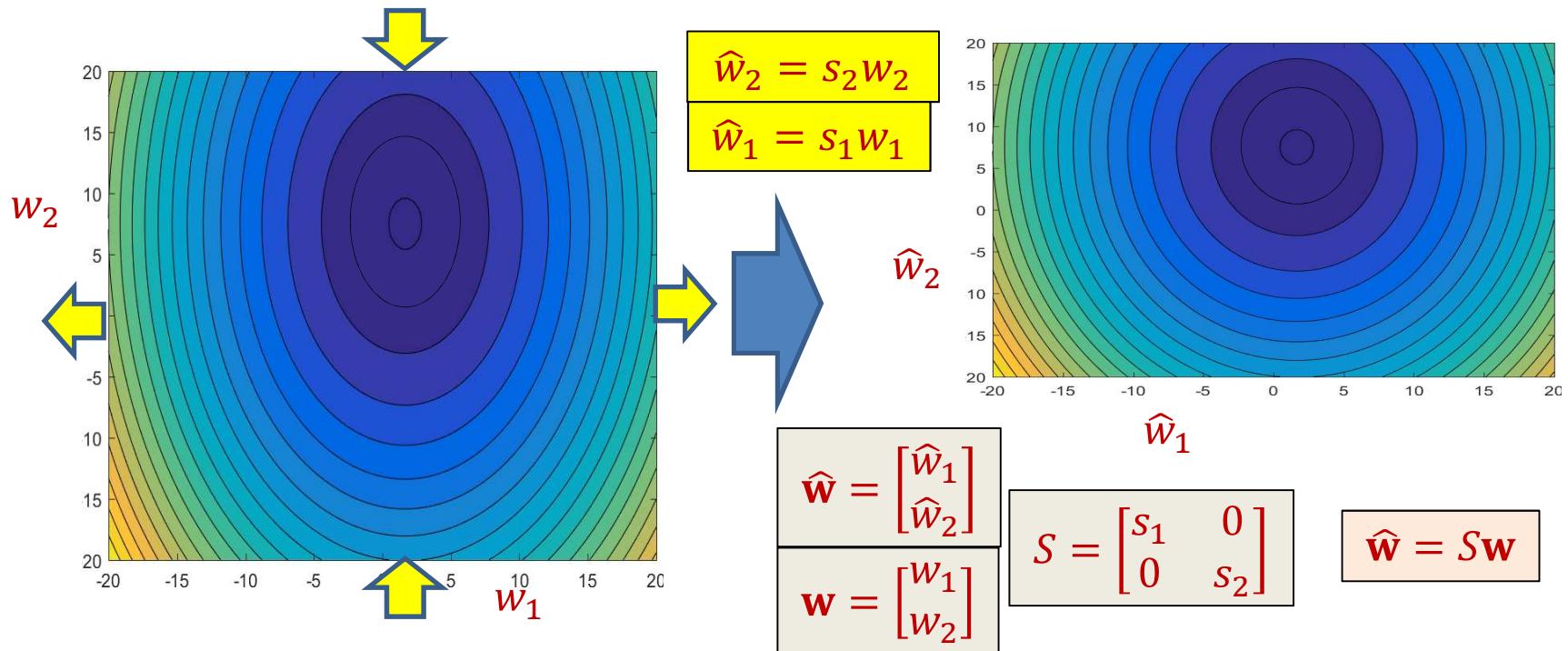


Convergence

- 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

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

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

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

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

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

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

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

- 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

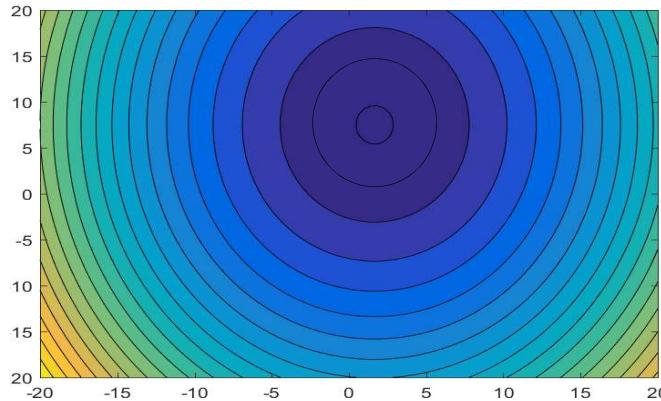
- For *any* positive definite \mathbf{A} , we can write

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

- 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

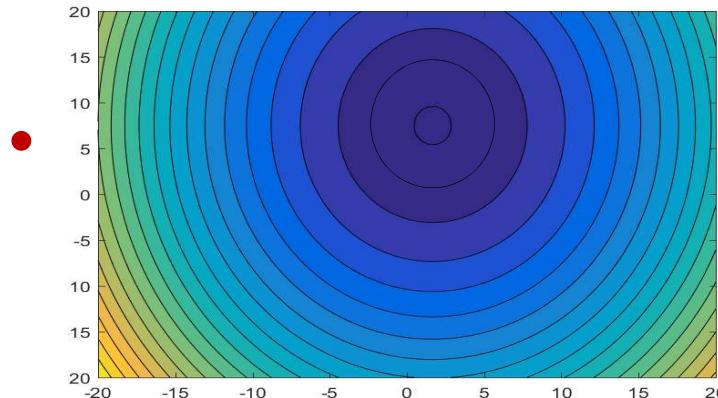


$$E = \frac{1}{2} \hat{\mathbf{w}}^T \hat{\mathbf{w}} + \hat{\mathbf{b}}^T \hat{\mathbf{w}} + c$$

- 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

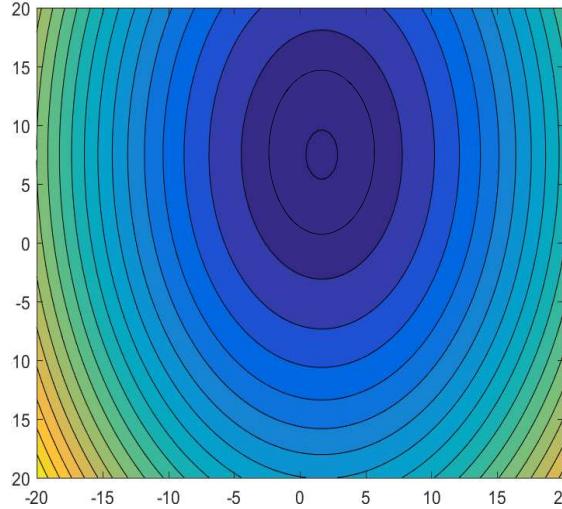


$$E = \frac{1}{2} \hat{\mathbf{w}}^T \hat{\mathbf{w}} + \hat{\mathbf{b}}^T \hat{\mathbf{w}} + c$$

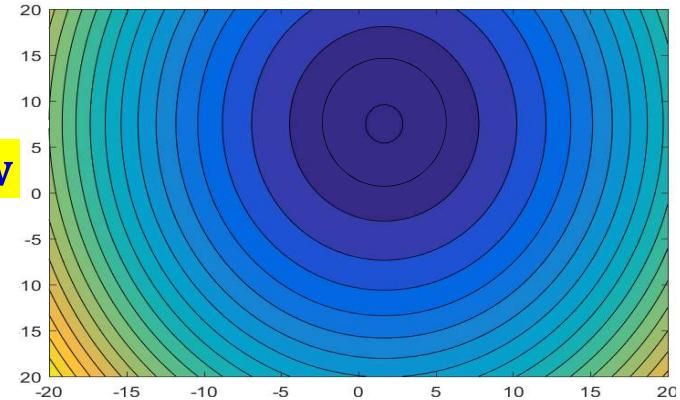
- Gradient descent rule:
 - $\hat{\mathbf{w}}^{(k+1)} = \hat{\mathbf{w}}^{(k)} - \eta \nabla_{\hat{\mathbf{w}}} E(\hat{\mathbf{w}}^{(k)})^T$
 - Learning rate is now independent of direction
- Using $\hat{\mathbf{w}} = \mathbf{A}^{0.5} \mathbf{w}$, and $\nabla_{\hat{\mathbf{w}}} E(\hat{\mathbf{w}})^T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

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

Modified update rule



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



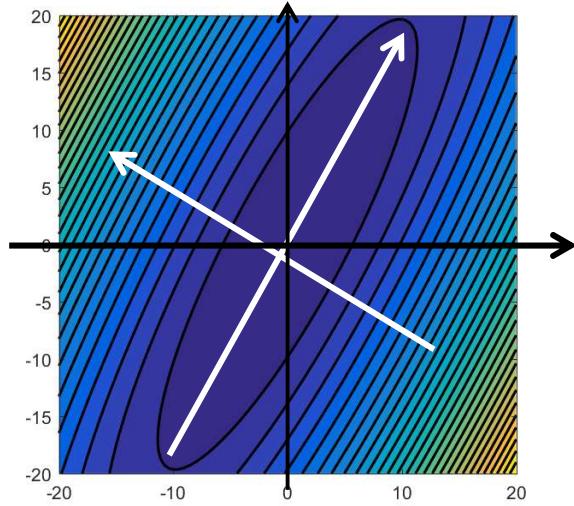
$$E = \frac{1}{2} \mathbf{w}^T \mathbf{A} \mathbf{w} + \mathbf{b}^T \mathbf{w} + c$$

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

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

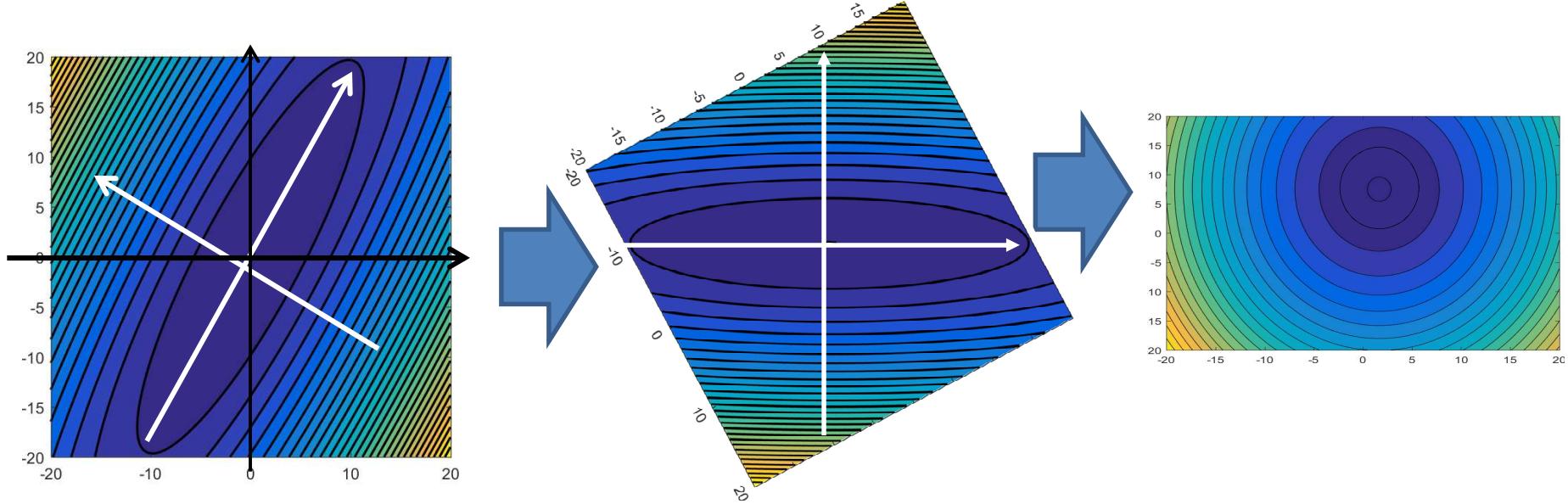


$$E = \frac{1}{2} \mathbf{w}^T \mathbf{A} \mathbf{w} + \mathbf{w}^T \mathbf{b} + c$$

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

- 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}

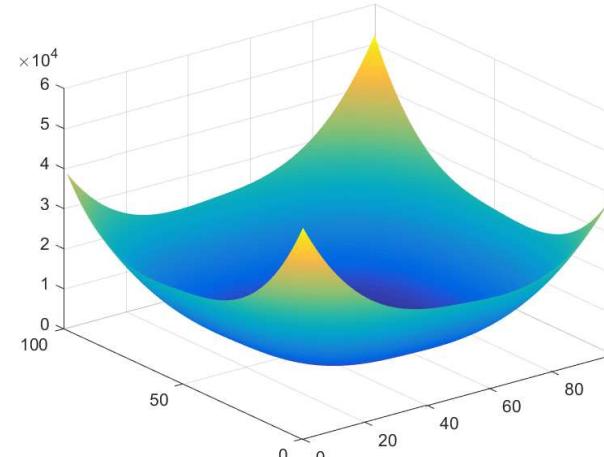
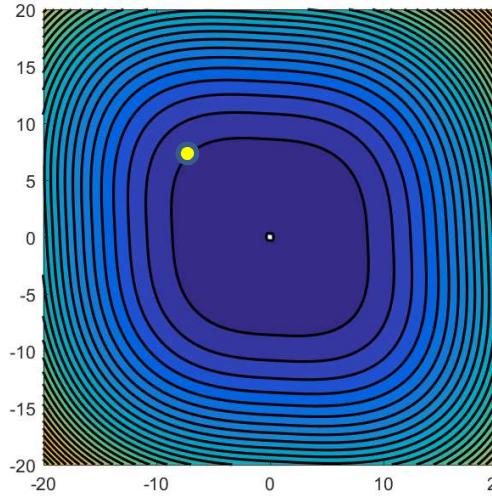
For non-axis-aligned quadratics..



- 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:

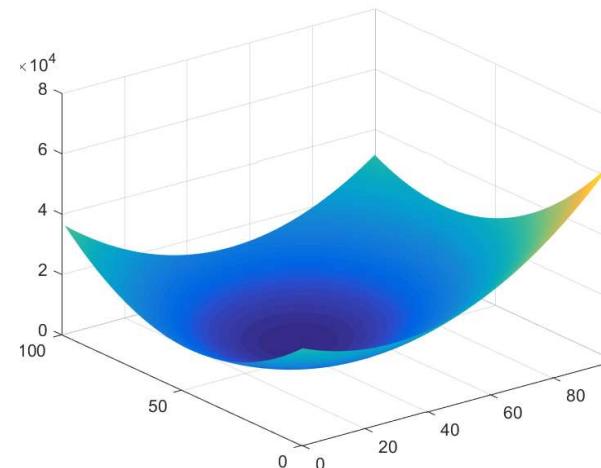
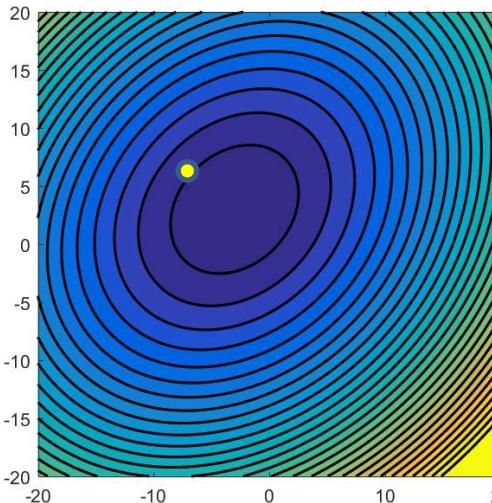
$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \mathbf{A}^{-1} \mathbf{b}$$

Generic differentiable *multivariate convex functions*

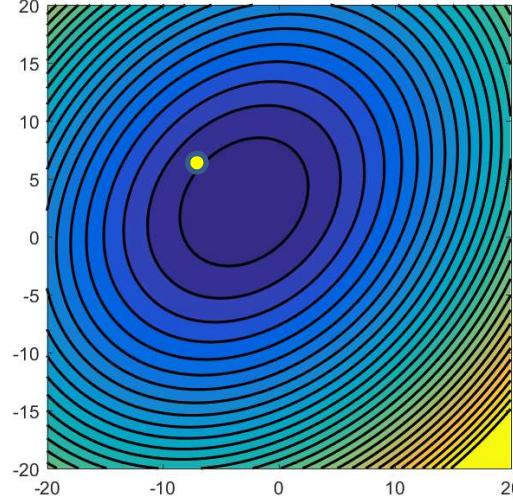
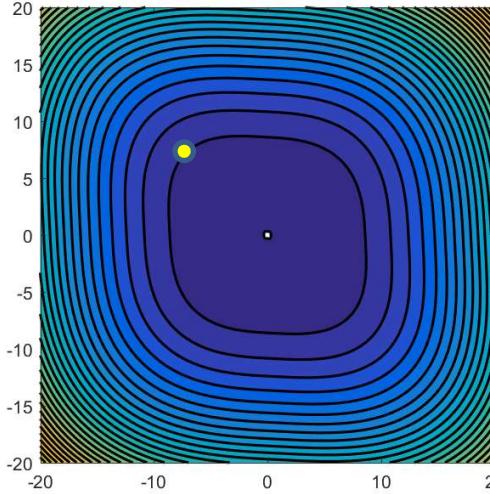


- Taylor expansion

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



Generic differentiable *multivariate convex functions*



- Taylor expansion

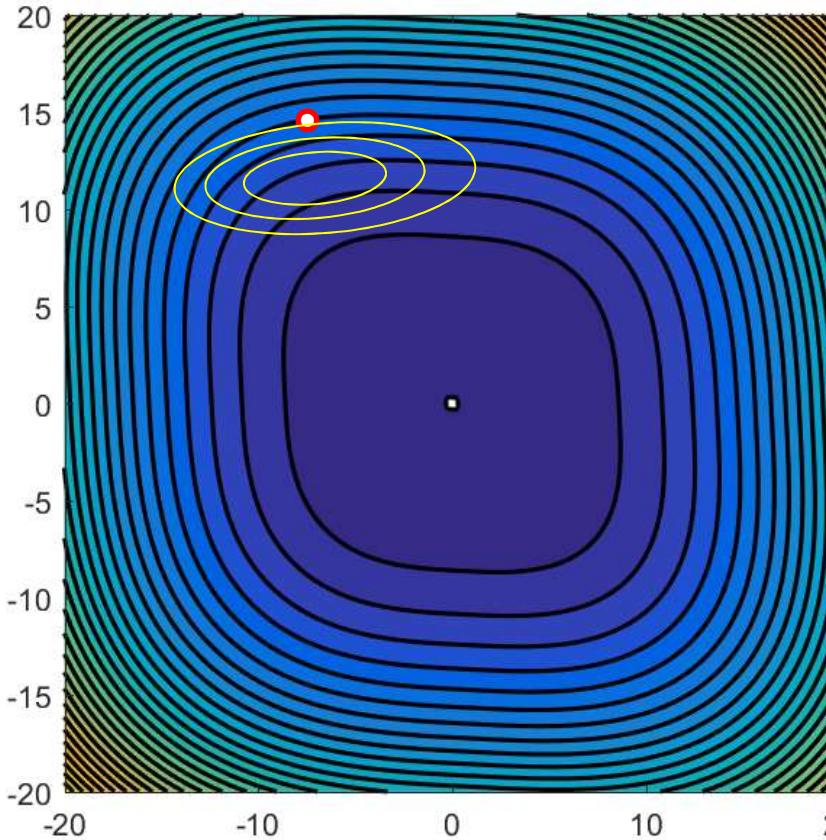
$$E(\mathbf{w}) \approx E(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} E(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2} (\mathbf{w} - \mathbf{w}^{(k)})^T H_E(\mathbf{w}^{(k)}) (\mathbf{w} - \mathbf{w}^{(k)}) + \dots$$

- 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 ($\eta = 1$)



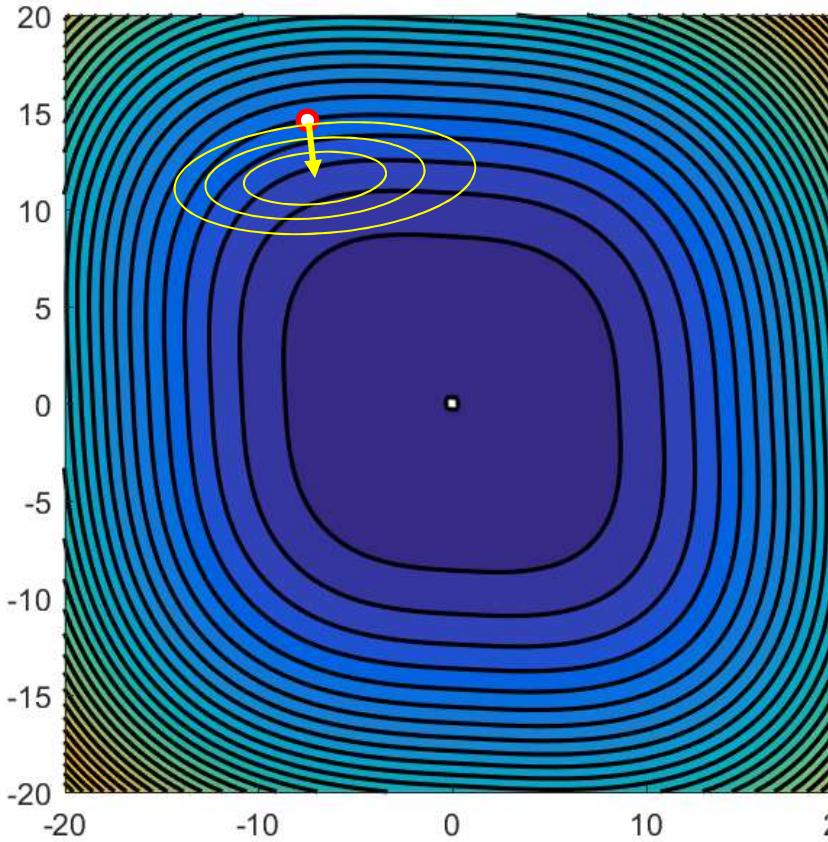
Fit a quadratic at each point and find the minimum of that quadratic

- 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 ($\eta = 1$)

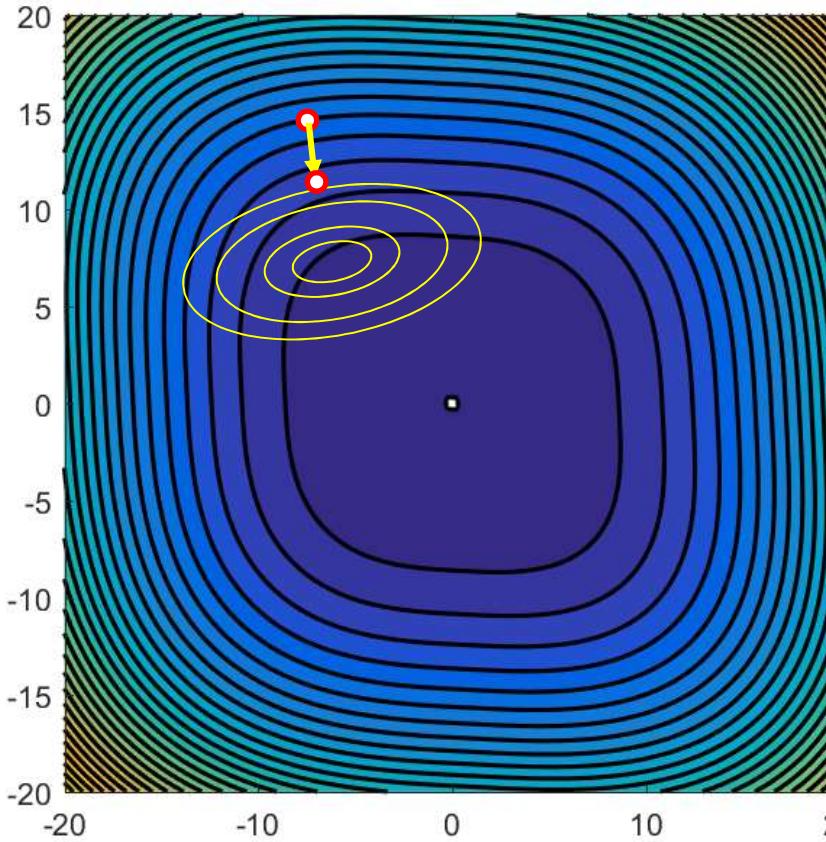


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

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

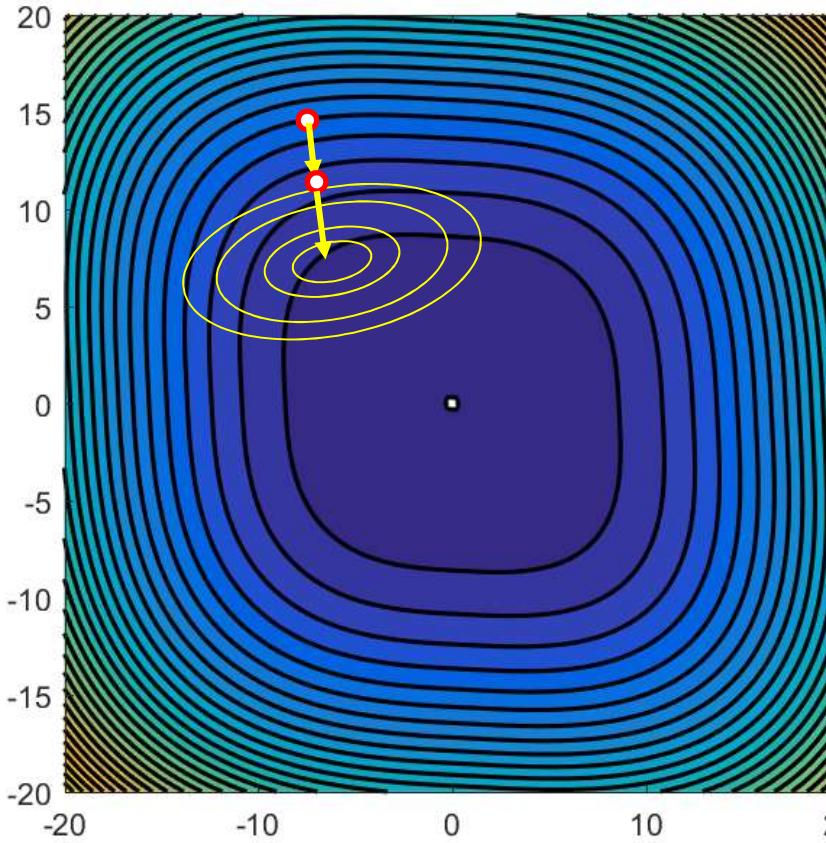


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

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

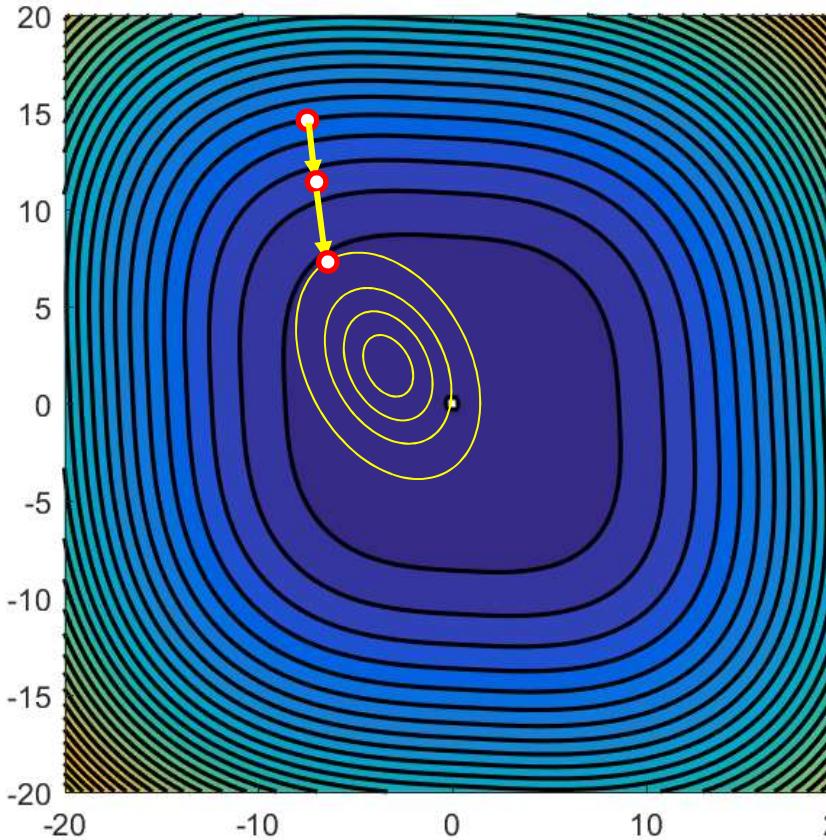


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method

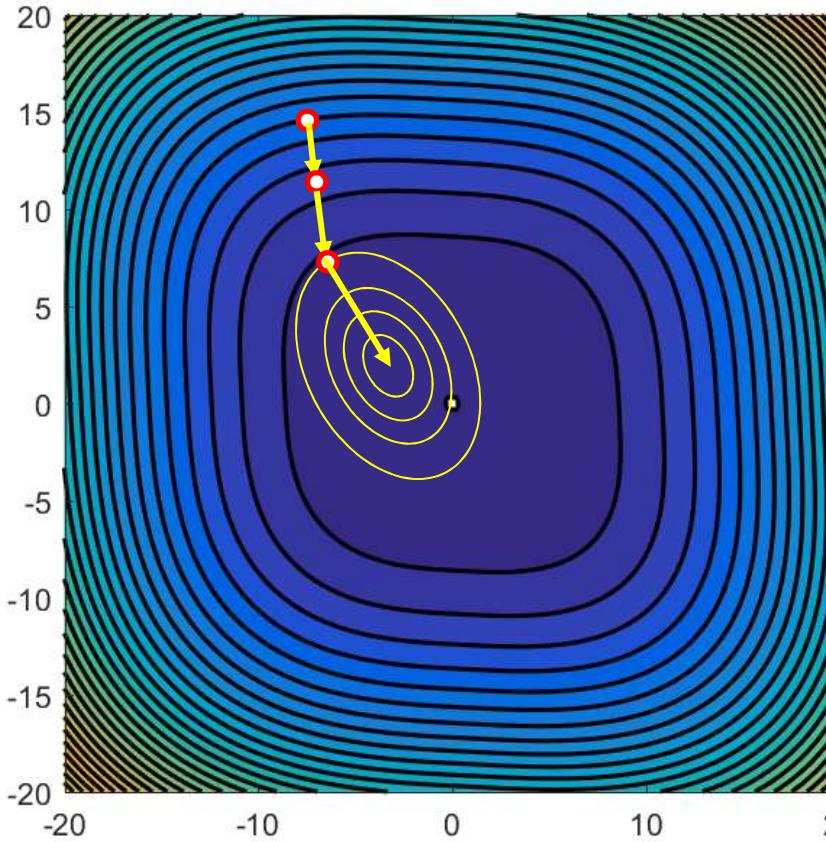


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method

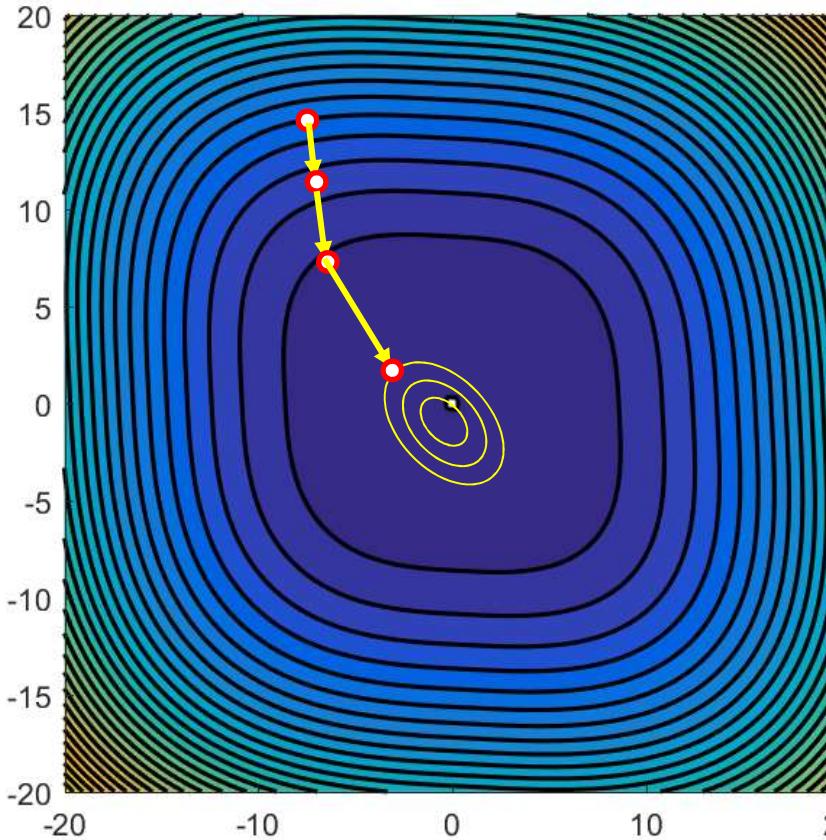


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method

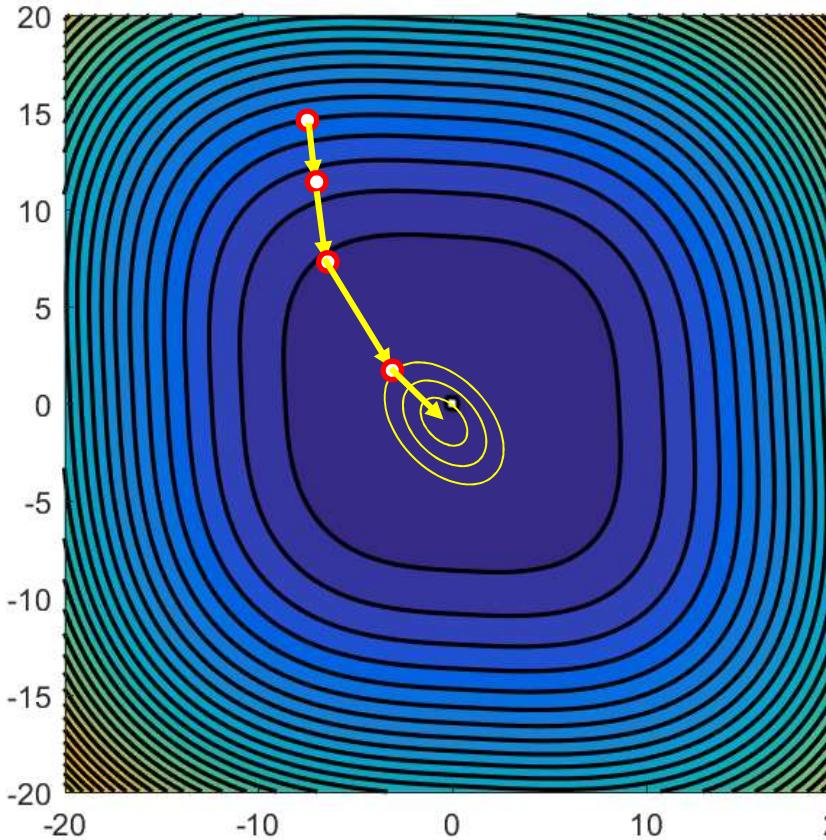


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method

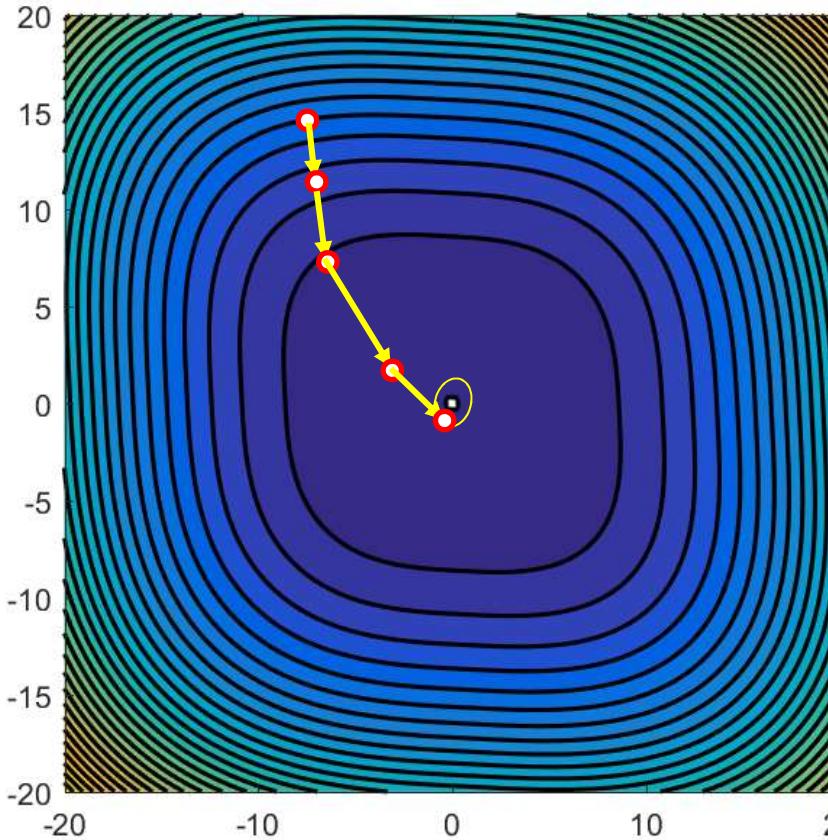


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method

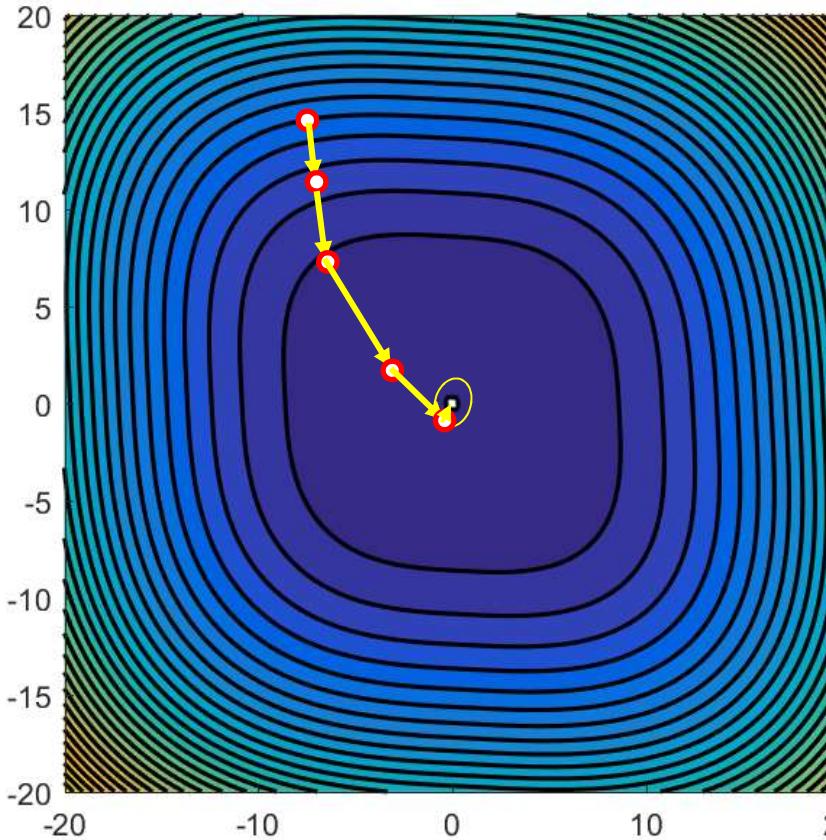


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

Minimization by Newton's method

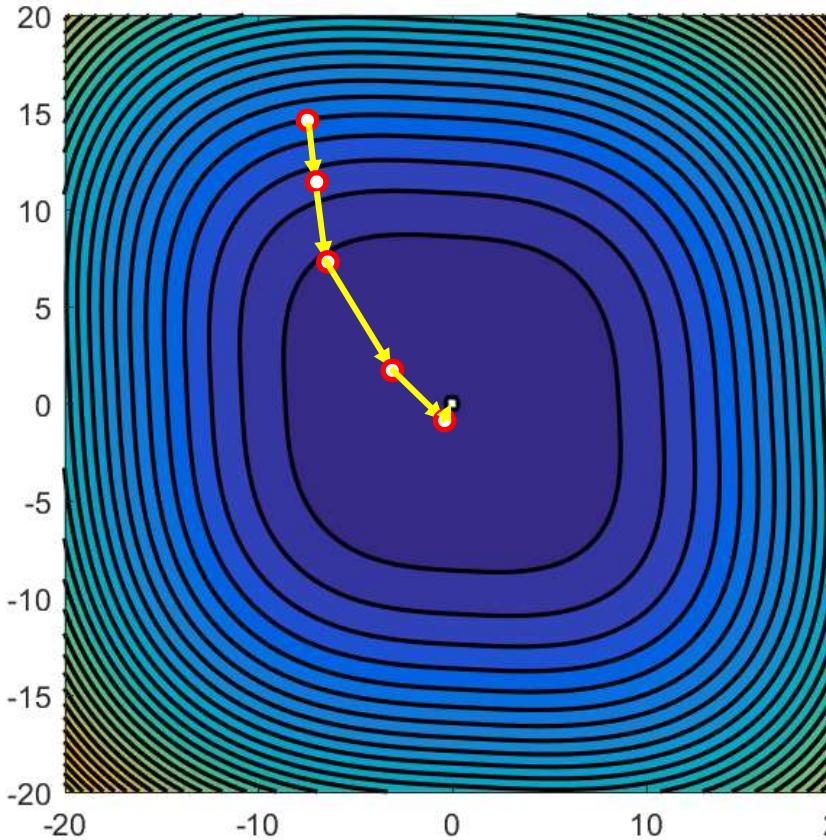


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

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

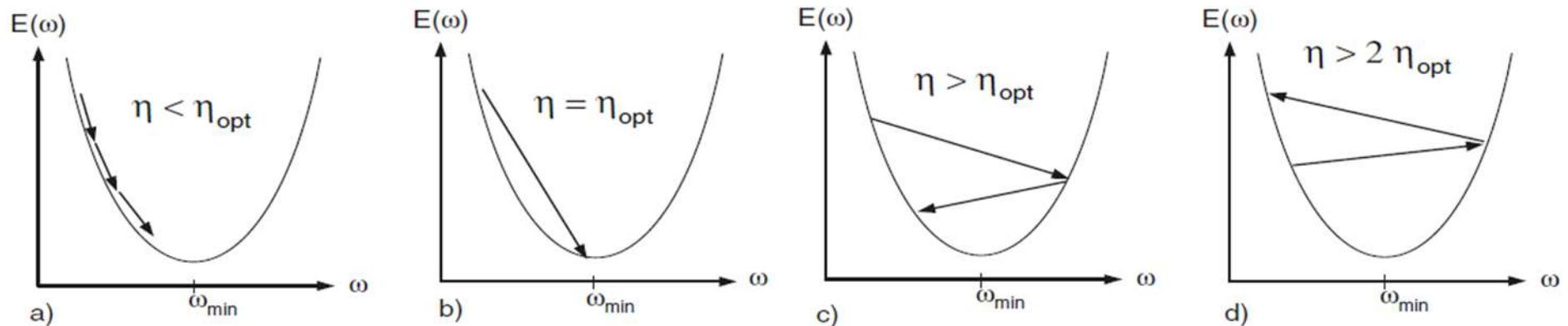


- For non-convex functions, the Hessian may not be positive semi-definite, in which case the algorithm can *diverge*
 - Goes away from, rather than towards the minimum
 - Now requires additional checks to avoid movement in directions corresponding to –ve Eigenvalues of the Hessian

Issues: 1 – contd.

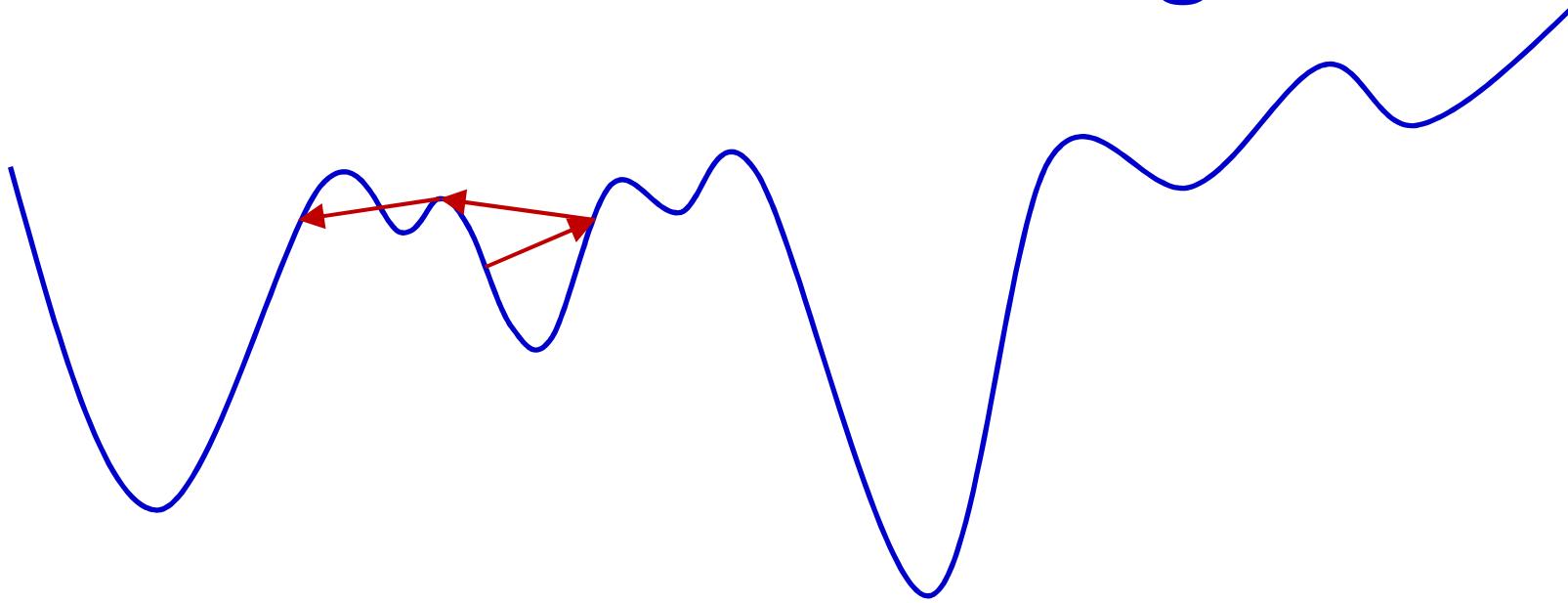
- A great many approaches have been proposed in the literature to *approximate* the Hessian in a number of ways and improve its positive definiteness
 - Boyden-Fletcher-Goldfarb-Shanno (BFGS)
 - And “low-memory” BFGS (L-BFGS)
 - Estimate Hessian from finite differences
 - Levenberg-Marquardt
 - Estimate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



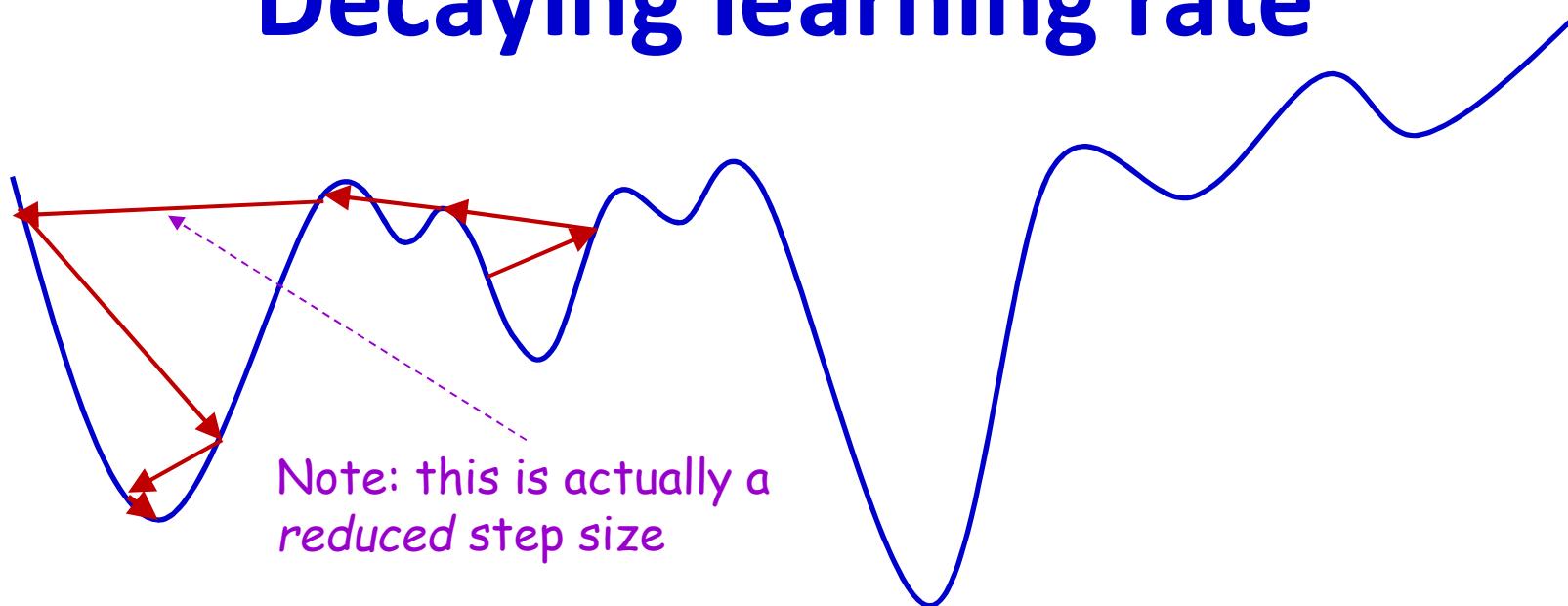
- Much of the analysis we just saw was based on trying to ensure that the step size was not so large as to cause divergence within a convex region
 - $\eta < 2\eta_{opt}$

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: $\eta_k = \frac{\eta_0}{k+1}$

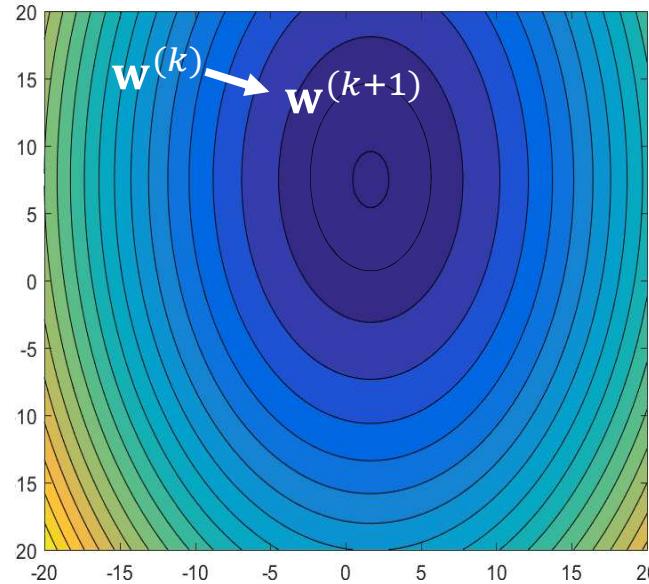
- Quadratic decay: $\eta_k = \frac{\eta_0}{(k+1)^2}$

- 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

Lets take a step back



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

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

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

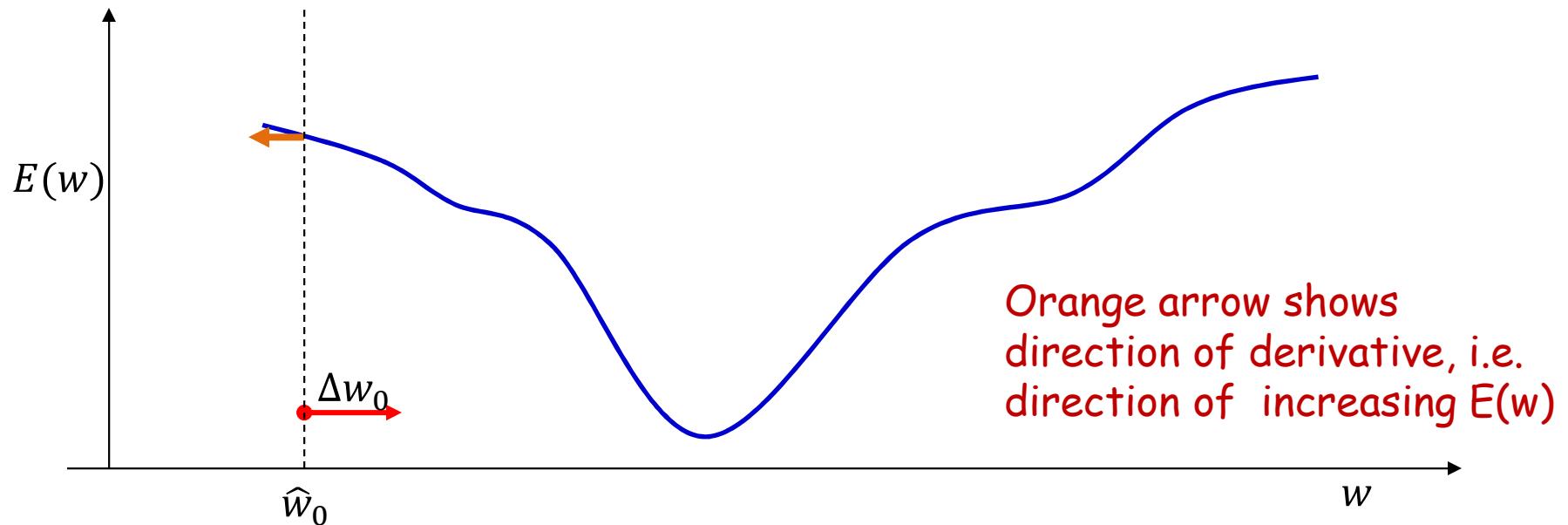
Derivative-*inspired* algorithms

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

RProp

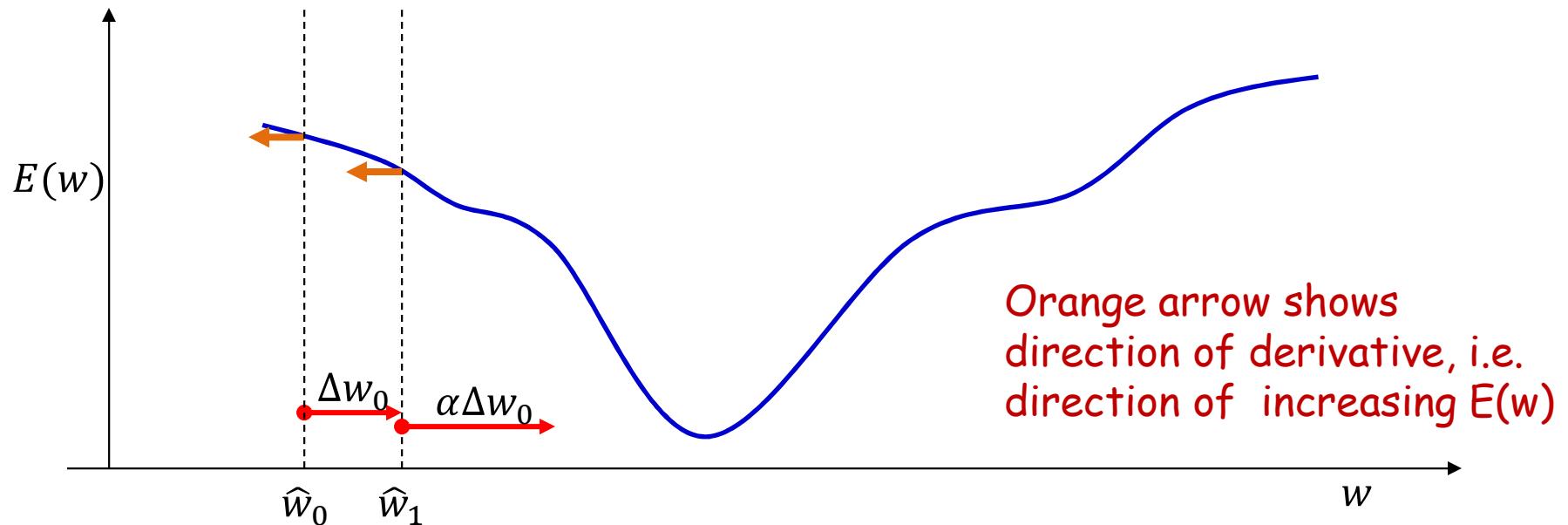
- *Resilient* propagation
- Simple algorithm, to be followed *independently* for each component
 - I.e. steps in different directions are not coupled
- At each time
 - If the derivative at the current location recommends continuing in the same direction as before (i.e. has not changed sign from earlier):
 - *increase* the step, and continue in the same direction
 - If the derivative has changed sign (i.e. we've overshot a minimum)
 - *reduce* the step and reverse direction

Rprop



- 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

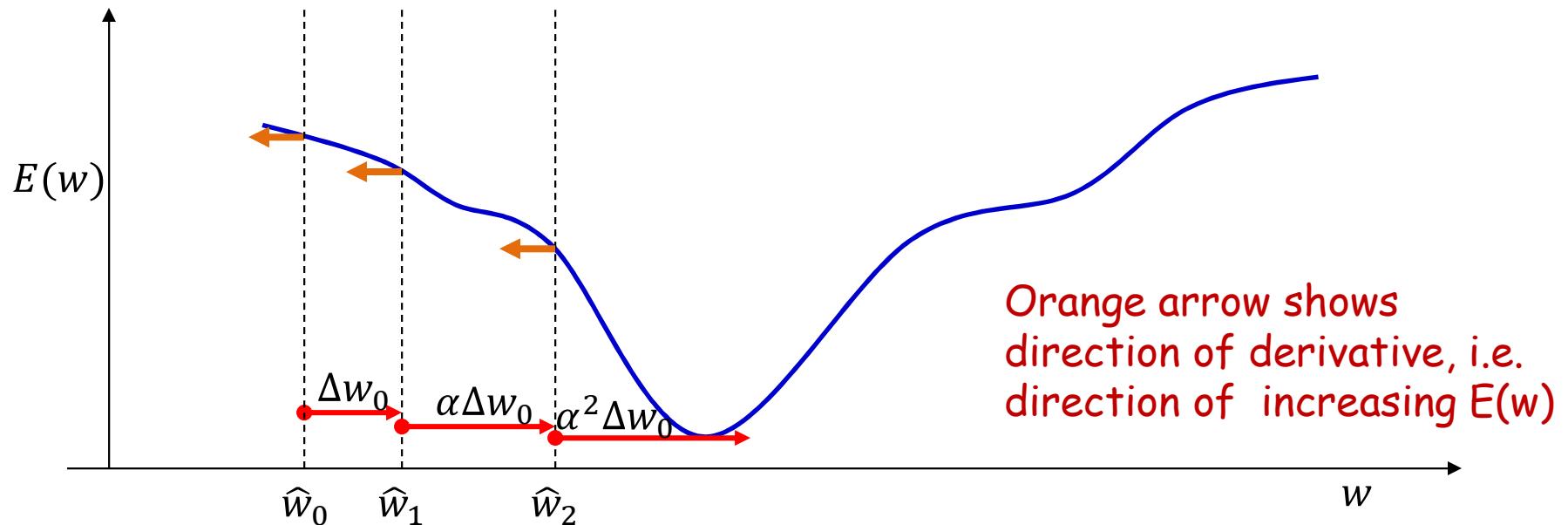


- Compute the derivative in the new location
 - If the derivative has not changed sign from the previous location, increase the step size and take a longer step

$$\alpha > 1$$

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

Rprop

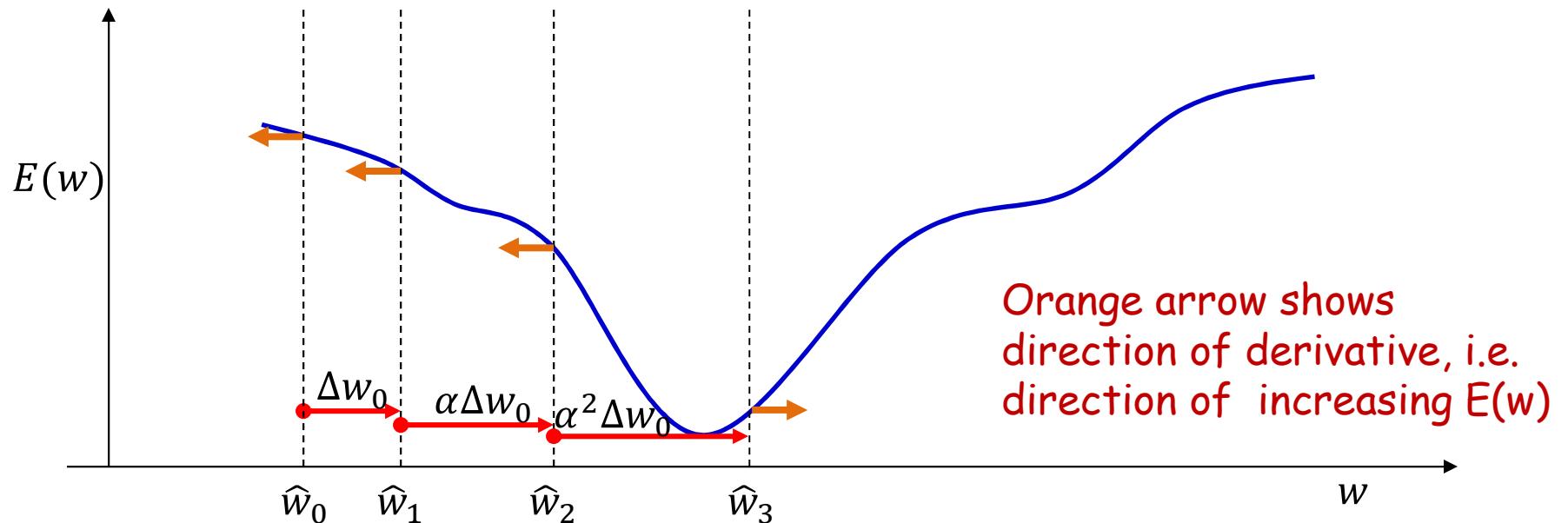


- Compute the derivative in the new location
 - If the derivative has not changed sign from the previous location, increase the step size and take a step

$$\alpha > 1$$

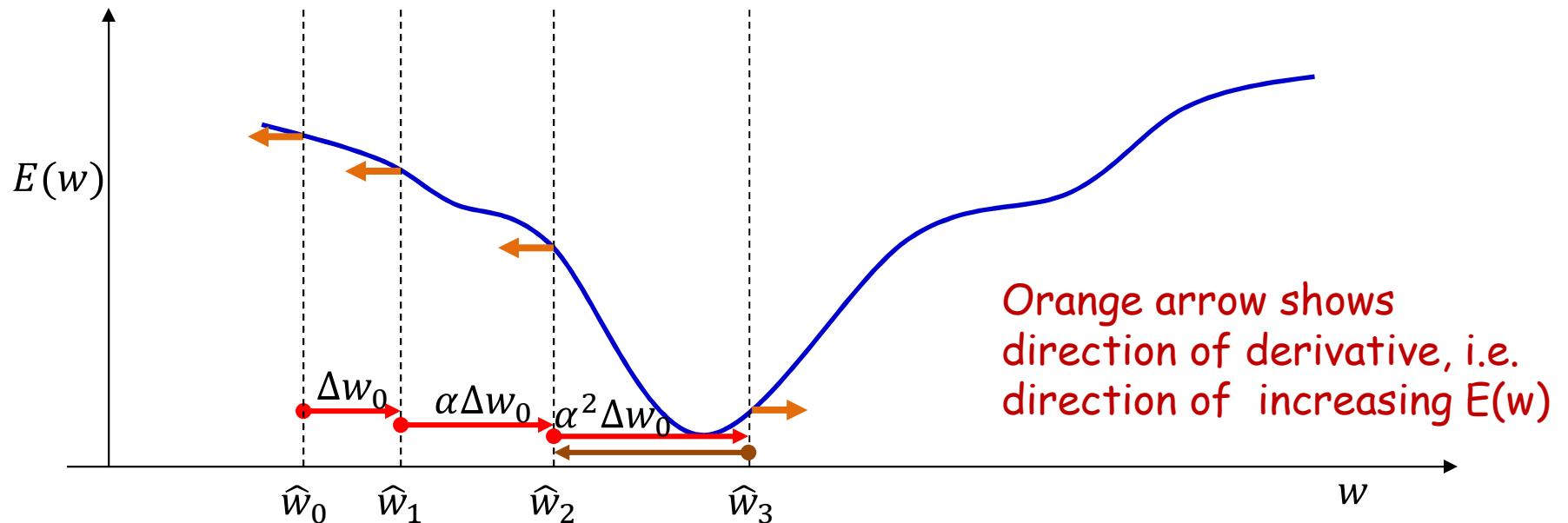
- $\Delta w = \alpha \Delta w$
- $\hat{w} = \hat{w} - \Delta w$

Rprop



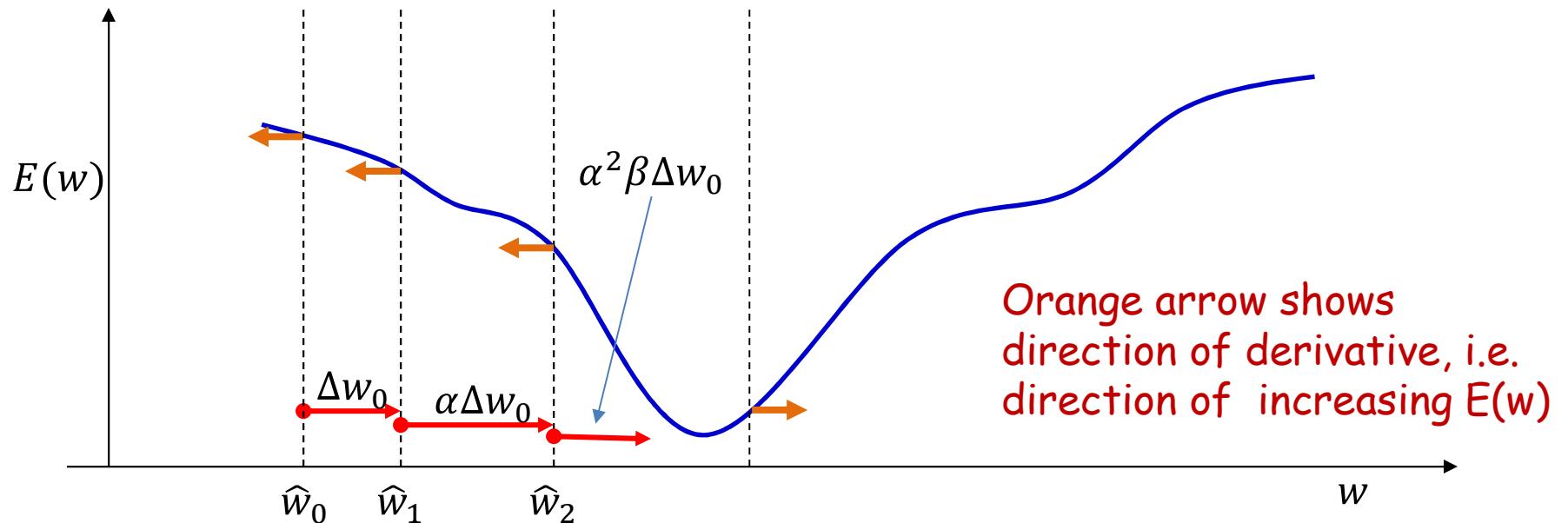
- Compute the derivative in the new location
 - If the derivative has changed sign

Rprop



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

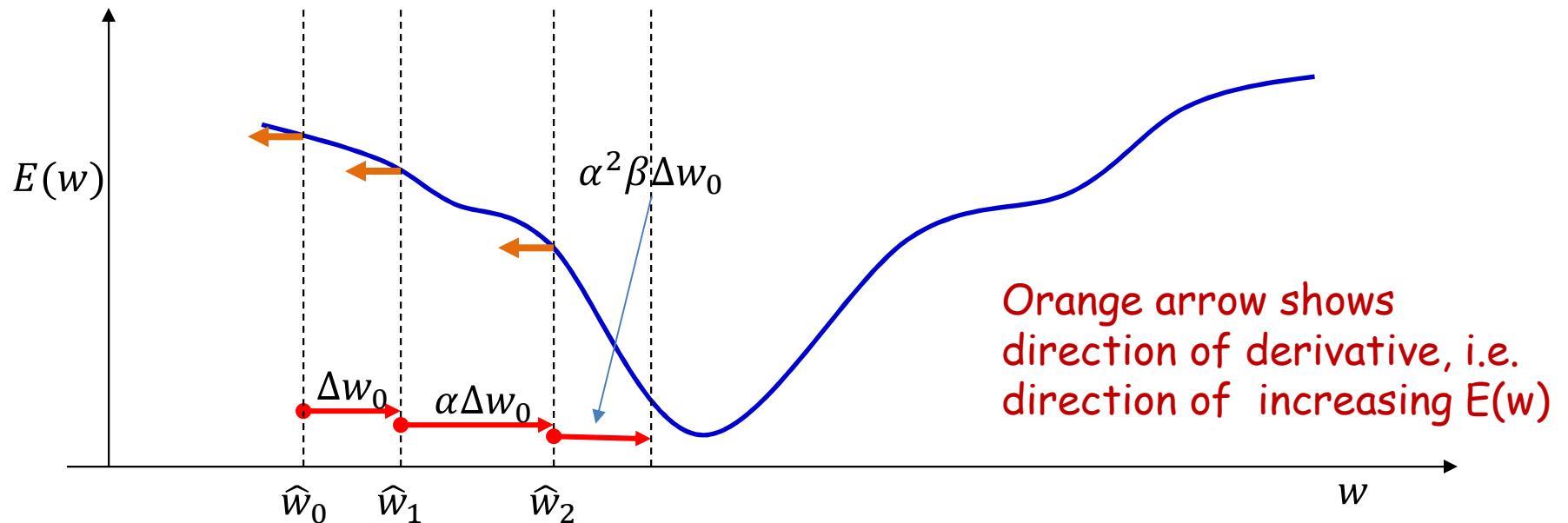
Rprop



- 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 $\text{sign}(prevD(l, i, j)) == \text{sign}(D(l, i, j))$:

- $\Delta w_{l,i,j} = \min(\alpha \Delta w_{l,i,j}, \Delta_{max})$

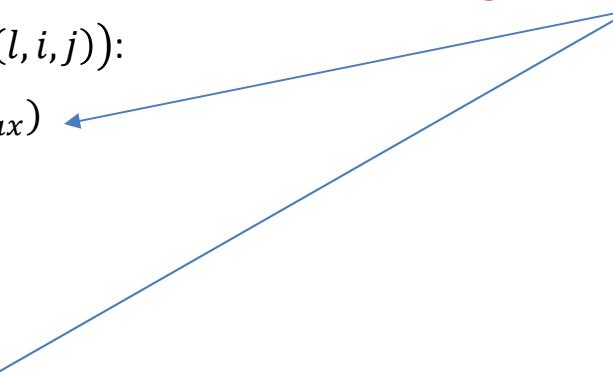
- $prevD(l, i, j) = D(l, i, j)$

- 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:
 - $w_{l,i,j} = w_{l,i,j} + \Delta w_{l,i,j}$
 - $\Delta w_{l,i,j} = \beta \Delta w_{l,i,j}$

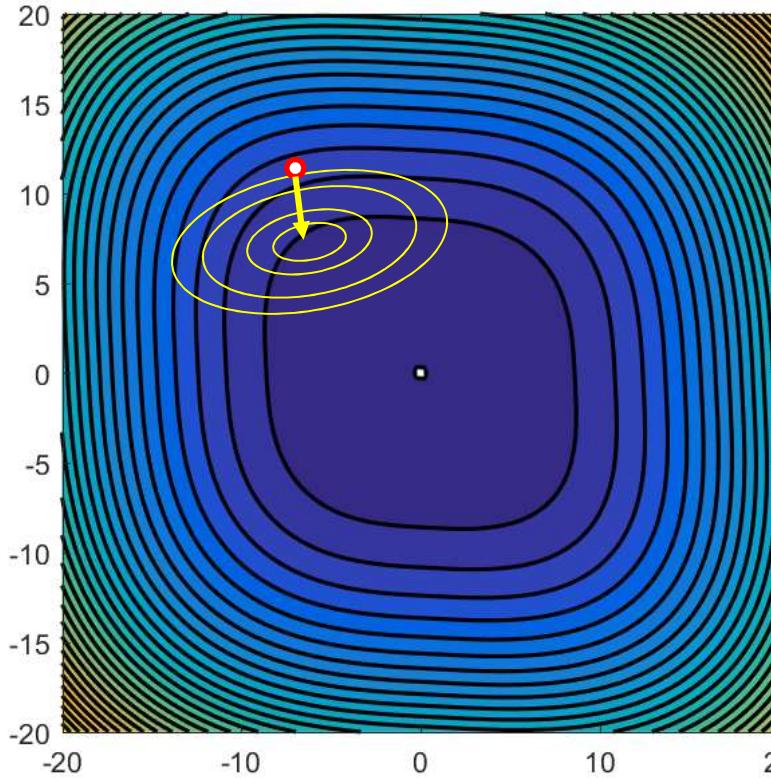
Obtained via backprop

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

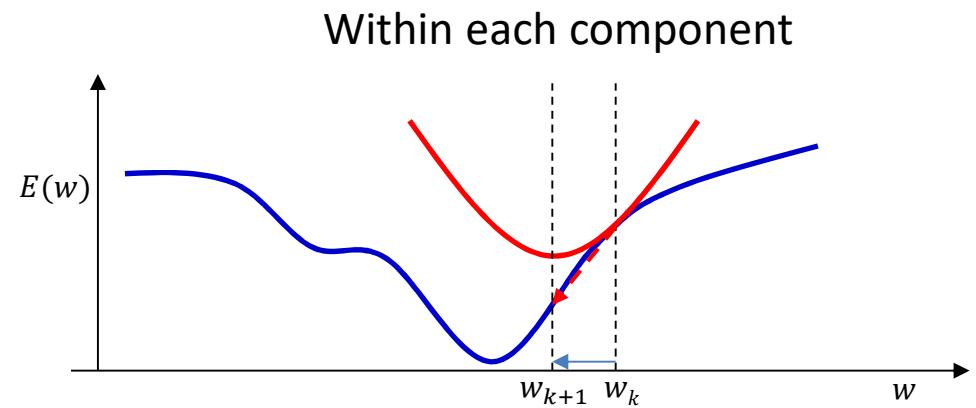
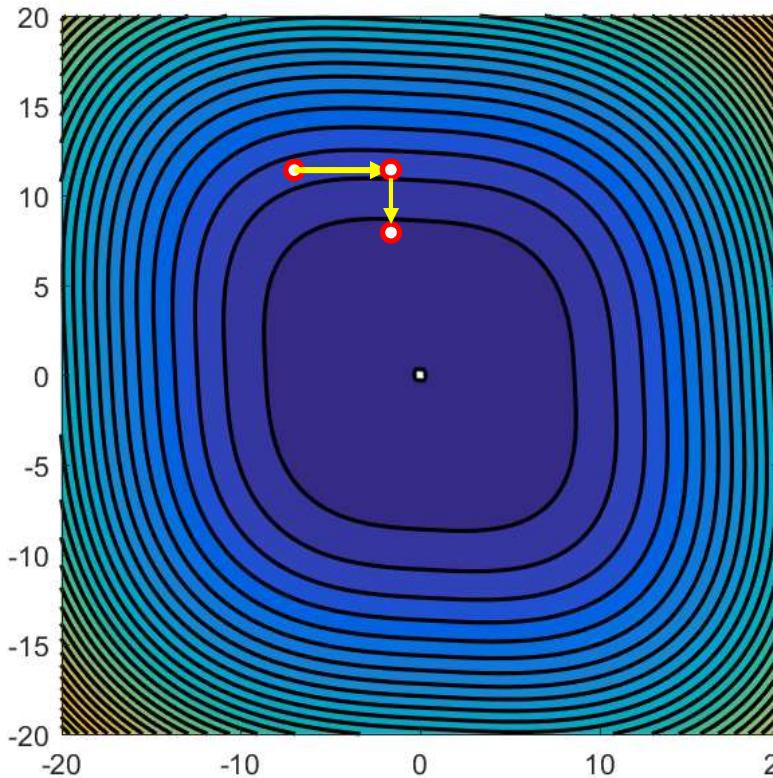


- Quickprop employs the Newton updates with two modifications

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

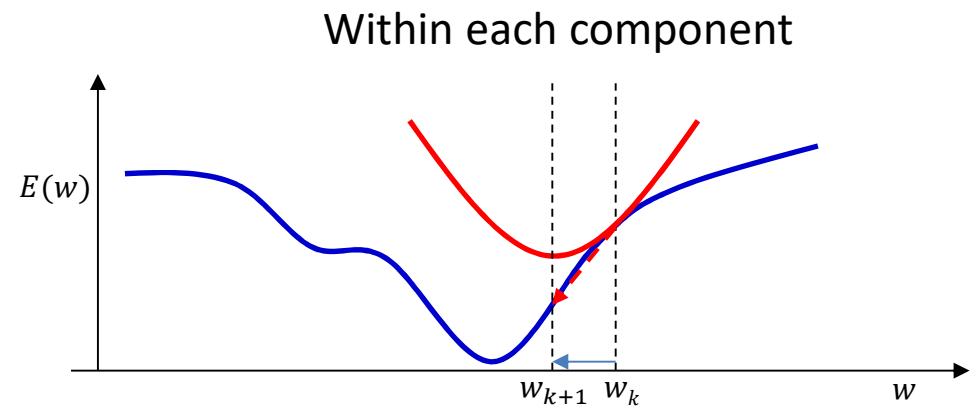
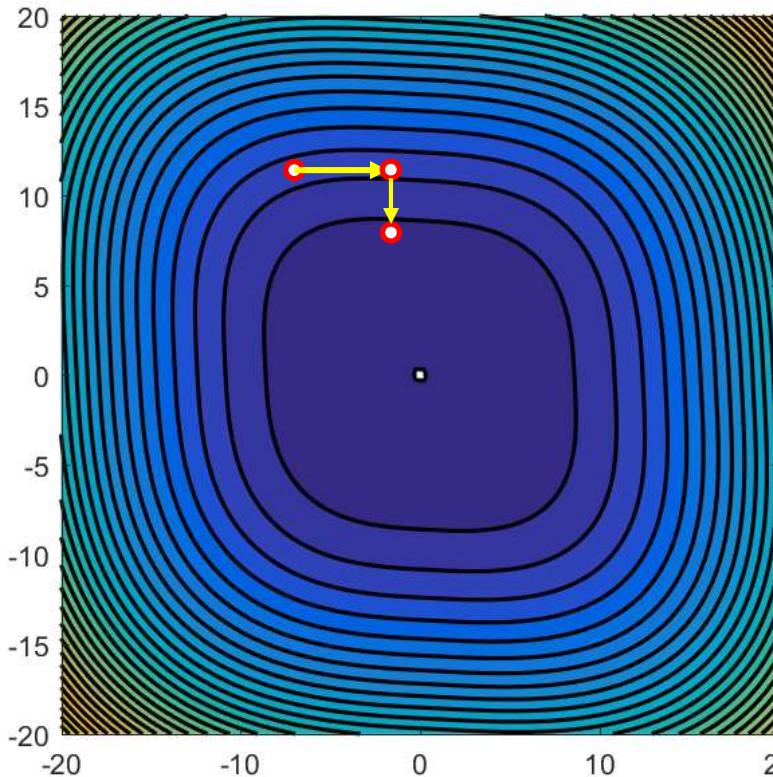
- But with two modifications

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

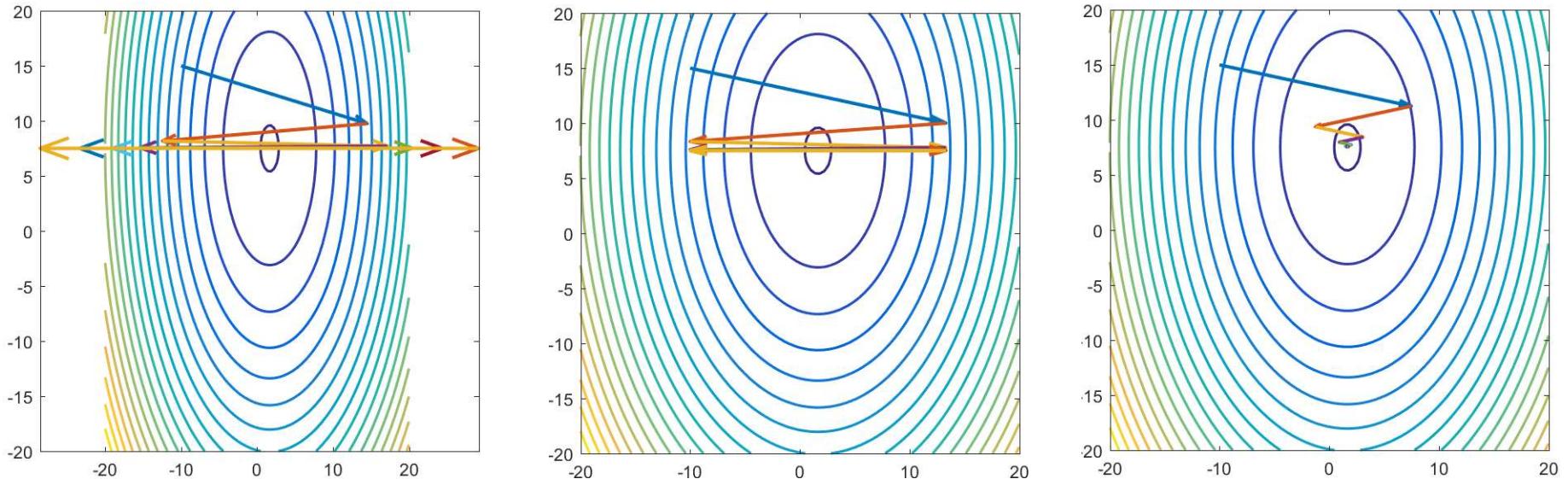
Quickprop

- Employs Newton updates with empirically derived derivatives
- Prone to some instability for non-convex objective functions
- But is still one of the fastest training algorithms for many problems

Story so far : Convergence

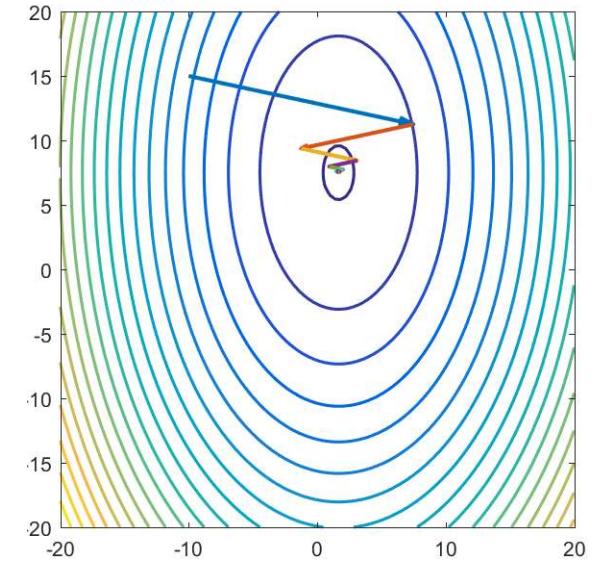
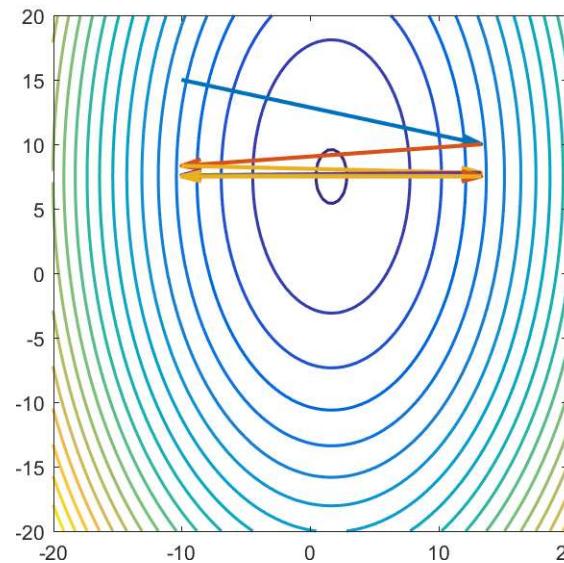
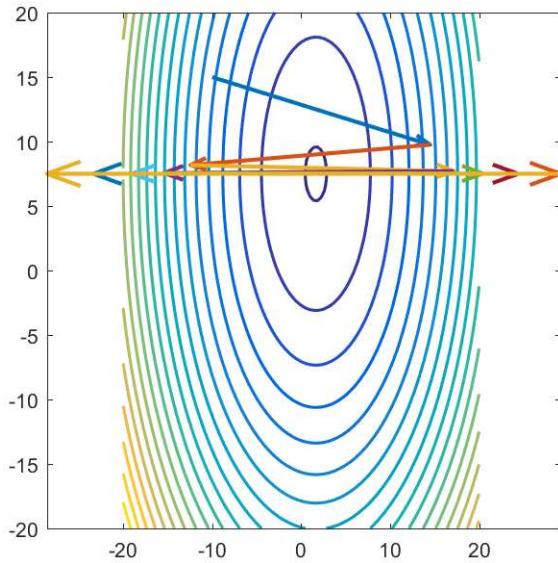
- Gradient descent can miss obvious answers
 - And this may be a *good* thing
- Vanilla gradient descent may be too slow or unstable due to the differences between the dimensions
- Second order methods can normalize the variation across dimensions, but are complex
- Adaptive or decaying learning rates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

A closer look at the convergence problem



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

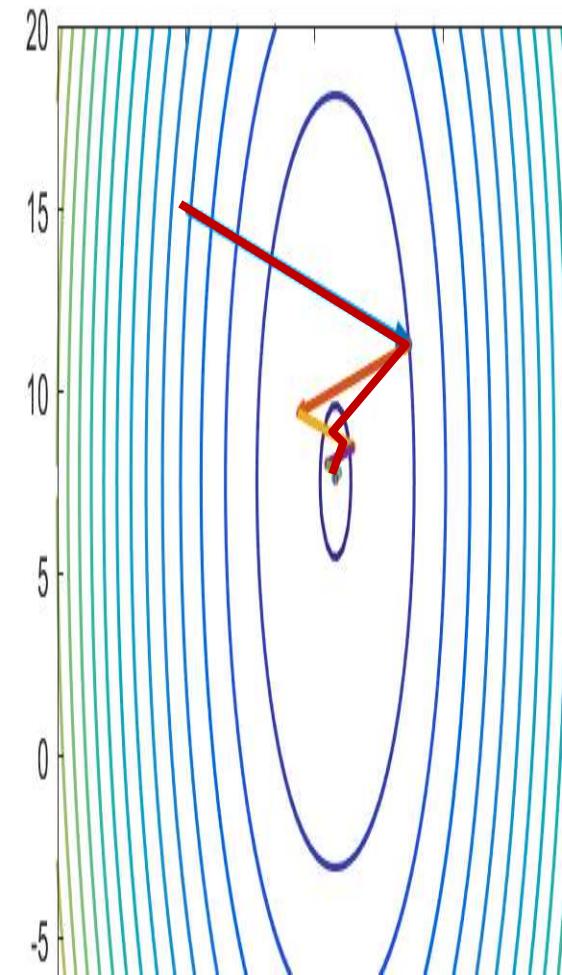
A closer look at the convergence problem



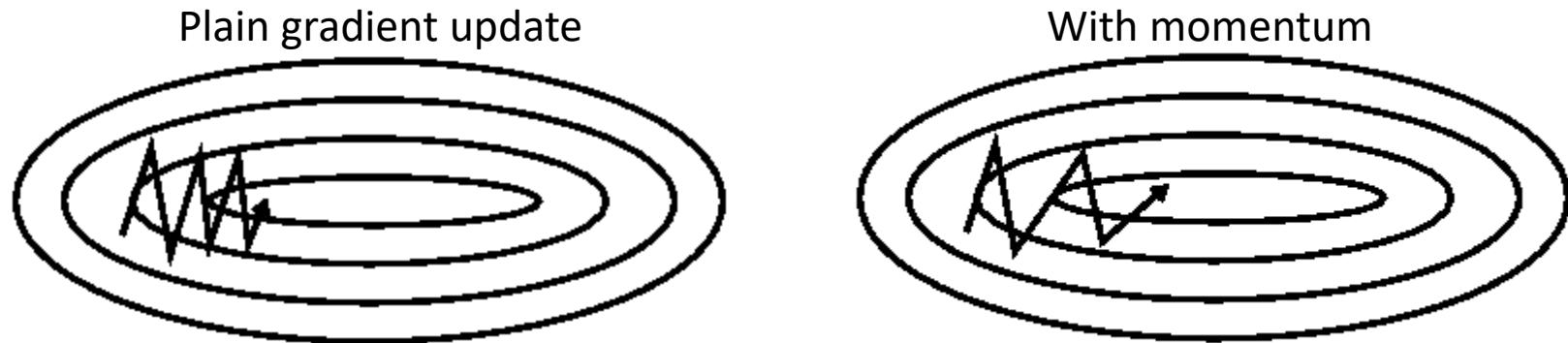
- With dimension-independent learning rates, the solution will converge smoothly in some directions, but oscillate or diverge in others
- Proposal:**
 - Keep track of oscillations
 - Emphasize steps in directions that converge smoothly
 - Shrink steps in directions that bounce around..

The momentum methods

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



Momentum Update



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

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

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

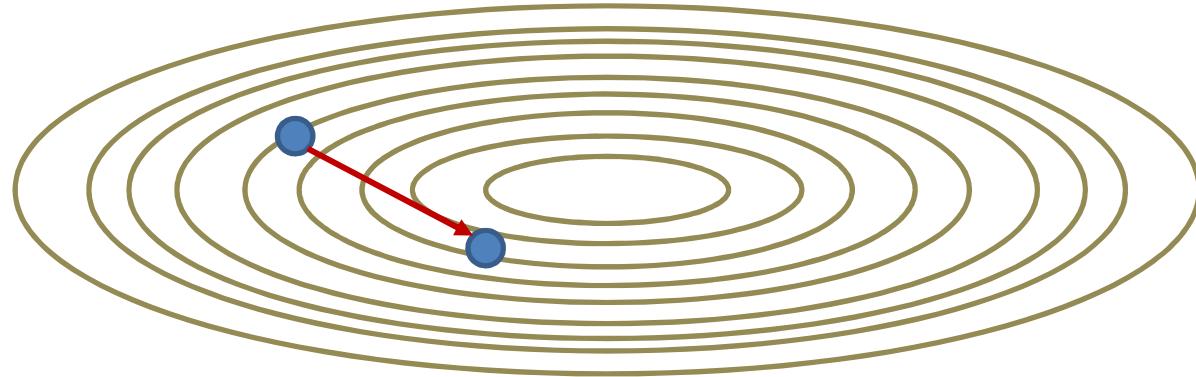
Training by gradient descent

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

Training with momentum

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

Momentum Update

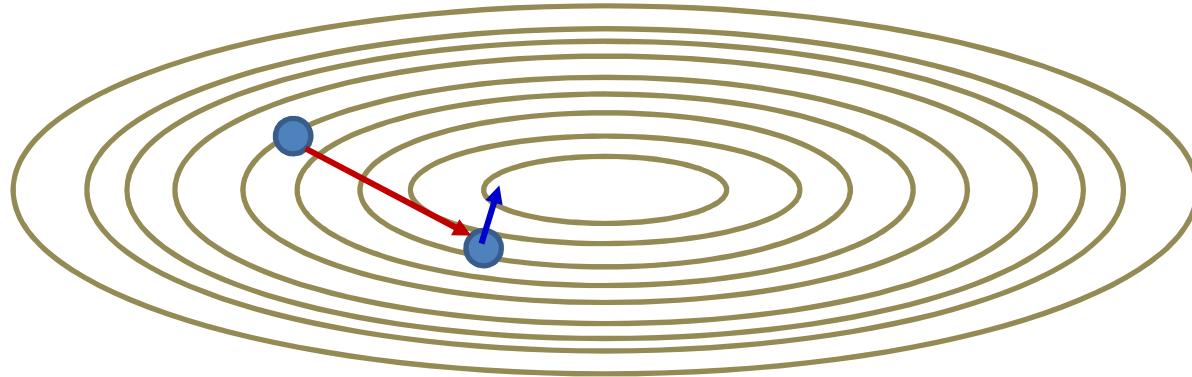


- The momentum method

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

- At any iteration, to compute the current step:

Momentum Update

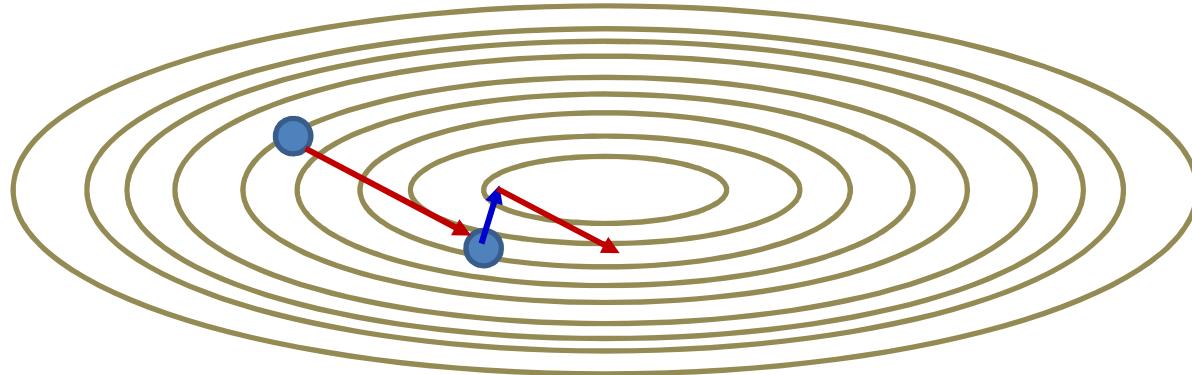


- The momentum method

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

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

Momentum Update

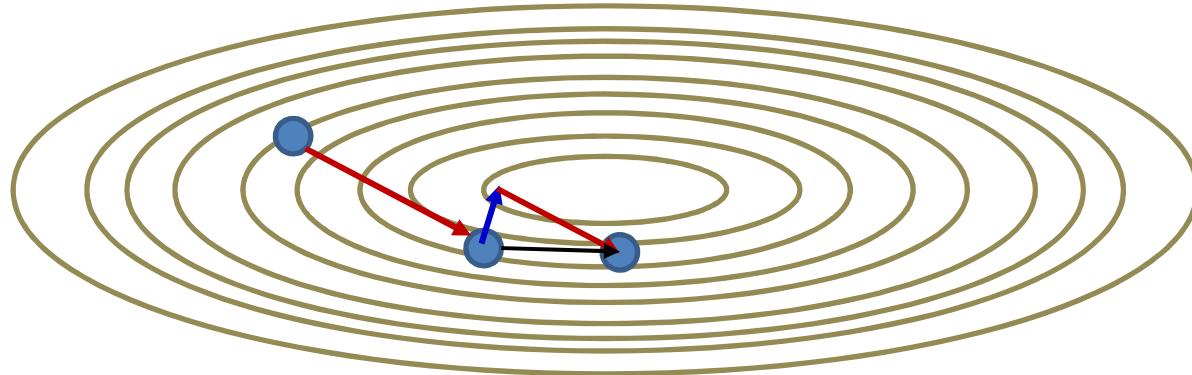


- The momentum method

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

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

Momentum Update



- The momentum method

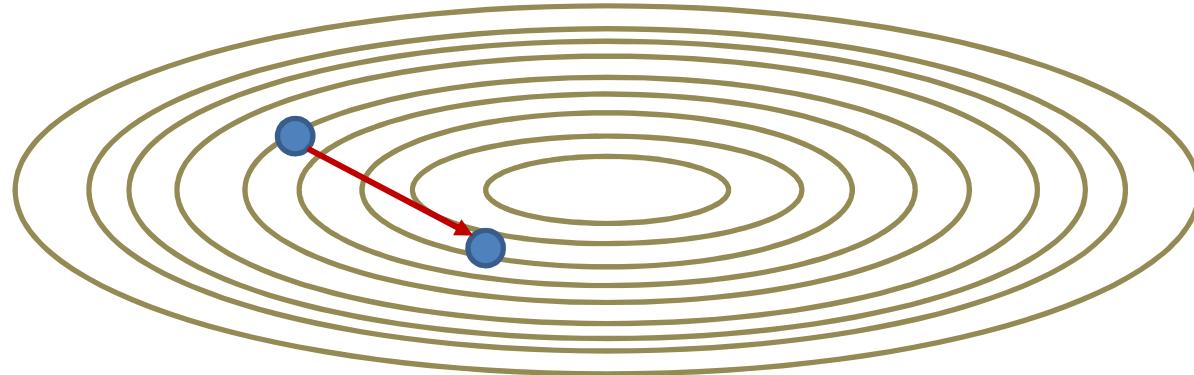
$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Loss(W^{(k-1)})^T$$

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

Momentum update

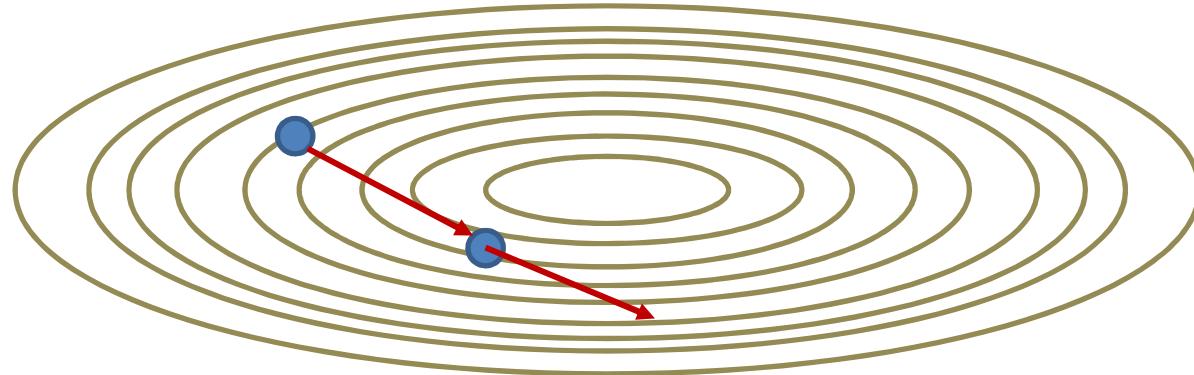
- Takes a step along the past running average *after* walking along the gradient
- The procedure can be made more optimal by reversing the order of operations..

Nestorov's Accelerated Gradient



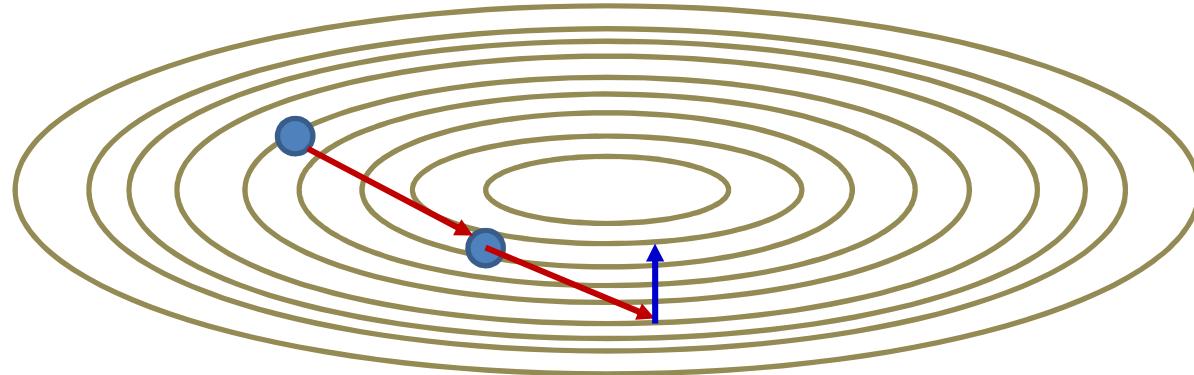
- Change the order of operations
- At any iteration, to compute the current step:

Nestorov's Accelerated Gradient



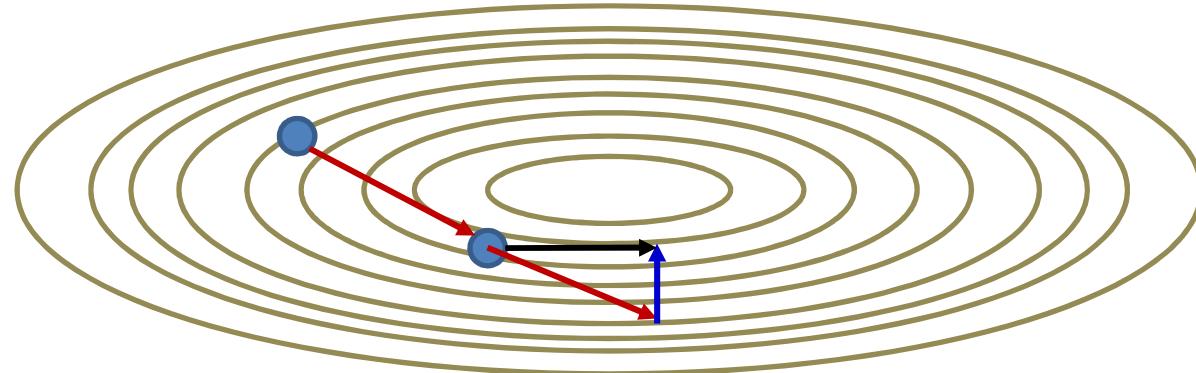
- Change the order of operations
- At any iteration, to compute the current step:
 - First extend the previous step

Nestorov's Accelerated Gradient



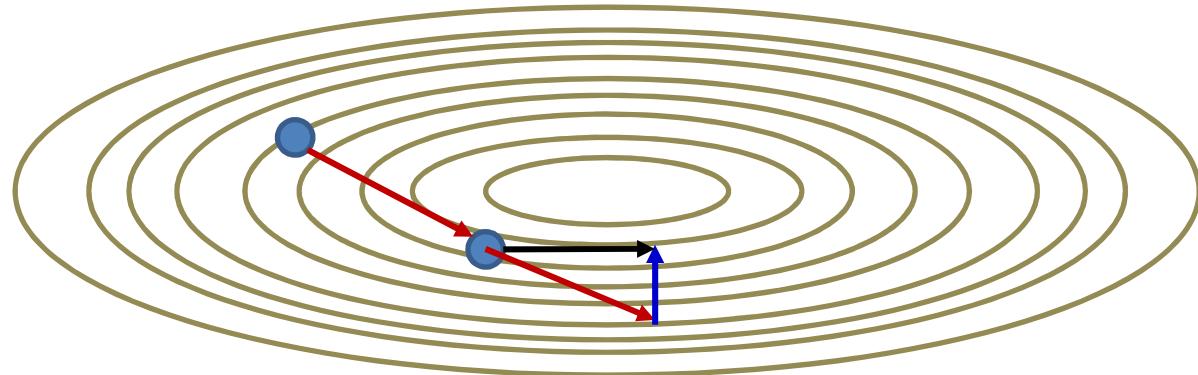
- Change the order of operations
- At any iteration, to compute the current step:
 - First extend the previous step
 - Then compute the gradient step at the resultant position

Nestorov's Accelerated Gradient



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

Nestorov's Accelerated Gradient

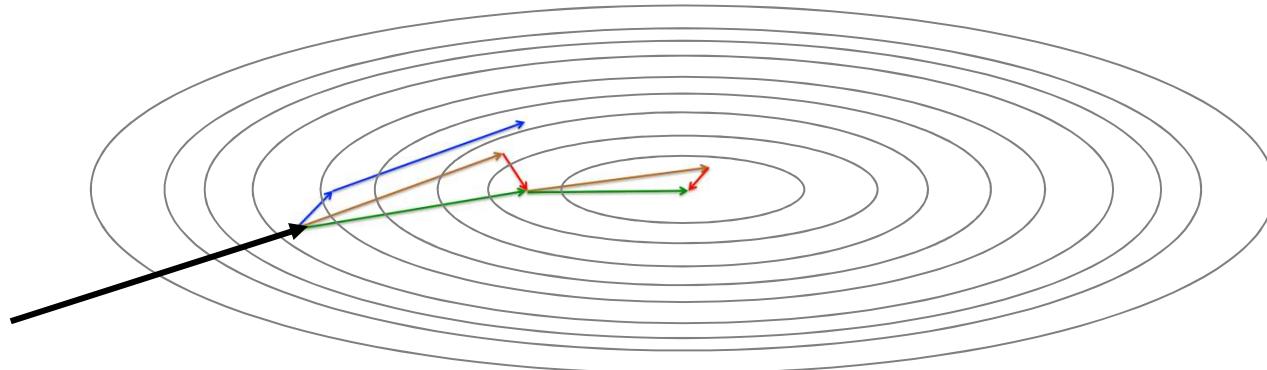


- Nestorov's method

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

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

Nestorov's Accelerated Gradient



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

Training with Nestorov

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