

# Bazele Inteligenței Artificiale

Prof. Dr. Călin-Adrian POPA

Cursul 1

30 Septembrie 2025

## ● Curs

- 14 cursuri în săptămânilile 1-14
- 1 examen scris în sesiune (cu toate materialele disponibile)
- 0.1 puncte bonus pentru fiecare prezență la curs
- 50% din nota finală

## ● Laborator

- 11 laboratoare
- 2 teste
  - săptămâna 7
  - săptămâna 13
- 50% din nota finală

## ● Site

- <https://cv.upt.ro/>

# Course contents

- ① Introduction. Linear models
- ② Nonparametric models
- ③ Decision trees. Ensemble methods
- ④ Dimensionality reduction
- ⑤ Clustering
- ⑥ Multilayer perceptrons
- ⑦ Convolutional neural networks
- ⑧ Recurrent neural networks
- ⑨ Attention mechanisms

# Laboratory contents

- ① Introduction to Python
- ② Linear models
- ③ Nonparametric models
- ④ Decision trees. Ensemble methods
- ⑤ Dimensionality reduction
- ⑥ Clustering
- ⑦ Test 1
- ⑧ Introduction to PyTorch
- ⑨ Multilayer perceptrons
- ⑩ Convolutional neural networks
- ⑪ Recurrent neural networks
- ⑫ Attention mechanisms
- ⑬ Test 2
- ⑭ Recoveries

# 1 Introduction. Linear models

## 1.1 Artificial intelligence, machine learning, and deep learning

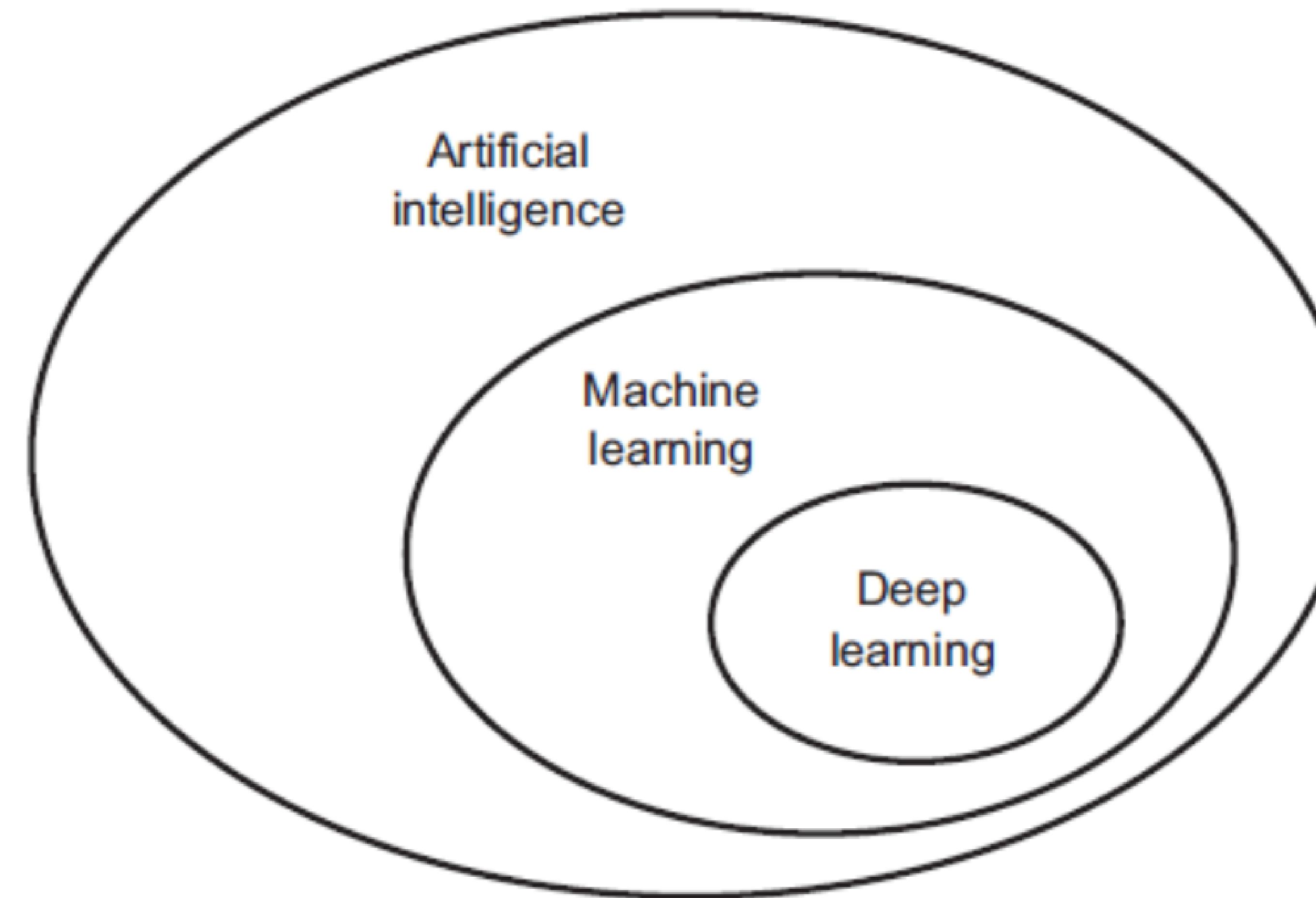


Figure 1: Artificial intelligence, machine learning, and deep learning.

## 1.1.1 Artificial intelligence

- *the effort to automate intellectual tasks normally performed by humans*
- *symbolic AI*: programmers handcraft a sufficiently large set of explicit rules for manipulating knowledge
- peak popularity during the *expert systems* boom of the 1980s
- symbolic AI – suitable to solve well-defined, logical problems, but intractable to figure out explicit rules for solving more complex, fuzzy problems

## 1.1.2 Machine learning

- rather than programmers crafting data processing rules by hand, could a computer automatically learn these rules by looking at data?

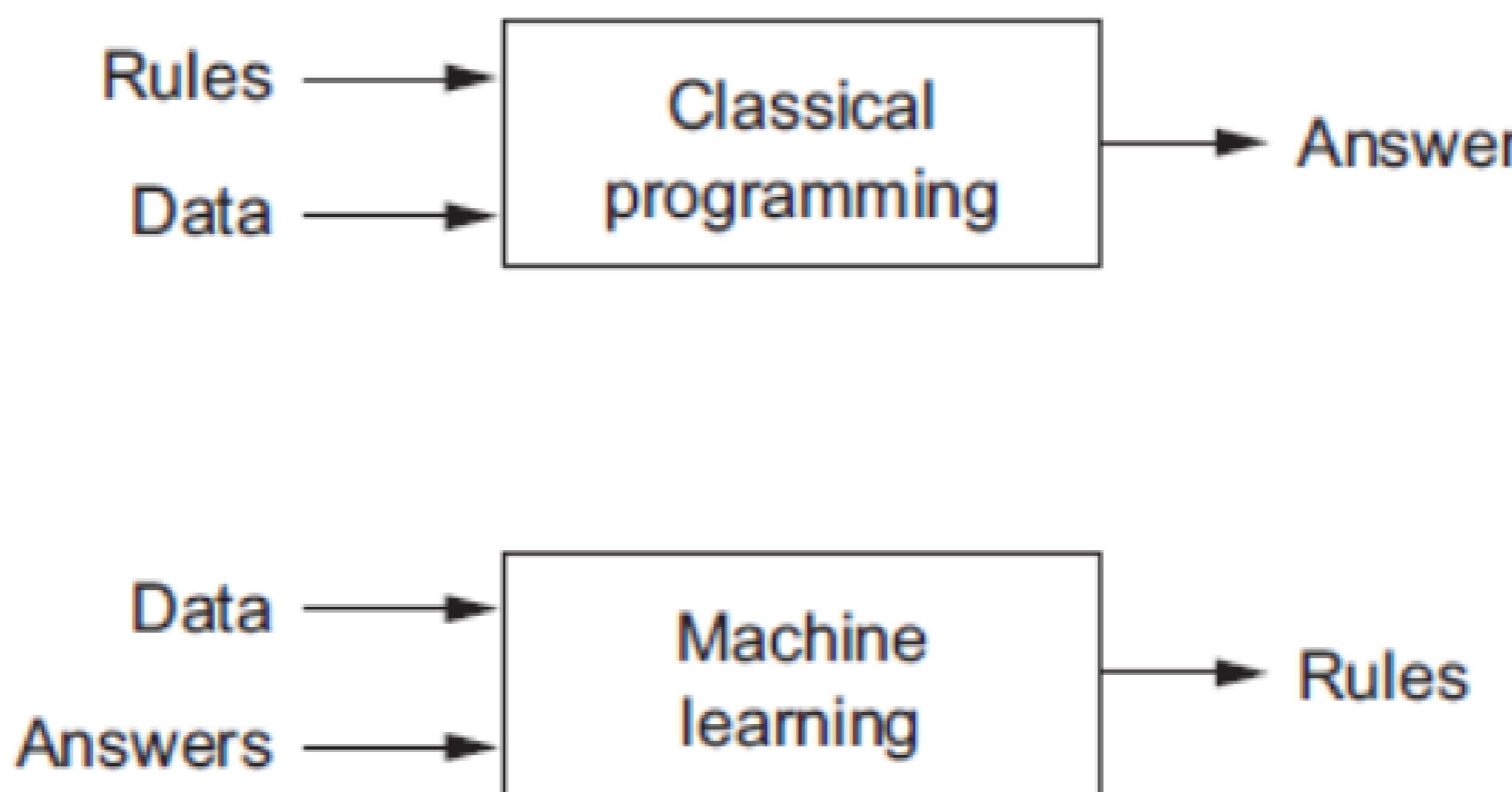


Figure 2: Machine learning: a new programming paradigm.

- a machine learning system is *trained* rather than explicitly programmed
- although machine learning only started to flourish in the 1990s, it has quickly become the most popular and most successful subfield of AI

## 1.1.3 Learning representations from data

- machine learning discovers rules to execute a data processing task, given examples of what's expected
- so, to do machine learning, we need three things:
  - *Input data points*
  - *Examples of the expected output*
  - *A way to measure whether the algorithm is doing a good job*
- the central problem in machine learning and deep learning is to *meaningfully transform data*: in other words, to learn useful *representations* of the input data at hand – representations that get us closer to the expected output
- representation – a different way to look at data – to *represent* or *encode* data

## 1.1.3 Learning representations from data

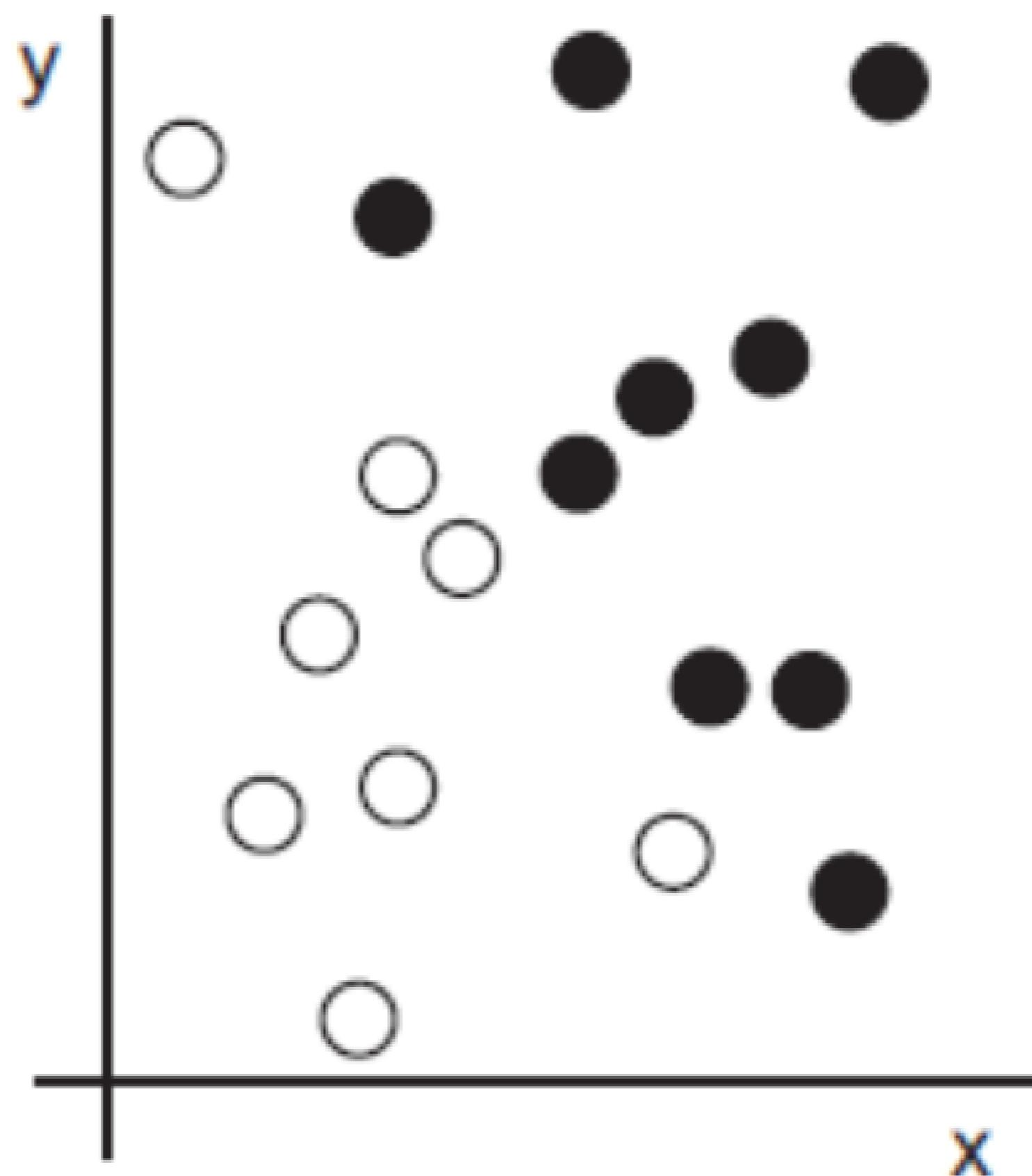


Figure 3: Some sample data.

- inputs?
- expected outputs?
- a way to measure whether the algorithm is doing a good job?

## 1.1.3 Learning representations from data

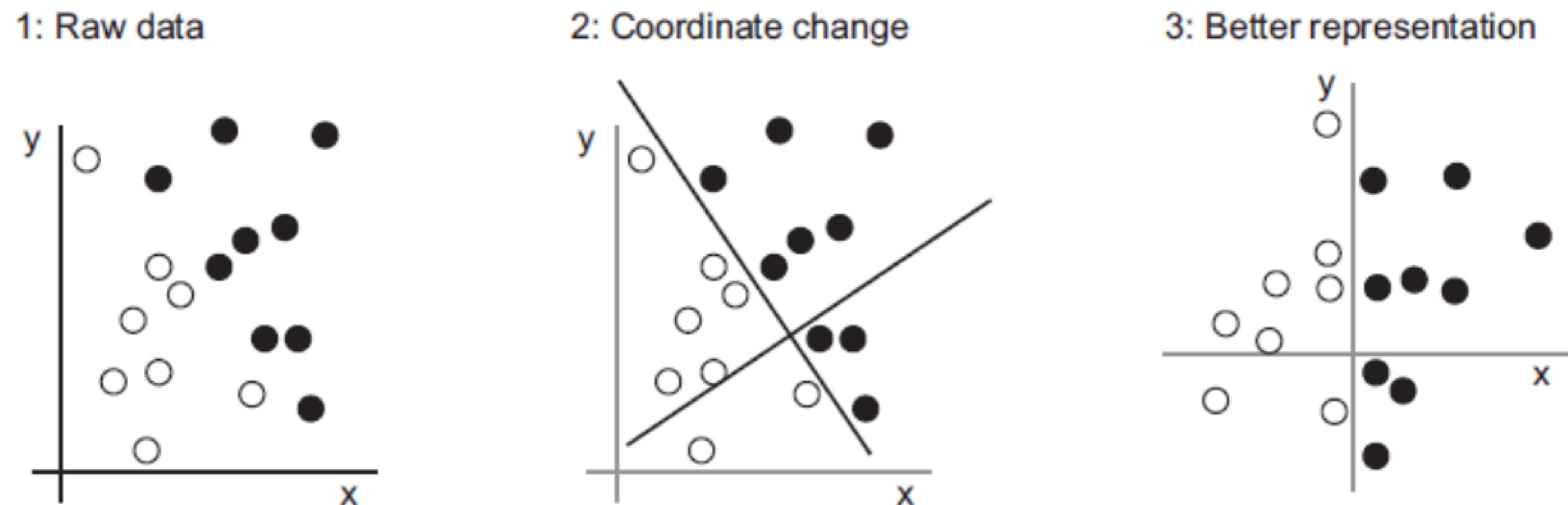


Figure 4: Coordinate change.

- with this representation, the black/white classification problem can be expressed as a simple rule: “Black points are such that  $x > 0$ ”, or “White points are such that  $x < 0$ ”

## 1.1.3 Learning representations from data

- if we tried systematically searching for different possible coordinate changes, and used as feedback the percentage of points being correctly classified – machine learning
- *learning*, in the context of machine learning, describes an automatic search process for better representations
- machine learning algorithms aren't creative in finding transformations; they're merely searching through a predefined set of operations, called a *hypothesis space*
- two main types of problems:
  - *classification*, where the goal is to predict a single discrete label for an input data point
  - *regression*, which consists of predicting a continuous value instead of a discrete label

## 1.1.4 Four branches of machine learning

### Supervised learning

- it consists of learning to map input data to known targets (also called *annotations*), given a set of examples (often annotated by humans)
- generally, almost all applications of machine learning that are in the spotlight these days belong in this category, such as speech recognition, image classification, and language translation
- although supervised learning mostly consists of classification and regression, there are more exotic variants as well, including the following:
  - *Sequence generation*—Given a picture, predict a caption describing it.
  - *Syntax tree prediction*—Given a sentence, predict its decomposition into a syntax tree.
  - *Object detection*—Given a picture, draw a bounding box around certain objects inside the picture. This can also be expressed as a classification problem (given many candidate bounding boxes, classify the contents of each one) or as a joint classification and regression problem, where the bounding box coordinates are predicted via vector regression.
  - *Image segmentation*—Given a picture, draw a pixel-level mask on a specific object.

# Unsupervised learning

- this branch of machine learning consists of finding interesting transformations of the input data without the help of any targets, for the purposes of data visualization, data compression, or data denoising, or to better understand the correlations present in the data at hand
- unsupervised learning is at the core of data analytics, and it's often a necessary step in better understanding a dataset before attempting to solve a supervised learning problem
- *dimensionality reduction* and *clustering* are well-known categories of unsupervised learning

# Self-supervised learning

- this is a specific instance of supervised learning, but it's different enough that it deserves its own category
- self-supervised learning is supervised learning without human-annotated labels
- there are still labels involved (because the learning has to be supervised by something), but they're generated from the input data, typically using a heuristic algorithm
- for instance, *autoencoders* are a well-known instance of self-supervised learning, where the generated targets are the input, unmodified
- in the same way, trying to predict the next frame in a video, given past frames, or the next word in a text, given previous words, are instances of self-supervised learning (*temporally supervised learning*, in this case: supervision comes from future input data)

# Reinforcement learning

- long overlooked, this branch of machine learning recently started to get a lot of attention after Google DeepMind successfully applied it to learning to play Atari games (and, later, learning to play Go at the highest level)
- in reinforcement learning, an *agent* receives information about its environment and learns to choose actions that will maximize some reward
- for instance, a neural network that “looks” at a video game screen and outputs game actions in order to maximize its score can be trained via reinforcement learning
- currently, reinforcement learning is mostly a research area and hasn’t yet had significant practical successes beyond games
- in time, however, we expect to see reinforcement learning take over an increasingly large range of real-world applications: self-driving cars, robotics, resource management, education, and so on

## 1.2 Linear regression

- *linear regression* is a very widely used method for predicting a real-valued output (dependent variable, also called the *label* or *target*)  $y \in \mathbb{R}$ , given a set of real-valued inputs (independent variables, also called *features* or *covariates*)  $x_1, x_2, \dots, x_d$
- we assume that the relationship between the independent variables  $x_1, x_2, \dots, x_d$  and the dependent variable  $y$  is *linear*, i.e., that  $y$  can be expressed as a weighted sum of the features  $x_1, x_2, \dots, x_d$ , given some noise on the observations
- also, we assume that the noise follows a normal distribution
- to develop a model for predicting  $y \in \mathbb{R}$ , we need to obtain a *dataset* consisting of known outputs for corresponding inputs
- the dataset is called a *training dataset* or *training set*, and each row is called an *example* or *data point* or *sample*

## 1.2 Linear regression

- we will use  $n$  to denote the number of examples in our dataset
- we index the data examples by  $i$ , denoting each input as  $x_1^{(i)}, x_2^{(i)}, \dots, x_d^{(i)}$  and the corresponding label as  $y^{(i)}$
- the linear regression model is expressed as:

$$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_d x_d, \quad (1)$$

where

- $\hat{y}$  is the *predicted* value
- $x_j$  is the  $j$ th *feature* value
- $d$  is the number of features
- $w_j$  is the  $j$ th model *weight* (including the *bias* or *intercept* term  $w_0$  and the *feature weights*  $w_1, w_2, \dots, w_d$ )

## 1.2 Linear regression

- strictly speaking, (1) is an *affine transformation* of input features, which is characterized by a *linear transformation* of features via weighted sum, combined with a *translation* via the added bias
- the weights determine the influence of each feature on our prediction and the bias just says what the predicted value should be when all of the features take value 0
- given a dataset, our goal is to choose the weights  $w_1, w_2, \dots, w_d$  and the bias  $w_0$  such that, on average, the predictions  $\hat{y}$  made by our model *best fit* the true labels  $y$  observed in the data
- models whose output prediction is determined by the affine transformation of input features are *linear models*, where the affine transformation is specified by the chosen weights and bias

## 1.2 Linear regression

- the linear regression model can be written more concisely in the vectorized form:

$$\hat{y} = \mathbf{w}^\top \mathbf{x}, \tag{2}$$

where

- $\mathbf{w}$  is the model's *weight vector*, containing the bias or intercept term  $w_0$  and the feature weights  $w_1, w_2, \dots, w_d$
- $\mathbf{x}$  is the instance's *feature vector*, containing  $x_0$  to  $x_d$ , where  $x_0$  is a “dummy feature” always equal to 1, which allows including the bias or intercept term into the weight vector
- in machine learning, vectors are often represented as *column vectors*, which are 2D arrays with a single column
- $\mathbf{w}^\top$  is the *transpose* of  $\mathbf{w}$  (a row vector instead of a column vector) and  $\mathbf{w}^\top \mathbf{x}$  is the matrix multiplication of  $\mathbf{w}^\top$  and  $\mathbf{x}$ , which represents the scalar product of  $\mathbf{w}$  and  $\mathbf{x}$
- because the scalar product is symmetric, (2) can also be written as:

$$\hat{y} = \mathbf{x}^\top \mathbf{w}. \tag{3}$$

## 1.2 Linear regression

- in (3), the vector  $\mathbf{x}$  corresponds to features of a single data example
- we will often find it convenient to refer to features of our entire dataset of  $n$  examples via the *design matrix*  $\mathbf{X} \in \mathbb{R}^{n \times (d+1)}$
- here,  $\mathbf{X}$  contains one row for every example and one column for every feature:

$$\mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_d^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_d^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(n)} & x_2^{(n)} & \dots & x_d^{(n)} \end{bmatrix} = \begin{bmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \vdots \\ \mathbf{x}^{(n)\top} \end{bmatrix}.$$

- for a collection of features  $\mathbf{X}$ , the predictions  $\hat{\mathbf{y}} \in \mathbb{R}^n$  can be expressed via the matrix-vector product:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}, \tag{4}$$

where  $\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(n)} \end{bmatrix}$

## 1.2 Linear regression

- given features of a training dataset  $\mathbf{X}$  and corresponding (known) labels  $\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}$ , the goal of linear regression is to find the weight vector  $\mathbf{w}$  that given features of a new data example sampled from the same distribution as  $\mathbf{X}$ , the new example's label will (in expectation) be predicted with the lowest error
- even if we believe that the best model for predicting  $y$  given  $\mathbf{x}$  is linear, we would not expect to find a real-world dataset of  $n$  examples where  $y^{(i)}$  exactly equals  $\mathbf{w}^\top \mathbf{x}^{(i)}$ , for all  $1 \leq i \leq n$
- for example, whatever instruments we use to observe the features  $\mathbf{X}$  and labels  $\mathbf{y}$  might suffer small amount of measurement error
- thus, even when we are confident that the underlying relationship is linear, we will incorporate a *noise term* to account for such errors

## 1.2 Linear regression

- to refresh our memory, the probability density of a *normal distribution*  $\mathcal{N}(\mu, \sigma^2)$  with *mean*  $\mu$  and *variance*  $\sigma^2$  (standard deviation  $\sigma$ ) is given as:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right).$$

- we assume that observations arise from noisy observations, where the noise is normally distributed as follows:

$$y = \mathbf{w}^\top \mathbf{x} + \epsilon,$$

where  $\epsilon \sim \mathcal{N}(0, \sigma^2)$

- thus, we can now write the *likelihood* of seeing a particular  $y^{(i)}$  for a given  $\mathbf{x}^{(i)}$  as:

$$p(y^{(i)} | \mathbf{x}^{(i)}, \mathbf{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)})^2\right).$$

## 1.2 Linear regression

- now, according to the principle of *maximum likelihood*, the best values of weights  $\mathbf{w}$  are those that maximize the likelihood of the entire dataset:

$$p(\mathbf{y}|\mathbf{X}, \mathbf{w}) = \prod_{i=1}^n p(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}). \quad (5)$$

- estimators chosen according to the principle of maximum likelihood are called *maximum likelihood estimators*
- while maximizing the product of many exponential functions might look difficult, we can simplify things significantly, without changing the objective, by maximizing the log of the likelihood instead
- for historical reasons, optimizations are more often expressed as minimization rather than maximization
- so, without changing anything, we can minimize the *negative log-likelihood* –  $-\log p(\mathbf{y}|\mathbf{X}, \mathbf{w})$ ; from (5), we have that:

$$-\log p(\mathbf{y}|\mathbf{X}, \mathbf{w}) = \sum_{i=1}^n \left[ \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} (y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)})^2 \right].$$

## 1.2 Linear regression

- now we just need one more assumption that  $\sigma$  is some fixed constant; thus we can ignore the first term because it does not depend on  $\mathbf{w}$
- because the solution does not depend on  $\sigma$ , we can also ignore the multiplicative constant  $\frac{1}{\sigma^2}$
- as a consequence, negative log-likelihood is given as:

$$\sum_{i=1}^n \frac{1}{2} (y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)})^2.$$

- because  $\hat{y}^{(i)} = \mathbf{w}^\top \mathbf{x}^{(i)}$  are the predictions of the linear regression model, according to (4), the negative log-likelihood finally becomes:

$$\sum_{i=1}^n \frac{1}{2} (y^{(i)} - \hat{y}^{(i)})^2.$$

## 1.2 Linear regression

- on the other hand, before we start searching for the best weights (or model weights)  $\mathbf{w}$ , we will need two more things:
  - a quality measure for some given model
  - a procedure for updating the model to improve its quality
- in general, the quality measure for a certain model is given by the *loss function*, which quantifies the distance between the *real* and *predicted* value of the target
- the loss will usually be a non-negative number, where smaller values are better and perfect predictions have a loss of 0
- the most popular loss function in regression problems is the *squared error*
- when the model prediction for an example  $i$  is  $\hat{y}^{(i)}$  and the corresponding true label is  $y^{(i)}$ , the squared error is given by:

$$l^{(i)}(\mathbf{w}) = \frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2.$$

- the constant  $\frac{1}{2}$  makes no real difference, but will prove notationally convenient, canceling out when we take the derivative of the loss

## 1.2 Linear regression

- since the training dataset is given to us, and thus out of our control, the empirical error is only a function of the model weights
- to make things more concrete, consider the example below, where we plot a regression problem for a one-dimensional case, as shown in Figure 5

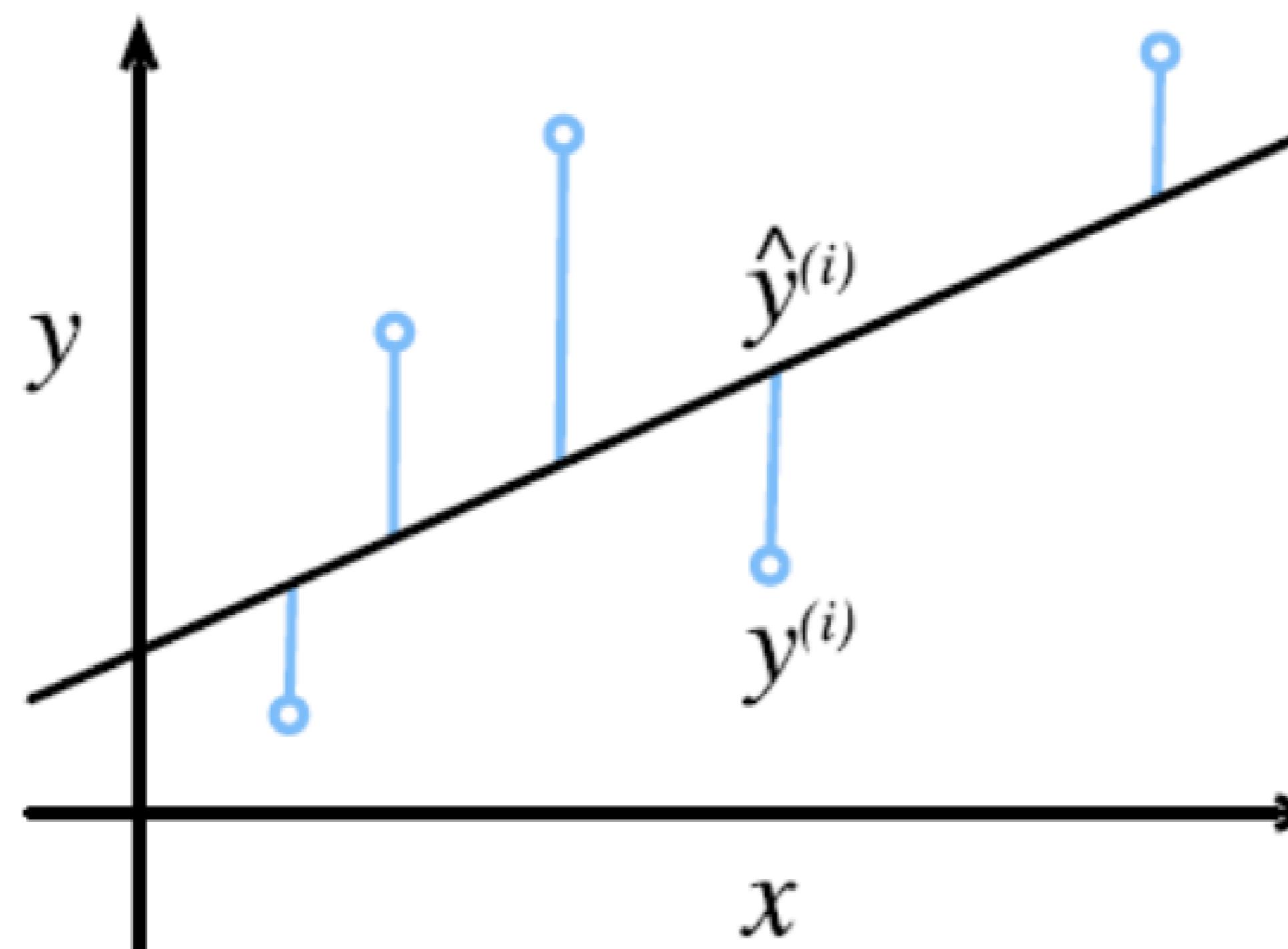


Figure 5: Linear regression with  $d = 1$ .

## 1.2 Linear regression

- note that large differences between estimates  $\hat{y}^{(i)}$  and observations  $y^{(i)}$  lead to even larger contributions to the loss, due to the quadratic dependence
- to measure the quality of a model on the entire dataset of  $n$  examples, we simply average (or equivalently, sum) the losses on the training set:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n l^{(i)}(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} (\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} (\mathbf{w}^\top \mathbf{x}^{(i)} - y^{(i)})^2. \quad (6)$$

- this is called the *mean squared error loss function*; using (4), the mean squared error can be written as:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2n} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}).$$

- we can now see that, in the case of linear regression, the negative log-likelihood is *equal* (up to multiplicative constants) to the mean squared error loss function
- when training the model, we want to find parameters  $\mathbf{w}$  that minimize the total loss across all training examples:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \mathcal{L}(\mathbf{w}).$$

## 1.2 Linear regression

- recall from first year calculus that, in order to find the minimum of the function  $\mathcal{L}(\mathbf{w})$ , we must compute its gradient, equal it with  $\mathbf{0}^\top$ , and extract the value of the minimum:

$$\begin{aligned}\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) &= \nabla_{\mathbf{w}} \left( \frac{1}{2n} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) \right) \\ &= \frac{1}{2n} \nabla_{\mathbf{w}} \left( \mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} - 2\mathbf{y}^\top \mathbf{X}\mathbf{w} + \mathbf{y}^\top \mathbf{y} \right) \\ &= \frac{1}{n} (\mathbf{X}^\top \mathbf{X}\mathbf{w} - \mathbf{X}^\top \mathbf{y}).\end{aligned}\tag{7}$$

- from  $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \mathbf{0}^\top$ , we obtain:

$$\begin{aligned}\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \mathbf{0}^\top &\Leftrightarrow \mathbf{X}^\top \mathbf{X}\mathbf{w} - \mathbf{X}^\top \mathbf{y} = \mathbf{0}^\top \\ &\Leftrightarrow \mathbf{X}^\top \mathbf{X}\mathbf{w} = \mathbf{X}^\top \mathbf{y} \\ &\Leftrightarrow \mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.\end{aligned}$$

- thus, for linear regression, we have the analytic (closed-form) solution:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

## 1.2 Linear regression

- this is called the *normal equation*
- the matrix  $\mathbf{X}^+ := (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$  is called the *pseudoinverse* or *Moore-Penrose inverse* of matrix  $\mathbf{X}$
- the pseudoinverse itself is computed using a standard matrix factorization technique called *singular value decomposition* (SVD) that can decompose the training set matrix  $\mathbf{X}$  into the matrix multiplication of three matrices:  $\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^\top$
- the pseudoinverse is computed as  $\mathbf{X}^+ = \mathbf{V}\mathbf{S}^+\mathbf{U}^\top$ ; to compute the matrix  $\mathbf{S}^+$ , the algorithm takes  $\mathbf{S}$  and sets to zero all values smaller than a tiny threshold value, then it replaces all the nonzero values with their inverse, and finally it transposes the resulting matrix
- this approach is more efficient than computing the normal equation, plus it handles edge cases nicely: indeed, the normal equation may not work if the matrix  $\mathbf{X}^\top \mathbf{X}$  is not invertible (i.e., singular), such as if  $n < d$  or if some features are redundant, but the pseudoinverse is always defined

## 1.2 Linear regression

- the normal equation computes the inverse of  $\mathbf{X}^\top \mathbf{X}$ , which is an  $(d + 1) \times (d + 1)$  matrix
- the computational complexity of inverting such a matrix is typically about  $\mathcal{O}(d^{2.4})$  to  $\mathcal{O}(d^3)$ , depending on the implementation
- the SVD approach is about  $\mathcal{O}(d^2)$
- both the normal equation and the SVD approach get very slow when the number of features grows large (e.g., 100,000)
- on the positive side, both are linear with regard to the number of instances in the training set (they are  $\mathcal{O}(n)$ ), so they handle large training sets efficiently, provided they can fit in memory

## 1.2 Linear regression

- also, once we have trained our linear regression model (using the normal equation or any other algorithm), predictions are very fast: the computational complexity is *linear* with regard to both the number of instances we want to make predictions on and the number of features
- in other words, making predictions on twice as many instances (or twice as many features) will take roughly twice as much time
- now we will look at a very different way to train a linear regression model, which is better suited for cases where there are a large number of features or too many training instances to fit in memory

## 1.3 Gradient descent

- *gradient descent* is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems
- the general idea of gradient descent is to update the parameters iteratively in order to minimize the loss function
- suppose we are lost in the mountains in a dense fog, and we can only feel the slope of the ground below our feet; a good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the *steepest slope*
- this is exactly what gradient descent does: it measures the local gradient of the loss function with respect to the parameter vector  $w$ , and it goes in the direction of the steepest descent, which is given by the *negative gradient vector*, because the gradient points uphill
- once the gradient is zero, a minimum of the loss function has been reached

# 1.3 Gradient descent

- the steps of the algorithm are the following:
  - initialize the values of the model weights, typically at random (this is called *random initialization*).
  - iteratively update the parameters one small step at a time, in the direction of the negative gradient, until the algorithm *converges* to a minimum (see Figure 6). This is done by multiplying the gradient by a predetermined positive value  $\eta$  (called *learning rate*) and subtracting the resulting term from the current parameter values.

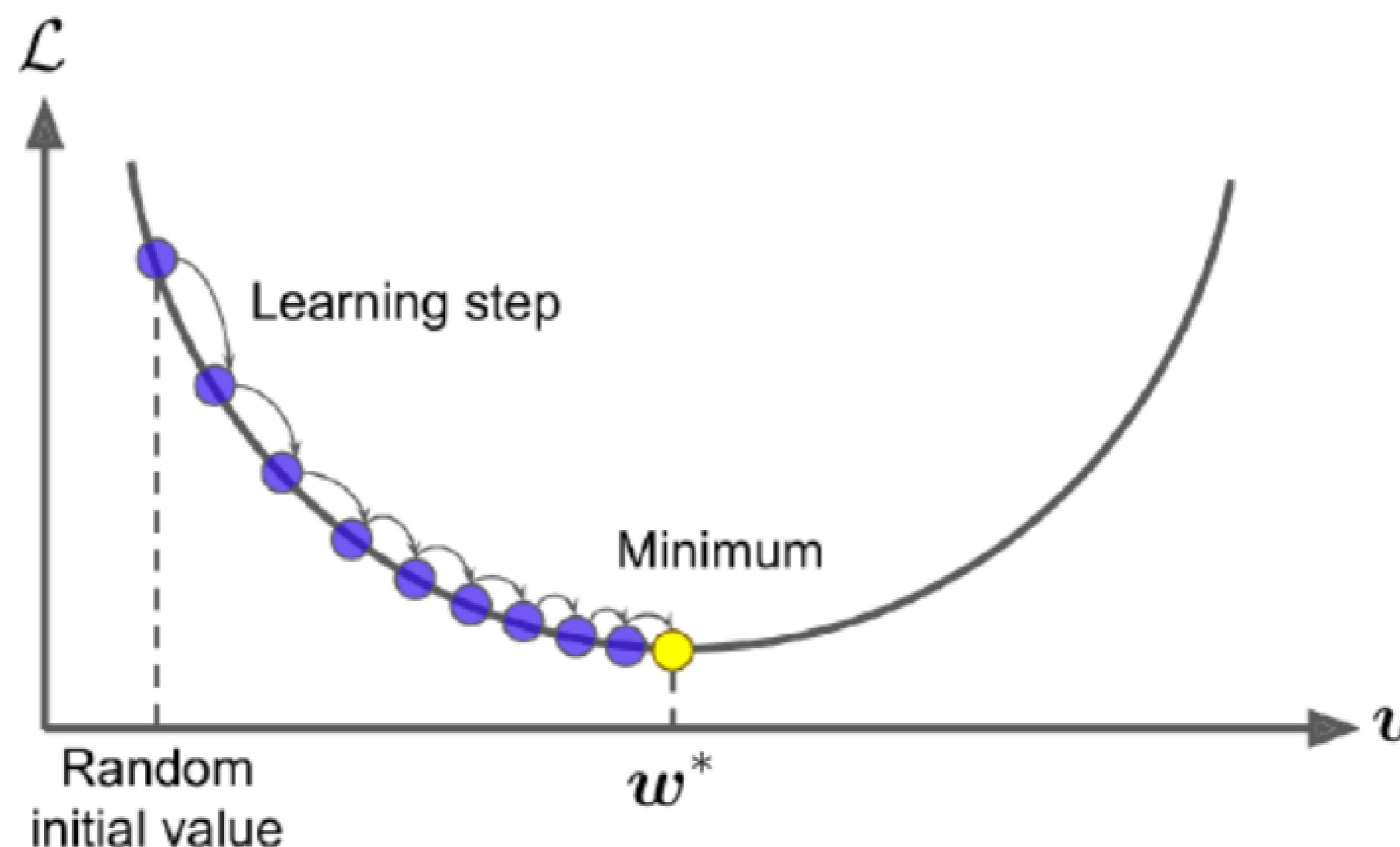


Figure 6: Gradient descent.

## 1.3 Gradient descent

- we can express the update mathematically as follows:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}),$$

or, equivalently, as

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \left( \frac{1}{n} \sum_{i=1}^n l^{(i)}(\mathbf{w}) \right),$$

where we used (6)

- according to (7), we can compute the gradient as follows:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{1}{n} (\mathbf{X}^\top \mathbf{X} \mathbf{w} - \mathbf{X}^\top \mathbf{y}) = \frac{1}{n} \mathbf{X}^\top (\mathbf{X} \mathbf{w} - \mathbf{y}). \quad (8)$$

## 1.3 Gradient descent

- notice that this formula involves calculations over the full training set  $X$ , at each gradient descent step
- this is why the algorithm is called *batch gradient descent*: it uses the whole batch of training data at every step
- as a result it is very slow on very large training sets (but we will see much faster gradient descent algorithms shortly)
- however, gradient descent scales well with the number of features; training a linear regression model when there are hundreds of thousands of features is *much faster* using gradient descent than using the normal equation or SVD decomposition

## 1.3 Gradient descent

- an important parameter in gradient descent is the size of the steps, determined by the learning rate  $\eta$
- the value of the learning rate is manually pre-specified and not typically learned through model training
- this type of parameters that are tunable but not updated in the training loop are called *hyperparameters*
- *hyperparameter tuning* is the process by which hyperparameters are chosen, and typically requires that we adjust them based on the results of the training loop as assessed on a separate *validation dataset* (or *validation set*)

## 1.3 Gradient descent

- if the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (see Figure 7)

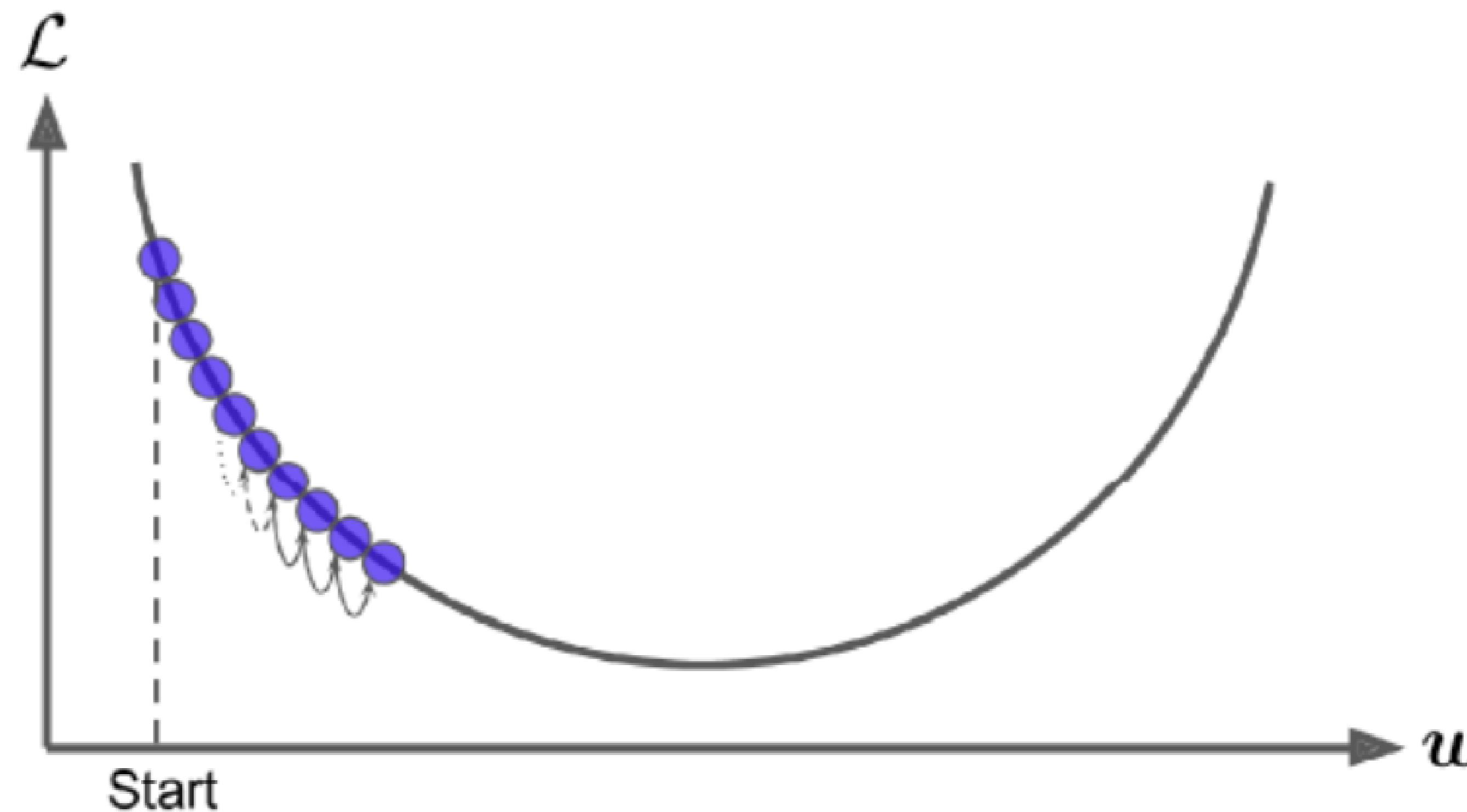


Figure 7: The learning rate is too small.

## 1.3 Gradient descent

- on the other hand, if the learning rate is too high, we might jump across the valley and end up on the other side, possibly even higher up than we were before
- this might make the algorithm diverge, with larger and larger values, failing to find a good solution (see Figure 8)

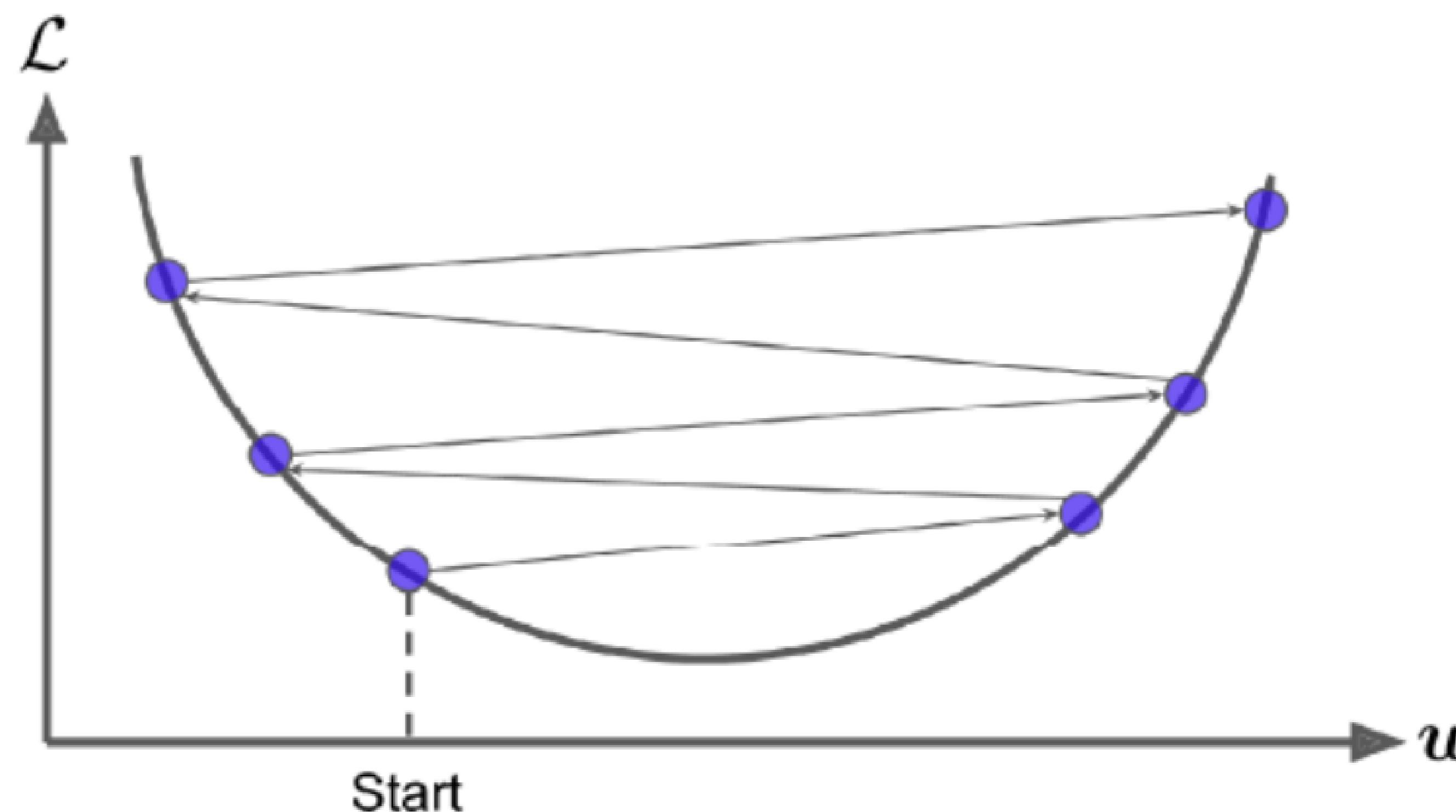


Figure 8: The learning rate is too big.

## 1.3 Gradient descent

- finally, not all loss functions look like nice, regular bowls
- there may be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult
- Figure 9 shows the two main challenges with gradient descent
- if the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*
- if it starts on the right, then it will take a very long time to cross the plateau
- and if we stop too early, we will never reach the global minimum

## 1.3 Gradient descent

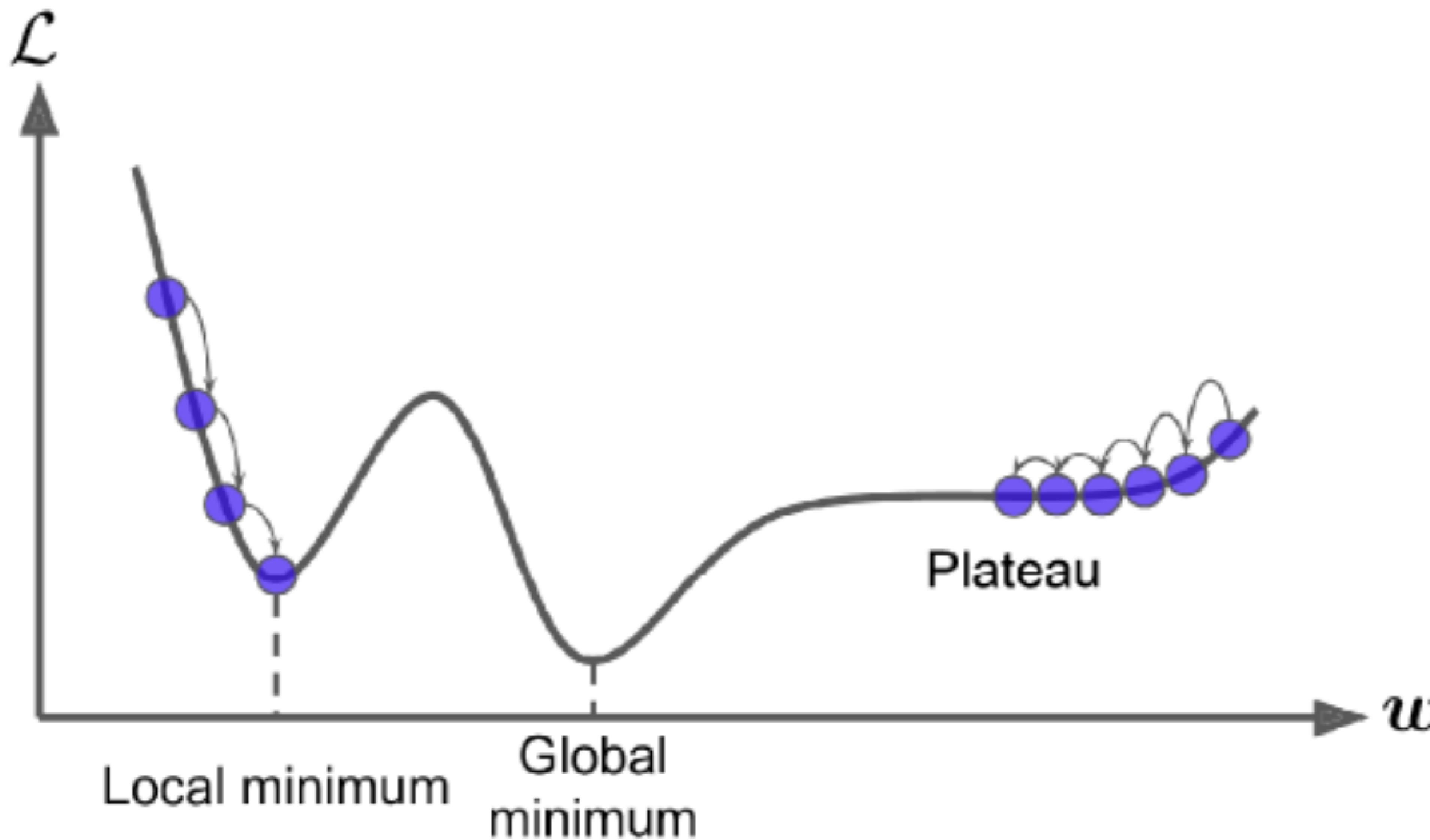


Figure 9: Gradient descent challenges.

## 1.3 Gradient descent

- fortunately, the mean squared error loss function for a linear regression model happens to be a *convex function*, which means that if we pick any two points on the curve, the line segment joining them never crosses the curve
- this implies that there are no local minima, just one global minimum
- it is also a continuous function with a slope that never changes abruptly
- these two facts have a great consequence: gradient descent is guaranteed to approach arbitrarily close the global minimum (if we wait long enough and if the learning rate is not too high)

## 1.3 Gradient descent

- in fact, the loss function has the shape of a bowl, but it can be an elongated bowl if the features have very *different scales*
- Figure 10 shows gradient descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right)

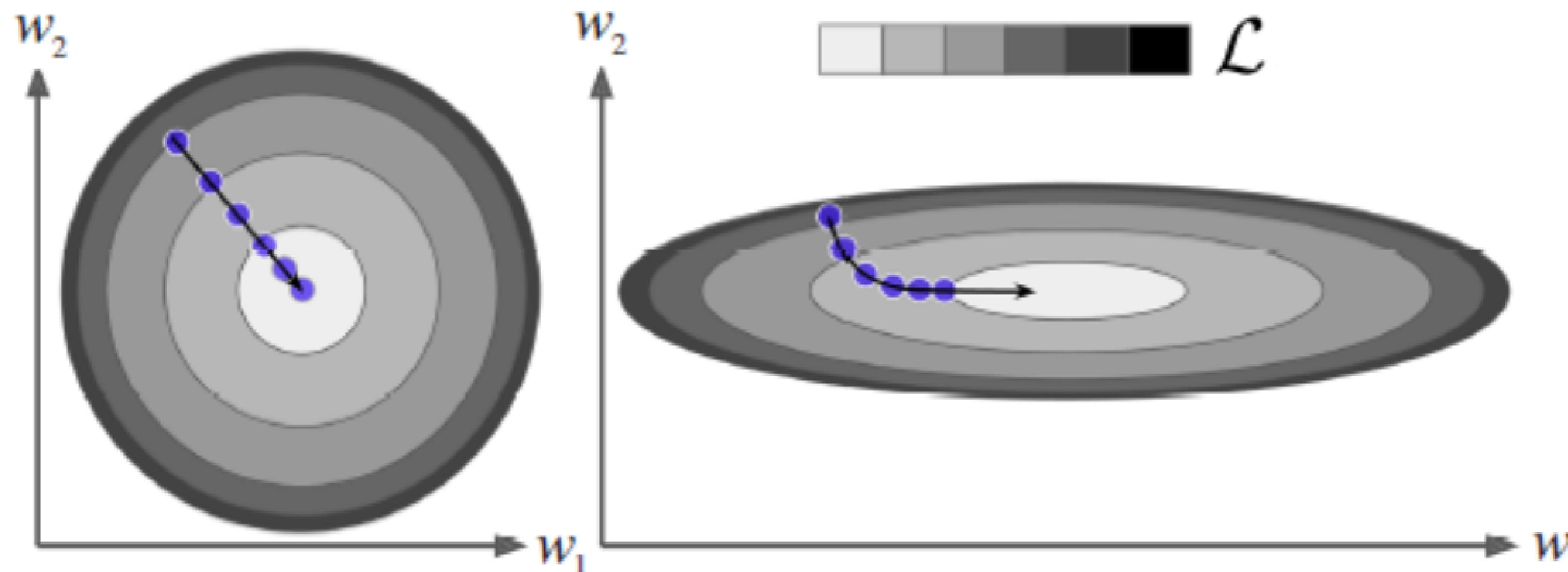


Figure 10: Gradient descent with (left) and without (right) feature scaling.

## 1.3 Gradient descent

- as we can see, on the left the gradient descent algorithm goes straight toward the minimum, thereby reaching it quickly, whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley
- it will eventually reach the minimum, but it will take a long time
- when using gradient descent, we should ensure that all features have a *similar scale*, or else it will take much longer to converge

## 1.3 Gradient descent

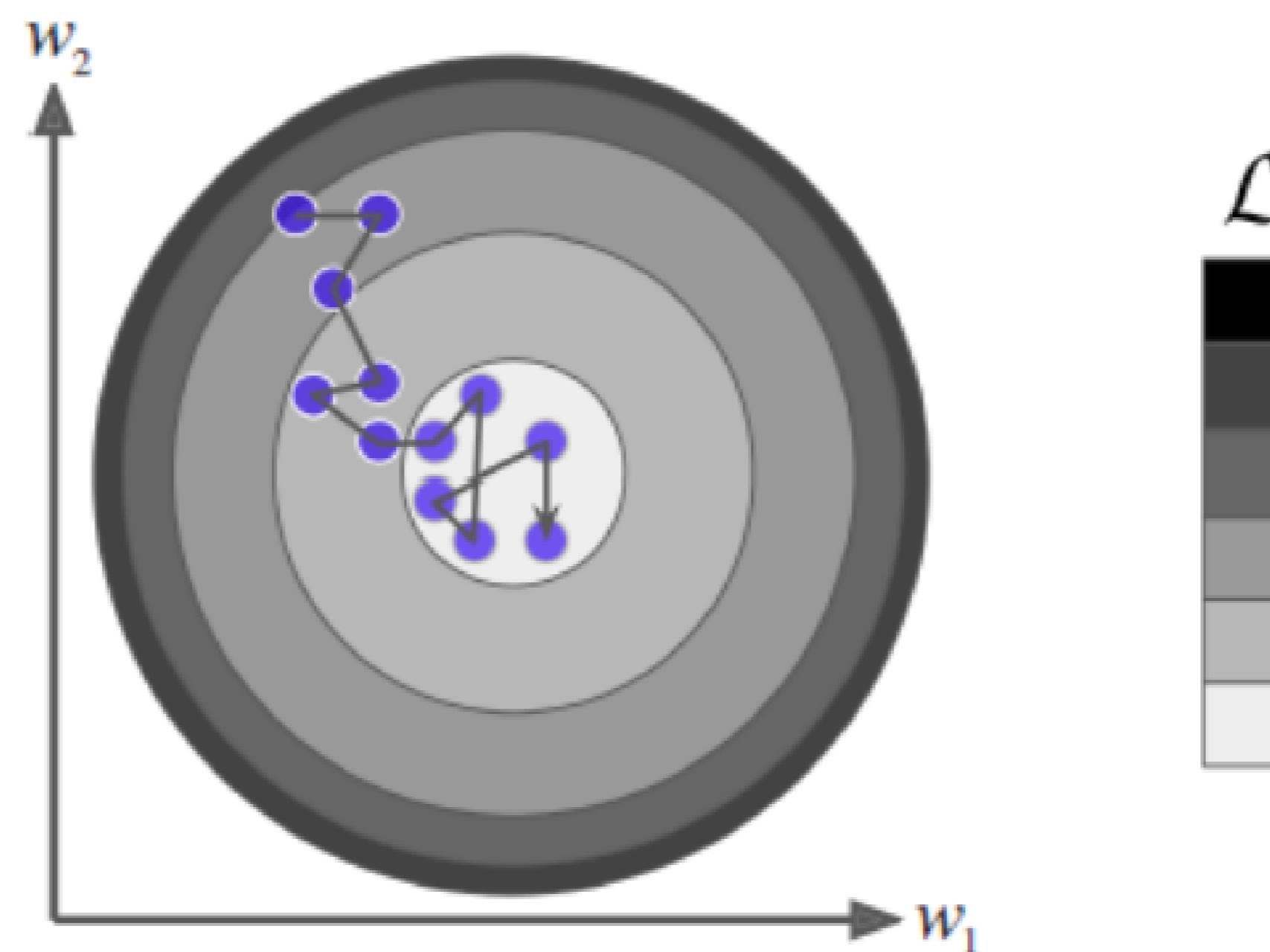
- the main problem with batch gradient descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large
- at the opposite extreme, *stochastic gradient descent* picks a random instance in the training set at every step and computes the gradients based only on that single instance
- obviously, working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration
- it also makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration
- mathematically, this can be expressed as:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} l^{(i)}(\mathbf{w}),$$

for some  $1 \leq i \leq n$

# 1.3 Gradient descent

- on the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than batch gradient descent: instead of gently decreasing until it reaches the minimum, the loss function will *bounce* up and down, decreasing only on average
- over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down (see Figure 11)
- so once the algorithm stops, the final parameter values are good, but not optimal



**Figure 11:** With stochastic gradient descent, each training step is much faster but also much more stochastic than when using batch gradient descent.

## 1.3 Gradient descent

- when the loss function is very irregular (as in Figure 9), this can actually help the algorithm jump out of local minima, so stochastic gradient descent has a better chance of finding the global minimum than batch gradient descent does
- therefore, randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum
- one solution to this dilemma is to gradually reduce the learning rate
- the steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum
- the function that determines the learning rate at each iteration is called the *learning schedule*
- if the learning rate is reduced too quickly, we may get stuck in a local minimum, or even end up frozen halfway to the minimum
- if the learning rate is reduced too slowly, we may jump around the minimum for a long time and end up with a suboptimal solution if we halt training too early

## 1.3 Gradient descent

- by convention we iterate by rounds of  $n$  iterations; each round is called an *epoch*
- thus, in one epoch, each example will be used to update the weights of the model
- if we want to be sure that the algorithm goes through every instance at each epoch, we must shuffle the training set
- the last gradient descent algorithm we will look at is called *mini-batch gradient descent*
- it is simple to understand once we know batch and stochastic gradient descent: at each step, instead of computing the gradients based on the full training set (as in batch GD) or based on just one instance (as in stochastic GD), mini-batch GD computes the gradients on small random sets of instances called *mini-batches*

## 1.3 Gradient descent

- the algorithm's progress in parameter space is less erratic than with stochastic GD, especially with fairly large mini-batches
- as a result, mini-batch GD will end up walking around a bit closer to the minimum than stochastic GD – but it may be harder for it to escape from local minima (in the case of problems that suffer from local minima, unlike linear regression)
- Figure 12 shows the paths taken by the three gradient descent algorithms in parameter space during training
- they all end up near the minimum, but batch GD's path actually stops at the minimum, while both stochastic GD and mini-batch GD continue to walk around
- however, we must not forget that batch GD takes a lot of time to take each step, and stochastic GD and mini-batch GD would also reach the minimum if we used a good learning schedule

## 1.3 Gradient descent

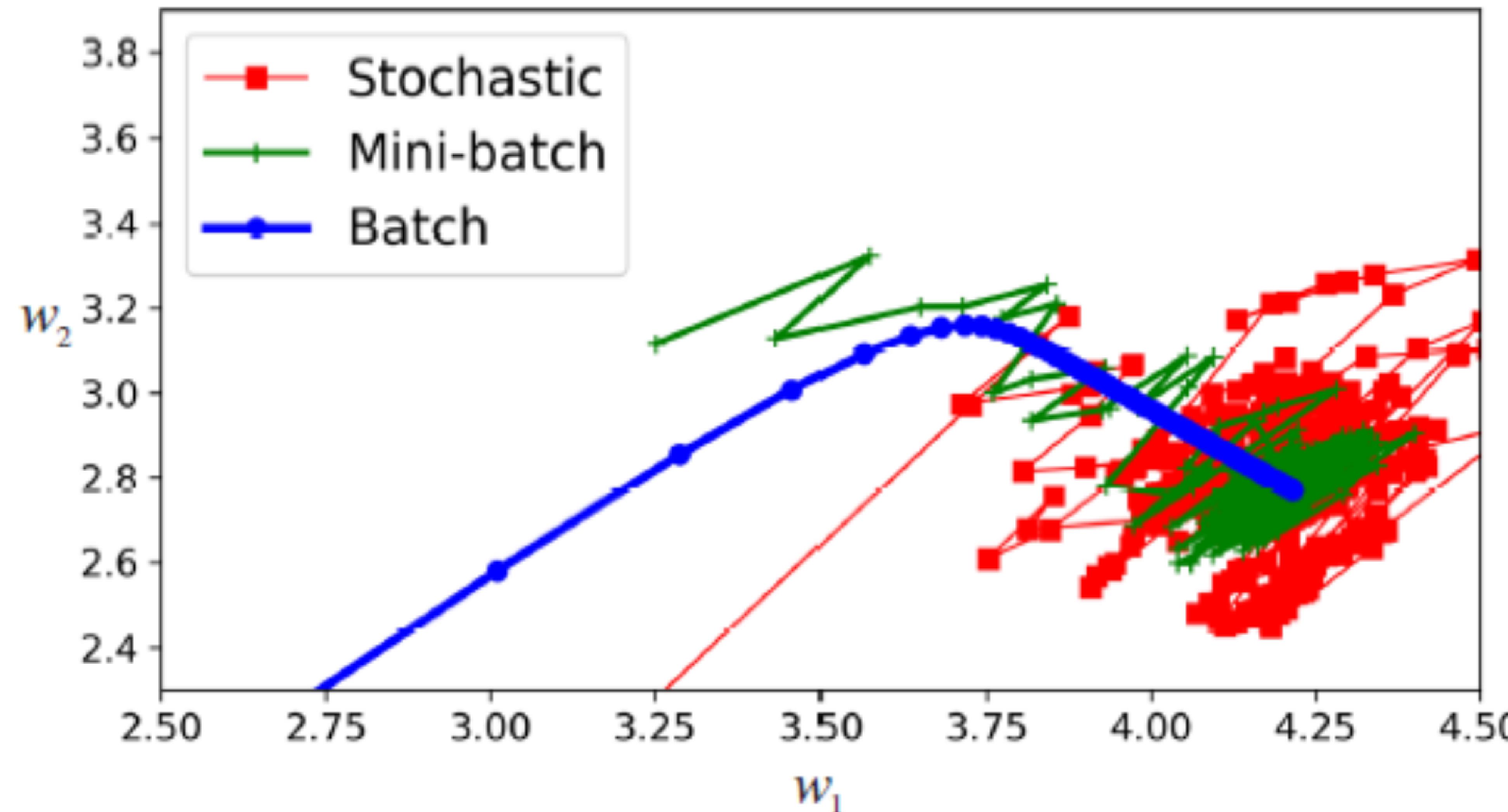


Figure 12: With stochastic gradient descent, each training step is much faster but also much more stochastic than when using batch gradient descent.

## 1.3 Gradient descent

- in each iteration, we first randomly sample a mini-batch  $\mathcal{B}$  consisting of a fixed number of training examples
- we then compute the derivative (gradient) of the average loss on the mini-batch with regard to the model parameters
- finally, we multiply the gradient by a predetermined positive value and subtract the resulting term from the current parameter values
- we can express the update mathematically as follows:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \left( \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l^{(i)}(\mathbf{w}) \right).$$

- the set cardinality  $|\mathcal{B}|$  represents the number of examples in each mini-batch (the *batch size*); the batch size is a hyperparameter of the model

## 1.3 Gradient descent

- after training for some predetermined number of iterations (or until some other stopping criteria are met), we record the *estimated model weights*, denoted  $\hat{w}$
- note that even if our function is truly linear and noiseless, these parameters will not be the exact minimizers of the loss because, although the algorithm converges slowly towards the minimizers, it cannot achieve it *exactly* in a finite number of steps
- linear regression happens to be a learning problem where there is only one minimum over the entire domain

## 1.3 Gradient descent

- however, for more complicated models, like deep networks, the loss surfaces contain many minima
- fortunately, for reasons that are not yet fully understood, deep learning practitioners rarely struggle to find parameters that minimize the loss *on training sets*
- the more formidable task is to find parameters that will achieve low loss on data that we have not seen before, a challenge called *generalization*
- given the learned linear regression model  $\hat{\mathbf{w}}^\top \mathbf{x}$ , we can now estimate target of a new sample (not contained in the training data) given its input features
- estimating targets given features is commonly called *prediction* or *inference*

## 1.4 Polynomial regression

- what if our data is more complex than a straight line?
- surprisingly, we can use a linear model to fit *nonlinear data*
- a simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features – this technique is called *polynomial regression*
- so far, we considered the linear regression setting described in (2), which allowed us to fit straight lines to data using maximum likelihood estimation
- however, straight lines are not sufficiently expressive when it comes to fitting more interesting data
- fortunately, linear regression offers us a way to fit nonlinear functions within the linear regression framework: since “linear regression” only refers to “linear in the parameters”, we can perform an arbitrary nonlinear transformation  $\phi(\mathbf{x})$  of the inputs  $\mathbf{x}$  and then linearly combine the components of this transformation

## 1.4 Polynomial regression

- the corresponding linear regression model is:

$$\hat{y} = \mathbf{w}^\top \phi(\mathbf{x}), \quad (9)$$

where  $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^p$  is a (nonlinear) transformation of the inputs  $\mathbf{x}$  and  $\phi_k : \mathbb{R}^d \rightarrow \mathbb{R}$  is the  $k$ th component of the feature vector  $\phi$

- note that the feature vector model parameters still appear only linearly
- note that when there are multiple features, polynomial regression is capable of finding relationships between features (which is something a plain linear regression model cannot do)
- this is made possible by the fact that we can also add all combinations of features up to the given degree
- for example, if there were two features  $a$  and  $b$ , we would not only add the features  $a^2$ ,  $a^3$ ,  $b^2$ , and  $b^3$ , but also the combinations  $ab$ ,  $a^2b$ , and  $ab^2$

## 1.4 Polynomial regression

- consider the dataset in Figure 13; the dataset consists of  $n = 10$  pairs  $(x^{(i)}, y^{(i)})$ , where  $x^{(i)} \sim \mathcal{U}[-5, 5]$  and

$$y^{(i)} = \sin\left(\frac{x^{(i)}}{5}\right) + \cos(x^{(i)}) + \epsilon,$$

where  $\epsilon \sim \mathcal{N}(0, 0.2^2)$

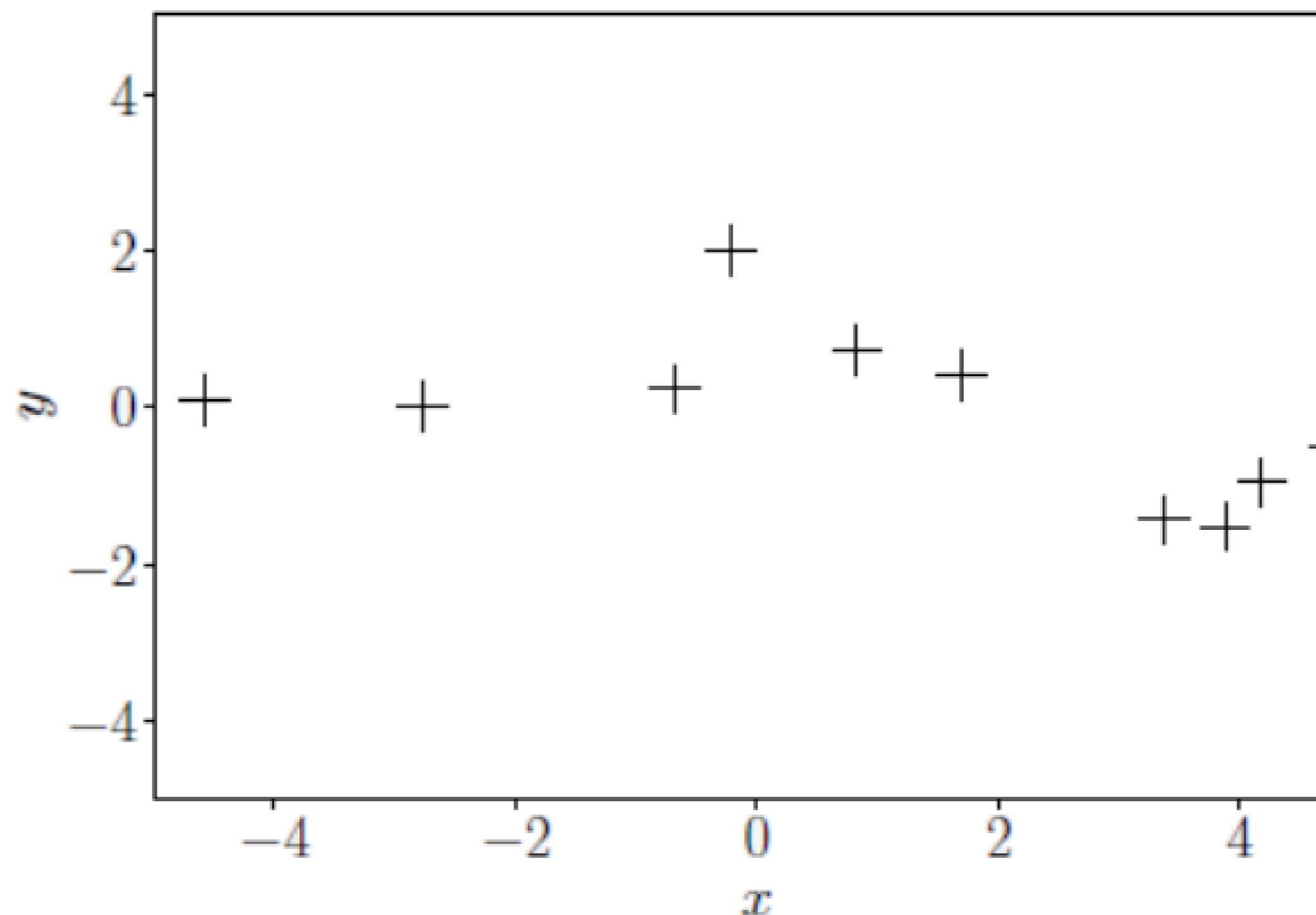


Figure 13: Polynomial regression: dataset consisting of  $(x^{(i)}, y^{(i)})$  pairs,  $i = 1, \dots, 10$ .

## 1.4 Polynomial regression

- assume that we use a polynomial model

$$\hat{y} = \mathbf{w}^\top \phi(x),$$

to fit it, where  $\mathbf{w} \in \mathbb{R}^M$ ,

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^{M-1} \end{bmatrix} \in \mathbb{R}^M.$$

- we can evaluate the quality of a model by computing the loss  $\mathcal{L}$
- instead of using this squared loss, we often use the *root mean square error (RMSE)*:

$$\sqrt{\frac{1}{n} \sum_{i=1}^n (\hat{y}^{(i)} - y^{(i)})^2} = \sqrt{\frac{1}{n} \sum_{i=1}^n (\mathbf{w}^\top \phi(\mathbf{x}^{(i)}) - y^{(i)})^2},$$

which

- allows us to compare errors of datasets with different sizes
- has the same scale and the same units as the observed function values  $y^{(i)}$

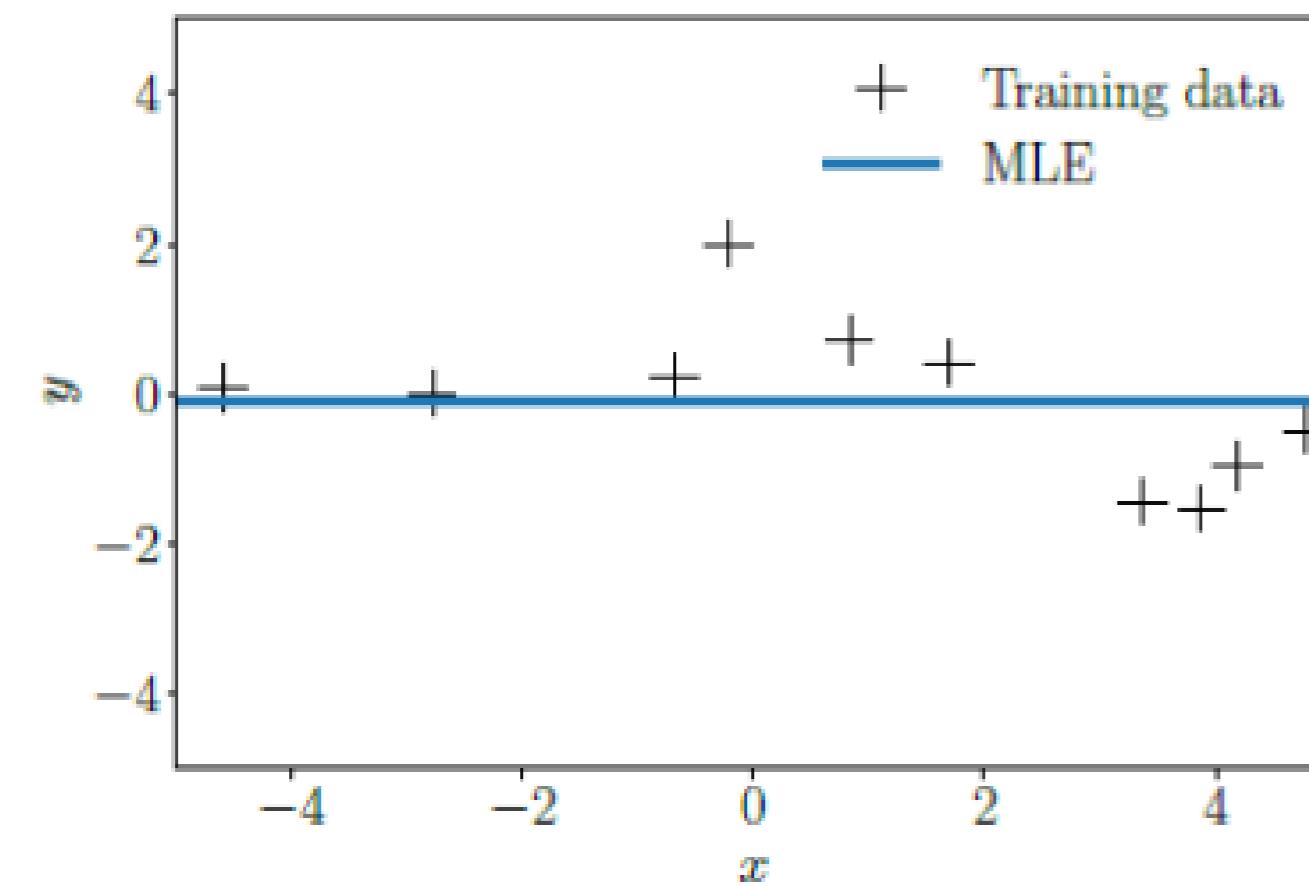
## 1.4 Polynomial regression

- in order to select the best model to fit our dataset, we can use the RMSE to determine the best degree of the polynomial by finding the polynomial degree  $M$  that minimizes the RMSE
- given that the polynomial degree is a natural number, we can perform a brute-force search and enumerate all (reasonable) values of  $M$
- for a training set of size  $n$  it is sufficient to test  $0 \leq M \leq n - 1$
- for  $M < n$ , the maximum likelihood estimator is unique
- for  $M \geq n$ , we have more parameters than data points, and would need to solve an underdetermined system of linear equations, so that there are infinitely many possible maximum likelihood estimators

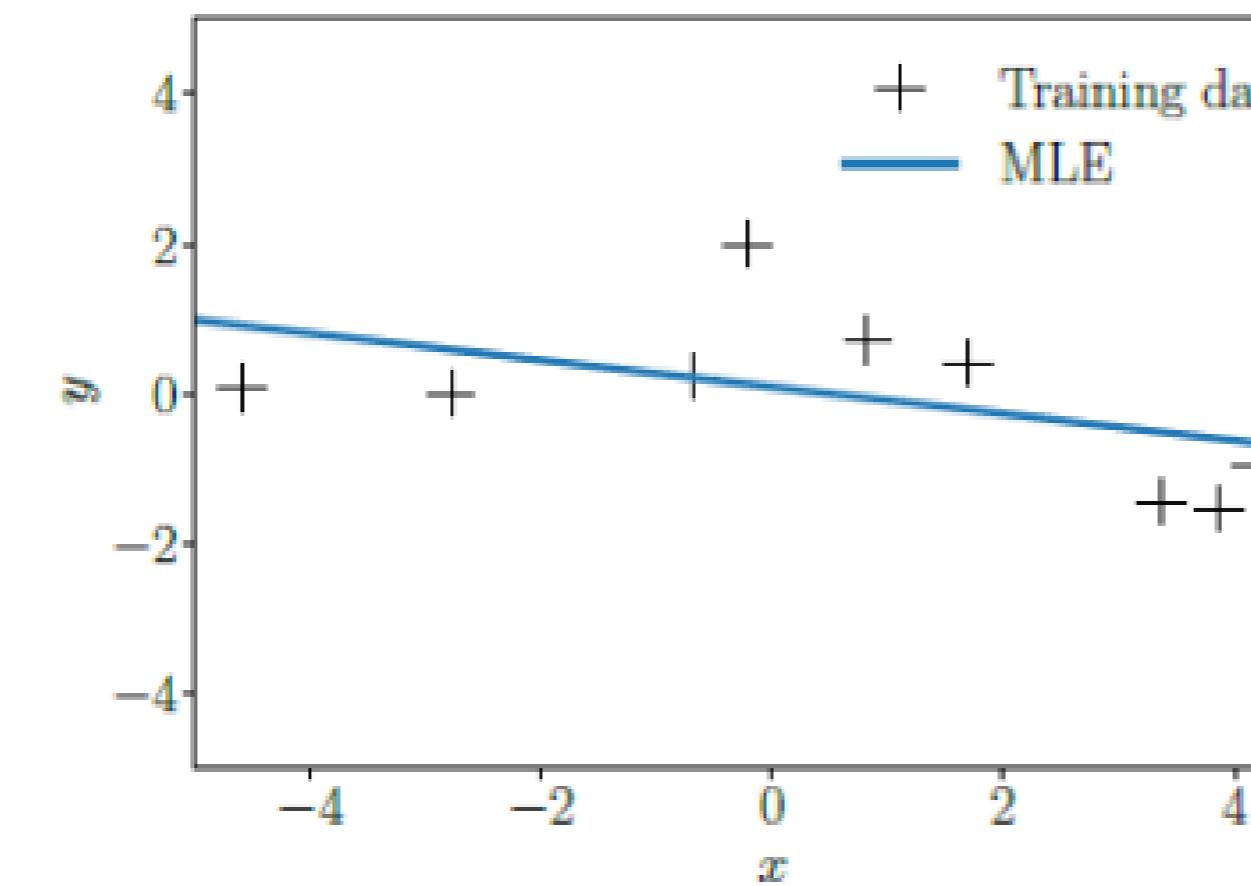
## 1.4 Polynomial regression

- Figure 14 shows a number of polynomial fits determined by polynomial regression for the dataset from Figure 13 with  $n = 10$  observations
- we notice that polynomials of low degree (e.g., constants ( $M = 0$ ) or linear ( $M = 1$ )) fit the data poorly and, hence, are poor representations of the true underlying function
- for degrees  $M = 3, \dots, 6$ , the fits look plausible and smoothly interpolate the data
- when we go to higher-degree polynomials, we notice that they fit the data better and better
- in the extreme case of  $M = n - 1 = 9$ , the function will pass through every single data point
- however, these high-degree polynomials oscillate wildly and are a poor representation of the underlying function that generated the data, such that we suffer from *overfitting*

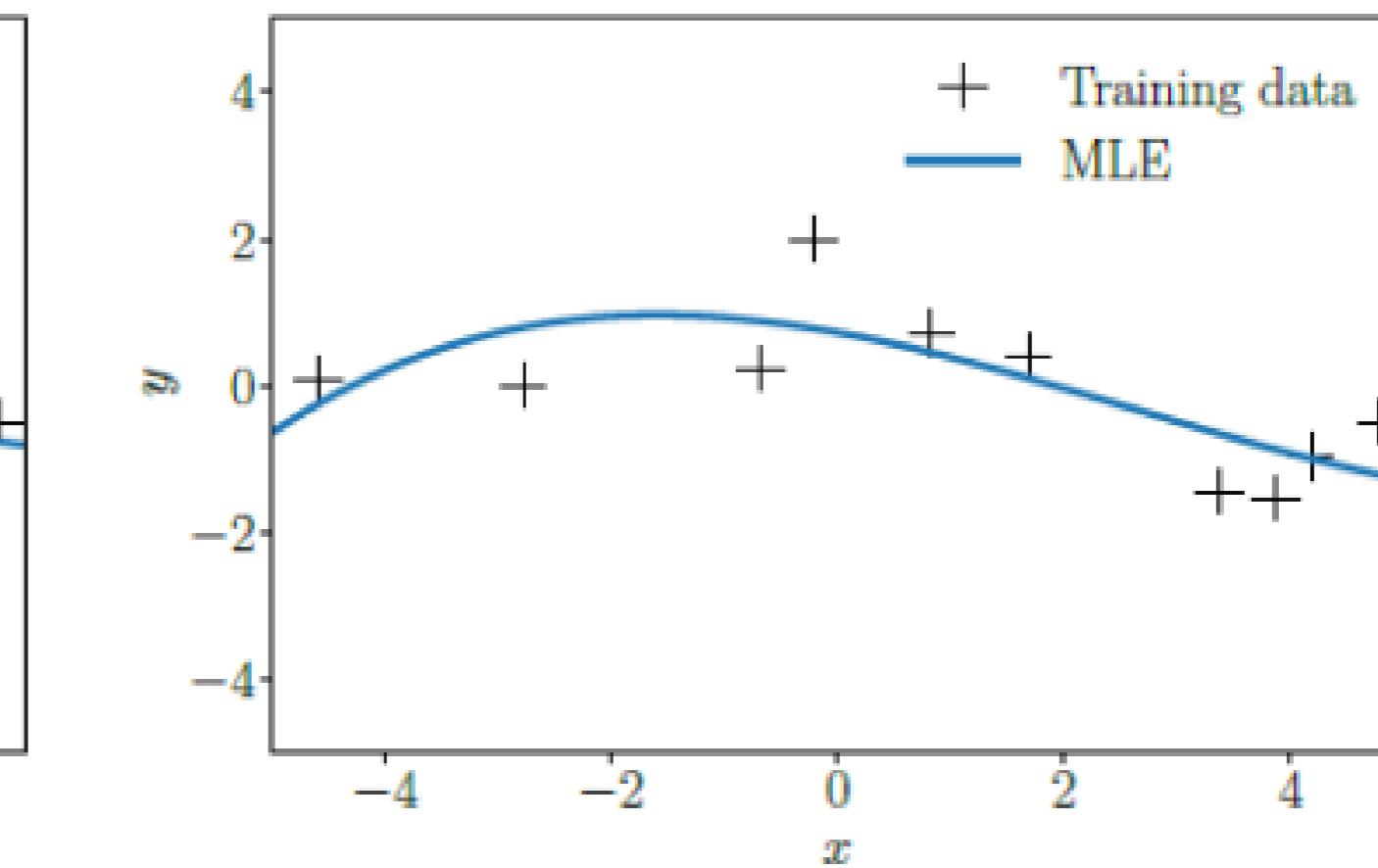
# 1.4 Polynomial regression



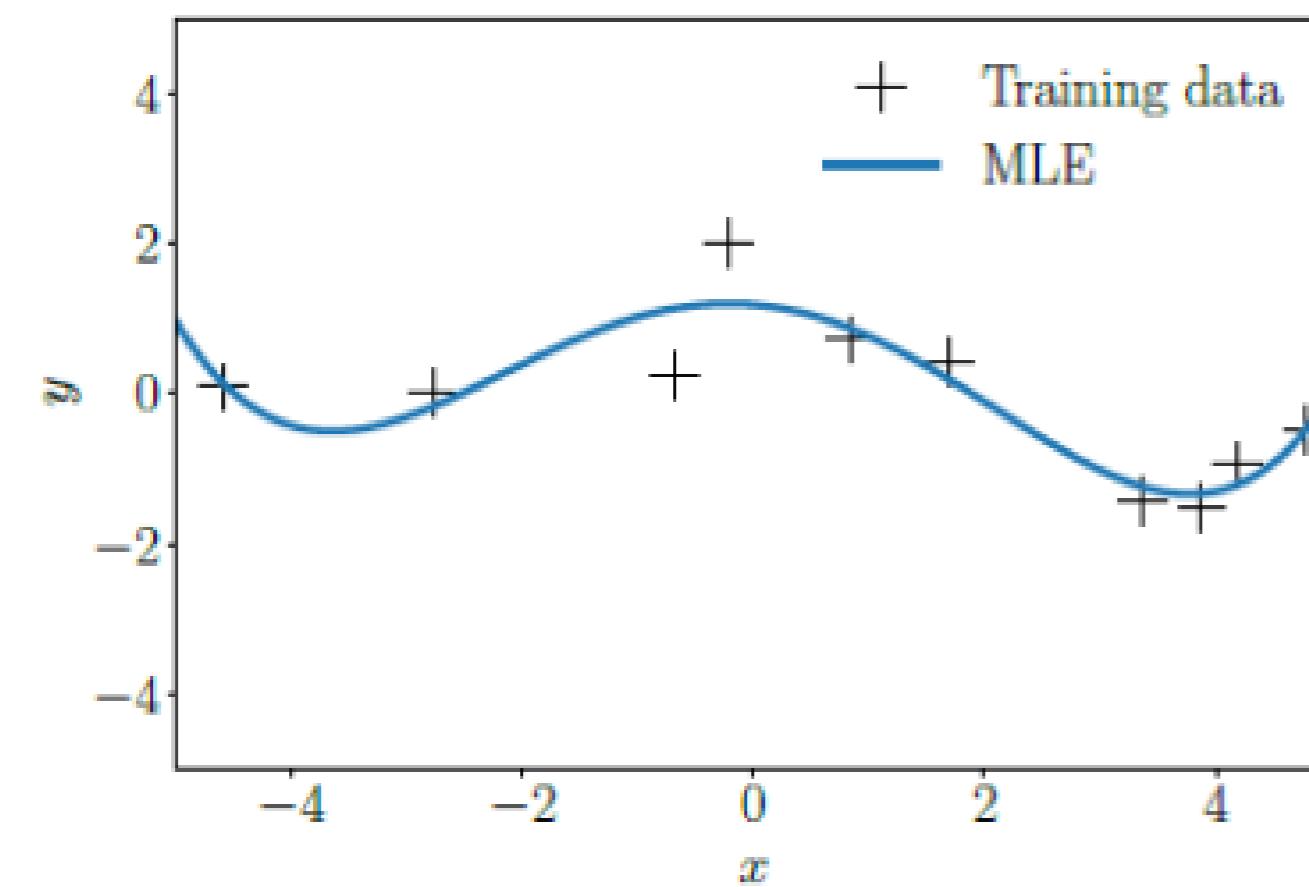
(a)  $M = 0$



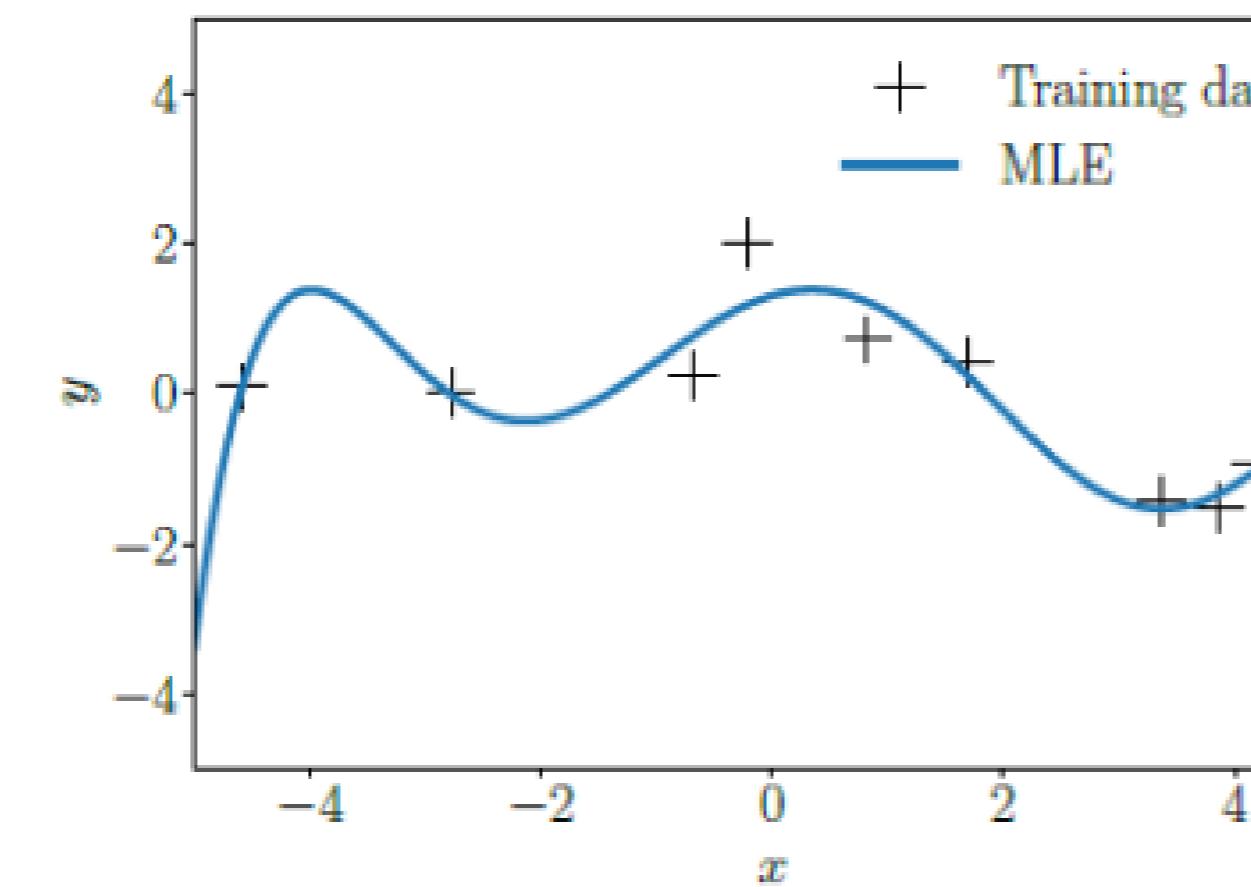
(b)  $M = 1$



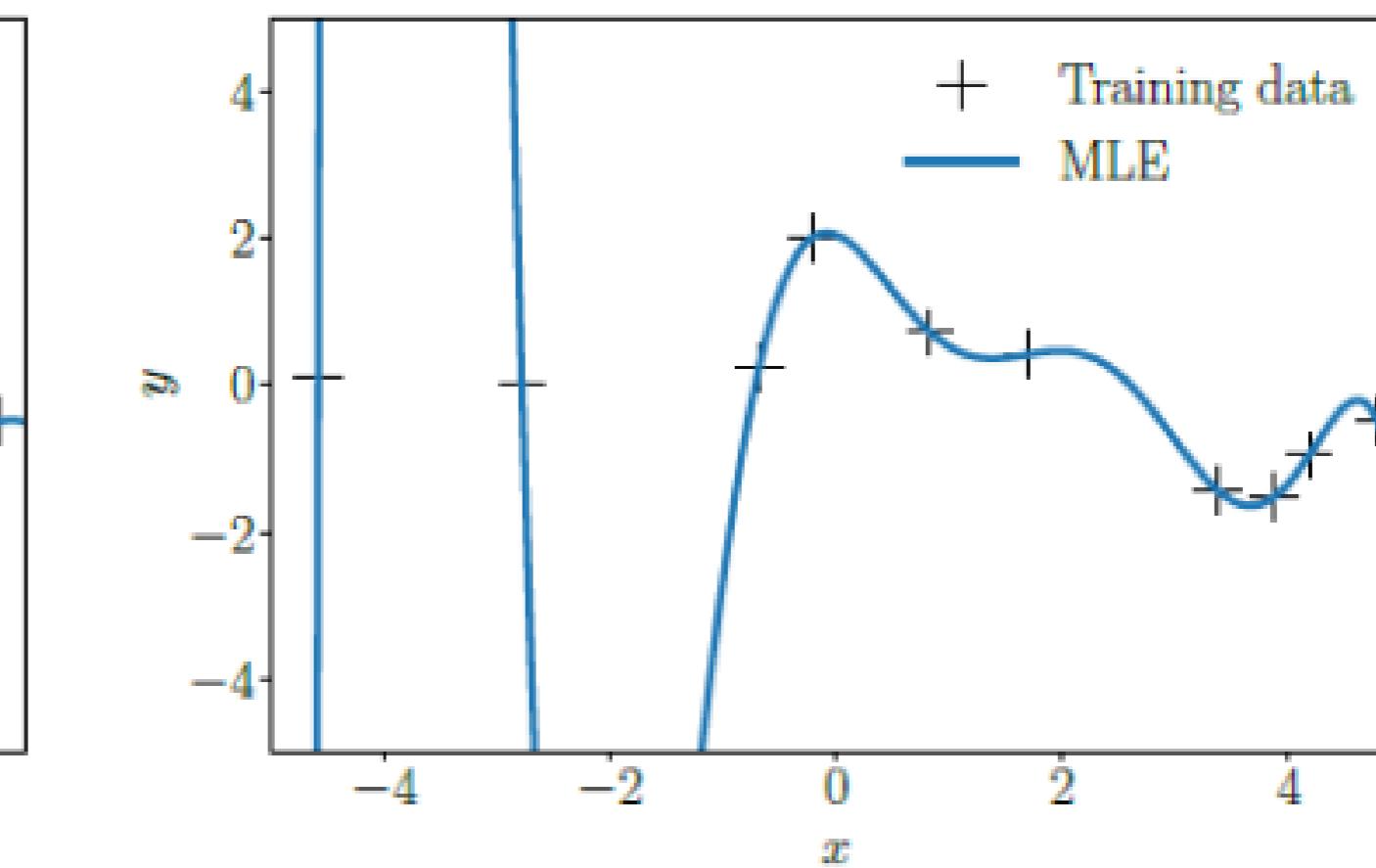
(c)  $M = 3$



(d)  $M = 4$



(e)  $M = 6$



(f)  $M = 9$

Figure 14: Maximum likelihood fits for different polynomial degrees  $M$ .

## 1.4 Polynomial regression

- however, the goal is to achieve good *generalization* by making accurate predictions for new (unseen) data
- we obtain some quantitative insight into the dependence of the generalization performance on the polynomial of degree  $M$  by considering a separate test set comprising 200 data points generated using exactly the same procedure used to generate the training set
- as test inputs, we choose a linear grid of 200 points in the interval of  $[-5, 5]$
- for each choice of  $M$ , we evaluate the RMSE for both the training data and the test data

## 1.4 Polynomial regression

- looking now at the *test error*, which is a qualitative measure of the generalization properties of the corresponding polynomial, we notice that initially the test error decreases; see Figure 15 (orange)
- for fourth-order polynomials, the test error is relatively low and stays relatively constant up to degree 5
- however, from degree 6 onward, the test error increases significantly, and high-order polynomials have very bad generalization properties
- in this particular example, this also is evident from the corresponding maximum likelihood fits in Figure 14

## 1.4 Polynomial regression

- note that the *training error* (blue curve in Figure 15) never increases when the degree of the polynomial increases
- in our example, the best generalization (the point of the smallest *test error*) is obtained for a polynomial of degree  $M = 4$

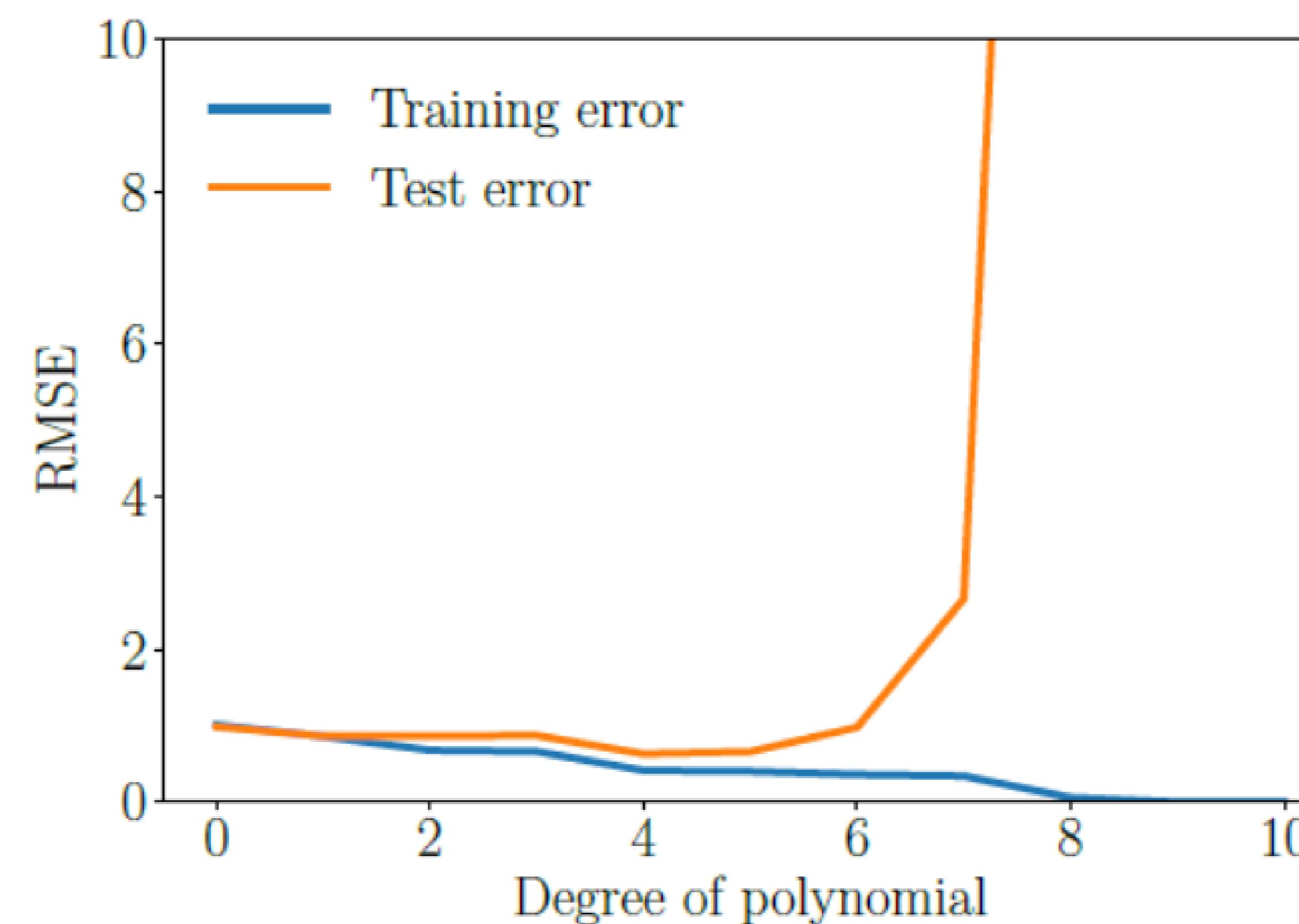


Figure 15: Training and test error.

# 1.4 Polynomial regression

- an important theoretical result of statistics and machine learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:
  - ① *Bias*. This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.
  - ② *Variance*. This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance and thus overfit the training data.
  - ③ *Irreducible error*. This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).
- increasing a model's complexity will typically increase its variance and reduce its bias
- conversely, reducing a model's complexity increases its bias and reduces its variance
- this is why it is called the *bias/variance trade-off*

# 1.5 Regularized linear models

- we just saw that maximum likelihood estimation is prone to overfitting
- we often observe that the magnitude of the parameter values becomes *relatively large* if we run into overfitting
- to mitigate the effect of huge parameter values, we can place a *prior distribution*  $p(\mathbf{w})$  on the parameters
- the prior distribution explicitly encodes what parameter values are plausible (before having seen any data)
- for example, a Gaussian prior  $w \sim \mathcal{N}(0, 1)$  on a single parameter encodes that parameter values are expected to lie in the interval  $[-2, 2]$  (two standard deviations around the mean value)

# 1.5 Regularized linear models

- once a dataset  $\mathbf{X}, \mathbf{y}$  is available, instead of maximizing the likelihood, we seek parameters that maximize the posterior distribution  $p(\mathbf{w}|\mathbf{X}, \mathbf{y})$
- this procedure is called *maximum a posteriori* estimation
- the posterior over the weights  $\mathbf{w}$ , given the training data  $\mathbf{X}, \mathbf{y}$ , is obtained by applying Bayes' theorem as:

$$p(\mathbf{w}|\mathbf{X}, \mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{X}, \mathbf{w})p(\mathbf{w})}{p(\mathbf{y}|\mathbf{X})}.$$

- to find the maximum a posteriori estimation, we minimize the *negative log-posterior distribution* with respect to  $\mathbf{w}$ :
$$-\log p(\mathbf{w}|\mathbf{X}, \mathbf{y}) = -\log p(\mathbf{y}|\mathbf{X}, \mathbf{w}) - \log p(\mathbf{w}) + \log p(\mathbf{y}|\mathbf{X}). \quad (10)$$

- we see that the log-posterior in (10) is the sum of the log-likelihood  $\log p(\mathbf{y}|\mathbf{X}, \mathbf{w})$  and the log-prior  $\log p(\mathbf{w})$ , so that the maximum a posteriori estimate will be a “compromise” between the prior (our suggestion for plausible parameter values before observing data) and the data-dependent likelihood

# 1.5 Regularized linear models

- first, assume that the weights are normally distributed with variance  $\tau$ ,  $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \tau^2 \mathbf{I})$ , which implies that:

$$-\log p(\mathbf{w} | \mathbf{X}, \mathbf{y}) = \frac{1}{2\sigma^2} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{1}{2\tau^2} \mathbf{w}^\top \mathbf{w} + \text{const.} \quad (11)$$

where const. represent terms that are independent of  $\mathbf{w}$

- here, the first term corresponds to the contribution from the log-likelihood, and the second term originates from the log-prior
- by taking the gradient of (11) and making it equal with  $\mathbf{0}^\top$ , the maximum a posteriori estimate for  $\mathbf{w}$  will be:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y},$$

where we denoted  $\lambda := \frac{\sigma^2}{\tau^2}$

# 1.5 Regularized linear models

- similarly as for maximum likelihood estimation, we can define a loss corresponding to maximum a posteriori estimation:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2n}(\mathbf{X}\mathbf{w} - \mathbf{y})^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{1}{2}\lambda\|\mathbf{w}\|_2^2,$$

where  $\|\mathbf{w}\|_2 = \left(\sum_{j=0}^d |\mathbf{w}_j|^2\right)^{1/2}$  is the  $\ell_2$  norm of  $\mathbf{w}$

- linear regression using this loss function is called *ridge regression*
- ridge regression is a *regularized* version of linear regression: a regularization term equal to  $\frac{1}{2}\lambda\|\mathbf{w}\|_2^2$  is added to the loss function
- this forces the learning algorithm to not only fit the data but also keep the model weights as *small* as possible
- note that the regularization term should only be added to the loss function during training
- once the model is trained, we want to use the unregularized performance measure to evaluate the model's performance

# 1.5 Regularized linear models

- the hyperparameter  $\lambda$  controls how much we want to regularize the model
- if  $\lambda = 0$ , then ridge regression is just linear regression
- if  $\lambda$  is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean
- as with linear regression, we can perform ridge regression either by computing a closed-form equation or by performing gradient descent
- the pros and cons are the same; for gradient descent, we just add  $\lambda\mathbf{w}$  to the gradient vector:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{1}{n} \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda\mathbf{w}.$$

- it is important to *scale* the data before performing ridge regression, as it is sensitive to the scale of the input features, which is true of most regularized models

# 1.5 Regularized linear models

- on the other hand, we can assume a Laplace distribution for the weights,  $\mathbf{w} \sim \text{Lap}(\mathbf{0}, b\mathbf{I})$
- the probability density for the Laplace distribution is given as:

$$p(x|\mu, b) = \frac{1}{2b} \exp\left(-\frac{1}{b}|x - \mu|\right).$$

- here,  $\mu$  is a location parameter and  $b > 0$  is a scale parameter
- the loss corresponding to the Laplace prior on the weights is:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2n} (\mathbf{X}\mathbf{w} - \mathbf{y})^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda \|\mathbf{w}\|_1,$$

where  $\|\mathbf{w}\|_1 = \sum_{j=0}^d |w_j|$  is the  $\ell_1$  norm of  $\mathbf{w}$

- linear regression using this loss function is called *lasso regression*

# 1.5 Regularized linear models

- *Least Absolute Shrinkage and Selection Operator Regression* (usually simply called lasso regression) is another regularized version of linear regression: just like ridge regression, it adds a regularization term to the loss function, but it uses the  $\ell_1$  norm of the weight vector instead of half the square of the  $\ell_2$  norm
- an important characteristic of lasso regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero)
- in other words, lasso regression automatically performs *feature selection* and outputs a *sparse model*
- that is, it often sets many weights to zero, effectively declaring the corresponding attributes to be completely irrelevant
- models that discard attributes can be easier for a human to understand, and may be less likely to overfit

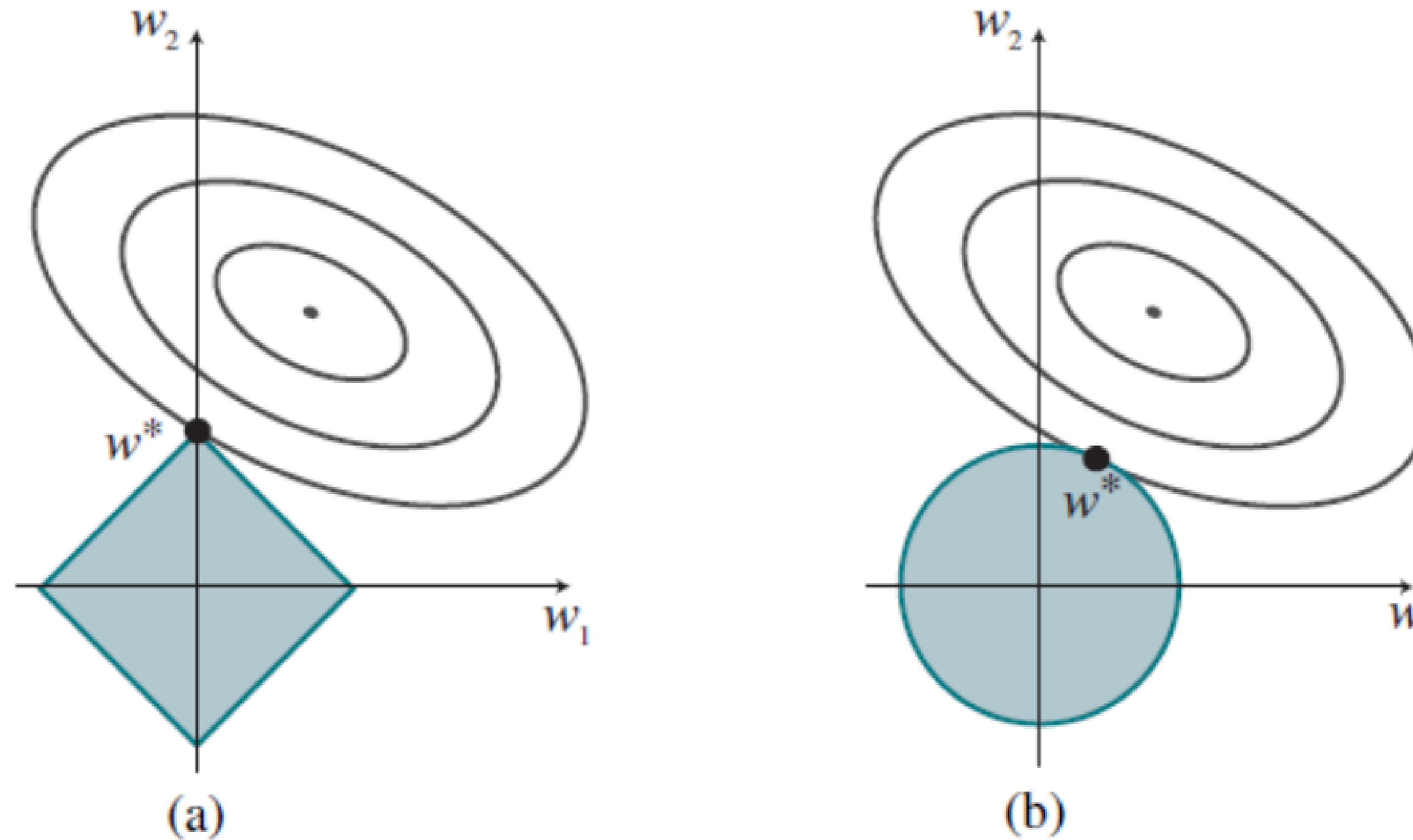
# 1.5 Regularized linear models

- Figure 16 gives an intuitive explanation of why  $\ell_1$  regularization leads to weights of zero, while  $\ell_2$  regularization does not
- note that minimizing  $\frac{1}{2n}(\mathbf{X}\mathbf{w} - \mathbf{y})^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) + \frac{1}{2}\lambda\|\mathbf{w}\|_2^2$  or  $\frac{1}{2n}(\mathbf{X}\mathbf{w} - \mathbf{y})^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda\|\mathbf{w}\|_1$  is equivalent to minimizing  $\frac{1}{2n}(\mathbf{X}\mathbf{w} - \mathbf{y})^\top(\mathbf{X}\mathbf{w} - \mathbf{y})$  subject to the constraint that  $\frac{1}{2}\lambda\|\mathbf{w}\|_2^2 \leq c$  or  $\lambda\|\mathbf{w}\|_1 \leq c$ , for some constant  $c$  that is related to  $\lambda$
- now, in Figure 16(a) the diamond-shaped box represents the set of points  $\mathbf{w}$  in two-dimensional weight space that have  $\ell_1$  norm less than  $c$ ; our solution will have to be somewhere inside this box
- the concentric ovals represent contours of the mean squared error function, with the minimum loss at the center

## 1.5 Regularized linear models

- we want to find the point in the box that is *closest* to the minimum; we can see from the diagram that, for an arbitrary position of the minimum and its contours, it will be common for the corner of the box to find its way closest to the minimum, just because the corners are pointy
- and of course the corners are the points that have a value of zero in some dimension
- in Figure 16(b), we have done the same for the  $\ell_2$  norm, which represents a circle rather than a diamond
- here we can see that, in general, there is no reason for the intersection to appear on one of the axes; thus  $\ell_2$  regularization does not tend to produce zero weights

# 1.5 Regularized linear models



**Figure 16:** Why  $\ell_1$  regularization tends to produce a sparse model. Left: with  $\ell_1$  regularization (box), the minimal achievable loss (concentric contours) often occurs on an axis, meaning a weight of zero. Right: with  $\ell_2$  regularization (circle), the minimal loss is likely to occur anywhere on the circle, giving no preference to zero weights.

# 1.5 Regularized linear models

- the lasso loss function is not differentiable whenever  $w_j = 0$  for some  $j = 0, 1, \dots, d$ , but gradient descent still works fine if we use a *subgradient vector*  $\mathbf{g}$  instead, when any  $w_j = 0$
- we can think of a subgradient vector at a nondifferentiable point as an intermediate vector between the gradient vectors around that point
- a subgradient vector we can use for gradient descent with the lasso loss function is:

$$\mathbf{g}(\mathbf{w}) = \frac{1}{n} \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) + \lambda \text{sign}(\mathbf{w}),$$

where  $\text{sign}(x) = \begin{cases} -1, & \text{if } x < 0 \\ 0, & \text{if } x = 0 \\ +1, & \text{if } x > 0 \end{cases}$

# 1.5 Regularized linear models

- *elastic net* is a middle ground between ridge regression and lasso regression
- the regularization term is a simple mix of both ridge and lasso's regularization terms, and we can control the mix ratio  $r \in [0, 1]$
- when  $r = 0$ , elastic net is equivalent to ridge regression, and when  $r = 1$ , it is equivalent to lasso regression:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2n}(\mathbf{X}\mathbf{w} - \mathbf{y})^\top(\mathbf{X}\mathbf{w} - \mathbf{y}) + r\lambda||\mathbf{w}||_1 + \frac{1-r}{2}\lambda||\mathbf{w}||_2^2.$$

# 1.5 Regularized linear models

- so when should we use plain linear regression (i.e., without any regularization), ridge, lasso, or elastic net?
- it is almost always preferable to have at least a *little bit of regularization*, so generally we should avoid plain linear regression
- ridge is a good default, but if we suspect that only a few features are useful, we should prefer lasso or elastic net because they tend to reduce the useless features' weights down to zero, as we have discussed
- in general, elastic net is preferred over lasso, because lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated

# 1.5 Regularized linear models

- a very different way to *regularize* iterative learning algorithms such as gradient descent is to stop training as soon as the validation error reaches a minimum; this is called *early stopping*
- Figure 17 shows a complex model (in this case, a high-degree polynomial regression model) being trained with batch gradient descent
- as the epochs go by, the algorithm learns, and its prediction error (RMSE) on the training set goes down, along with its prediction error on the validation set
- after a while though, the validation error stops decreasing and starts to go back up
- this indicates that the model has started to overfit the training data

# 1.5 Regularized linear models

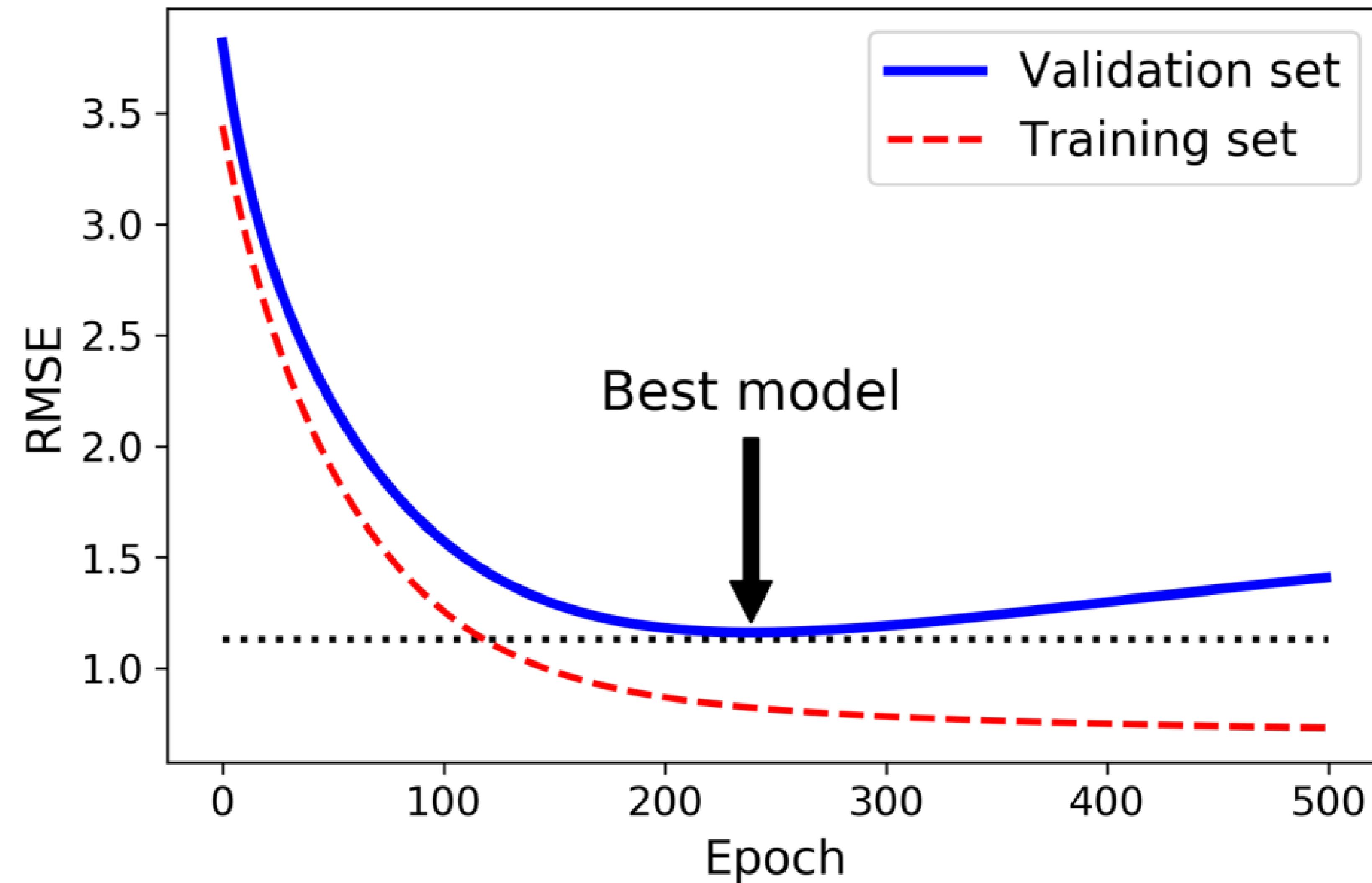


Figure 17: Early stopping regularization.

# 1.5 Regularized linear models

- with early stopping, we just stop training as soon as the validation error reaches the minimum
- with stochastic and mini-batch gradient descent, the curves are not so smooth, and it may be hard to know whether we have reached the minimum or not
- one solution is to stop only after the validation error has been above the minimum for some time (when we are confident that the model will not do any better), then roll back the model parameters to the point where the validation error was at a minimum

# 1.6 Logistic regression

- some regression algorithms can be used for classification (and vice versa)
- *logistic regression* (also called *logit regression*) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?)
- if the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the *positive class*, labeled “1”), and otherwise it predicts that it does not (i.e., it belongs to the *negative class*, labeled “0”)
- this makes it a *binary classifier*

# 1.6 Logistic regression

- just like a linear regression model, a logistic regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly, like the linear regression model does, it outputs the *logistic* of this result:

$$\hat{p} = \sigma(\mathbf{w}^T \mathbf{x}).$$

- the logistic, denoted  $\sigma(\cdot)$ , is the *sigmoid function* (i.e., S-shaped) that outputs a number between 0 and 1; it is defined as:

$$\sigma(t) = \frac{1}{1 + \exp(-t)},$$

and its graph is given in Figure 18

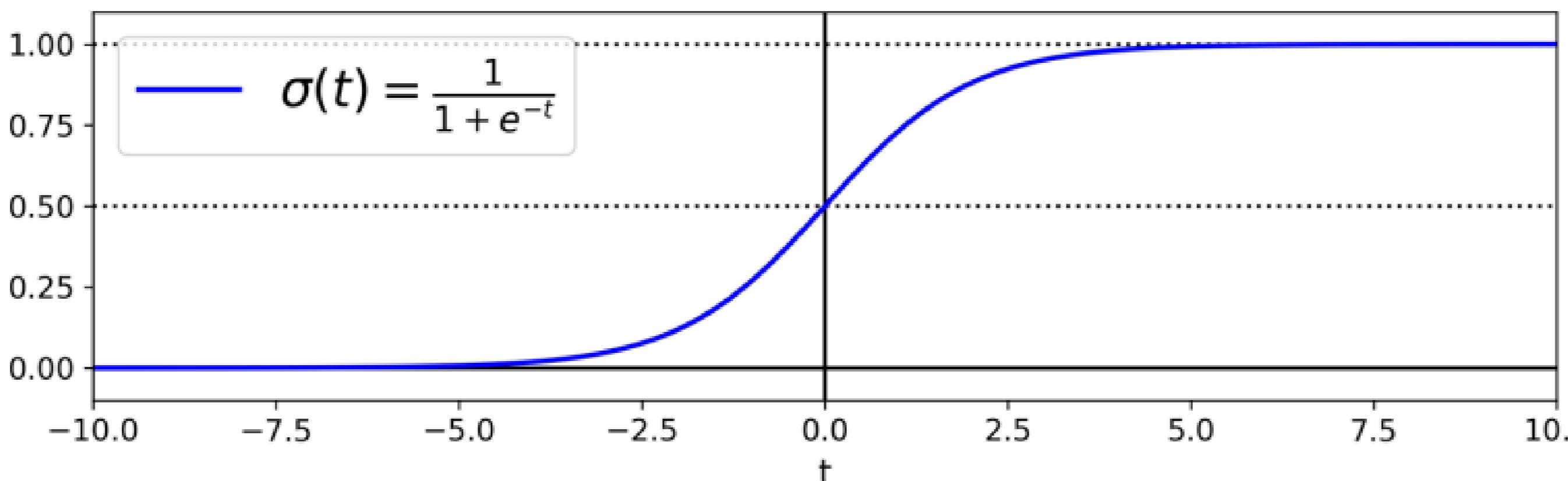


Figure 18: Sigmoid function.

# 1.6 Logistic regression

- once the logistic regression model has estimated the probability  $\hat{p}$  that an instance  $\mathbf{x}$  belongs to the positive class, it can make its prediction  $\hat{y}$  easily as:

$$\hat{y} = \begin{cases} 0, & \text{if } \hat{p} < 0.5 \\ 1, & \text{if } \hat{p} \geq 0.5 \end{cases}.$$

- notice that  $\sigma(t) < 0.5$  when  $t < 0$ , and  $\sigma(t) \geq 0.5$  when  $t \geq 0$ , so a logistic regression model predicts 1 if  $\mathbf{w}^T \mathbf{x}$  is positive and 0 if it is negative
- the score  $t$  is often called the *logit*; the name comes from the fact that the *logit function*, defined as  $\text{logit}(\hat{p}) = \log(\hat{p}/(1 - \hat{p}))$ , is the inverse of the sigmoid function
- indeed, if we compute the logit of the estimated probability  $\hat{p}$ , we will find that the result is  $t$
- the logit is also called the *log-odds*, since it is the log of the ratio between the estimated probability for the positive class and the estimated probability for the negative class

# 1.6 Logistic regression

- for training the logistic regression, the objective is to set the weight vector  $\mathbf{w}$  so that the model estimates high probabilities for positive instances ( $y = 1$ ) and low probabilities for negative instances ( $y = 0$ )
- this idea is captured by the loss function shown below, for a single training instance  $\mathbf{x}$ :

$$l(\mathbf{w}) = \begin{cases} -\log \hat{p}, & \text{if } y = 1 \\ -\log(1 - \hat{p}), & \text{if } y = 0 \end{cases}.$$

- this loss function makes sense because  $-\log t$  grows very large when  $t$  approaches 0, so the loss will be large if the model estimates a probability close to 0 for a positive instance, and it will also be very large if the model estimates a probability close to 1 for a negative instance
- on the other hand,  $-\log t$  is close to 0 when  $t$  is close to 1, so the loss will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance, which is precisely what we want

# 1.6 Logistic regression

- it can be also written as:

$$l(\mathbf{w}) = -y \log \hat{p} - (1 - y) \log(1 - \hat{p}).$$

- the loss function over the whole training set is the average loss over all training instances
- it can be written in a single expression called the *log loss*, as:

$$\mathcal{L}(\mathbf{w}) = -\frac{1}{n} \sum_{i=1}^n \left[ y^{(i)} \log \hat{p}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]. \quad (12)$$

- the bad news is that there is no known closed-form equation to compute the value of  $\mathbf{w}$  that minimizes this loss function (there is no equivalent of the normal equation)
- the good news is that this loss function is convex, so gradient descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and we wait long enough)

# 1.6 Logistic regression

- the gradient of the loss function is given by:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{1}{n} \mathbf{X}^\top (\sigma(\mathbf{X}\mathbf{w}) - \mathbf{y}).$$

- this equation looks very much like equation (8): for each instance it computes the prediction error and multiplies it by the feature values, and then it computes the average over all training instances
- once we have the gradient vector containing all the partial derivatives, we can use it in the batch gradient descent algorithm
- for stochastic GD, we would take one instance at a time, and for mini-batch GD we would use a mini-batch at a time
- just like the other linear models, logistic regression models can be regularized using  $\ell_1$  or  $\ell_2$  penalties

# 1.7 Softmax regression

- the logistic regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers
- this is called *softmax regression*, or *multinomial logistic regression*
- assume we have a classification problem with  $q$  classes
- the idea is simple: when given an instance  $\mathbf{x}$ , the softmax regression model first computes a score  $o_k(\mathbf{x})$  for each class  $k = 1, 2, \dots, q$ , then estimates the probability of each class by applying the *softmax function* (also called the *normalized exponential*) to the scores
- the equation to compute  $o_k(\mathbf{x})$  should look familiar, as it is just like the equation for linear regression prediction:

$$o_k(\mathbf{x}) = \mathbf{w}_k^\top \mathbf{x}.$$

# 1.7 Softmax regression

- note that each class has its own dedicated parameter vector  $\mathbf{w}_k \in \mathbb{R}^d$
- all these vectors are typically stored as columns in a parameter matrix  $\mathbf{W} \in \mathbb{R}^{d \times q}$ :

$$\mathbf{W} = [ \mathbf{w}_1 \quad \mathbf{w}_2 \quad \cdots \quad \mathbf{w}_q ] .$$

- once we have computed the score of every class for the instance  $\mathbf{x}$ , we can estimate the probability  $\hat{p}_k$  that the instance belongs to class  $k$  by running the scores through the softmax function:

$$\hat{p}_k = \text{softmax}(\mathbf{o}(\mathbf{x}))_k = \frac{\exp(o_k(\mathbf{x}))}{\sum_{l=1}^q \exp(o_l(\mathbf{x}))},$$

where

- $\mathbf{o}(\mathbf{x})$  is a vector containing the scores of each class for the instance  $\mathbf{x}$
- $\text{softmax}(\mathbf{o}(\mathbf{x}))_k$  is the estimated probability that the instance  $\mathbf{x}$  belongs to class  $k$ , given the scores of each class for that instance

## 1.7 Softmax regression

- the function computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials)
- the scores are generally called *logits* or *log-odds* (although they are actually *unnormalized log-odds*)
- just like the logistic regression classifier, the softmax regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score):

$$\hat{y} = \operatorname{argmax}_k \text{softmax}(\mathbf{o}(\mathbf{x}))_k = \operatorname{argmax}_k o_k(\mathbf{x}) = \operatorname{argmax}_k \mathbf{w}_k^\top \mathbf{x}.$$

- the argmax operator returns the value of a variable that maximizes a function
- in this equation, it returns the value of  $k$  that maximizes the estimated probability  $\text{softmax}(\mathbf{o}(\mathbf{x}))_k$
- the softmax regression classifier predicts only one class at a time (i.e., it is multiclass, not multioutput), so it should be used only with mutually exclusive classes, such as different types of plants, not multiple people in one picture

# 1.7 Softmax regression

- for training, the objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes)
- minimizing the loss function shown below, called the *cross entropy*, should lead to this objective, because it penalizes the model when it estimates a low probability for a target class
- cross entropy is frequently used to measure how well a set of estimated class probabilities matches the target classes

$$\mathcal{L}(\mathbf{W}) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^q y_k^{(i)} \log \hat{p}_k^{(i)},$$

where

- $y_k^{(i)}$  is the target probability that the  $i$ th instance belongs to class  $k$ . In general, it is either equal to 1 or 0, depending on whether the instance belongs to the class or not.
- notice that when there are just two classes ( $q = 2$ ), this loss function is equivalent to the logistic regression's loss function in (12)

# 1.7 Softmax regression

- the cross entropy between probability distributions  $p$  and  $q$ , denoted  $H(p, q)$ , measures the *mismatch* between probabilities  $q$  upon seeing data that were actually generated according to probabilities  $p$ :

$$H(p, q) = - \sum_x p(x) \log q(x).$$

- the lowest possible cross entropy is achieved when  $p = q$ ; in this case, the cross-entropy between  $p$  and  $q$  is:

$$H(p, p) = H(p) = - \sum_x p(x) \log p(x),$$

i.e., cross entropy is equal to the entropy of probability distribution  $p$

- the gradient of the cross entropy loss function with regard to  $\mathbf{w}_k$  is given by:

$$\nabla_{\mathbf{w}_k} \mathcal{L}(\mathbf{W}) = \frac{1}{n} \mathbf{X}^\top (\hat{\mathbf{p}}_k - \mathbf{y}_k) = \frac{1}{n} \mathbf{X}^\top (\text{softmax}(\mathbf{o}(\mathbf{x}))_k - \mathbf{y}_k).$$

- now we can compute the gradient vector for every class, then use gradient descent (or any other optimization algorithm) to find the parameter matrix  $\mathbf{W}$  that minimizes the loss function

# Exercises

## Exercise 1

Assume we have a dataset  $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)})\} = \{(1, 2), (2, 4), (3, 5)\}$  of points in the 2D plane, and we want to estimate the following linear regression model:

$$\hat{y} = w_1 x + w_0.$$

Compute  $w_0$ ,  $w_1$  and  $\hat{y}^{(1)}$ ,  $\hat{y}^{(2)}$ ,  $\hat{y}^{(3)}$ .

**Solution:**

- we have the following formulas:

$$\mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots & x_d^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots & x_d^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(n)} & x_2^{(n)} & \dots & x_d^{(n)} \end{bmatrix} = \begin{bmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \vdots \\ \mathbf{x}^{(n)\top} \end{bmatrix} \in \mathbb{R}^{n \times (d+1)},$$

# Exercises

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix} \in \mathbb{R}^{n \times 1},$$

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \in \mathbb{R}^{(d+1) \times 1},$$

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \vdots \\ \hat{y}^{(n)} \end{bmatrix} \in \mathbb{R}^{n \times 1},$$

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} \in \mathbb{R}^{n \times 1}.$$

# Exercises

- in our case,  $n = 3$ ,  $d = 1$ , so we apply the formulas:

$$\mathbf{X} = \begin{bmatrix} 1 & x^{(1)} \\ 1 & x^{(2)} \\ 1 & x^{(3)} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{bmatrix} \in \mathbb{R}^{3 \times 2},$$

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ y^{(3)} \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 5 \end{bmatrix} \in \mathbb{R}^{3 \times 1},$$

# Exercises

$$\begin{aligned}\mathbf{w}^* &= \left( \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ 5 \end{bmatrix} \\ &= \begin{bmatrix} 3 & 6 \\ 6 & 14 \end{bmatrix}^{-1} \begin{bmatrix} 11 \\ 25 \end{bmatrix} \\ &= \begin{bmatrix} \frac{7}{3} & -1 \\ -1 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} 11 \\ 25 \end{bmatrix} \\ &= \begin{bmatrix} \frac{2}{3} \\ \frac{3}{2} \end{bmatrix} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} \in \mathbb{R}^{2 \times 1},\end{aligned}$$

$$\hat{\mathbf{y}} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} \frac{2}{3} \\ \frac{3}{2} \end{bmatrix} = \begin{bmatrix} \frac{13}{6} \\ \frac{22}{6} \\ \frac{31}{6} \end{bmatrix} = \begin{bmatrix} \hat{y}^{(1)} \\ \hat{y}^{(2)} \\ \hat{y}^{(3)} \end{bmatrix} \in \mathbb{R}^{3 \times 1}.$$

# Vă mulțumesc!