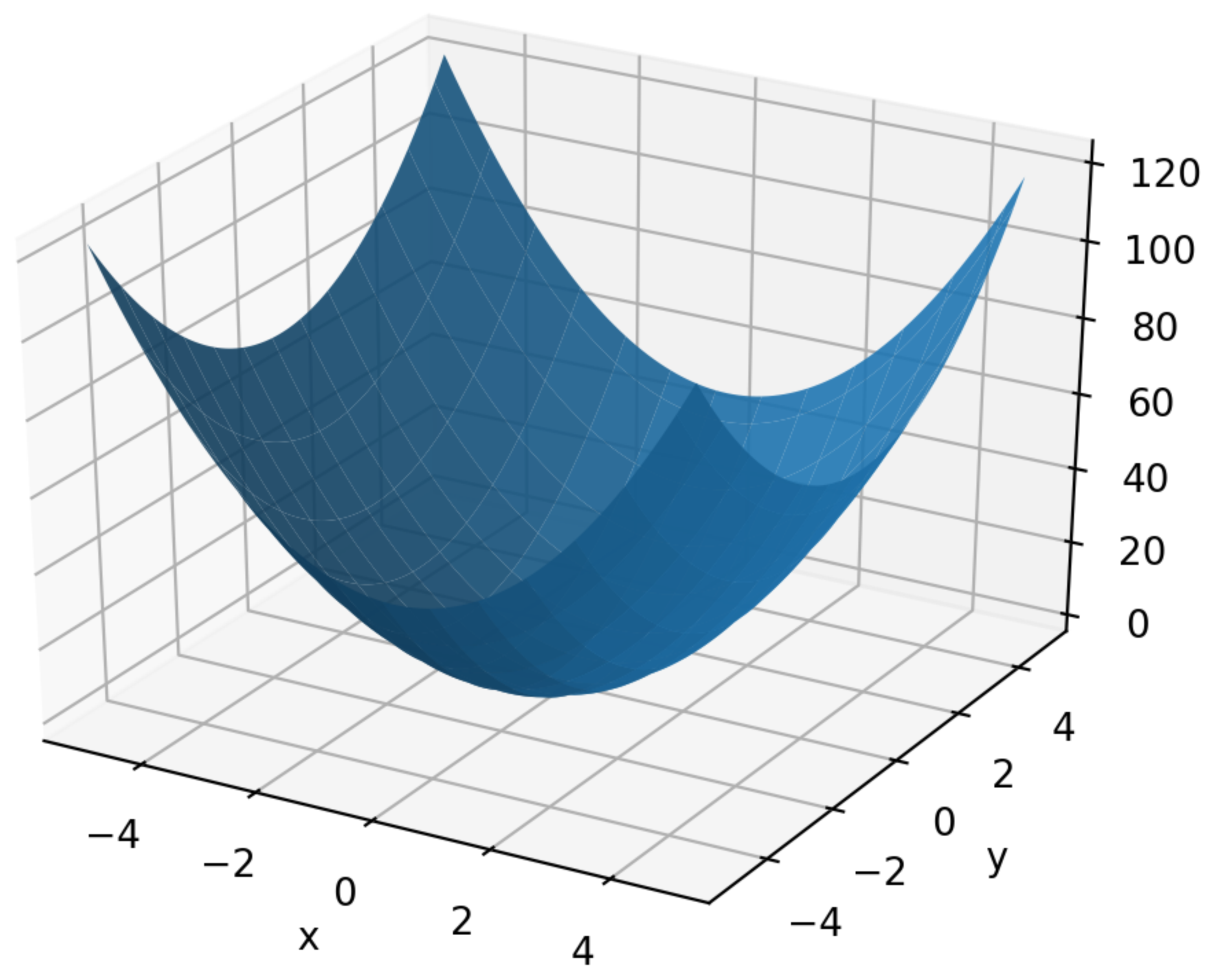
# CS/DS 541: Class 4

Jacob Whitehill

# Optimization of ML models

- With linear regression, the cost function $f_{\mathrm{MSE}}$ has a single local minimum w.r.t. the weights **w**:



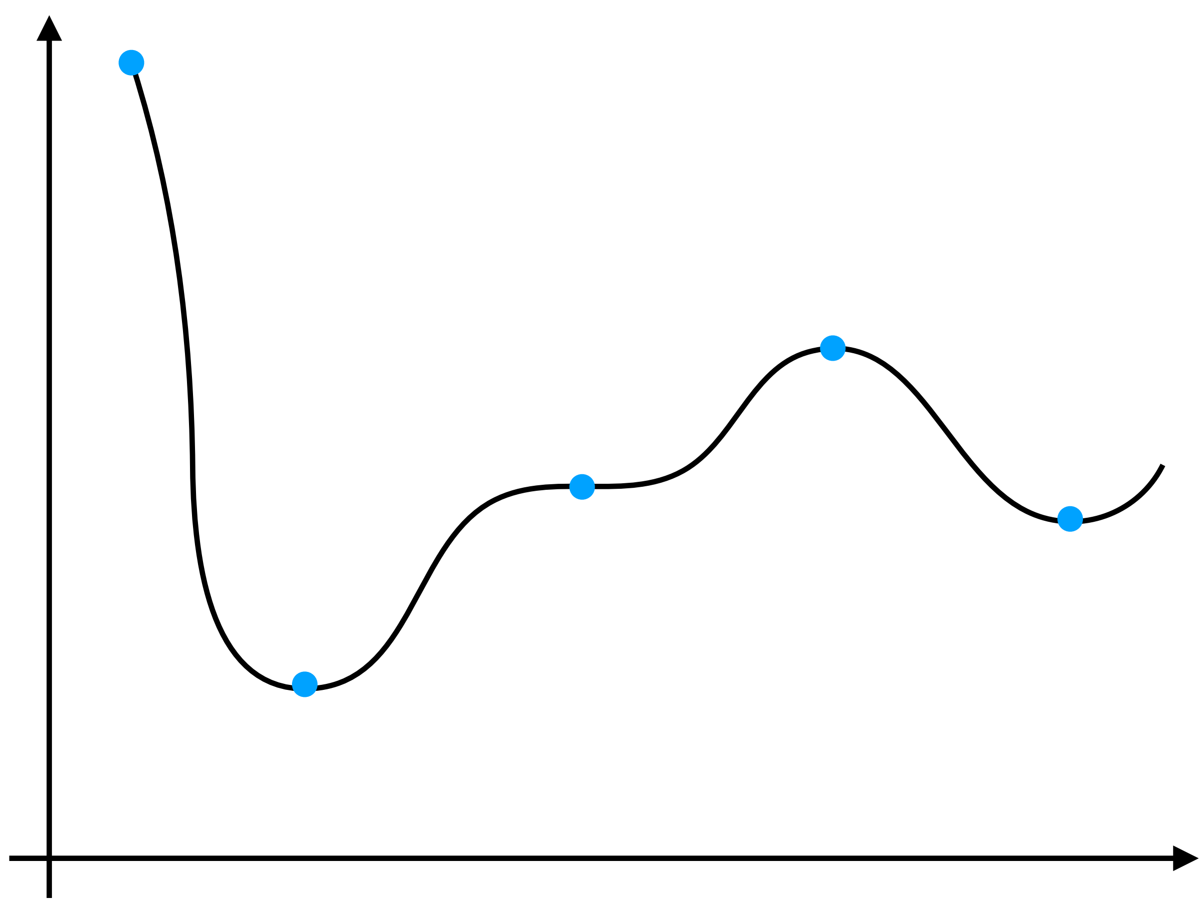

- As long as our learning rate is small enough, we will eventually find **the optimal w**.

# Optimization: what can go wrong?

- In general ML and DL models, optimization is usually not so simple, due to:

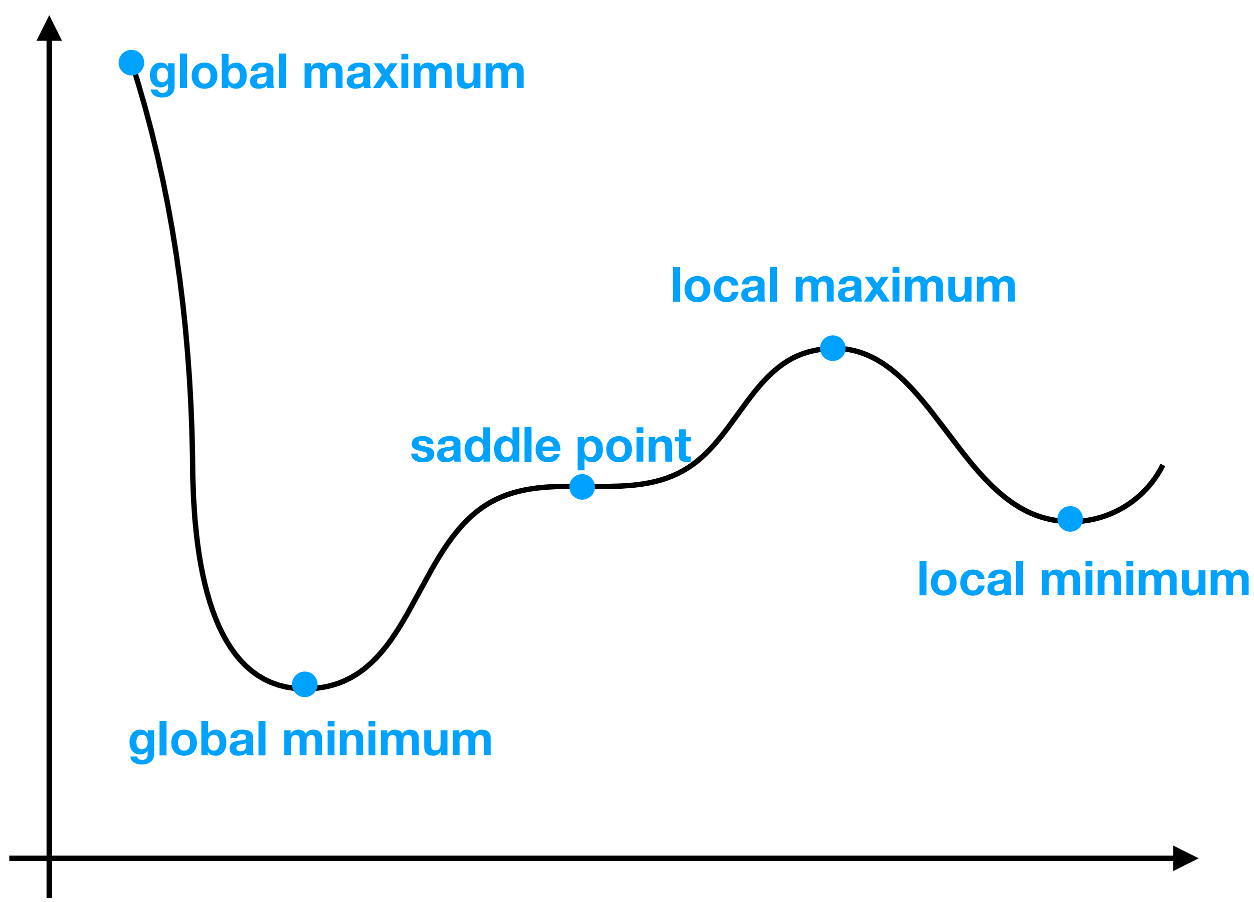
# Optimization: what can go wrong?

- In general ML and DL models, optimization is usually not so simple, due to:
   1. Presence of multiple local minima & saddle points



# Optimization: what can go wrong?

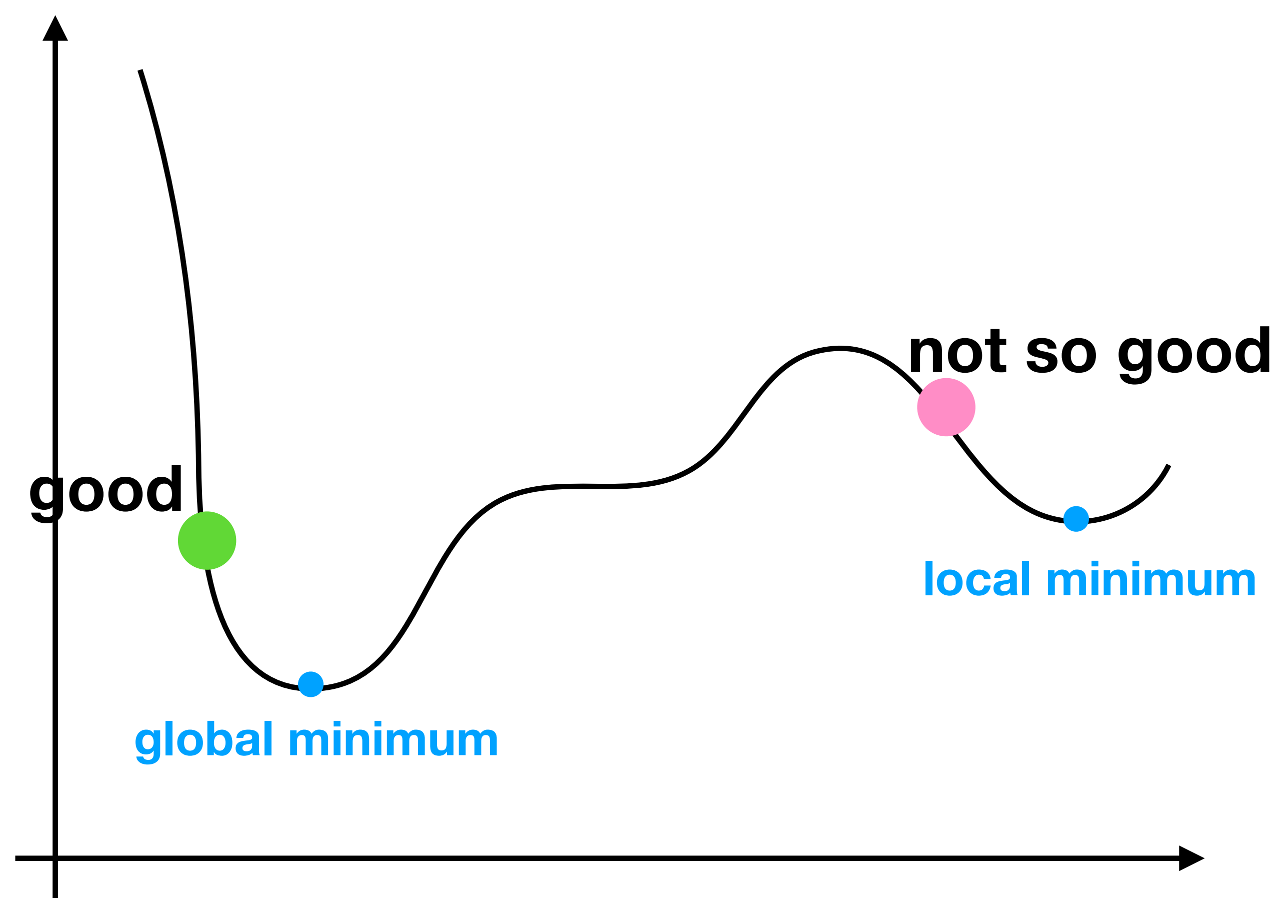
- In general ML and DL models, optimization is usually not so simple, due to:
    1. Presence of multiple local minima & saddle points



# Optimization: what can go wrong?

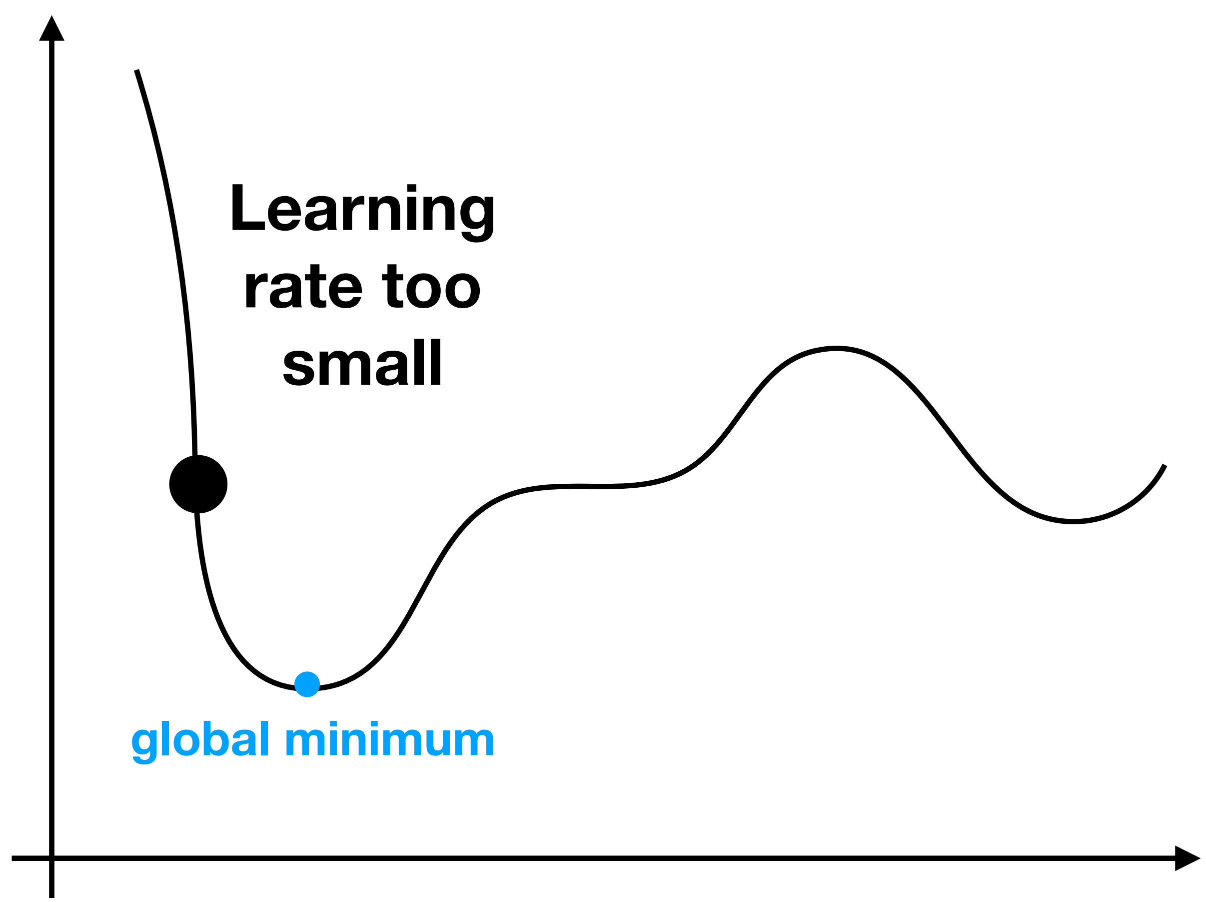
- In general ML and DL models, optimization is usually not so simple, due to:

  2. Bad initialization of the weights **w**.



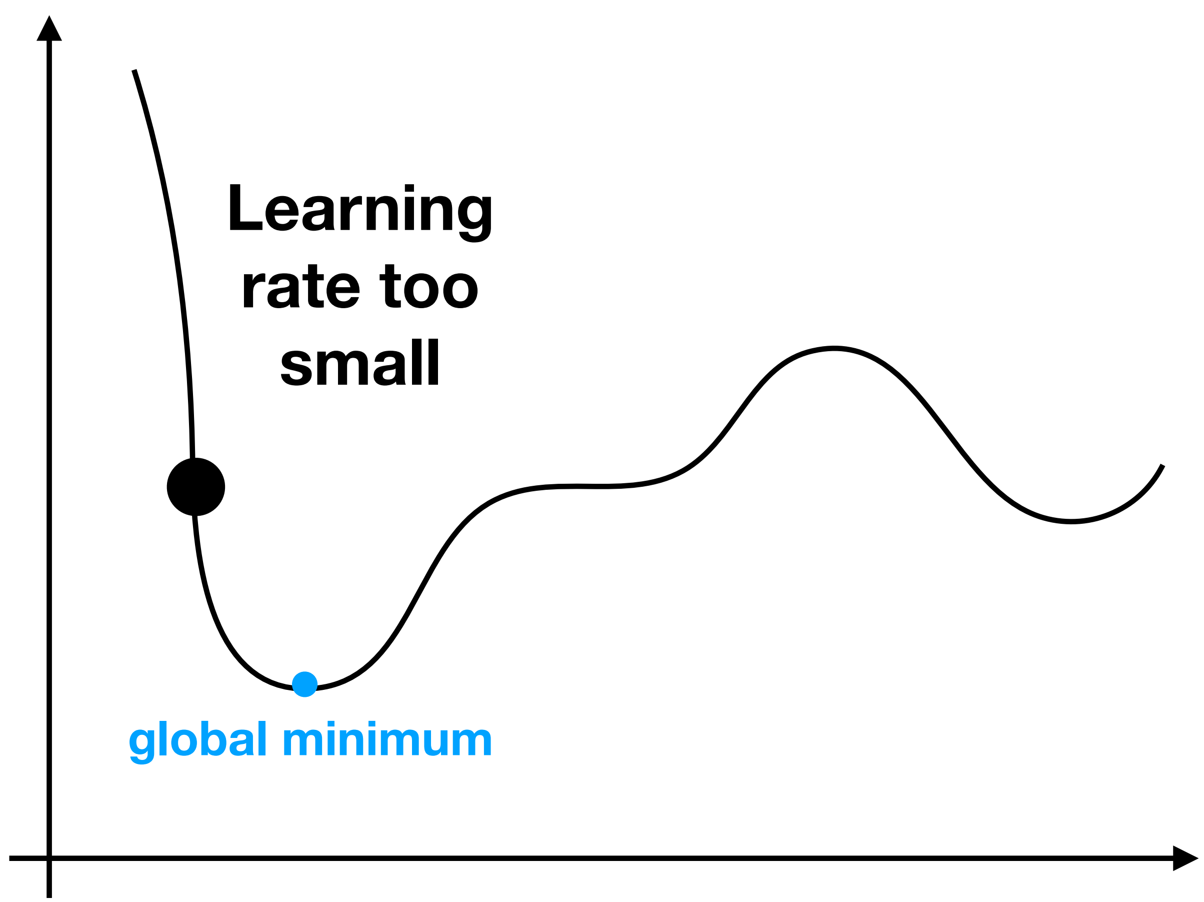
# Optimization: what can go wrong?

- In general ML and DL models, optimization is usually not so simple, due to:
  3. Learning rate is too small.



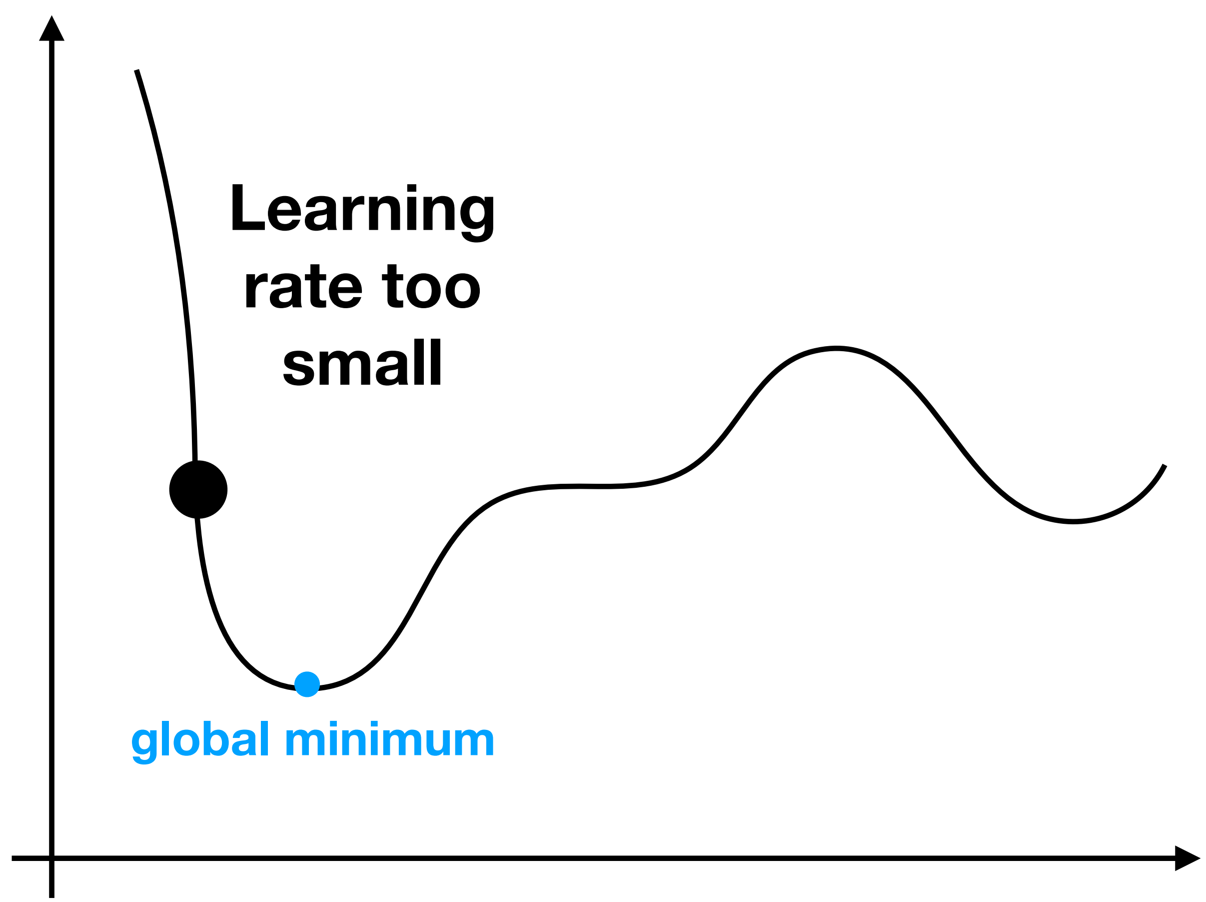
# Optimization: what can go wrong?

- In general ML and DL models, optimization is usually not so simple, due to:
  3. Learning rate is too small.



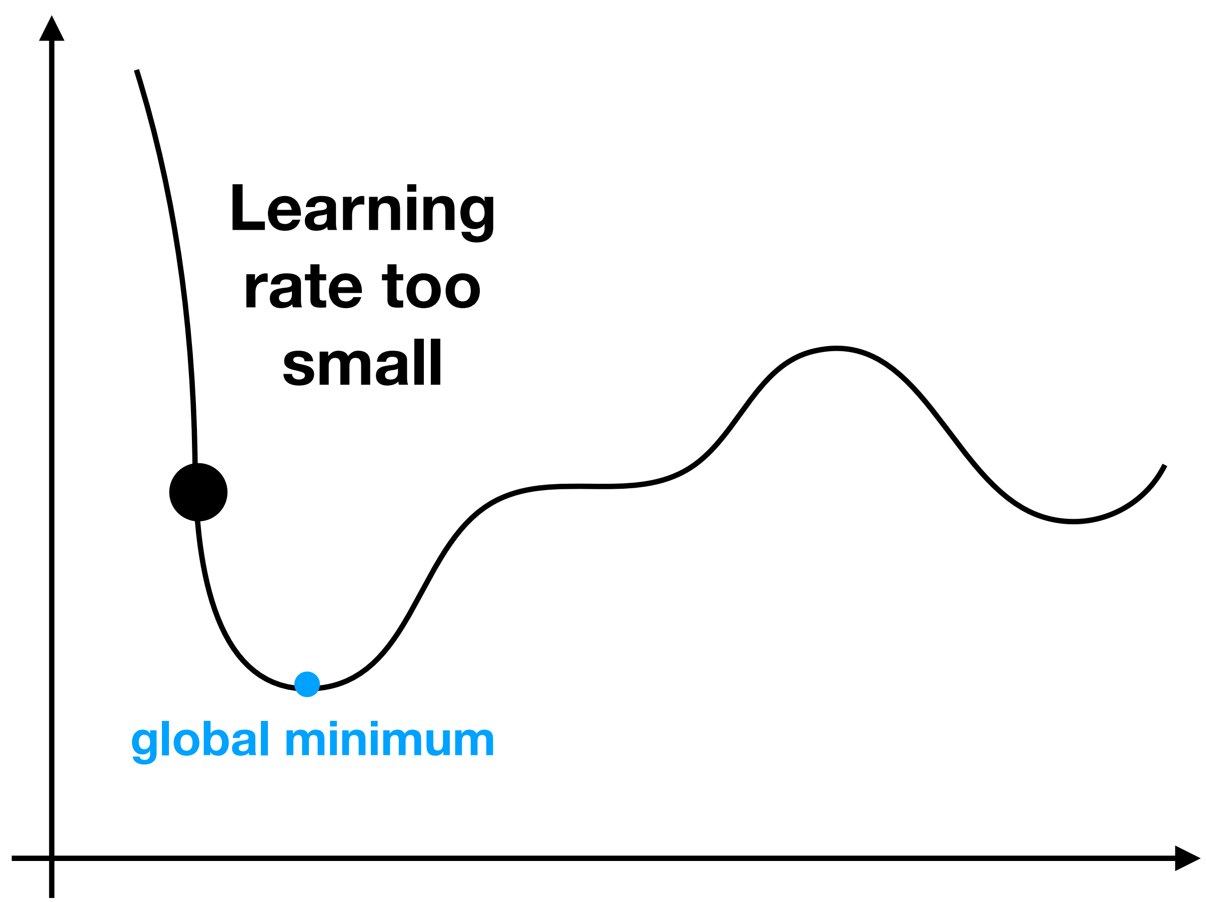
# Optimization: what can go wrong?

- In general ML and DL models, optimization is usually not so simple, due to:
    3. Learning rate is too small.



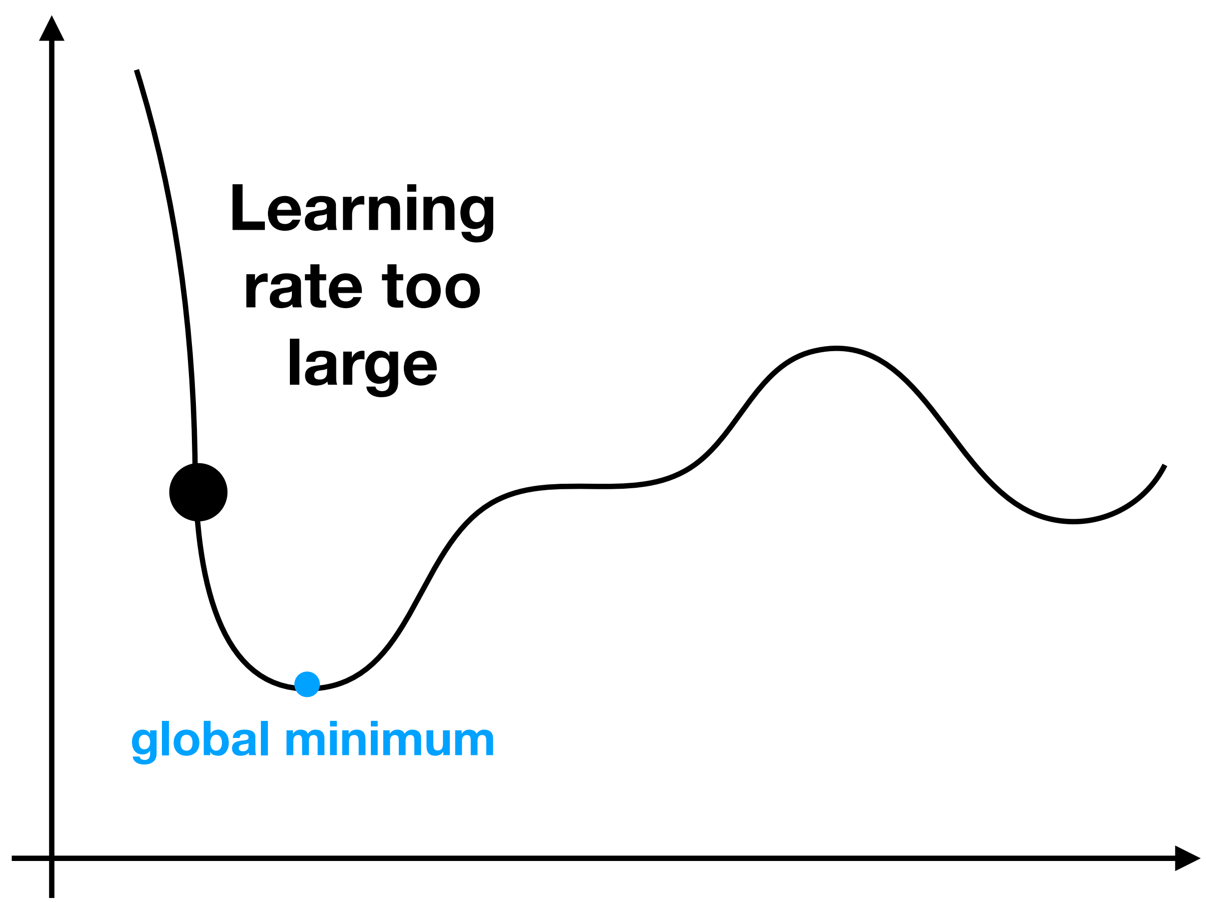
# Optimization: what can go wrong?

- In general ML and DL models, optimization is usually not so simple, due to:
    3. Learning rate is too small.



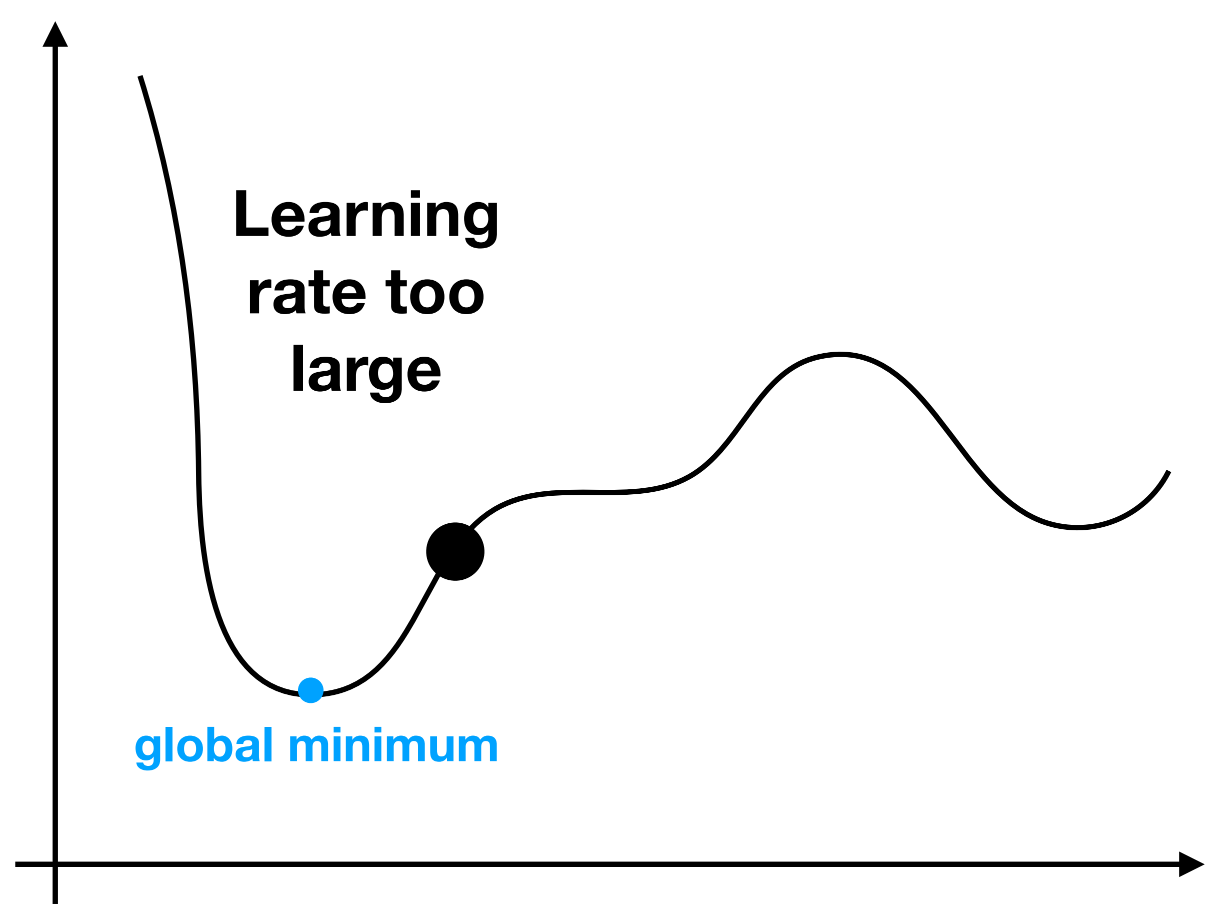
# Optimization: what can go wrong?

- In general ML and DL models, optimization is usually not so simple, due to:
  4. Learning rate is too large.



# Optimization: what can go wrong?

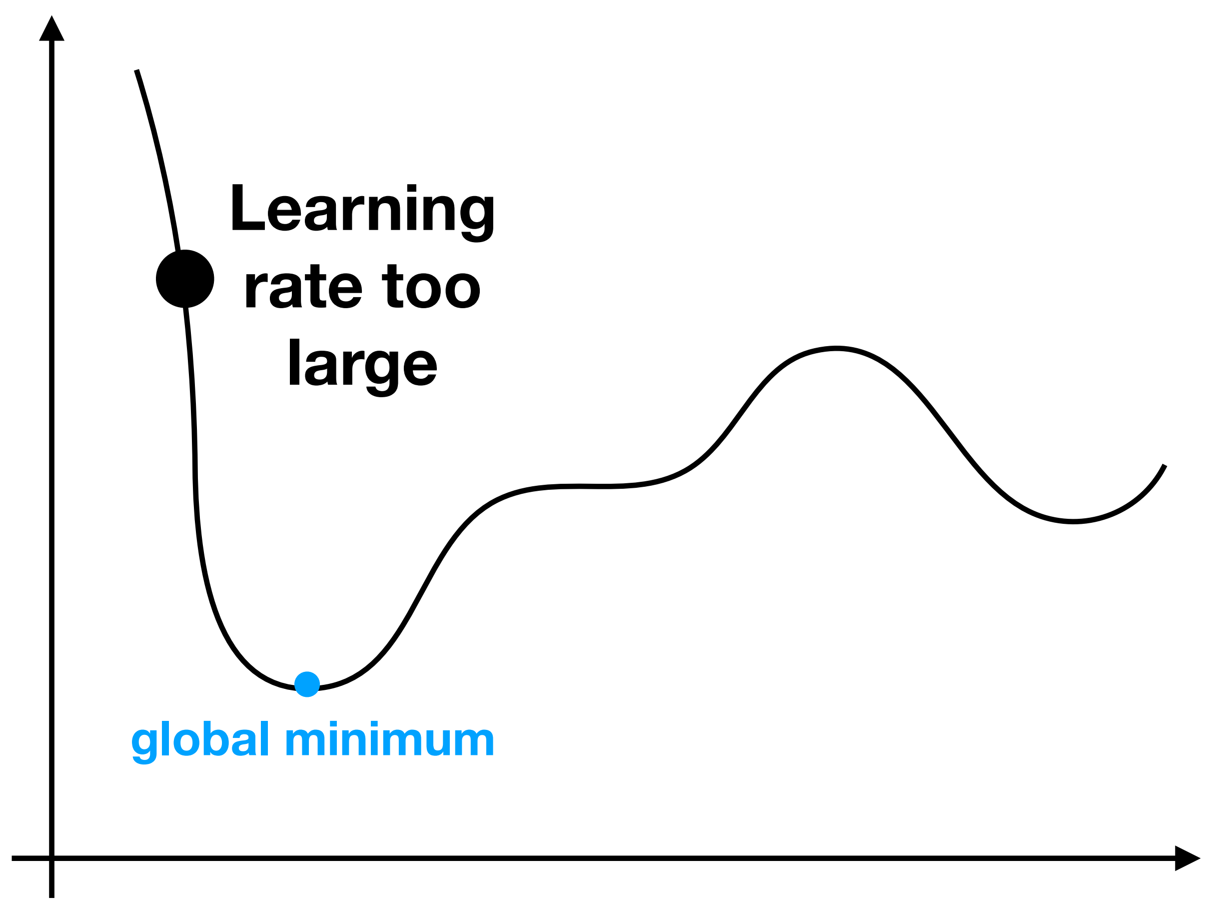
- In general ML and DL models, optimization is usually not so simple, due to:
    4. Learning rate is too large.



# Optimization: what can go wrong?

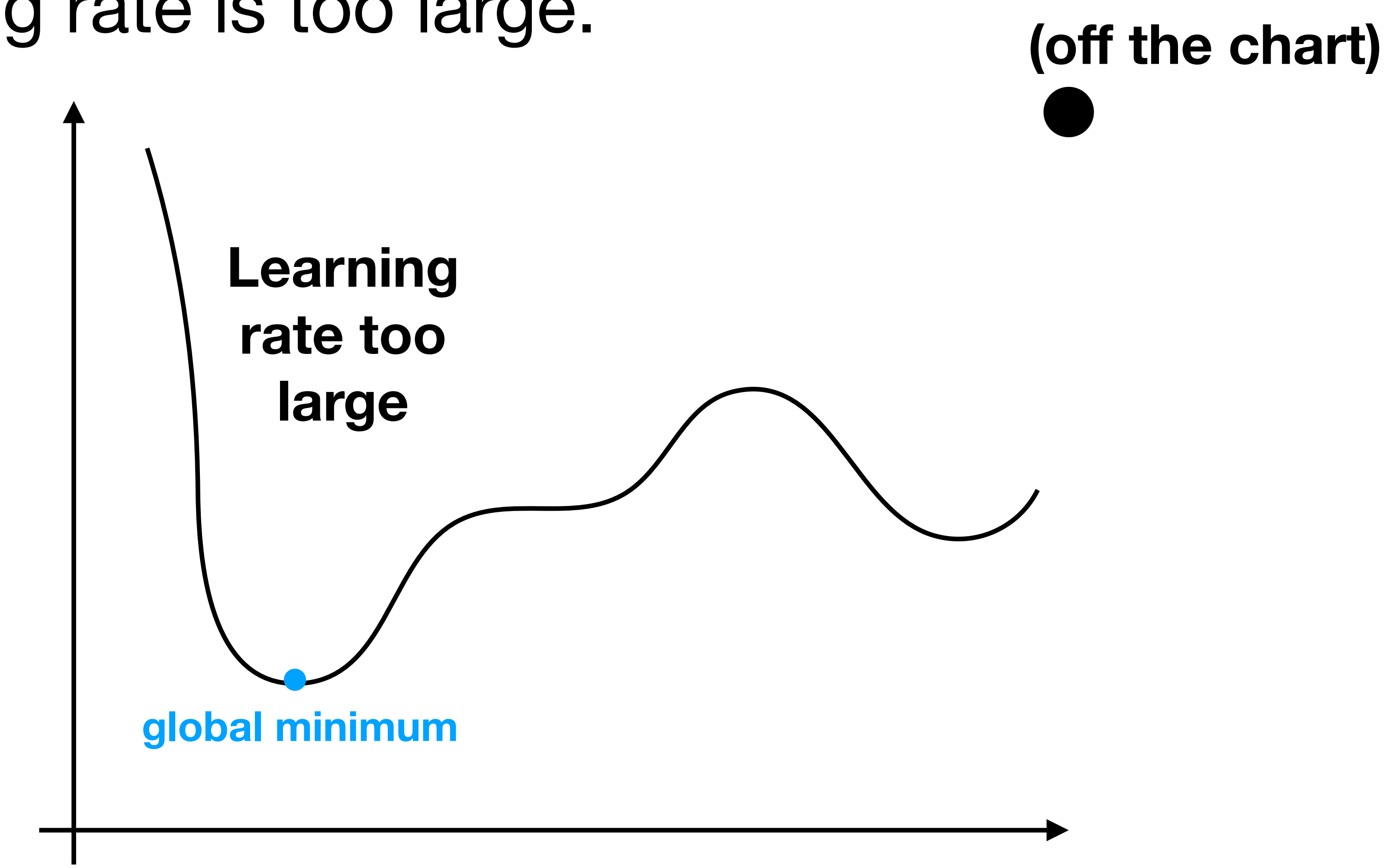
- In general ML and DL models, optimization is usually not so simple, due to:

    4. Learning rate is too large.



# Optimization: what can go wrong?

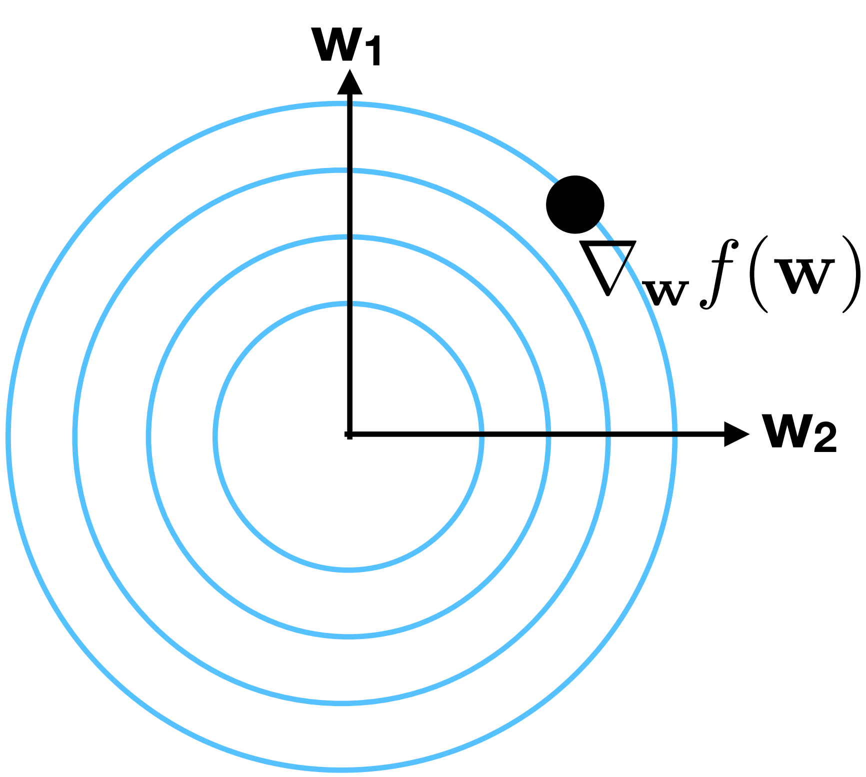
- In general ML and DL models, optimization is usually not so simple, due to:
    4. Learning rate is too large.



# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

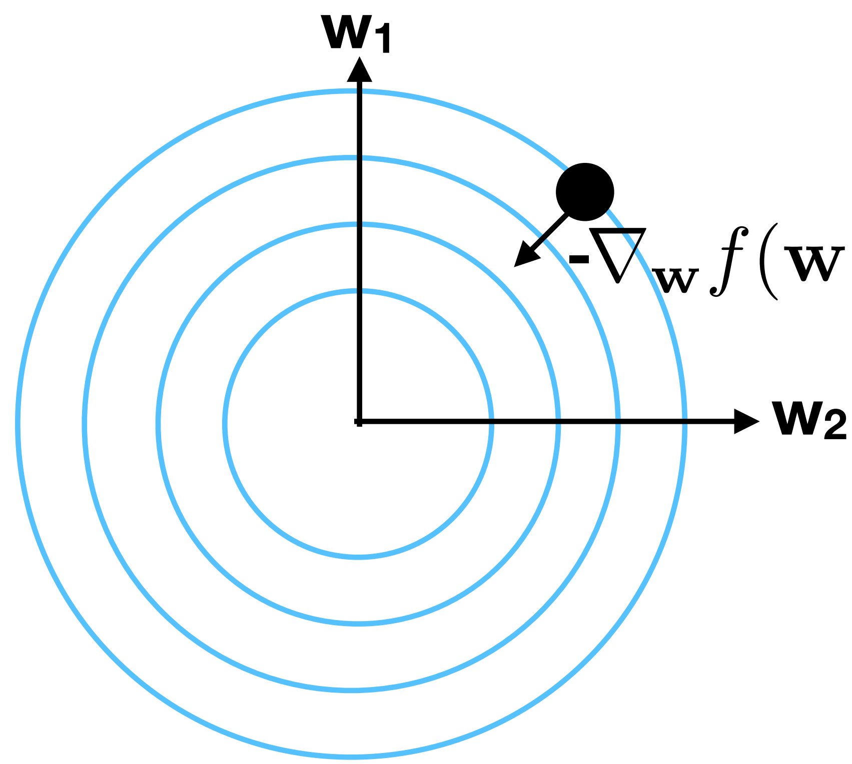
- Consider the cost *f* whose level sets are shown below:




**Which direction does the gradient point?**

# Optimization: what can go wrong?

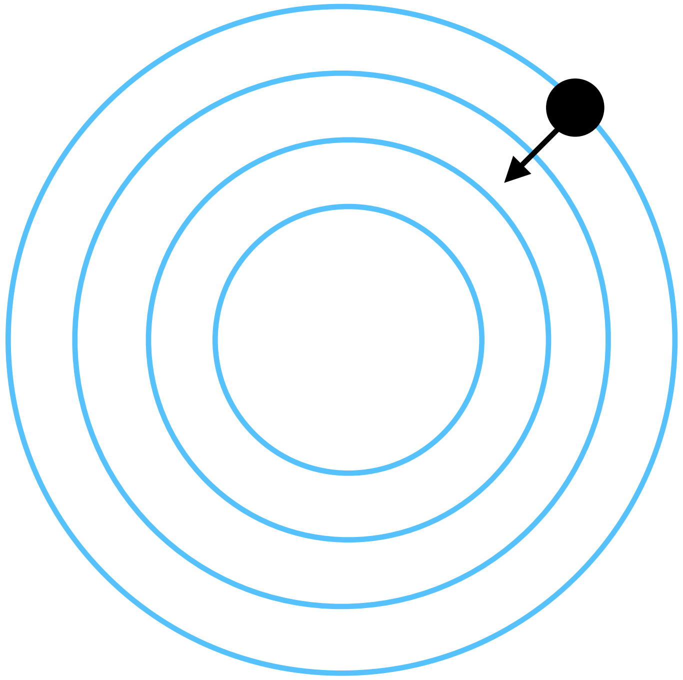
- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.
- Consider the cost *f* whose level sets are shown below:



# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

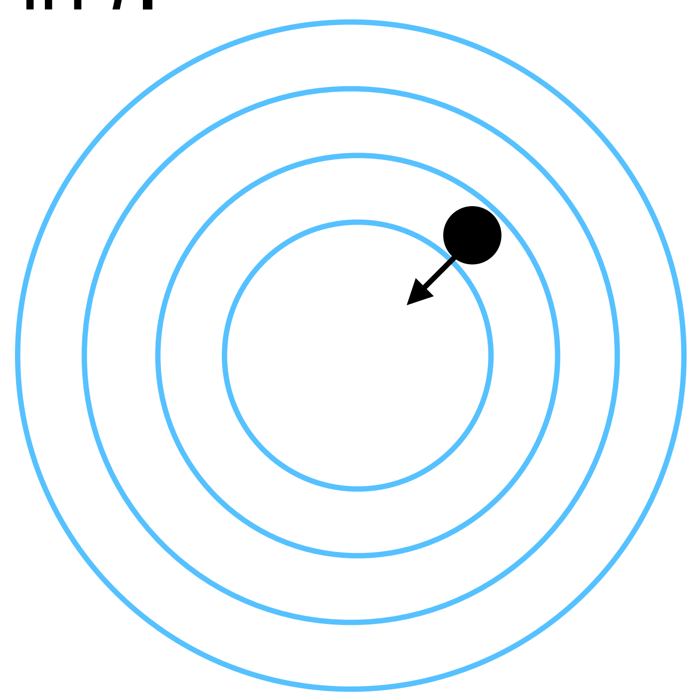
- Gradient descent guides the search along the direction of steepest decrease in $f$.



# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

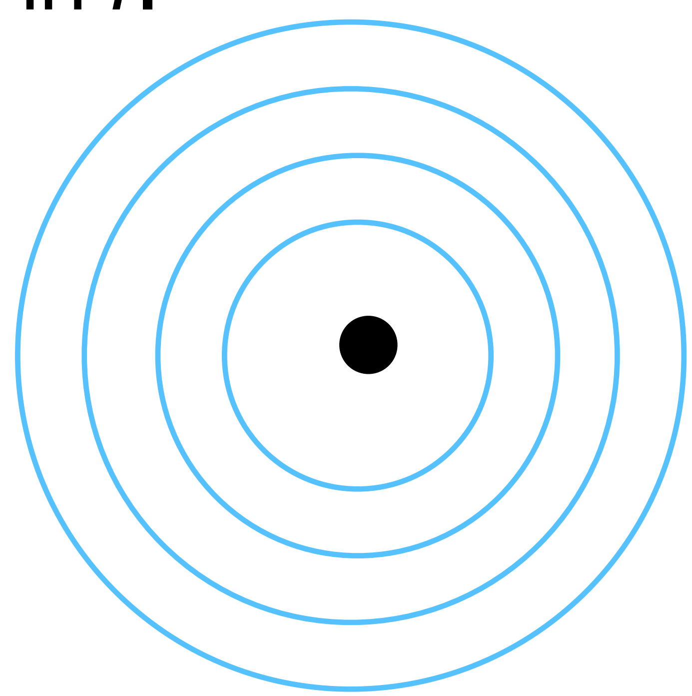
- Gradient descent guides the search along the direction of steepest decrease in *f*.



# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

- Gradient descent guides the search along the direction of steepest decrease in $f$.



# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

- But what if the level sets are ellipsoids instead of spheres?

# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

- But what if the level sets are ellipsoids instead of spheres?

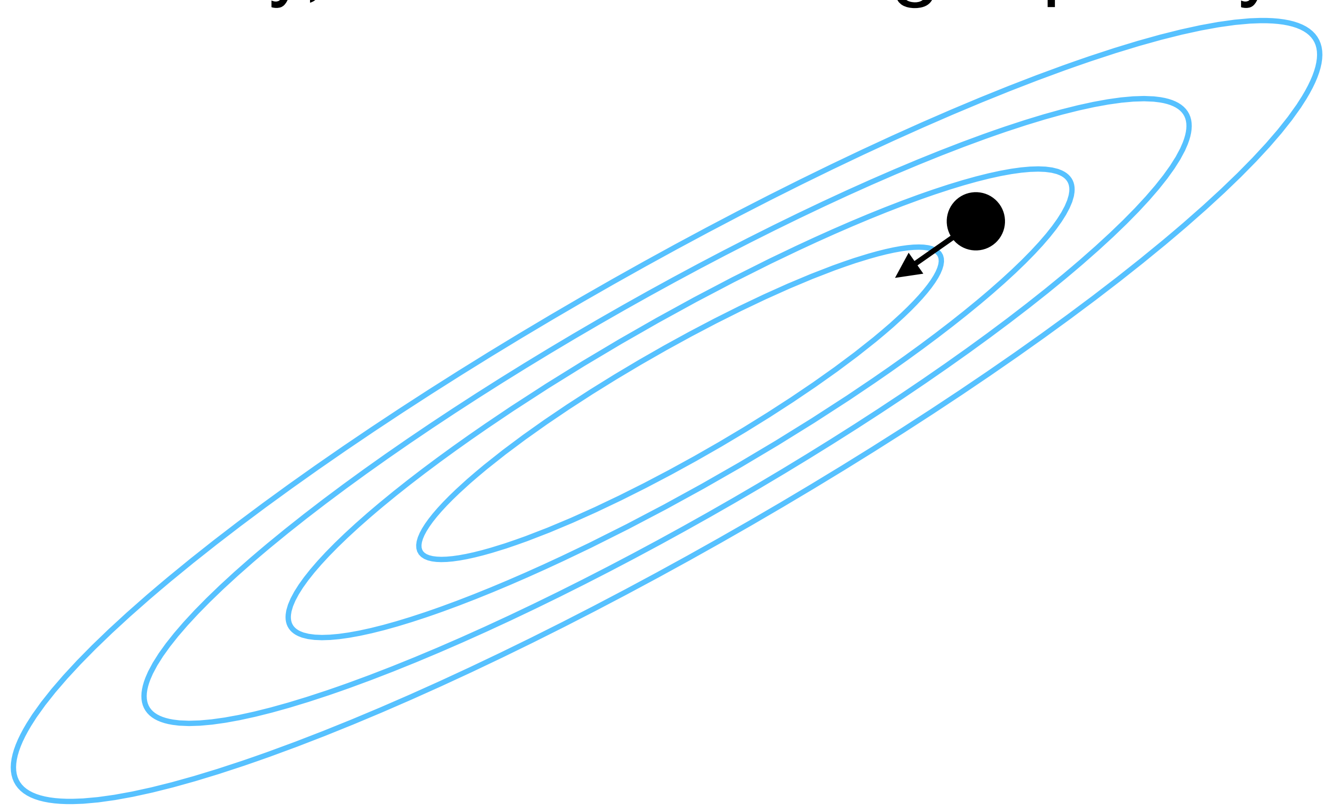
  - If we are lucky, we still converge quickly.

# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

- But what if the level sets are ellipsoids instead of spheres?

  - If we are lucky, we still converge quickly.

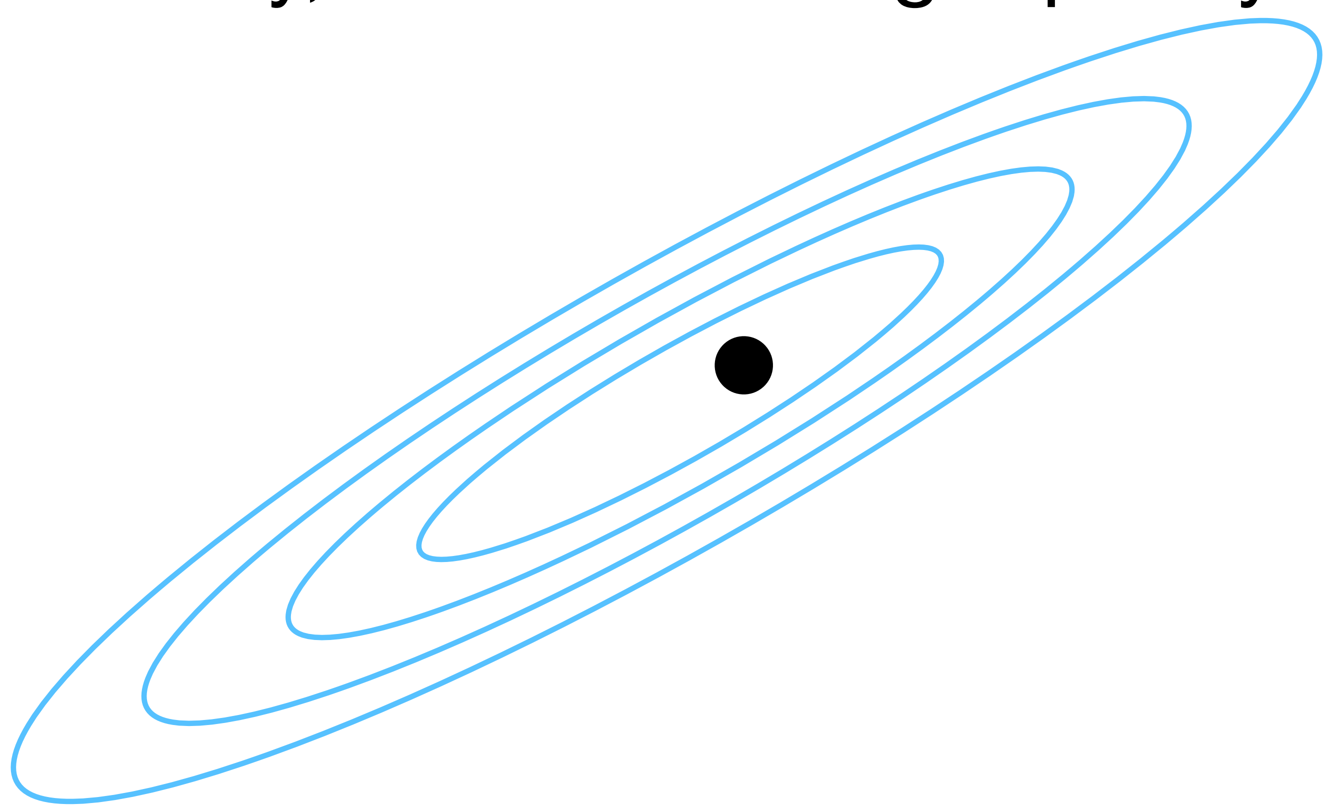


# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

- But what if the level sets are ellipsoids instead of spheres?

    - If we are lucky, we still converge quickly.



# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

- But what if the level sets are ellipsoids instead of spheres?

  - If we are unlucky, convergence is very slow.

# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

- But what if the level sets are ellipsoids instead of spheres?

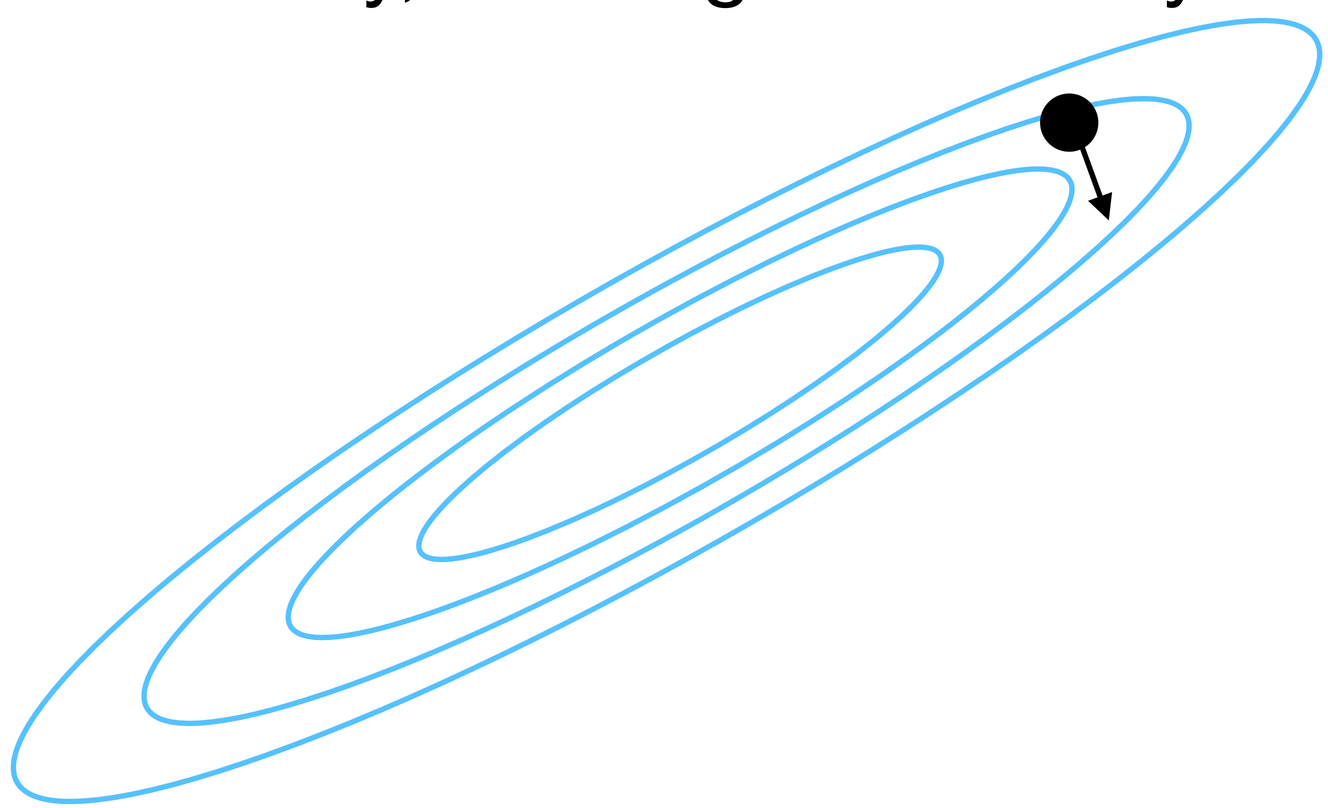
  - If we are unlucky, convergence is very slow.

# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

- But what if the level sets are ellipsoids instead of spheres?

    - If we are unlucky, convergence is very slow.

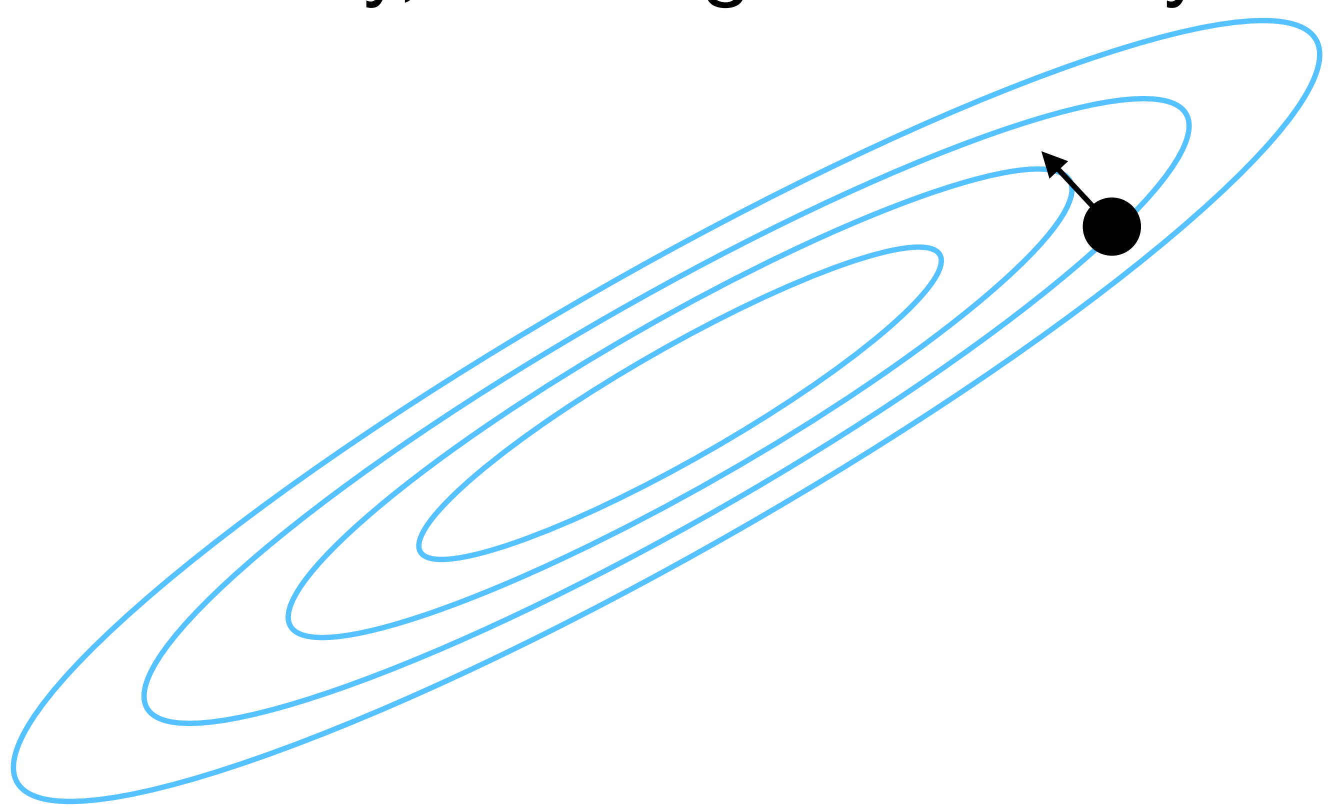


# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

- But what if the level sets are ellipsoids instead of spheres?

  - If we are unlucky, convergence is very slow.

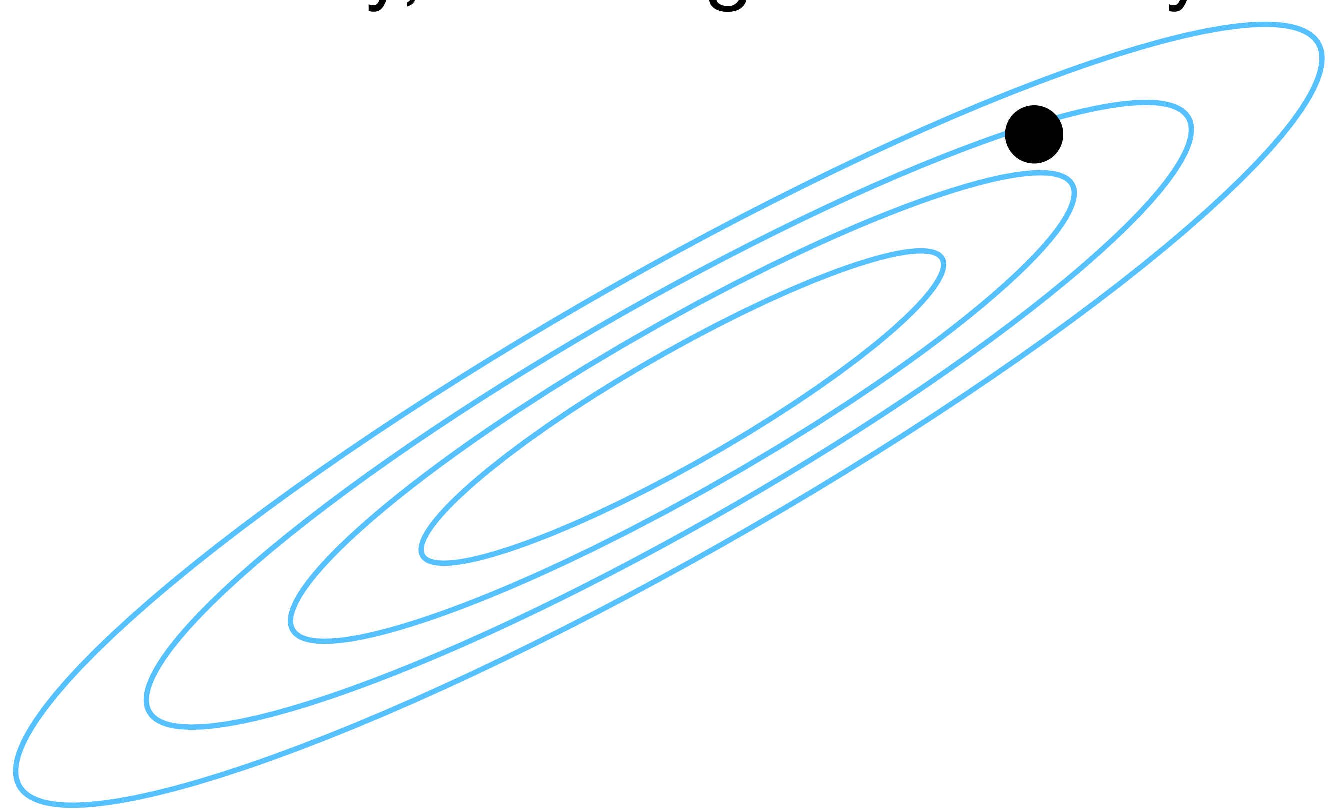


# Optimization: what can go wrong?

- With multidimensional weight vectors, badly chosen learning rates can cause more subtle problems.

- But what if the level sets are ellipsoids instead of spheres?

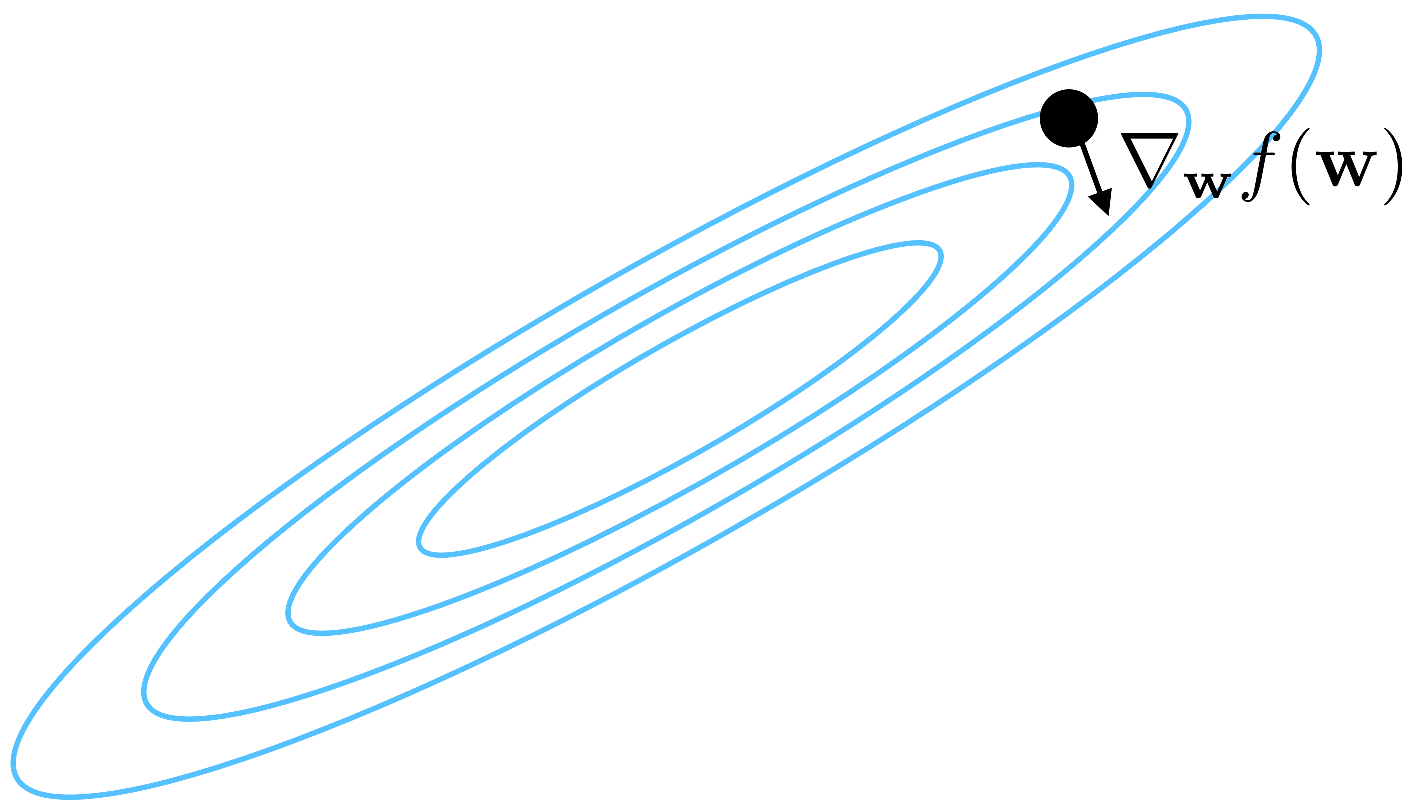
  - If we are unlucky, convergence is very slow.



**DEMO**

# Curvature

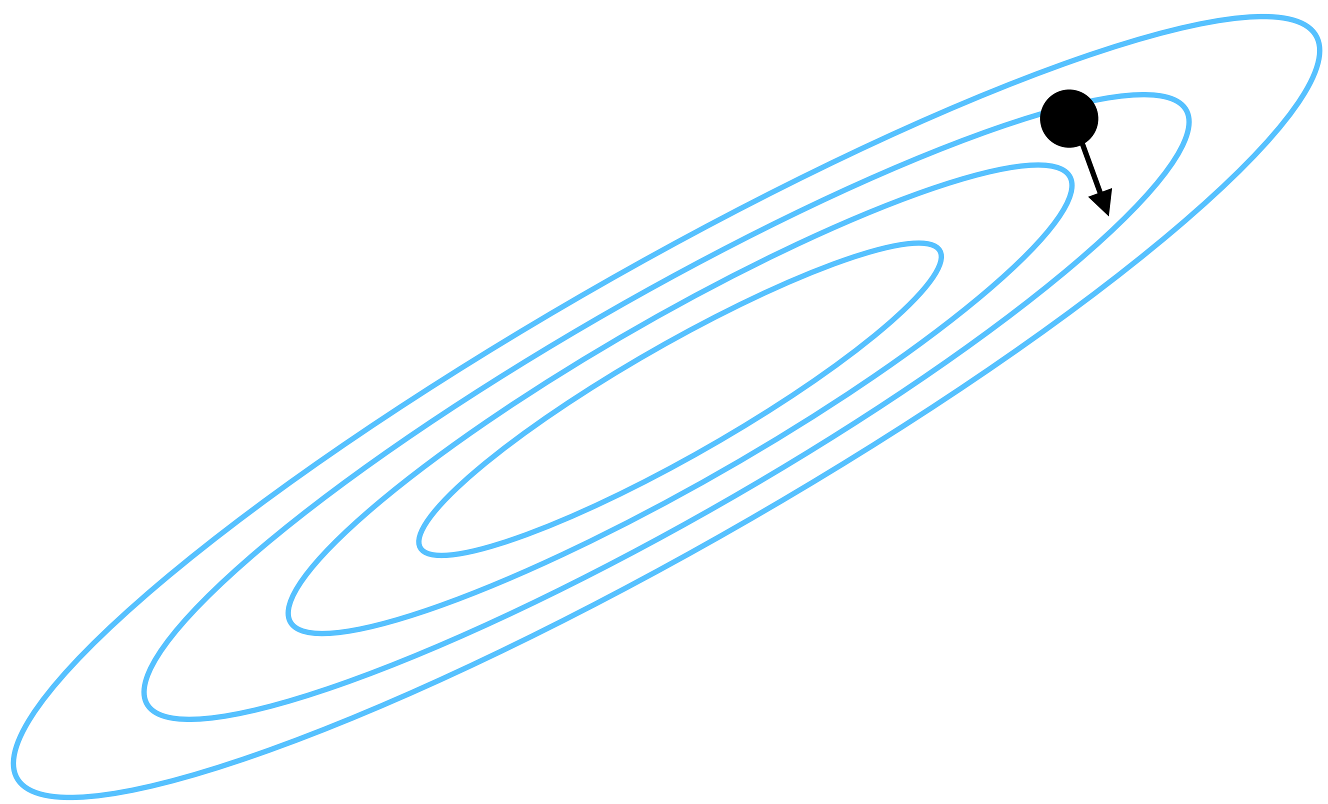
- The problem is that gradient descent only considers slope (1st-order effect), i.e., how *f* changes with **w**.



# Curvature

- The problem is that gradient descent only considers slope (1st-order effect), i.e., how *f* changes with **w**.

- The gradient does not consider how the slope *itself* changes with **w** (2nd-order effect).

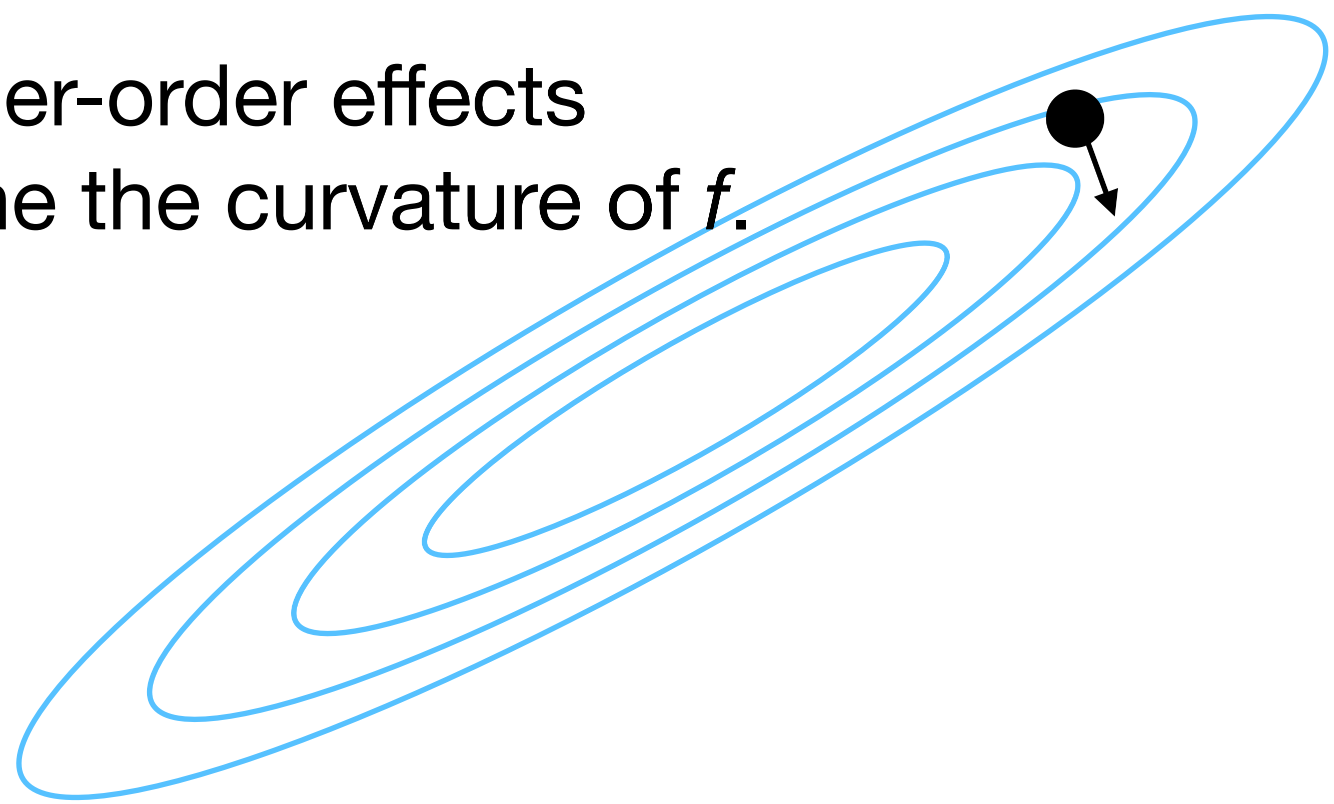


# Curvature

- The problem is that gradient descent only considers slope (1st-order effect), i.e., how *f* changes with **w**.

- The gradient does not consider how the slope *itself* changes with **w** (2nd-order effect).

- The higher-order effects determine the curvature of *f*.



# Optimization: what can we do?

- To accelerate optimization of the weights, we can either:
  - Alter the curvature of the loss by transforming the input data.
  - Change our optimization method to account for the curvature.

# Higher-order optimization algorithms

# Higher-order methods for optimization

- Higher (usually 2nd)-order optimization procedures can examine the curvature of the loss function to accelerate convergence.

- From the classical optimization literature, one of the most common methods is Newton-Raphson (aka Newton's method).

# Newton's method

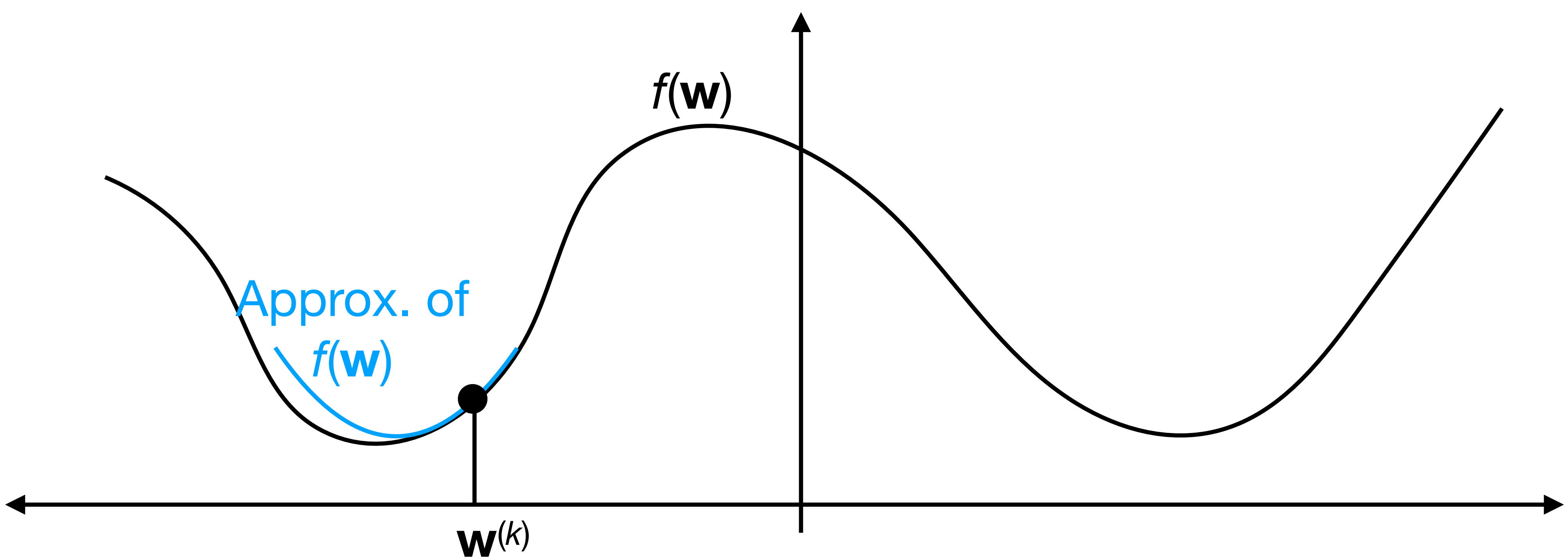
- When applicable, it offers faster convergence guarantees.
- Newton's method is an iterative method for finding the **roots** of a real-valued function $f$, i.e., $\mathbf{w}$ such that $f(\mathbf{w})=0$.
    - This is useful because we can use it to maximize/minimize a function by finding the roots of the gradient.

# Newton's method

- Let the 2nd-order Taylor expansion of *f* around $\mathbf{w}^{(k)}$ be:

$$f(\mathbf{w}) \approx f(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^{(k)})$$

  where **H** is the Hessian of *f* evaluated at $\mathbf{w}^{(k)}$.

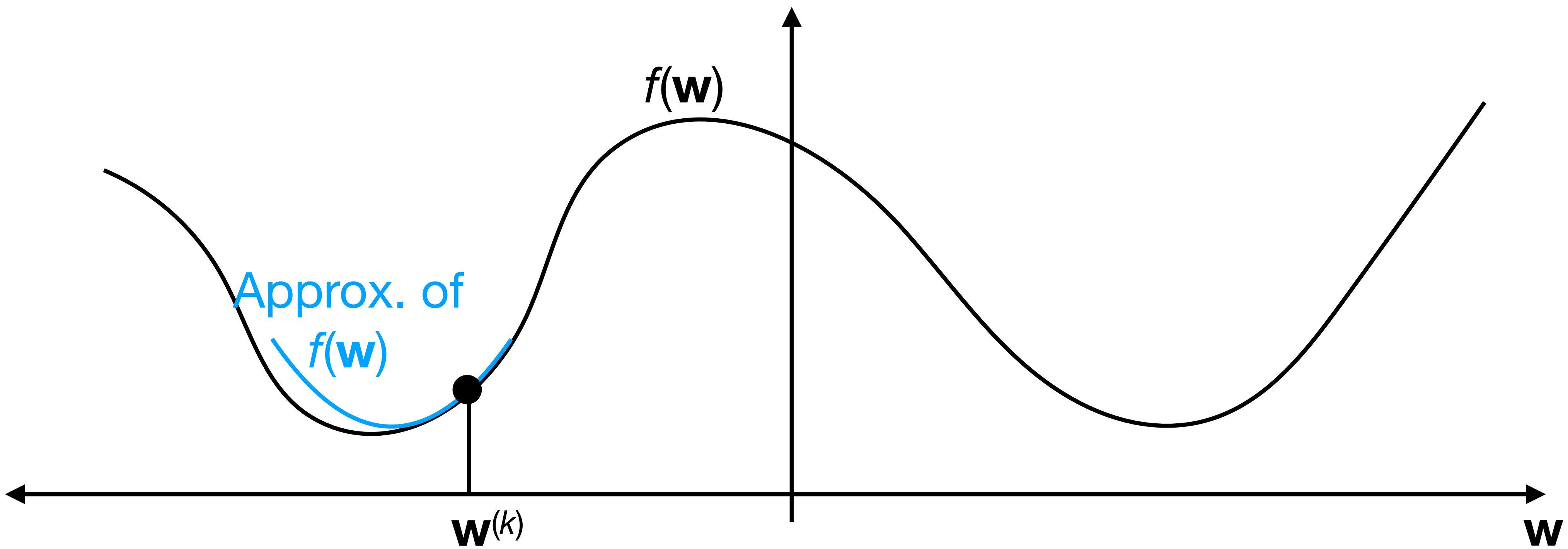


# Newton's method

- To minimize *f*, we find the root of the gradient of *f*'s Taylor expansion:

$$f(\mathbf{w}) \approx f(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^{(k)})$$

$$\nabla_{\mathbf{w}} f(\mathbf{w}) \approx \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \frac{1}{2}\nabla_{\mathbf{w}} \left(\mathbf{w}^\top \mathbf{H}\mathbf{w} - \mathbf{w}^\top \mathbf{H}\mathbf{w}^{(k)} - \mathbf{w}^{(k)\top} \mathbf{H}\mathbf{w} + \mathbf{w}^{(k)\top} \mathbf{H}\mathbf{w}^{(k)}\right)$$



# Newton's method

- To minimize *f*, we find the root of the gradient of *f*'s Taylor expansion:

$$f(\mathbf{w}) \approx f(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^{(k)})$$

$$\nabla_{\mathbf{w}} f(\mathbf{w}) \approx \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \frac{1}{2}\nabla_{\mathbf{w}} \left(\mathbf{w}^\top \mathbf{H}\mathbf{w} - \mathbf{w}^\top \mathbf{H}\mathbf{w}^{(k)} - {\mathbf{w}^{(k)}}^\top \mathbf{H}\mathbf{w} + {\mathbf{w}^{(k)}}^\top \mathbf{H}\mathbf{w}^{(k)}\right)$$

$$= \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)}$$

$$= \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \mathbf{H}\mathbf{w}^{(k)}$$

# Newton's method

- To minimize *f*, we find the root of the gradient of *f*'s Taylor expansion:

$$f(\mathbf{w}) \approx f(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^{(k)})$$

$$\nabla_{\mathbf{w}} f(\mathbf{w}) \approx \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \frac{1}{2}\nabla_{\mathbf{w}} \left(\mathbf{w}^\top \mathbf{H}\mathbf{w} - \mathbf{w}^\top \mathbf{H}\mathbf{w}^{(k)} - {\mathbf{w}^{(k)}}^\top \mathbf{H}\mathbf{w} + {\mathbf{w}^{(k)}}^\top \mathbf{H}\mathbf{w}^{(k)}\right)$$

$$= \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)}$$

$$= \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \mathbf{H}\mathbf{w}^{(k)}$$

$$0 = \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \mathbf{H}\mathbf{w}^{(k)}$$

# Newton's method

- To minimize *f*, we find the root of the gradient of *f*'s Taylor expansion:

$$f(\mathbf{w}) \approx f(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^{(k)})$$

$$\nabla_{\mathbf{w}} f(\mathbf{w}) \approx \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \frac{1}{2}\nabla_{\mathbf{w}} \left(\mathbf{w}^\top \mathbf{H}\mathbf{w} - \mathbf{w}^\top \mathbf{H}\mathbf{w}^{(k)} - \mathbf{w}^{(k)\top} \mathbf{H}\mathbf{w} + \mathbf{w}^{(k)\top} \mathbf{H}\mathbf{w}^{(k)}\right)$$

$$= \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)}$$

$$= \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \mathbf{H}\mathbf{w}^{(k)}$$

$$0 = \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \mathbf{H}\mathbf{w}^{(k)}$$

$$\mathbf{H}\mathbf{w} = \mathbf{H}\mathbf{w}^{(k)} - \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})$$

# Newton's method

- To minimize $f$, we find the root of the gradient of $f$'s Taylor expansion:

$$f(\mathbf{w}) \approx f(\mathbf{w}^{(k)}) + \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})(\mathbf{w} - \mathbf{w}^{(k)}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^{(k)})^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^{(k)})$$

$$\nabla_{\mathbf{w}} f(\mathbf{w}) \approx \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \frac{1}{2}\nabla_{\mathbf{w}} \left(\mathbf{w}^\top \mathbf{H}\mathbf{w} - \mathbf{w}^\top \mathbf{H}\mathbf{w}^{(k)} - {\mathbf{w}^{(k)}}^\top \mathbf{H}\mathbf{w} + {\mathbf{w}^{(k)}}^\top \mathbf{H}\mathbf{w}^{(k)}\right)$$

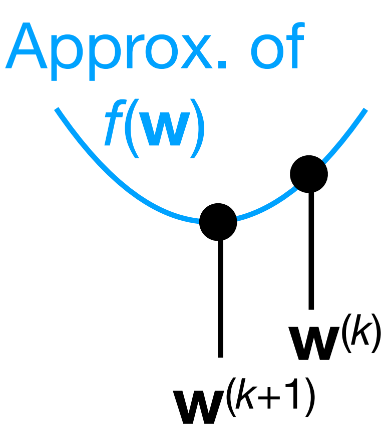
$$= \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)}$$

$$= \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \mathbf{H}\mathbf{w}^{(k)}$$

$$0 = \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \mathbf{H}\mathbf{w}^{(k)}$$

$$\mathbf{H}\mathbf{w} = \mathbf{H}\mathbf{w}^{(k)} - \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})$$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \mathbf{H}^{-1}\nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})$$



# Newton's method

- Note that, compared to gradient descent, the update rule in Newton's method replaces the step size $\varepsilon$ with the Hessian evaluated at $\mathbf{w}^{(k)}$:

  - Gradient descent:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \epsilon \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})$$

  - Newton's method:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \mathbf{H}^{-1} \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)})$$

# Newton’s method

- Newton’s method requires computation of **H**.
    - For high-dimensional feature spaces, **H** is huge, i.e., $O(m^3)$.
- Hence, Newton’s method in its pure form is impractical for DL.
- However, it has inspired modern DL optimization methods such as the **Adam** optimizer (Kingma & Ba 2014) (more to come later).

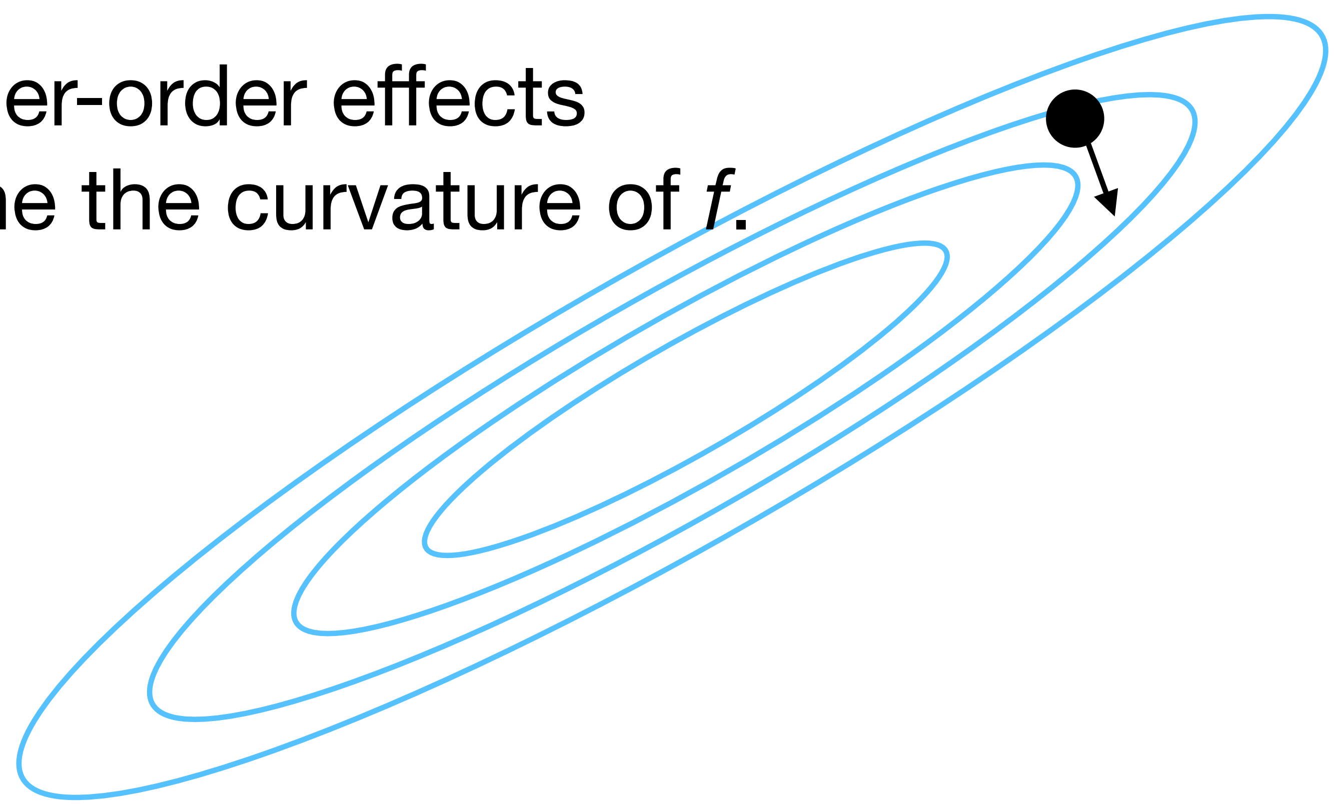
# Feature transformations

# Curvature

- The problem is that gradient descent only considers slope (1st-order effect), i.e., how *f* changes with **w**.

- The gradient does not consider how the slope *itself* changes with **w** (2nd-order effect).

- The higher-order effects determine the curvature of *f*.



# Curvature

- For linear regression with cost $f_{\text{MSE}}$,

$$f_{\text{MSE}}(\mathbf{w}) = \frac{1}{2n}(\mathbf{X}^{\top}\mathbf{w} - \mathbf{y})^{\top}(\mathbf{X}^{\top}\mathbf{w} - \mathbf{y})$$

the Hessian is:

$$\mathbf{H}[f](\mathbf{w}) = \frac{1}{n}\mathbf{X}\mathbf{X}^{\top}$$

# Curvature

- For linear regression with cost $f_{\mathrm{MSE}}$,

$$f_{\mathrm{MSE}}(\mathbf{w}) = \frac{1}{2n}(\mathbf{X}^\top\mathbf{w} - \mathbf{y})^\top(\mathbf{X}^\top\mathbf{w} - \mathbf{y})$$

  the Hessian is:

$$\mathbf{H}[f](\mathbf{w}) = \frac{1}{n}\mathbf{X}\mathbf{X}^\top$$

- Hence, **H** is constant and is proportional to the (uncentered) **auto-covariance matrix** of **X**, which is the multidimensional analog of the **variance** of a dataset.

$$\mathbb{E}[(\mathbf{X} - \mathbb{E}[\mathbf{X}])(\mathbf{X} - \mathbb{E}[\mathbf{X}])^\top]$$

# Whitening transformations

- We can sometimes accelerate training of an ML model by transforming the features so that the auto-covariance $\mathbf{X}\mathbf{X}^\mathrm{T}$ induces a loss function more amenable to SGD.

- Whitening: we “spherize” the input features using a **whitening transformation T**, which makes the auto-covariance matrix equal the identity matrix **I**.

- We compute this transformation **T** on the training data **X**, and then apply it to both training and testing data.

# Whitening transformations

- We can find a whitening transform **T** as follows:
  - Let the auto-covariance* of our training data be $\mathbf{X}\mathbf{X}^T$.

*** (uncentered).**

# Whitening transformations

- We can find a whitening transform **T** as follows:
  - Let the auto-covariance* of our training data be $\mathbf{X}\mathbf{X}^{\top}$.
  - We can write its eigendecomposition as:

$$\mathbf{X}\mathbf{X}^{\top}\Phi = \Phi\Lambda$$

  where $\Phi$ is the matrix of eigenvectors and $\Lambda$ is the corresponding diagonal matrix of eigenvalues.

*** (uncentered).**

# Whitening transformations

- We can find a whitening transform **T** as follows:
  - Let the auto-covariance* of our training data be $\mathbf{X}\mathbf{X}^\top$.
  - We can write its eigendecomposition as:

$$\mathbf{X}\mathbf{X}^\top \Phi = \Phi \Lambda$$

    where $\Phi$ is the matrix of eigenvectors and $\Lambda$ is the corresponding diagonal matrix of eigenvalues.

  - For real-valued features, $\mathbf{X}\mathbf{X}^\top$ is real and symmetric; hence, $\Phi$ is orthonormal. Also, $\Lambda$ is non-negative.

*** (uncentered).**

# Whitening transformations

- We can find a whitening transform **T** as follows:
  - Therefore, we can multiply both sides by $\Phi^T$:

$$\mathbf{X}\mathbf{X}^\top \Phi = \Phi \Lambda$$

$$\Phi^\top \mathbf{X}\mathbf{X}^\top \Phi = \Phi^\top \Phi \Lambda = \Lambda$$

# Whitening transformations

- We can find a whitening transform **T** as follows:
  - Therefore, we can multiply both sides by $\Phi^T$:

$$\mathbf{X}\mathbf{X}^\top\Phi = \Phi\Lambda$$

$$\Phi^\top\mathbf{X}\mathbf{X}^\top\Phi = \Phi^\top\Phi\Lambda = \Lambda$$

  - Since $\Lambda$ is diagonal and non-negative, we can easily compute $\mathbf{\Lambda}^{-\frac{1}{2}}$.

# Whitening transformations

- We can find a whitening transform **T** as follows:
  - Therefore, we can multiply both sides by $\Phi^\top$:

$$\mathbf{X}\mathbf{X}^\top\Phi = \Phi\Lambda$$

$$\Phi^\top\mathbf{X}\mathbf{X}^\top\Phi = \Phi^\top\Phi\Lambda = \Lambda$$

  - Since $\Lambda$ is diagonal and non-negative, we can easily compute $\mathbf{\Lambda}^{-\frac{1}{2}}$.
  - We then multiply both sides (2x) to obtain **I** on the RHS.

$${\Lambda^{-\frac{1}{2}}}^\top\Phi^\top\mathbf{X}\mathbf{X}^\top\Phi\Lambda^{-\frac{1}{2}} = {\Lambda^{-\frac{1}{2}}}^\top\Lambda\Lambda^{-\frac{1}{2}}$$

# Whitening transformations

- We can find a whitening transform **T** as follows:
  - Therefore, we can multiply both sides by $\Phi^T$:

$$\mathbf{X}\mathbf{X}^\top \Phi = \Phi\Lambda$$

$$\Phi^\top \mathbf{X}\mathbf{X}^\top \Phi = \Phi^\top \Phi\Lambda = \Lambda$$

  - Since Λ is diagonal and non-negative, we can easily compute $\boldsymbol{\Lambda}^{-\frac{1}{2}}$.
  - We then multiply both sides (2x) to obtain **I** on the RHS.

$$\Lambda^{-\frac{1}{2}}{}^\top \Phi^\top \mathbf{X}\mathbf{X}^\top \Phi\Lambda^{-\frac{1}{2}} = \Lambda^{-\frac{1}{2}}{}^\top \Lambda\Lambda^{-\frac{1}{2}}$$

$$\left(\Lambda^{-\frac{1}{2}}{}^\top \Phi^\top \mathbf{X}\right)\left(\Lambda^{-\frac{1}{2}}{}^\top \Phi^\top \mathbf{X}\right)^\top = \mathbf{I}$$

# Whitening transformations

- We can find a whitening transform **T** as follows:
  - Therefore, we can multiply both sides by $\Phi^T$:

$$\mathbf{X}\mathbf{X}^\top \Phi = \Phi \Lambda$$

$$\Phi^\top \mathbf{X}\mathbf{X}^\top \Phi = \Phi^\top \Phi \Lambda = \Lambda$$

  - Since Λ is diagonal and non-negative, we can easily compute $\mathbf{\Lambda}^{-\frac{1}{2}}$.
  - We then multiply both sides (2x) to obtain **I** on the RHS.

$$\Lambda^{-\frac{1}{2}}{}^\top \Phi^\top \mathbf{X}\mathbf{X}^\top \Phi \Lambda^{-\frac{1}{2}} = \Lambda^{-\frac{1}{2}}{}^\top \Lambda \Lambda^{-\frac{1}{2}}$$

$$\left(\Lambda^{-\frac{1}{2}}{}^\top \Phi^\top \mathbf{X}\right)\left(\Lambda^{-\frac{1}{2}}{}^\top \Phi^\top \mathbf{X}\right)^\top = \mathbf{I}$$

$$(\mathbf{T}\mathbf{X})(\mathbf{T}\mathbf{X})^\top = \mathbf{I}$$

# Whitening transformations

- We have thus derived a transform $\mathbf{T} = {\Lambda^{-\frac{1}{2}}}^\top \Phi^\top$ such that the (uncentered) auto-covariance of the transformed data $\tilde{\mathbf{X}} = \mathbf{TX}$ is the identity matrix **I**.

# Whitening transformations

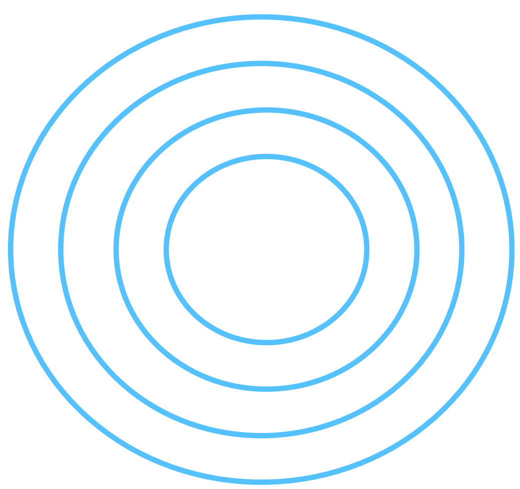
- We have thus derived a transform $\mathbf{T} = \Lambda^{-\frac{1}{2}}{}^{\top}\Phi^{\top}$ such that the (uncentered) auto-covariance of the transformed data $\tilde{\mathbf{X}} = \mathbf{T}\mathbf{X}$ is the identity matrix **I**.

- **T** transforms the cost from $f_{\mathrm{MSE}}(\mathbf{w}; \mathbf{X})$

# Whitening transformations

- We have thus derived a transform $\mathbf{T} = {\Lambda^{-\frac{1}{2}}}^\top \Phi^\top$ such that the (uncentered) auto-covariance of the transformed data $\tilde{\mathbf{X}} = \mathbf{T}\mathbf{X}$ is the identity matrix **I**.

- **T** transforms the cost from $f_{\mathrm{MSE}}(\mathbf{w}; \mathbf{X})$ to $f_{\mathrm{MSE}}(\mathbf{w}; \tilde{\mathbf{X}})$:



# Whitening transformations

- Whitening transformations are a technique from "classical" ML rather than DL.
  - Time cost is $O(m^3)$, which for high-dimensional feature spaces is too large.
- However, whitening has inspired modern DL techniques such as **batch normalization** (Szegedy & Ioffe, 2015) (more to come later) and **concept whitening** (Chen et al. 2020).

# Exercise

- Consider a 2-layer linear neural network (NN) whose input is a matrix **X** (where each column is a training example), whose output is $\hat{y}$, and whose weight vector **w** is trained so as to minimize the $L_2$-regularized mean squared error (MSE) loss. Examine the two Python functions below:

- ```
  def f (X, y, w, alpha):
      yhat = X.dot(w)
      return np.mean((yhat - y) ** 2) + alpha * w.dot(w)
  ```

- ```
  def grad (X, y, w, alpha):
      yhat = X.dot(w)
      return np.mean(yhat - y) + alpha * w
  ```

- Which of the following statements are true?

  A. The grad function correctly computes the gradient of the *f* function.

  B. With an appropriate choice of the learning rate and regularization strength alpha, the NN can be trained using gradient descent with the grad function above to reach a local minimum of the regularized MSE.

  C. For some alpha < 0, the *f* function implemented above may no longer be convex with respect to **w**.

# Exercise

- Consider a 2-layer linear neural network (NN) whose input is a matrix **X** (where each column is a training example), whose output is $\hat{y}$, and whose weight vector **w** is trained so as to minimize the $L_2$-regularized mean squared error (MSE) loss. Examine the two Python functions below:

- ```
  def f (X, y, w, alpha):
      yhat = X.T.dot(w)
      return np.mean((yhat - y) ** 2) + alpha * w.dot(w)
  ```

- ```
  def grad (X, y, w, alpha):
      yhat = X.T.dot(w)
      return X.dot(yhat - y) + alpha * w
  ```

- Which of the following statements are true?

  A. The grad function correctly computes the gradient of the *f* function.

  B. With an appropriate choice of the learning rate and regularization strength alpha, the NN can be trained using gradient descent with the grad function above to reach a local minimum of the regularized MSE.

  C. For some alpha < 0, the *f* function implemented above may no longer be convex with respect to **w**.