Train Your Model: Optimization

CSE 849 Deep Learning

Spring 2025

Zijun Cui

Partial slides are adapted from CMU 11-785 Introduction to Deep Learning by Bhiksha Raj

Project 0 Correction

- Question 3: Two-layer MLP
  - It should have an activation function involved
  - Results with be different with different seeds

- You will not be penalized due to this oversight

Piazza Questions

- Piazza questions will be answered asap, within 1 day

- A more effective way to resolve programming-related questions is using TA hours

Neural network training algorithm

- Initialize all weights and biases $(\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \ldots, \mathbf{W}_N, \mathbf{b}_N)$
- Do:
  - $Loss = 0$
  - For all k, initialize $\nabla_{\mathbf{W}_k} Loss = 0$, $\nabla_{\mathbf{b}_k} Loss = 0$
  - For all $t = 1{:}T$ # Loop through training instances
    - Forward pass : Compute
      - Output $\boldsymbol{Y}(\boldsymbol{X_t})$,
      - Divergence $\boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$
    - Backward pass: For all k compute:
      - $\nabla_{\mathbf{W}_k} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$, $\nabla_{\mathbf{b}_k} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$
      - $\nabla_{\mathbf{W}_k} Loss \mathrel{+}= \nabla_{\mathbf{W}_k} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$; $\nabla_{\mathbf{b}_k} Loss \mathrel{+}= \nabla_{\mathbf{b}_k} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$

    Computing gradient (uses backprop)
  - For all k, update:

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

    Gradient descent
- Until $Loss$ has converged

Moving Forward...

- Does backprop always work?
- Convergence of gradient descent
  - Learning Rates
  - Newton's Method
  - Momentum Method and Nestorov's Accelerated Gradient

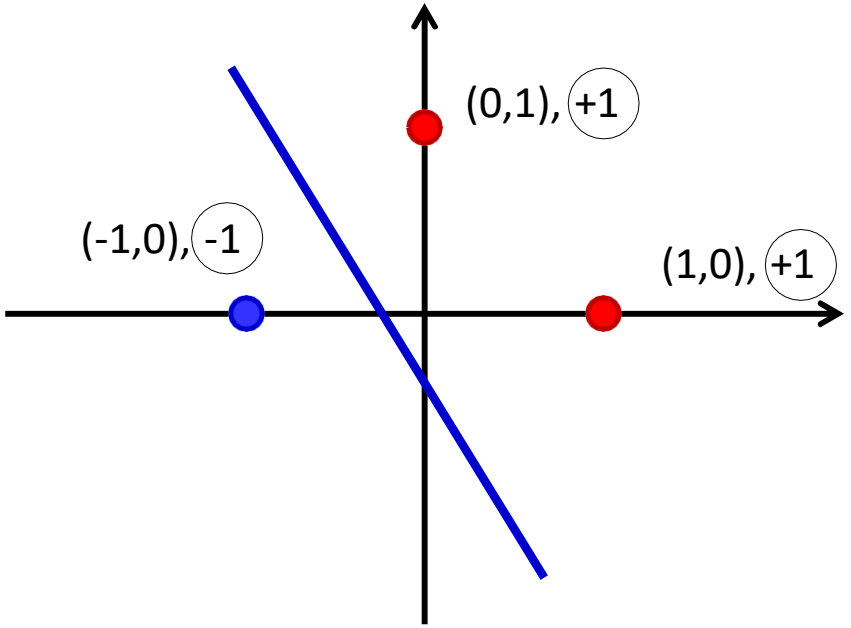
Does backprop do the right thing?

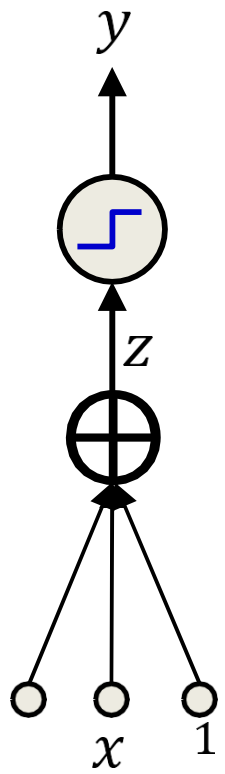
- **Is backprop always right?**
  - Assuming it actually finds the global minimum of the loss (average divergence)?
  - -- Actual question: Does gradient descent find the right solution, even when it finds the actual minimum?

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

Backprop fails to separate where perceptron succeeds

- Brady, Raghavan, Slawny, ’89
- Simple problem, 3 training instances, single neuron



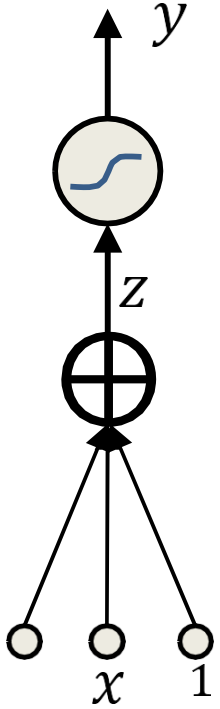


- Perceptron training rule trivially find a perfect solution

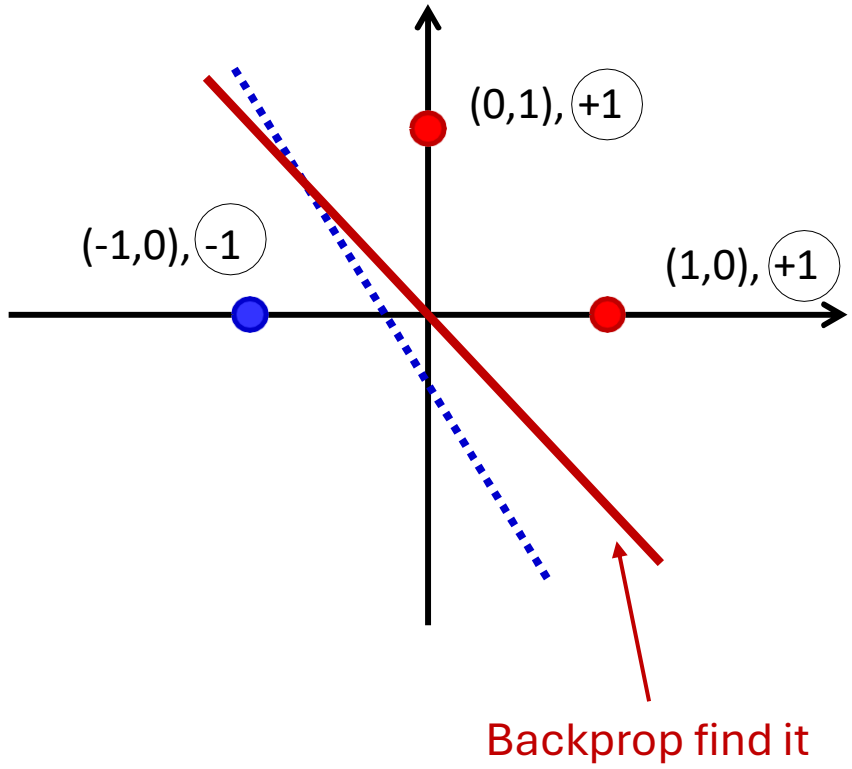
Backprop vs. Perceptron

- Back propagation using logistic function f and L_2 divergence, i.e.,

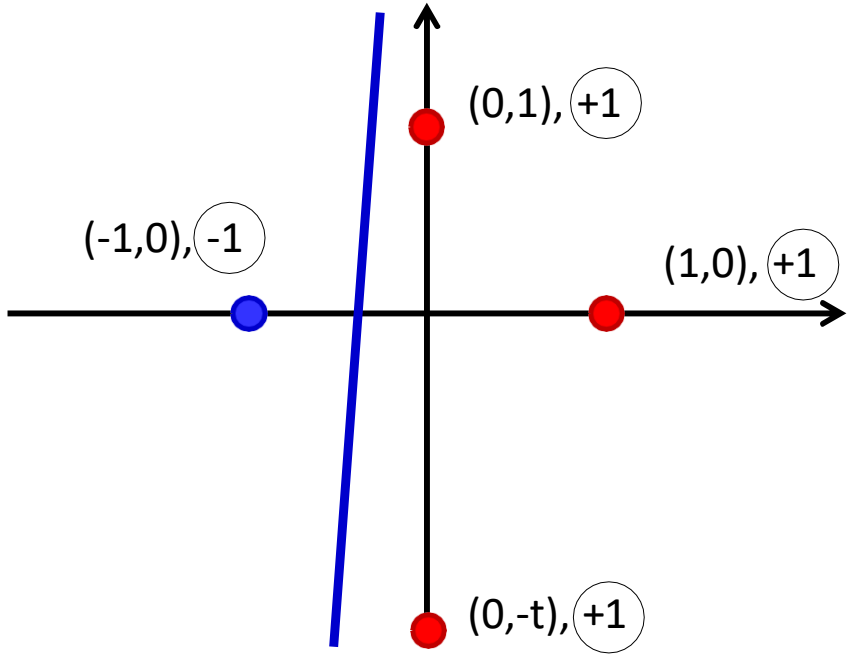
$$Div = (y - d)^2$$

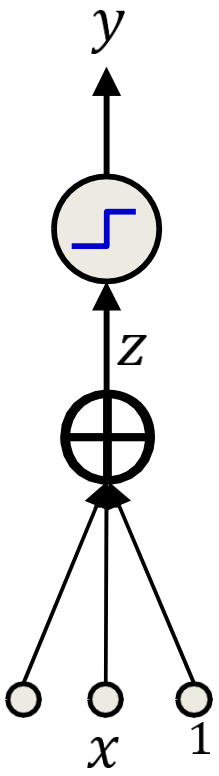


- Unique minimum trivially proved to exist



Backprop vs. Perceptron

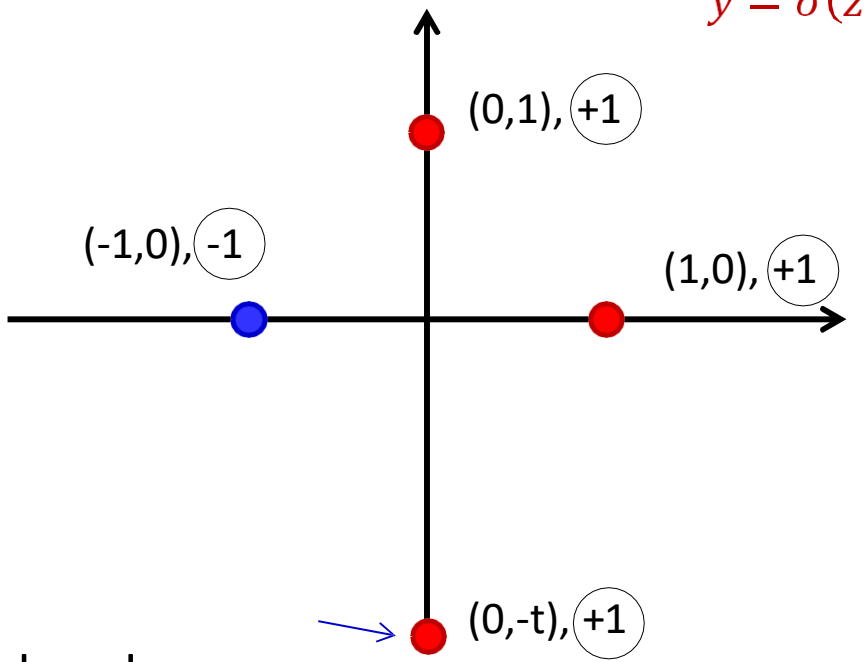


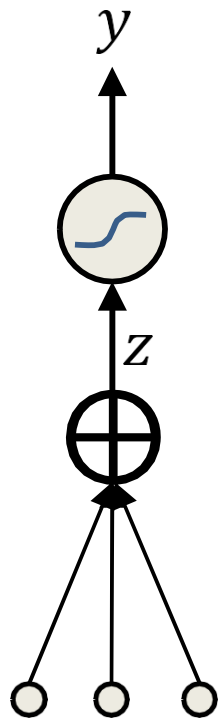


- Now add a fourth point
- t is very large (point near $-\infty$)
- Perceptron trivially finds a solution (may take t^2 iterations)

Backprop

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





- Consider backprop:
- Suppose 1-ε is the achievable value
- Contribution of fourth point to derivative of L_2 error:

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

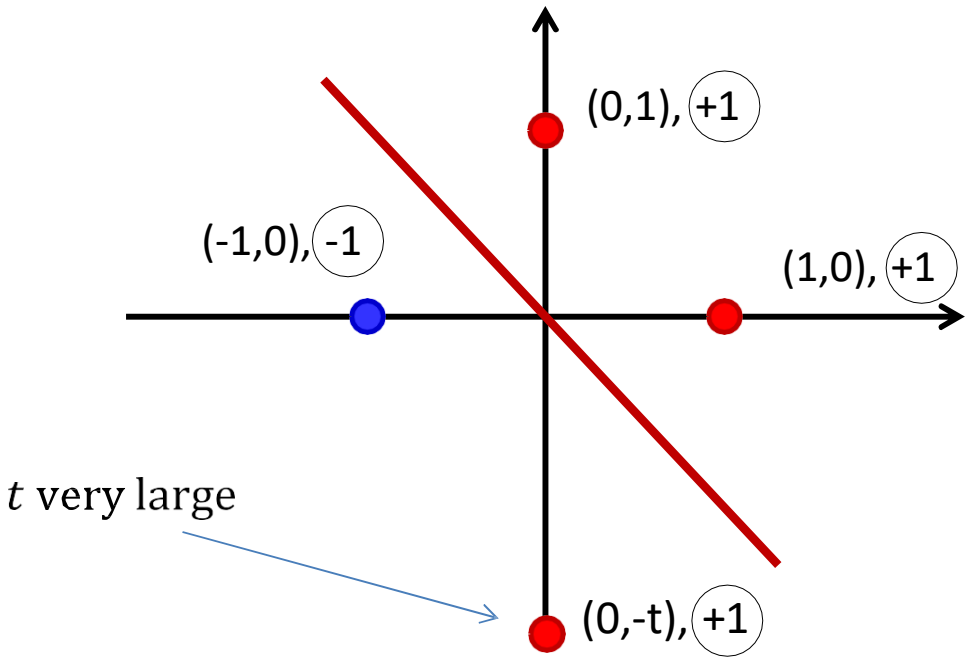
$$\frac{d\, div_4}{dw_y} = 2\left(1 - \varepsilon - \sigma(-w_y t + b)\right)\sigma'(-w_y t + b)t$$

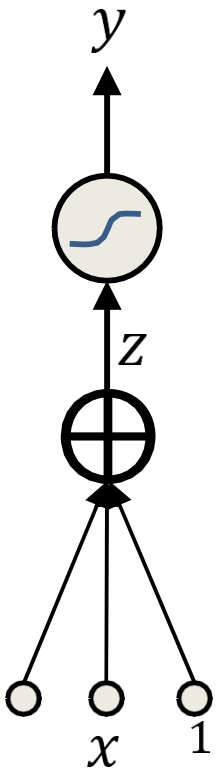
$$\frac{d\, div_4}{db} = -2\left(1 - \varepsilon - \sigma(-w_y t + b)\right)\sigma'(-w_y t + b)$$

- For very large positive t, $|w_y| > \epsilon$
- $\left(1 - \varepsilon - \sigma(-w_y t + b)\right) \to 1$ as $t \to \infty$ if w_y is positive
- $\sigma'(-w_y t + b) \to 0$ *exponentially* as $t \to \infty$
- Therefore, for very large positive t

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

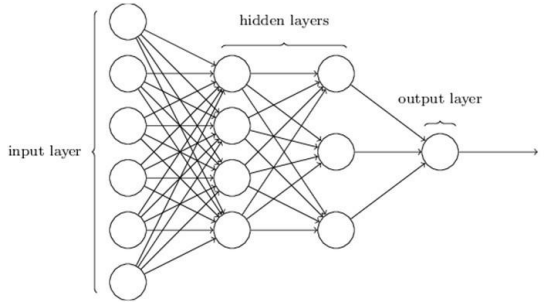
Backprop

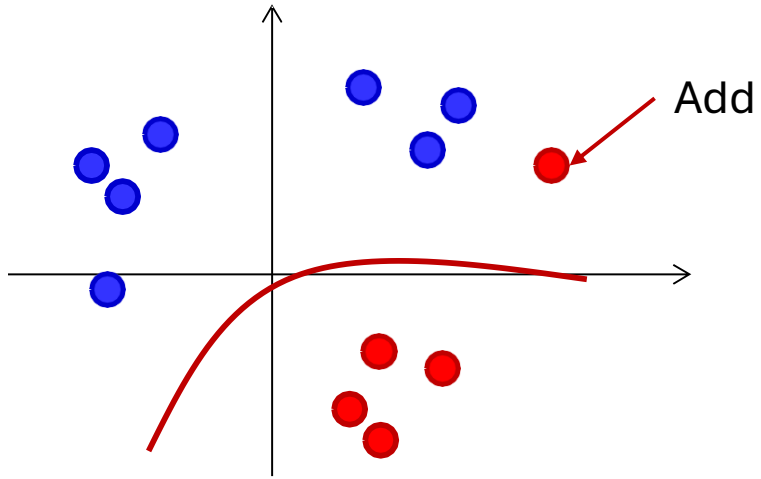




- The fourth point at (0, -t) does not change the gradient of the L_2 divergence near the optimal solution for 3 points
- The optimum solution for 3 points is also a broad *local* minimum (0 gradient) for the 4-point problem.
  - Will be found by backprop nearly all the time
- Local optimum solution found by backprop
- Does not separate the points *even though the points are linearly separable!*

Backprop fails to separate even when possible





- This is not restricted to single perceptrons
- An MLP learns non-linear decision boundaries that are determined from the entirety of the training data
- Adding a few “spoilers” will not change their behavior

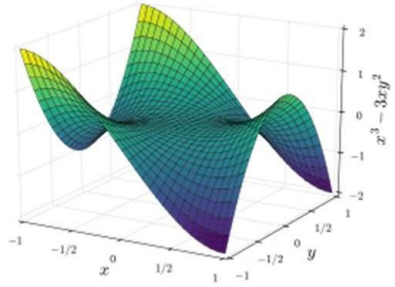
Take away

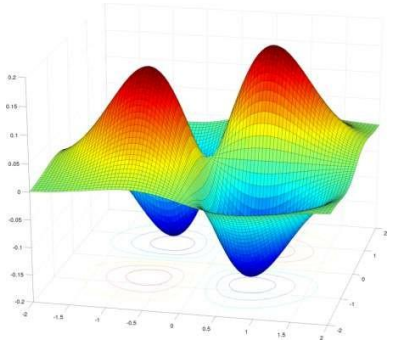
- Backpropagation will often not find a separating solution ***even though the solution is within the class of functions learnable by the network***

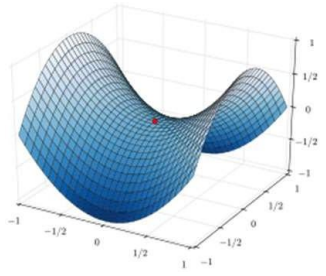
- This is because the separating solution is not a feasible optimum for the loss function
- One resulting benefit is that a backprop-trained neural network classifier has lower variance than an optimal classifier for the training data

The Loss Surface

- Assumed the loss objective had a single global optimum. Now let's talk about local optima
- **Saddle point:** A point where the slope is zero
  - The surface increases in some directions, but decreases in others
    - Some of the Eigenvalues of the Hessian are positive; others are negative
  - Gradient descent algorithms often get "stuck" in saddle points







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

BTW, there are lots of studies/statements on loss surfaces…

Convergence

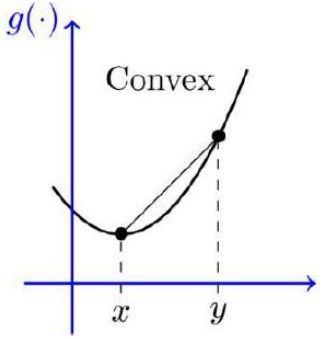
- In the discussion so far, we have assumed the training arrives at a local minimum

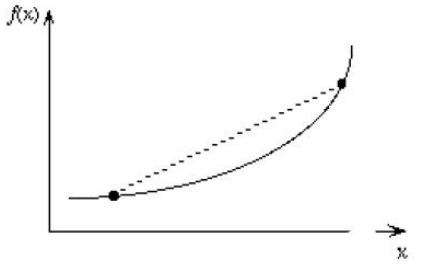
- Does it always converge?
- How long does it take?

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

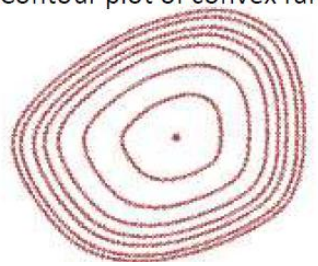
Convex Loss Functions

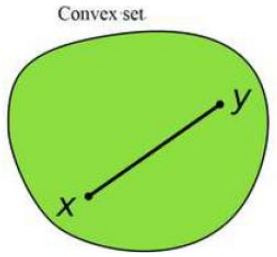
- A surface is “convex” if it is continuously curving upward
  - We can connect any two points on or above the surface without intersecting it



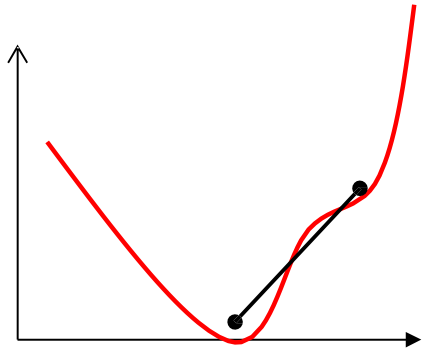


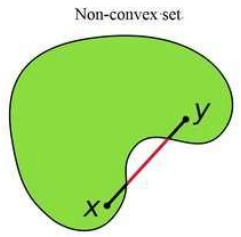
Contour plot of convex function





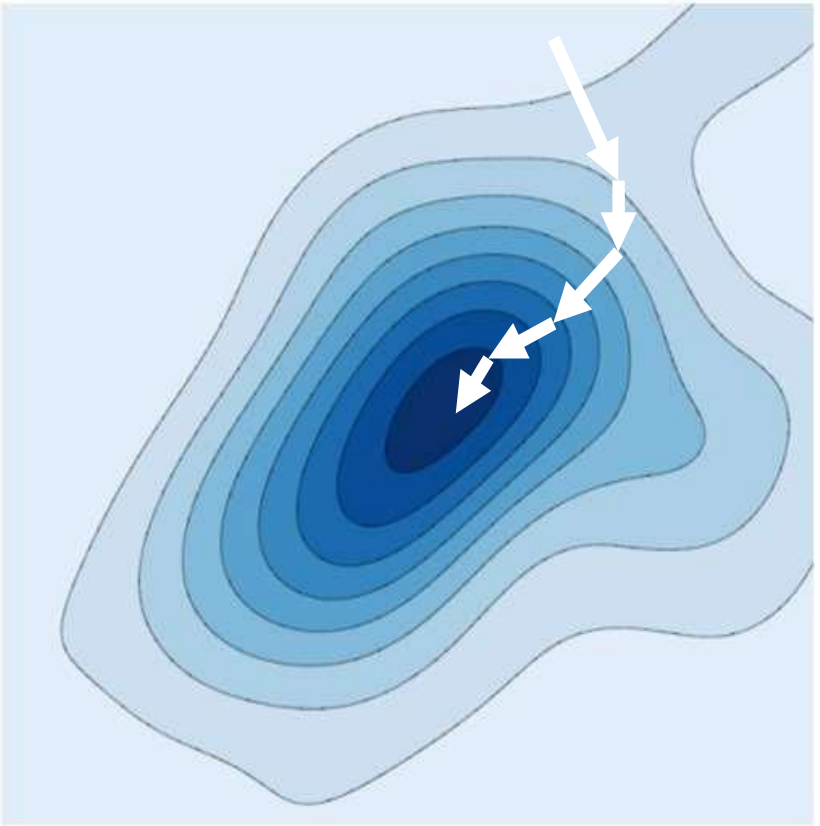
- Not convex : Neural network loss surface is generally not convex
  - Streetlight effect





Convergence of gradient descent

converging



- 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

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

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

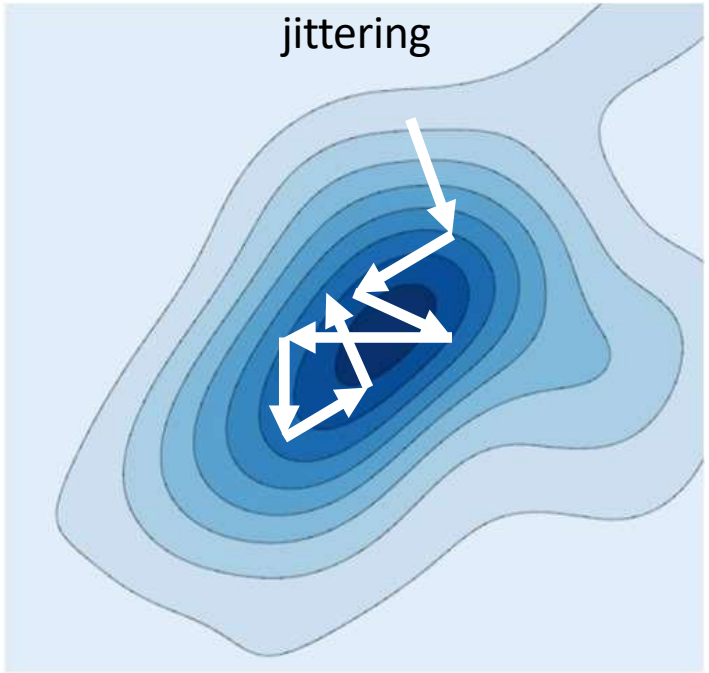
  - $x^{(k+1)}$ is the k-th iteration
  - x^* is the optimal value of x

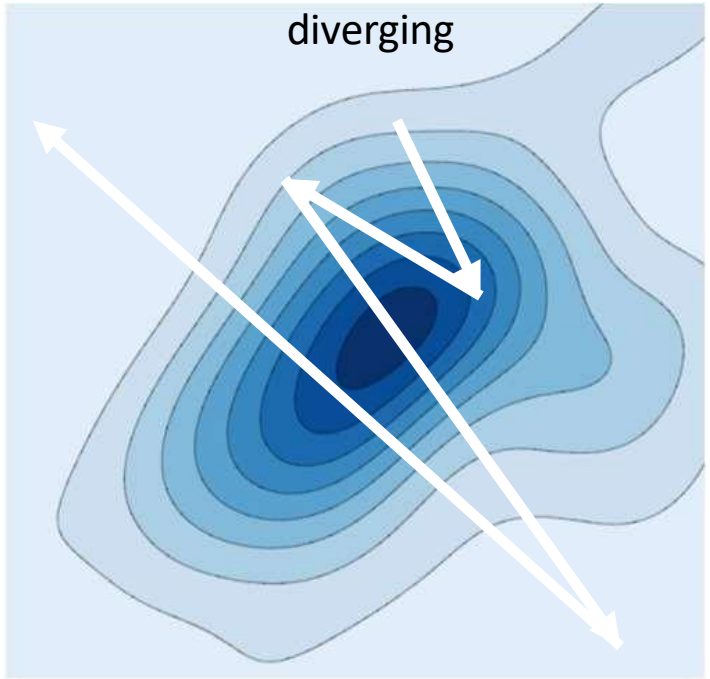
  - If R is a constant (or upper bounded), the convergence is *linear*

Convergence of gradient descent

- The algorithm may not actually converge
  - It may jitter around the local minimum
  - It may even diverge

- Conditions for convergence?



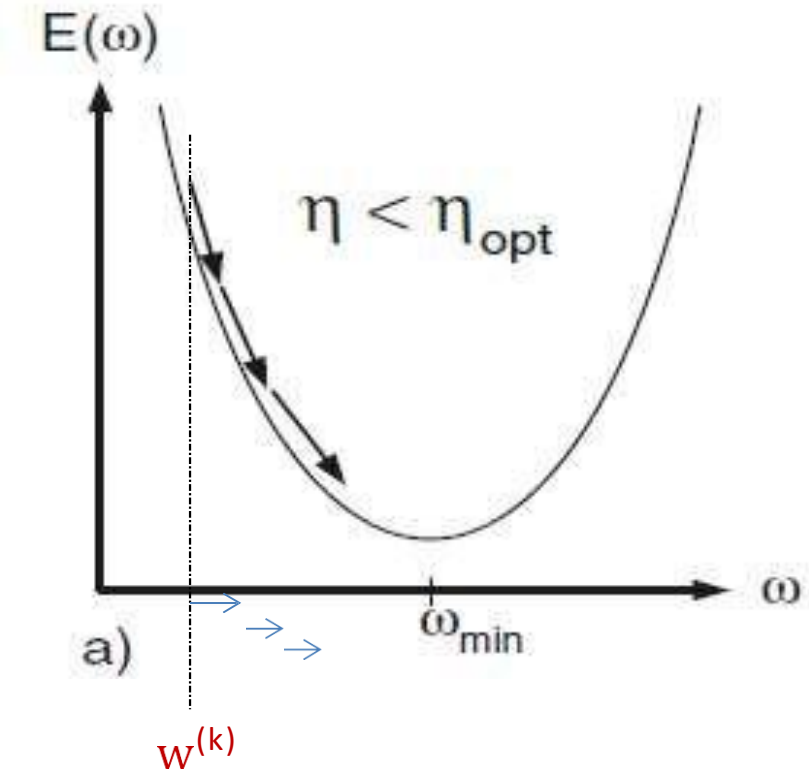


Convergence for quadratic surfaces

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

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

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



Question: What is the optimal step size η to get there fastest?

- Any quadratic objective can be written as

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

  - Taylor expansion

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

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

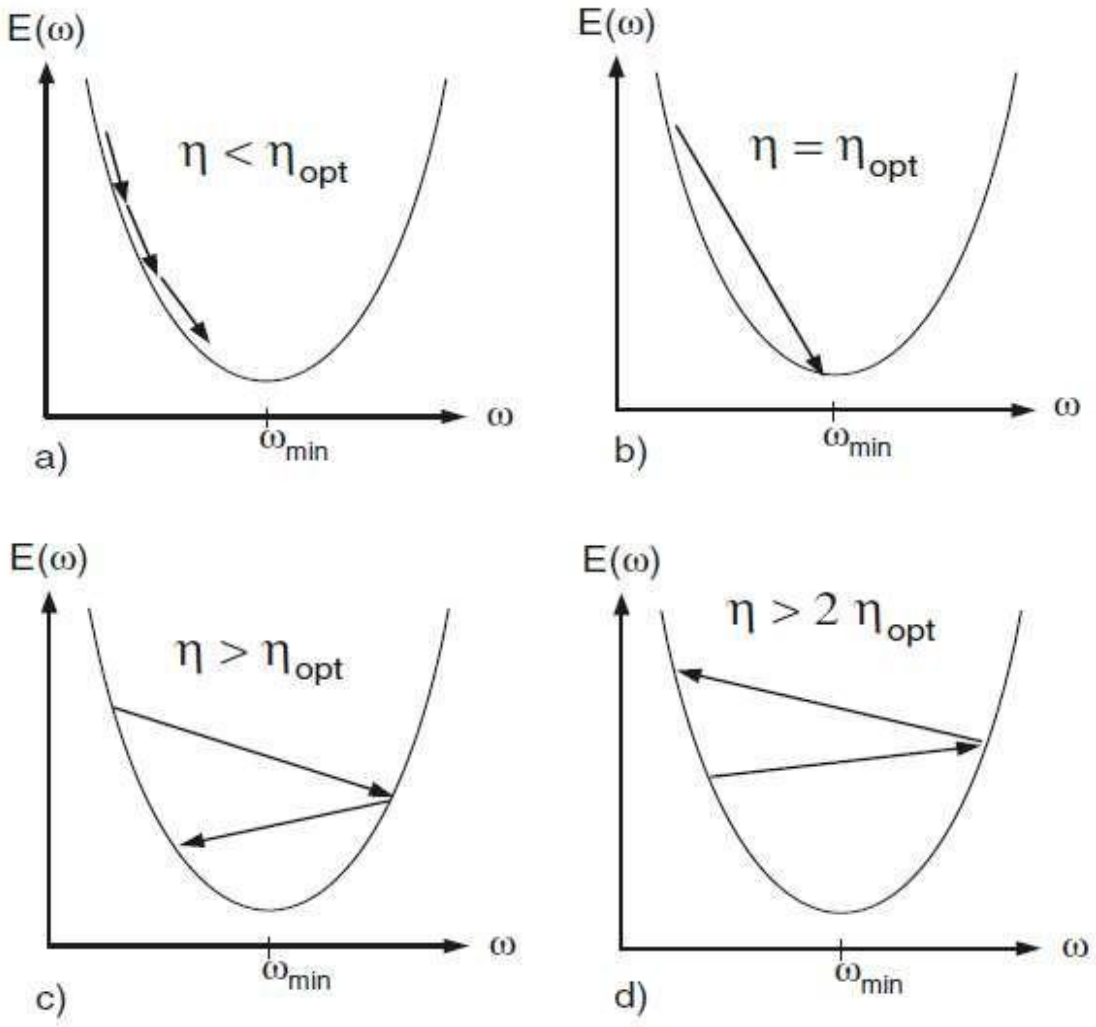
- Note:

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

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

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

With non-optimal step size



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

Functions of Multivariate Inputs

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

- Consider a simple quadratic convex function

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

  - Since $E^T = E$ (E is scalar), $\mathbf{A}$ can always be made symmetric

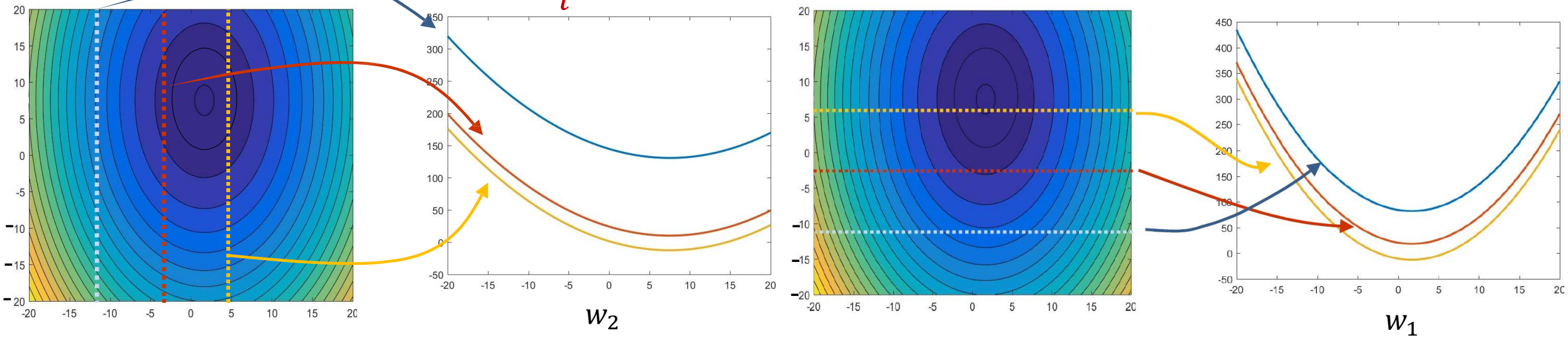
- When $\mathbf{A}$ is diagonal:

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

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

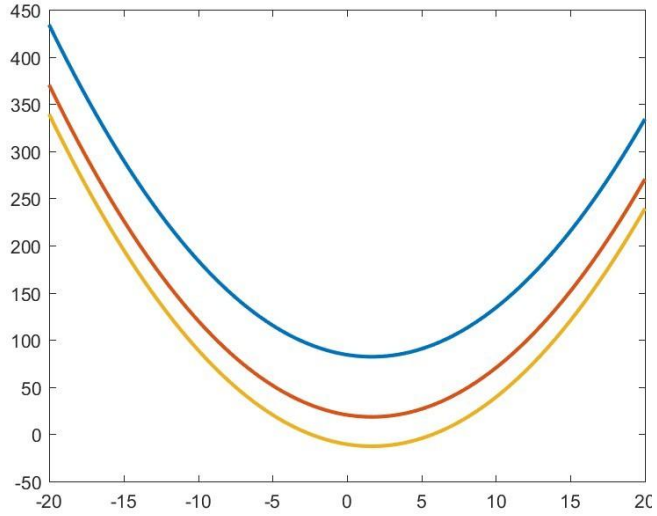
Multivariate Quadratic with Diagonal A

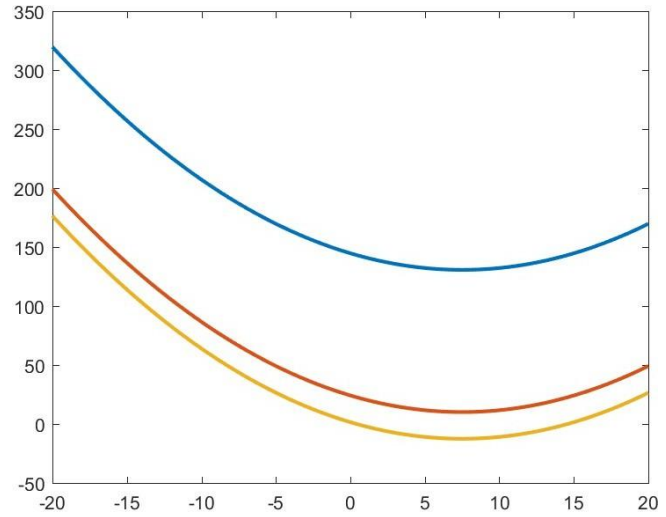
$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c = \frac{1}{2}\sum_i \left(a_{ii}w_i^2 + b_i w_i\right) + c = \frac{1}{2}a_{ii}w_i^2 + b_i w_i + c + C(\neg w_i)$$



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

“Descents” are uncoupled





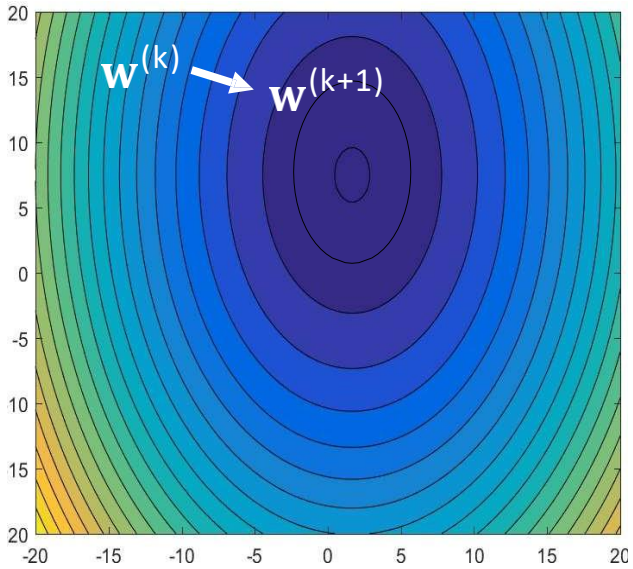
$$E = \frac{1}{2}a_{11}w_1^2 + b_1 w_1 + c + C(\neg w_1) \qquad E = \frac{1}{2}a_{22}w_2^2 + b_2 w_2 + c + C(\neg w_2)$$

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

- 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**

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

Vector update rule



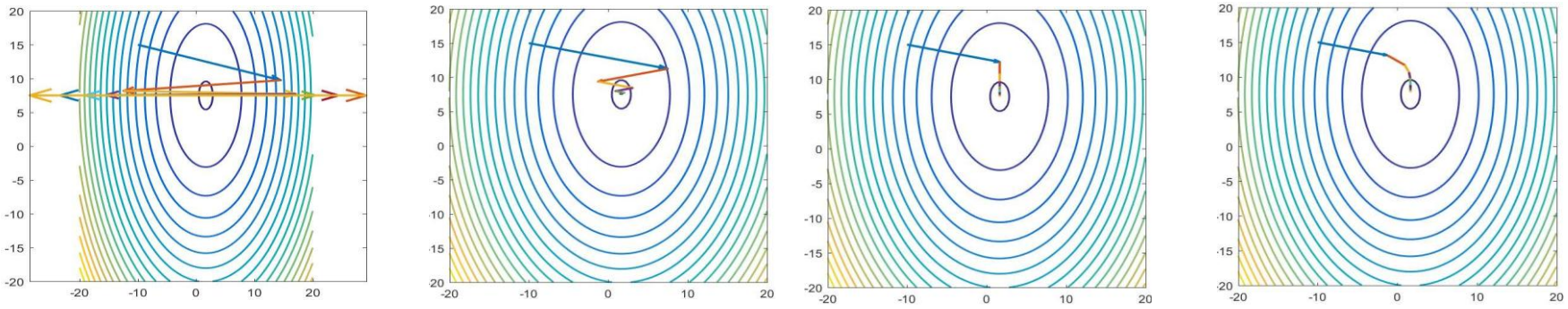
$$\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta \nabla_{w} E^{\top}$$

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

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

Dependence on learning rate

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



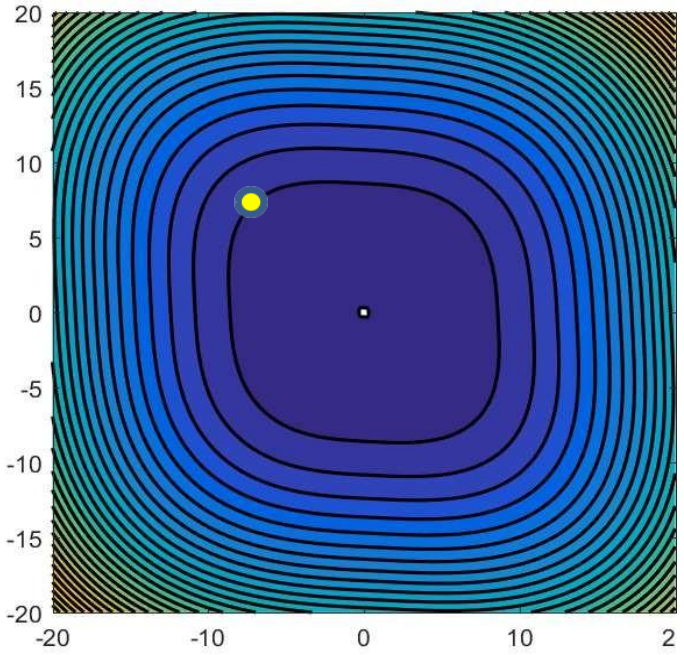
$\eta = 2.1\eta_{1,\text{opt}}$ $\eta = 1.5\eta_{1,\text{opt}}$ $\eta = \eta_{1,\text{opt}}$ $\eta = 0.75\eta_{1,\text{opt}}$

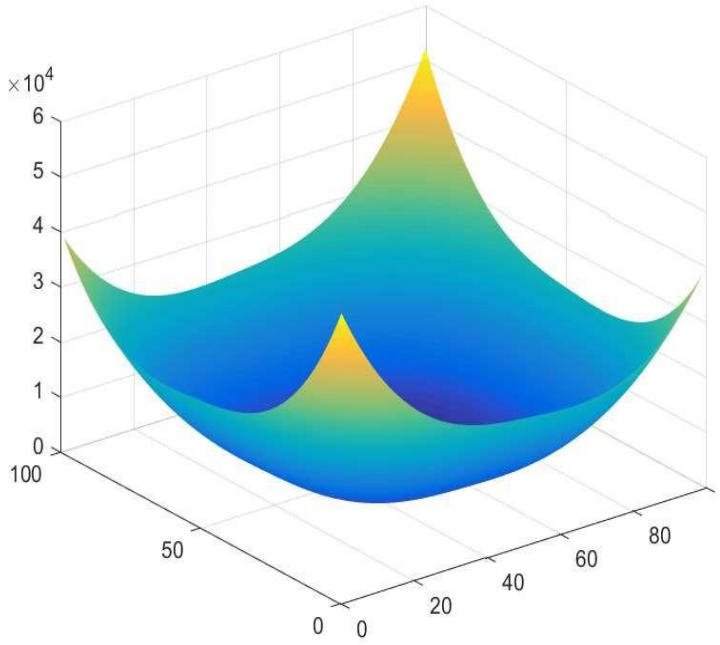
- The learning rate must be lower than twice the *smallest* optimal learning rate for any component

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

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

Generic differentiable multivariate convex functions





- For generic convex multivariate functions (not necessarily quadratic), we can employ quadratic Taylor series expansions and much of the analysis still applies
- Taylor expansion

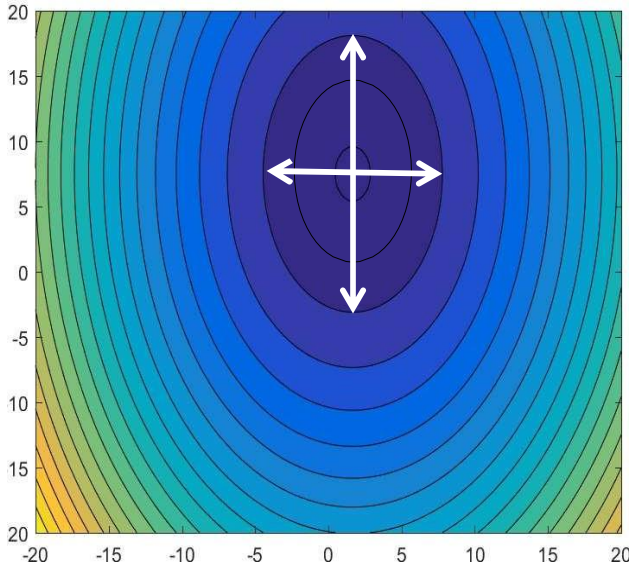
$$E(\mathbf{w}) \approx E\left(\mathbf{w}^{(k)}\right) + \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)\left(\mathbf{w} - \boldsymbol{w}^{(k)}\right) + \frac{1}{2}\left(\mathbf{w} - \boldsymbol{w}^{(k)}\right)^{T} \boldsymbol{H}_{E}\left(\boldsymbol{w}^{(k)}\right)\left(\mathbf{w} - \boldsymbol{w}^{(k)}\right)$$

- The optimal step size is inversely proportional to the Eigen values of the Hessian

Convergence

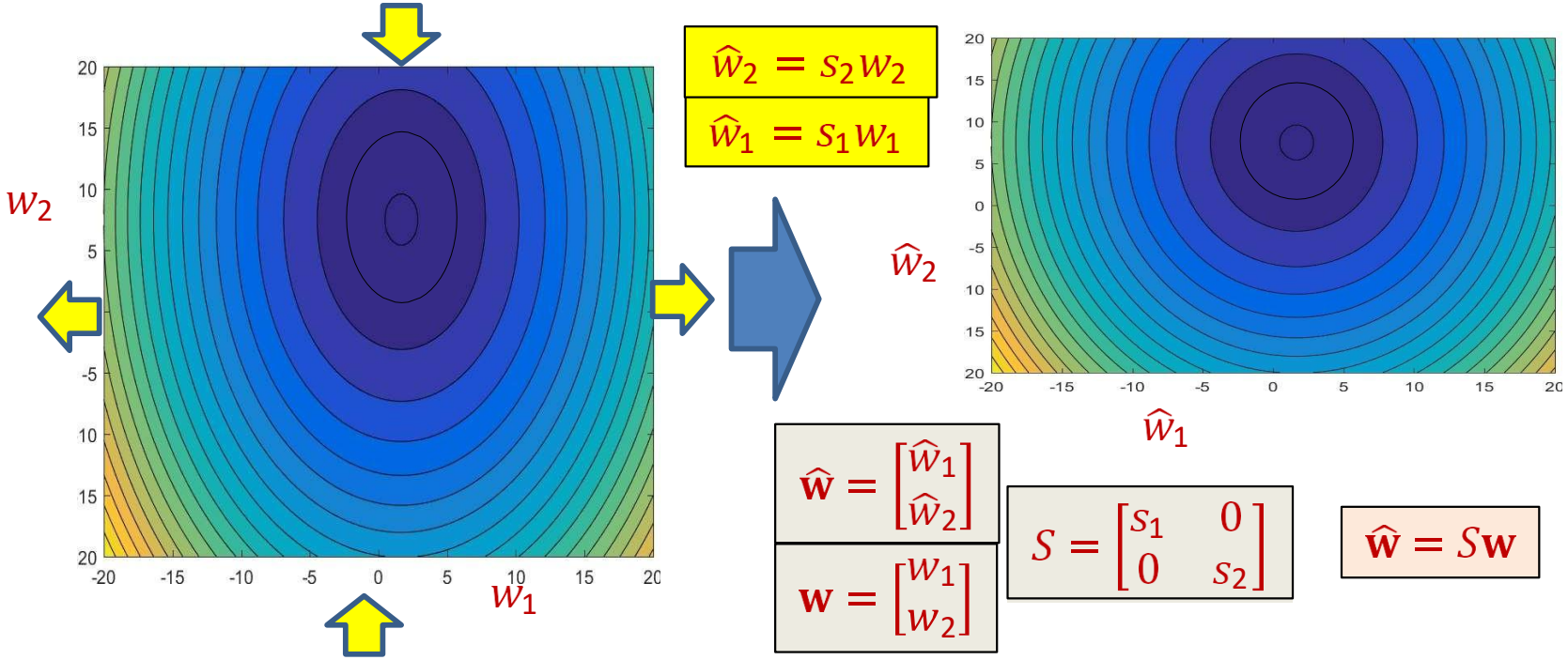
- Convergence behaviors become increasingly unpredictable as dimensions increase
- For the fastest convergence, ideally, the learning rate η must be close to both, the largest $\eta_{i,opt}$ and the smallest $\eta_{i,opt}$
  - To ensure convergence in every direction
  - Generally infeasible
- Convergence is particularly slow if $\frac{\max_i \eta_{i,opt}}{\min_i \eta_{i,opt}}$ is large
  - The "condition" number
    - Must be close to 1.0 for fast convergence

One reason for the problem



- The objective function has different eccentricities in different directions
  - Resulting in different optimal learning rates for different directions
- Solution: *Normalize* the objective to have identical eccentricity in all directions
  - Then all of them will have identical optimal learning rates
  - Easier to find a working learning rate

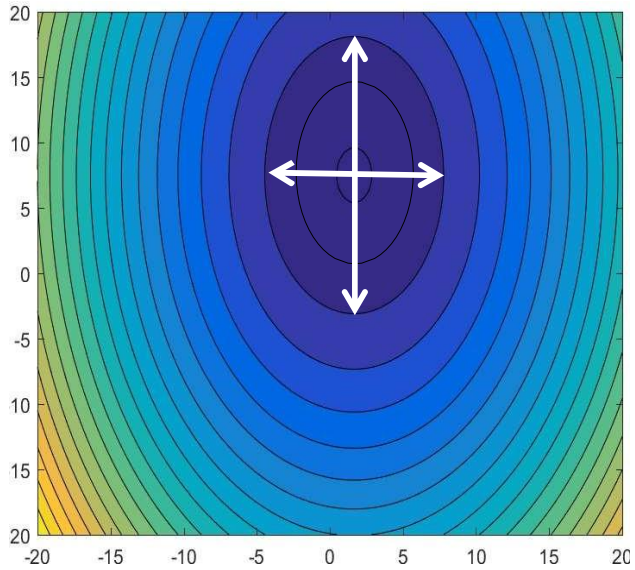
Solution: Scale the axes

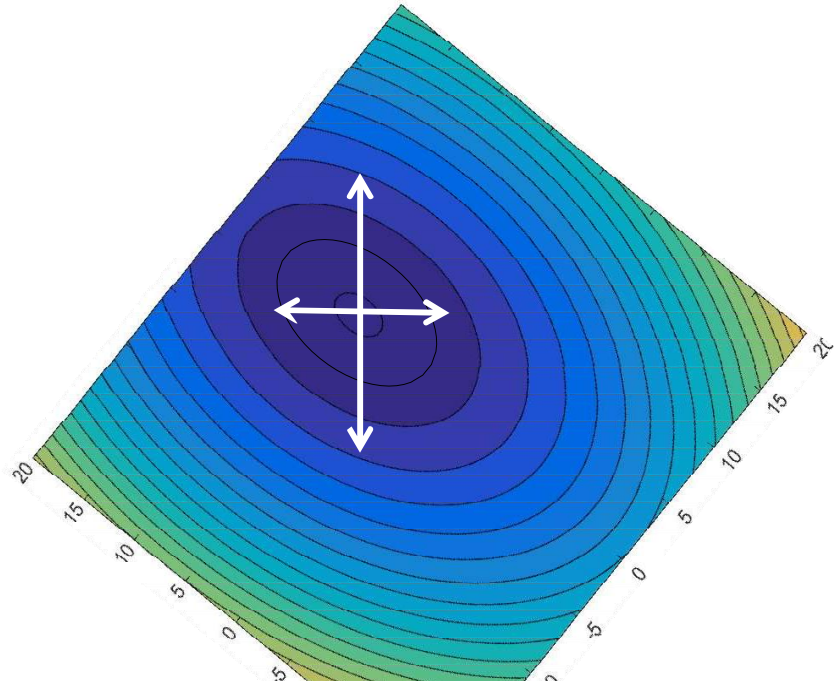


- Scale (and rotate) the axes, such that all of them have identical (identity) "spread"
  - Equal-value contours are circular
  - Movement along the coordinate axes become independent
- **Note:** equation of a quadratic surface with circular equal-value contours can be written as

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

One reason for the problem

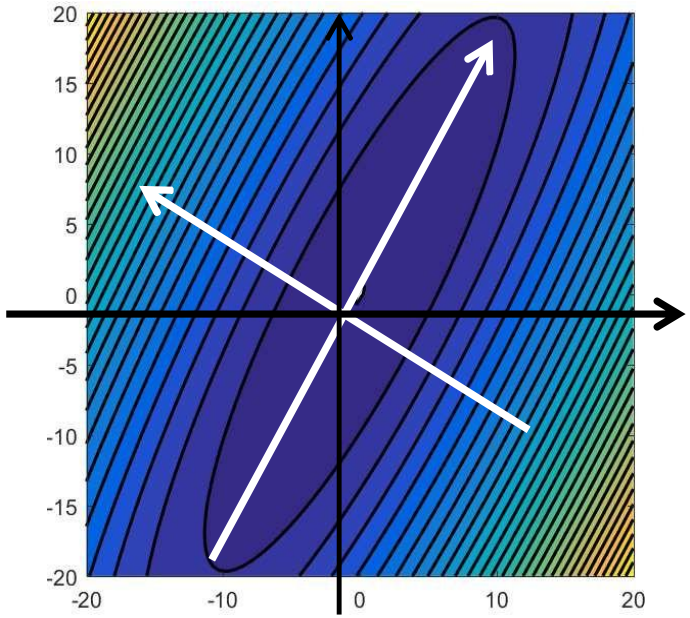




- The objective function has different eccentricities in different directions
  - Resulting in different optimal learning rates for different directions
- Solution: *Normalize* the objective to have identical eccentricity in all directions
  - Then all of them will have identical optimal learning rates
  - Easier to find a working learning rate

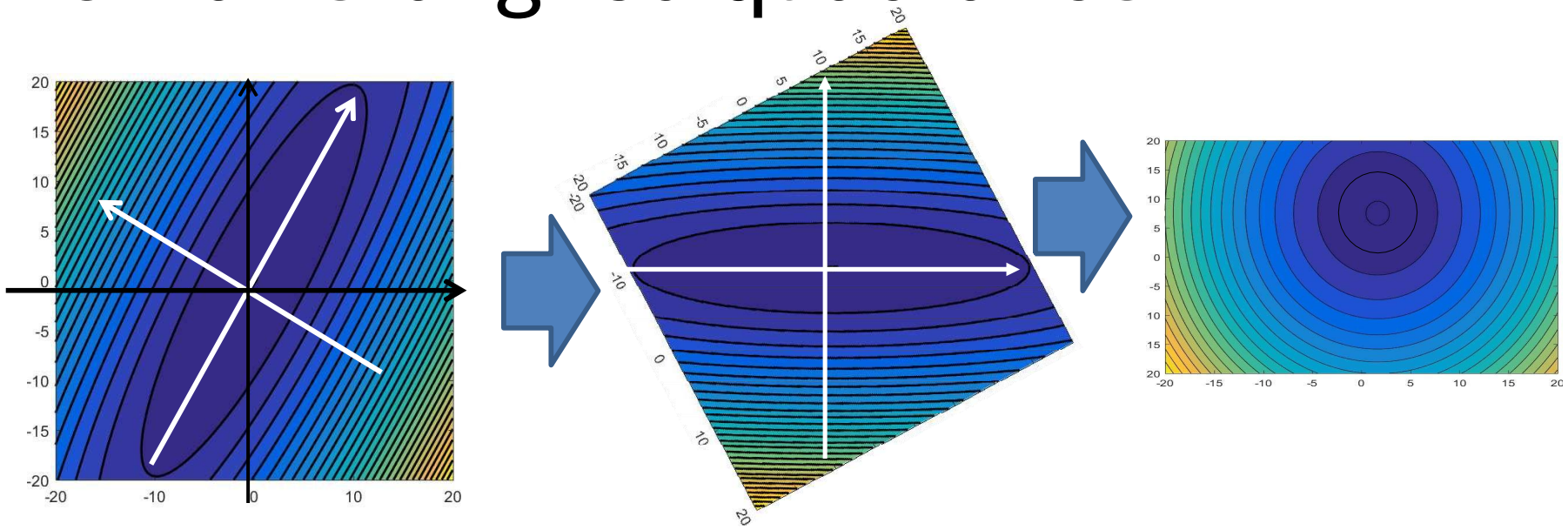
- The problem is more difficult when the ellipsoid is not axis aligned: the steps along the two directions are coupled! Moving in one direction changes the gradient along the other

For non-axis-aligned quadratics



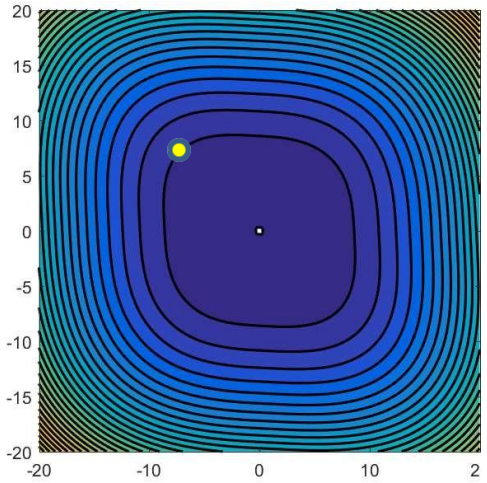
- If $\mathbf{A}$ is not diagonal, the contours are not axis-aligned
  - Because of the cross-terms $a_{ij}w_i w_j$
  - The major axes of the ellipsoids are the *Eigenvectors* of $\mathbf{A}$, and their diameters are proportional to the Eigen values of $\mathbf{A}$

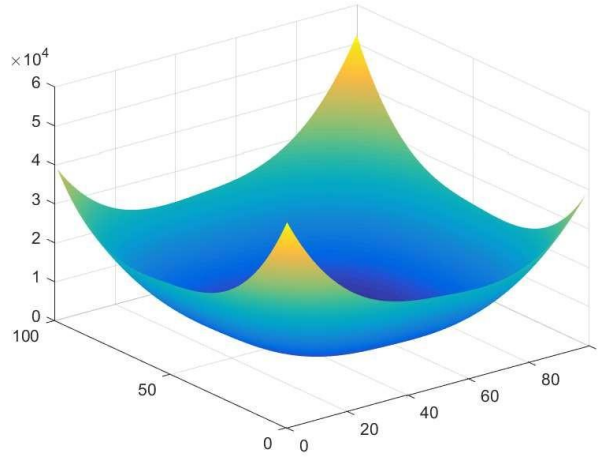
For non-axis-aligned quadratics



- 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 by rotating and resizing the different directions

Generic differentiable multivariate convex functions





- Taylor expansion

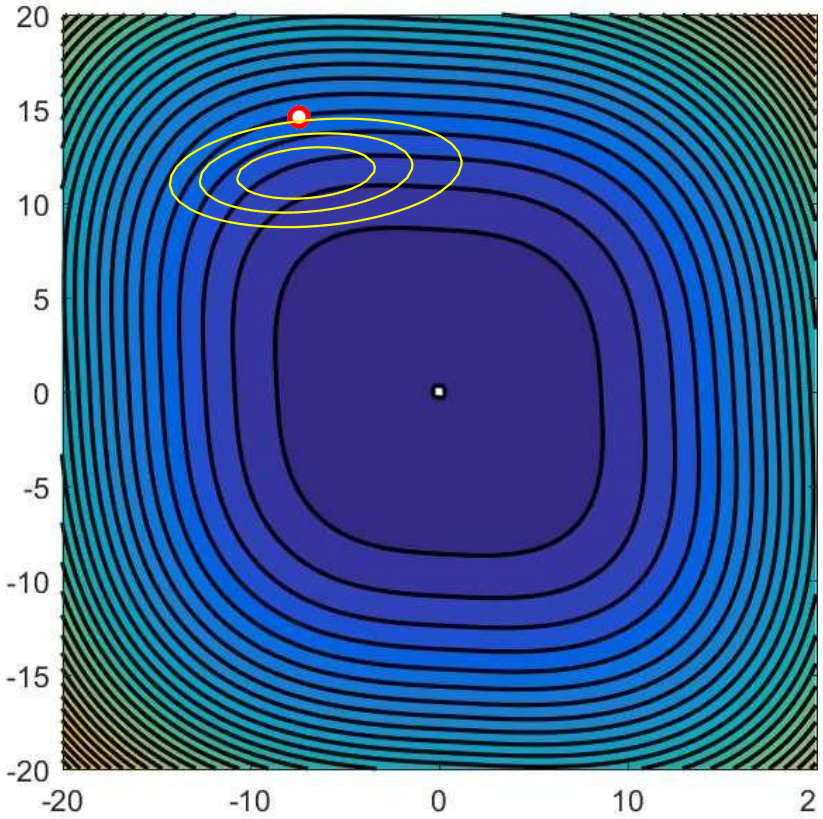
$$E(\mathbf{w}) \approx E\left(\mathbf{w}^{(k)}\right) + \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)\left(\mathbf{w} - \boldsymbol{w}^{(k)}\right) + \frac{1}{2}\left(\mathbf{w} - \boldsymbol{w}^{(k)}\right)^{T} \boldsymbol{H}_{E}\left(\boldsymbol{w}^{(k)}\right)\left(\mathbf{w} - \boldsymbol{w}^{(k)}\right)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

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

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

- **For 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



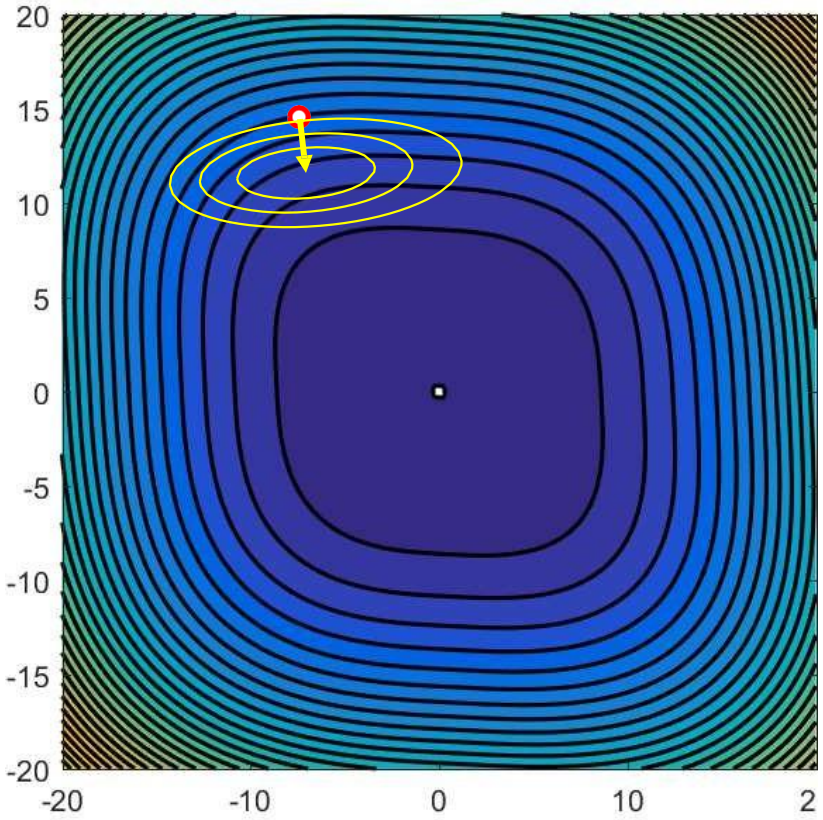
Fit a quadratic at each point and find the minimum of that quadratic

- Iterated localized optimization with quadratic approximations

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

with $\eta = 1$

Minimization by Newton's method

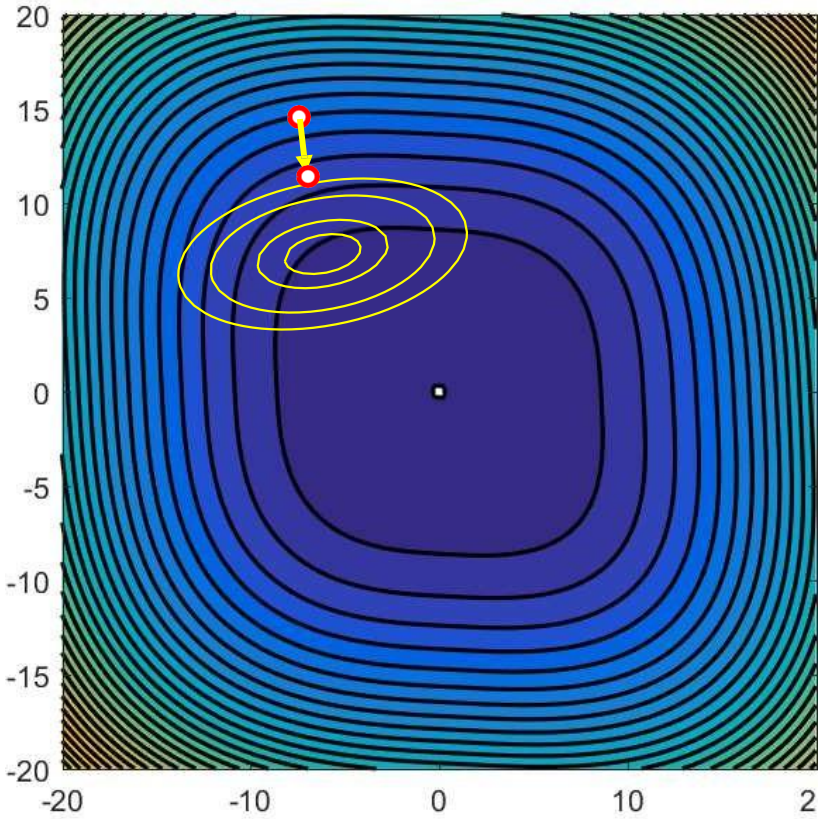


- Iterated localized optimization with quadratic approximations

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

with $\eta = 1$

Minimization by Newton's method

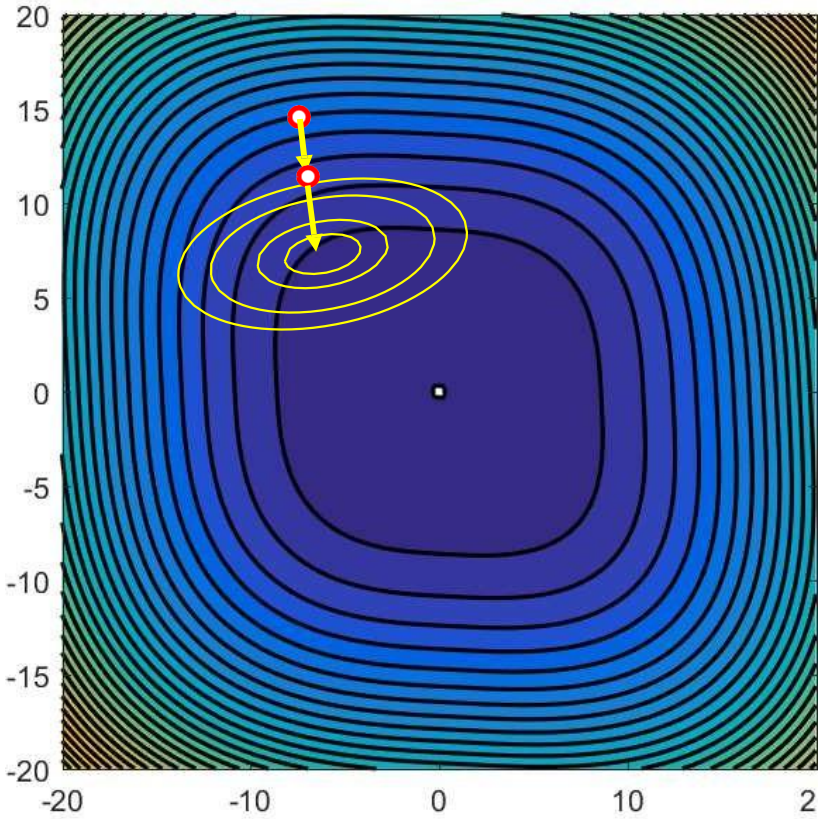


- Iterated localized optimization with quadratic approximations

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

with $\eta = 1$

Minimization by Newton's method

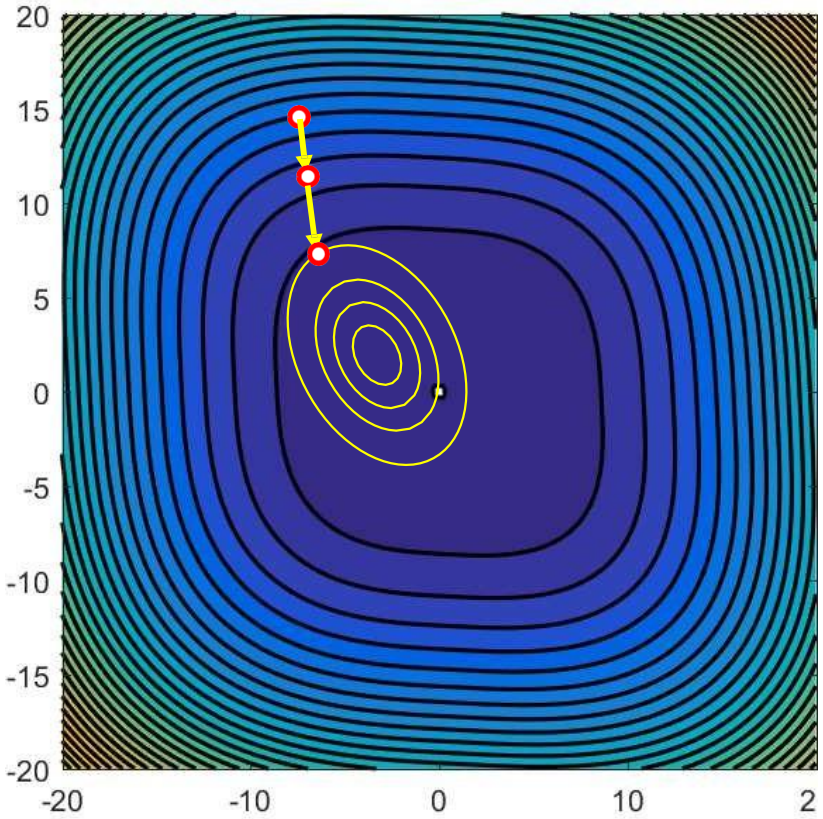


- Iterated localized optimization with quadratic approximations

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

with $\eta = 1$

Minimization by Newton's method

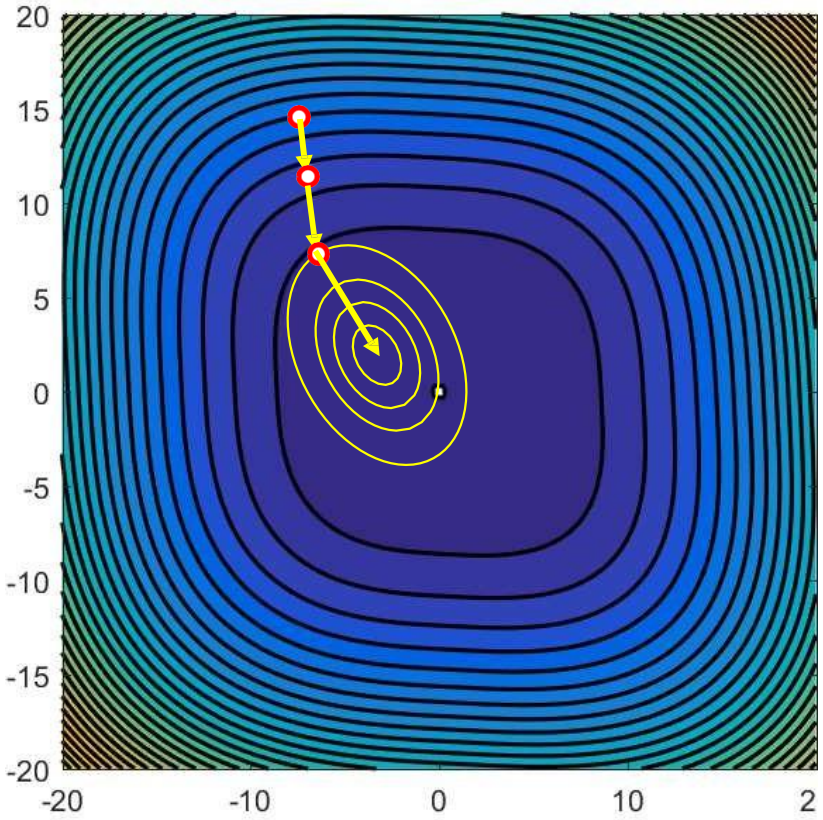


- Iterated localized optimization with quadratic approximations

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

with $\eta = 1$

Minimization by Newton's method

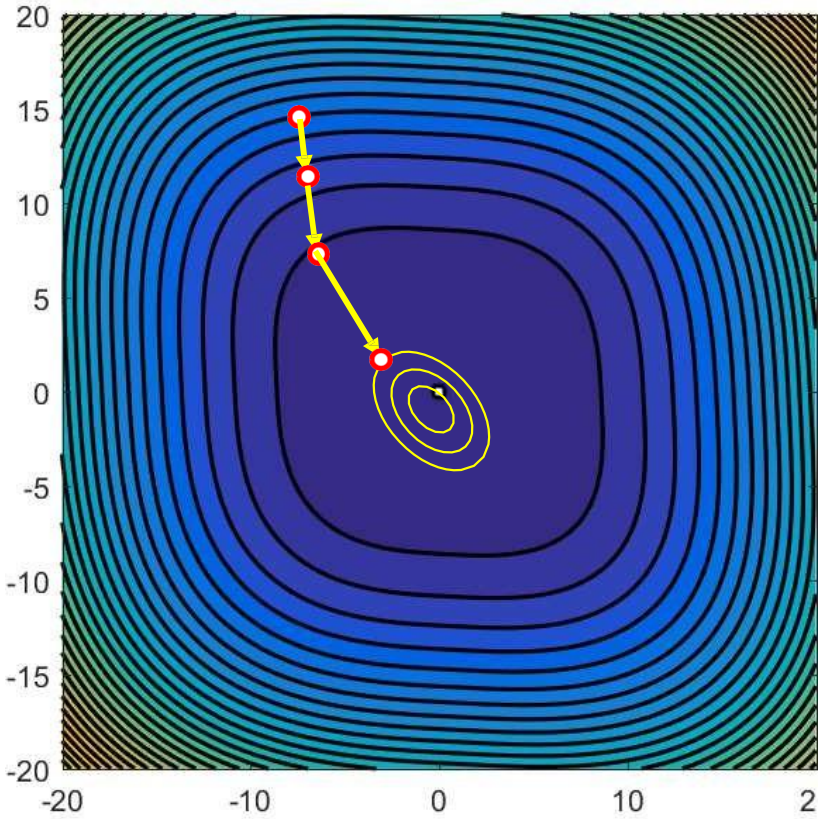


- Iterated localized optimization with quadratic approximations

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

with $\eta = 1$

Minimization by Newton's method

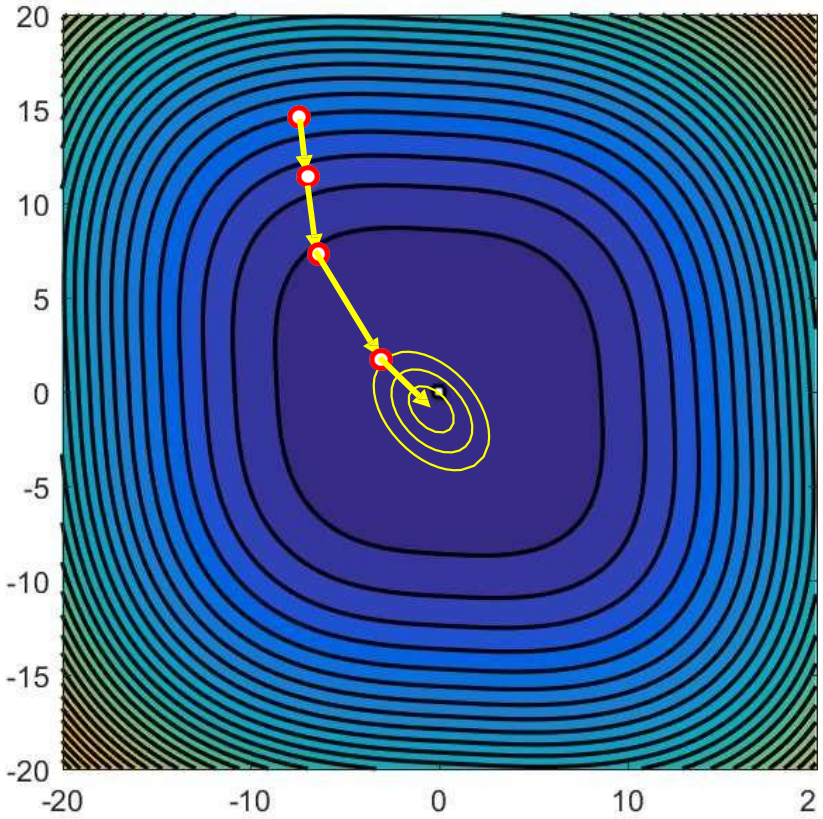


- Iterated localized optimization with quadratic approximations

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

with $\eta = 1$

Minimization by Newton's method

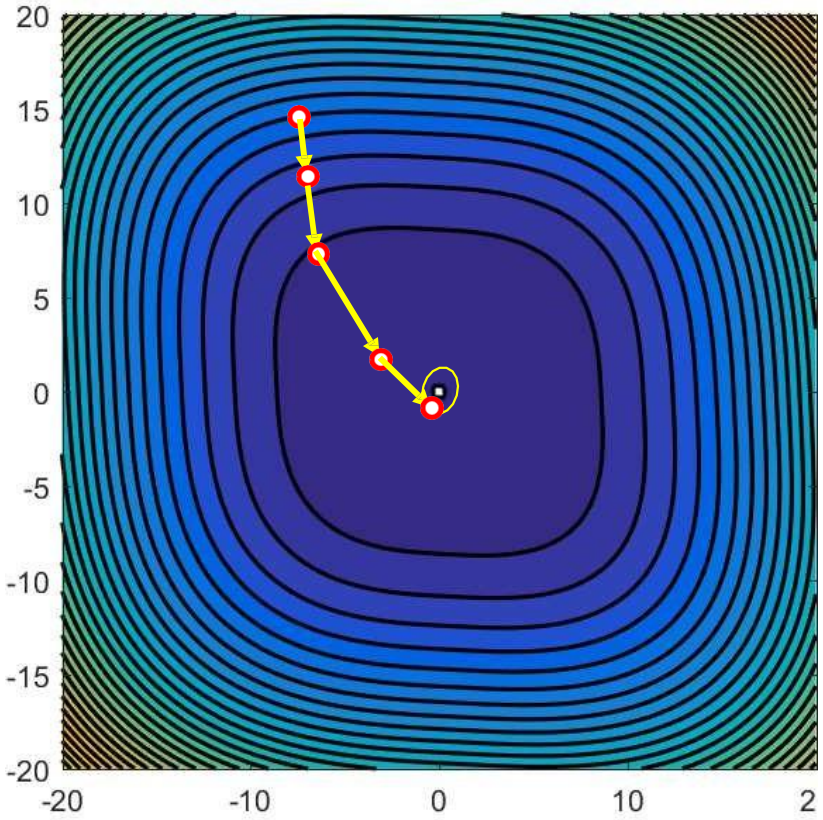


- Iterated localized optimization with quadratic approximations

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

with $\eta = 1$

Minimization by Newton's method

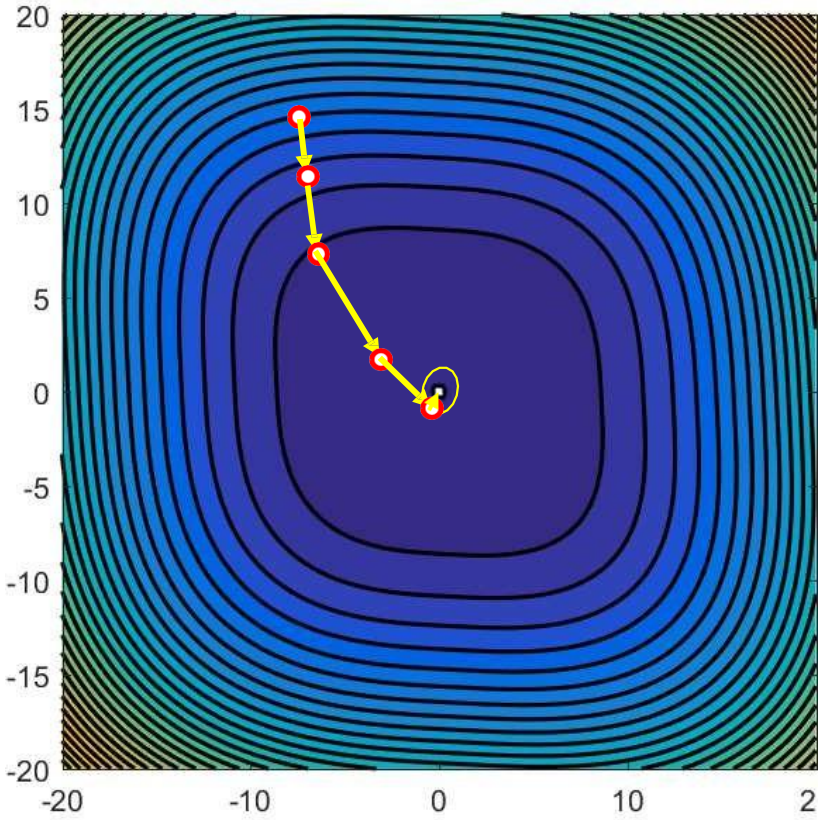


- Iterated localized optimization with quadratic approximations

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

with $\eta = 1$

Minimization by Newton's method

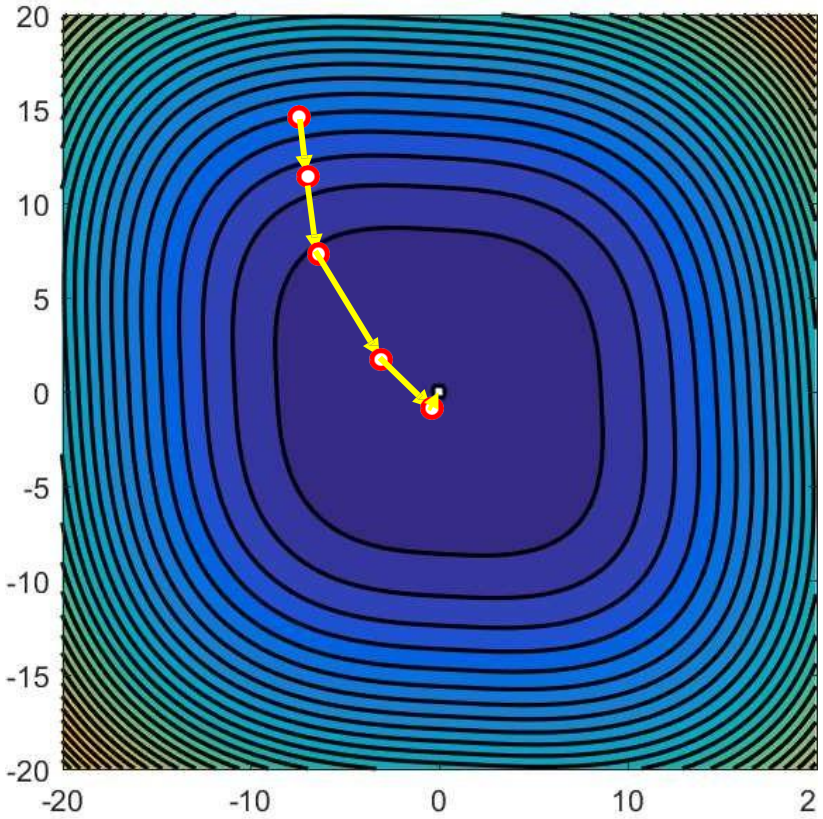


- Iterated localized optimization with quadratic approximations

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

with $\eta = 1$

Minimization by Newton's method



- Iterated localized optimization with quadratic approximations

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

with $\eta = 1$

Issues: 1. The Hessian

- Normalized update rule

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

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

Issues: 1. The Hessian

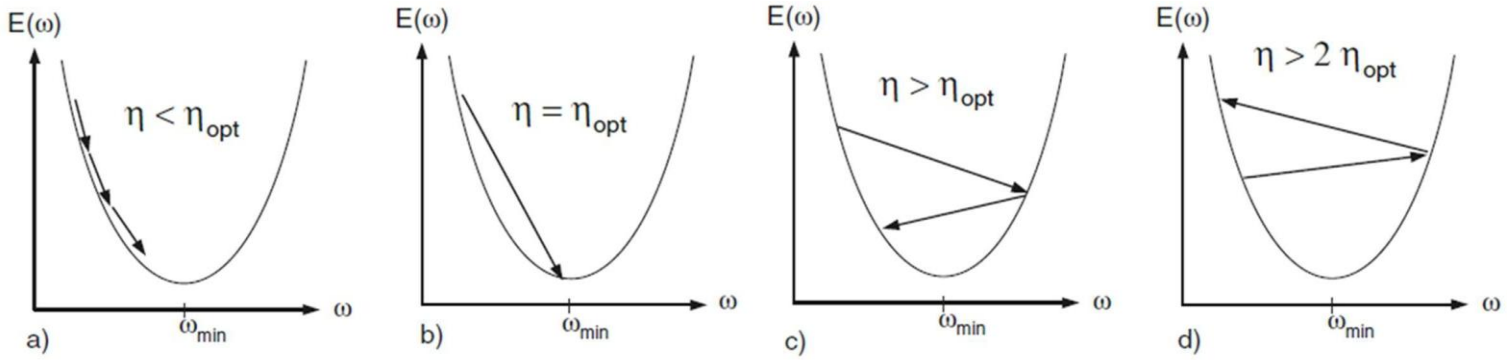
- For non-convex functions, the Hessian may not be positive semi-definite, in which case the algorithm can *diverge*
  - Goes away from, rather than towards the minimum
  - Now requires additional checks to avoid movement in directions corresponding to negative Eigenvalues of the Hessian

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

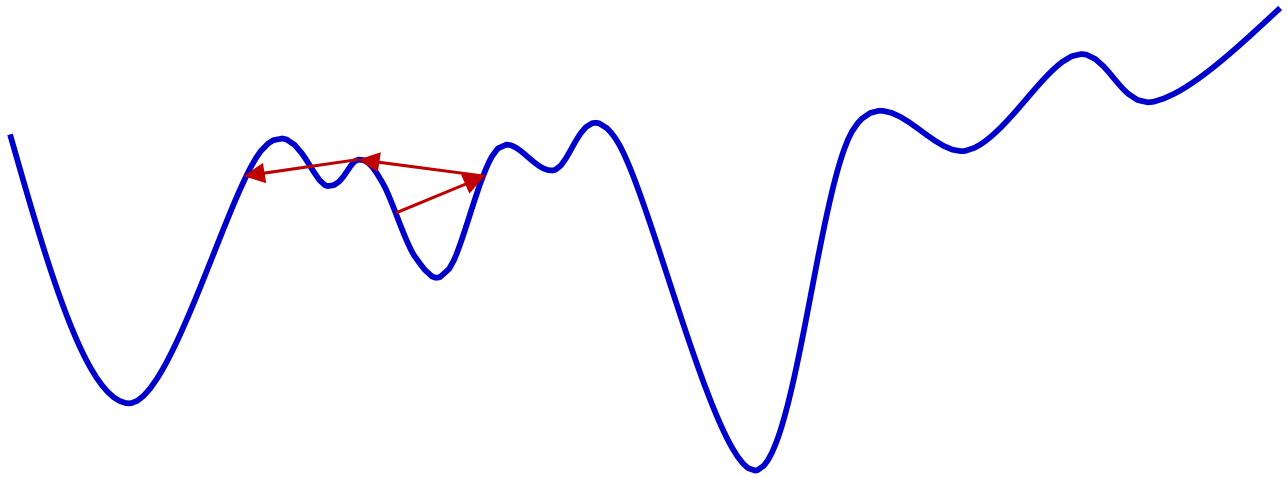
- Not particularly popular anymore for large neural networks..

Issues: 2. The learning rate



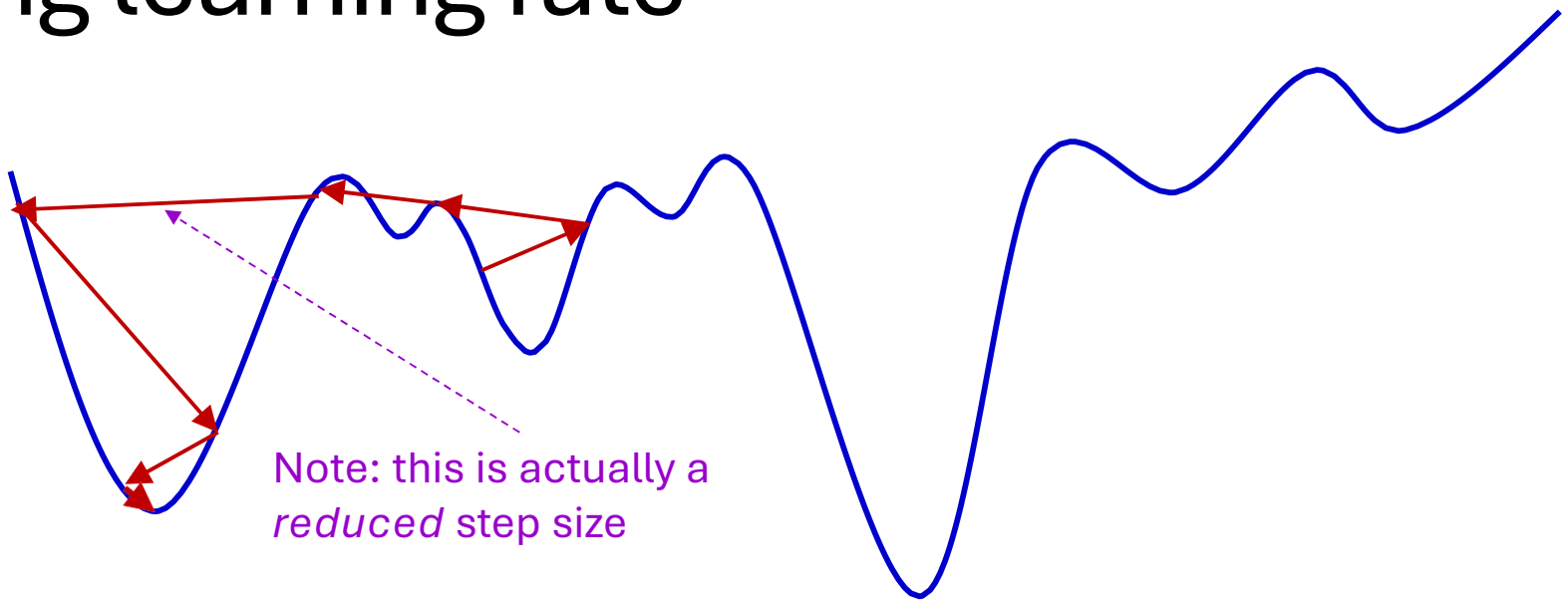
- Much of the analysis we just saw was based on trying to ensure that the step size was not so large as to cause divergence within a convex region

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



- 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 make you never ever actually find a solution

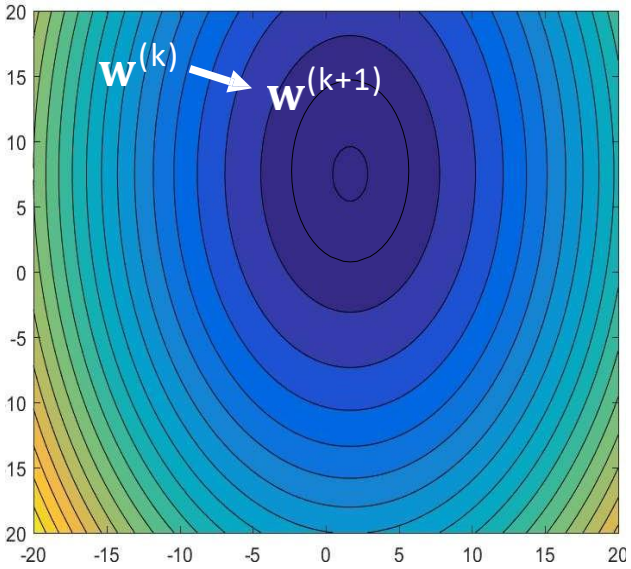
Decaying learning rate



- Start with a large learning rate
  - Gradually reduce it with iterations
- 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 practical 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

Let's take a step back



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

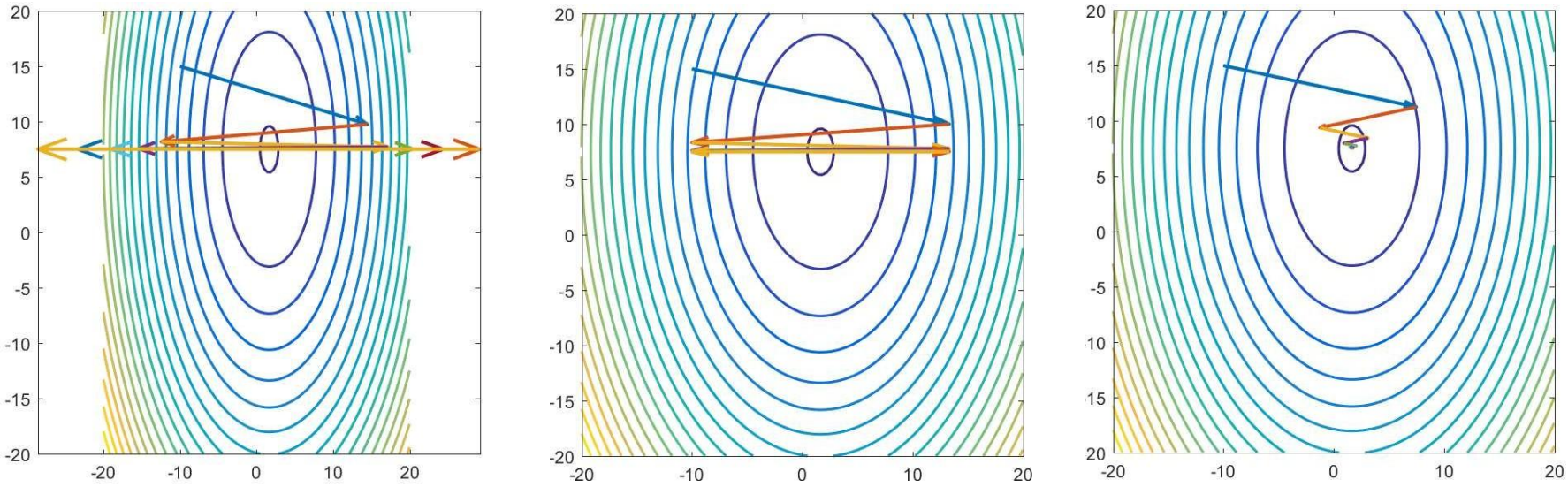
$$w_i^{(k+1)} = w_i^{(k)} - \eta \frac{dE\left(w_i^{(k)}\right)}{d\mathrm{w}}$$

- Many of the convergence issues arise because we force the same learning rate on all parameters
  - We require a fixed step size across all dimensions
  - ***Because steps are "tied" to the gradient***
- Try releasing this requirement?

Derivative-inspired algorithms

- Algorithms that ***use derivative information for trends, but do not follow them absolutely***
- For example:
  - Resilient propagation (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
  - Quick prop
    - 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

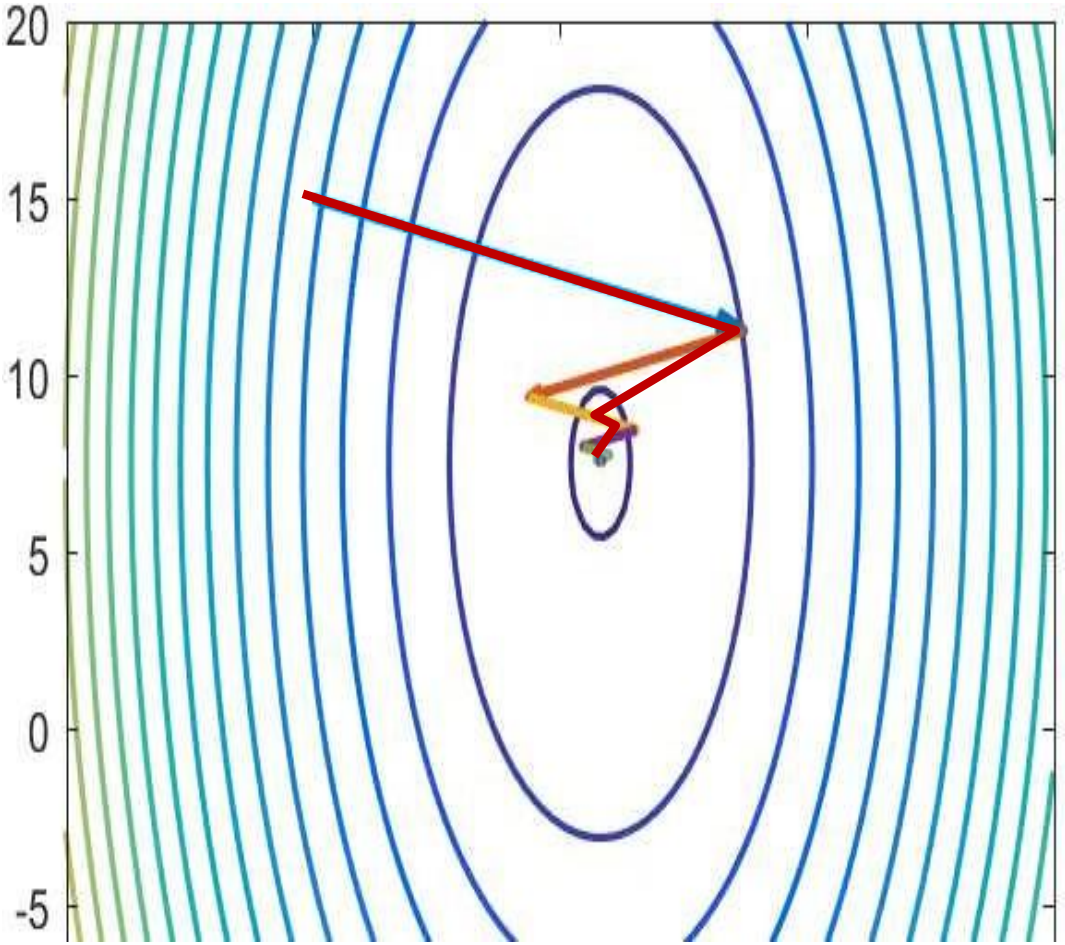
A closer look at the convergence



- 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..

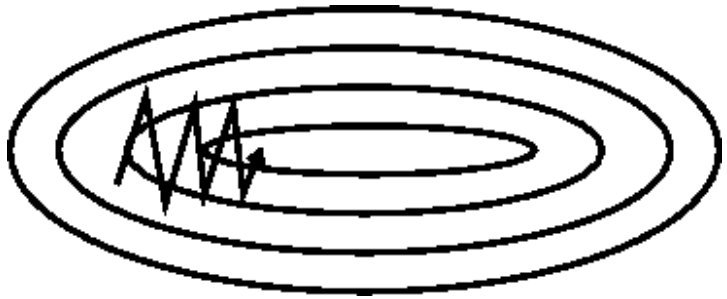
The momentum methods

- 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

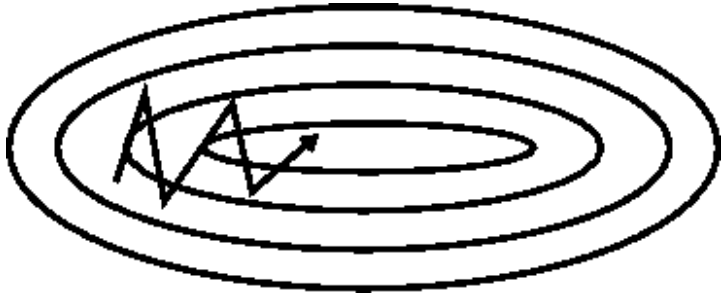


Momentum Update

Plain gradient update



With momentum



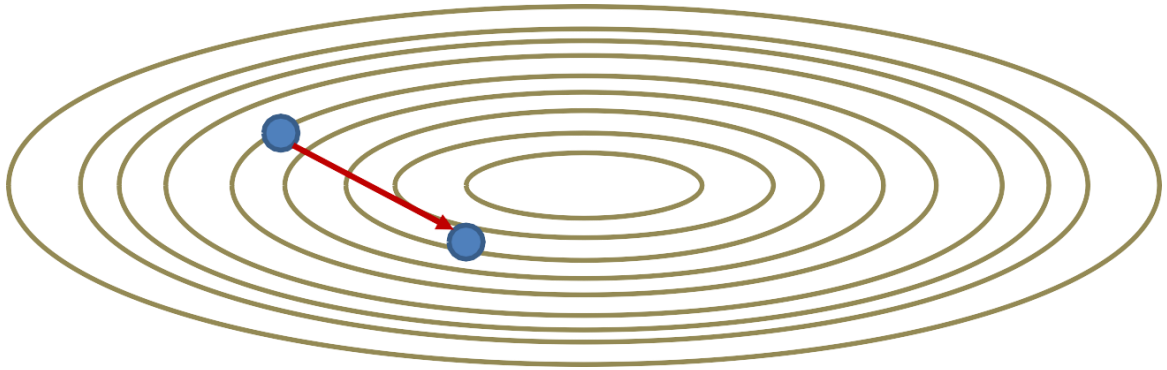
- The momentum method maintains a running average of all gradients until the current step

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

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

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

Momentum Update

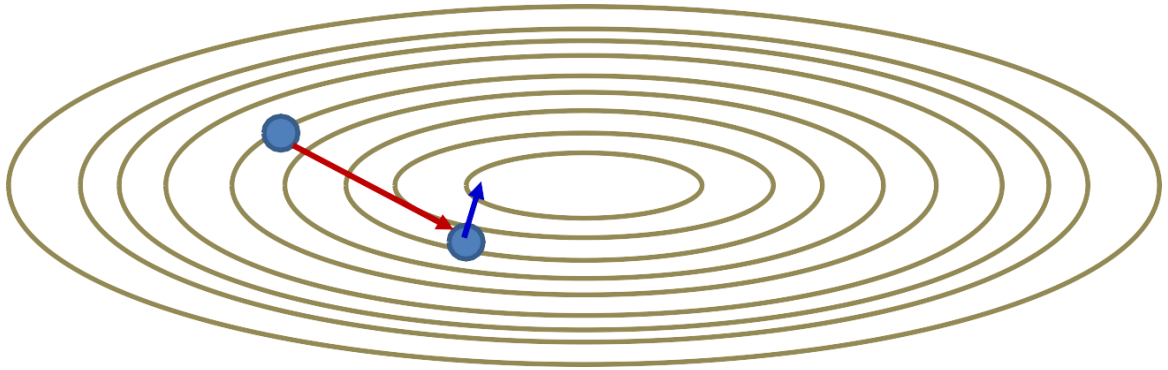


- The momentum method

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

- At any iteration, to compute the current step:

Momentum Update

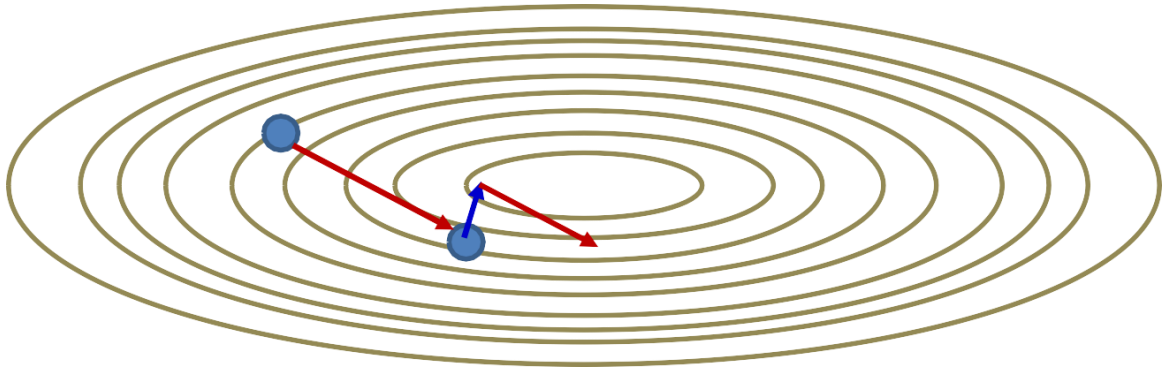


- The momentum method

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

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

Momentum Update

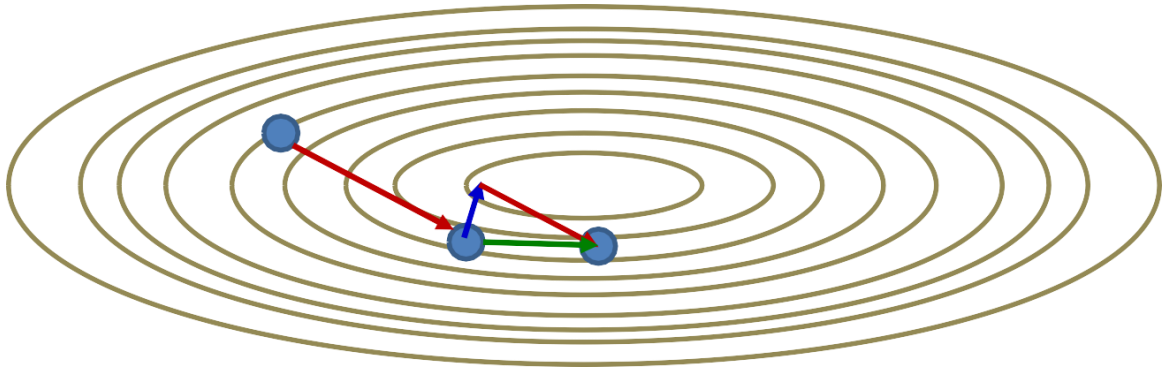


- The momentum method

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

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

Momentum Update

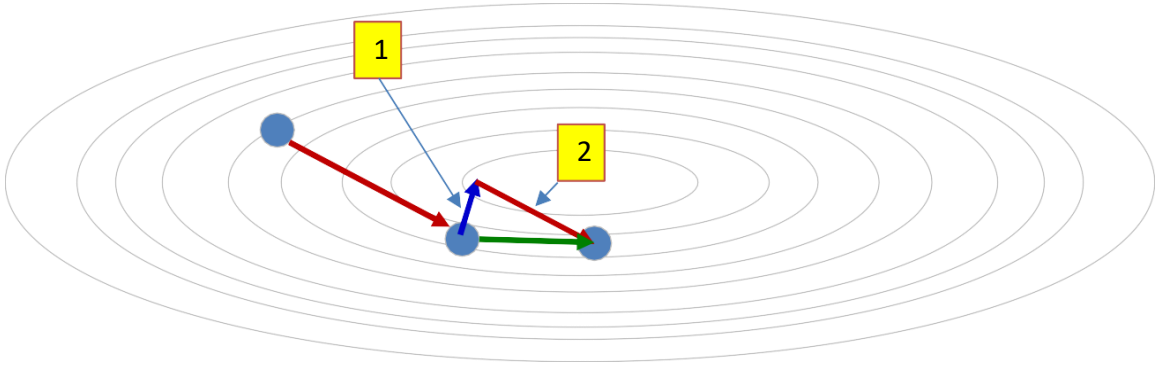


- The momentum method

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

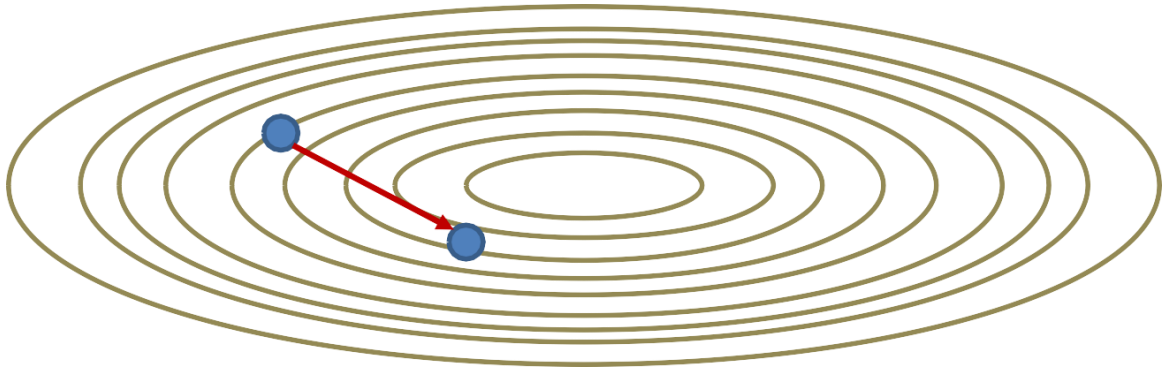
- At any iteration, to compute the current step:
  - First computes the gradient step at the current location
  - Then adds in the scaled *previous* step
    - Which is actually a running average
  - To get the final step

Momentum update



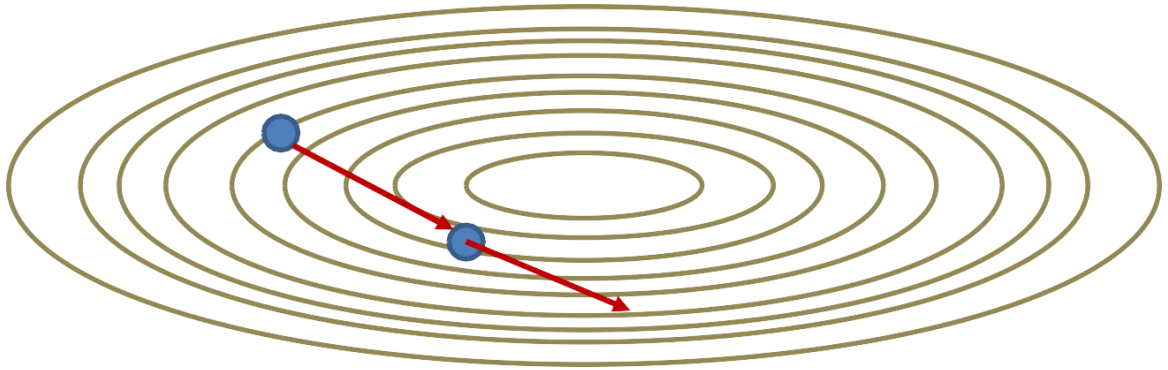
- Momentum update steps are actually computed in two stages
  - First: We take a step against the gradient at the current location
  - Second: Then we add a scaled version of the previous step
- The procedure can be made more optimal by reversing the order of operations..

Nestorov's Accelerated Gradient



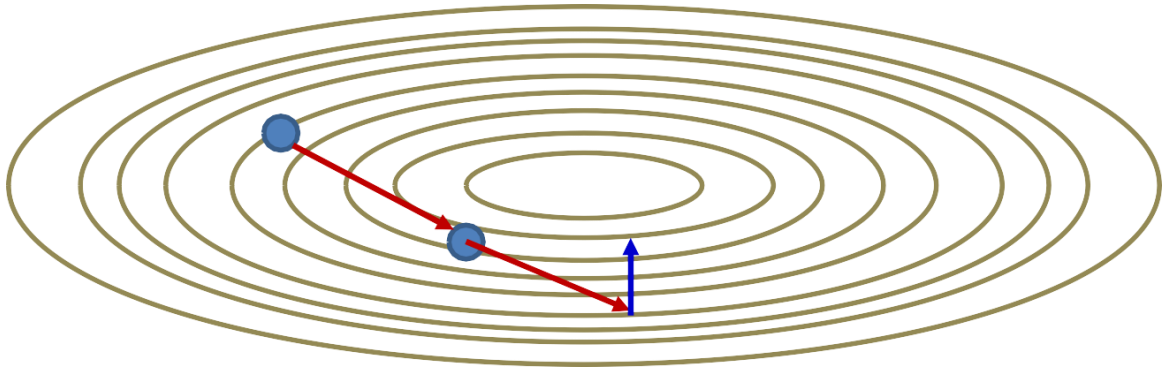
- Change the order of operations
- At any iteration, to compute the current step:

Nestorov's Accelerated Gradient



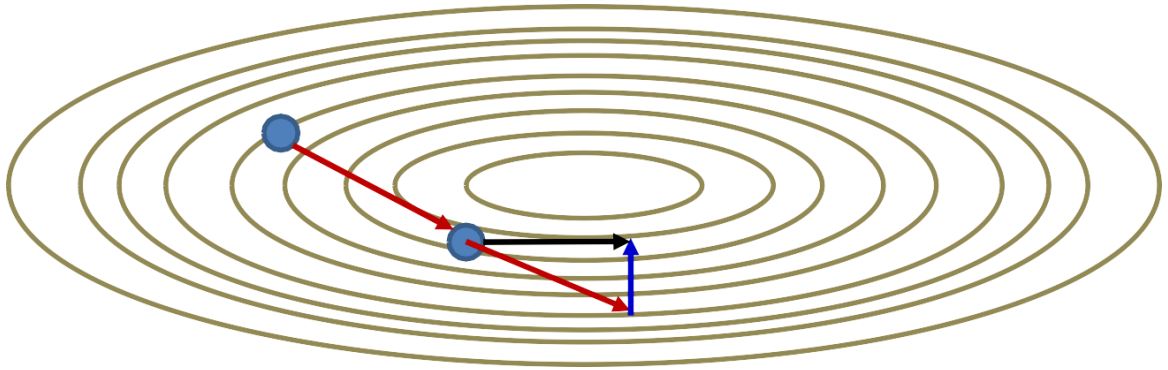
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step

Nestorov's Accelerated Gradient



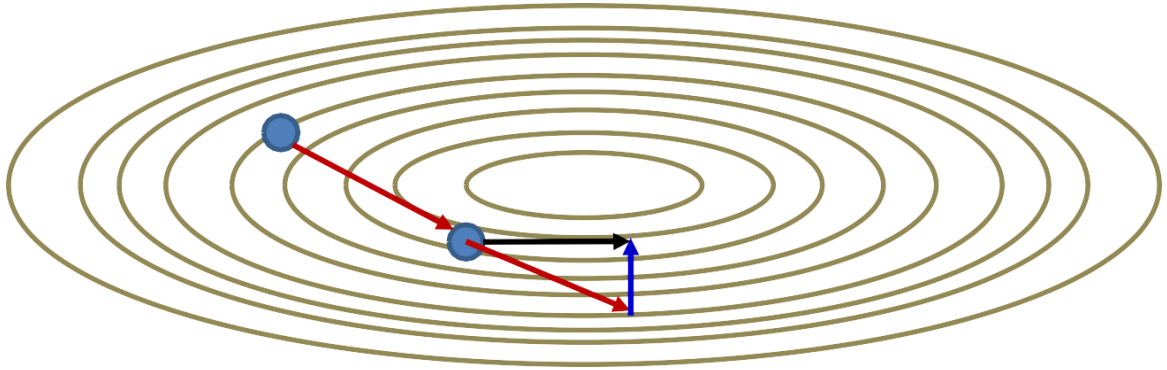
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step
  - Then compute the gradient step at the resultant position

Nestorov's Accelerated Gradient



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

Nestorov's Accelerated Gradient

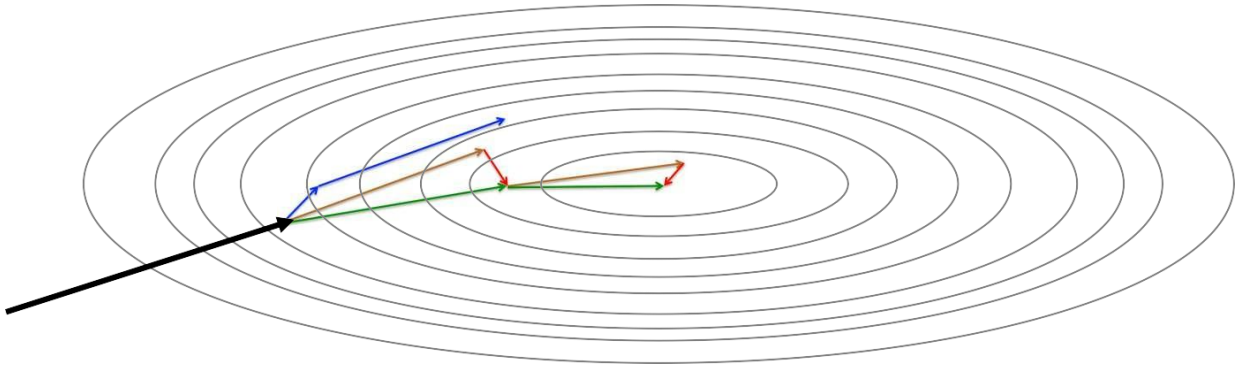


- Nestorov's method

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

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

Nestorov's Accelerated Gradient



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

Summary

- Gradient descent can miss obvious answers
  - And this may not be *a bad* thing
- Vanilla gradient descent may be too slow or unstable due to the differences between the dimensions
- Methods that normalize/decouple the dimensions can improve convergence; Adaptive or decaying learning rates can improve convergence
- Momentum methods which emphasize directions of steady improvement are demonstrably superior to other methods