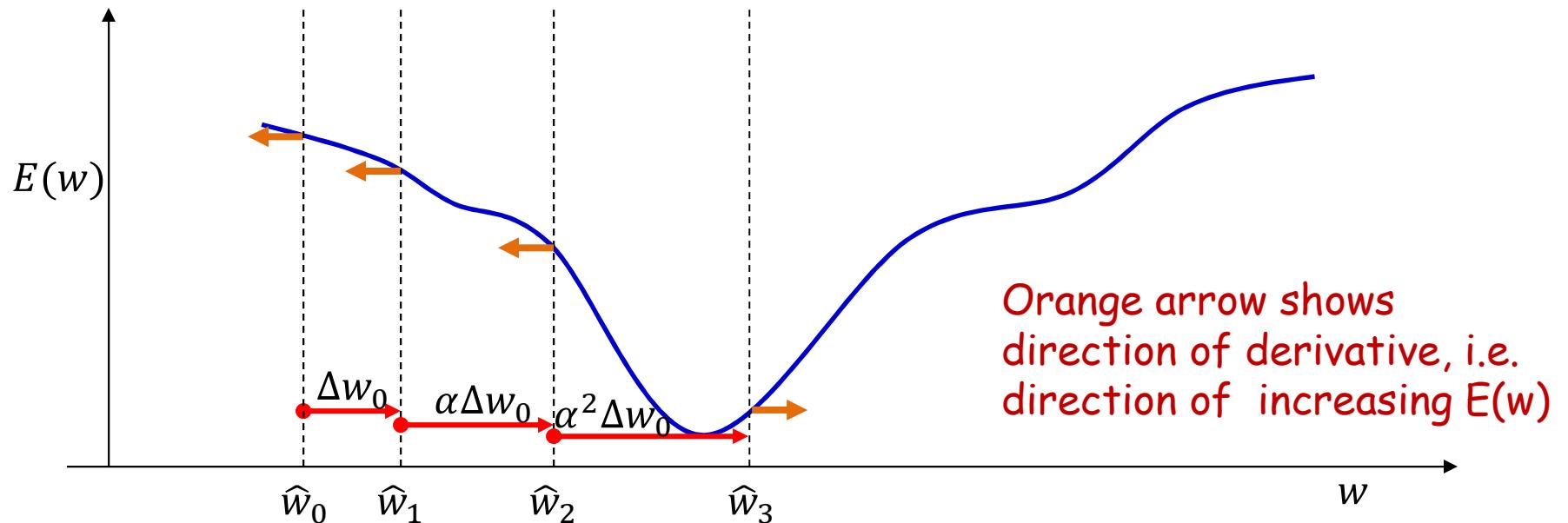
- $\Delta w = \alpha \Delta w$
- $\hat{w} = \hat{w} - \Delta w$

Rprop



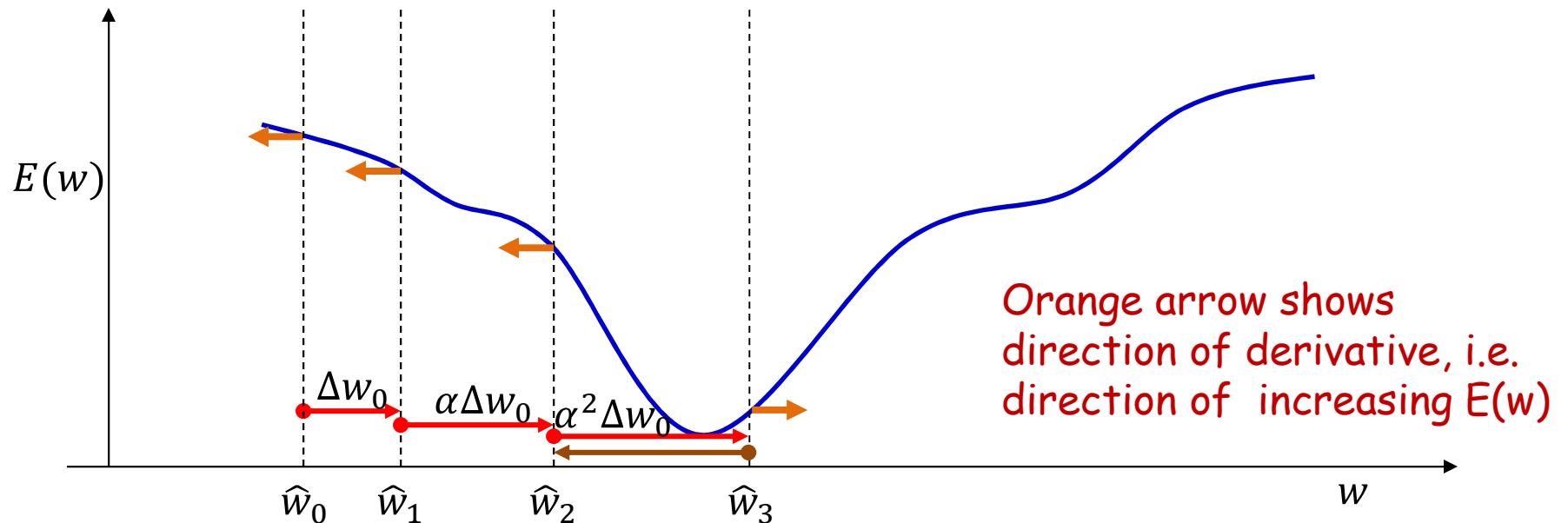
- Compute the derivative in the new location
 - If the derivative has not changed sign from the previous location, increase the step size and take a step
- $\alpha > 1$
- $\Delta w = \alpha \Delta w$
 - $\hat{w} = \hat{w} - \Delta w$

Rprop



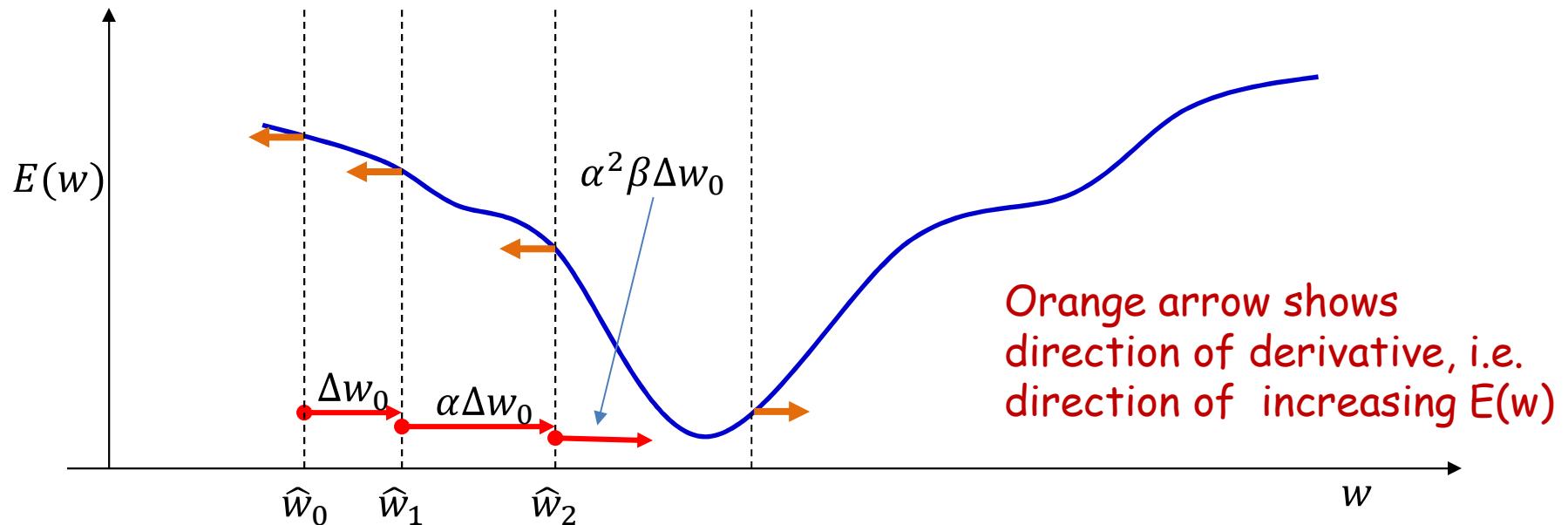
- Compute the derivative in the new location
 - If the derivative has changed sign

Rprop



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

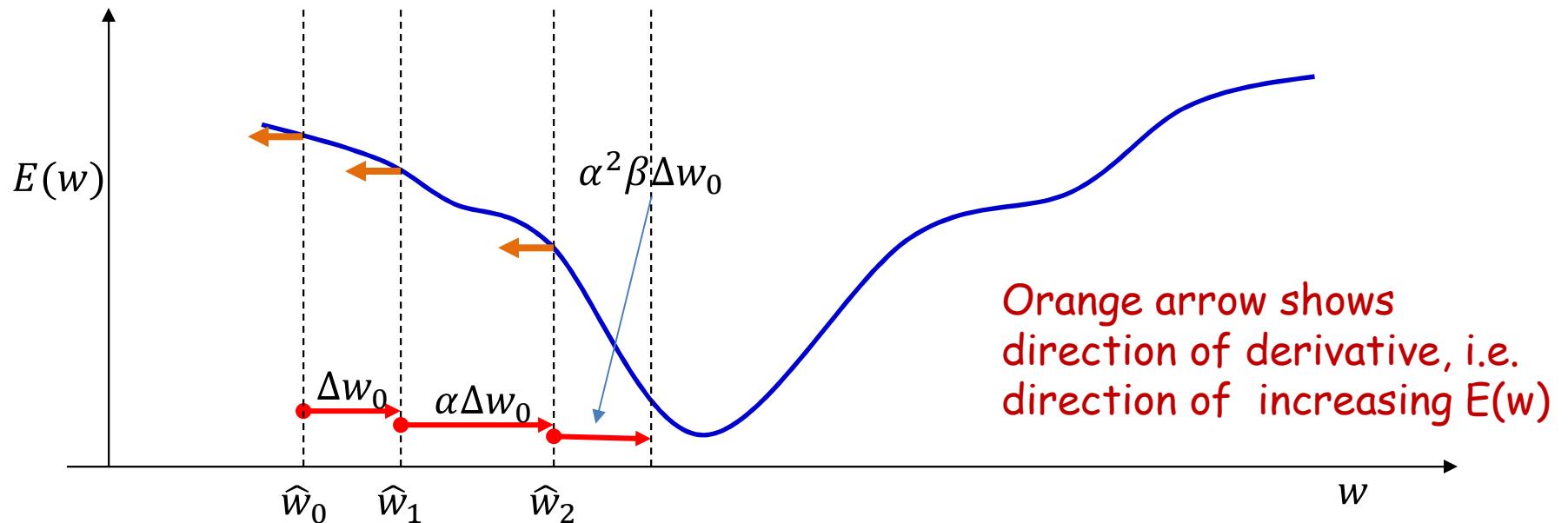
Rprop



- 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
 - $\Delta w = \beta \Delta w$

$\beta < 1$

Rprop



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

$\beta < 1$

Rprop (simplified)

- Set $\alpha = 1.2, \beta = 0.5$
- For each layer l , for each i, j :

- Initialize $w_{l,i,j}, \Delta w_{l,i,j} > 0$,
- $prevD(l, i, j) = \frac{dErr(w_{l,i,j})}{dw_{l,i,j}}$
- $\Delta w_{l,i,j} = \text{sign}(prevD(l, i, j))\Delta w_{l,i,j}$
- 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 $\text{sign}(prevD(l, i, j)) == \text{sign}(D(l, i, j))$:
 - $\Delta w_{l,i,j} = \min(\alpha\Delta w_{l,i,j}, \Delta_{max})$
 - $prevD(l, i, j) = D(l, i, j)$
 - 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 $w_{l,i,j}$, $\Delta w_{l,i,j} > 0$,
- $prevD(l, i, j) = \frac{d\text{Err}(w_{l,i,j})}{dw_{l,i,j}}$
- $\Delta w_{l,i,j} = \text{sign}(prevD(l, i, j))\Delta w_{l,i,j}$
- While not converged:
 - $w_{l,i,j} = w_{l,i,j} - \Delta w_{l,i,j}$
 - $D(l, i, j) = \frac{d\text{Err}(w_{l,i,j})}{dw_{l,i,j}}$
 - If $\text{sign}(prevD(l, i, j)) == \text{sign}(D(l, i, j))$:
 - $\Delta w_{l,i,j} = \alpha \Delta w_{l,i,j}$
 - $prevD(l, i, j) = D(l, i, j)$
 - else:
 - $w_{l,i,j} = w_{l,i,j} + \Delta w_{l,i,j}$
 - $\Delta w_{l,i,j} = \beta \Delta w_{l,i,j}$

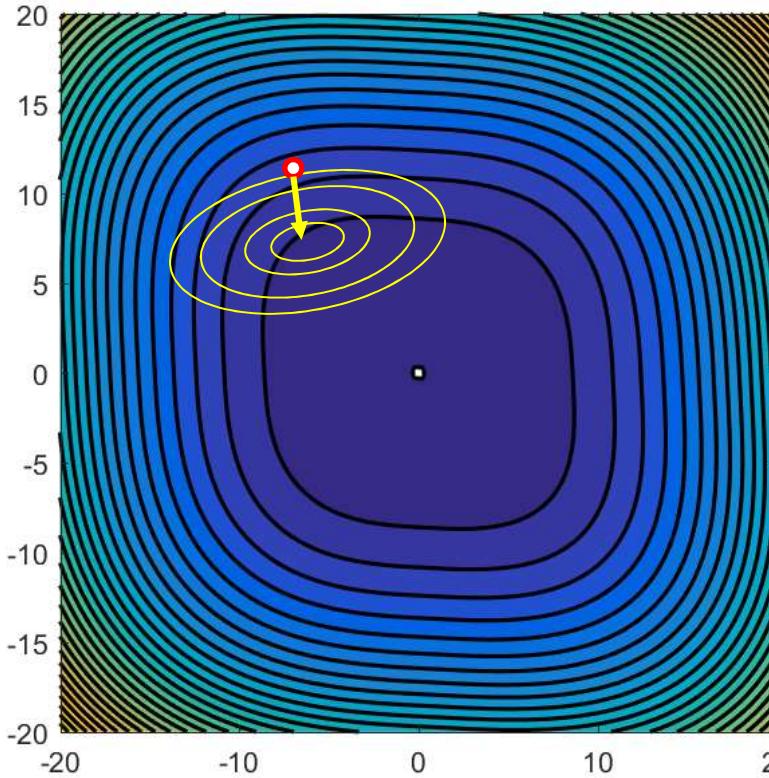
Obtained via backprop

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

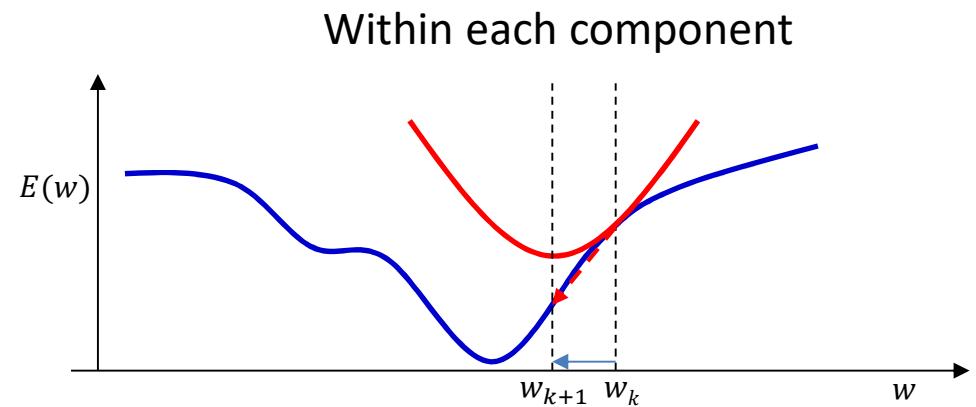
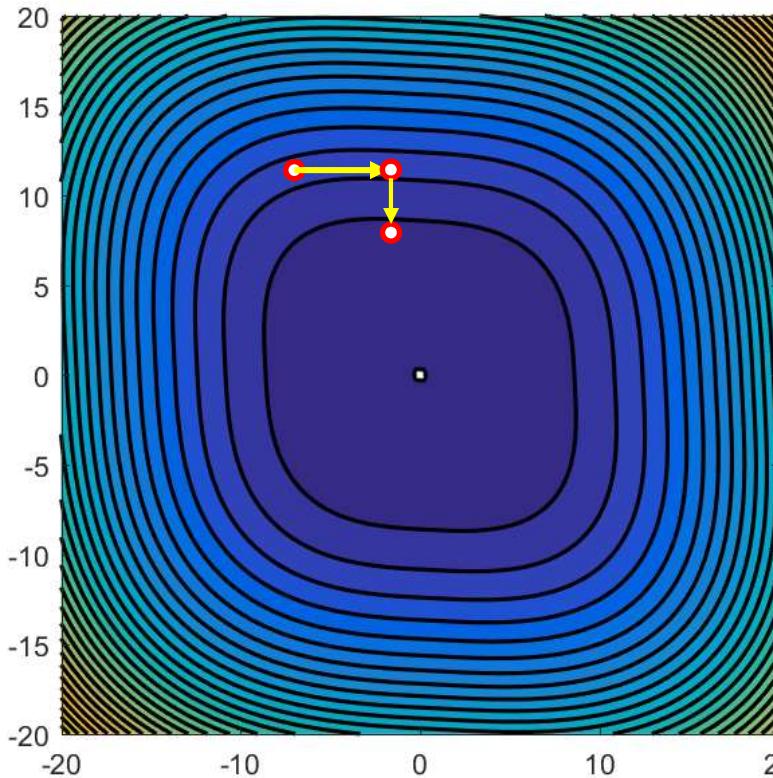


- Quickprop employs the Newton updates with two modifications

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

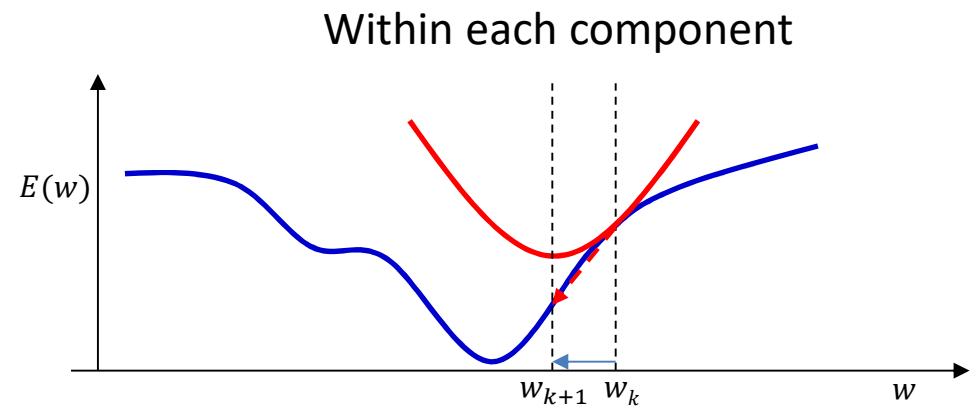
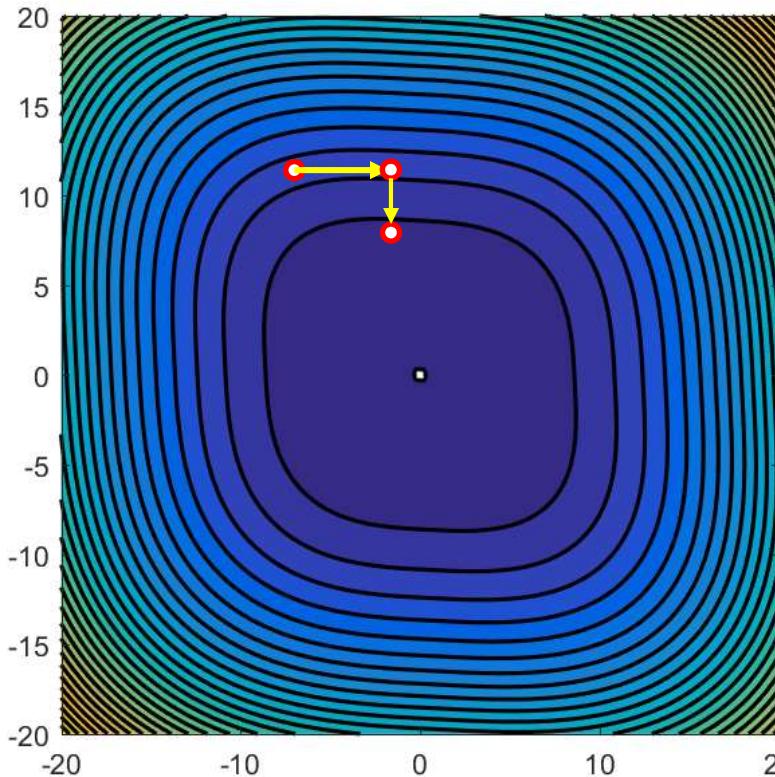
- But with two modifications

QuickProp: Modification 1



- It treats each dimension independently
 - For $i = 1:N$
- $$w_i^{k+1} = w_i^k - E''(w_i^k | w_j^k, j \neq i)^{-1} E'(w_i^k | w_j^k, j \neq i)$$
- This eliminates the need to compute and invert expensive Hessians

QuickProp: Modification 2



- It approximates the second derivative through finite differences
 - For $i = 1:N$
- $$w_i^{k+1} = w_i^k - D(w_i^k, w_i^{k-1})^{-1} E'(w_i^k | w_j^k, j \neq i)$$
- This eliminates the need to compute expensive double derivatives

QuickProp

$$w^{(k+1)} = w^{(k)} - \left(\frac{E'(w^{(k)}) - E'(w^{(k-1)})}{\Delta w^{(k-1)}} \right)^{-1} E'(w^{(k)})$$

Finite-difference approximation to double derivative
obtained assuming a quadratic $E()$

- 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:

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

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

QuickProp

$$w^{(k+1)} = w^{(k)} - \left(\frac{E'(w^{(k)}) - E'(w^{(k-1)})}{\Delta w^{(k-1)}} \right)^{-1} E'(w^{(k)})$$

Finite-difference approximation to double derivative
obtained assuming a quadratic $E()$

- 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:

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

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

Computed using
backprop

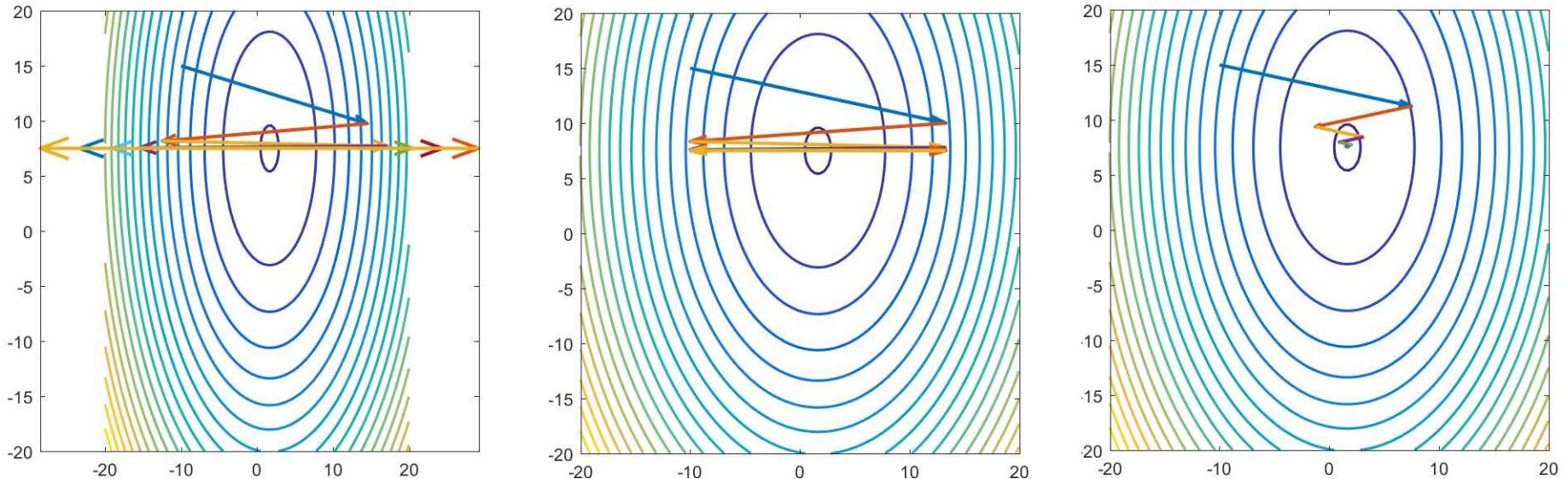
Quickprop

- 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

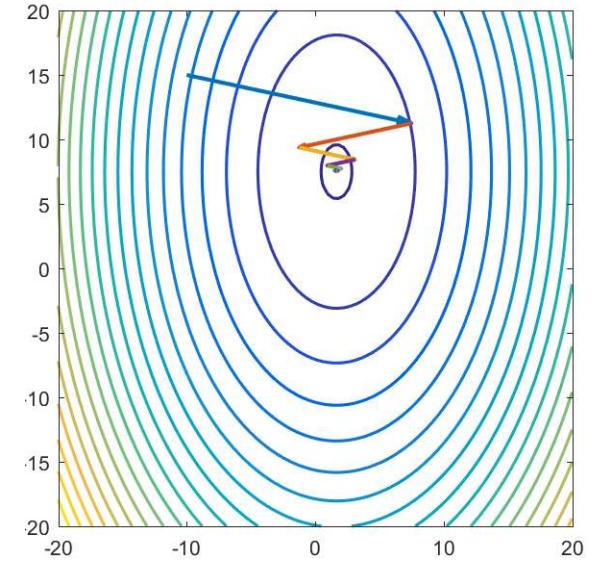
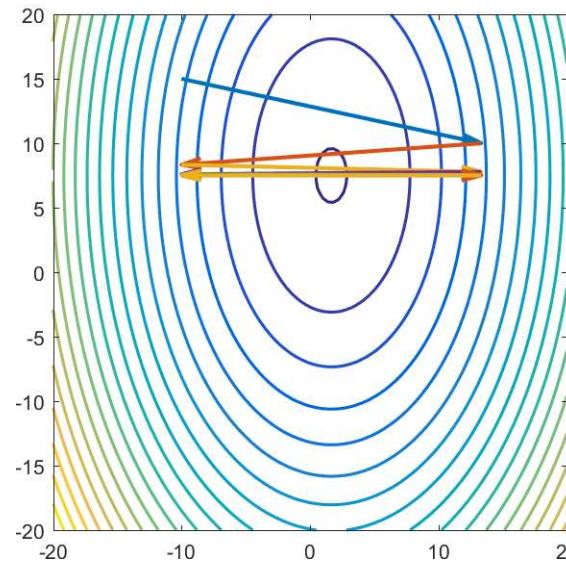
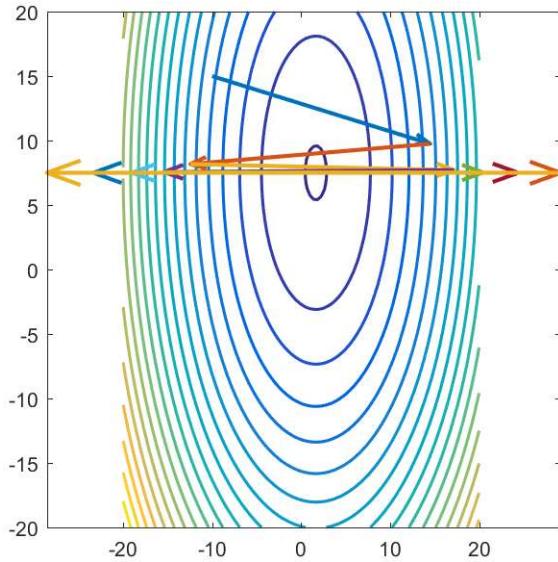
- Gradient descent can miss obvious answers
 - And this may be a *good* thing
- Vanilla gradient descent may be too slow or unstable due to the differences between the dimensions
- Second order methods can normalize the variation across dimensions, but are complex
- Adaptive or decaying learning rates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

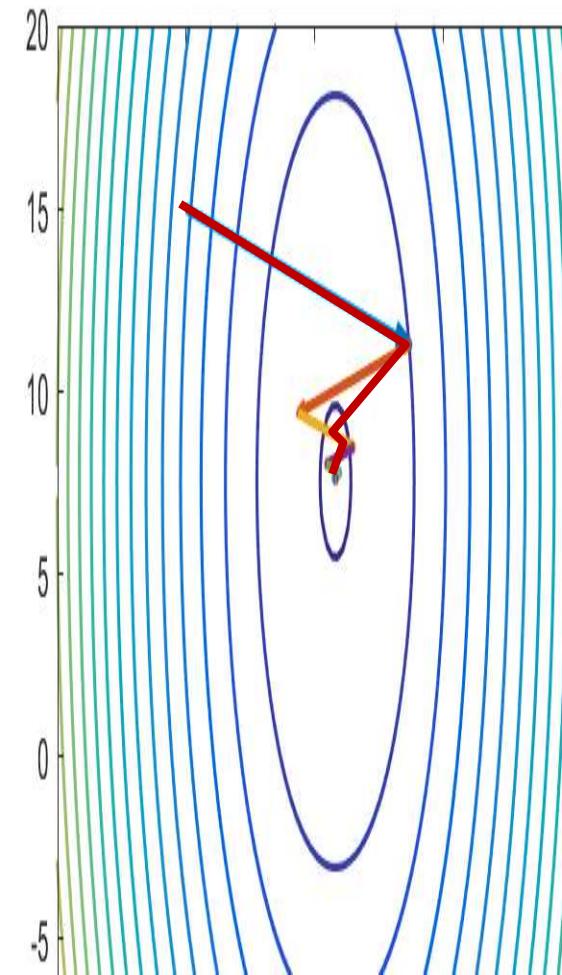
A closer look at the convergence problem



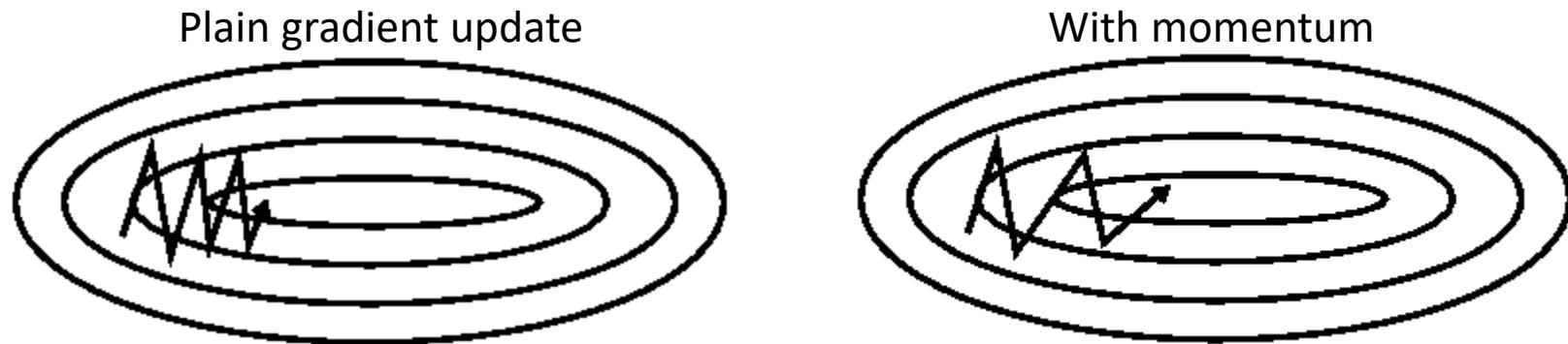
- With dimension-independent learning rates, the solution will converge smoothly in some directions, but oscillate or diverge in others
- Proposal:**
 - Keep track of oscillations
 - Emphasize steps in directions that converge smoothly
 - Shrink steps in directions that bounce around..

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

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err(W^{(k-1)})$$

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

- 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

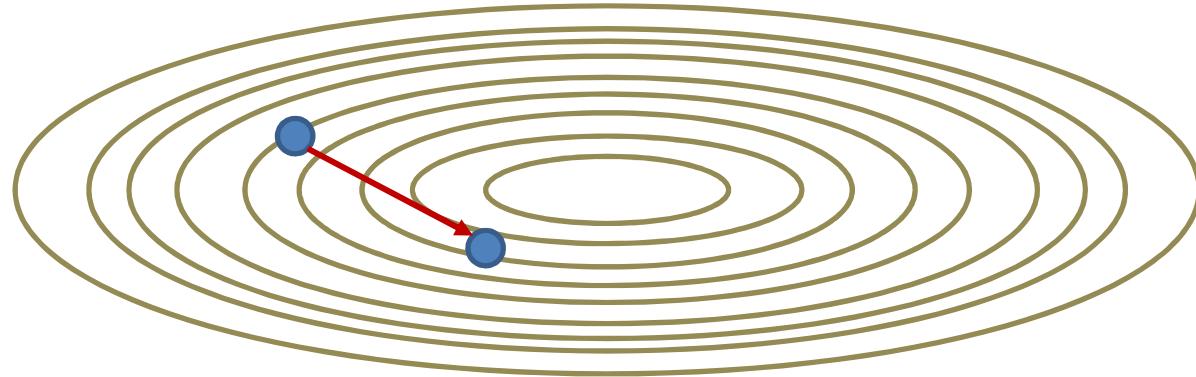
Training by gradient descent

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

Training with momentum

- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_K$
- Do:
 - For all layers k , initialize $\nabla_{\mathbf{W}_k} Err = 0, \Delta W_k = 0$
 - For all $t = 1:T$
 - For every layer k :
 - Compute gradient $\nabla_{\mathbf{W}_k} \text{Div}(Y_t, d_t)$
 - $\nabla_{\mathbf{W}_k} Err += \frac{1}{T} \nabla_{\mathbf{W}_k} \text{Div}(Y_t, d_t)$
 - For every layer k
$$\Delta W_k = \beta \Delta W_k - \eta \nabla_{\mathbf{W}_k} Err$$
$$W_k = W_k + \Delta W_k$$
 - Until Err has converged

Momentum Update

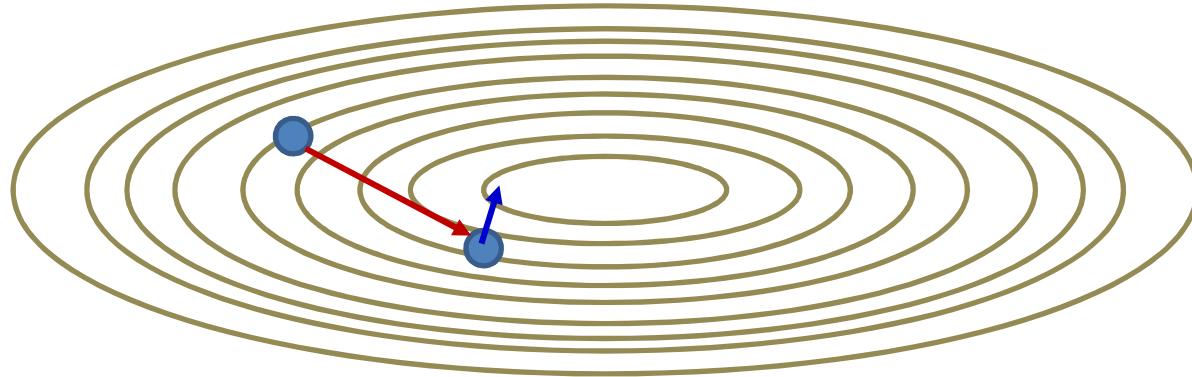


- The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err(W^{(k-1)})$$

- At any iteration, to compute the current step:

Momentum Update

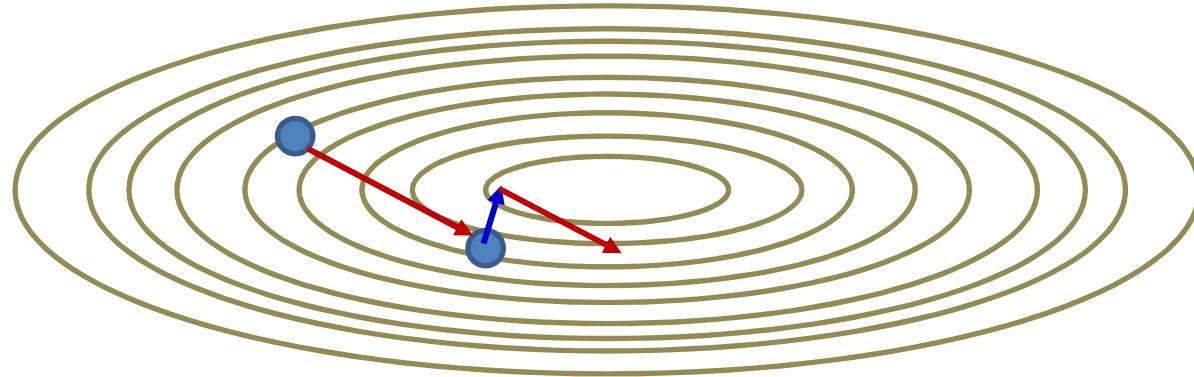


- The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err(W^{(k-1)})$$

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

Momentum Update

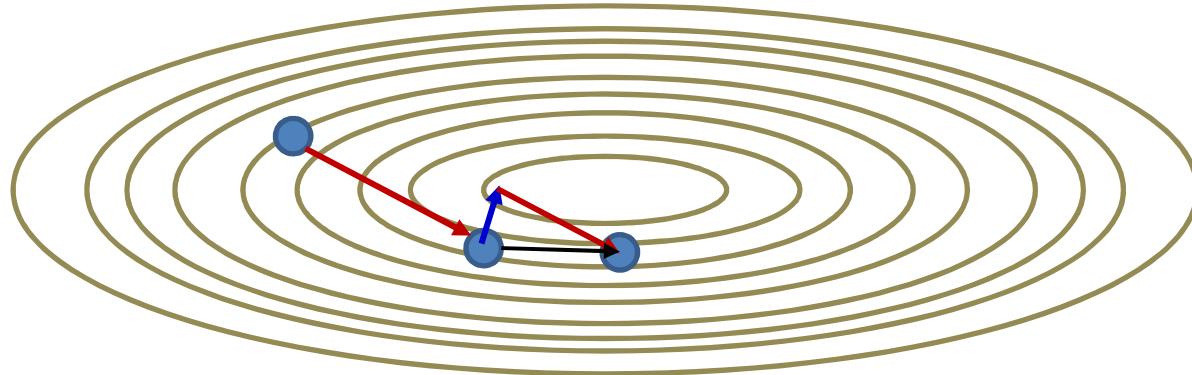


- The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err(W^{(k-1)})$$

- 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

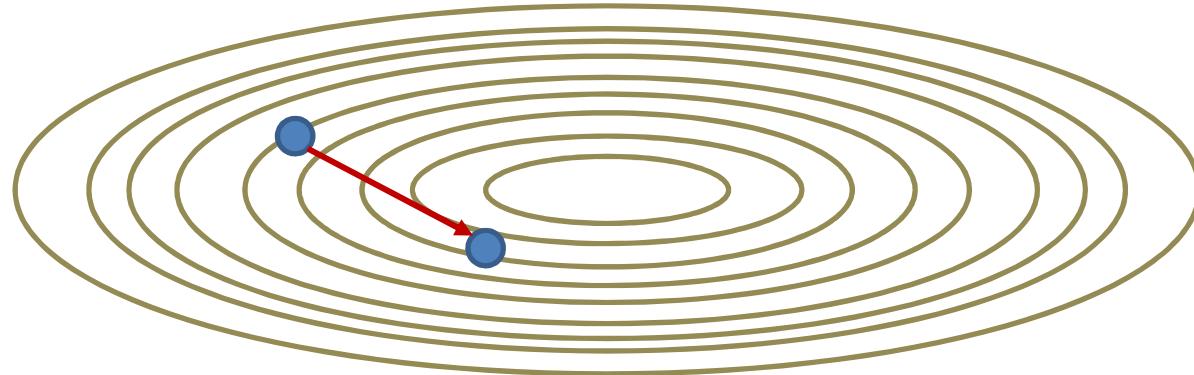
$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err(W^{(k-1)})$$

- 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

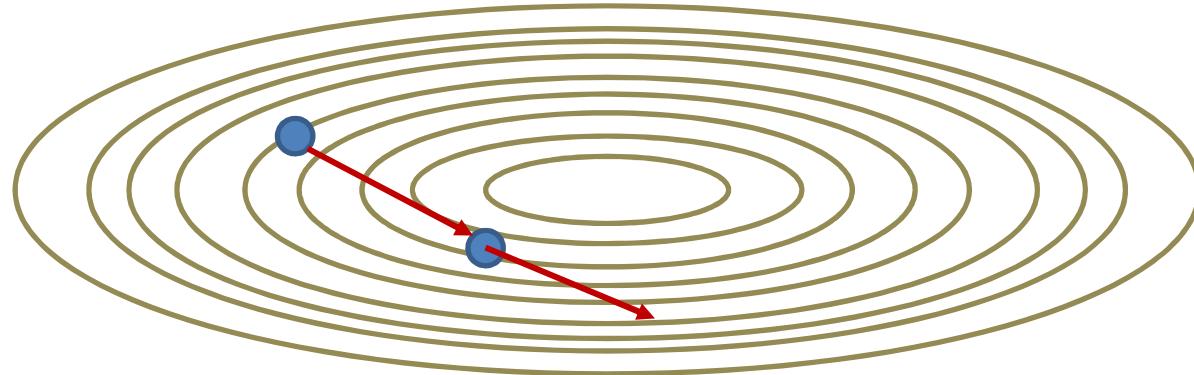
- Takes a step along the past running average *after* walking along the gradient
- The procedure can be made more optimal by reversing the order of operations..

Nestorov's Accelerated Gradient



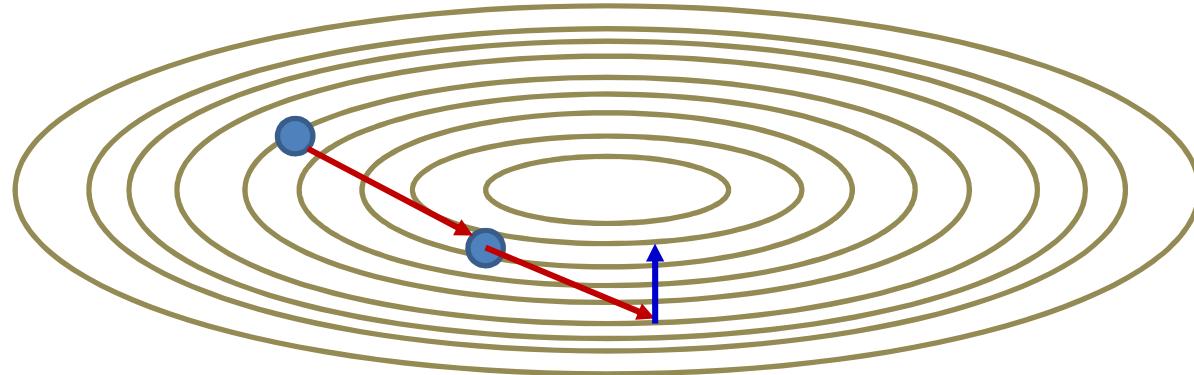
- Change the order of operations
- At any iteration, to compute the current step:

Nestorov's Accelerated Gradient



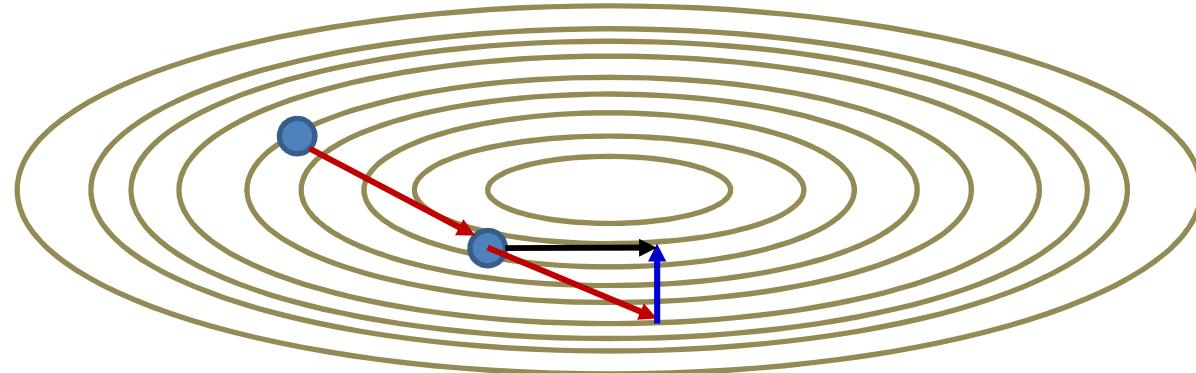
- Change the order of operations
- At any iteration, to compute the current step:
 - First extend the previous step

Nestorov's Accelerated Gradient



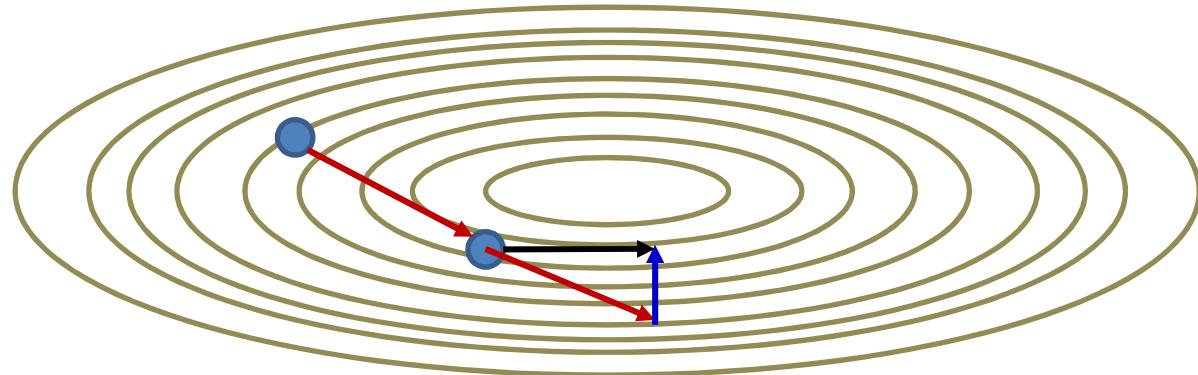
- Change the order of operations
- At any iteration, to compute the current step:
 - First extend the previous step
 - Then compute the gradient step at the resultant position

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

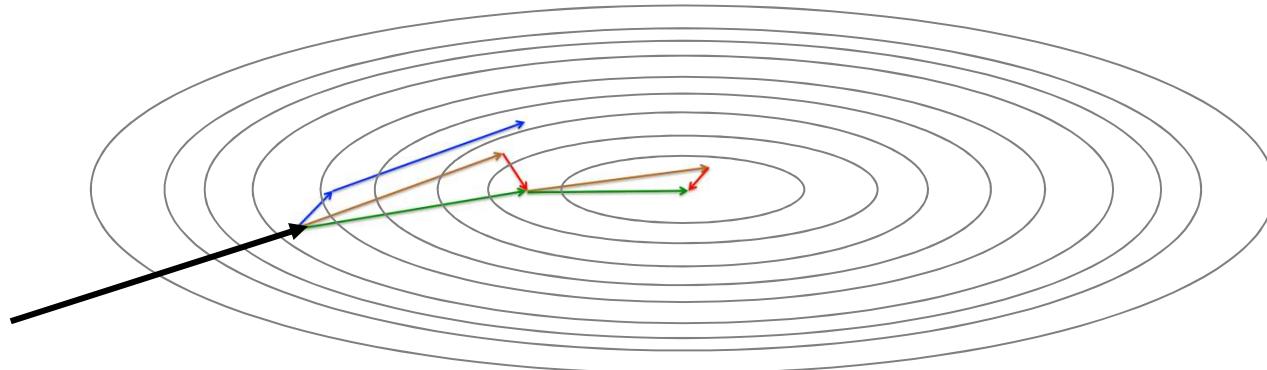
Nestorov's Accelerated Gradient



- Nestorov's method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err(W^{(k-1)} + \beta \Delta W^{(k-1)})$$
$$W^{(k)} = W^{(k-1)} + \Delta W^{(k)}$$

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 $\nabla_{W_k} Err = 0, \Delta W_k = 0$
 - For every layer k
$$W_k = W_k + \beta \Delta W_k$$
 - For all $t = 1:T$
 - For every layer k :
 - Compute gradient $\nabla_{W_k} \text{Div}(Y_t, d_t)$
 - $\nabla_{W_k} Err += \frac{1}{T} \nabla_{W_k} \text{Div}(Y_t, d_t)$
 - For every layer k
$$W_k = W_k - \eta \nabla_{W_k} Err$$
$$\Delta W_k = \beta \Delta W_k - \eta \nabla_{W_k} Err$$
 - Until Err 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