

# Training Neural Networks Optimization

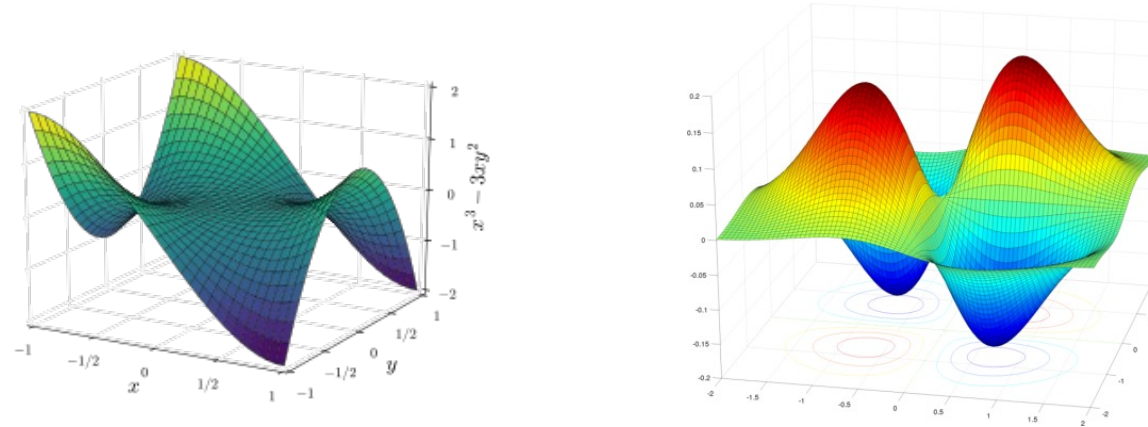
M. Soleymani

Sharif University of Technology

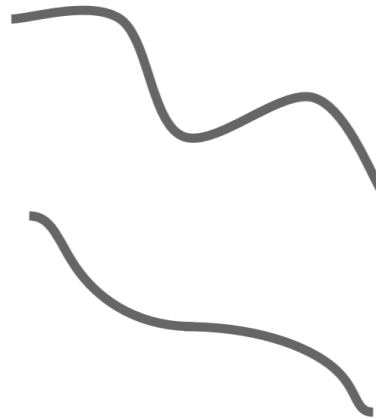
Spring 2024

Most slides have been adapted from Bhiksha Raj, 11-785, CMU  
and some from Fei Fei Li et. al, cs231n

# Optimization: Problems

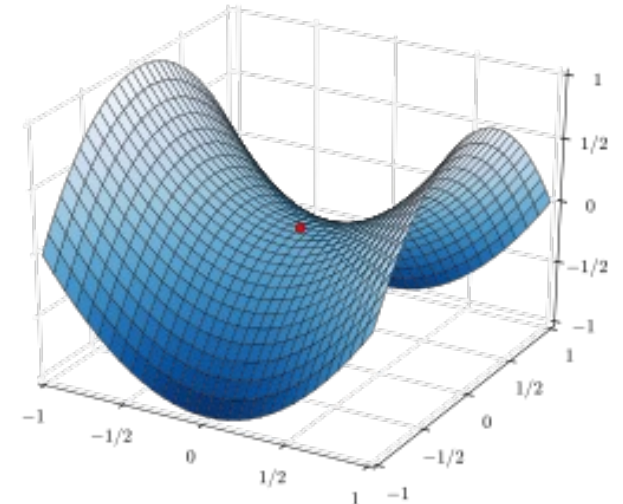


- What if the loss function has a local minima or saddle point?



# Saddle point

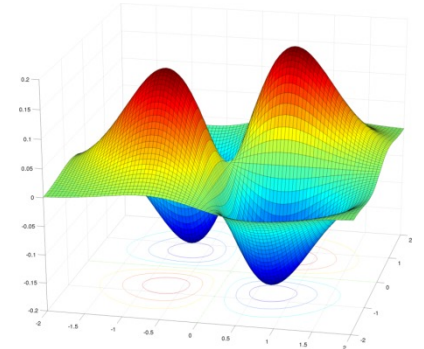
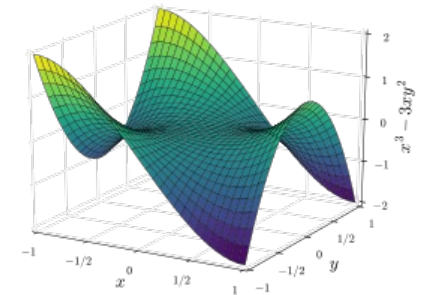
- **Saddle point:** A point where
  - The gradient is zero
  - The surface increases in some directions, but decreases in others
    - Some of the Eigenvalues of the Hessian are positive; others are negative



# The Error Surface

- **Popular hypothesis:**

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



# 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 small size, trained on finite data, you *can* have horrible local minima
- Watch this space...

# Convergence

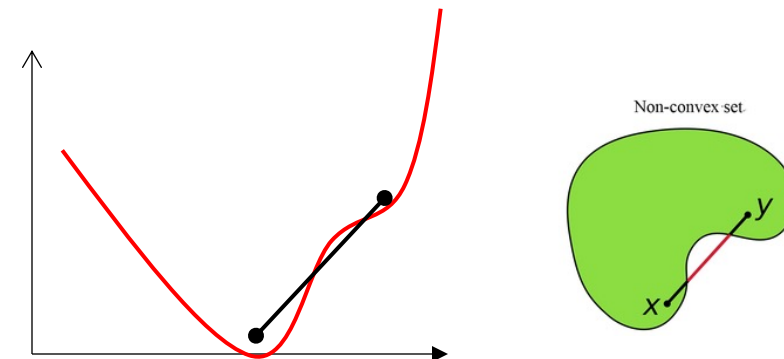
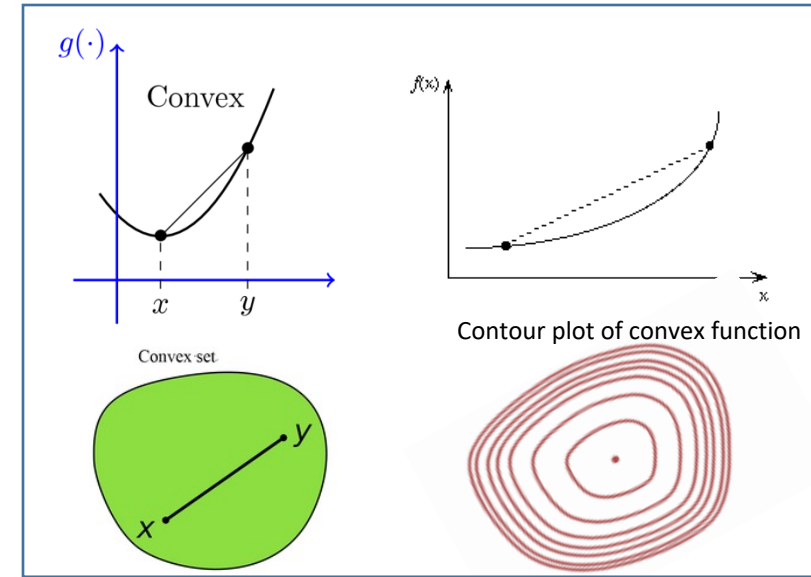
- In the discussion so far we have assumed the training arrives at a local minimum
- Does it always converge?
- How long does it take?
- Hard to analyze for an MLP, but we can look at the problem through the lens of convex optimization

# A quick tour of (convex) optimization



# Convex loss functions

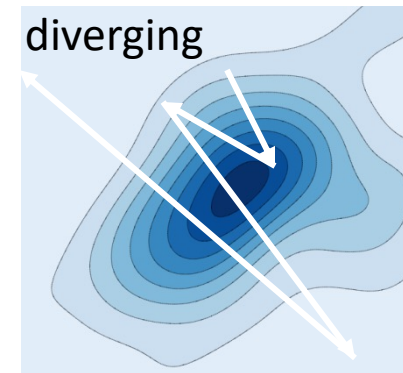
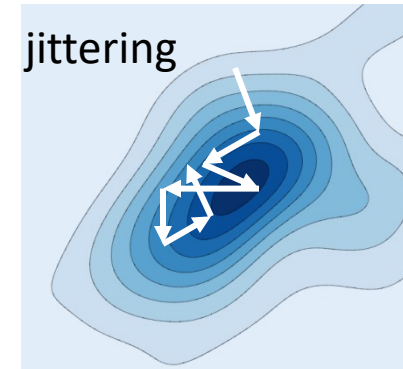
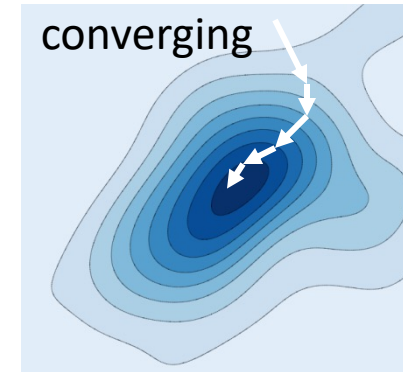
- A surface is “convex” if it is continuously curving upward
  - We can connect any two points above the surface without intersecting it
  - Many mathematical definitions that are equivalent
- Caveat: Neural network error surface is generally not convex
  - Streetlight effect





# 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?

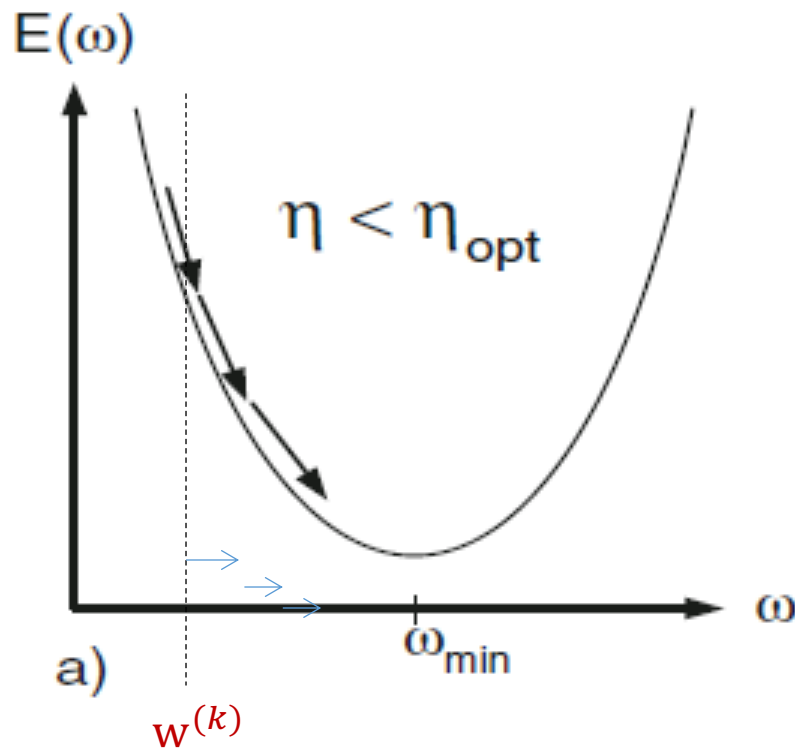


# Convergence for quadratic surfaces

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

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

Gradient descent with fixed step size  $\eta$  to estimate *scalar* parameter  $w$

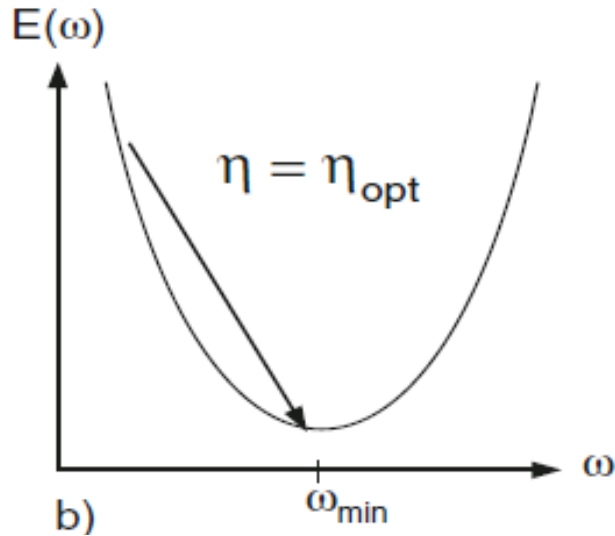


- Gradient descent to find the optimum of a quadratic, starting from  $w^{(k)}$
- Assuming fixed step size  $\eta$
- What is the optimal step size  $\eta$  to get there fastest?

# 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

$$E(w) = E(w^{(k)}) + E'(w^{(k)})(w - w^{(k)}) + \frac{1}{2}E''(w^{(k)})(w - w^{(k)})^2$$

– Taylor expansion

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

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

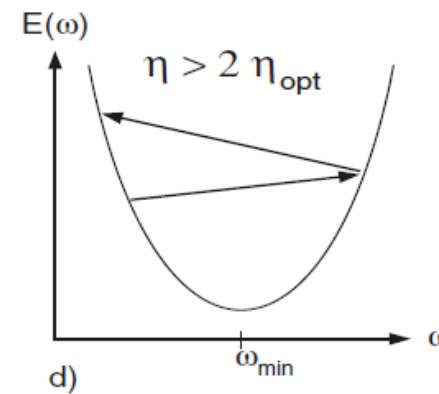
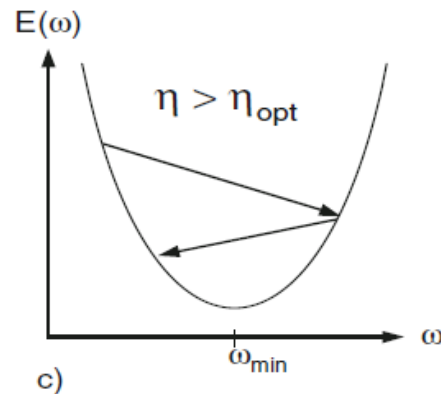
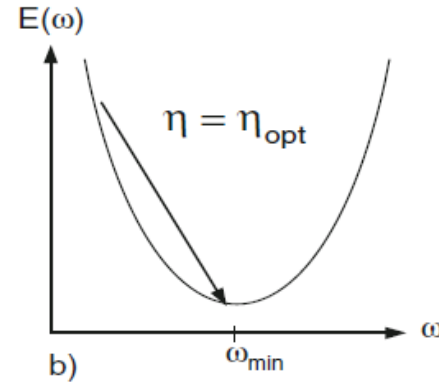
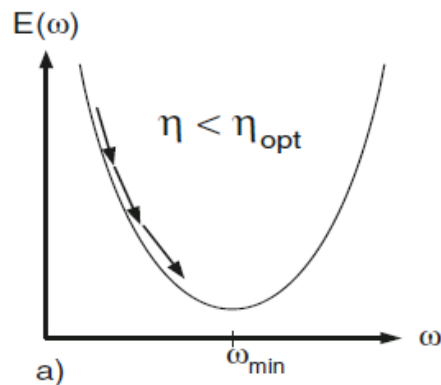
- We can arrive at the optimum in a single step using the optimum step size

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

# With non-optimal step size

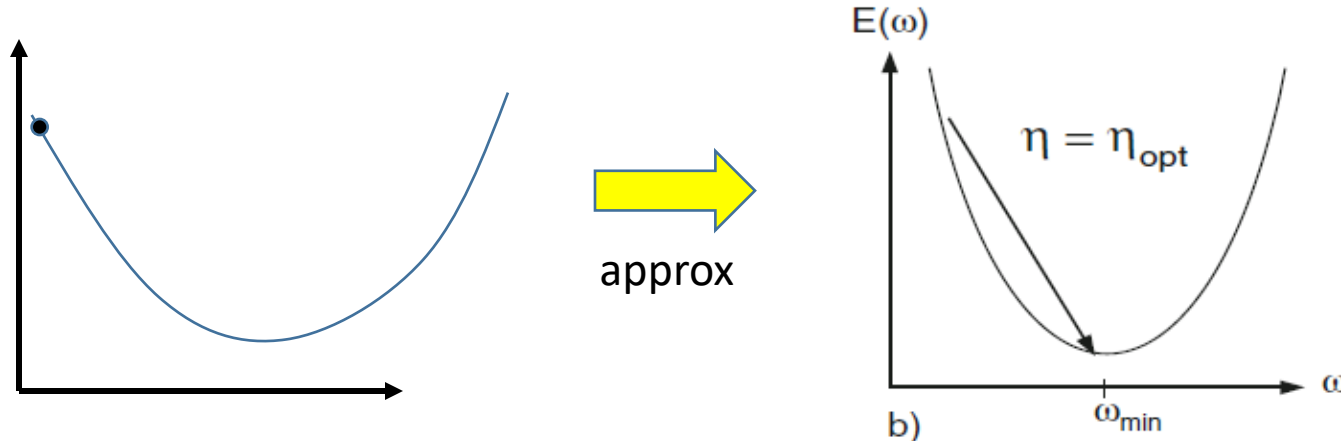
$$w^{(k+1)} = w^{(k)} - \eta \frac{dE(w^{(k)})}{dw}$$

Gradient descent with fixed step size  $\eta$  to estimate scalar parameter  $w$



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

# For generic differentiable convex objectives



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

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

– Taylor expansion

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

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

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

# For functions of *multivariate* inputs

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

- Consider a simple quadratic convex (paraboloid) function

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

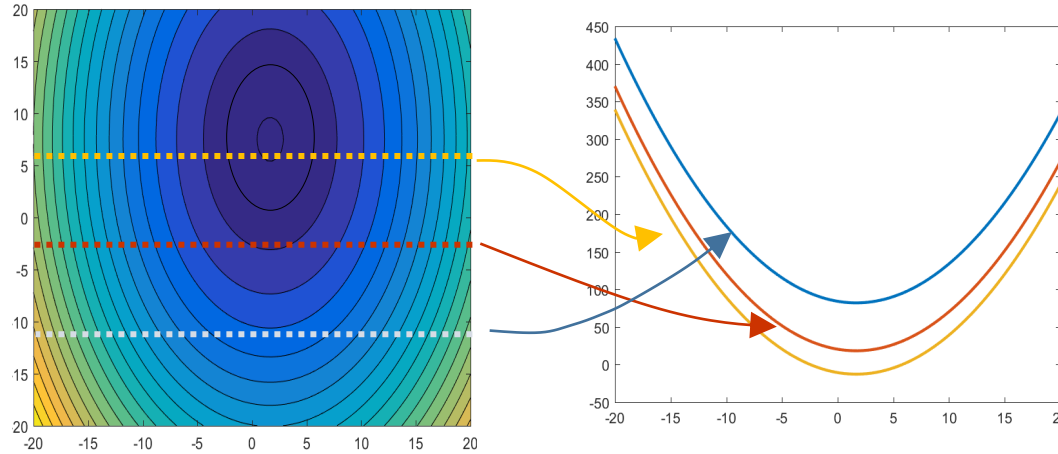
- Since  $E^T = E$  ( $E$  is scalar),  $\mathbf{A}$  can always be made symmetric
  - For **convex**  $E$ ,  $\mathbf{A}$  is always positive definite (has positive eigenvalues)

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

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

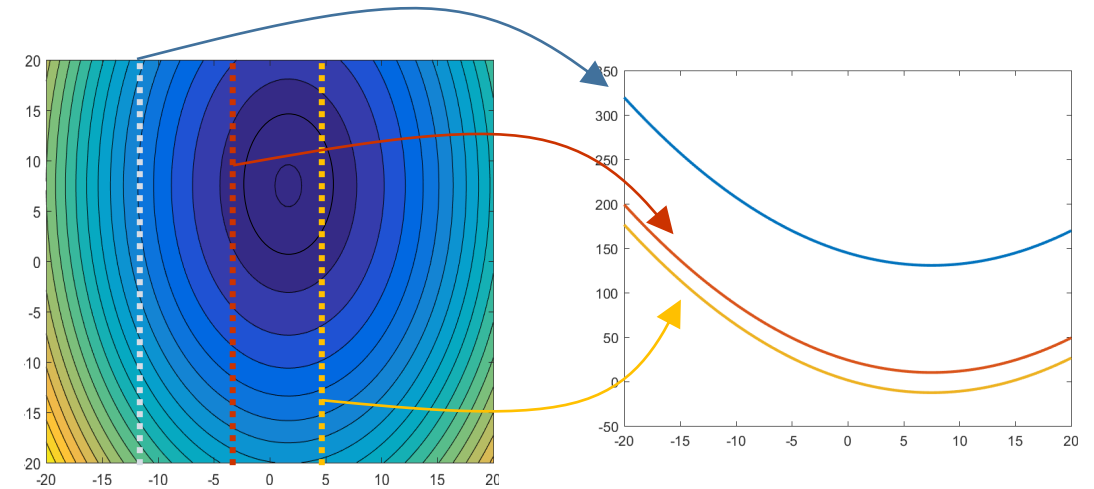
- The  $w_i$ s are *uncoupled*
- For *convex* (paraboloid)  $E$ , the  $a_{ii}$  values are all positive
- Just an sum of  $N$  independent quadratic functions

# “Descents” are uncoupled



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

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

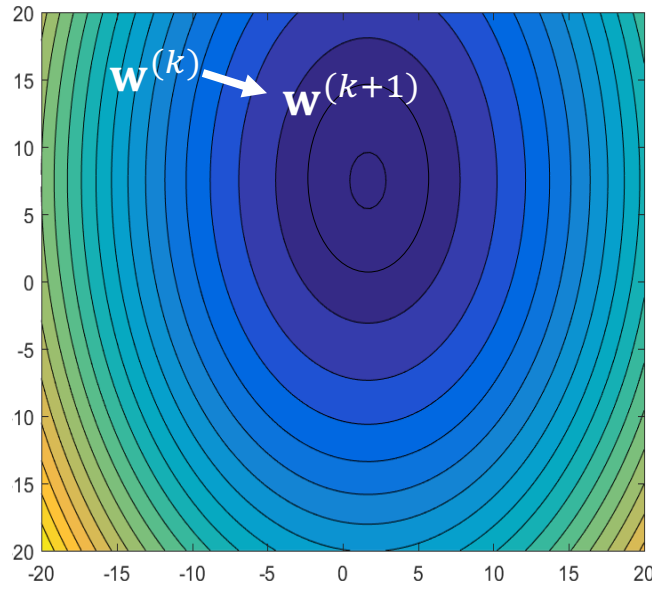


$$E = \frac{1}{2}a_{22}w_2^2 + b_2w_2 + c + C(\neg w_2)$$

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

- 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



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

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

- 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

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

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

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

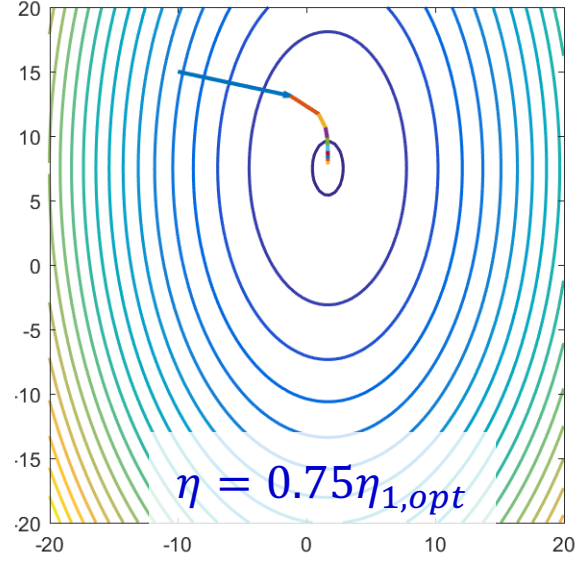
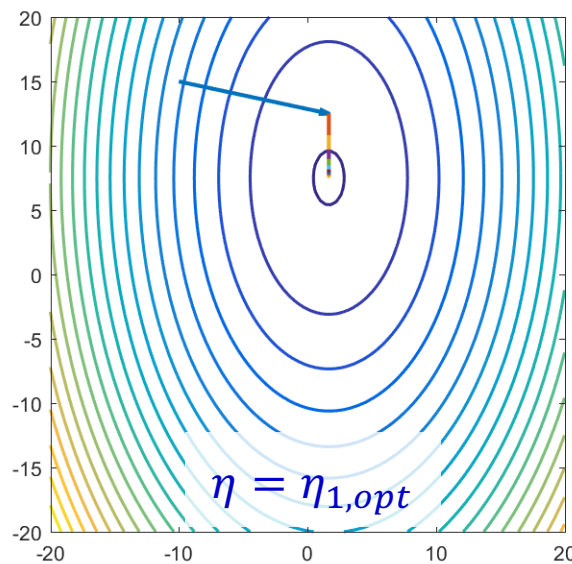
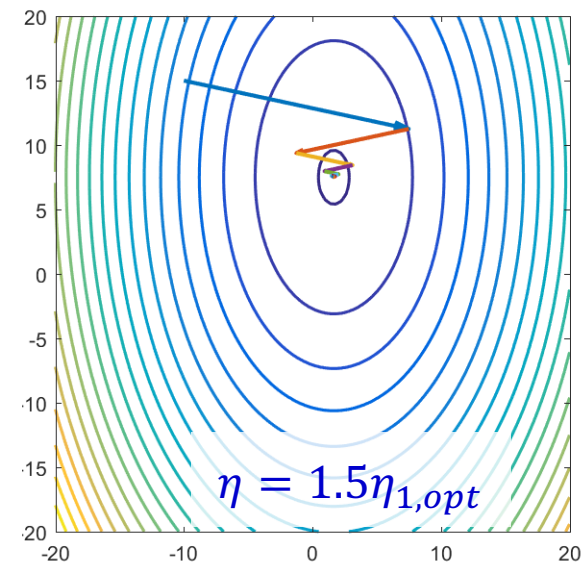
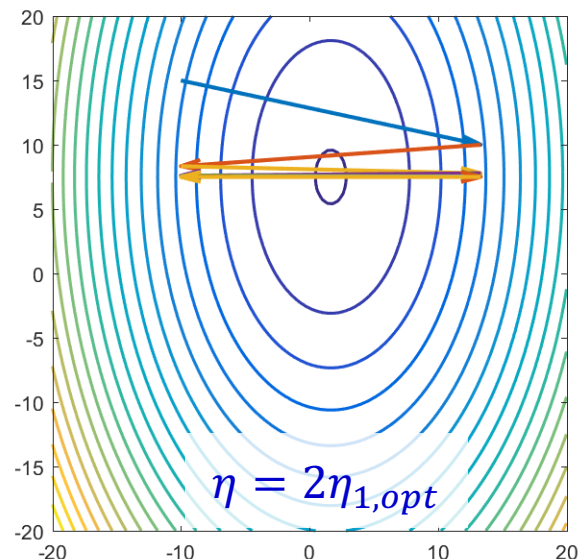
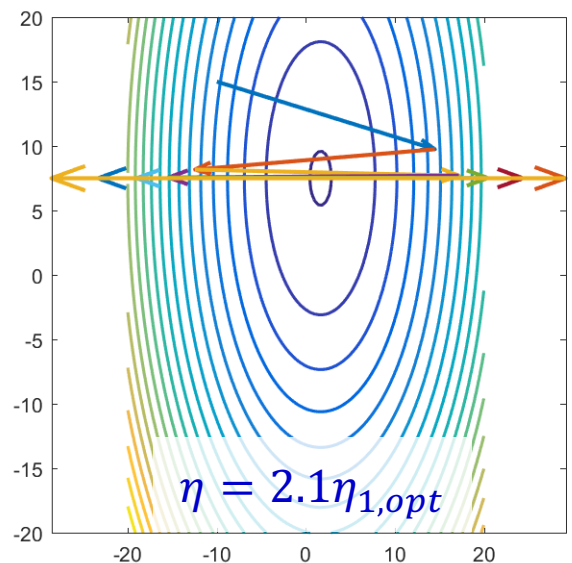
- The learning rate must be lower than twice the *smallest* optimal learning rate for any component

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

- 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

$$\eta_{1,opt} = 0.33;$$
$$\eta_{2,opt} = 1$$



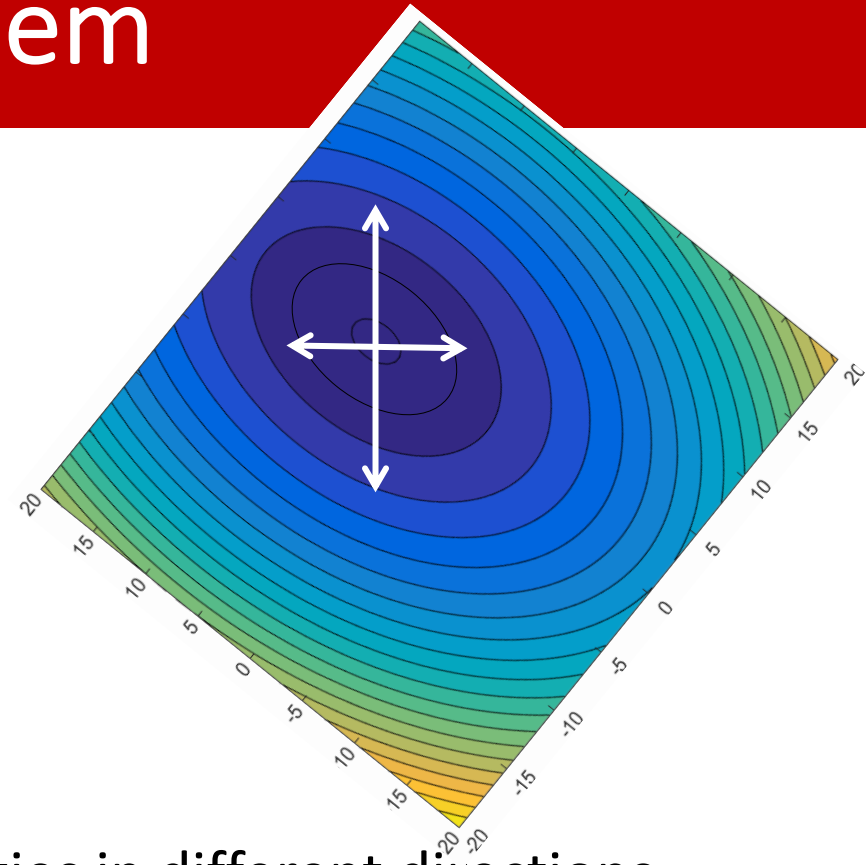
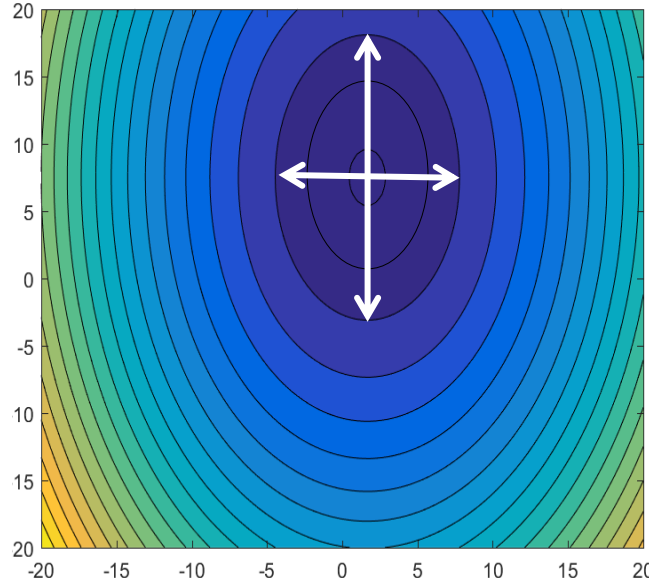
# Convergence

- Convergence behaviors become increasingly unpredictable as dimensions increase
- For the fastest convergence, ideally, the learning rate  $\eta$  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

# The reason for the problem

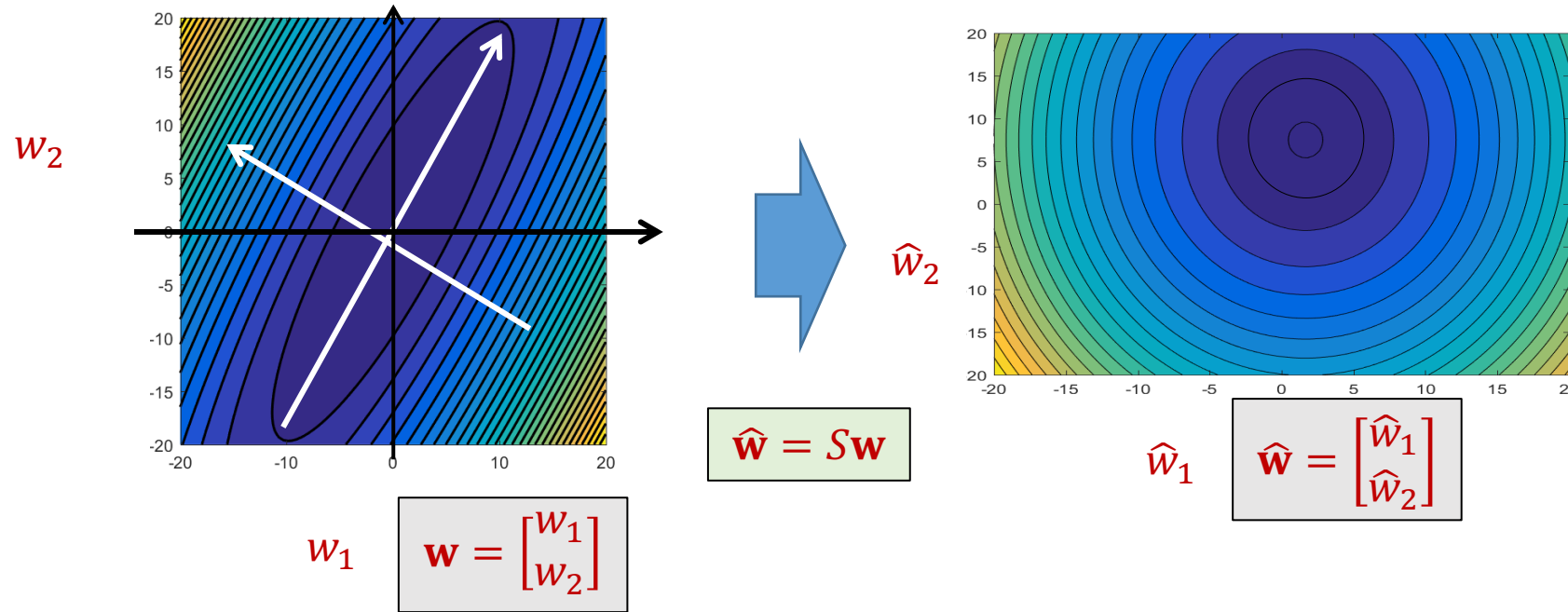
$$E = \frac{1}{2} \mathbf{w}^T \mathbf{A} \mathbf{w} + \mathbf{w}^T \mathbf{b} + c$$

Diagonal **A**



- 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: Scale and rotate 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 and rotating the axes

- Original equation:

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

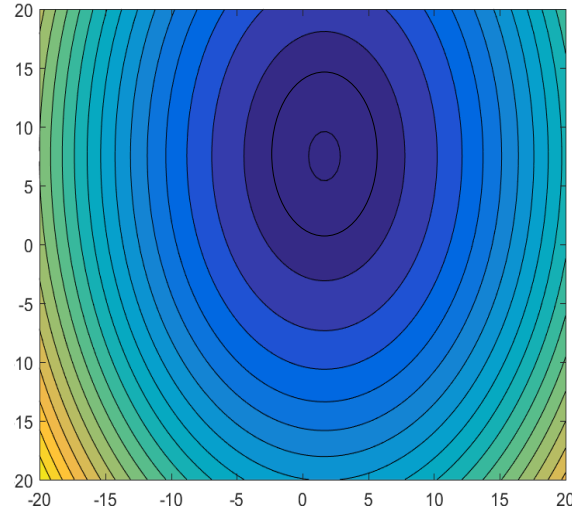
- We want to find a matrix  $\mathbf{S}$  such that

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

- And

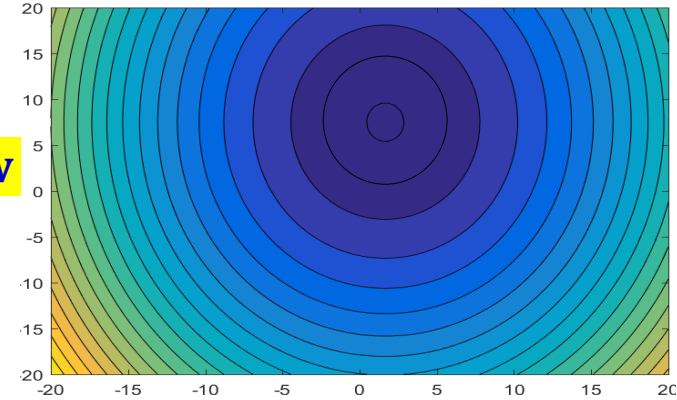
$$\begin{aligned} E &= \frac{1}{2} \hat{\mathbf{w}}^T \hat{\mathbf{w}} + \hat{\mathbf{b}}^T \hat{\mathbf{w}} + c \\ &= \frac{1}{2} \mathbf{w}^T \boxed{\mathbf{S}^T \mathbf{S}} \mathbf{w} + \boxed{\hat{\mathbf{b}}^T \mathbf{S}} \mathbf{w} + c \\ &\quad \mathbf{A} \quad \mathbf{b}^T \end{aligned}$$

# Modified update rule



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

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



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

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

- Leads to the modified gradient descent rule

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

# Modified update rule: Proof

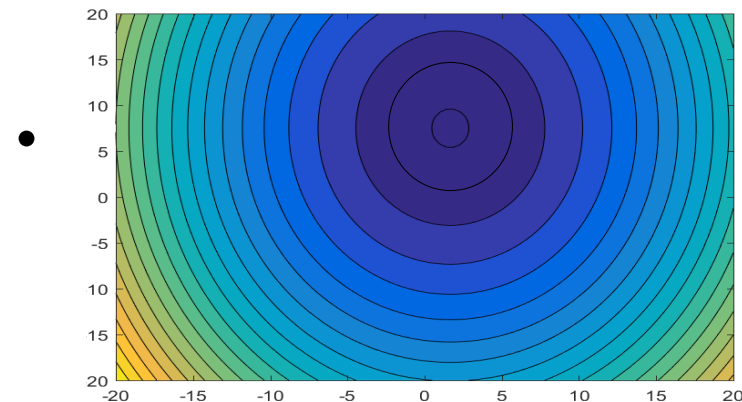
- Equating linear and quadratic coefficients, we get

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

- Solving:  $\mathbf{S} = \mathbf{A}^{0.5}$ ,  $\hat{\mathbf{b}} = \mathbf{A}^{-0.5} \mathbf{b}$
- For *any* positive definite  $\mathbf{A}$ , we can write:  $\mathbf{A} = \mathbf{E} \mathbf{\Lambda} \mathbf{E}^T$ 
  - Eigen decomposition
  - $\mathbf{E}$  is an orthogonal matrix
  - $\mathbf{\Lambda}$  is a diagonal matrix of non-zero diagonal entries
- Defining  $\mathbf{A}^{0.5} = \mathbf{E} \mathbf{\Lambda}^{0.5} \mathbf{E}^T$
- Defining  $\mathbf{A}^{-0.5} = \mathbf{E} \mathbf{\Lambda}^{-0.5} \mathbf{E}^T$



# Modified update rule: Proof



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

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

- Gradient descent rule:

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

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

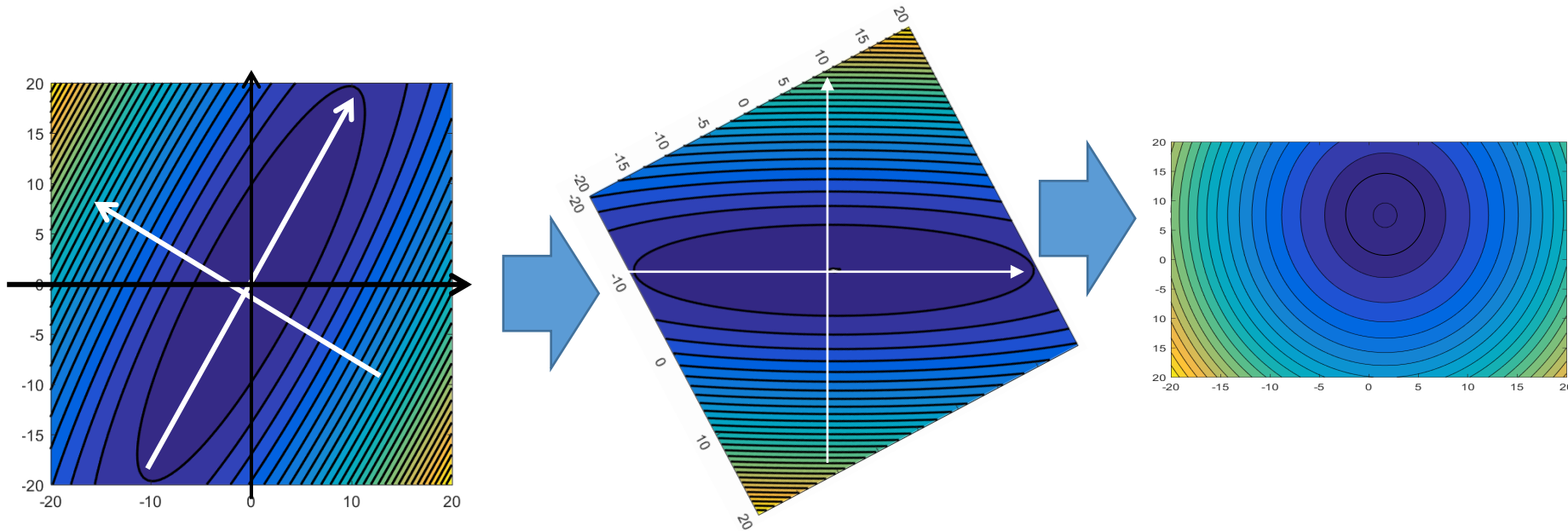
$$- \nabla_{\hat{\mathbf{w}}} E(\hat{\mathbf{w}}) = \mathbf{A}^{-0.5} \nabla_{\mathbf{w}} E(\mathbf{w})$$

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

$$\mathbf{A}^{0.5} \mathbf{w}^{(k+1)} = \mathbf{A}^{0.5} \mathbf{w}^{(k)} - \eta \mathbf{A}^{-0.5} \nabla_{\mathbf{w}} E(\mathbf{w}^{(k)})$$

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

# 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:

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

# Optimal update

**Indeed, for a quadratic function**

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

- Using the same logic as before, we get the normalized update rule

$$\mathbf{w}_{\min} = \mathbf{w}^{(k)} - H_E(\mathbf{w}^{(k)})^{-1} \nabla_{\mathbf{w}} E(\mathbf{w}^{(k)})$$

# Quadratic function: Properties

- Given  $\nabla E(\mathbf{w}) = 0$  at  $\mathbf{w} = \mathbf{w}^*$  and  $\mathbf{H}$  shows Hessian or  $\nabla\nabla E(\mathbf{w})$  at  $\mathbf{w} = \mathbf{w}^*$ :

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T \mathbf{H}(\mathbf{w} - \mathbf{w}^*)$$

- $\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i$  where  $\mathbf{u}_i$ s form a complete orthonormal set  $\mathbf{u}_i^T \mathbf{u}_j = \delta_{ij}$

$$\mathbf{w} - \mathbf{w}^* = \sum_i \alpha_i \mathbf{u}_i$$

- Thus:

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2} \sum \lambda_i \alpha_i^2$$

- If all eigenvalues are positive/negative then  $\mathbf{w}^*$  corresponds to a min/max of  $E(\mathbf{w})$

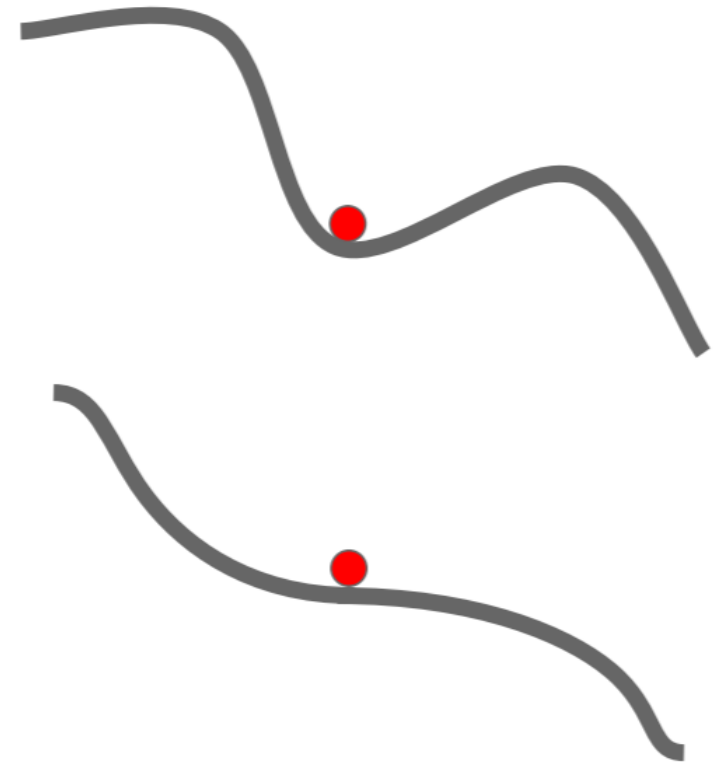
# Optimization: Problems with GD

- What if the loss function has a local minima or saddle point?
- Gradient descent algorithms often get “stuck” in saddle points

Zero gradient, gradient descent gets stuck

Saddle points much more common in high dimension

For high dimensional parameter space, most points of zero gradients are not local optima and are saddle points.

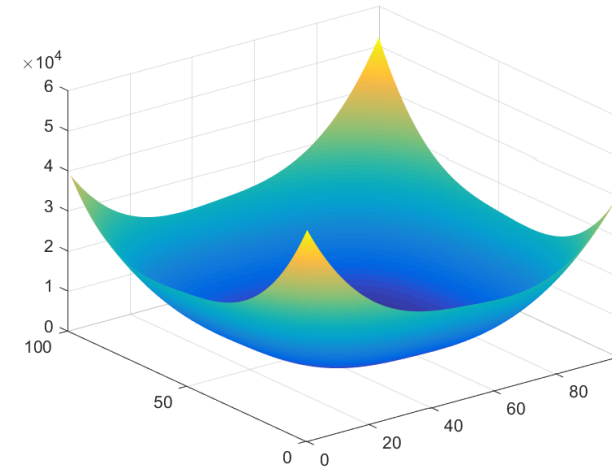
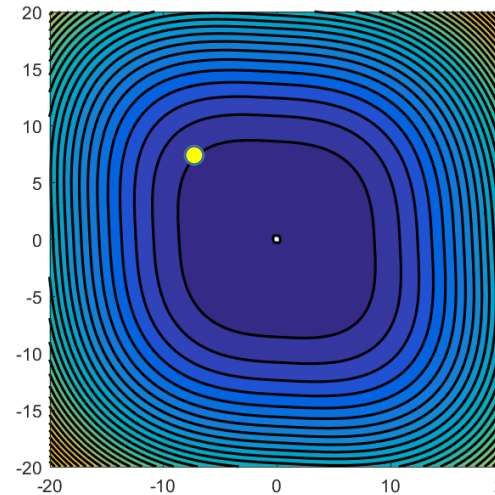


# The problem of local optima

- A function of very high dimensional space:
  - if the gradient is zero, then in each direction it can either be a convex light function or a concave light function.
  - for it to be a local minima, all directions need to be convex light altogether
    - And so the chance of that happening is maybe very small
- For large networks, the loss function may have a large number of unpleasant saddle points
  - Which backpropagation may find

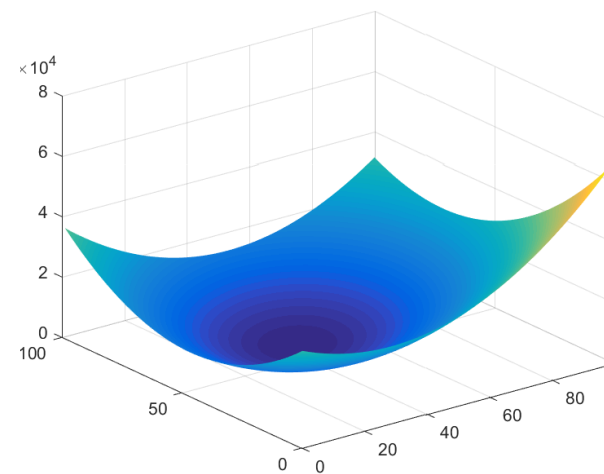
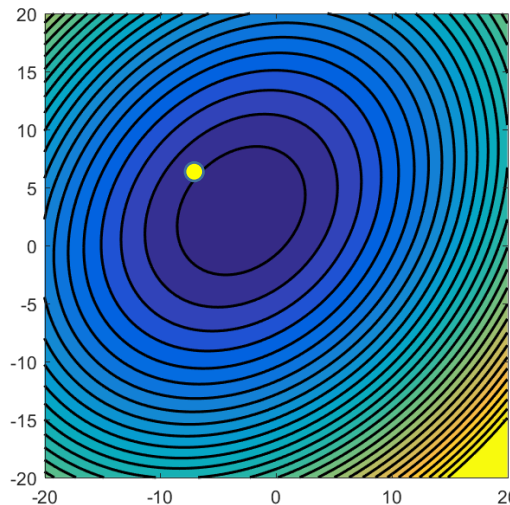
# Generic differentiable *multivariate* convex functions

- Taylor

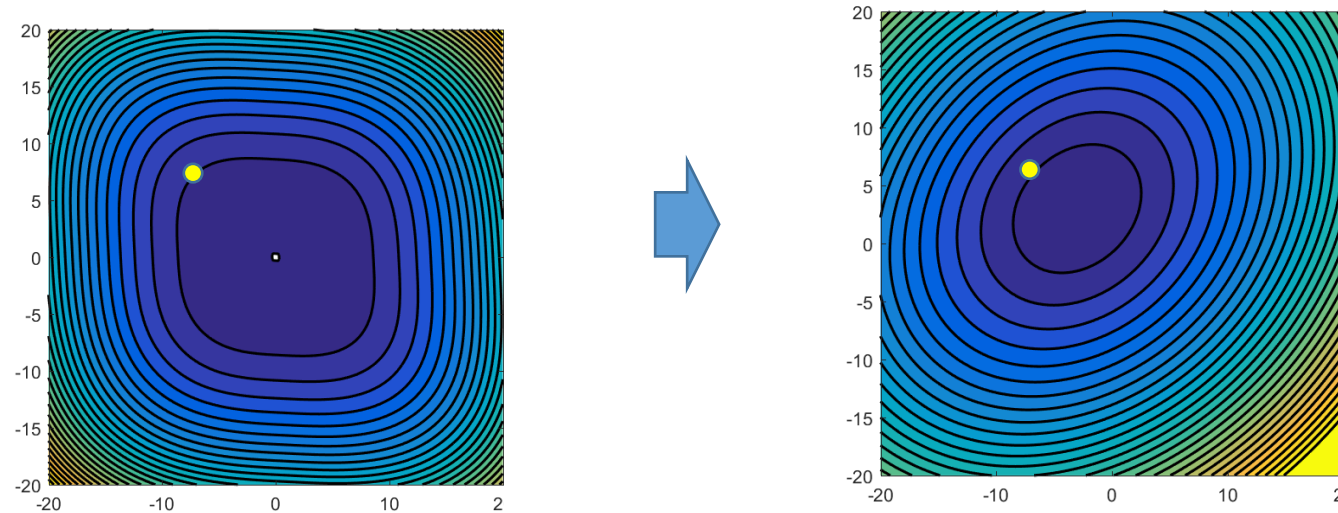


expansion

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



# Generic differentiable *multivariate* convex functions



- Taylor expansion

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

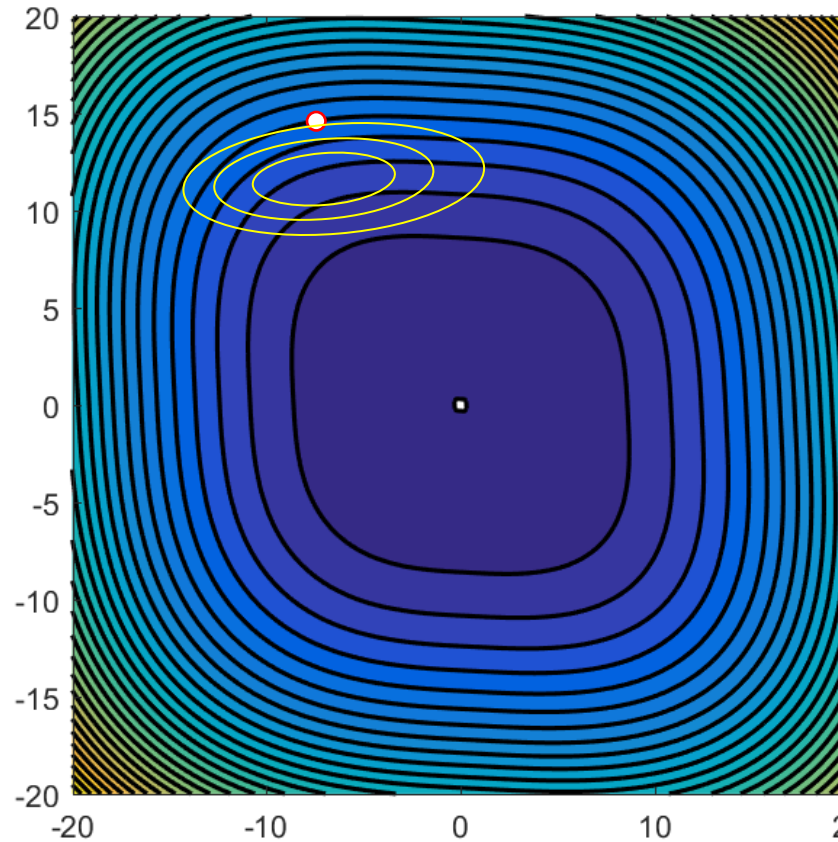
- Note that this has the form  $\frac{1}{2} \mathbf{w}^T \mathbf{A} \mathbf{w} + \mathbf{w}^T \mathbf{b} + c$
- Using the same logic as before, we get the normalized update rule

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

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



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



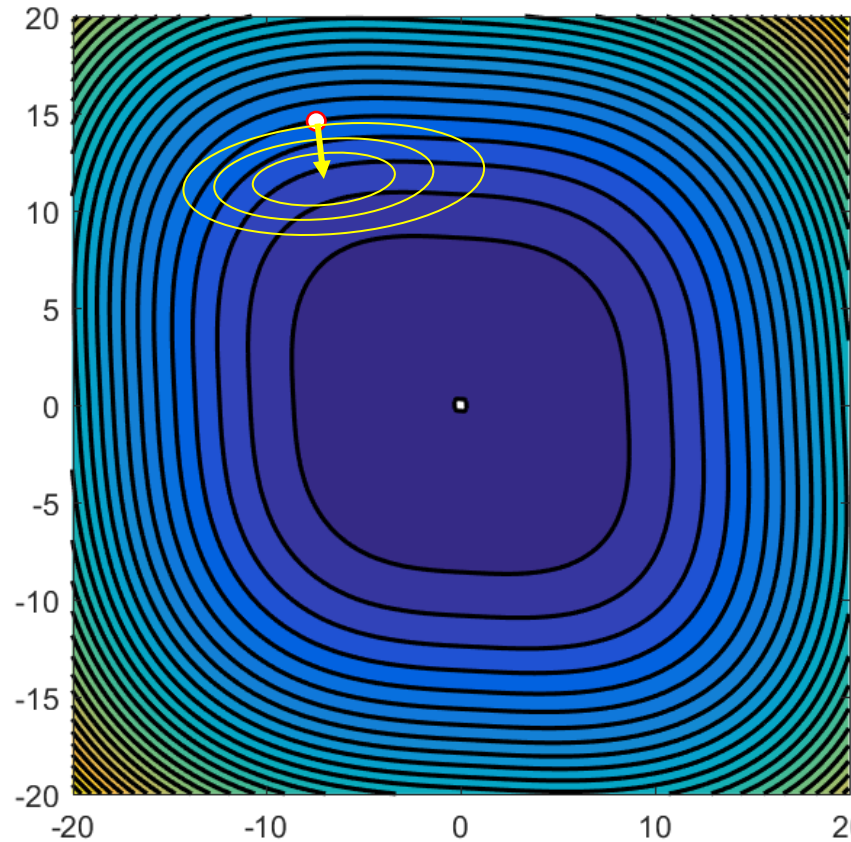
Fit a quadratic at each point and find the minimum of that quadratic

- 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 ( $\eta = 1$ )

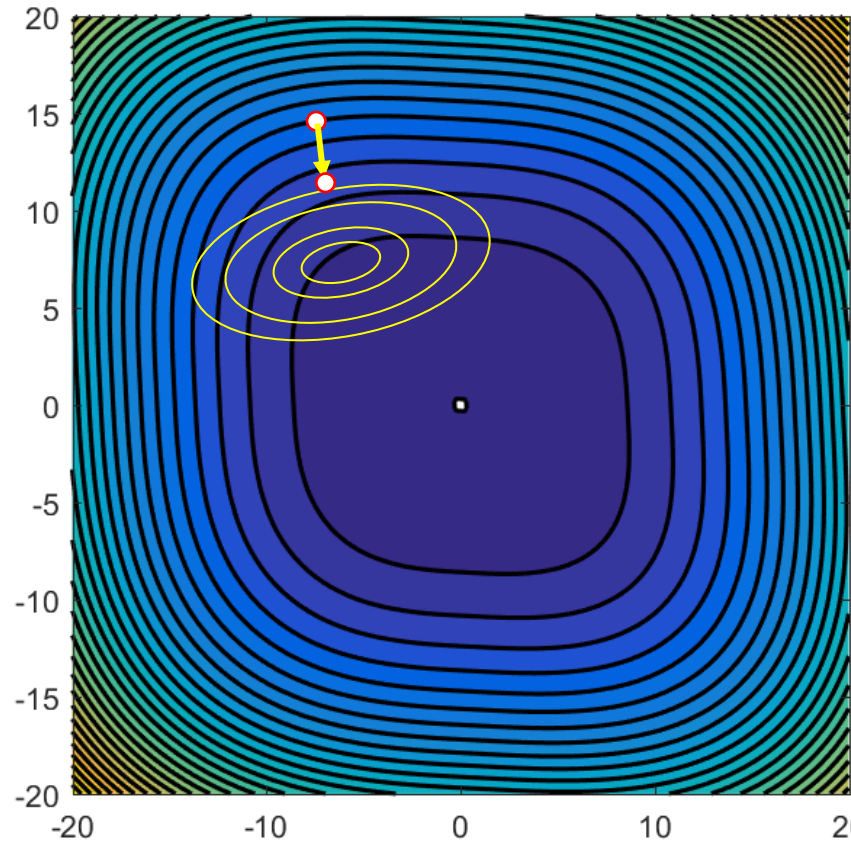


- Iterated localized optimization with quadratic approximations

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

$$- \eta = 1$$

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

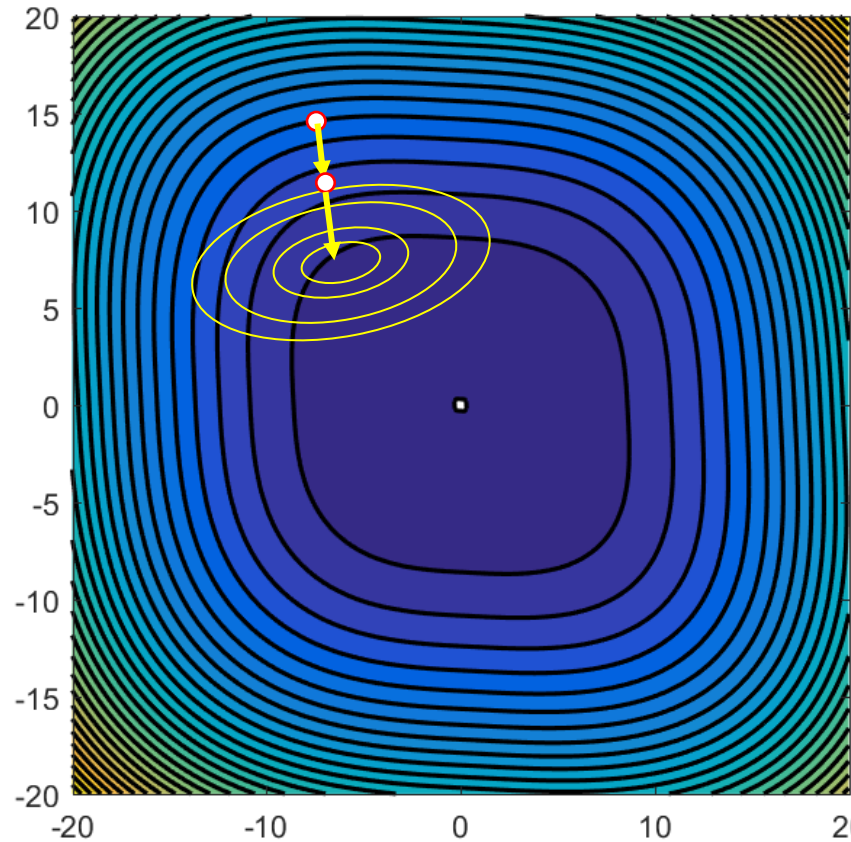


- Iterated localized optimization with quadratic approximations

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

$$- \eta = 1$$

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

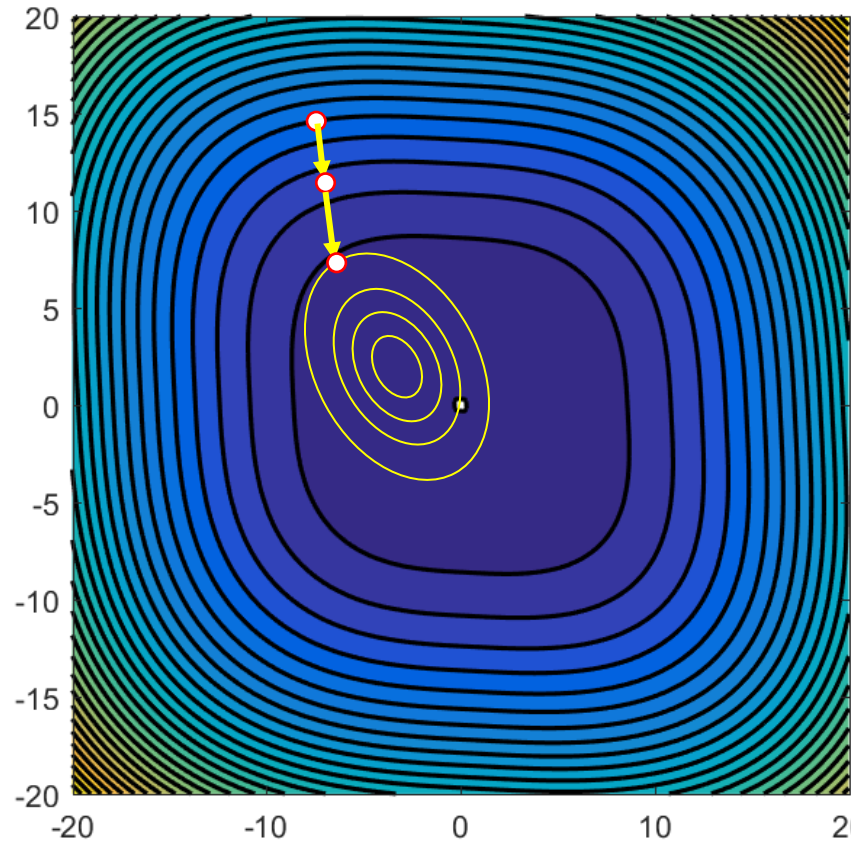


- Iterated localized optimization with quadratic approximations

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

$$- \eta = 1$$

# Minimization by Newton's method

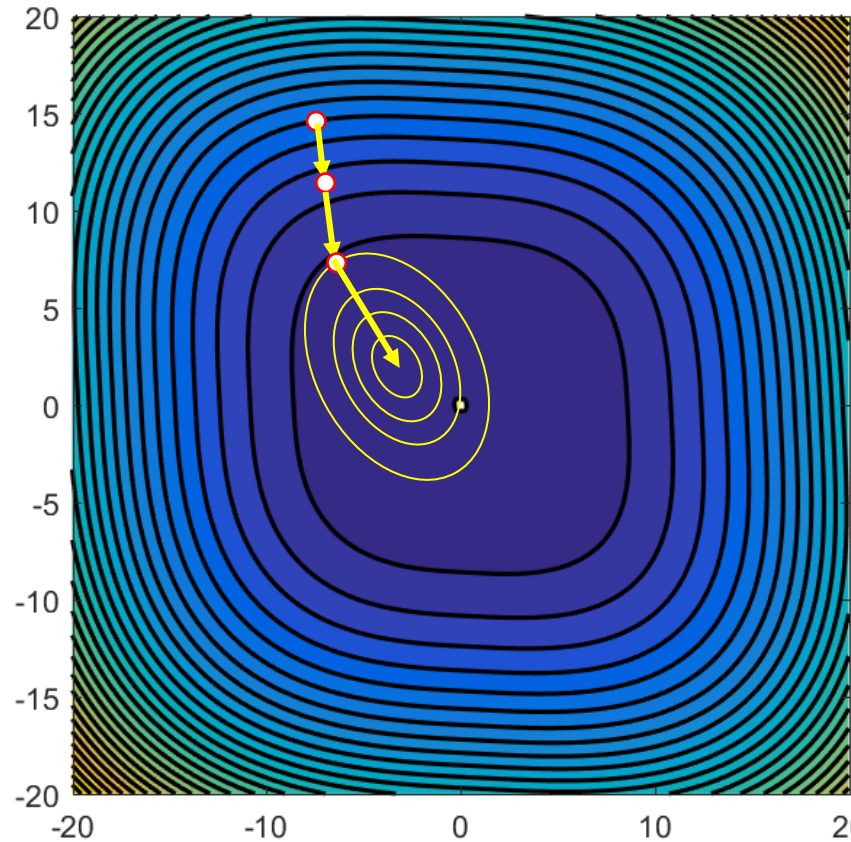


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

# Minimization by Newton's method



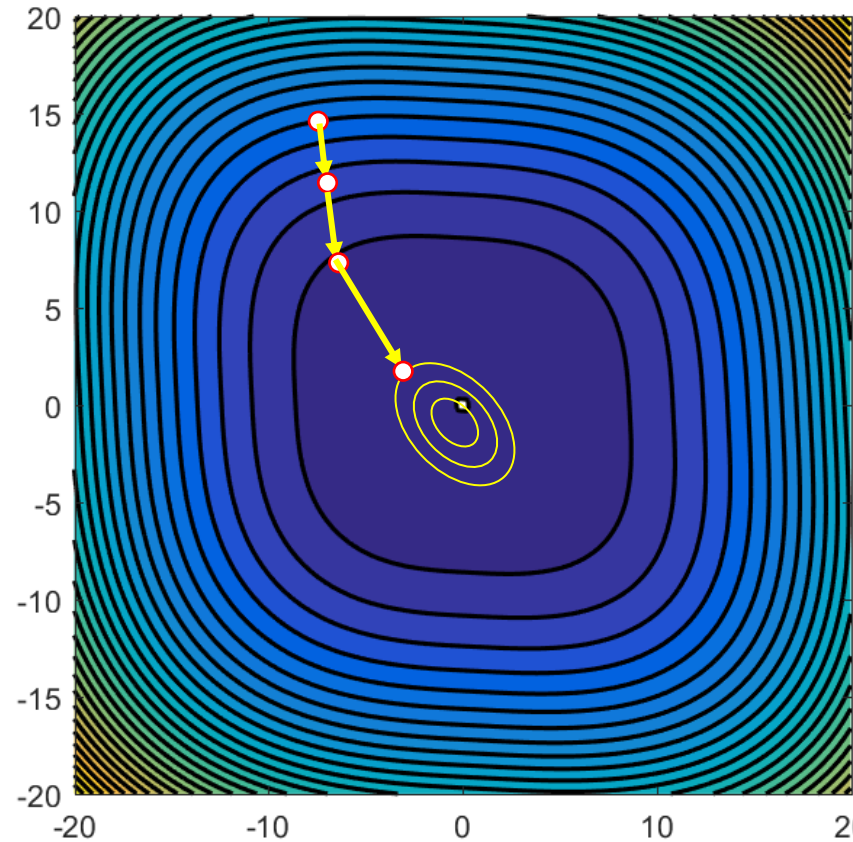
- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$



# Minimization by Newton's method

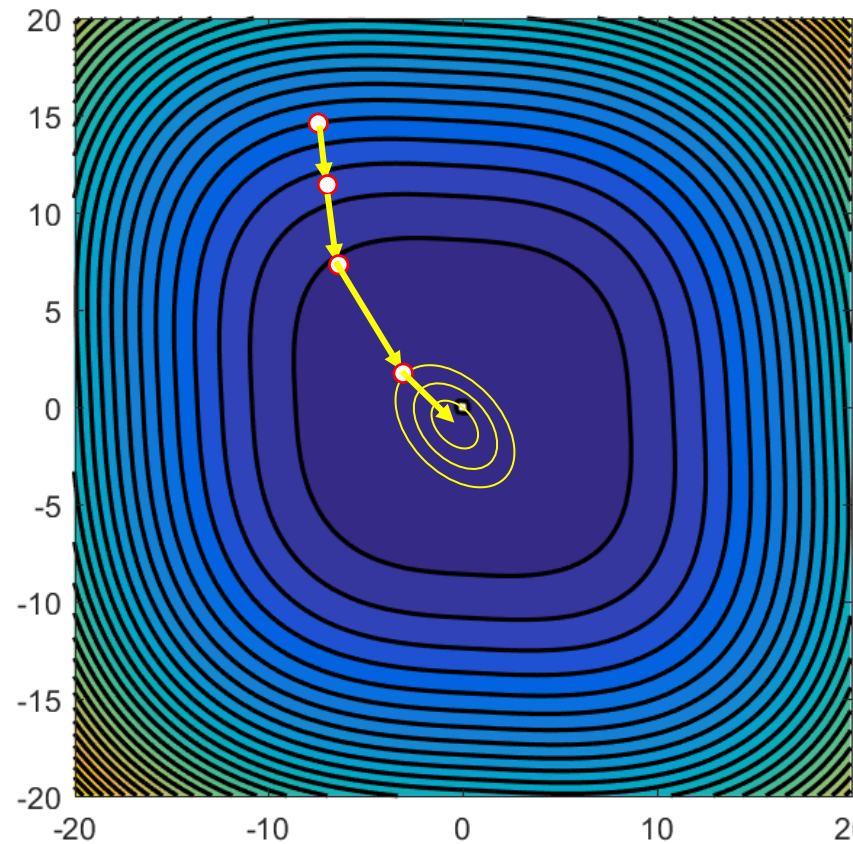


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

# Minimization by Newton's method



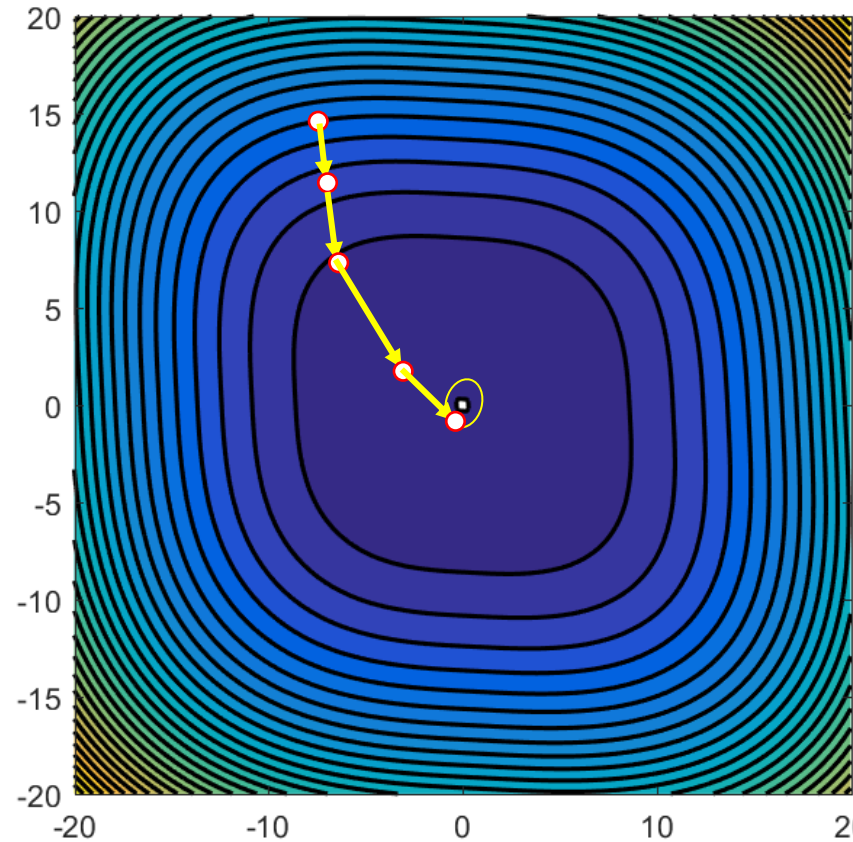
- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$



# Minimization by Newton's method

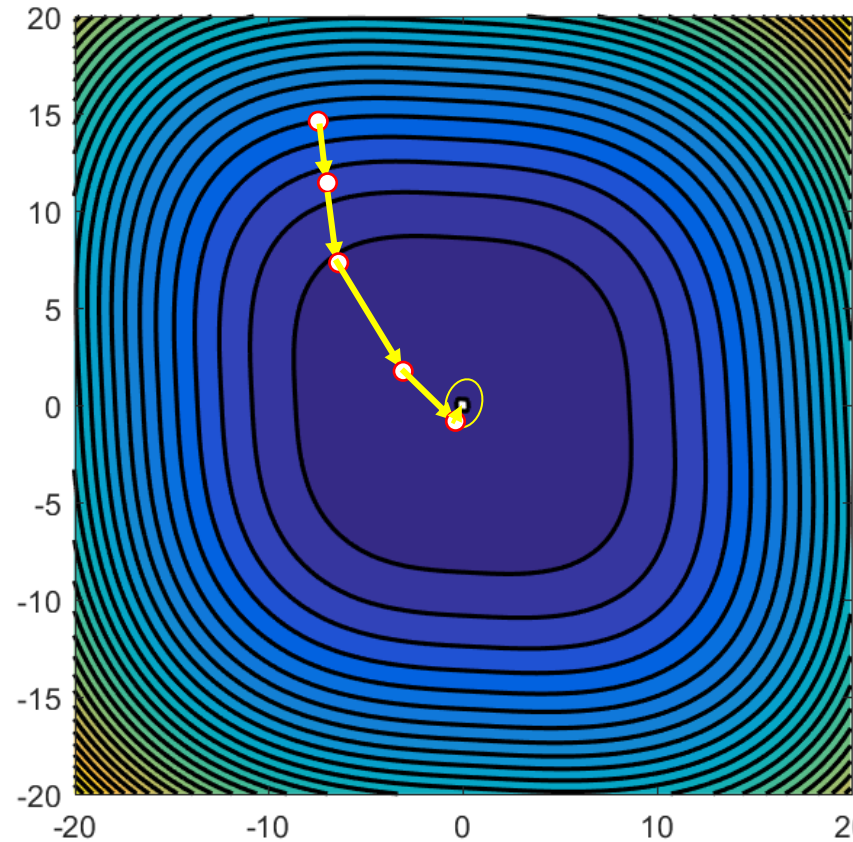


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

# Minimization by Newton's method

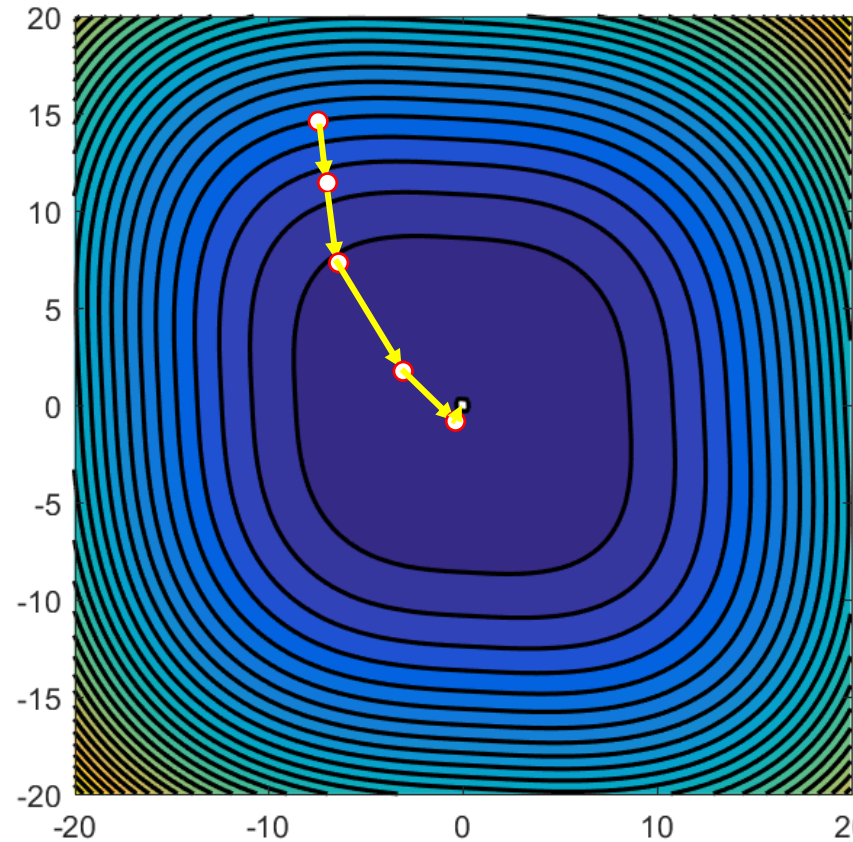


- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

# Minimization by Newton's method



- Iterated localized optimization with quadratic approximations

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

$$-\eta = 1$$

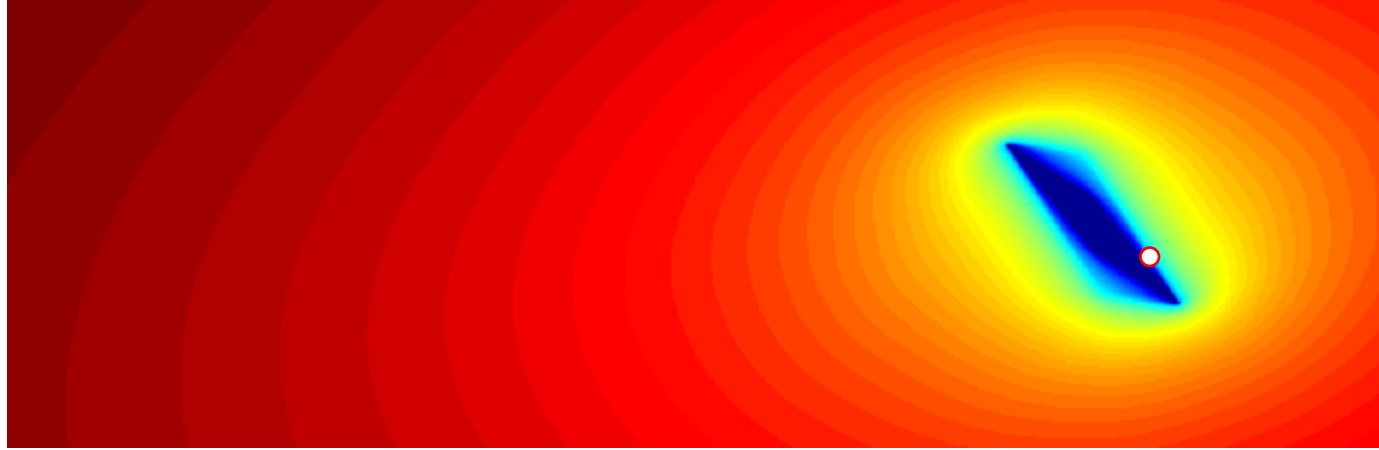
# Issues: 1. The Hessian

- Normalized update rule

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

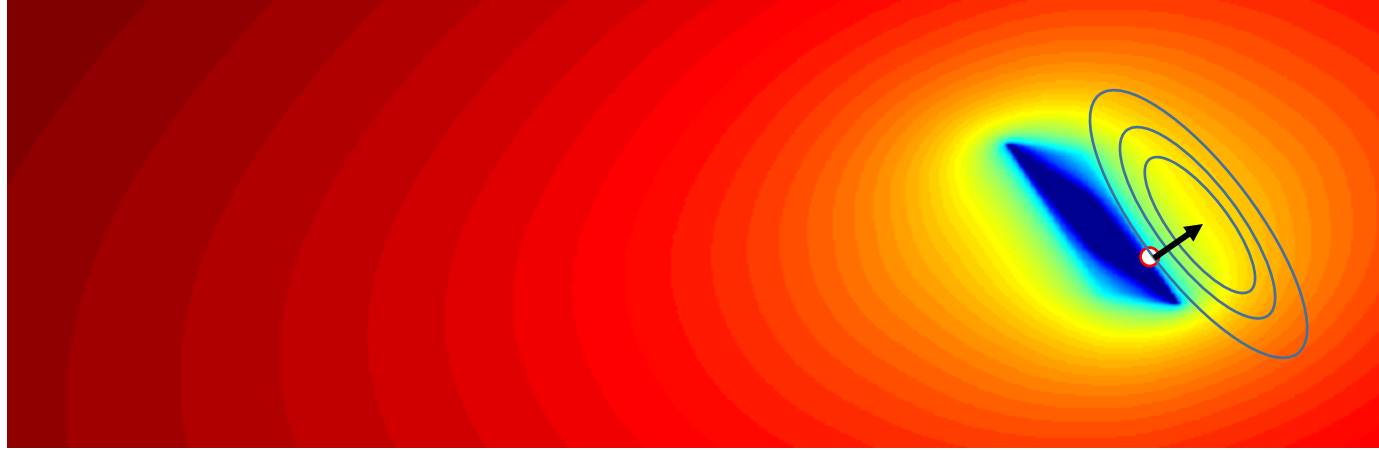
- For complex models such as neural networks, with a very large number of parameters,  $H_E(\mathbf{w}^{(k)})$  is extremely difficult to compute
  - For a network with only 100,000 parameters, will have  $10^{10}$  elements
  - And its even harder to invert, since it will be enormous

# Issues: 1. The Hessian



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

# Issues: 1. The Hessian

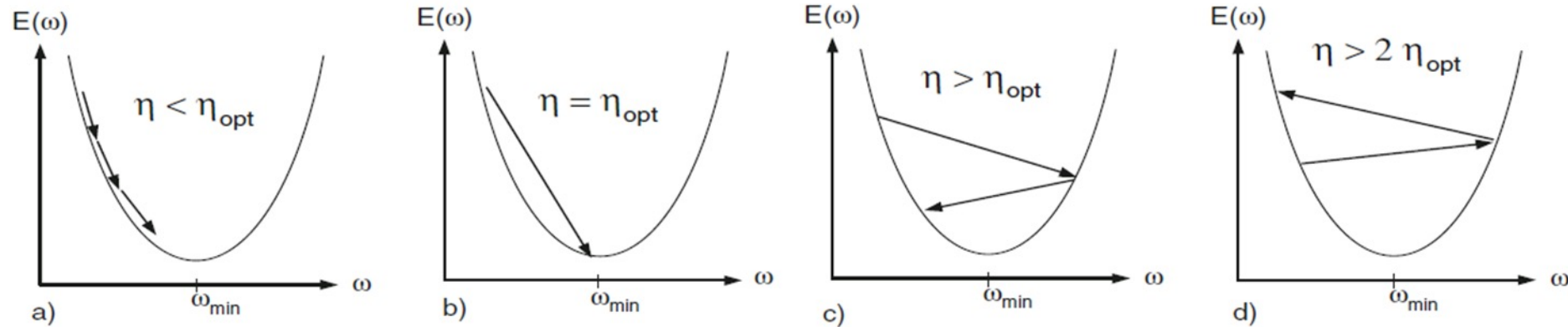


- For non-convex functions, the Hessian may not be positive semi-definite, in which case the algorithm can *diverge*
  - Goes away from, rather than towards the minimum
  - Now requires additional checks to avoid movement in directions corresponding to negative Eigenvalues of the Hessian

# 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

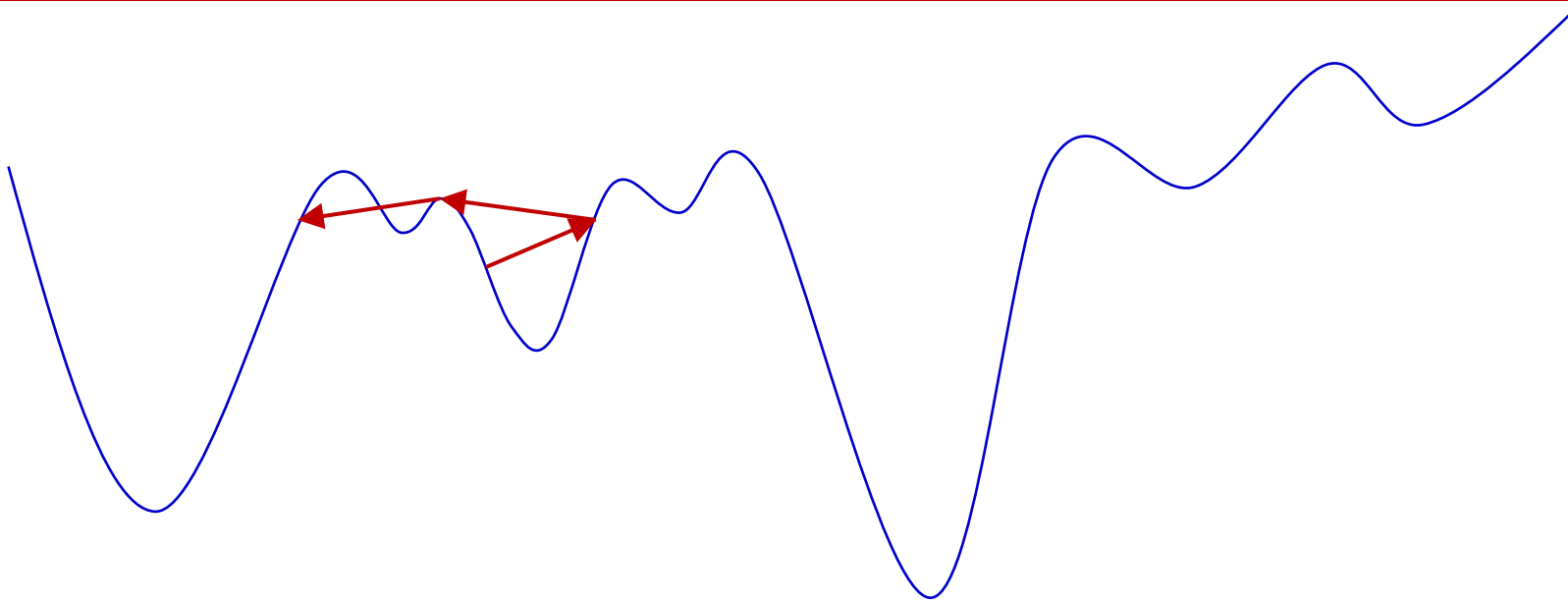


- Much of the analysis we just saw was based on trying to ensure that the step size was not so large as to cause divergence within a convex region

$$-\eta < 2\eta_{\text{opt}}$$

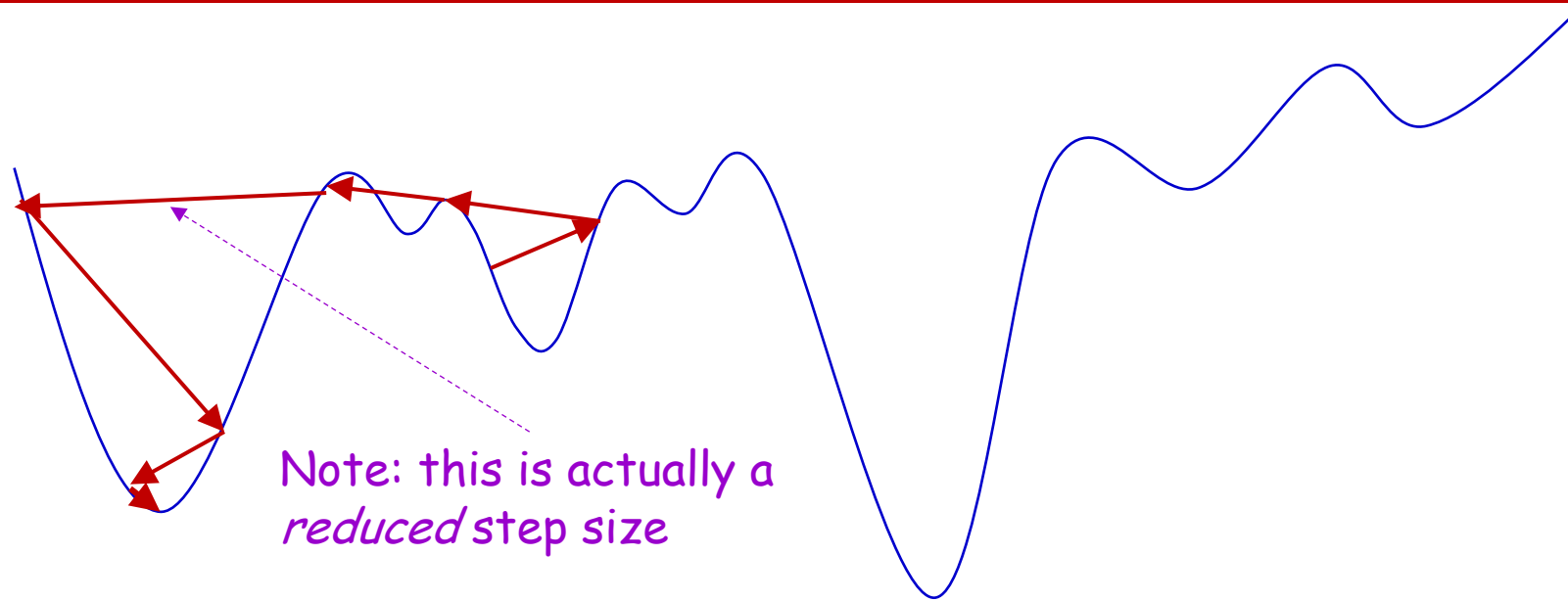


# Issues: 2. The learning rate



- 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

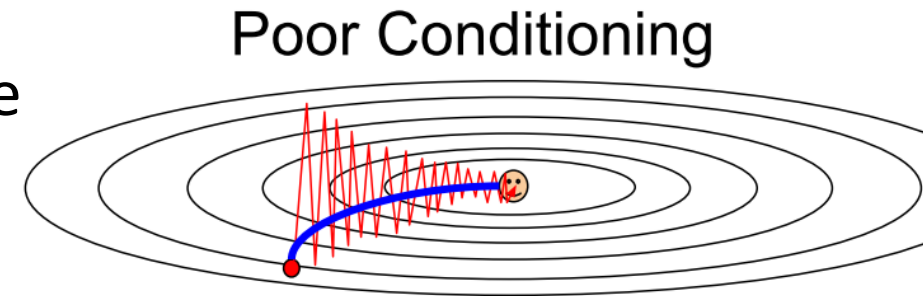
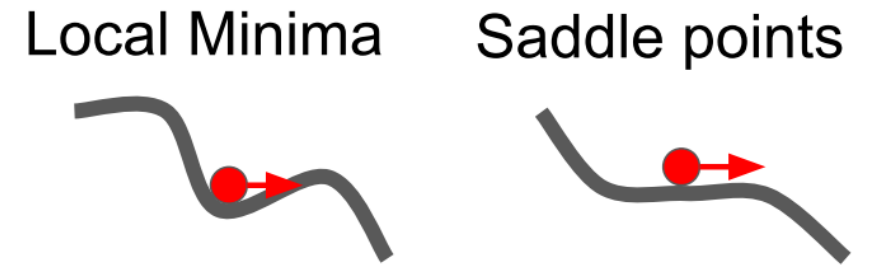
# Decaying learning rate



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

# Story so far : Convergence

- 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 be too slow or unstable due to the differences between the dimensions
- *Many of the convergence issues arise because we force the same learning rate on all parameters*



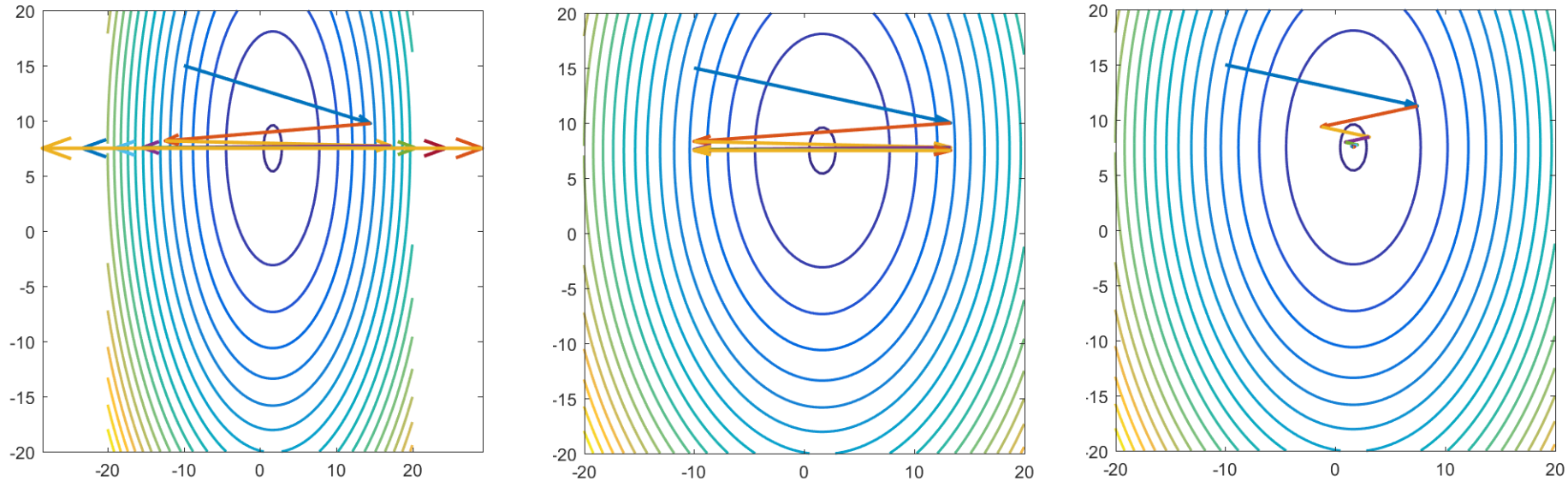
# Story so far : Second-Order Methods

- 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

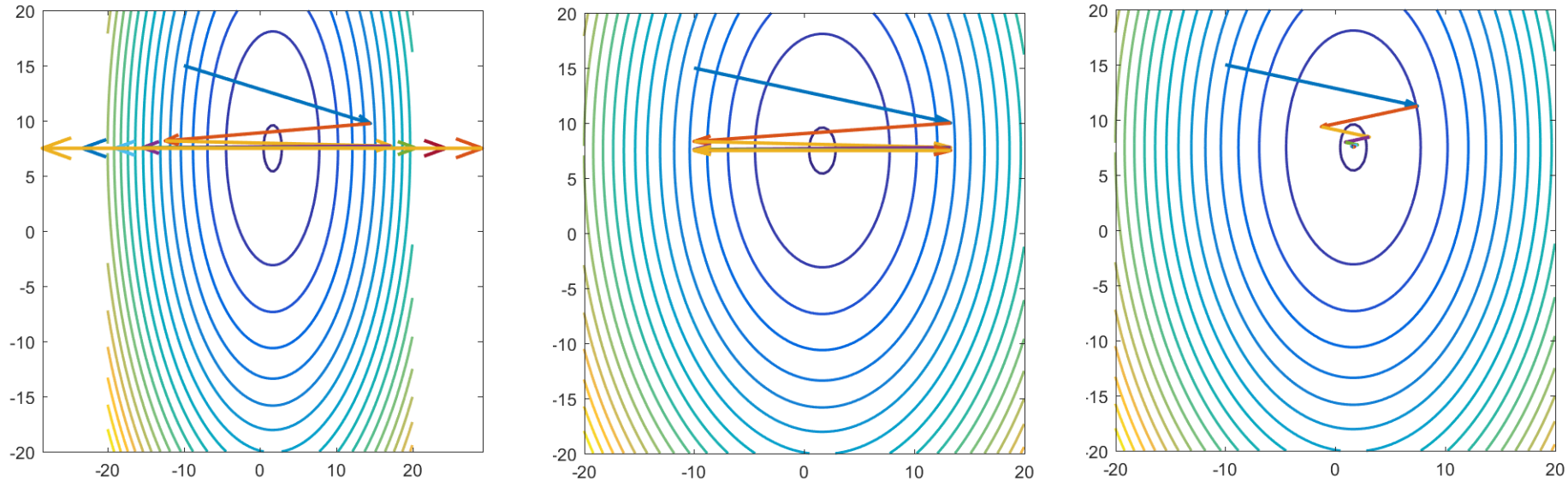
- Divergence-causing learning rates may not be a bad thing
  - Particularly for ugly loss functions
- *Decaying* learning rates provide good compromise between escaping poor local minima and convergence
- Adaptive or decaying learning rates can improve convergence

# 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



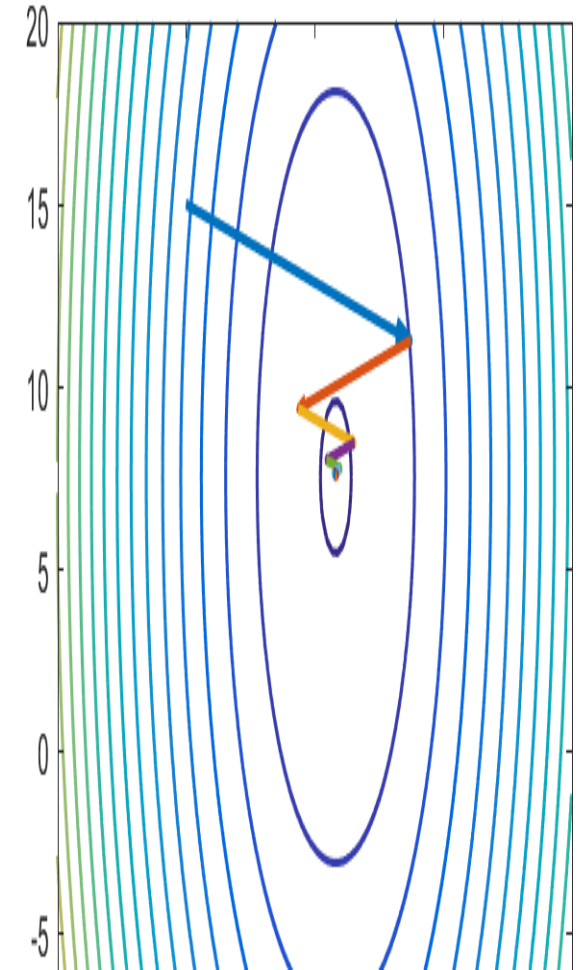
- With dimension-independent learning rates, the solution will converge smoothly in some directions, but oscillate or diverge in others

- **Proposal:**

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

# The momentum methods

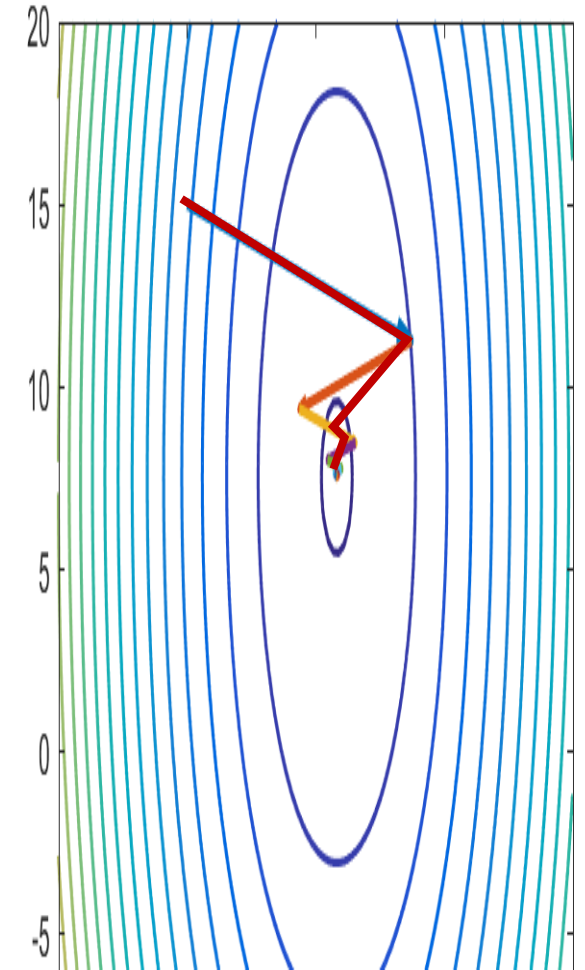
- Maintain a running average of all past steps
  - In directions in which the convergence is smooth, the average will have a large value
  - In directions in which the estimate swings, the positive and negative swings will cancel out in the average
- Update with the running average, rather than the current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



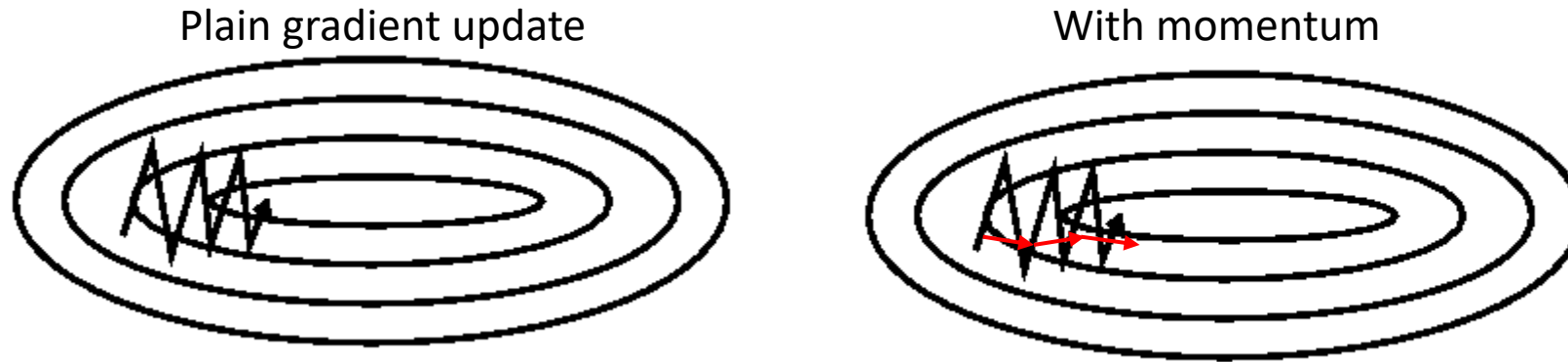


# The momentum methods

- 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

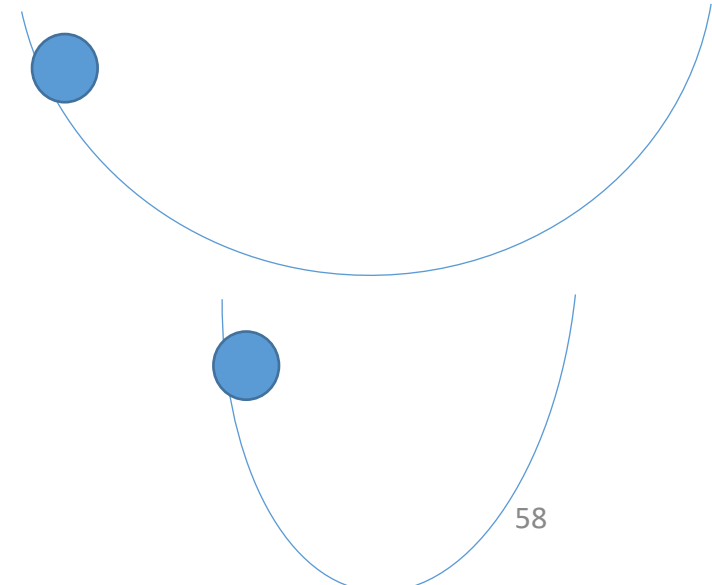


- The momentum method maintains a running average of all gradients until the *current* step

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

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

- Typical  $\beta$  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



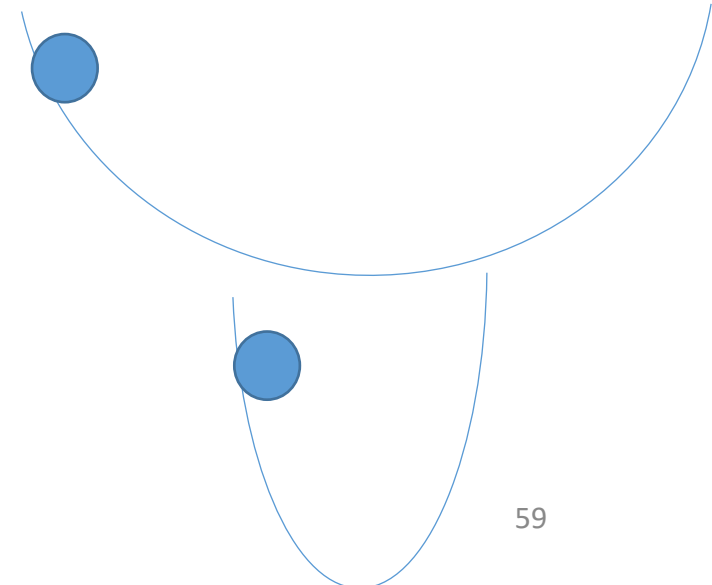
# Physical interpretation

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

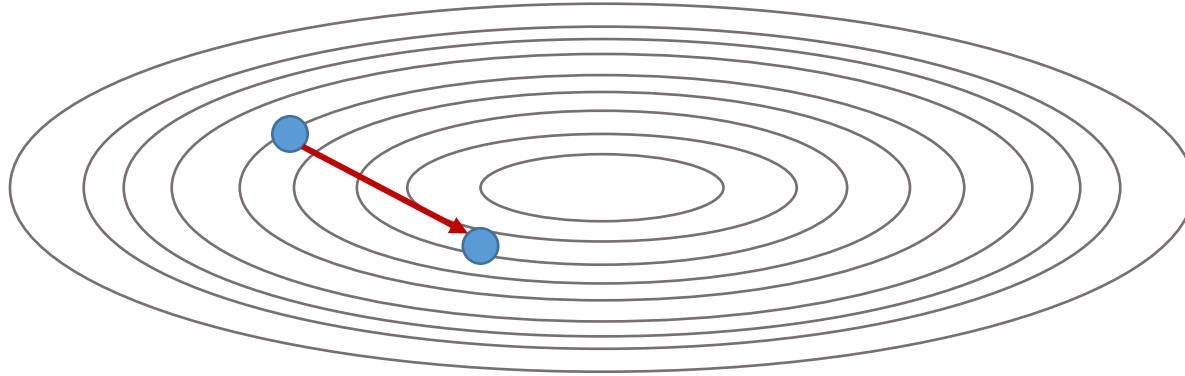
$$W^{(k)} = W^{(k-1)} - \Delta W^{(k)}$$

Velocity is built up along shallow direction

Velocity becomes damped in steep direction due to quickly changing sign



# Momentum Update

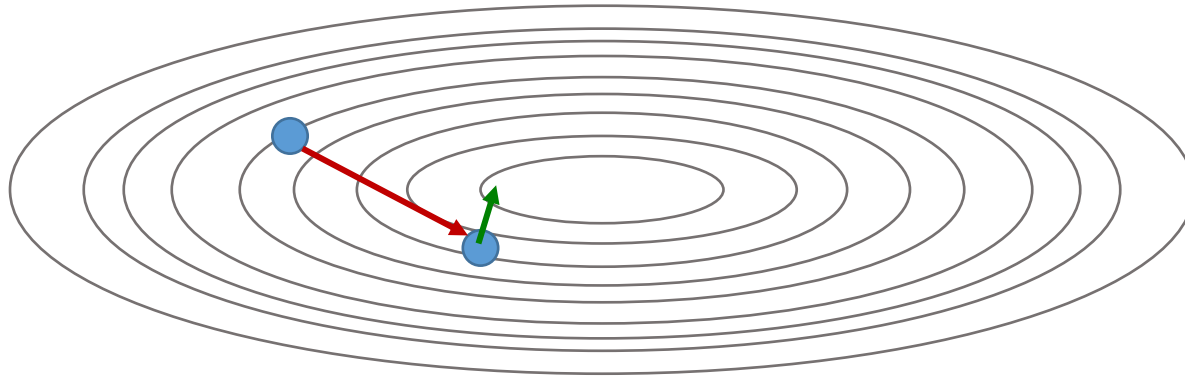


- The momentum method

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

- At any iteration, to compute the current step:

# Momentum Update

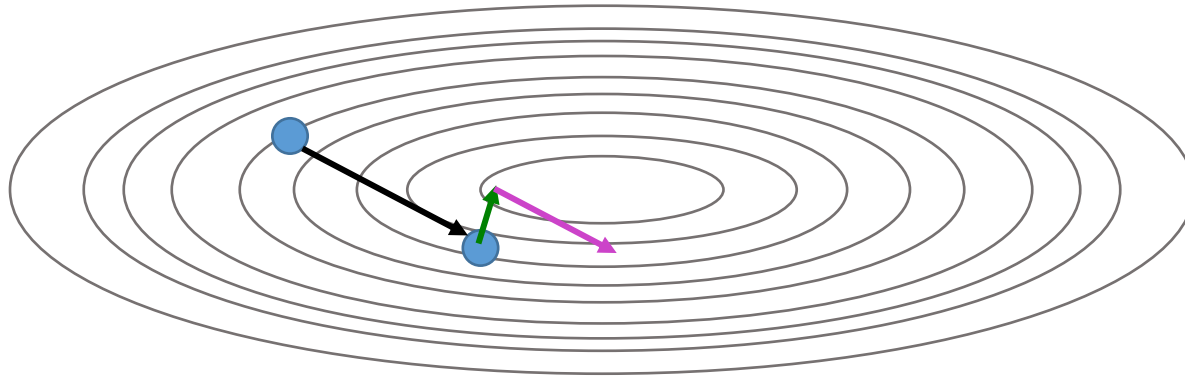


- The momentum method

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

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

# Momentum Update

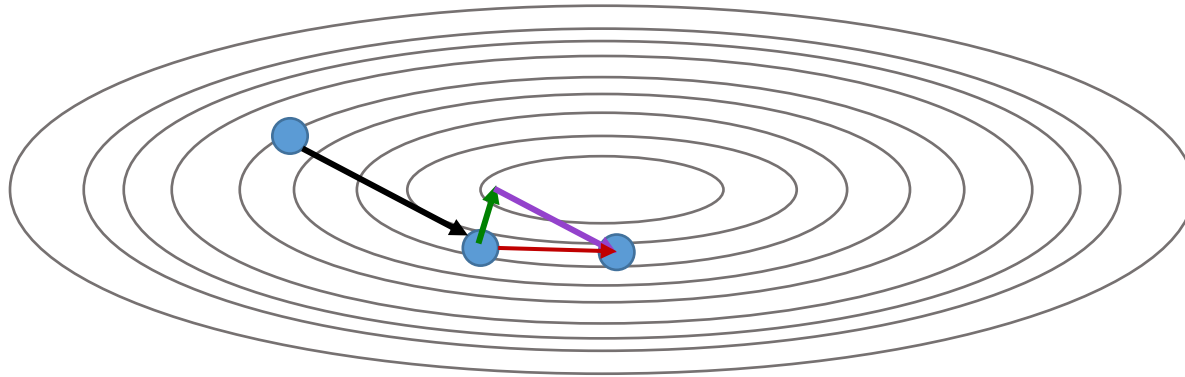


- The momentum method

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

- 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

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

- 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

# Exponentially weighted averages

- $v_0 = 0$
- $v_t = \beta v_{t-1} + a_t$

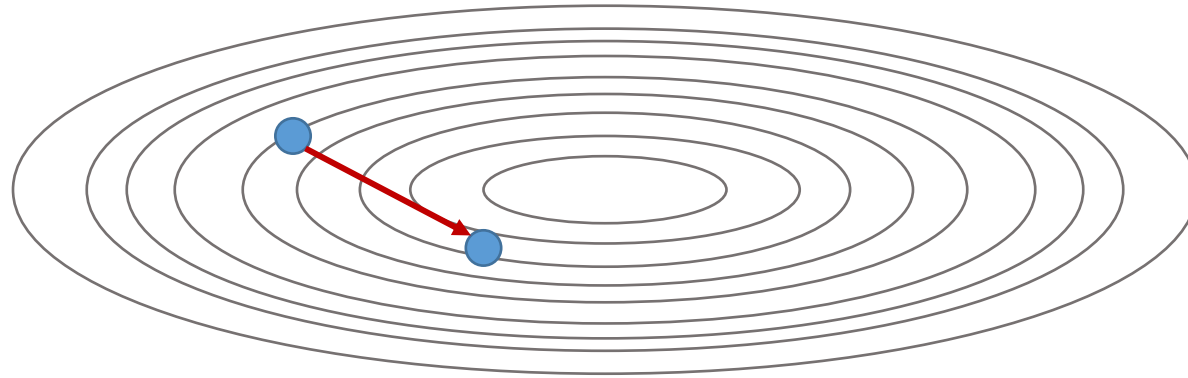
$$v_t = a_t + \beta a_{t-1} + \beta^2 a_{t-2} + \cdots + \beta^{t-1} a_1$$



# Momentum update

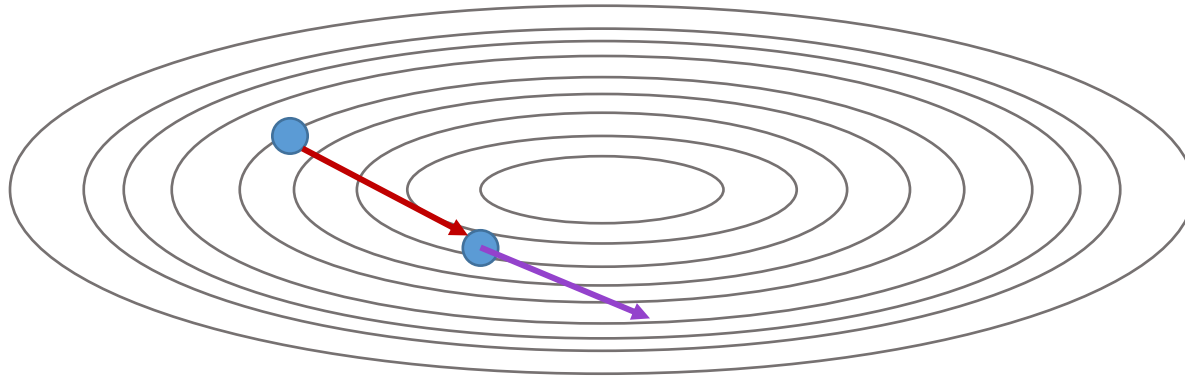
- Takes a step along the past running average *after* walking along the gradient
- The procedure can be made more optimal by reversing the order of operations..

# Nestorov's Accelerated Gradient



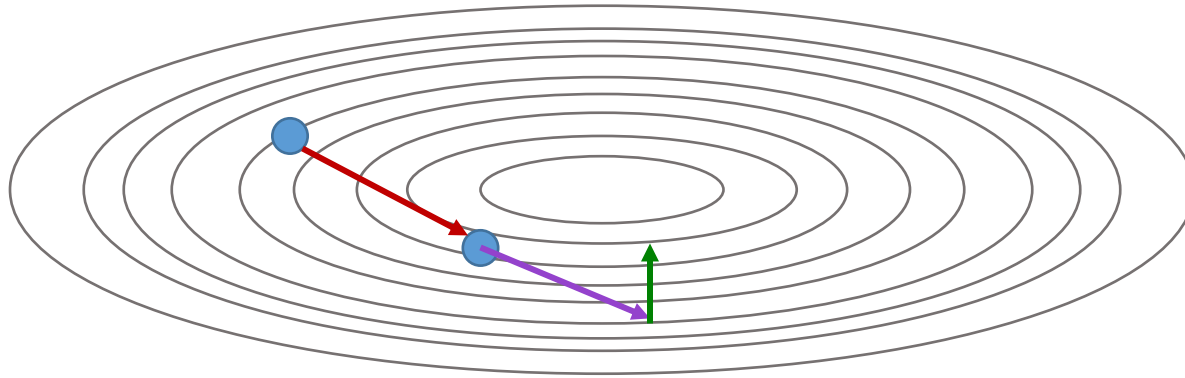
- Change the order of operations
- At any iteration, to compute the current step:

# Nestorov's Accelerated Gradient



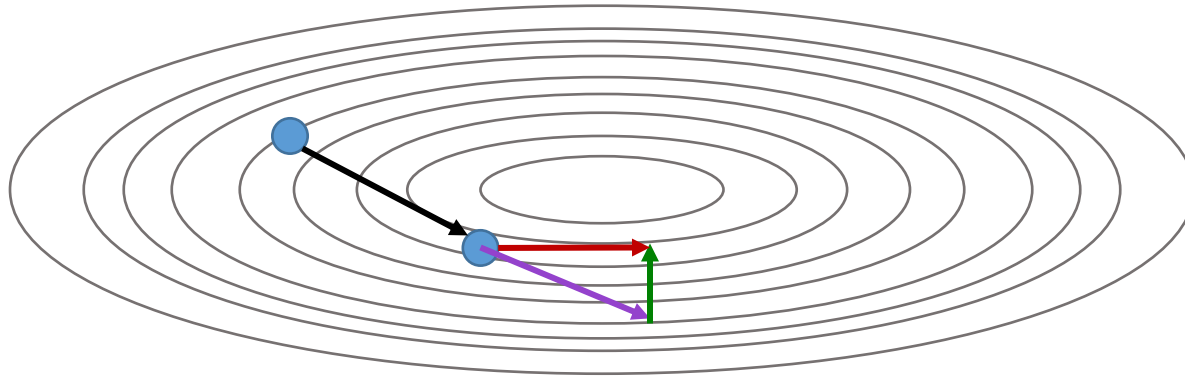
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step

# Nestorov's Accelerated Gradient



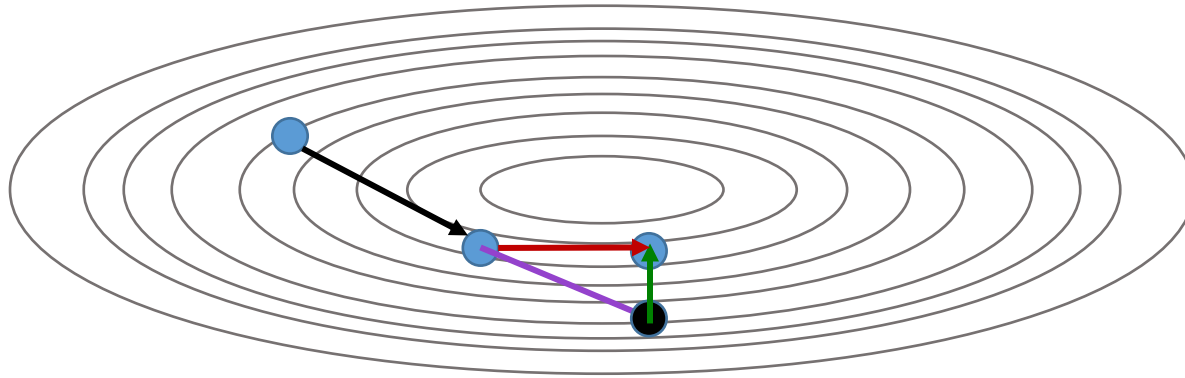
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step
  - Then compute the gradient step at the resultant position

# 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

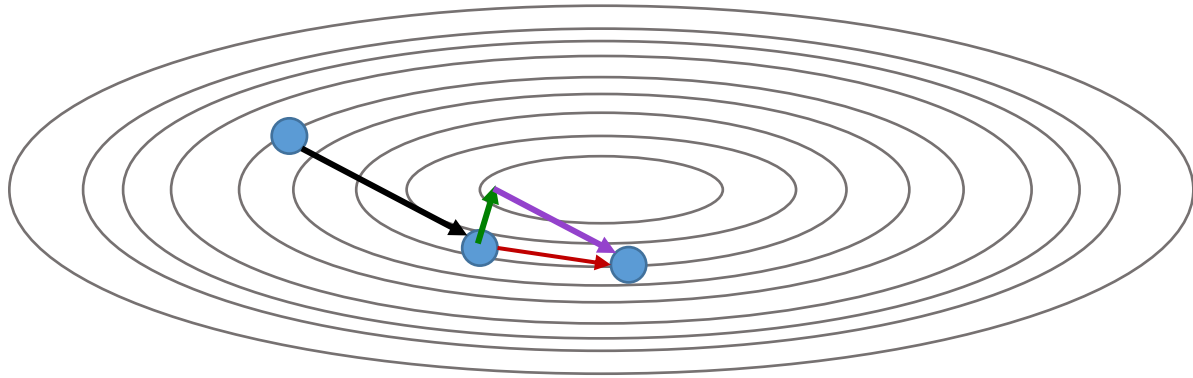


- Nestorov's method

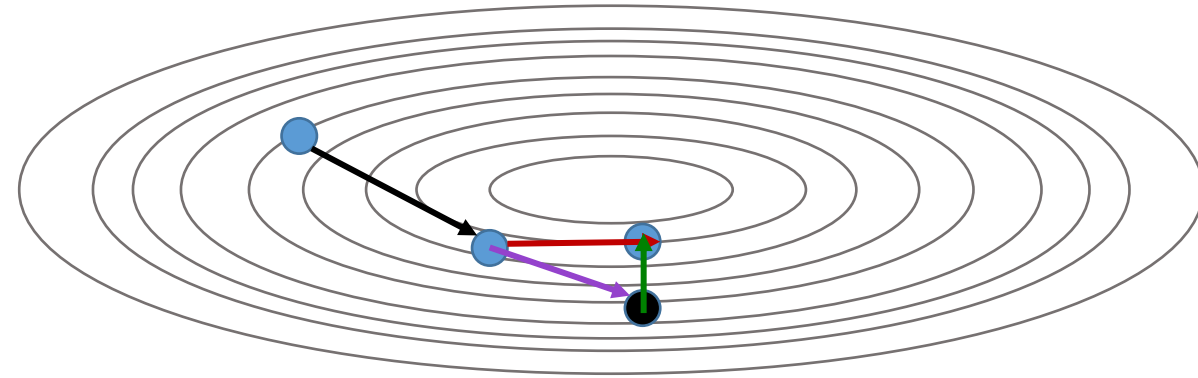
$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} + \eta \nabla_W E(W^{(k-1)} + \beta \Delta W^{(k-1)})$$

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

# Momentum vs Nesterov



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



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

Nesterov converges much faster

# Nesterov Momentum

- **Nesterov Momentum** is a slightly different version of the momentum update
  - It uses a **look-ahead gradient step**
  - It enjoys stronger theoretical converge guarantees for convex functions and in practice it also consistently works slightly better than standard momentum

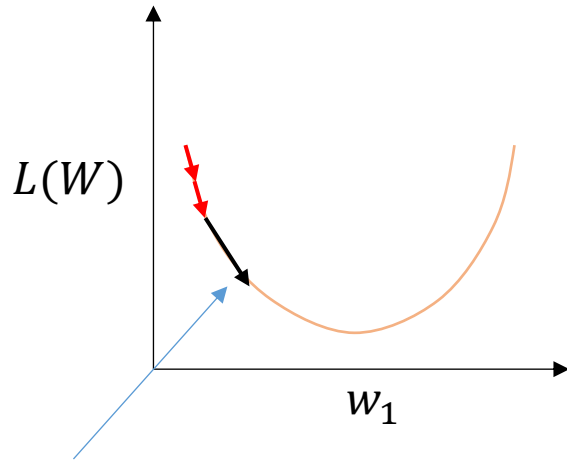
Nesterov, “A method of solving a convex programming problem with convergence rate  $O(1/k^2)$ ”, 1983

Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004

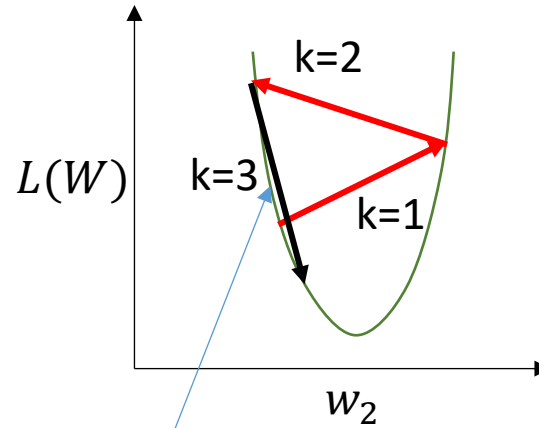
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013



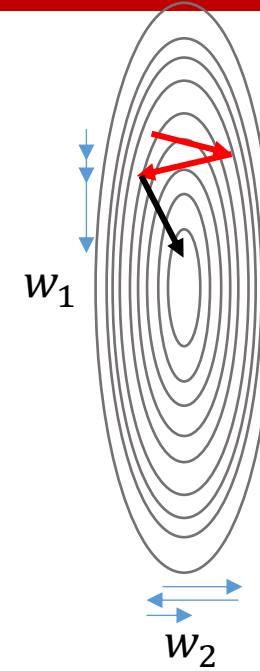
# Momentum methods: summary



Increase step size because previous updates consistently moved weight right



Decrease step size because previous updates kept changing direction

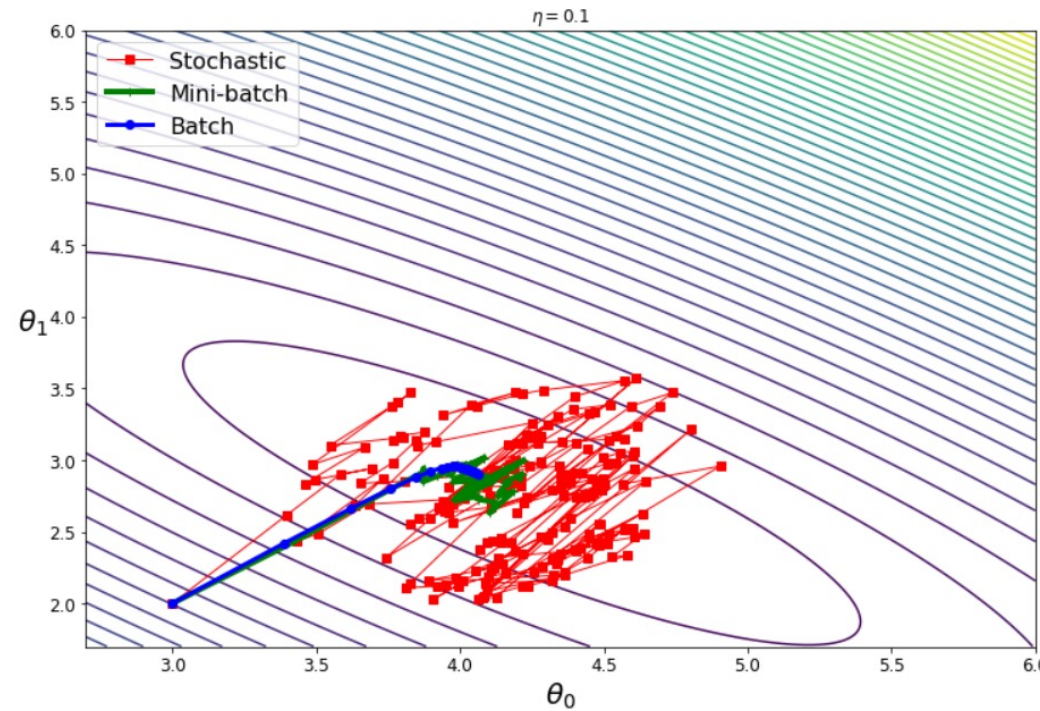


Step size shrinks along  $w_2$  but increases along  $w_1$

- Ideally: Have component-specific step size
  - Too many independent parameters (maintain a step size for every weight/bias)
- Adaptive solution: Start with a common step size
  - *Shrink* step size in directions where the weight oscillates
  - *Expand* step size in directions where the weight moves consistently in one direction

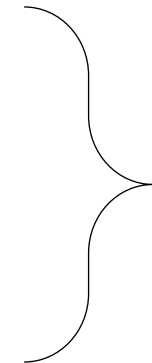
# Momentum and trend-based methods..

- Momentum methods which emphasize directions of steady improvement are demonstrably superior to other methods
- **Incremental SGD and mini-batch gradients** that will be discussed more later still tend to have high variance



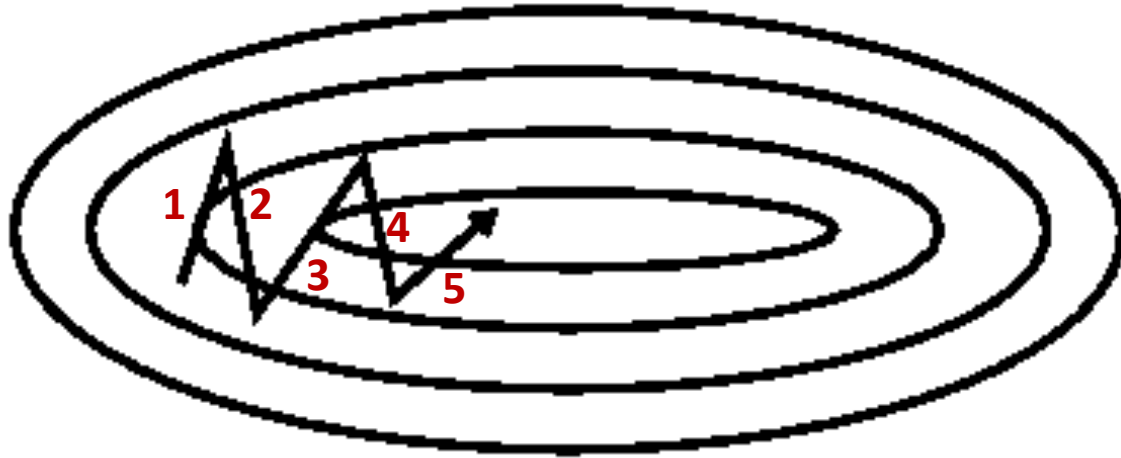
# More recent methods

- Several newer methods have been proposed that follow the general pattern of enhancing long-term trends to smooth out the variations of the mini-batch gradient
  - RMS Prop
  - Adagrad
  - AdaDelta
  - **ADAM: very popular in practice**
  - ...
- All roughly equivalent in performance



Adaptive learning rate

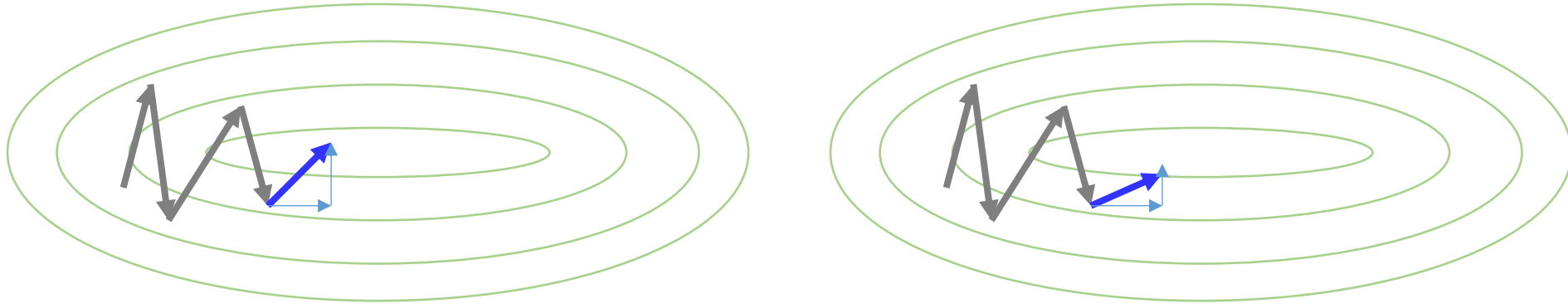
# Smoothing the trajectory



Step	X component	Y component
1	1	+2.5
2	1	-3
3	3	+2.5
4	1	-2
5	2	1.5

- Simple gradient and acceleration methods still demonstrate oscillatory behavior
- Observation: Steps in “oscillatory” directions show large total movement
  - In the example, total motion in the vertical direction is much greater than the horizontal
- Improvement: Dampen step size in directions with high motion
  - *Second order term*

# Variance-normalized step



- In recent past
  - Total movement in  $Y$  component of updates is high
  - Movement in  $X$  components is lower
- Current update, modify usual gradient-based update:
  - Scale *down*  $Y$  component
  - Scale *up*  $X$  component
  - *According to their variation (and not just their average)*
- A variety of algorithms have been proposed on this premise
  - We will see a popular example

# Adagrad

- Downscale a model parameter by square-root of sum of squares of all its historical derivatives
- Notation:
  - The **squared** derivative is  $(\partial_w E)^2$ 
    - element-wise squared derivative, **not the second derivative**
    - The same size as the parameter vector
- This is an example of an **adaptive learning rate**.

$$s^{(k)} = s^{(k-1)} + (\partial_w E)^2{}^{(k)}$$

$$w^{(k+1)} = w^{(k)} - \frac{\eta}{\sqrt{s^{(k)} + \epsilon}} (\partial_w E)^{(k)}$$

# RMS Prop

*G. Hinton. Lecture 6e on neural networks (**RMSprop**: Divide the gradient by a running average of its recent magnitude). 2014*

- AdaGrad might shrink the learning rate too aggressively due to the accumulating sum of squared derivatives
  - RMS Prop uses exponentially decaying average of squared gradients
- Notation:
  - The **mean squared** derivative is a running estimate of the average squared derivative of parameters. We will show this as  $\overline{(\partial_w E)^2}$
- Modified update rule: We want to
  - scale down updates with large mean squared derivatives
  - scale up updates with small mean squared derivatives

# RMS Prop

*G. Hinton. Lecture 6e on neural networks (**RMSprop**: Divide the gradient by a running average of its recent magnitude). 2014*

- **Procedure:**

- Maintain a running estimate of the mean squared value of derivatives for each parameter
- Scale update of the parameter by the *inverse* of the **root mean squared** elements of gradient

$$\overline{(\partial_w E)^2}^{(k)} = \beta \overline{(\partial_w E)^2}^{(k-1)} + (1 - \beta) (\partial_w E)^2^{(k)}$$

$$w^{(k+1)} = w^{(k)} - \frac{\eta}{\sqrt{\overline{(\partial_w E)^2}^{(k)} + \epsilon}} (\partial_w E)^{(k)}$$

Scaling of the gradient in each dimension based on sum of squares derivatives in that dimension

The magnitude of the derivative is being normalized out



# ADAM: RMSprop with momentum

- **RMS prop** only considers a second-moment normalized version of the current gradient
- **ADAM** utilizes a smoothed version of the ***momentum-augmented*** gradient
- **Procedure:**
  - Maintain a running estimate of the **mean derivative** for each parameter
  - Maintain a running estimate of the **mean squared value of derivatives** for each parameter
  - Scale update of the parameter by the *inverse* of the *root mean squared* derivative (bias correction)

First moment:  $m^{(k)} = \beta_1 m^{(k-1)} + (1 - \beta_1)(\partial_w E)^{(k)}$

Second moment:  $v^{(k)} = \beta_2 v^{(k-1)} + (1 - \beta_2)(\partial_w E)^2{}^{(k)}$

Problem: may cause very large steps at the beginning

$$w^{(k+1)} = w^{(k)} - \frac{\eta}{\sqrt{v^{(k)} + \epsilon}} m^{(k)}$$

# ADAM: RMSprop with momentum

- RMS prop only considers a second-moment normalized version of the current gradient
- ADAM utilizes a smoothed version of the *momentum-augmented* gradient
- **Procedure:**
  - Maintain a running estimate of the mean derivative for each parameter
  - Maintain a running estimate of the mean squared value of derivatives for each parameter
  - Scale update of the parameter by the *inverse* of the *root mean squared* derivative (bias correction)

First moment:  $m^{(k)} = \beta_1 m^{(k-1)} + (1 - \beta_1)(\partial_w E)^{(k)}$

Second moment:  $v^{(k)} = \beta_2 v^{(k-1)} + (1 - \beta_2)(\partial_w E)^2{}^{(k)}$

Problem: may cause very large steps at the beginning

$$\hat{m}^{(k)} = \frac{\hat{m}^{(k)}}{1 - \beta_1^k} \quad \hat{v}^{(k)} = \frac{v^{(k)}}{1 - \beta_2^k}$$

Solution: Bias correction to prevent first and second moment estimates start at zero

$$w^{(k+1)} = w^{(k)} - \frac{\eta}{\sqrt{\hat{v}^{(k)} + \epsilon}} \hat{m}^{(k)}$$

# Exponentially weighted averages

- $v_0 = 0$
- $v_t = \gamma v_{t-1} + (1 - \gamma)\theta_t$

# Exponentially weighted moving averages

- $v_0 = 0$
- $v_t = \gamma v_{t-1} + (1 - \gamma)\theta_t$

$$v_t = (1 - \gamma)(\theta_t + \gamma\theta_{t-1} + \gamma^2\theta_{t-2} + \cdots + \gamma^{t-1}\theta_1)$$

$$1 - \gamma^t = (1 - \gamma)(1 + \gamma + \gamma^2 + \cdots + \gamma^{t-1})$$

$$\frac{v_t}{1 - \gamma^t} = \frac{\theta_t + \gamma\theta_{t-1} + \gamma^2\theta_{t-2} + \cdots + \gamma^{t-1}\theta_1}{1 + \gamma + \gamma^2 + \cdots + \gamma^{t-1}}$$

# Bias correction

- Computation of these averages more accurately:

$$\hat{v}_t = \frac{v_t}{1 - \gamma^t}$$

- Example:

during warming up, the bias correction can help  
when  $t$  is large enough, it makes almost no difference

$$v_0 = 0$$

$$v_1 = 0.9v_0 + 0.1\theta_1 = 0.1\theta_1$$

$$v_2 = 0.9v_1 + 0.1\theta_2 = 0.09\theta_1 + 0.1\theta_2$$

$$\hat{v}_2 = \frac{0.09\theta_1 + 0.1\theta_2}{1 - 0.9^2} = \frac{0.09\theta_1 + 0.1\theta_2}{0.19}$$

# ADAM: RMSprop with momentum

- RMS prop only considers a second-moment normalized version of the current gradient
- ADAM utilizes a smoothed version of the *momentum-augmented* gradient
- Takes **RMSprop** and **momentum** idea together
- **Procedure:**
  - Maintain a running estimate of the mean derivative for each parameter
  - Maintain a running estimate of the mean squared value of derivatives for each parameter
  - Scale update of the parameter by the *inverse* of the *root mean squared* derivative

$$m^{(k)} = \beta_1 m^{(k-1)} + (1 - \beta_1)(\partial_w E)^{(k)}$$

$$v^{(k)} = \beta_2 v^{(k-1)} + (1 - \beta_2)(\partial_w E)^2{}^{(k)}$$

$$\hat{m}^{(k)} = \frac{\hat{m}^{(k)}}{1 - \beta_1^k}, \quad \hat{v}^{(k)} = \frac{v^{(k)}}{1 - \beta_2^k}$$

$$w^{(k+1)} = w^{(k)} - \frac{\eta}{\sqrt{\hat{v}^{(k)} + \epsilon}} \hat{m}^{(k)}$$

Hyperparameter Choice:

$\eta$ : need to be tuned

$$\beta_1 = 0.9$$

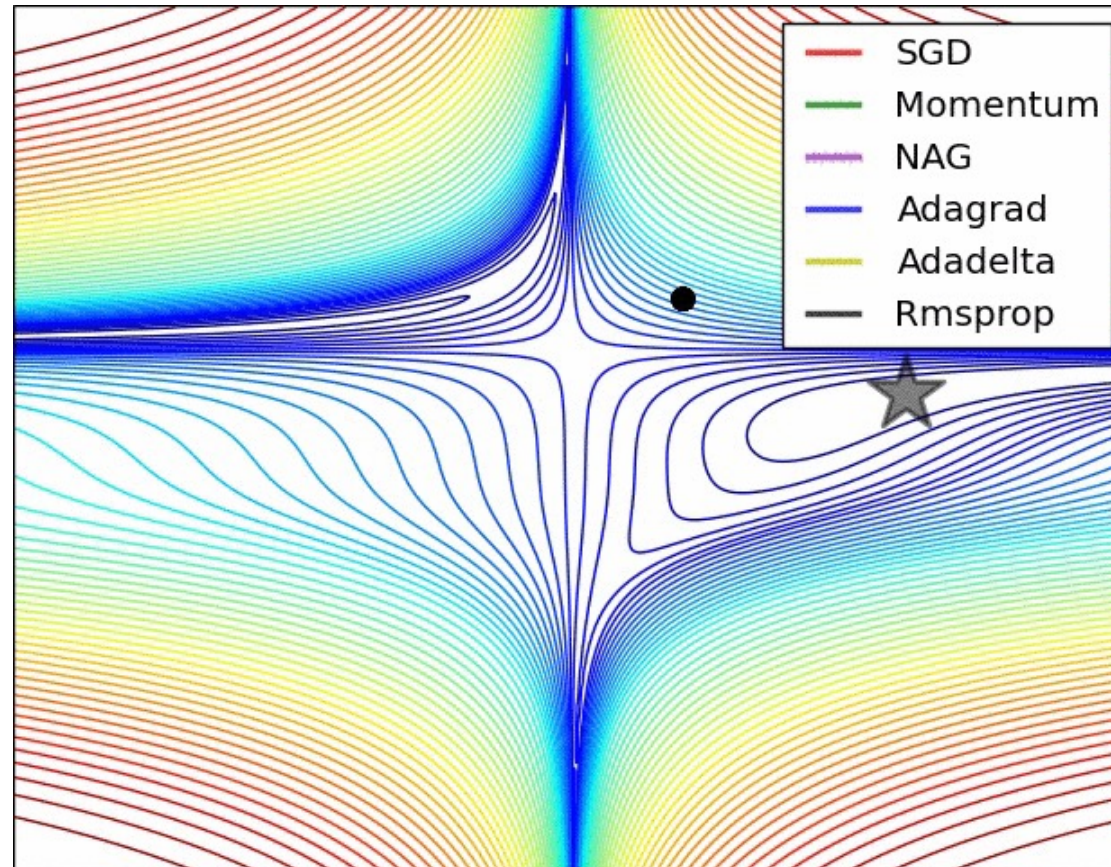
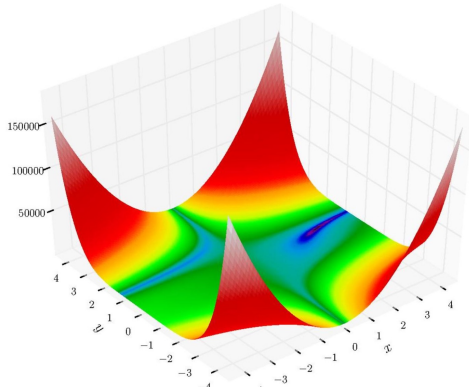
$$\beta_2 = 0.999$$

$$\epsilon = 10^{-8}$$

# Other variants of the same theme

- Many:
  - Adagrad
  - AdaDelta
  - ADAM
  - AdaMax
  - ...
- Generally no explicit learning rate to optimize
  - But come with other hyper parameters to be optimized
  - Typical params:
    - RMSProp:  $\eta = 0.001, \beta = 0.9$
    - ADAM:  $\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$

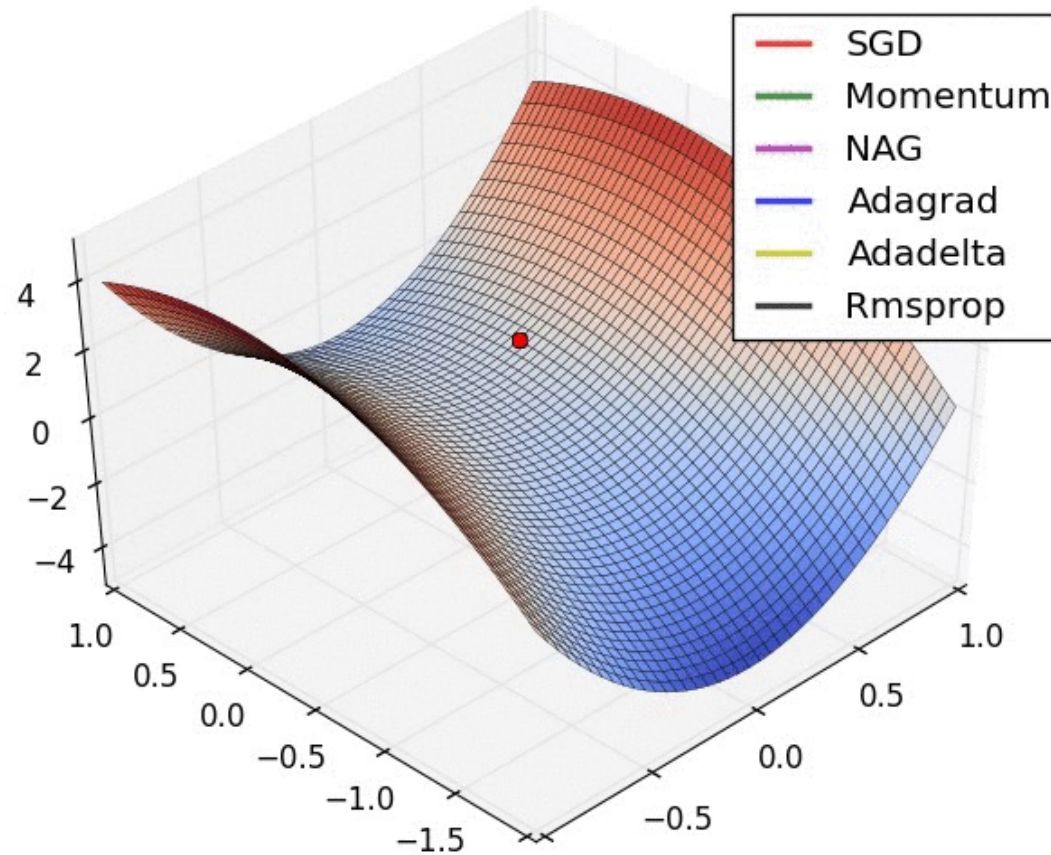
# Visualizing the optimizers: Beale's Function



<http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>

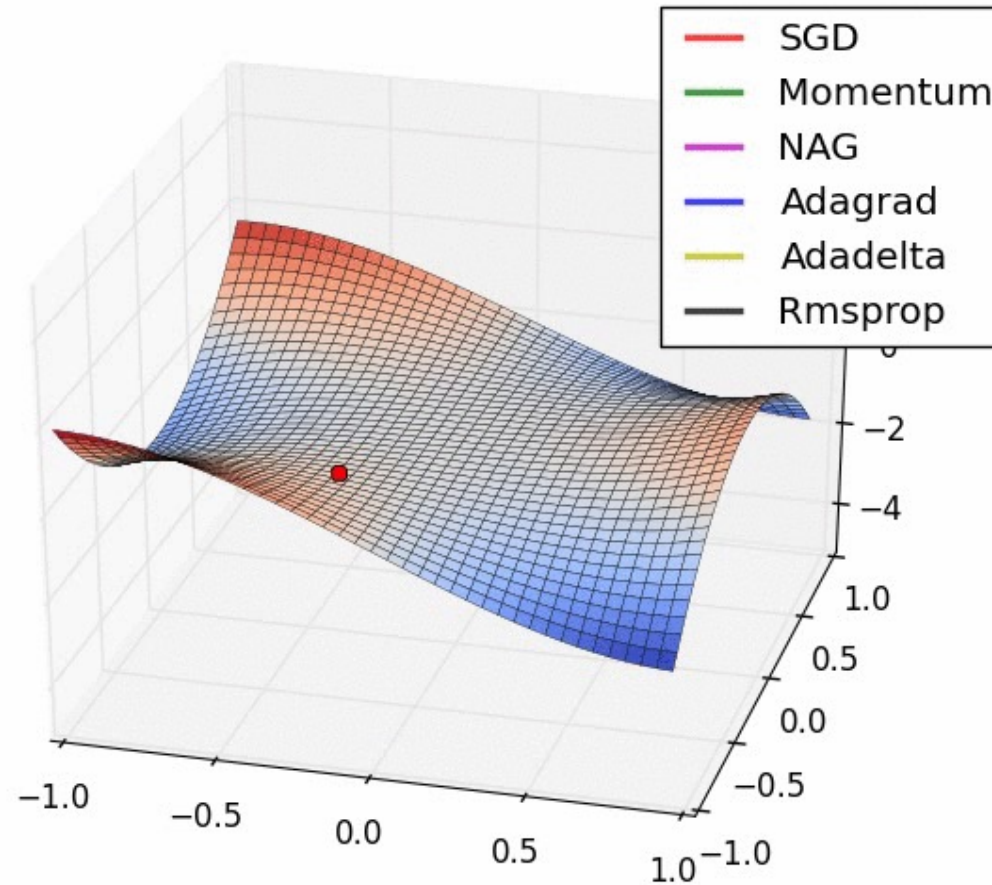


# Visualizing the optimizers: Long Valley



<http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>

# Visualizing the optimizers: Saddle Point



<http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>

# Issues with adaptive learning rates

- For simple overparameterized problems, adaptive methods often find drastically different solutions than GD or SGD
- For some applications, the solutions found by adaptive methods **generalize worse** (often significantly worse) than SGD, even when they have better training performance.
- Adaptive gradient algorithms tend to converge to sharp minima whose local basin has large curvature and usually generalize poorly
  - while SGD prefers to find flat minima and thus generalizes better

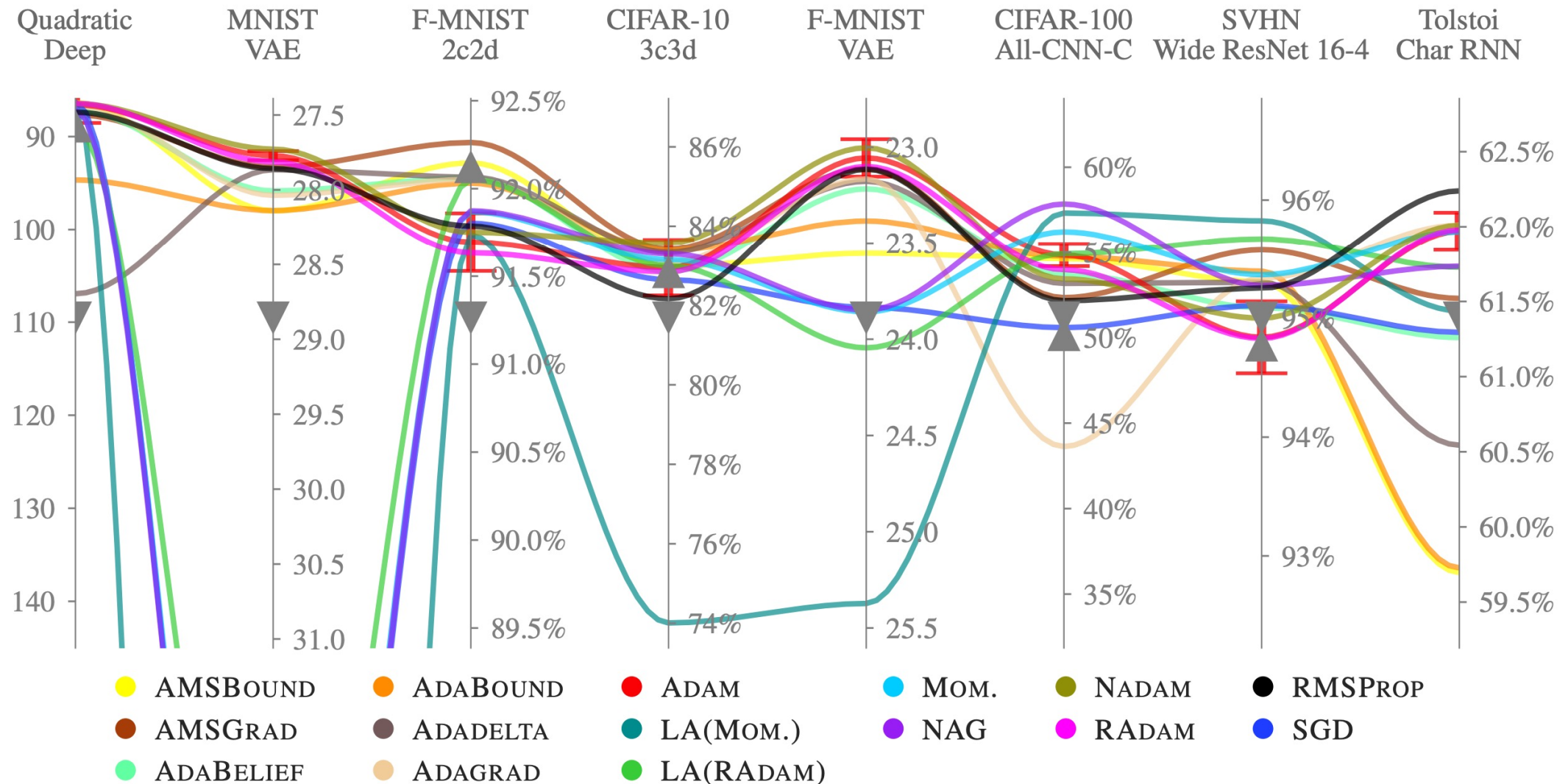
Wilson et al. "[The Marginal Value of Adaptive Gradient Methods in Machine Learning](#)", NeurIPS 2017

Zhou et al., "Towards Theoretically Understanding Why SGD Generalizes Better Than ADAM in Deep Learning", NeurIPS 2020

# Benchmarking Deep Learning Optimizers

- No optimization method clearly dominates across all tested tasks
  - ADAM remains a strong contender, with newer methods failing to significantly and consistently outperform it.
- Optimizer's performance is heavily problem-dependent and that there is no single best optimizer across workloads

# Benchmarking Deep Learning Optimizers



Schmidt et al., "Descending through a Crowded Valley — Benchmarking Deep Learning Optimizers", NeurIPS 2021.