CS/DS 541: Class 4

Jacob Whitehill

Optimization of ML models

Optimization of ML models

- Gradient descent is guaranteed to converge to a local minimum (eventually) if the learning rate is small enough relative to the steepness of *f*.
- A function $f : \mathbb{R}^m \to \mathbb{R}$ is Lipschitz-continuous if:

$$\exists L : \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^m : \|f(\mathbf{x}) - f(\mathbf{y})\|_2 \leq L\|\mathbf{x} - \mathbf{y}\|_2$$

- *L* is essentially an upper bound on the absolute slope of *f*.

Optimization of ML models

- Gradient descent is guaranteed to converge to a local minimum (eventually) if the learning rate is small enough relative to the steepness of *f*.

- A function $f : \mathbb{R}^m \to \mathbb{R}$ is Lipschitz-continuous if:

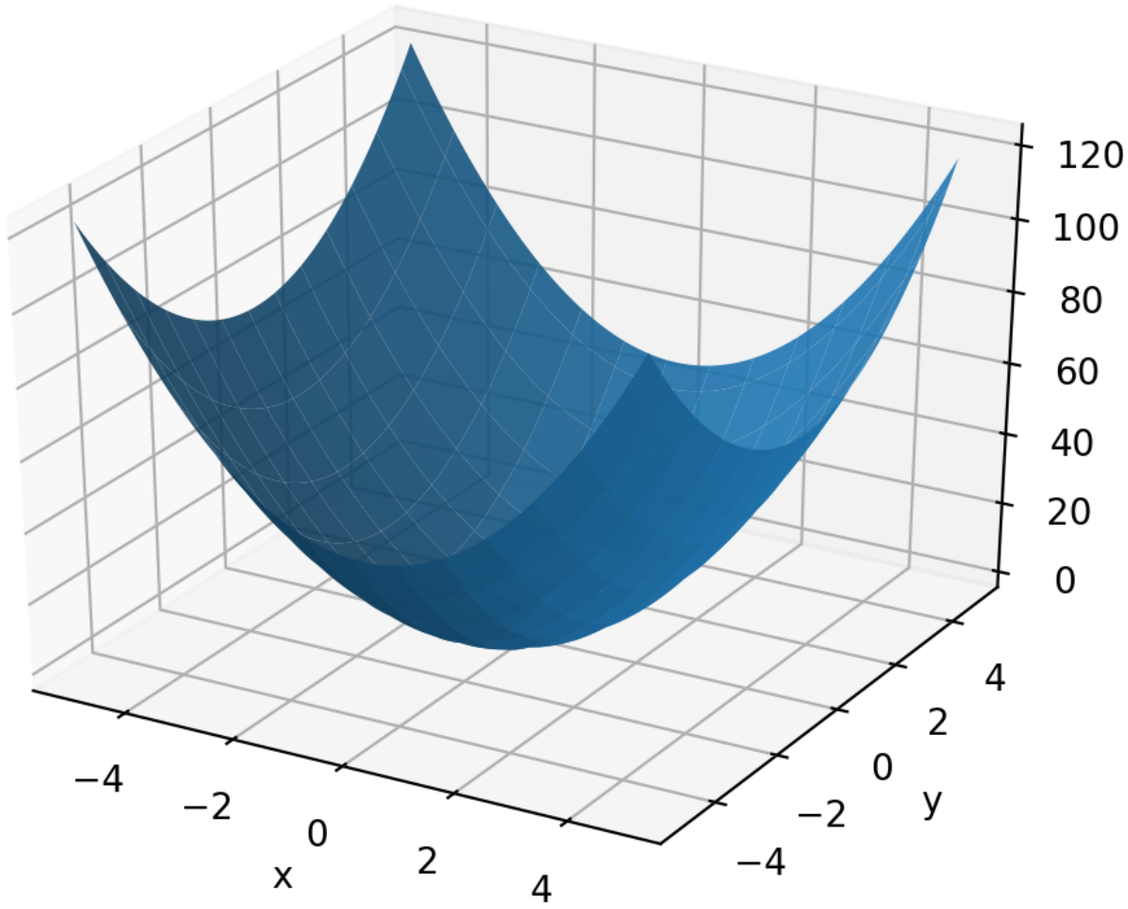
$$\exists L : \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^m : \|f(\mathbf{x}) - f(\mathbf{y})\|_2 \leq L\|\mathbf{x} - \mathbf{y}\|_2$$

- *L* is essentially an upper bound on the absolute slope of *f*.

- For learning rate $\epsilon \leq \frac{1}{L}$, gradient descent will converge to a local minimum linearly, i.e., the error is *O*(1/*k*) in the iterations *k*.

Optimization of ML models

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



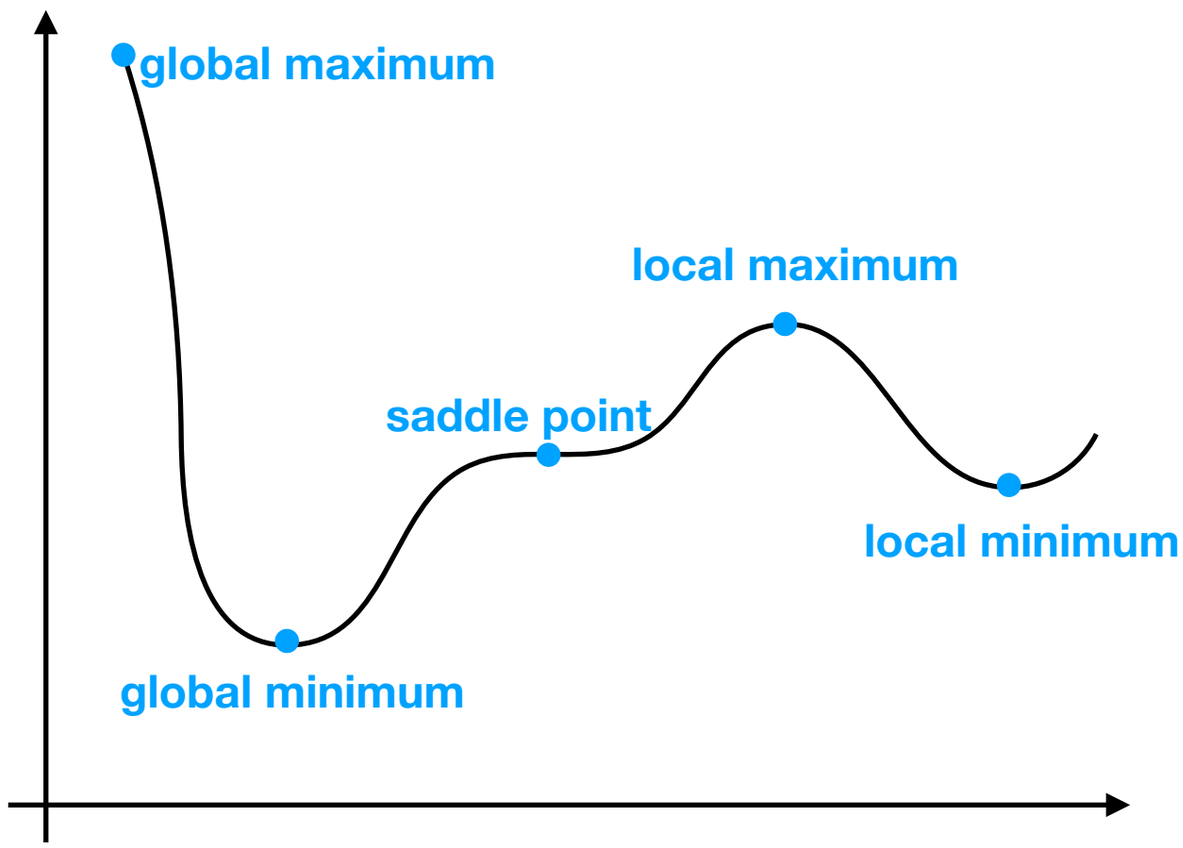
- As long as our learning rate is small enough, we will find **the optimal w**.

Optimization: what can go wrong?

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

Optimization: what can go wrong?

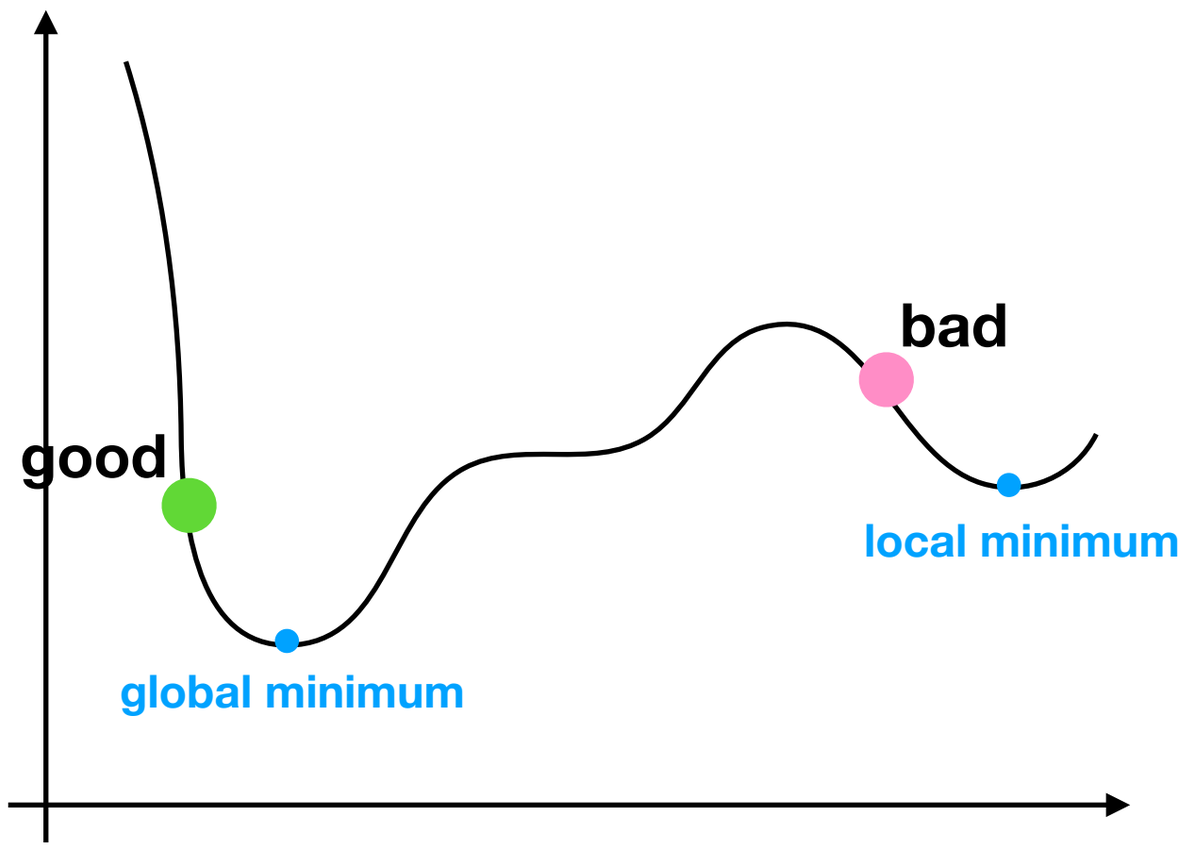
- In general ML and DL models, optimization is usually not so simple, due to:
    1. Presence of multiple local minima & saddle points



Optimization: what can go wrong?

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

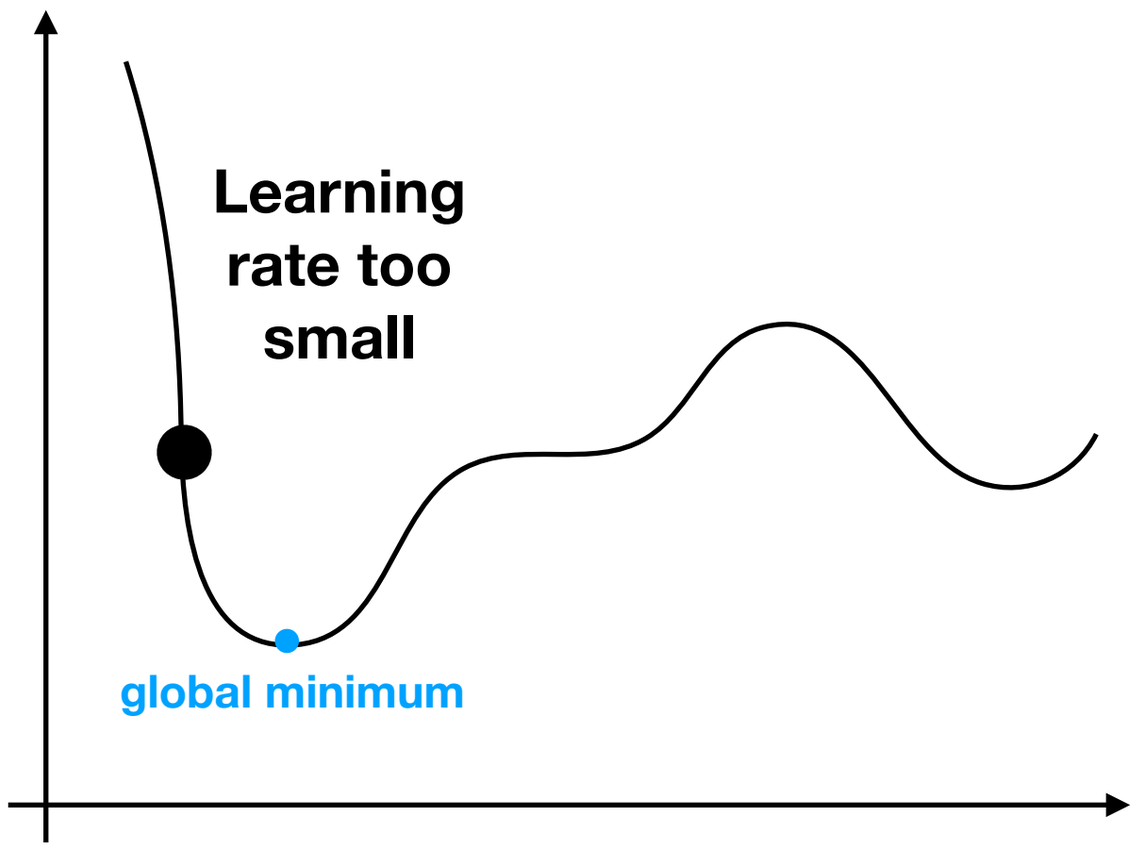
  2. Bad initialization of the weights **w**.



Optimization: what can go wrong?

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

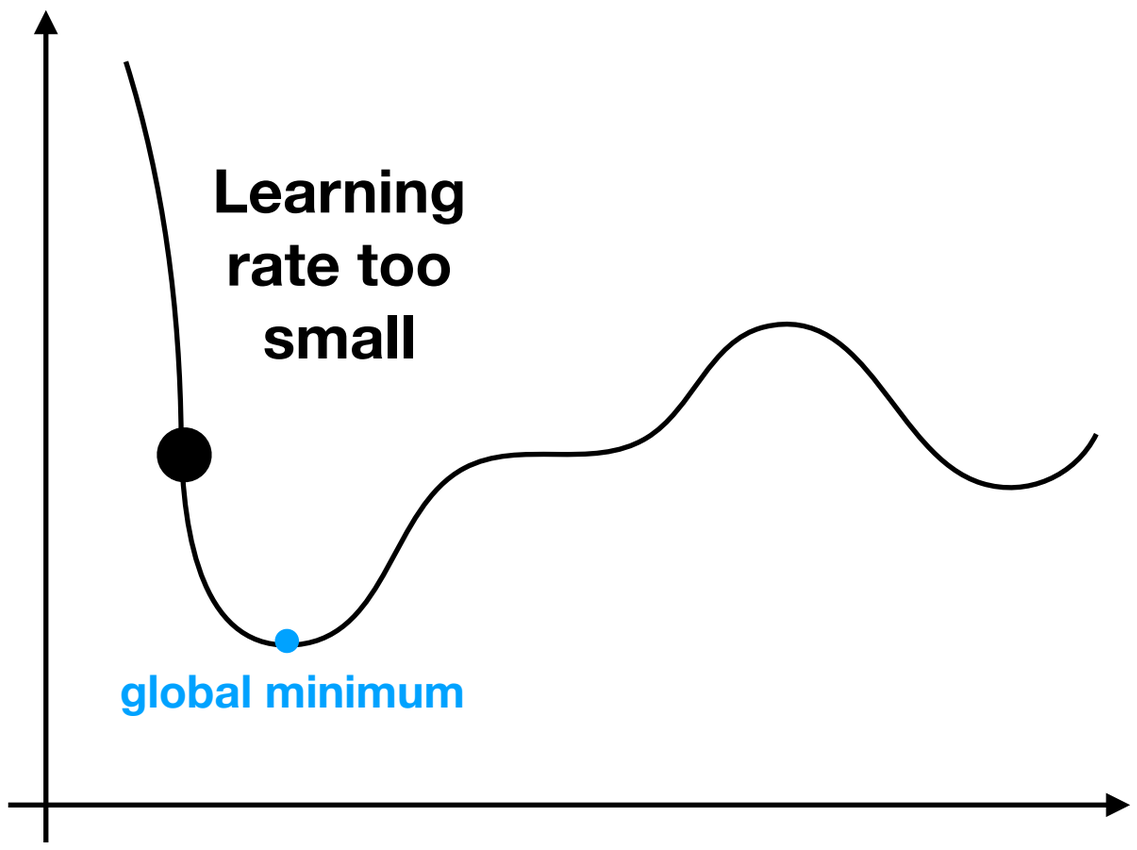
3. Learning rate is too small.



Optimization: what can go wrong?

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

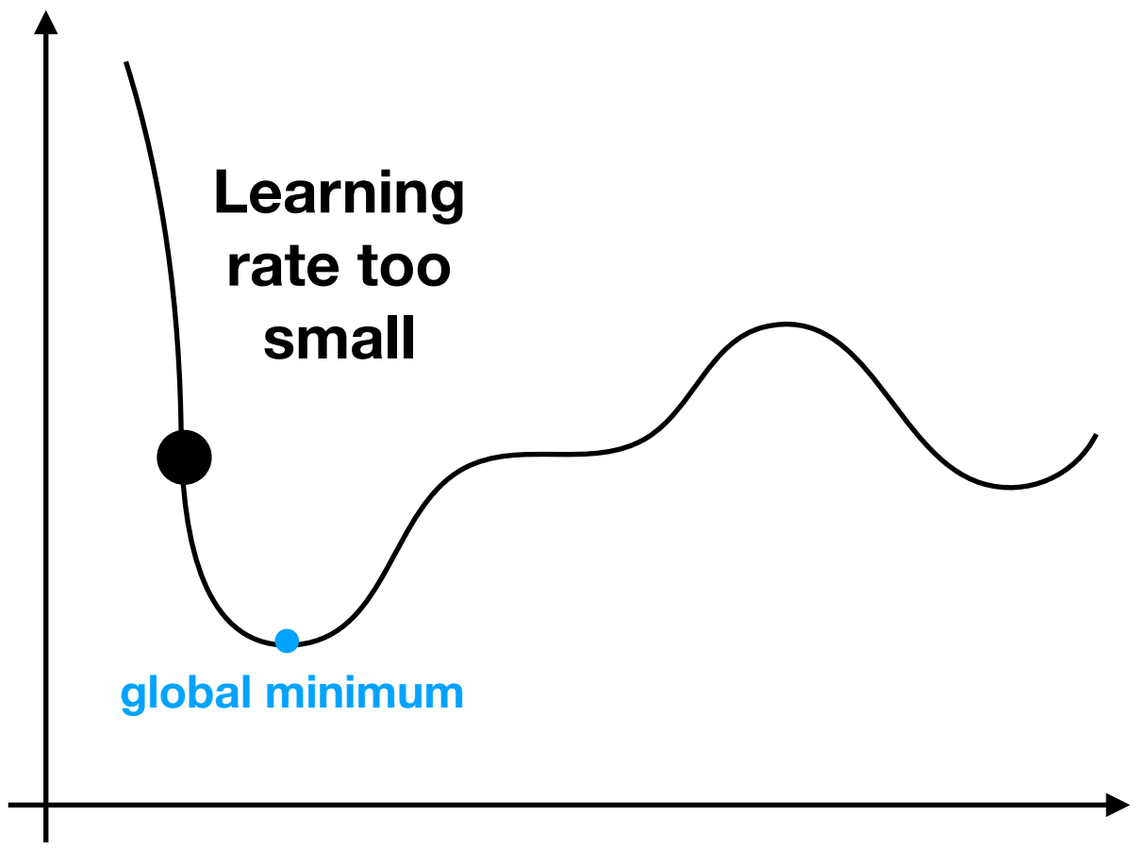
3. Learning rate is too small.



Optimization: what can go wrong?

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

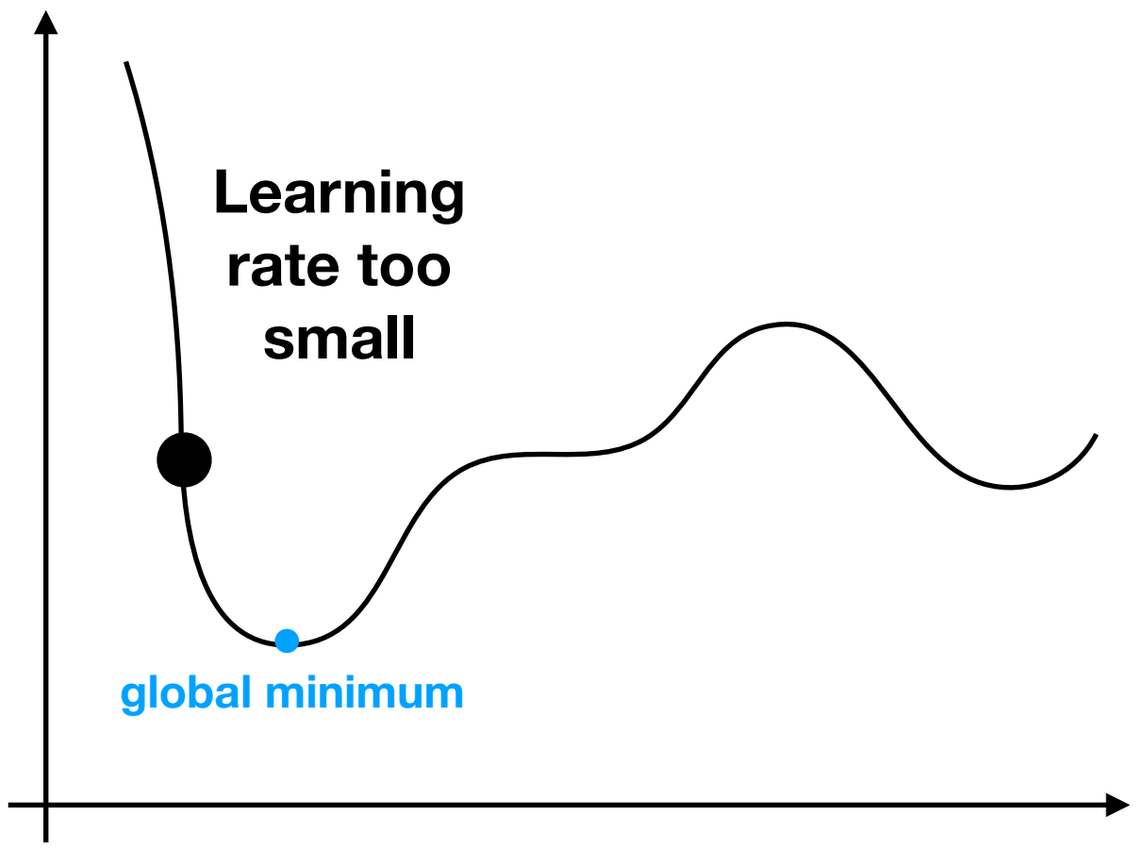
3. Learning rate is too small.



Optimization: what can go wrong?

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

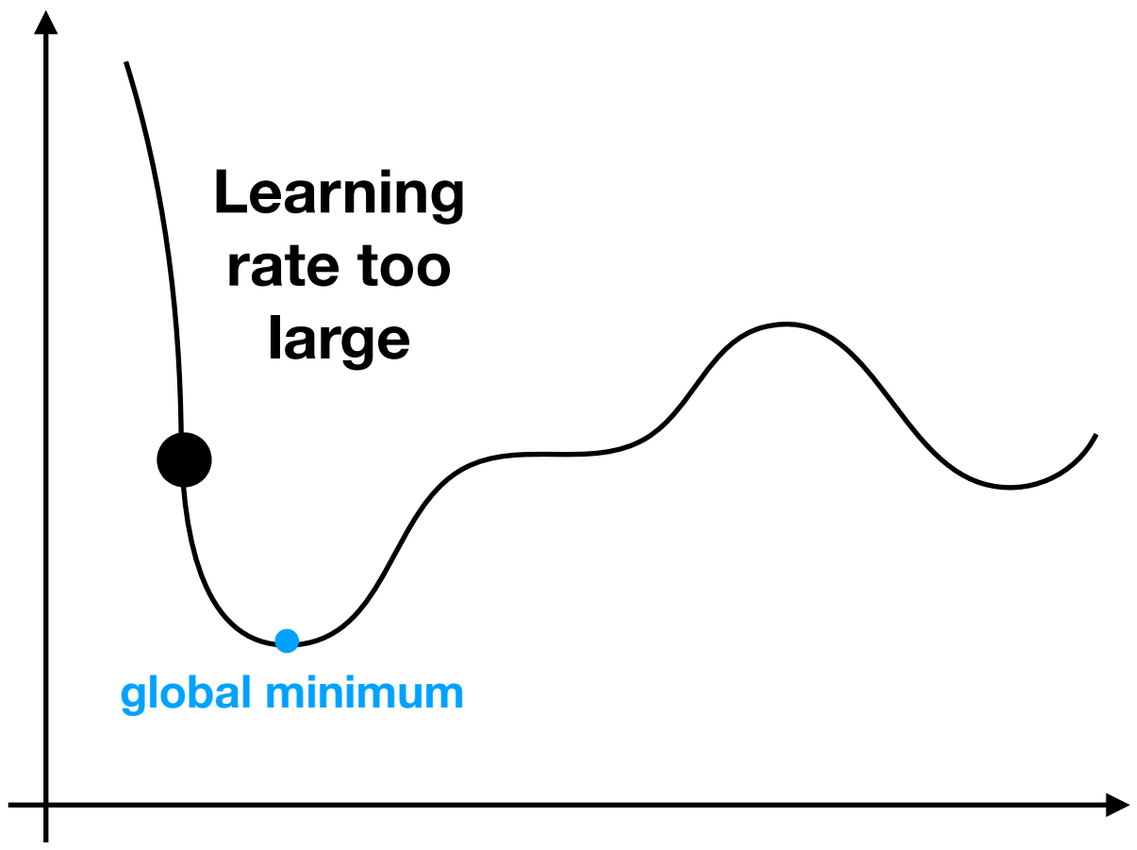
3. Learning rate is too small.



Optimization: what can go wrong?

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

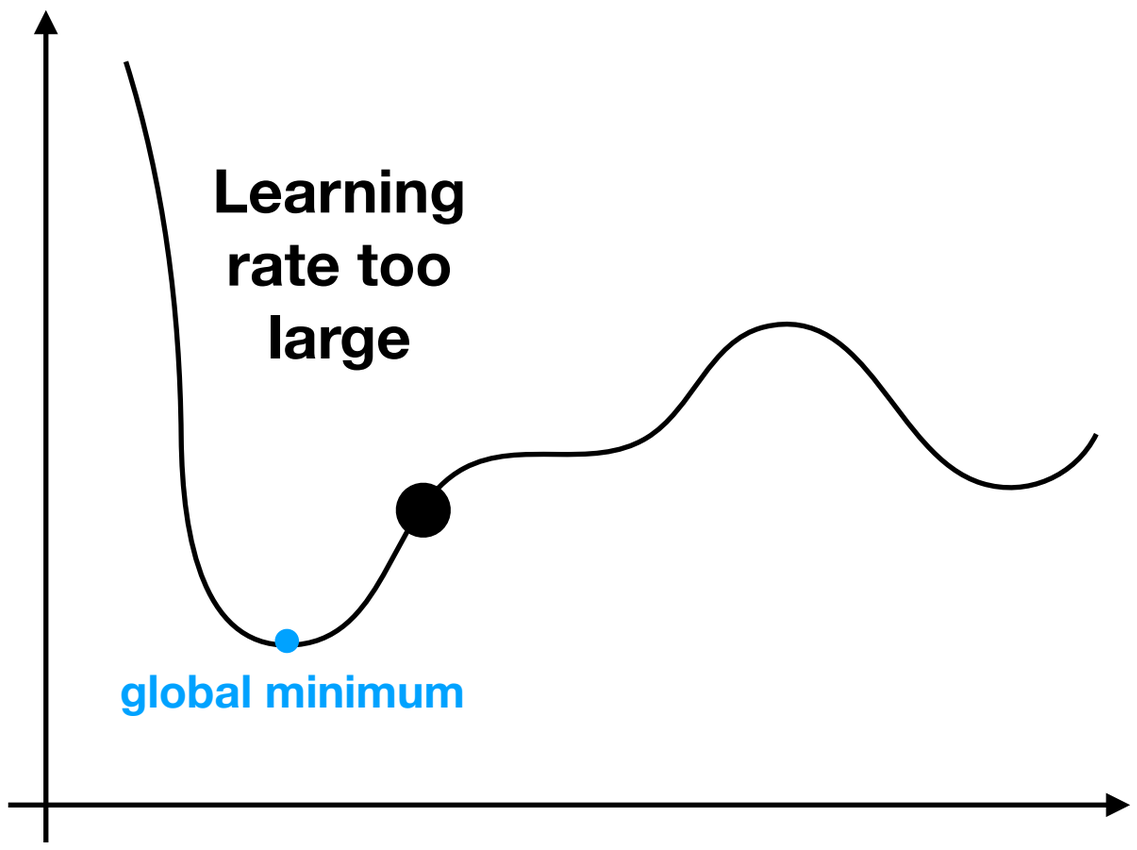
3. Learning rate is too small.



Optimization: what can go wrong?

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

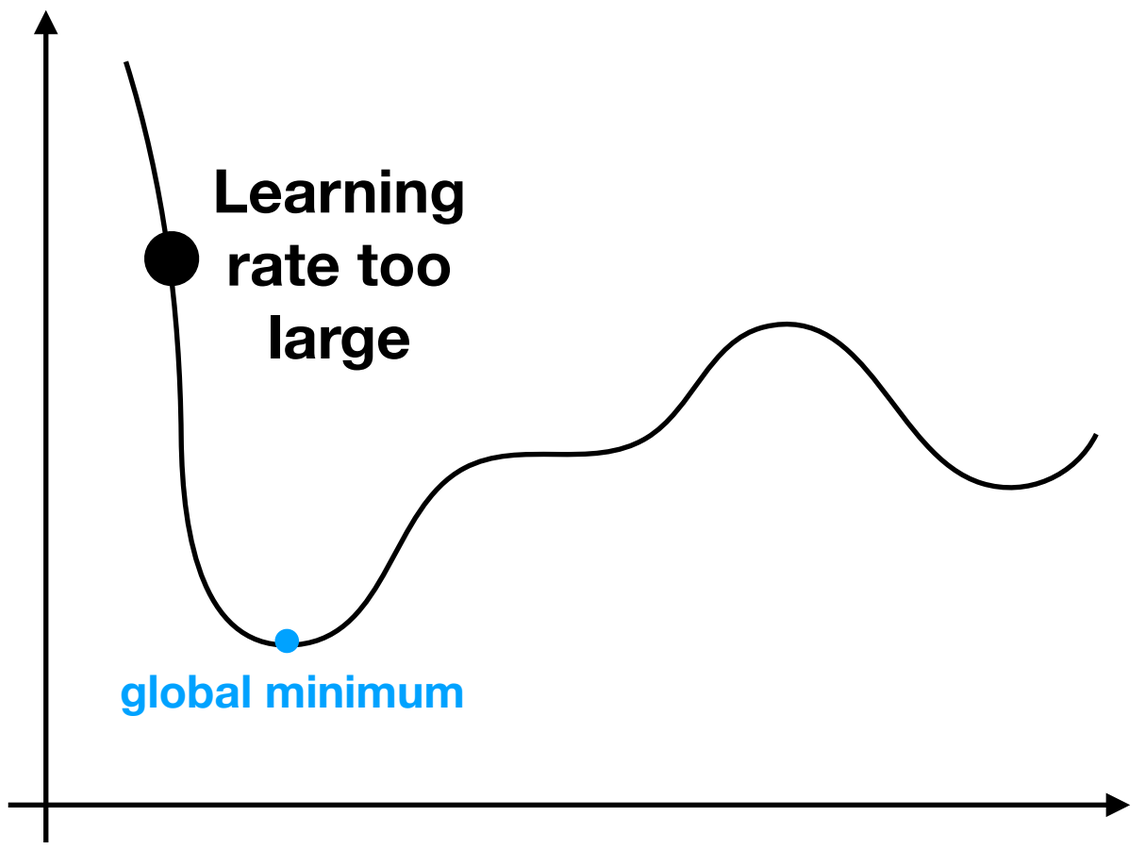
3. Learning rate is too small.



Optimization: what can go wrong?

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

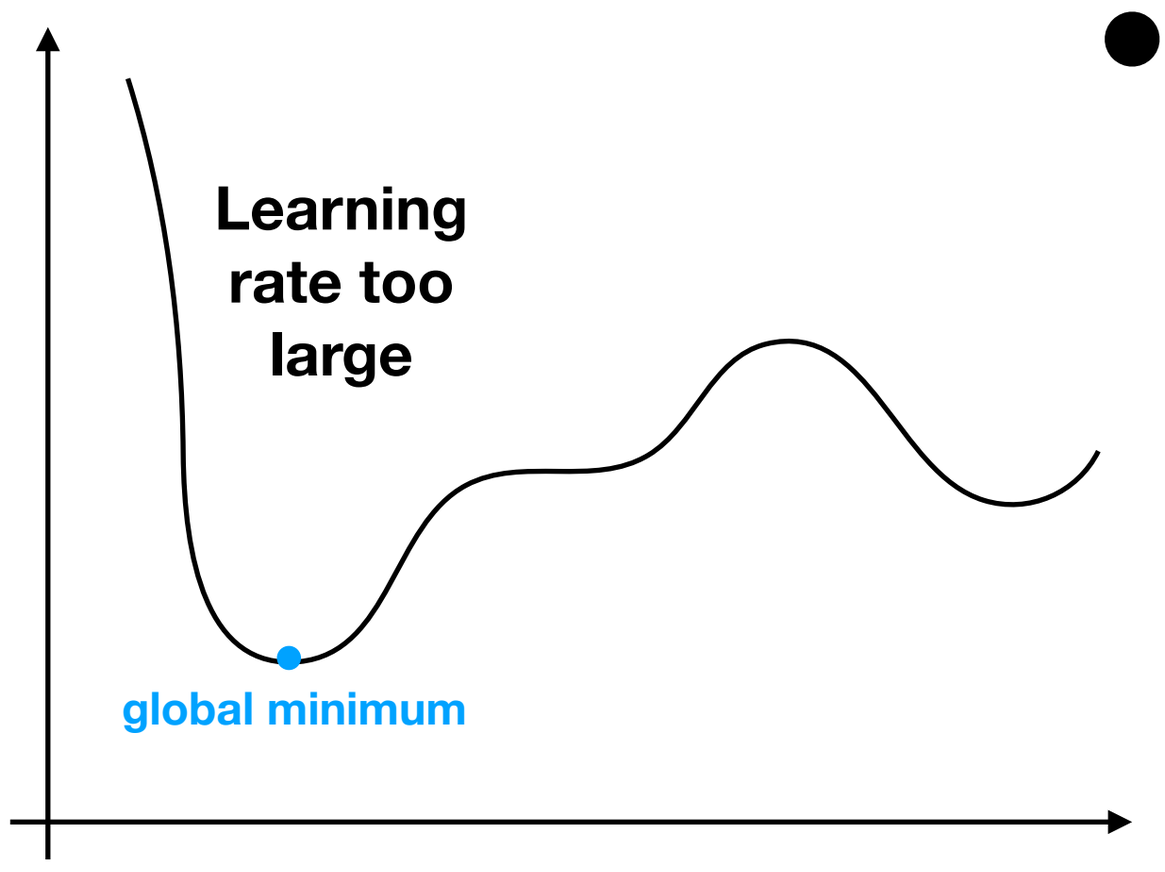
3. Learning rate is too small.



Optimization: what can go wrong?

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

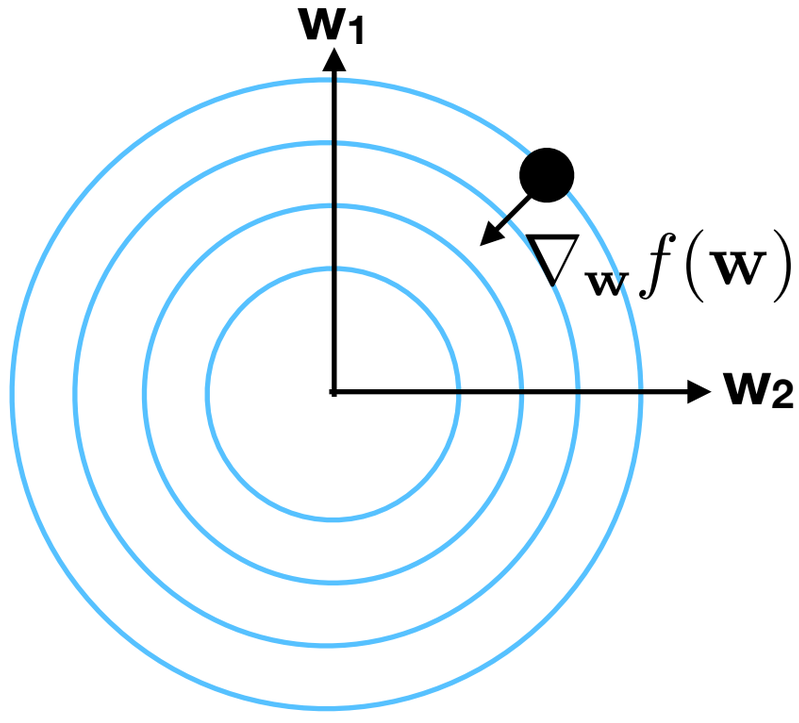
3. Learning rate is too small.



Optimization: what can go wrong?

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

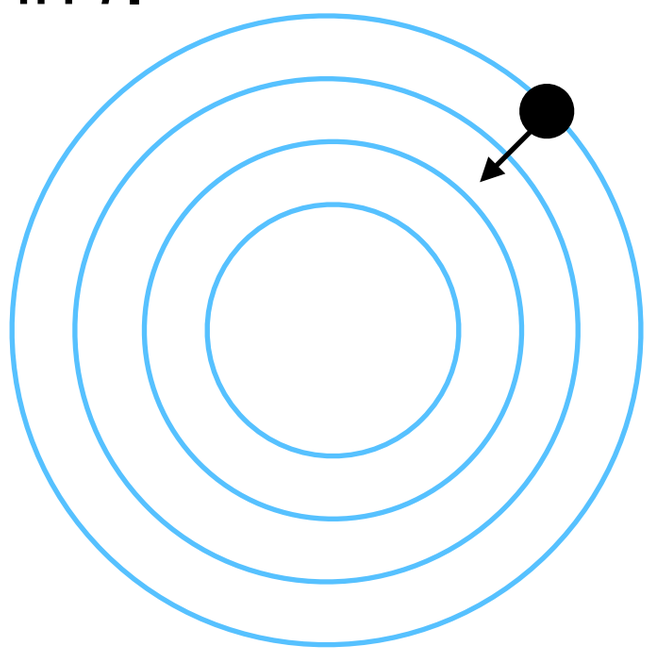
- Consider the cost *f* whose level sets are shown below:



Optimization: what can go wrong?

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

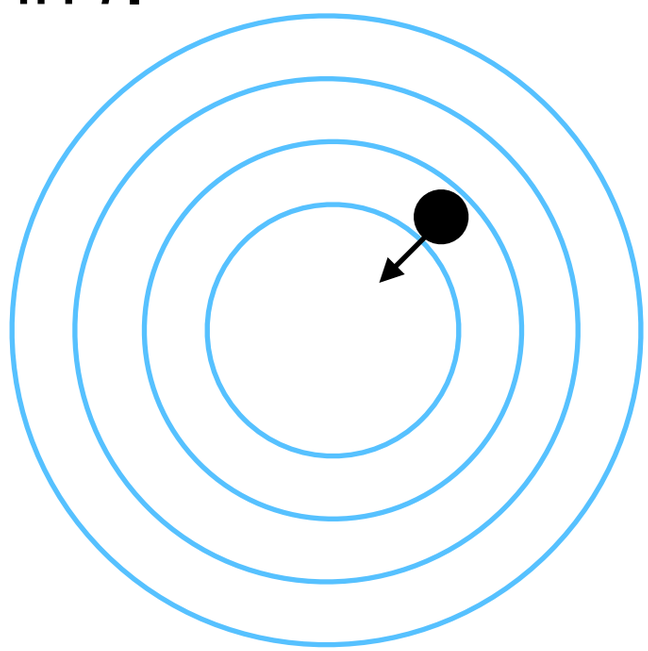
- Gradient descent guides the search along the direction of steepest decrease in *f*.



Optimization: what can go wrong?

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

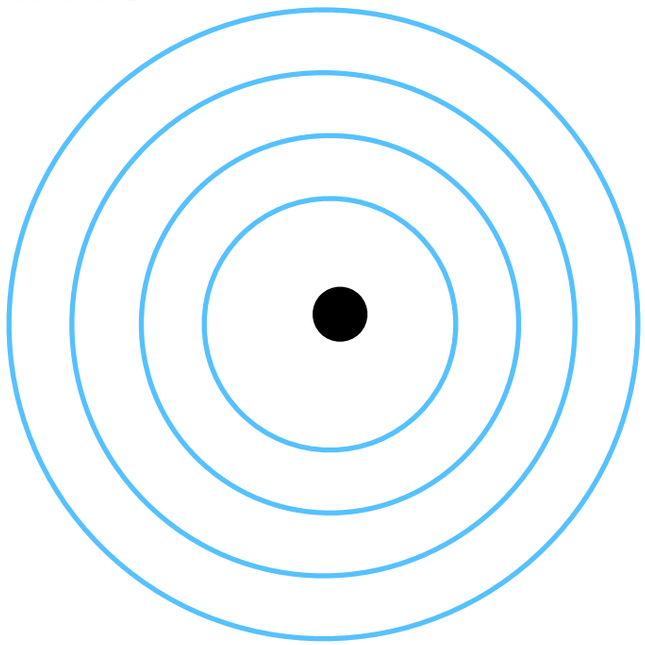
- Gradient descent guides the search along the direction of steepest decrease in *f*.



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?

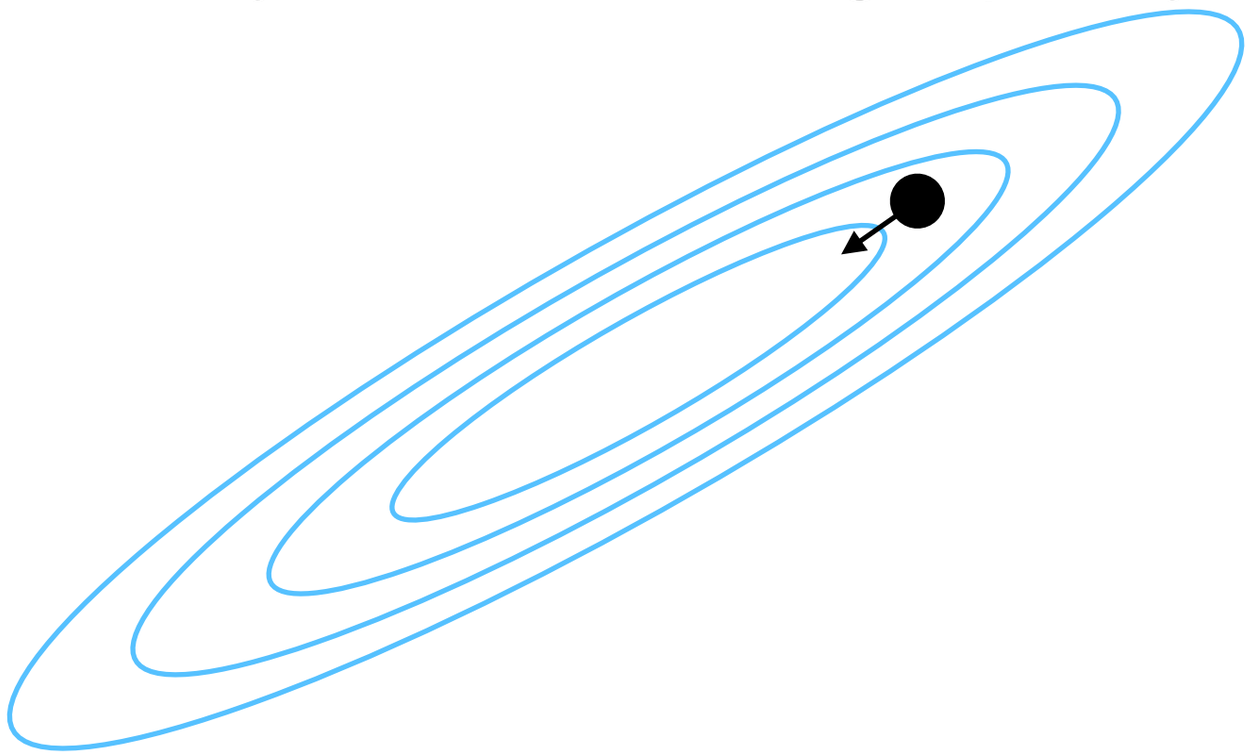
- If we are lucky, we still converge quickly.

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.

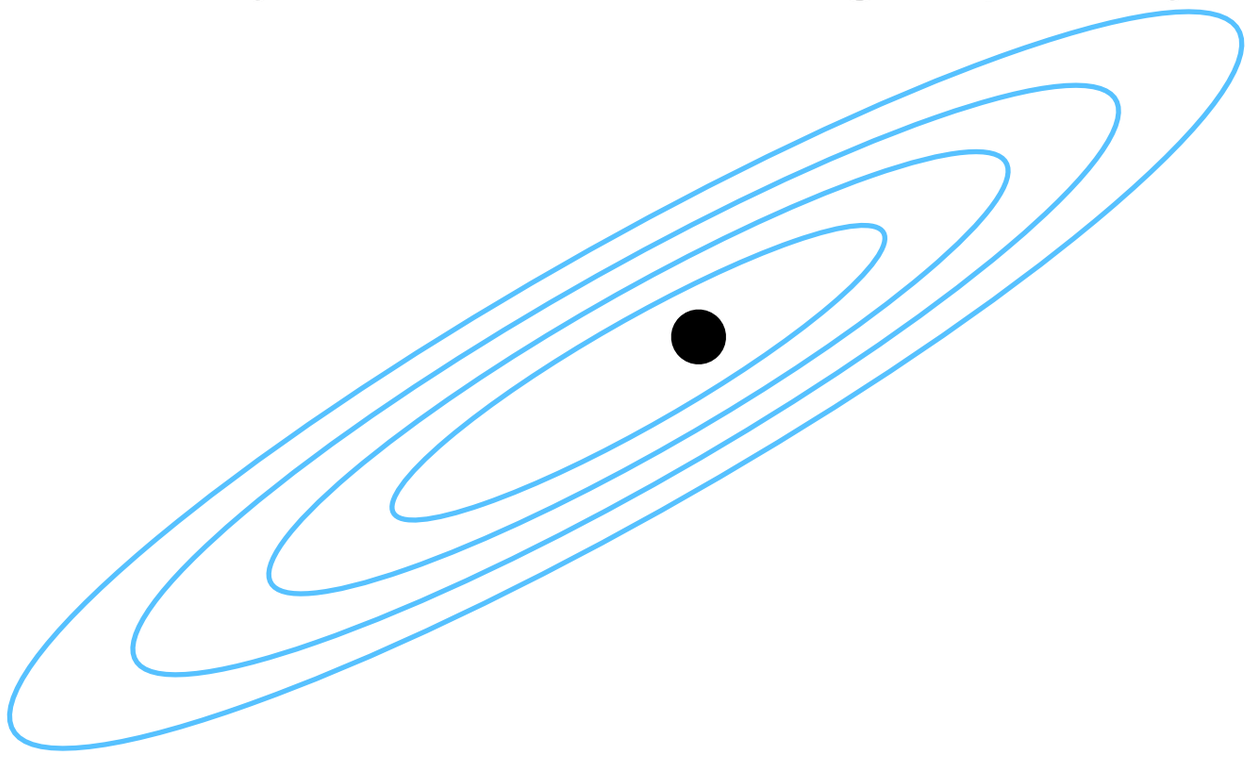


Optimization: what can go wrong?

- 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.

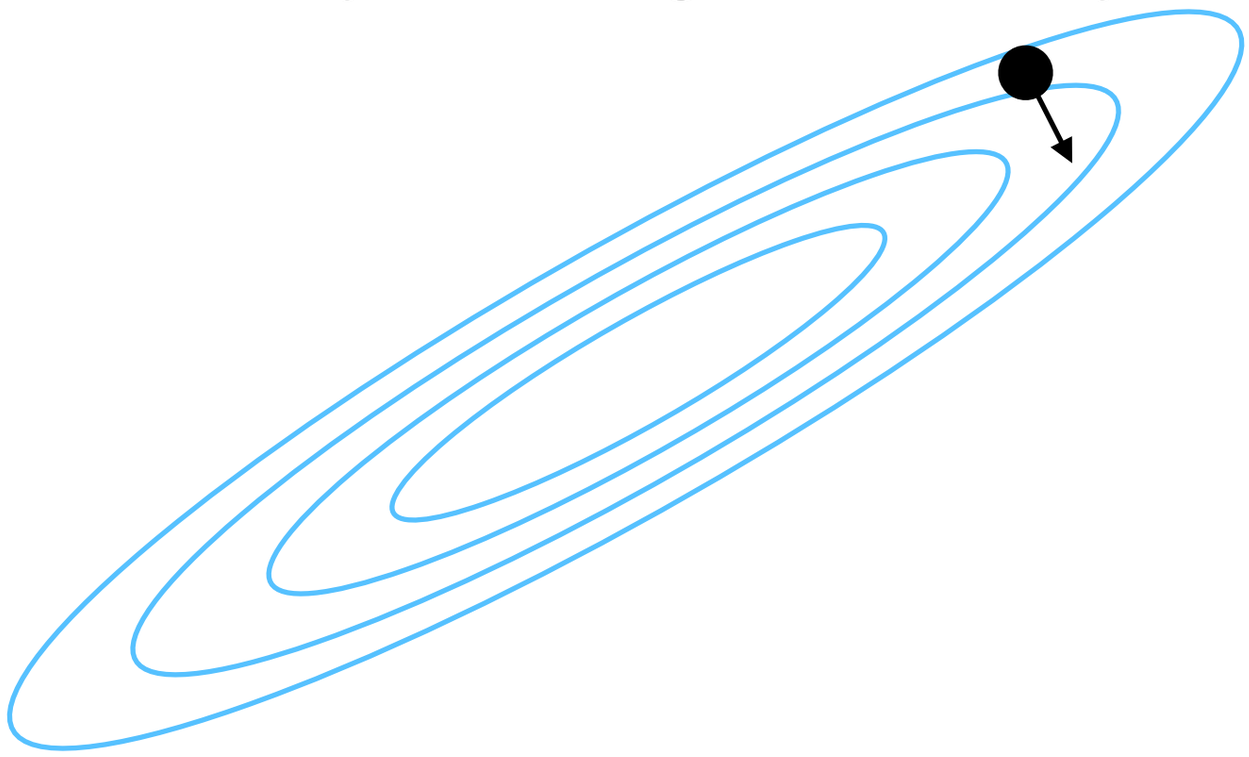


Optimization: what can go wrong?

- 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?

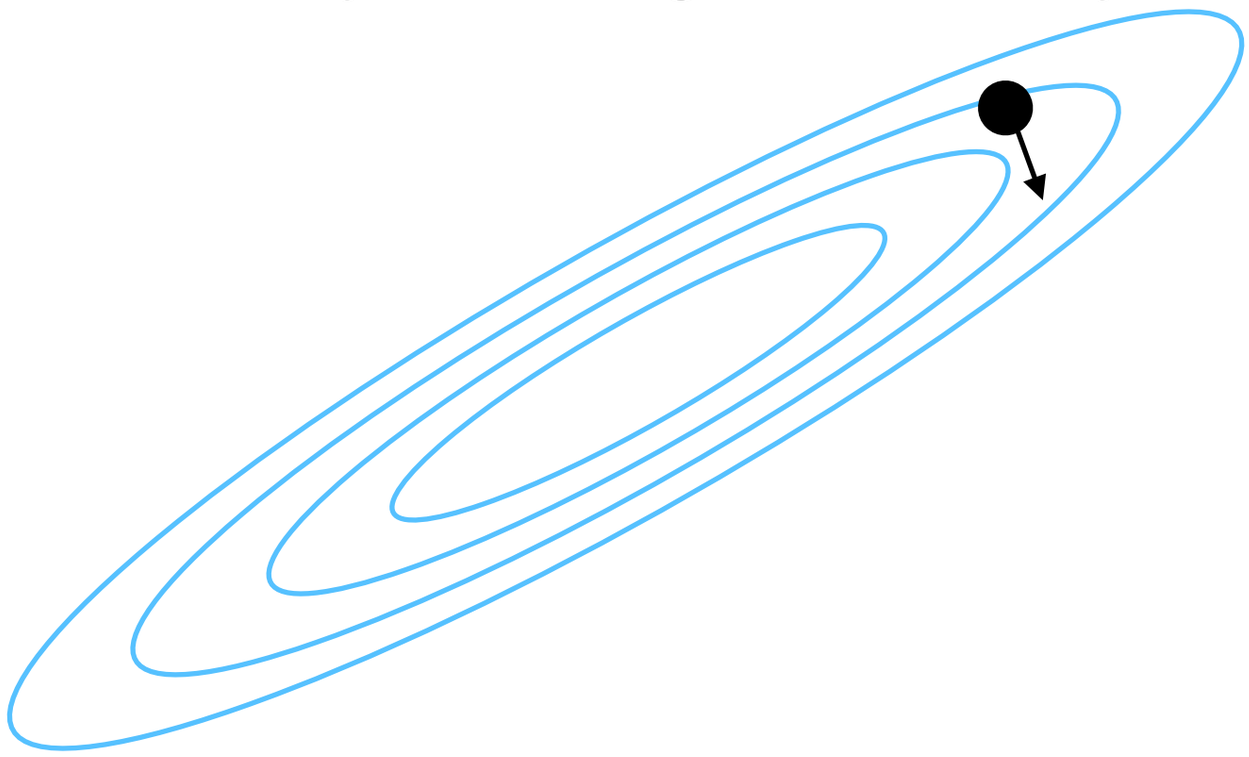
- If we are unlucky, convergence is very slow.

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.

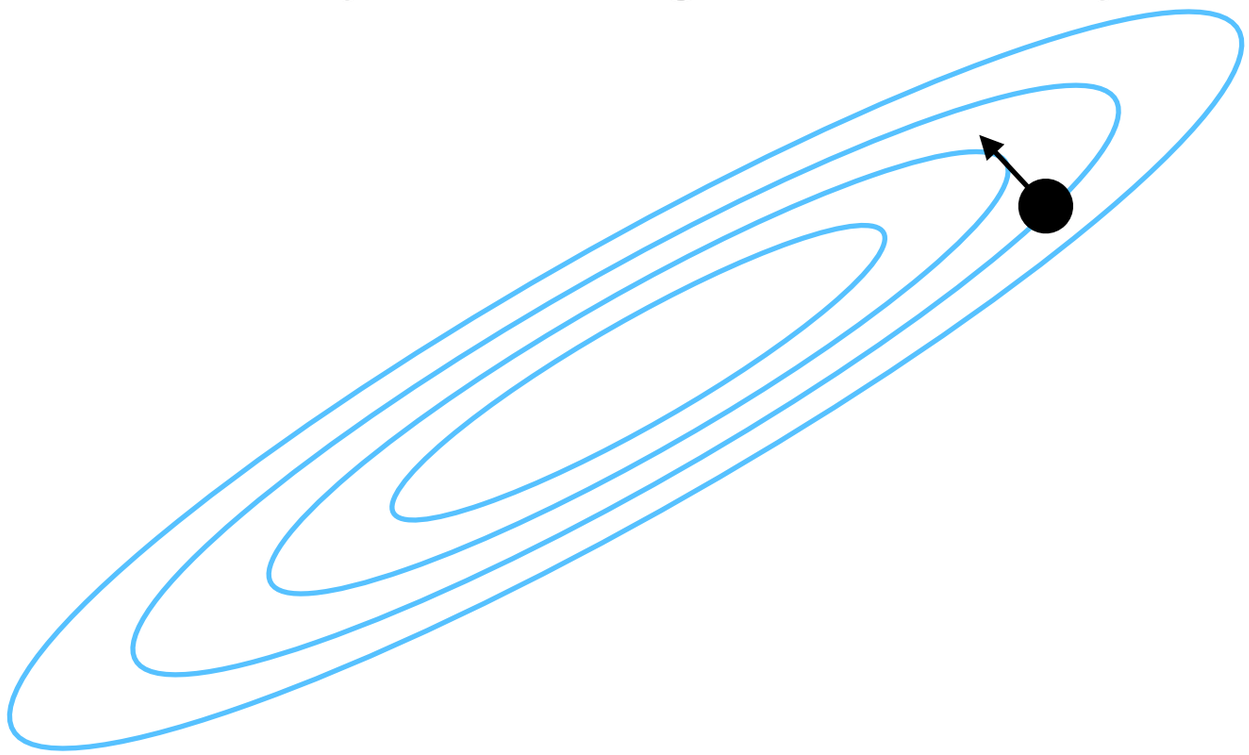


Optimization: what can go wrong?

- 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.

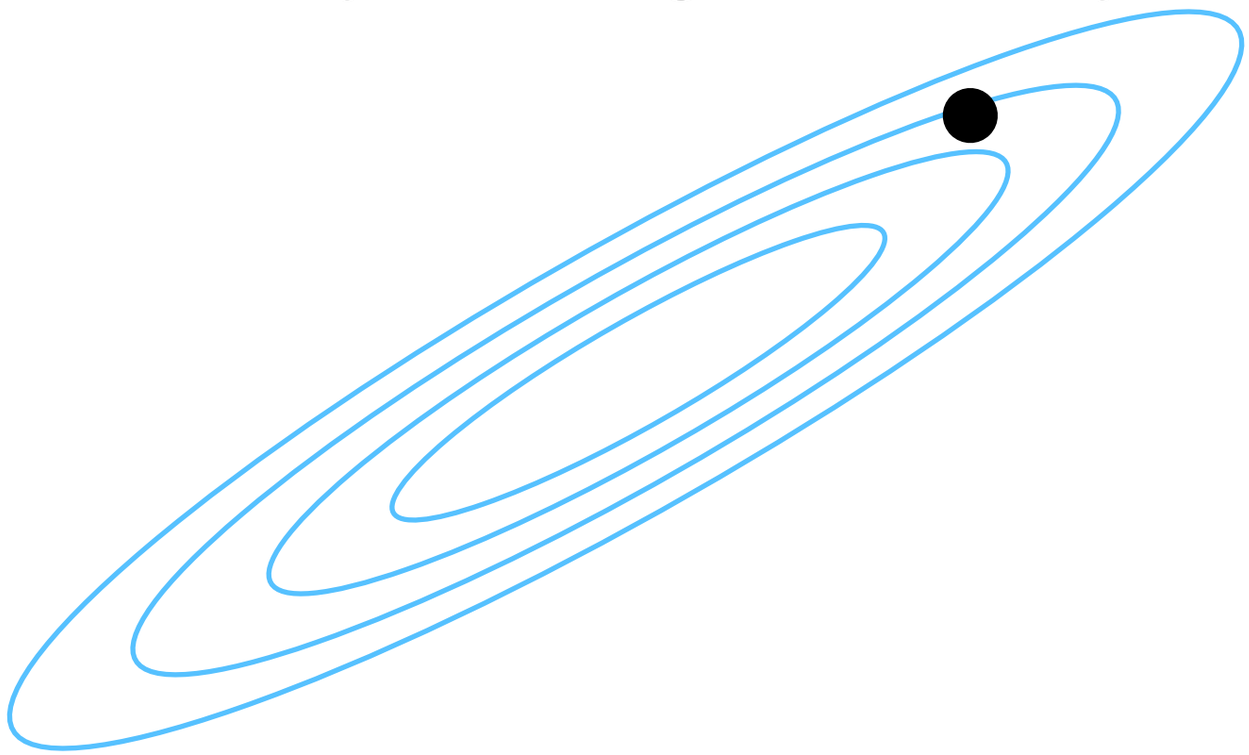


Optimization: what can go wrong?

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

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

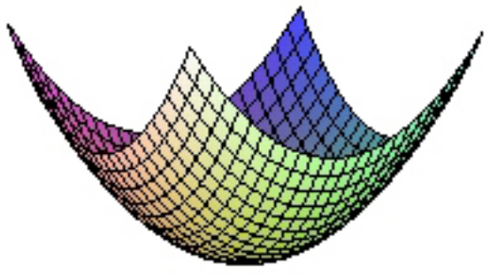
- If we are unlucky, convergence is very slow.



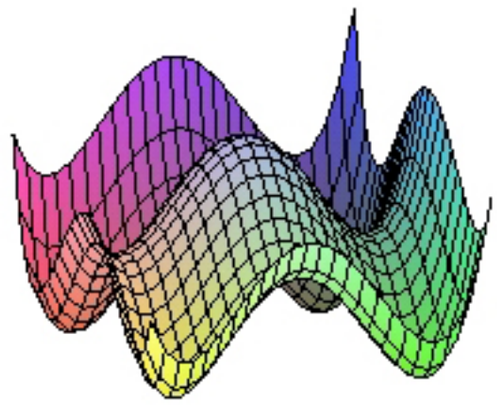
Convexity

Convex ML models

- Linear regression has a loss function that is **convex**.
- With a convex function *f*, *every local minimum is also a global minimum*.



convex



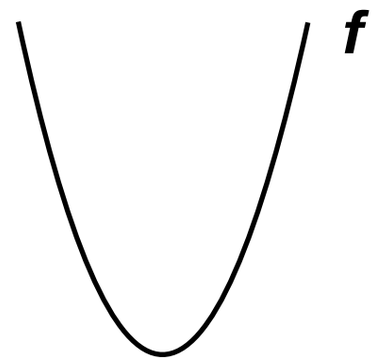
non-convex

- Convex functions are ideal for conducting gradient descent.

https://plus.maths.org/content/convexity

Convexity in 1-d

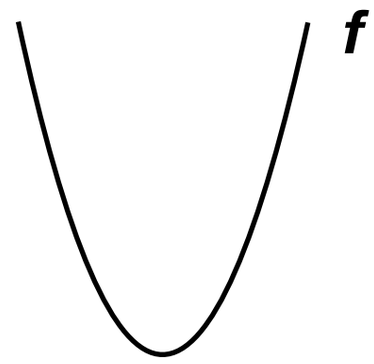
- How can we tell if a 1-d function *f* is convex?



- What property of *f* ensures there is only one local minimum?

Convexity in 1-d

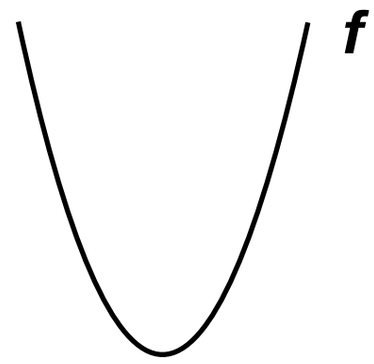
- How can we tell if a 1-d function *f* is convex?



- What property of *f* ensures there is only one local minimum?
  - From left to right, the slope of *f never decreases*.

Convexity in 1-d

- How can we tell if a 1-d function *f* is convex?

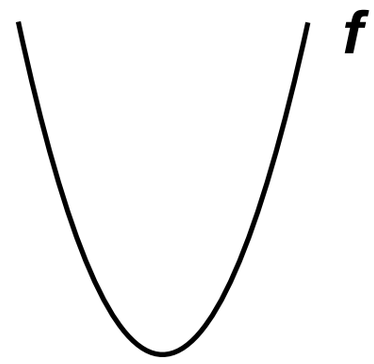


- What property of *f* ensures there is only one local minimum?

  - From left to right, the slope of *f never decreases*.
    ==> the derivative of the slope is always non-negative.

Convexity in 1-d

- How can we tell if a 1-d function *f* is convex?



- What property of *f* ensures there is only one local minimum?

  - From left to right, the slope of *f never decreases*.
    ==> the derivative of the slope is always non-negative.
    ==> the second derivative of *f* is always non-negative.

Convexity in higher dimensions

- For higher-dimensional *f*, convexity is determined by the the Hessian of *f*.

$$\mathrm{H}[f] = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_m \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_m \partial x_m} \end{bmatrix}$$

- For $f : \mathbb{R}^m \to \mathbb{R}$, *f* is convex if the Hessian matrix is positive semi-definite for *every* input **x**.

Positive semi-definite

- Positive semi-definite is the matrix analog of being “non-negative”.

- A real symmetric matrix **A** is **positive semi-definite (PSD)** if (equivalent conditions):

Positive semi-definite

- Positive semi-definite is the matrix analog of being “non-negative”.

- A real symmetric matrix **A** is **positive semi-definite (PSD)** if (equivalent conditions):
  - All its eigenvalues are ≥ 0.
    - In particular, if **A** happens to be diagonal, then **A** is PSD if its eigenvalues are the diagonal elements.

Positive semi-definite

- Positive semi-definite is the matrix analog of being “non-negative”.

- A real symmetric matrix **A** is **positive semi-definite (PSD)** if (equivalent conditions):
  - All its eigenvalues are ≥ 0.
    - In particular, if **A** happens to be diagonal, then **A** is PSD if its eigenvalues are the diagonal elements.
  - For every vector **v**: $\mathbf{v}^T\mathbf{A}\mathbf{v} \geq 0$
    - Therefore: If there exists *any* vector **v** such that $\mathbf{v}^T\mathbf{A}\mathbf{v} < 0$, then **A** is *not* PSD.

Example

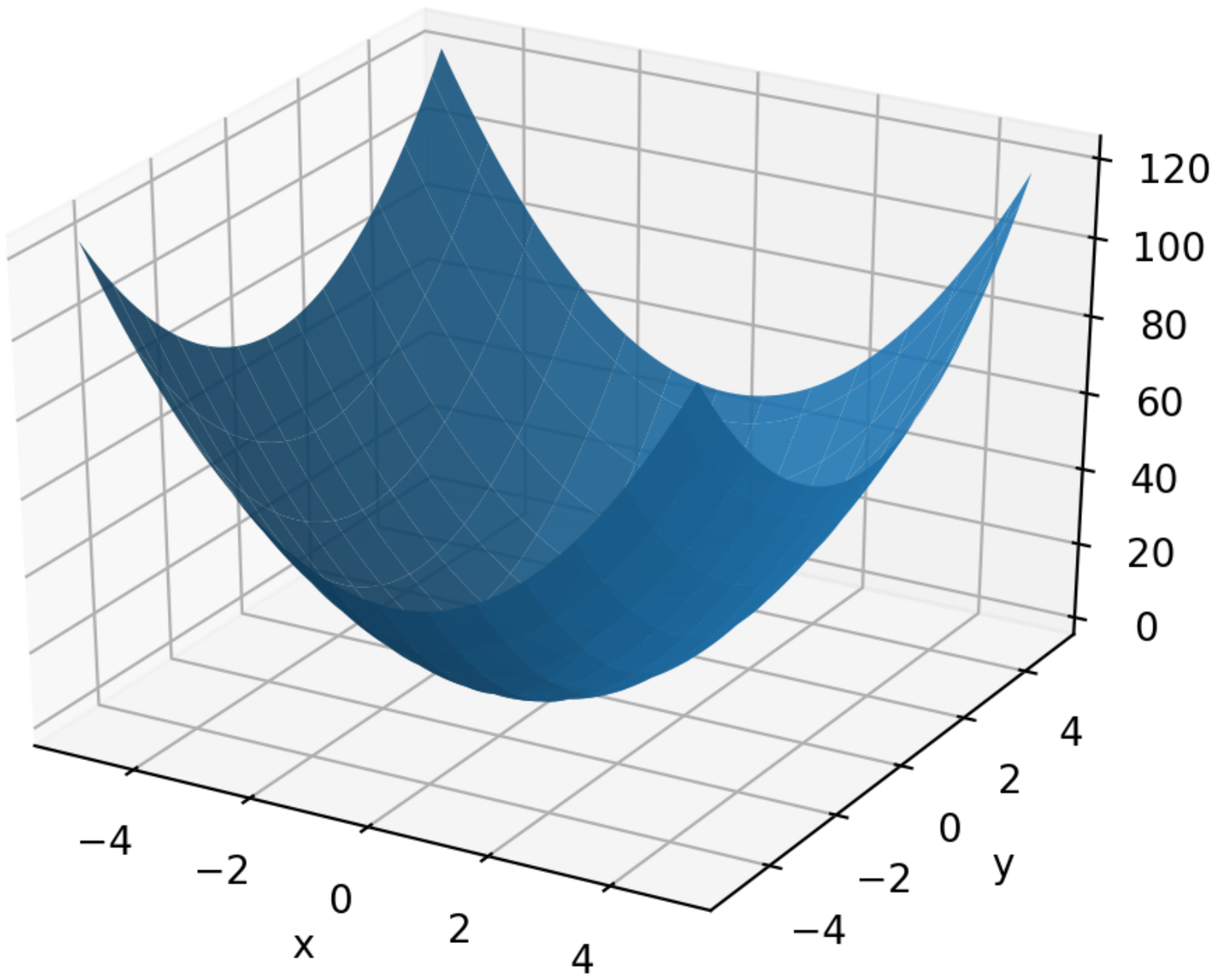
- Suppose $f(x, y) = 3x^2 + 2y^2 - 2$.
- Then the first derivatives are: $\frac{\partial f}{\partial x} = 6x \quad \frac{\partial f}{\partial y} = 4y$
- The Hessian matrix is therefore:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x \partial x} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y \partial y} \end{bmatrix} = \begin{bmatrix} 6 & 0 \\ 0 & 4 \end{bmatrix}$$

- Notice that **H** for this *f* does not depend on (*x*,*y*).
- Also, **H** is a diagonal matrix (with 6 and 4 on the diagonal). Hence, the eigenvalues are just 6 and 4. Since they are both non-negative, then *f* is convex.

Example

- Graph of $f(x, y) = 3x^2 + 2y^2 - 2$:



Exercise

- Recall: if **H** is the Hessian of *f*, then *f* is convex if — at *every* (*x*,*y*) — we can show (equivalently):
  - $\mathbf{v}^T\mathbf{H}\mathbf{v} \geq 0$ for *every* **v**
  - All eigenvalues of **H** are non-negative.
- Which of the following function(s) are convex?
  - $x^2 + y + 5$
  - $x^4 + xy + x^2$

Exercise

- Recall: if **H** is the Hessian of *f*, then *f* is convex if — at *every* (*x*,*y*) — we can show (equivalently):
  - $\mathbf{v}^T\mathbf{H}\mathbf{v} \geq 0$ for *every* **v**
  - All eigenvalues of **H** are non-negative.

- Which of the following function(s) are convex?
  - $x^2 + y + 5 \quad \mathbf{H} = \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix}$
  - $x^4 + xy + x^2$

Convexity of linear regression

- How do we know linear regression is a convex ML model?

- First, recall that, for any matrices **A**, **B** that can be multiplied:
  - $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$

Convexity of linear regression

- How do we know linear regression is a convex ML model?

- Next, recall the gradient and Hessian of f_{MSE} (for linear regression): $f_{\text{MSE}} = \frac{1}{2n}(\mathbf{X}^\top \mathbf{w} - \mathbf{y})^\top (\mathbf{X}^\top \mathbf{w} - \mathbf{y})$

$$\begin{aligned}\nabla_{\mathbf{w}} f_{\text{MSE}} &= \frac{1}{n}\mathbf{X}(\hat{\mathbf{y}} - \mathbf{y}) \\ &= \frac{1}{n}\mathbf{X}(\mathbf{X}^\top \mathbf{w} - \mathbf{y}) \\ \mathbf{H} &= \frac{1}{n}\mathbf{X}\mathbf{X}^\top\end{aligned}$$

Convexity of linear regression

- How do we know linear regression is a convex ML model?

- Next, recall the gradient and Hessian of f_{MSE} (for linear regression): $f_{\text{MSE}} = \frac{1}{2n}(\mathbf{X}^\top \mathbf{w} - \mathbf{y})^\top (\mathbf{X}^\top \mathbf{w} - \mathbf{y})$

$$\begin{aligned}\nabla_{\mathbf{w}} f_{\text{MSE}} &= \frac{1}{n}\mathbf{X}(\hat{\mathbf{y}} - \mathbf{y}) \\ &= \frac{1}{n}\mathbf{X}(\mathbf{X}^\top \mathbf{w} - \mathbf{y}) \\ \mathbf{H} &= \frac{1}{n}\mathbf{X}\mathbf{X}^\top\end{aligned}$$

- For any vector **v**, we have:

$$\begin{aligned}\mathbf{v}^\top \mathbf{X}\mathbf{X}^\top \mathbf{v} &= \left(\mathbf{X}^\top \mathbf{v}\right)^\top \left(\mathbf{X}^\top \mathbf{v}\right) \\ &\geq 0\end{aligned}$$

Convex ML models

- Prominent convex models in ML include linear regression, logistic regression, softmax regression, and support vector machines (SVM).

- However, models in deep learning are generally not convex.

    - Much DL research is devoted to how to optimize the weights to deliver good generalization performance.

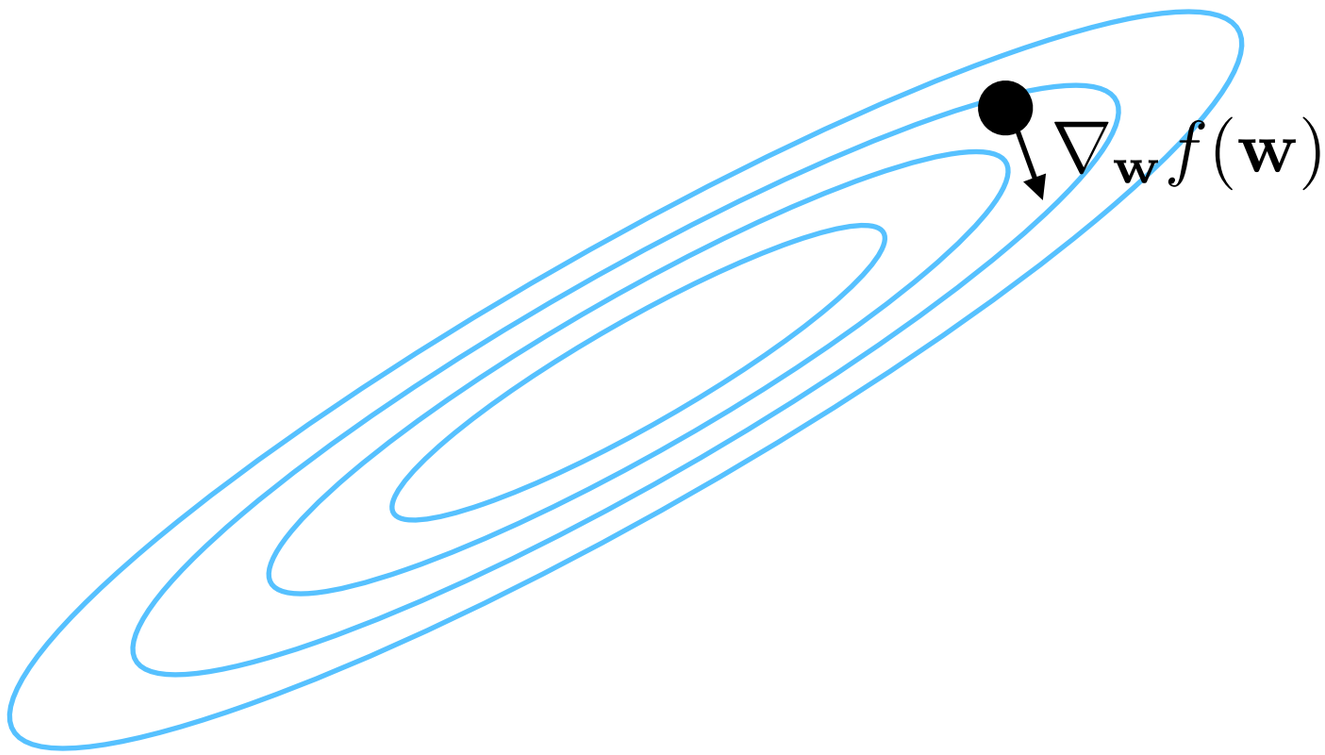
Non-spherical loss functions

Non-spherical loss functions

- As described previously, loss functions that are non-spherical can make hill climbing via gradient descent more difficult:

Curvature

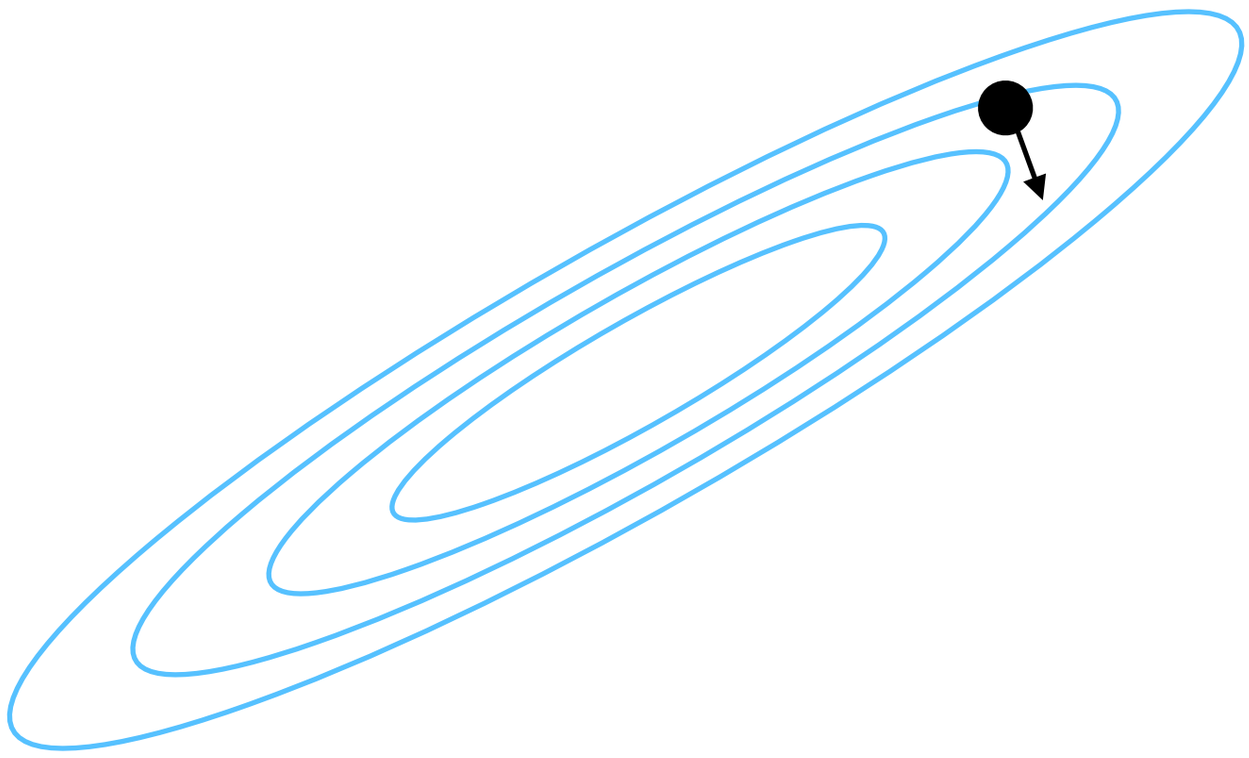
- The problem is that gradient descent only considers slope (1st-order effect), i.e., how f changes with **w**.



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).

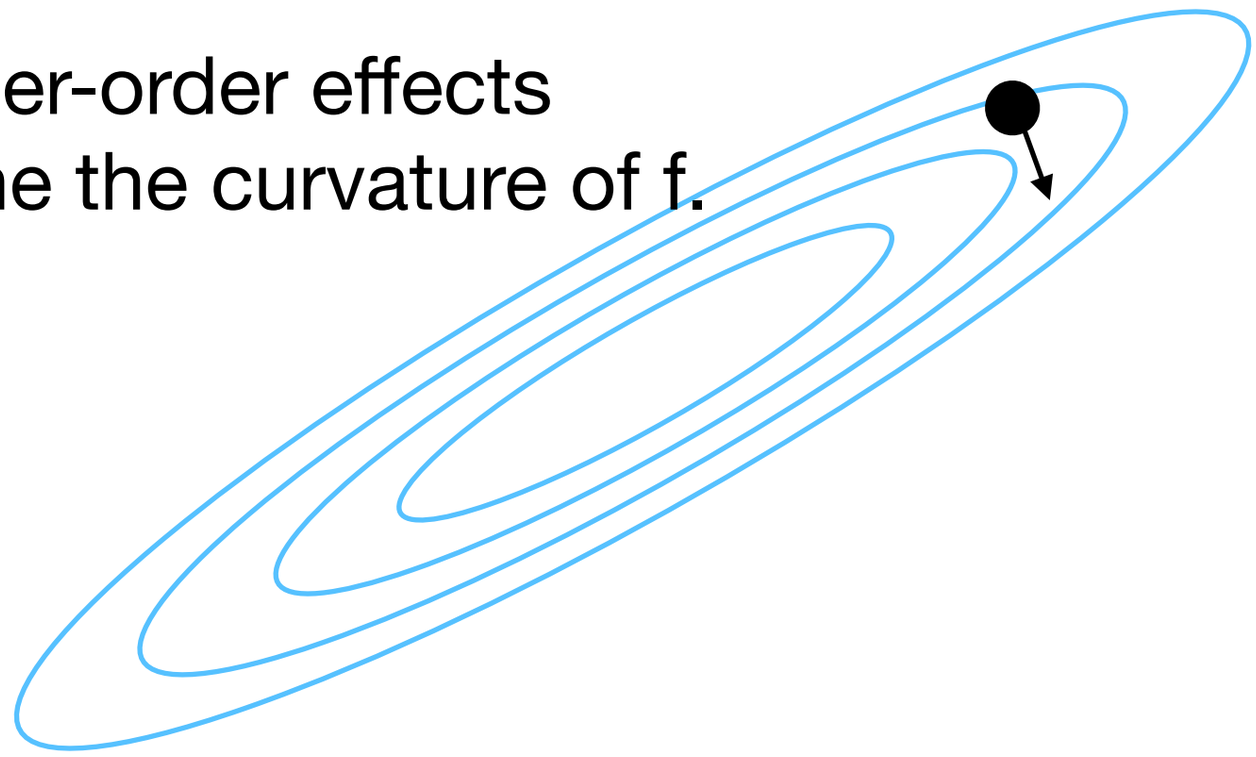


Curvature

- 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.



Curvature

- For linear regression with cost f_{MSE},

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

  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**.

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

Curvature

- For linear regression with cost f_{MSE},

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

  the Hessian is:

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

- In other words, the curvature depends solely on the matrix of training data **X**.

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

Curvature

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

Feature transformations

Whitening transformations

- Gradient descent works best when the level sets of the cost function are spherical.

- We can “spherize” the input features using a **whitening transformation**, which makes the auto-covariance matrix equal the identity matrix **I**.

- We compute this transformation on the training data, 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 $\mathbf{X}\mathbf{X}^{\mathsf{T}}$.
  - We can write its eigendecomposition as:

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

where Φ is the matrix of eigenvectors and Λ 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 $\mathbf{X}\mathbf{X}^{\mathsf{T}}$.
  - We can write its eigendecomposition as:

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

    where Φ is the matrix of eigenvectors and Λ is the corresponding diagonal matrix of eigenvalues.

  - For real-valued features, $\mathbf{X}\mathbf{X}^{\mathsf{T}}$ is real and symmetric; hence, Φ is orthonormal. Also, Λ is non-negative.

*** (uncentered).**

Whitening transformations

- We can find a whitening transform **T** as follows:
  - Therefore, we can multiply both sides by Φ^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 Φ^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 $\mathbf{\Lambda}^{-\frac{1}{2}}$.

Whitening transformations

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

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

Whitening transformations

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

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

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

Whitening transformations

- We can find a whitening transform **T** as follows:
  - Therefore, we can multiply both sides by Φ^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 $\tilde{\mathbf{X}} = \mathbf{T}\mathbf{X}$ is the identity matrix $\mathbf{I}$.

Whitening transformations

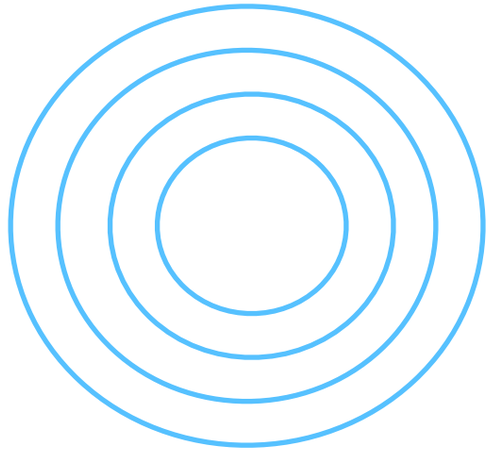
- We have thus derived a transform $\mathbf{T} = \Lambda^{-\frac{1}{2}^\top} \Phi^\top$ such that the (uncentered) auto-covariance of the transformed data $\tilde{\mathbf{X}} = \mathbf{T}\mathbf{X}$ is the identity matrix $\mathbf{I}$.

- $\mathbf{T}$ 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).

Whitening transformations

- Demos (gradient_descent, zscore).

Second-order methods for optimization

Second-order methods for optimization

- An alternative to changing the input features is to use an optimization procedure that considers the 2nd- (or even higher) order terms of the loss function.

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

Newton’s method

- When applicable, it offers faster convergence guarantees (quadratic rather than linear convergence).

- Newton’s method is an iterative method for finding the **roots** of a real-valued function *f*, i.e., **w** such that *f*(**w**)=0.

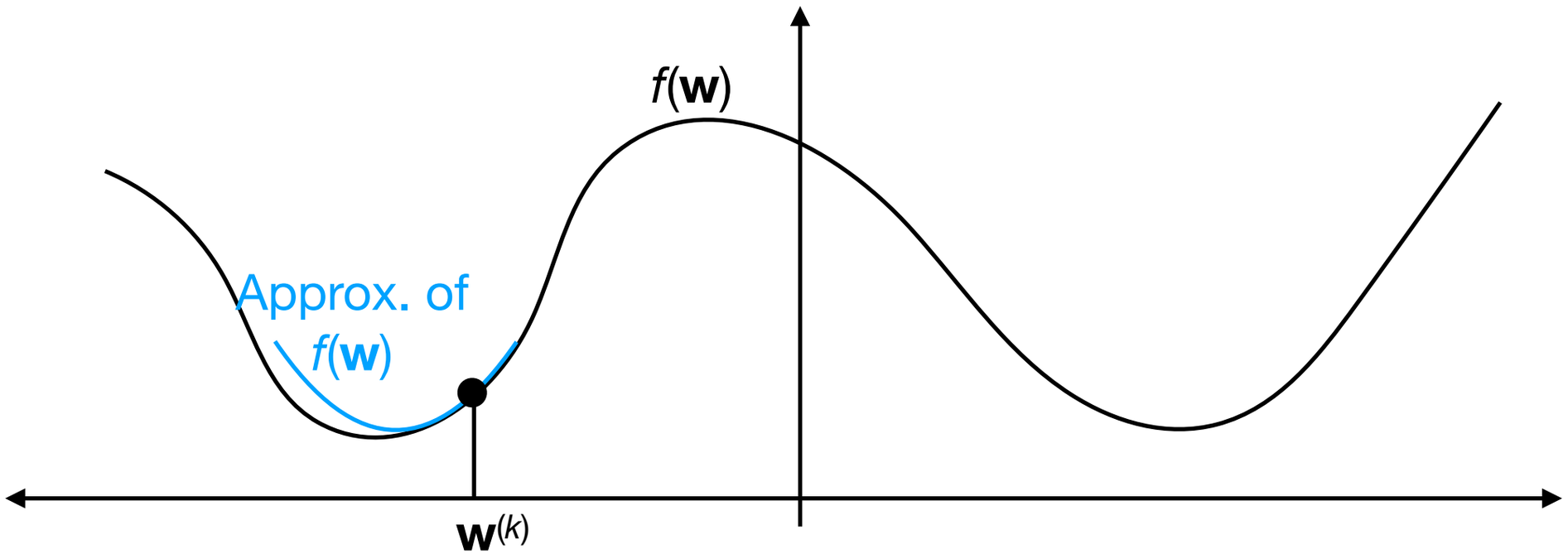
  - This is useful because we can use it to maximize/ minimize a function by finding the roots of the gradient.

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 $\mathbf{w}^{(k)}$.

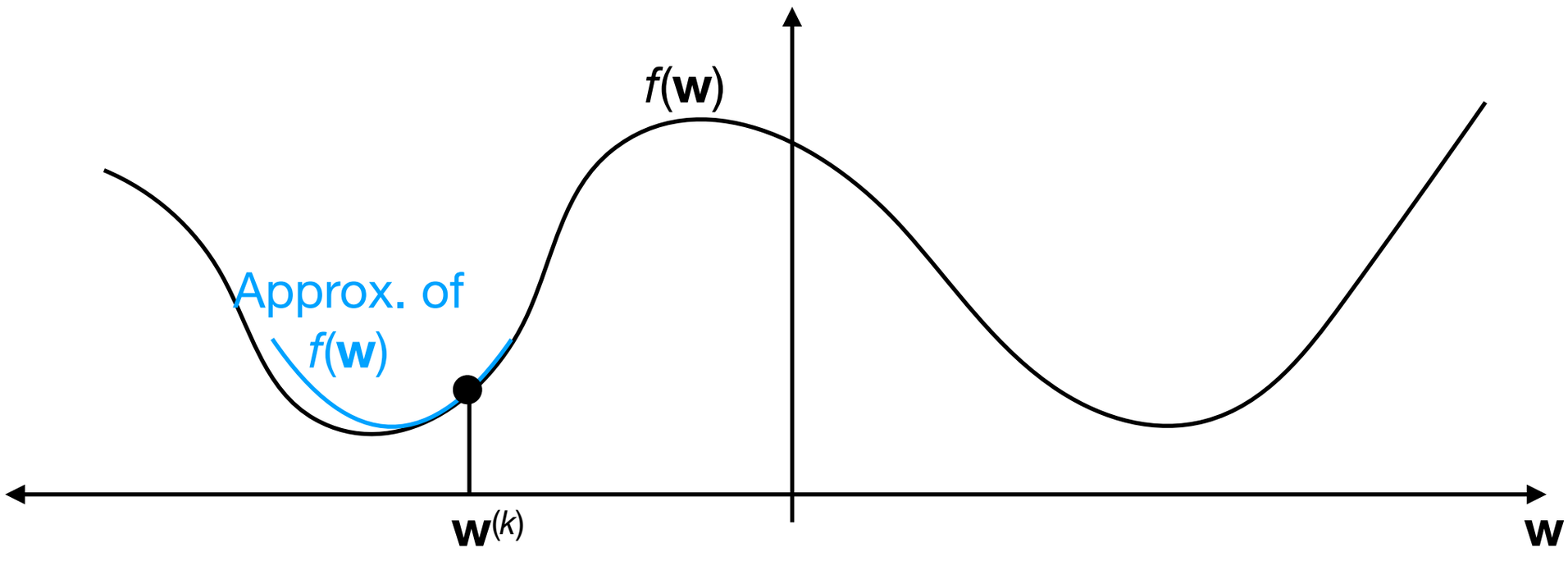


Newton's method

- 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:

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

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

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

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

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

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

Newton's method

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

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

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

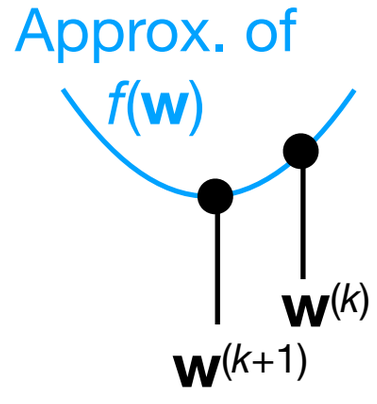
$$= \nabla_{\mathbf{w}} f(\mathbf{w}^{(k)}) + \mathbf{H}\mathbf{w} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)} - \frac{1}{2}\mathbf{H}\mathbf{w}^{(k)}$$

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

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

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

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



Newton's method

- Note that, compared to gradient descent, the update rule in Newton's method replaces the step size ϵ 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).

Logistic regression

Regression vs. classification

- Recall the two main supervised learning cases.
  - **Regression**: predict any real number.
  - **Classification**: choose from a finite set (e.g., {0, 1}).
- So far, we have talked only about the first case.

Binary classification

- The simplest classification problem consists of just 2 classes (binary classification), i.e., $y \in \{ 0, 1 \}$.

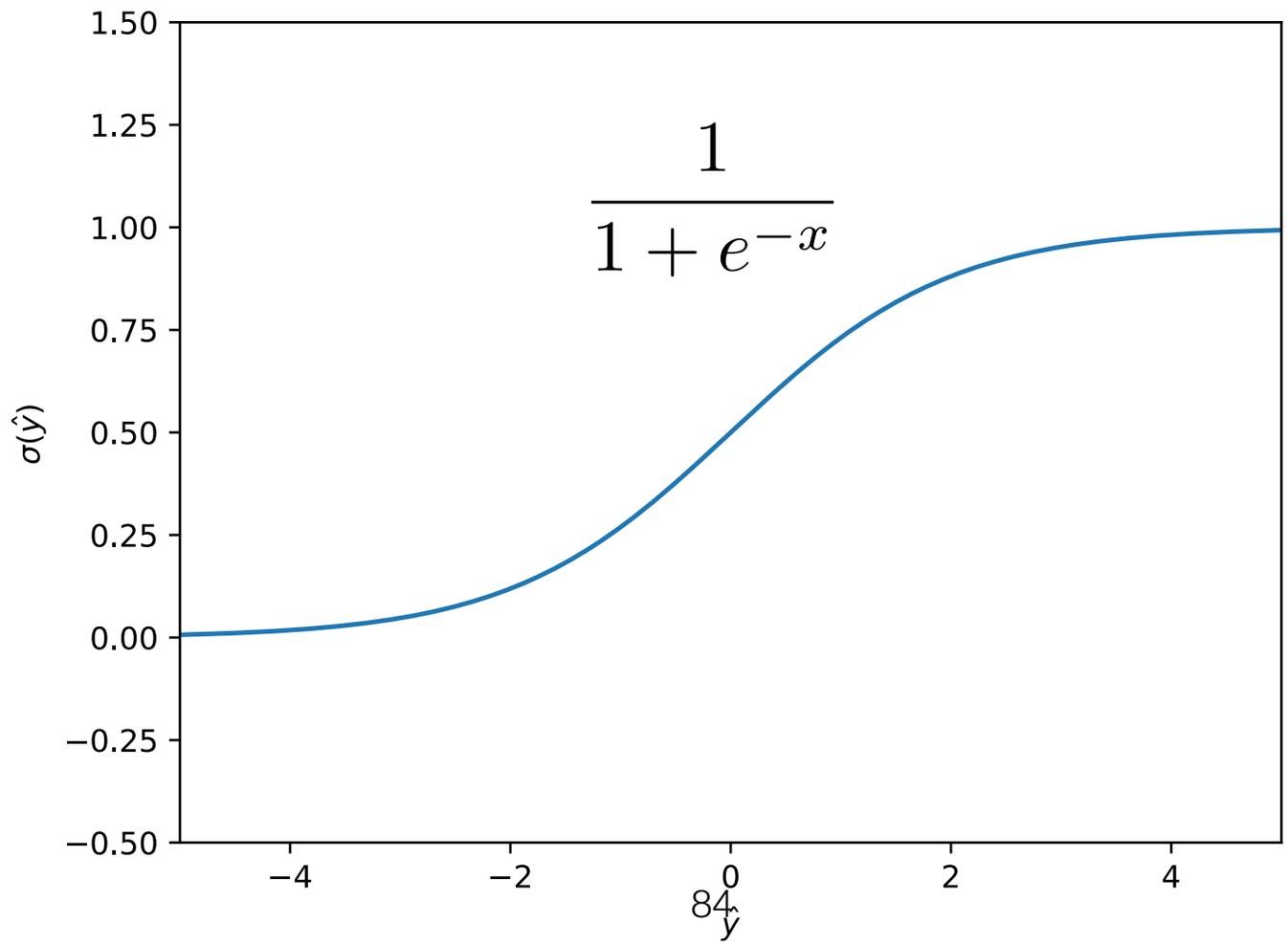
- One of the simplest and most common classification techniques is **logistic regression**.

- Logistic regression is similar to linear regression but also uses a sigmoidal "squashing" function to ensure that $\hat{y} \in (0, 1)$.

Sigmoid: a “squashing” function

- A sigmoid function is an “s”-shaped, monotonically increasing and bounded function.

- Here is the **logistic sigmoid** function σ:



Logistic sigmoid

- The logistic sigmoid function σ has some nice properties:
  - σ(-*z*) = 1 - σ(*z*)

$$
\begin{aligned}
\sigma(z) &= \frac{1}{1+e^{-z}} \\
1-\sigma(z) &= 1-\frac{1}{1+e^{-z}} \\
&= \frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}} \\
&= \frac{e^{-z}}{1+e^{-z}} \\
&= \frac{1}{1/e^{-z}+1} \\
&= \frac{1}{1+e^{z}} \\
&= \sigma(-z)
\end{aligned}
$$

Logistic sigmoid

- The logistic sigmoid function σ has some nice properties:
  - σ'(*z*) = σ(*z*)(1 - σ(*z*))

$$
\begin{aligned}
\sigma(z) &= \frac{1}{1+e^{-z}} \\
\frac{\partial \sigma}{\partial z} = \sigma'(z) &= -\frac{1}{(1+e^{-z})^2}(e^{-z} \times (-1)) \\
&= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{e^{-z}}{1+e^{-z}} \times \frac{1}{1+e^{-z}} \\
&= \frac{1}{1/e^{-z}+1} \times \frac{1}{1+e^{-z}} \\
&= \frac{1}{1+e^{z}} \times \frac{1}{1+e^{-z}} \\
&= \sigma(z)(1-\sigma(z))
\end{aligned}
$$

Logistic regression

- With logistic regression, our predictions are defined as:

$$\hat{y} = \sigma\left(\mathbf{x}^\top \mathbf{w}\right)$$

- Hence, they are forced to be in (0,1).

- For classification, we can interpret the real-valued outputs as probabilities that express how confident we are in a prediction, e.g.:

  - $\hat{y}$=0.95: very confident that a face contains a smile.

  - $\hat{y}$=0.58: not very confident that a face contains a smile.

Logistic regression

- How to train? Unlike linear regression, logistic regression has no analytical (closed-form) solution.

- We can use (stochastic) gradient descent instead.

- We have to apply the **chain-rule of differentiation** to handle the sigmoid function.

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.

- For simplicity, we'll consider just a single example:

$$
\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&=
\end{aligned}
$$

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.

- For simplicity, we'll consider just a single example:

$$
\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&= \left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)
\end{aligned}
$$

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.

- For simplicity, we'll consider just a single example:

$$
\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&= \left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right) \sigma(\mathbf{x}^\top \mathbf{w}) \left(1 - \sigma(\mathbf{x}^\top \mathbf{w})\right)
\end{aligned}
$$

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.

- For simplicity, we'll consider just a single example:

$$
\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&= \mathbf{x}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)\sigma(\mathbf{x}^\top \mathbf{w})\left(1 - \sigma(\mathbf{x}^\top \mathbf{w})\right)
\end{aligned}
$$

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.

- For simplicity, we'll consider just a single example:

$$
\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&= \mathbf{x}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)\sigma(\mathbf{x}^\top \mathbf{w})\left(1 - \sigma(\mathbf{x}^\top \mathbf{w})\right) \\
&= \mathbf{x}\,(\hat{y} - y)\,\hat{y}\,(1 - \hat{y})
\end{aligned}
$$

Gradient descent for logistic regression

- Let's compute the gradient of f_{MSE} for logistic regression.

- For simplicity, we'll consider just a single example:

$$
\begin{aligned}
f_{\text{MSE}}(\mathbf{w}) &= \frac{1}{2}(\hat{y} - y)^2 \\
&= \frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2 \\
\nabla_{\mathbf{w}} f_{\text{MSE}}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[\frac{1}{2}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)^2\right] \\
&= \mathbf{x}\left(\sigma(\mathbf{x}^\top \mathbf{w}) - y\right)\sigma(\mathbf{x}^\top \mathbf{w})\left(1 - \sigma(\mathbf{x}^\top \mathbf{w})\right) \\
&= \mathbf{x}\,(\hat{y} - y)\,\hat{y}\,(1 - \hat{y})
\end{aligned}
$$

Notice the extra multiplicative terms compared to the gradient for *linear* regression: x(*ŷ* - *y*)

Attenuated gradient

- What if the weights **w** are initially chosen badly, so that $\hat{y}$ is very close to 1, even though $y = 0$ (or vice-versa)?
  - Then $\hat{y}(1 - \hat{y})$ is close to 0.
- In this case, the gradient:

$$\nabla_{\mathbf{w}} f_{\mathrm{MSE}}(\mathbf{w}) = \mathbf{x}\,(\hat{y} - y)\,\hat{y}\,(1 - \hat{y})$$

  will be very close to 0.

- If the gradient is 0, then no learning will occur!

Different cost function

- For this reason, logistic regression is typically trained using a different cost function from f_{MSE}.

- One particularly well-suited cost function uses logarithms.

- Logarithms and the logistic sigmoid interact well:

$$\frac{\partial}{\partial \mathbf{w}} \left[\log \sigma(\mathbf{x}^\top \mathbf{w}) \right] =$$

Different cost function

- For this reason, logistic regression is typically trained using a different cost function from f_{MSE}.

- One particularly well-suited cost function uses logarithms.

- Logarithms and the logistic sigmoid interact well:

$$\frac{\partial}{\partial \mathbf{w}} \left[\log \sigma(\mathbf{x}^\top \mathbf{w}) \right] = \mathbf{x} \frac{1}{\sigma(\mathbf{x}^\top \mathbf{w})} \sigma(\mathbf{x}^\top \mathbf{w}) \left(1 - \sigma(\mathbf{x}^\top \mathbf{w}) \right)$$

Different cost function

- For this reason, logistic regression is typically trained using a different cost function from f_{MSE}.

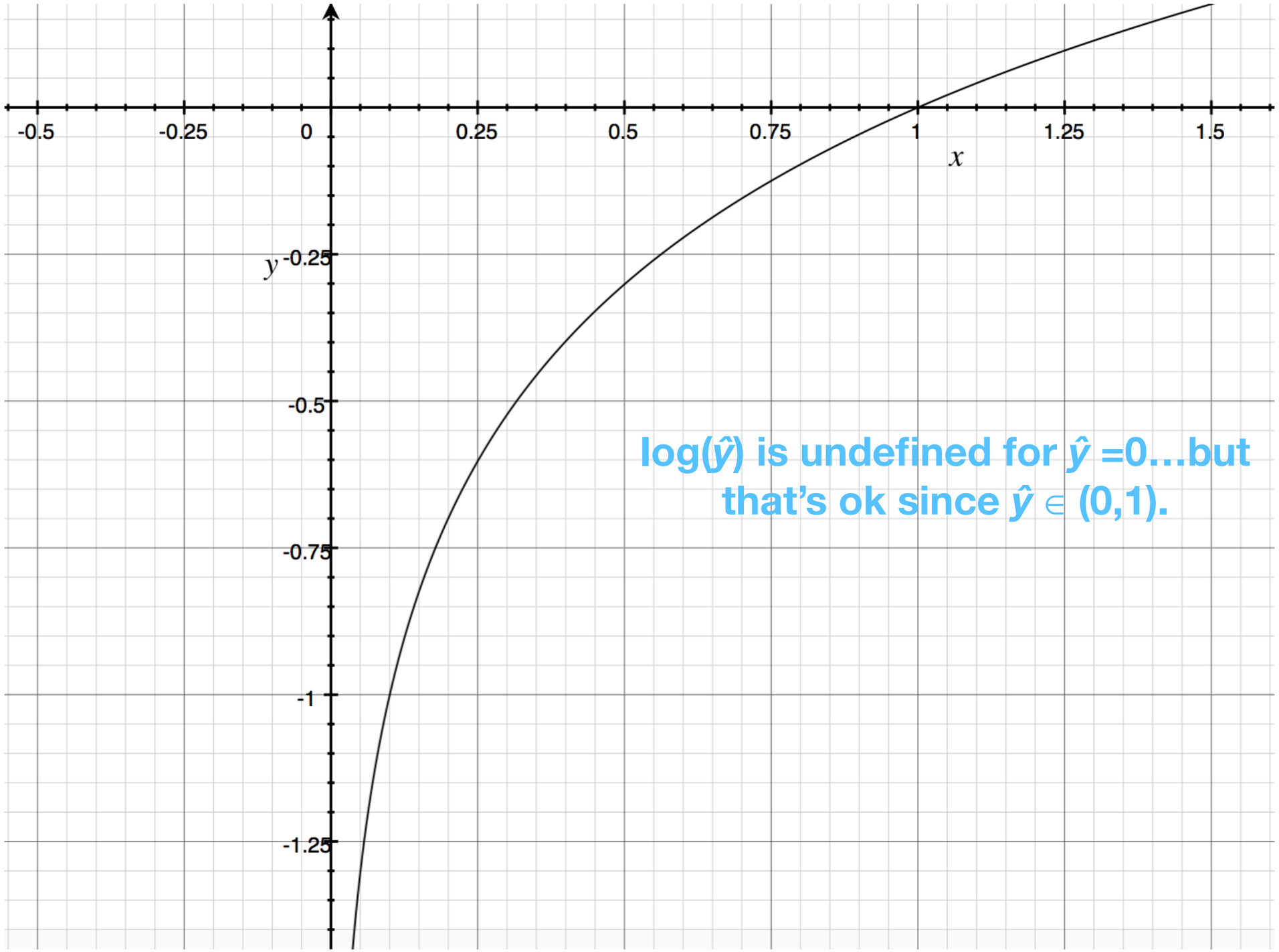
- One particularly well-suited cost function uses logarithms.

- Logarithms and the logistic sigmoid interact well:

$$\frac{\partial}{\partial \mathbf{w}} \left[\log \sigma(\mathbf{x}^\top \mathbf{w})\right] = \mathbf{x} \frac{1}{\sigma(\mathbf{x}^\top \mathbf{w})} \sigma(\mathbf{x}^\top \mathbf{w}) \left(1 - \sigma(\mathbf{x}^\top \mathbf{w})\right)$$
$$= \mathbf{x} \left(1 - \sigma(\mathbf{x}^\top \mathbf{w})\right)$$

The gradient of log(σ) is surprisingly simple.

Logarithm function



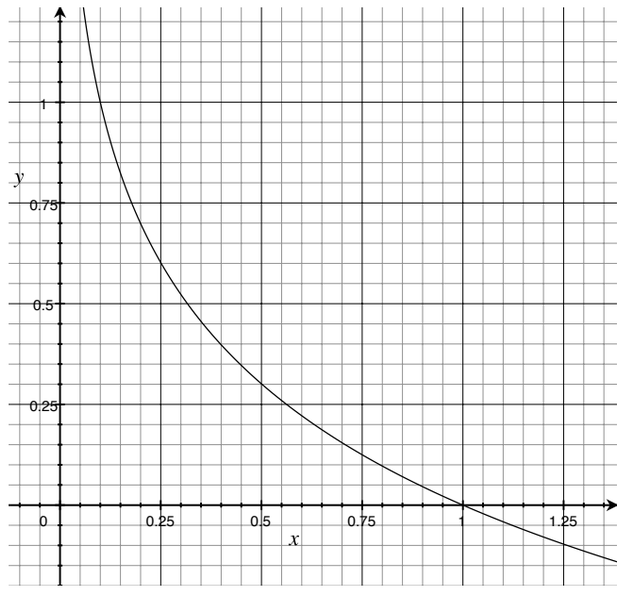
Log loss

- We want to assign a large loss when y=1 but $\hat{y}$=0

- We typically use the **log-loss** for logistic regression:

$$-y \log \hat{y}$$

The *y* or (1-*y*) "selects" which term in the expression is active, based on the ground-truth label.



-log($\hat{y}$)

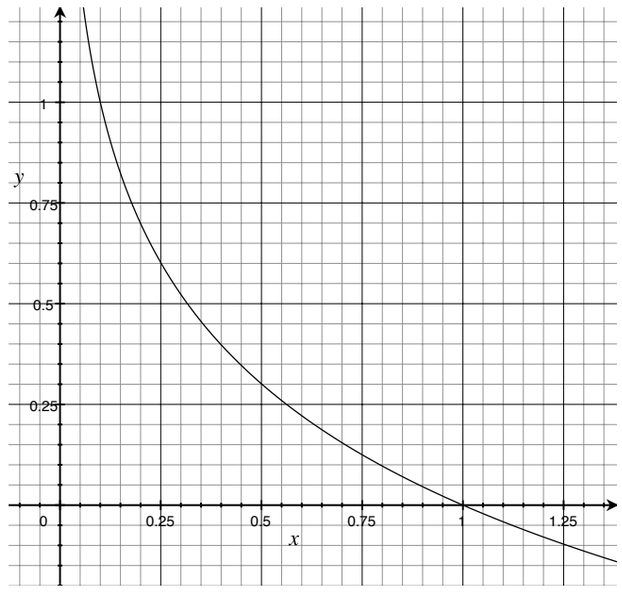
Log loss

- We want to assign a large loss when y=1 but $\hat{y}$=0, and for y=0 but $\hat{y}$=1.

- We typically use the **log-loss** for logistic regression:

$$-y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

The *y* or (1-*y*) "selects" which term in the expression is active, based on the ground-truth label.



-log($\hat{y}$)

Gradient descent for logistic regression with log-loss

$$\nabla_{\mathbf{w}} f_{\log}(\mathbf{w}) \quad = \quad \nabla_{\mathbf{w}} \left[-\left(y \log \hat{y} - (1-y) \log(1-\hat{y}) \right) \right]$$

Gradient descent for logistic regression with log-loss

$$
\begin{aligned}
\nabla_{\mathbf{w}} f_{\log}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[-\left(y \log \hat{y} - (1-y) \log(1-\hat{y}) \right) \right] \\
&= -\nabla_{\mathbf{w}} \left(y \log \sigma(\mathbf{x}^\top \mathbf{w}) + (1-y) \log(1 - \sigma(\mathbf{x}^\top \mathbf{w})) \right)
\end{aligned}
$$

Gradient descent for logistic regression with log-loss

$$
\begin{aligned}
\nabla_{\mathbf{w}} f_{\log}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[-\left(y \log \hat{y} - (1-y) \log(1-\hat{y}) \right) \right] \\
&= -\nabla_{\mathbf{w}} \left(y \log \sigma(\mathbf{x}^\top \mathbf{w}) + (1-y) \log(1 - \sigma(\mathbf{x}^\top \mathbf{w})) \right) \\
&= -\left(y \frac{\mathbf{x} \sigma(\mathbf{x}^\top \mathbf{w})(1 - \sigma(\mathbf{x}^\top \mathbf{w}))}{\sigma(\mathbf{x}^\top \mathbf{w})} - (1-y) \frac{\mathbf{x} \sigma(\mathbf{x}^\top \mathbf{w})(1 - \sigma(\mathbf{x}^\top \mathbf{w}))}{1 - \sigma(\mathbf{x}^\top \mathbf{w})} \right)
\end{aligned}
$$

Gradient descent for logistic regression with log-loss

$$
\begin{aligned}
\nabla_{\mathbf{w}} f_{\log}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left[-\left(y \log \hat{y} - (1-y) \log(1-\hat{y}) \right) \right] \\
&= -\nabla_{\mathbf{w}} \left(y \log \sigma(\mathbf{x}^\top \mathbf{w}) + (1-y) \log(1 - \sigma(\mathbf{x}^\top \mathbf{w})) \right) \\
&= -\left(y \frac{\mathbf{x}\sigma(\mathbf{x}^\top \mathbf{w})(1-\sigma(\mathbf{x}^\top \mathbf{w}))}{\sigma(\mathbf{x}^\top \mathbf{w})} - (1-y) \frac{\mathbf{x}\sigma(\mathbf{x}^\top \mathbf{w})(1-\sigma(\mathbf{x}^\top \mathbf{w}))}{1-\sigma(\mathbf{x}^\top \mathbf{w})} \right) \\
&= -\left(y\mathbf{x}(1-\sigma(\mathbf{x}^\top \mathbf{w})) - (1-y)\mathbf{x}\sigma(\mathbf{x}^\top \mathbf{w}) \right) \\
&= -\mathbf{x} \left(y - y\sigma(\mathbf{x}^\top \mathbf{w}) - \sigma(\mathbf{x}^\top \mathbf{w}) + y\sigma(\mathbf{x}^\top \mathbf{w}) \right) \\
&= -\mathbf{x} \left(y - \sigma(\mathbf{x}^\top \mathbf{w}) \right) \\
&= \mathbf{x}(\hat{y} - y) \quad \textbf{Same as for linear regression!}
\end{aligned}
$$

Linear regression versus logistic regression

	Linear regression	Logistic regression
Primary use	Regression	Classification
Prediction ($\hat{y}$)	$\hat{y} = \mathbf{x}^\top \mathbf{w}$	$\hat{y} = \sigma(\mathbf{x}^\top \mathbf{w})$
Cost/Loss	f_{MSE}	$f_{\log}$
Gradient	$\mathbf{x}(\hat{y} - y)$	$\mathbf{x}(\hat{y} - y)$

- Logistic regression is used primarily for *classification* even though it's called "regression".
- Logistic regression is an instance of a **generalized linear model** — a linear model combined with a **link function** (e.g., logistic sigmoid).
  - In DL, link functions are typically called **activation functions**.

Softmax regression (aka multinomial logistic regression)

Multi-class classification

- So far we have talked about classifying only 2 classes (e.g., smile versus non-smile).
  - This is sometimes called **binary classification**.
- But there are many settings in which multiple (>2) classes exist, e.g., emotion recognition, hand-written digit recognition:

6 classes (fear, anger, sadness, happiness, disgust, surprise)

10 classes (0-9)

Classification versus regression

- Note that, even though the hand-written digit recognition ("MNIST") problem has classes called "0" , "1", …, "9", there is no sense of "distance" between the classes.

- Misclassifying a 1 as a 2 is just as "bad" as misclassifying a 1 as a 9.

Multi-class classification

- It turns out that logistic regression can easily be extended to support an arbitrary number (≥2) of classes.
  - The multi-class case is called **softmax regression** or sometimes **multinomial logistic regression.**

- How to represent the ground-truth y and prediction $\hat{y}$?
  - Instead of just a scalar y, we will use a vector **y**.

Example: 2 classes

- Suppose we have a dataset of 3 examples, where the ground-truth class labels are 0, 1, 0.

Example: 2 classes

- Suppose we have a dataset of 3 examples, where the ground-truth class labels are 0, 1, 0.

- Then we would define our ground-truth vectors as:

$$\mathbf{y}^{(1)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\mathbf{y}^{(2)} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\mathbf{y}^{(3)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

- Exactly 1 coordinate of each **y** is 1; the others are 0.

Example: 2 classes

- Suppose we have a dataset of 3 examples, where the ground-truth class labels are 0, 1, 0.

- Then we would define our ground-truth vectors as:

$$\mathbf{y}^{(1)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \mathbf{y}^{(2)} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \mathbf{y}^{(3)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This “slot” is for class 0.

- This is called a **one-hot encoding** of the class label.

Example: 2 classes

- Suppose we have a dataset of 3 examples, where the ground-truth class labels are 0, 1, 0.

- Then we would define our ground-truth vectors as:

$$\mathbf{y}^{(1)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$\mathbf{y}^{(2)} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\mathbf{y}^{(3)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This “slot” is for class 1.

- This is called a **one-hot encoding** of the class label.

Example: 2 classes

- The machine's predictions $\hat{\mathbf{y}}$ about each example's label are also **probabilistic**.

- They could consist of:

$$\hat{\mathbf{y}}^{(1)} = \begin{bmatrix} 0.93 \\ 0.07 \end{bmatrix} \longleftarrow \text{Machine's "belief" that the label is 0.}$$

$$\hat{\mathbf{y}}^{(2)} = \begin{bmatrix} 0.4 \\ 0.6 \end{bmatrix}$$

$$\hat{\mathbf{y}}^{(3)} = \begin{bmatrix} 0.99 \\ 0.01 \end{bmatrix}$$

- Each coordinate of $\hat{\boldsymbol{y}}$ is a probability.

Example: 2 classes

- The machine's predictions $\hat{\mathbf{y}}$ about each example's label are also **probabilistic**.

- They could consist of:

$$\hat{\mathbf{y}}^{(1)} = \begin{bmatrix} 0.93 \\ 0.07 \end{bmatrix} \longleftarrow \text{Machine's "belief" that the label is 1.}$$

$$\hat{\mathbf{y}}^{(2)} = \begin{bmatrix} 0.4 \\ 0.6 \end{bmatrix}$$

$$\hat{\mathbf{y}}^{(3)} = \begin{bmatrix} 0.99 \\ 0.01 \end{bmatrix}$$

- The sum of the coordinates in each $\hat{\mathbf{y}}$ is 1.

Softmax activation function

- Logistic regression outputs a *scalar* label $\hat{y}$ representing the probability that the label is 1.
  - We needed just a single weight vector $\mathbf{w}$, so that $\hat{y} = \sigma(\mathbf{x}^T\mathbf{w})$.

Softmax activation function

- Logistic regression outputs a *scalar* label $\hat{y}$ representing the probability that the label is 1.
  - We needed just a single weight vector $\mathbf{w}$, so that $\hat{y} = \sigma(\mathbf{x}^T\mathbf{w})$.

- Softmax regression outputs a *c-vector* representing the probabilities that the label is k=1, …, c.
  - We need c different vectors of weights $\mathbf{w}^{(1)}$, …, $\mathbf{w}^{(c)}$.
  - Weight vector $\mathbf{w}^{(k)}$ computes how much input $\mathbf{x}$ “agrees” with class k.

Softmax activation function

- With softmax regression, we first compute:

$$z_1 = \mathbf{x}^\top \mathbf{w}^{(1)}$$

$$z_2 = \mathbf{x}^\top \mathbf{w}^{(2)}$$

$$\vdots$$

$$z_c = \mathbf{x}^\top \mathbf{w}^{(c)}$$

I will refer to the z's as "pre-activation scores".

Softmax activation function

- With softmax regression, we first compute:

$$z_1 = \mathbf{x}^\top \mathbf{w}^{(1)}$$

$$z_2 = \mathbf{x}^\top \mathbf{w}^{(2)}$$

$$\vdots$$

$$z_c = \mathbf{x}^\top \mathbf{w}^{(c)}$$

- We then **normalize** across all *c* classes so that:
  1. Each output $\hat{y}_k$ is non-negative.
  2. The sum of $\hat{y}_k$ over all *c* classes is 1.

Normalization of the $\hat{y}_k$

1. To enforce non-negativity, we can exponentiate each z_k:

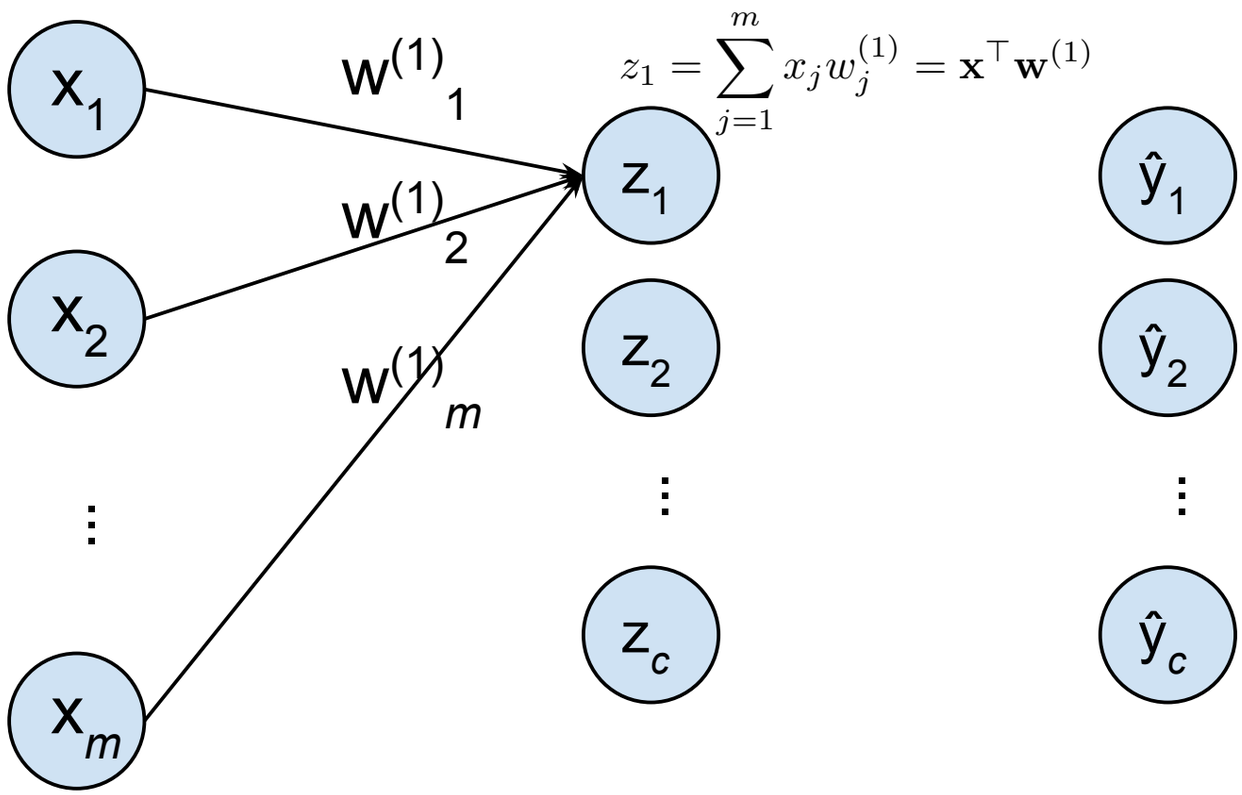
$$\hat{y}_k = \frac{\exp(z_k)}{}$$

Normalization of the $\hat{y}_k$

2. To enforce that the $\hat{y}_k$ sum to 1, we can divide each entry by the sum:

$$\hat{y}_k = \frac{\exp(z_k)}{\sum_{k'=1}^{c} \exp(z_{k'})}$$

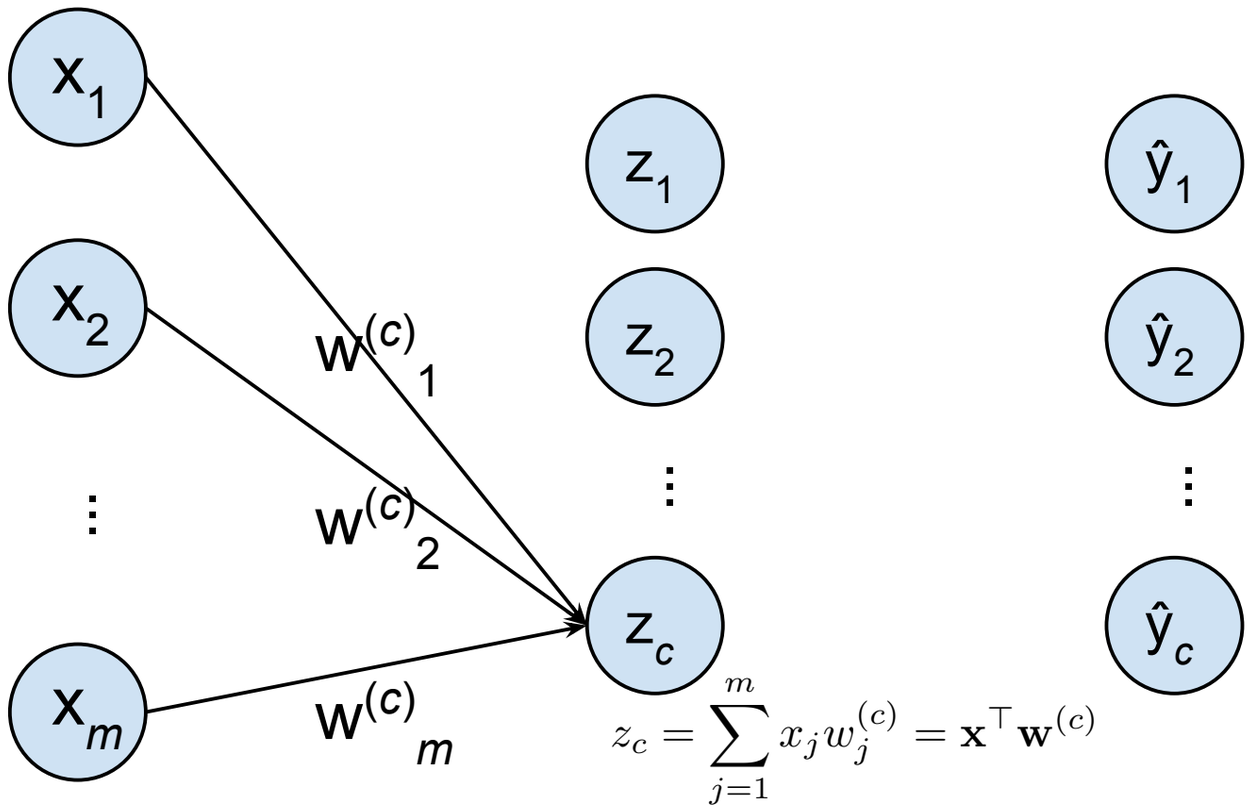
Softmax regression diagram



- With softmax regression, we first compute:

$$z_1 = \mathbf{x}^\top \mathbf{w}^{(1)}$$

Softmax regression diagram



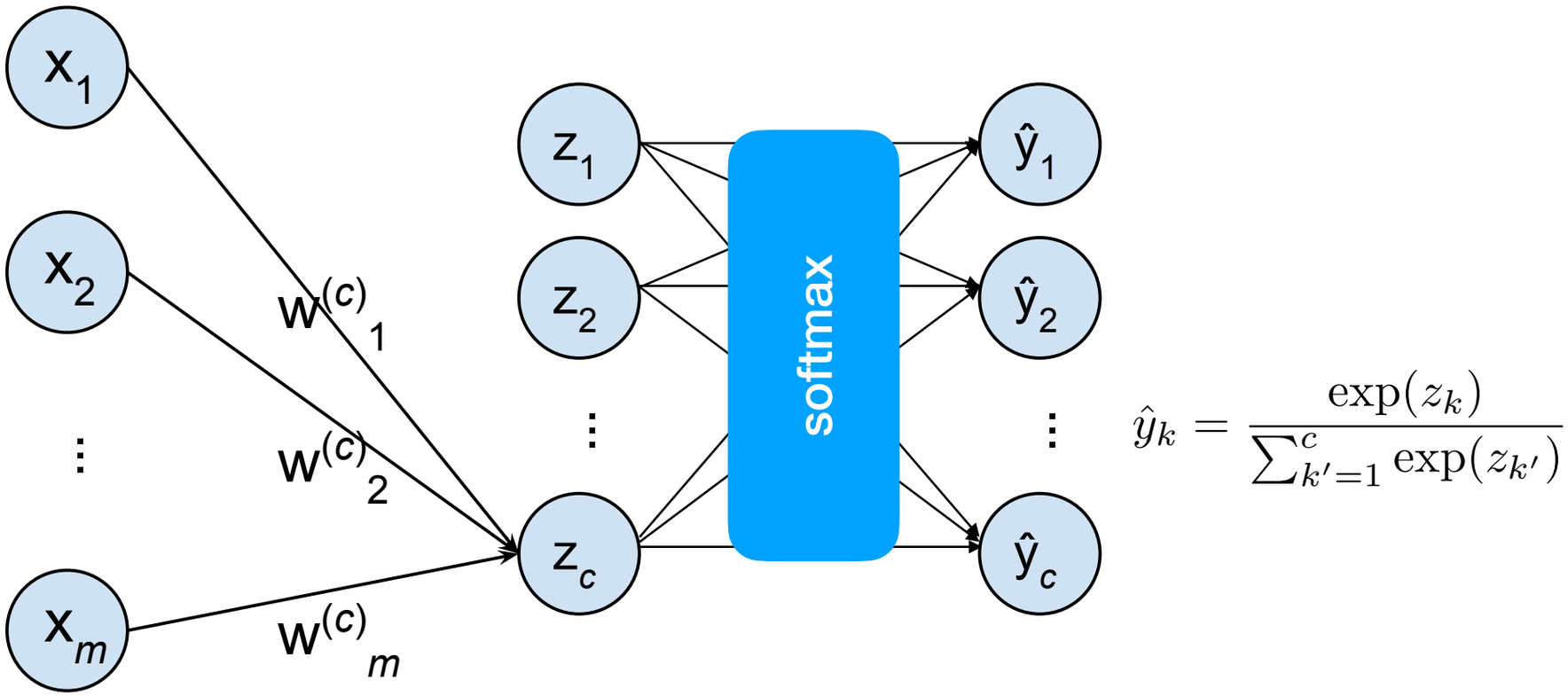
- With softmax regression, we first compute:

$$z_1 = \mathbf{x}^\top \mathbf{w}^{(1)}$$

$$\vdots$$

$$z_c = \mathbf{x}^\top \mathbf{w}^{(c)}$$

Softmax regression diagram



- We then **normalize** across all *c* classes.

$$\hat{y}_k = P(y = k \mid \mathbf{x}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) = \frac{\exp(z_k)}{\sum_{k'=1}^{c} \exp(z_{k'})}$$

Cross-entropy loss

- We need a loss function that can support $c \geq 2$ classes.
- We will use the **cross-entropy** (CE) loss:

$$f_{\mathrm{CE}} = -\sum_{i=1}^{n}\sum_{k=1}^{c} y_k^{(i)} \log \hat{y}_k^{(i)}$$

- Note that the CE loss subsumes the log-loss for $c=2$.

Cross-entropy loss

- The origin of the cross-entropy function is in coding & information theory.

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

$$\begin{aligned}
\text{NLL} &= -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)})
\end{aligned}$$

Conditional independence

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

$$\text{NLL} = -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)})$$

$$= -\log \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)})$$

Our NN's estimate of the probability that x belongs to a particular class.

E.g., if $\mathbf{y}^{(1)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\hat{\mathbf{y}}^{(1)} = \begin{bmatrix} 0.72 \\ 0.28 \end{bmatrix}$

then this probability is 0.72.

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

$$\text{NLL} = -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)})$$

$$= -\log \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)})$$

Our NN's estimate of the probability that x belongs to a particular class.

E.g., if $\mathbf{y}^{(1)} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $\hat{\mathbf{y}}^{(1)} = \begin{bmatrix} 0.72 \\ 0.28 \end{bmatrix}$

then this probability is 0.28.

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

$$\begin{aligned}
\mathrm{NLL} &= -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} \prod_{k=1}^{c} \left(\hat{y}_k^{(i)} \right)^{\left(y_k{}^{(i)} \right)}
\end{aligned}$$

For only one value of *k* is the exponent 1. Otherwise it is 0.

Example: $(0.72)^1(0.28)^0$

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

$$\begin{aligned}
\mathrm{NLL} &= -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} \prod_{k=1}^{c} \left(\hat{y}_k^{(i)} \right)^{\left(y_k{}^{(i)} \right)}
\end{aligned}$$

For only one value of *k* is the exponent 1. Otherwise it is 0.

Example: $(0.72)^0(0.28)^1$

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

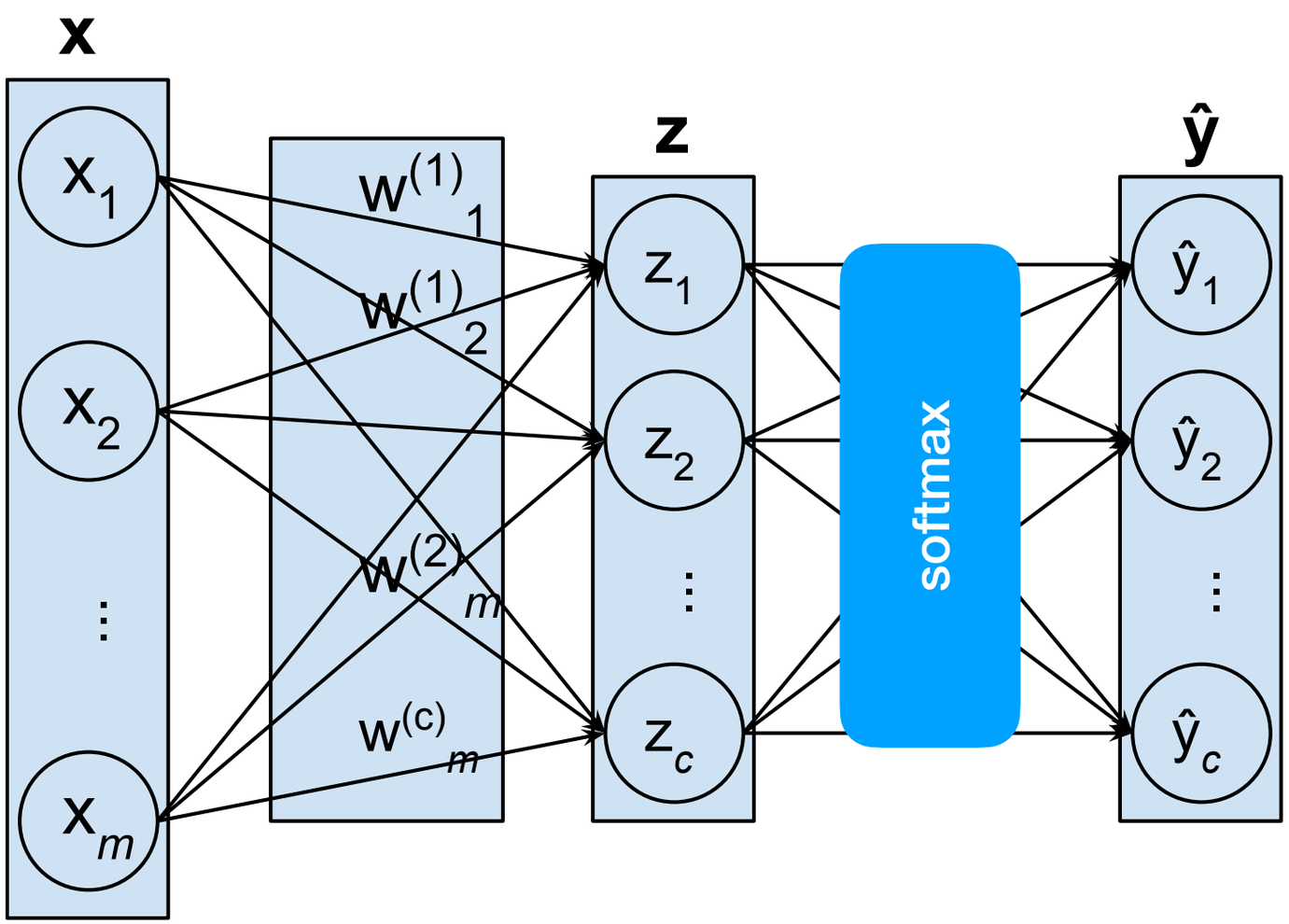
$$\begin{aligned}
\mathrm{NLL} &= -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} \prod_{k=1}^{c} \left(\hat{y}_k^{(i)} \right)^{\left({y_k}^{(i)} \right)} \\
&= -\sum_{i=1}^{n} \sum_{k=1}^{c} y_k^{(i)} \log \hat{y}_k^{(i)}
\end{aligned}$$

Cross-entropy loss

- However, the cross-entropy can also be derived as the **negative log-likelihood** (NLL) of the model predictions:

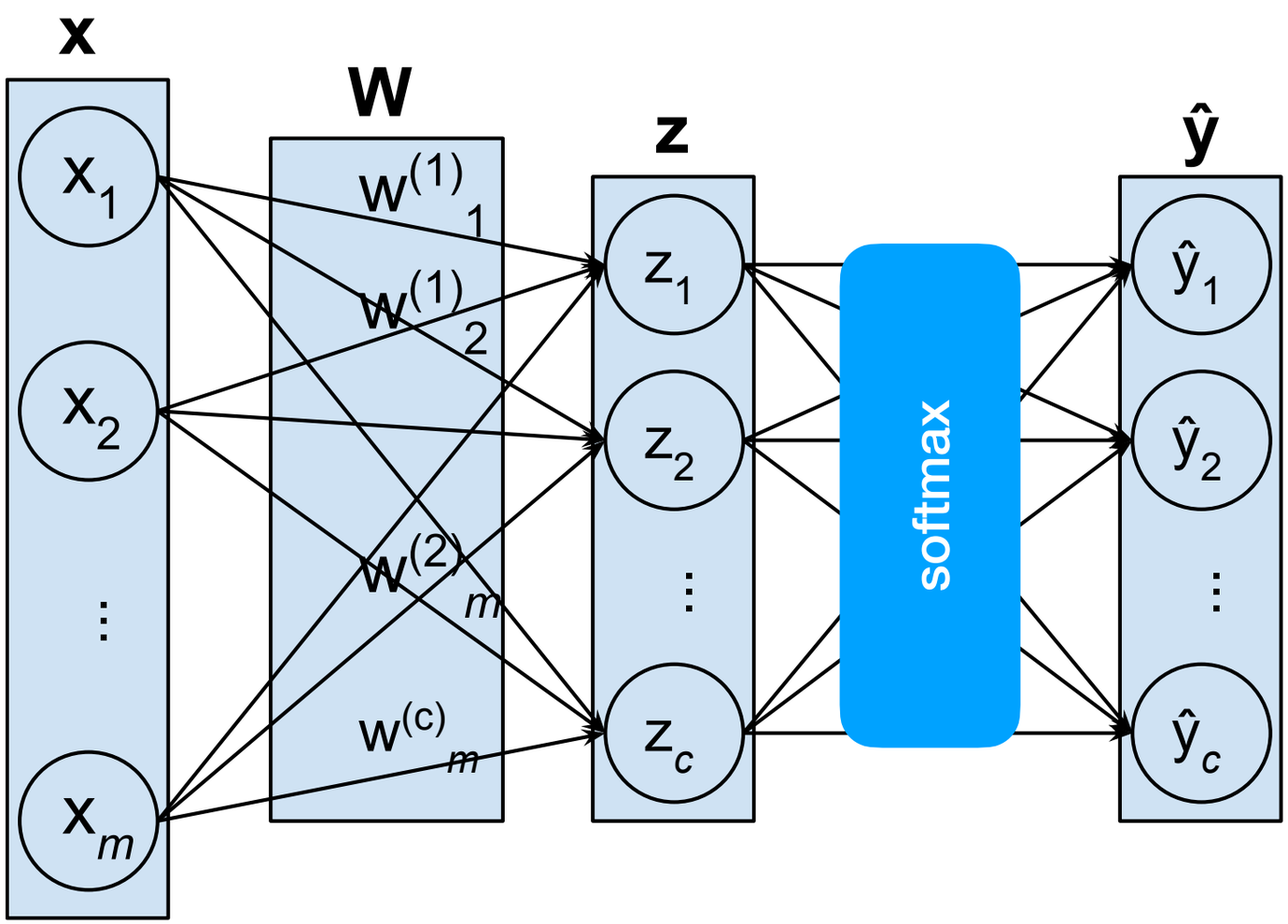
$$
\begin{aligned}
\mathrm{NLL} &= -\log P(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n)} \mid \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(c)}) \\
&= -\log \prod_{i=1}^{n} \prod_{k=1}^{c} \left(\hat{y}_k^{(i)} \right)^{\left({y_k}^{(i)} \right)} \\
&= -\sum_{i=1}^{n} \sum_{k=1}^{c} y_k^{(i)} \log \hat{y}_k^{(i)} \\
&= f_{\mathrm{CE}}
\end{aligned}
$$

Softmax regression: vectorization



- We can represent each **layer** as a vector (**x**, **z**, **ŷ**).

Softmax regression: vectorization



- We can represent the collection of all *c* weight vectors $\mathbf{w}^{(1)}$, …, $\mathbf{w}^{(c)}$ as an (*m* x *c*) matrix **W**.

Softmax regression: vectorization

- By vectorizing, we can compute the pre-activation scores for all *n* examples in one-fell-swoop as:

$$\mathbf{Z} = \mathbf{X}^\top \mathbf{W}$$

Softmax regression: vectorization

- By vectorizing, we can compute the pre-activation scores for all *n* examples in one-fell-swoop as:

$$\mathbf{Z} = \mathbf{X}^\top \mathbf{W}$$

- With numpy, we can call **`np.exp`** to exponentiate every element of **Z**.

- We can then use **`np.sum`** and **`/`** (element-wise division) to compute the softmax.

Gradient descent for softmax regression

- With softmax regression, we need to conduct gradient descent on all *c* of the weights vectors.

- As usual, let's just consider the gradient of the cross-entropy loss for a single example **x**.

- We will compute the gradient w.r.t. each weight vector $\mathbf{w}^{(k)}$ separately (where $k = 1, \ldots, c$).

Gradient descent for softmax regression

- Gradient for each weight vector $\mathbf{w}^{(k)}$:

$$\nabla_{\mathbf{w}^{(k)}} f_{\mathrm{CE}}(\mathbf{y}, \hat{\mathbf{y}}; \mathbf{W}) = \mathbf{x}(\hat{y}_k - y_k)$$

- This is the same expression (for each *k*) as for linear regression and logistic regression!

- We can vectorize this to compute all *c* gradients over all *n* examples…

Gradient descent for softmax regression

- Let $\mathbf{Y}$ and $\hat{\mathbf{Y}}$ both be n x c matrices:

$$\mathbf{Y} = \begin{bmatrix} y_1^{(1)} & \cdots & y_c^{(1)} \\ & \vdots & \\ y_1^{(n)} & \cdots & y_c^{(n)} \end{bmatrix}$$

One-hot encoded vector of class labels for example 1.

Gradient descent for softmax regression

- Let $\mathbf{Y}$ and $\hat{\mathbf{Y}}$ both be n x c matrices:

$$\mathbf{Y} = \begin{bmatrix} y_1^{(1)} & \cdots & y_c^{(1)} \\ & \vdots & \\ y_1^{(n)} & \cdots & y_c^{(n)} \end{bmatrix}$$

One-hot encoded vector of class labels for example *n*.

Gradient descent for softmax regression

- Let $\mathbf{Y}$ and $\hat{\mathbf{Y}}$ both be n x c matrices:

$$\mathbf{Y} = \begin{bmatrix} y_1^{(1)} & \cdots & y_c^{(1)} \\ & \vdots & \\ y_1^{(n)} & \cdots & y_c^{(n)} \end{bmatrix} \qquad \hat{\mathbf{Y}} = \begin{bmatrix} \hat{y}_1^{(1)} & \cdots & \hat{y}_c^{(1)} \\ & \vdots & \\ \hat{y}_1^{(n)} & \cdots & \hat{y}_c^{(n)} \end{bmatrix}$$

The machine's estimates of the *c* class probabilities for example *n*.

Gradient descent for softmax regression

- Let $\mathbf{Y}$ and $\hat{\mathbf{Y}}$ both be n x c matrices:

$$\mathbf{Y} = \begin{bmatrix} y_1^{(1)} & \cdots & y_c^{(1)} \\ & \vdots & \\ y_1^{(n)} & \cdots & y_c^{(n)} \end{bmatrix} \qquad \hat{\mathbf{Y}} = \begin{bmatrix} \hat{y}_1^{(1)} & \cdots & \hat{y}_c^{(1)} \\ & \vdots & \\ \hat{y}_1^{(n)} & \cdots & \hat{y}_c^{(n)} \end{bmatrix}$$

- Then we can compute all c gradient vectors as:

$$\nabla_{\mathbf{W}} f_{\mathrm{CE}}(\mathbf{Y}, \hat{\mathbf{Y}}; \mathbf{W}) = \frac{1}{n}\mathbf{X}(\hat{\mathbf{Y}} - \mathbf{Y})$$

Gradient descent for softmax regression

- Let $\mathbf{Y}$ and $\hat{\mathbf{Y}}$ both be n x c matrices:

$$\mathbf{Y} = \begin{bmatrix} y_1^{(1)} & \cdots & y_c^{(1)} \\ & \vdots & \\ y_1^{(n)} & \cdots & y_c^{(n)} \end{bmatrix} \qquad \hat{\mathbf{Y}} = \begin{bmatrix} \hat{y}_1^{(1)} & \cdots & \hat{y}_c^{(1)} \\ & \vdots & \\ \hat{y}_1^{(n)} & \cdots & \hat{y}_c^{(n)} \end{bmatrix}$$

- Then we can compute all c gradient vectors as:

$$\nabla_{\mathbf{W}} f_{\text{CE}}(\mathbf{Y}, \hat{\mathbf{Y}}; \mathbf{W}) = \frac{1}{n}\mathbf{X}(\hat{\mathbf{Y}} - \mathbf{Y})$$

How far the guesses are from ground-truth.

Gradient descent for softmax regression

- Let $\mathbf{Y}$ and $\hat{\mathbf{Y}}$ both be n x c matrices:

$$\mathbf{Y} = \begin{bmatrix} y_1^{(1)} & \cdots & y_c^{(1)} \\ & \vdots & \\ y_1^{(n)} & \cdots & y_c^{(n)} \end{bmatrix} \qquad \hat{\mathbf{Y}} = \begin{bmatrix} \hat{y}_1^{(1)} & \cdots & \hat{y}_c^{(1)} \\ & \vdots & \\ \hat{y}_1^{(n)} & \cdots & \hat{y}_c^{(n)} \end{bmatrix}$$

- Then we can compute all c gradient vectors as:

$$\nabla_{\mathbf{W}} f_{\text{CE}}(\mathbf{Y}, \hat{\mathbf{Y}}; \mathbf{W}) = \frac{1}{n}\mathbf{X}(\hat{\mathbf{Y}} - \mathbf{Y})$$

The input features (e.g., pixel values).

Bias term

- Like in linear regression, softmax regression also benefits from the use of a bias term.

- Instead of a scalar b, we have a bias vector $\mathbf{b}$ with c dimensions (one for each class).

- You will derive the gradient update for $\mathbf{b}$ as part of homework 3.

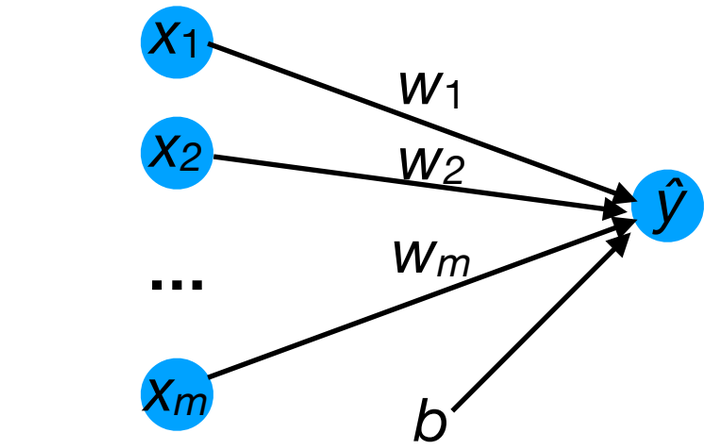
Softmax regression demo

- In HW3, you will apply softmax regression to train a **handwriting recognition system** that can recognize all 10 digits (0-9).
- You will use the popular MNIST dataset consisting of 60K training examples and 10K testing examples:

Review: shallow prediction models

Review

- Linear regression (2-layer NN with linear activation)
  - MSE formulation

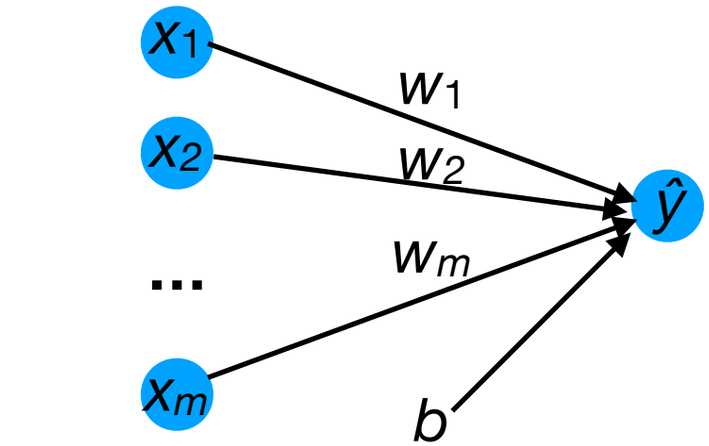


$$\hat{y} = \mathbf{x}^\top \mathbf{w} + b$$

Input layer Output layer

Review

- Linear regression (2-layer NN with linear activation)
  - Probabilistic (MLE) formulation



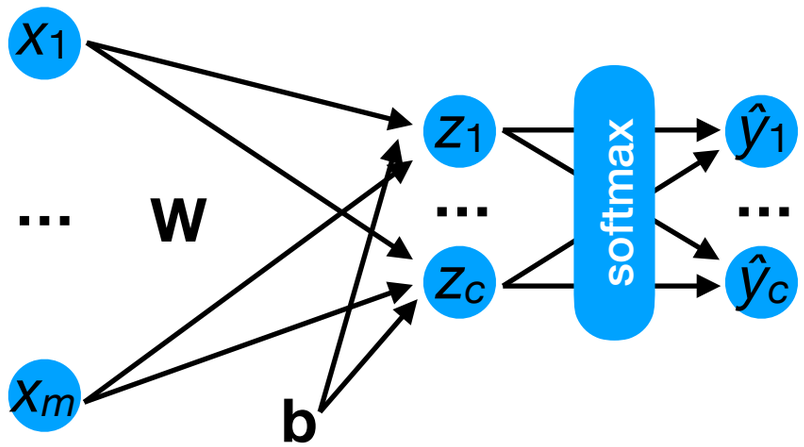
Input layer Output layer

$$P(y \mid \mathbf{x}, \mathbf{w}, b) = \mathcal{N}(y; \mathbf{x}^\top \mathbf{w} + b, \sigma^2)$$

$$\hat{y} = \mathbb{E}[y \mid \mathbf{x}, \mathbf{w}]$$

Review

- Softmax regression (2-layer NN with softmax activation)
  - Probabilistic (MLE) formulation



Input layer Output layer

$$\hat{y}_k = P(y_k \mid \mathbf{x}, \mathbf{W}, \mathbf{b}) = \frac{\exp(z_k)}{\sum_{k'} \exp(z_{k'})}$$

$$z_k = \mathbf{x}^\top \mathbf{w}^{(k)} + b_k$$

Shallow models

- Before diving into deeper models, we will examine one more shallow model.

- Instead of predicting a target value *y* from an input vector **x**, we will instead try to **generate** novel input vectors.

- One way to achieve this is using a latent variable model (LVM).