



Xi'an Jiaotong-Liverpool University

西交利物浦大学

**INT305 Machine Learning**

**Lecture 2**

**Linear Methods for Regression, Optimization**

**Jimin Xiao**

**Department Intelligence Science**

**[Jimin.xiao@xjtlu.edu.cn](mailto:Jimin.xiao@xjtlu.edu.cn)**

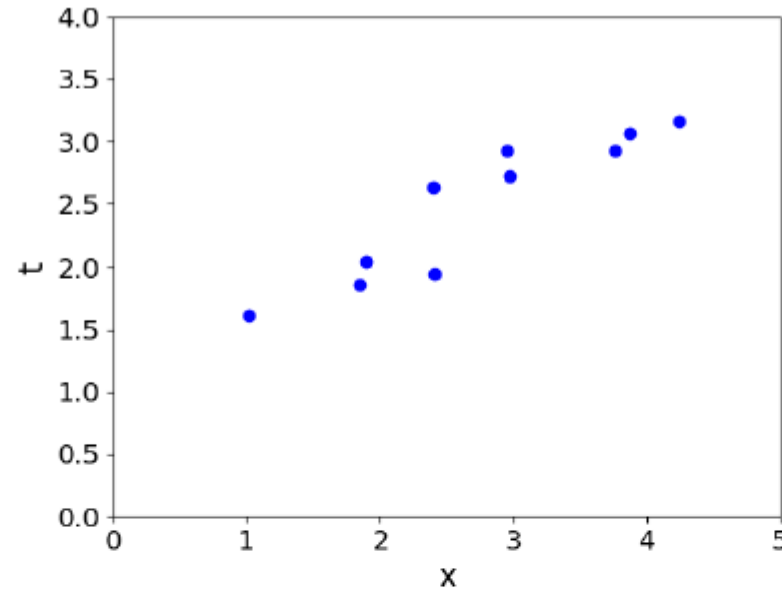
# Overview

---

- Second learning algorithm of the course: **linear regression**.
  - ▶ **Task**: predict scalar-valued targets (e.g. stock prices)
  - ▶ **Architecture**: linear function of the inputs
- While KNN was a complete algorithm, linear regression exemplifies a modular approach that will be used throughout this course:
  - ▶ choose a **model** describing the relationships between variables of interest
  - ▶ define a **loss function** quantifying how bad the fit to the data is
  - ▶ choose a **regularizer** saying how much we prefer different candidate models (or explanations of data)
  - ▶ fit a model that minimizes the loss function and satisfies the constraint/penalty imposed by the regularizer, possibly using an **optimization algorithm**
- Mixing and matching these modular components give us a lot of new ML methods.

# Supervised Learning Setup

---



In supervised learning:

- There is input  $\mathbf{x} \in \mathcal{X}$ , typically a vector of features (or covariates)
- There is target  $t \in \mathcal{T}$  (also called response, outcome, output, class)
- Objective is to learn a function  $f : \mathcal{X} \rightarrow \mathcal{T}$  such that  $t \approx y = f(\mathbf{x})$  based on some data  $\mathcal{D} = \{(\mathbf{x}^{(i)}, t^{(i)}) \text{ for } i = 1, 2, \dots, N\}$ .

# Linear Regression - Model

---

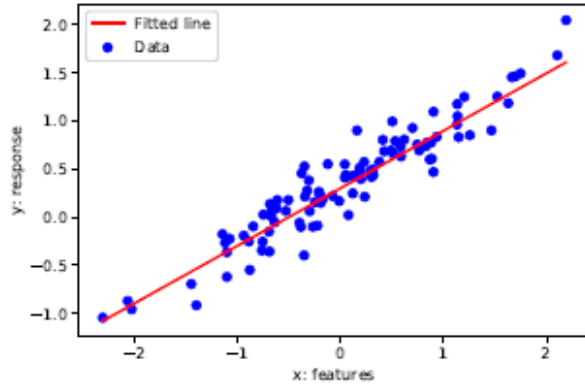
- **Model:** In linear regression, we use a *linear* function of the features  $\mathbf{x} = (x_1, \dots, x_D) \in \mathbb{R}^D$  to make predictions  $y$  of the target value  $t \in \mathbb{R}$ :

$$y = f(\mathbf{x}) = \sum_j w_j x_j + b$$

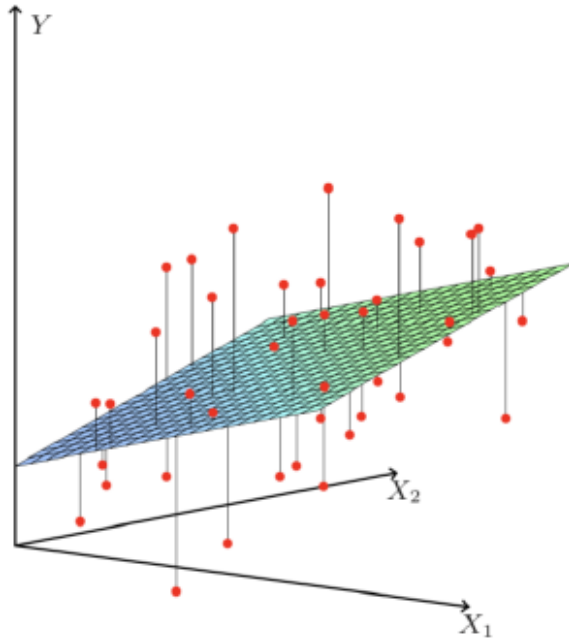
- ▶  $y$  is the prediction
  - ▶  $\mathbf{w}$  is the weights
  - ▶  $b$  is the bias (or intercept)
- $\mathbf{w}$  and  $b$  together are the parameters
- We hope that our prediction is close to the target:  $y \approx t$ .

# What is Linear? 1 feature vs D features

---



- If we have only 1 feature:  
 $y = wx + b$  where  $w, x, b \in \mathbb{R}$ .
- $y$  is linear in  $x$ .



- If we have  $D$  features:  
 $y = \mathbf{w}^\top \mathbf{x} + b$  where  $\mathbf{w}, \mathbf{x} \in \mathbb{R}^D$ ,  
 $b \in \mathbb{R}$
- $y$  is linear in  $\mathbf{x}$ .

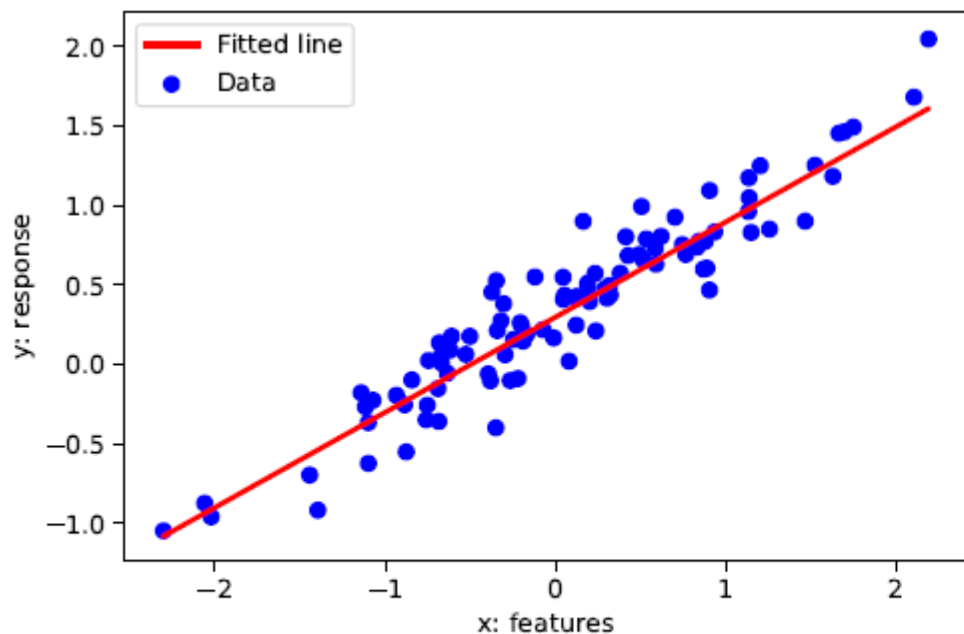
Relation between the prediction  $y$  and inputs  $\mathbf{x}$  is linear in both cases.

# Linear Regression

---

We have a dataset  $\mathcal{D} = \{(\mathbf{x}^{(i)}, t^{(i)}) \text{ for } i = 1, 2, \dots, N\}$  where,

- $\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, \dots, x_D^{(i)})^\top \in \mathbb{R}^D$  are the inputs (e.g. age, height)
- $t^{(i)} \in \mathbb{R}$  is the target or response (e.g. income)
- predict  $t^{(i)}$  with a linear function of  $\mathbf{x}^{(i)}$ :



- $t^{(i)} \approx y^{(i)} = \mathbf{w}^\top \mathbf{x}^{(i)} + b$
- Different  $(\mathbf{w}, b)$  define different lines.
- We want the “best” line  $(\mathbf{w}, b)$ .
- How to quantify “best”?

# Linear Regression - Loss Function

---

- A **loss function**  $\mathcal{L}(y, t)$  defines how bad it is if, for some example  $\mathbf{x}$ , the algorithm predicts  $y$ , but the target is actually  $t$ .
- **Squared error loss function:**

$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$$

- $y - t$  is the **residual**, and we want to make this small in magnitude
- The  $\frac{1}{2}$  factor is just to make the calculations convenient.
- **Cost function:** loss function averaged over all training examples

$$\begin{aligned}\mathcal{J}(\mathbf{w}, b) &= \frac{1}{2N} \sum_{i=1}^N \left( y^{(i)} - t^{(i)} \right)^2 \\ &= \frac{1}{2N} \sum_{i=1}^N \left( \mathbf{w}^\top \mathbf{x}^{(i)} + b - t^{(i)} \right)^2\end{aligned}$$

- Terminology varies. Some call “cost” *empirical* or *average loss*.

# Vectorization

---

- Notation-wise,  $\frac{1}{2N} \sum_{i=1}^N (y^{(i)} - t^{(i)})^2$  gets messy if we expand  $y^{(i)}$ :

$$\frac{1}{2N} \sum_{i=1}^N \left( \sum_{j=1}^D (w_j x_j^{(i)} + b) - t^{(i)} \right)^2$$

- The code equivalent is to compute the prediction using a for loop:

```
y = b
for j in range(M):
    y += w[j] * x[j]
```

- Excessive super/sub scripts are hard to work with, and Python loops are slow, so we **vectorize** algorithms by expressing them in terms of vectors and matrices.

$$\mathbf{w} = (w_1, \dots, w_D)^\top \quad \mathbf{x} = (x_1, \dots, x_D)^\top$$

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

- This is simpler and executes much faster:

```
y = np.dot(w, x) + b
```



# Vectorization

---

Why vectorize?

- The equations, and the code, will be simpler and more readable.  
Gets rid of dummy variables/indices!
- Vectorized code is much faster
  - ▶ Cut down on Python interpreter overhead
  - ▶ Use highly optimized linear algebra libraries (hardware support)
  - ▶ Matrix multiplication very fast on GPU (Graphics Processing Unit)

Switching in and out of vectorized form is a skill you gain with practice

- Some derivations are easier to do element-wise
- Some algorithms are easier to write/understand using for-loops and vectorize later for performance

# Vectorization

---

- We can organize all the training examples into a **design matrix**  $\mathbf{X}$  with one row per training example, and all the targets into the **target vector**  $\mathbf{t}$ .

one feature across  
all training examples

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \mathbf{x}^{(3)\top} \end{pmatrix} = \begin{pmatrix} 8 & 0 & 3 & 0 \\ 6 & -1 & 5 & 3 \\ 2 & 5 & -2 & 8 \end{pmatrix}$$

one training  
example (vector)

- Computing the predictions for the whole dataset:

$$\mathbf{X}\mathbf{w} + b\mathbf{1} = \begin{pmatrix} \mathbf{w}^T \mathbf{x}^{(1)} + b \\ \vdots \\ \mathbf{w}^T \mathbf{x}^{(N)} + b \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{pmatrix} = \mathbf{y}$$

# Vectorization

---

- Computing the squared error cost across the whole dataset:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b\mathbf{1}$$

$$\mathcal{J} = \frac{1}{2N} \|\mathbf{y} - \mathbf{t}\|^2$$

- Sometimes we may use  $\mathcal{J} = \frac{1}{2} \|\mathbf{y} - \mathbf{t}\|^2$ , without a normalizer. This would correspond to the sum of losses, and not the averaged loss. The minimizer does not depend on  $N$  (but optimization might!).
- We can also add a column of 1's to design matrix, combine the bias and the weights, and conveniently write

$$\mathbf{X} = \begin{bmatrix} 1 & [\mathbf{x}^{(1)}]^\top \\ 1 & [\mathbf{x}^{(2)}]^\top \\ \vdots & \vdots \end{bmatrix} \in \mathbb{R}^{N \times (D+1)} \quad \text{and} \quad \mathbf{w} = \begin{bmatrix} b \\ w_1 \\ w_2 \\ \vdots \end{bmatrix} \in \mathbb{R}^{D+1}$$

Then, our predictions reduce to  $\mathbf{y} = \mathbf{X}\mathbf{w}$ .

# Solving the Minimization Problem

---

We defined a cost function. This is what we'd like to minimize.

Two commonly applied mathematical approaches:

- Algebraic, e.g., using inequalities:
  - ▶ to show  $z^*$  minimizes  $f(z)$ , show that  $\forall z, f(z) \geq f(z^*)$
  - ▶ to show that  $a = b$ , show that  $a \geq b$  and  $b \geq a$
- Calculus: minimum of a smooth function (if it exists) occurs at a **critical point**, i.e. point where the derivative is zero.
  - ▶ multivariate generalization: set the partial derivatives to zero (or equivalently the gradient).

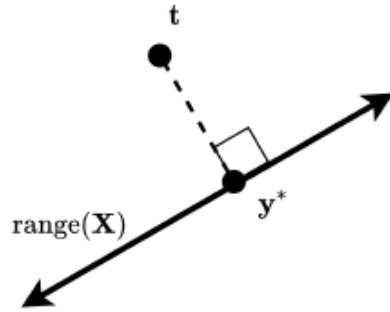
Solutions may be direct or iterative

- Sometimes we can directly find provably optimal parameters (e.g. set the gradient to zero and solve in closed form). We call this a **direct solution**.
- We may also use optimization techniques that iteratively get us closer to the solution. We will get back to this soon.

# Direct Solution I: Linear Algebra

---

- We seek  $\mathbf{w}$  to minimize  $\|\mathbf{X}\mathbf{w} - \mathbf{t}\|^2$ , or equivalently  $\|\mathbf{X}\mathbf{w} - \mathbf{t}\|$
- $\text{range}(\mathbf{X}) = \{\mathbf{X}\mathbf{w} \mid \mathbf{w} \in \mathbb{R}^D\}$  is a  $D$ -dimensional subspace of  $\mathbb{R}^N$ .
- Recall that the closest point  $\mathbf{y}^* = \mathbf{X}\mathbf{w}^*$  in subspace  $\text{range}(\mathbf{X})$  of  $\mathbb{R}^N$  to arbitrary point  $\mathbf{t} \in \mathbb{R}^N$  is found by orthogonal projection.

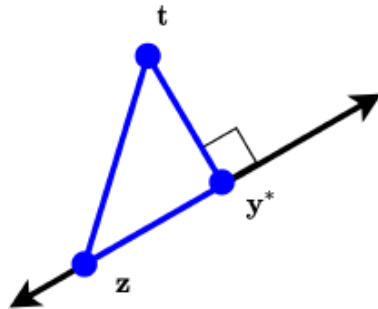


- We have  $(\mathbf{y}^* - \mathbf{t}) \perp \mathbf{X}\mathbf{w}$ ,  $\forall \mathbf{w} \in \mathbb{R}^D$

- Why is  $\mathbf{y}^*$  the closest point to  $\mathbf{t}$ ?

- ▶ Consider any  $\mathbf{z} = \mathbf{X}\mathbf{w}$
- ▶ By Pythagorean theorem and the trivial inequality ( $x^2 \geq 0$ ):

$$\begin{aligned}\|\mathbf{z} - \mathbf{t}\|^2 &= \|\mathbf{y}^* - \mathbf{t}\|^2 + \|\mathbf{y}^* - \mathbf{z}\|^2 \\ &\geq \|\mathbf{y}^* - \mathbf{t}\|^2\end{aligned}$$



# Direct Solution I: Linear Algebra

---

- From the previous slide, we have  $(\mathbf{y}^* - \mathbf{t}) \perp \mathbf{X}\mathbf{w}$ ,  $\forall \mathbf{w} \in \mathbb{R}^D$
- Equivalently, the columns of the design matrix  $\mathbf{X}$  are all orthogonal to  $(\mathbf{y}^* - \mathbf{t})$ , and we have that:

$$\mathbf{X}^\top (\mathbf{y}^* - \mathbf{t}) = \mathbf{0}$$

$$\mathbf{X}^\top \mathbf{X}\mathbf{w}^* - \mathbf{X}^\top \mathbf{t} = \mathbf{0}$$

$$\mathbf{X}^\top \mathbf{X}\mathbf{w}^* = \mathbf{X}^\top \mathbf{t}$$

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

- While this solution is clean and the derivation easy to remember, like many algebraic solutions, it is somewhat ad hoc.
- On the hand, the tools of calculus are broadly applicable to differentiable loss functions...

## Direct Solution II: Calculus

---

- **Partial derivative:** derivative of a multivariate function with respect to one of its arguments.

$$\frac{\partial}{\partial x_1} f(x_1, x_2) = \lim_{h \rightarrow 0} \frac{f(x_1 + h, x_2) - f(x_1, x_2)}{h}$$

- To compute, take the single variable derivative, pretending the other arguments are constant.
- Example: partial derivatives of the prediction  $y$

$$\begin{aligned} \frac{\partial y}{\partial w_j} &= \frac{\partial}{\partial w_j} \left[ \sum_{j'} w_{j'} x_{j'} + b \right] \\ &= x_j \end{aligned}$$

$$\begin{aligned} \frac{\partial y}{\partial b} &= \frac{\partial}{\partial b} \left[ \sum_{j'} w_{j'} x_{j'} + b \right] \\ &= 1 \end{aligned}$$



## Direct Solution II: Calculus

---

- For loss derivatives, apply the **chain rule**:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{d\mathcal{L}}{dy} \frac{\partial y}{\partial w_j} \\ &= \frac{d}{dy} \left[ \frac{1}{2}(y - t)^2 \right] \cdot x_j \\ &= (y - t)x_j\end{aligned}\qquad \begin{aligned}\frac{\partial \mathcal{L}}{\partial b} &= \frac{d\mathcal{L}}{dy} \frac{\partial y}{\partial b} \\ &= y - t\end{aligned}$$

- For cost derivatives, use **linearity** and average over data points:

$$\frac{\partial \mathcal{J}}{\partial w_j} = \frac{1}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) x_j^{(i)} \qquad \frac{\partial \mathcal{J}}{\partial b} = \frac{1}{N} \sum_{i=1}^N y^{(i)} - t^{(i)}$$

- Minimum must occur at a point where partial derivatives are zero.

$$\frac{\partial \mathcal{J}}{\partial w_j} = 0 \quad (\forall j), \qquad \frac{\partial \mathcal{J}}{\partial b} = 0.$$

(if  $\partial \mathcal{J} / \partial w_j \neq 0$ , you could reduce the cost by changing  $w_j$ )



## Direct Solution II: Calculus

---

- The derivation on the previous slide gives a system of linear equations, which we can solve efficiently.
- As is often the case for models and code, however, the solution is easier to characterize if we vectorize our calculus.
- We call the vector of partial derivatives the **gradient**
- Thus, the “gradient of  $f : \mathbb{R}^D \rightarrow \mathbb{R}$ ”, denoted  $\nabla f(\mathbf{w})$ , is:

$$\left( \frac{\partial}{\partial w_1} f(\mathbf{w}), \dots, \frac{\partial}{\partial w_D} f(\mathbf{w}) \right)^\top$$

- The gradient points in the direction of the greatest rate of increase.
- Analogue of second derivative (the “Hessian” matrix):  
 $\nabla^2 f(\mathbf{w}) \in \mathbb{R}^{D \times D}$  is a matrix with  $[\nabla^2 f(\mathbf{w})]_{ij} = \frac{\partial^2}{\partial w_i \partial w_j} f(\mathbf{w})$ .

## Direct Solution II: Calculus

---

- We seek  $\mathbf{w}$  to minimize  $\mathcal{J}(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{t}\|^2$
- Taking the gradient with respect to  $\mathbf{w}$  (see course notes for **additional details**) we get:

$$\nabla_{\mathbf{w}}\mathcal{J}(\mathbf{w}) = \mathbf{X}^\top\mathbf{X}\mathbf{w} - \mathbf{X}^\top\mathbf{t} = \mathbf{0}$$

- We get the same optimal weights as before:

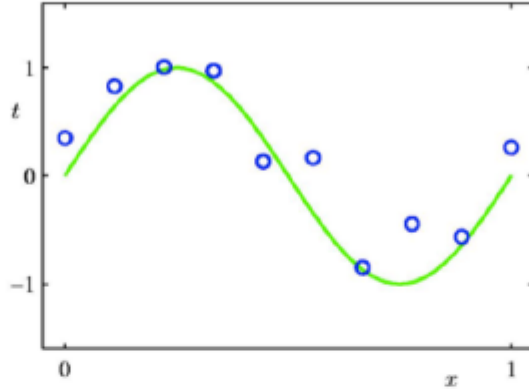
$$\mathbf{w}^* = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{t}$$

- Linear regression is one of only a handful of models in this course that permit direct solution.

# Feature Mapping (Basis Expansion)

---

The relation between the input and output may not be linear.

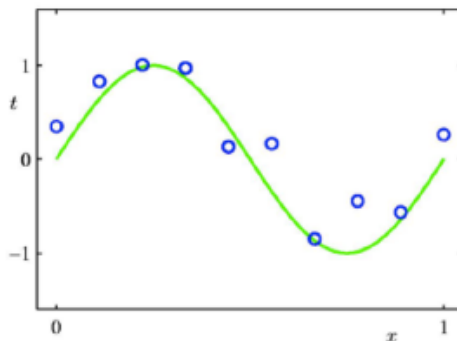


- We can still use linear regression by mapping the input features to another space using **feature mapping** (or **basis expansion**).  
 $\psi(\mathbf{x}) : \mathbb{R}^D \rightarrow \mathbb{R}^d$  and treat the mapped feature (in  $\mathbb{R}^d$ ) as the input of a linear regression procedure.
- Let us see how it works when  $\mathbf{x} \in \mathbb{R}$  and we use a polynomial feature mapping.

# Polynomial Feature Mapping

---

If the relationship doesn't look linear, we can fit a polynomial.



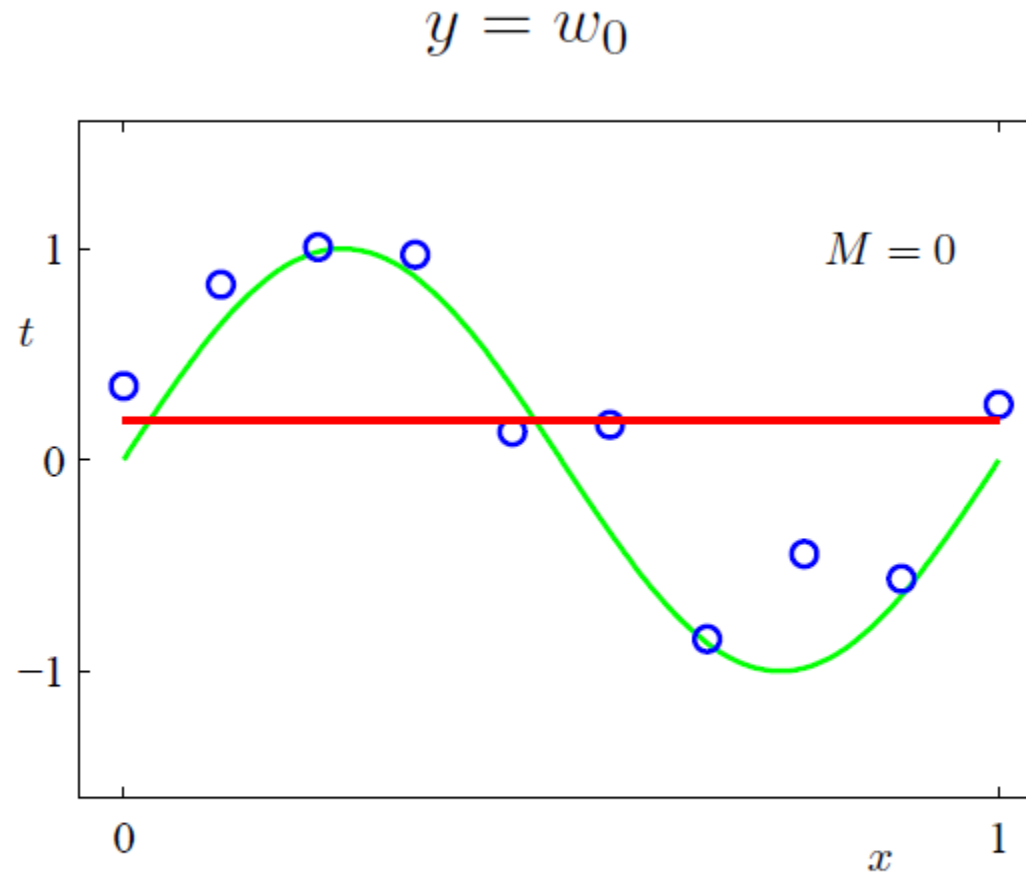
Fit the data using a degree- $M$  polynomial function of the form:

$$y = w_0 + w_1x + w_2x^2 + \dots + w_Mx^M = \sum_{i=0}^M w_i x^i$$

- Here the feature mapping is  $\psi(x) = [1, x, x^2, \dots, x^M]^\top$ .
- We can still use linear regression to find  $\mathbf{w}$  since  $y = \psi(x)^\top \mathbf{w}$  is linear in  $w_0, w_1, \dots$ .
- In general,  $\psi$  can be any function. Another example:  
 $\psi(x) = [1, \sin(2\pi x), \cos(2\pi x), \sin(4\pi x), \dots]^\top$ .

# Polynomial Feature Mapping with $M = 0$

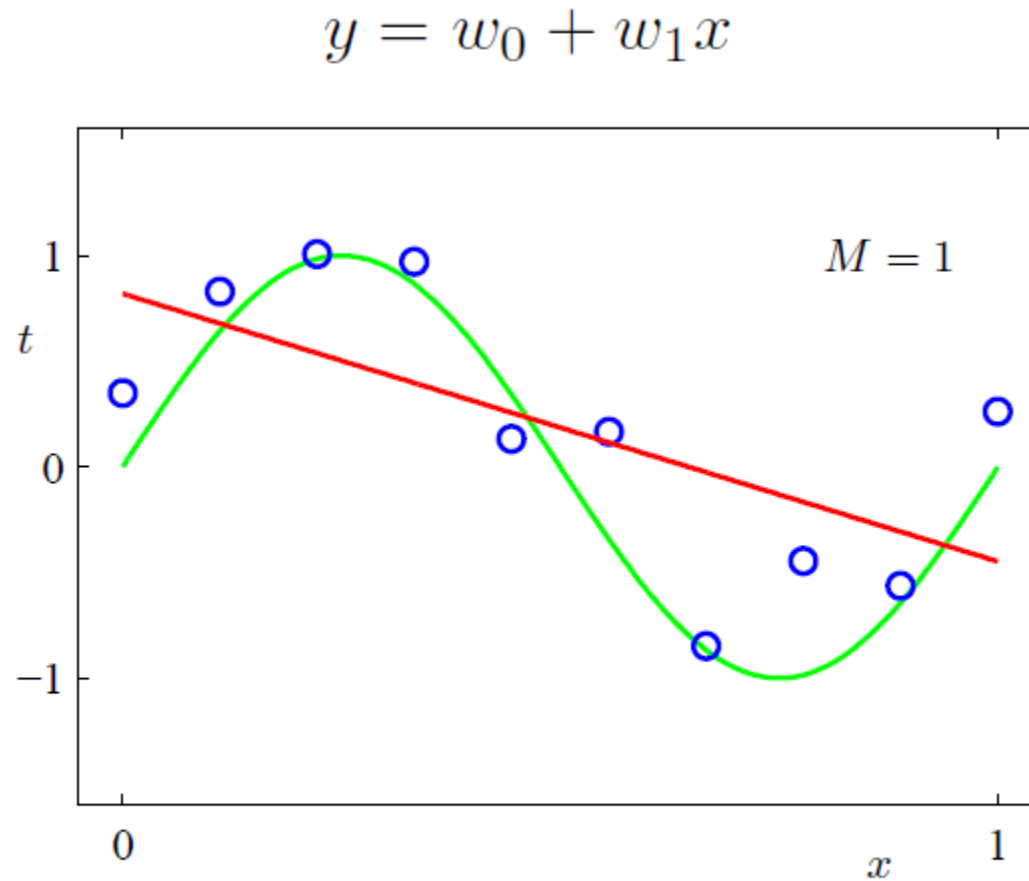
---



-Pattern Recognition and Machine Learning, Christopher Bishop.

# Polynomial Feature Mapping with $M = 1$

---

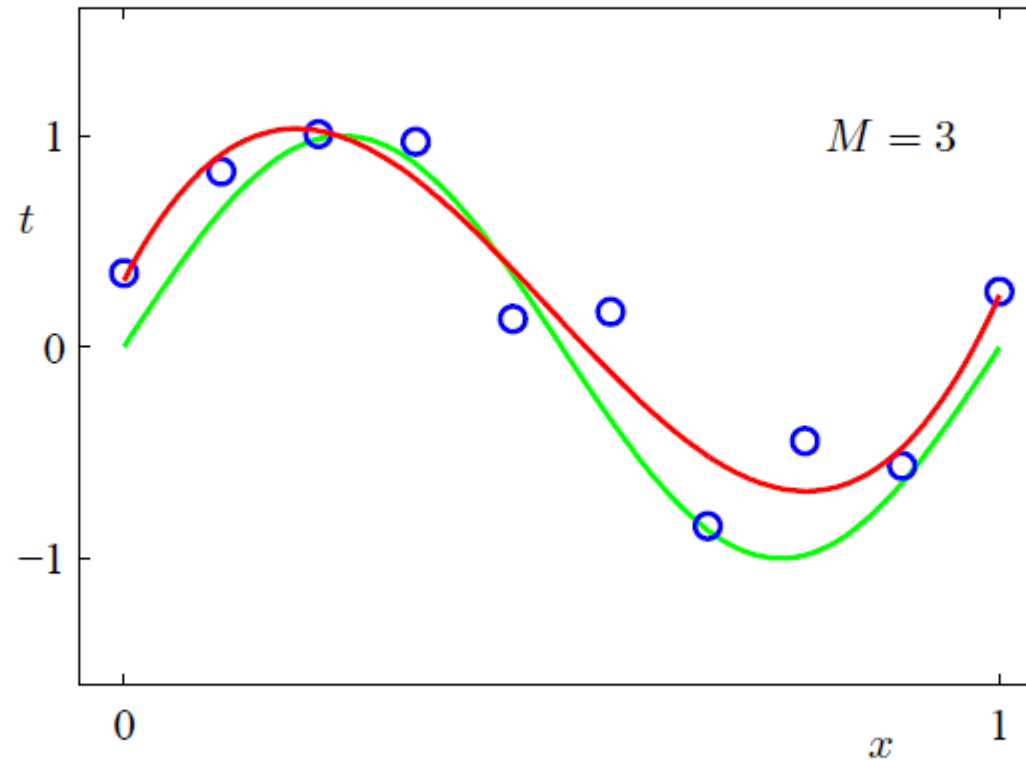


-Pattern Recognition and Machine Learning, Christopher Bishop.

# Polynomial Feature Mapping with $M = 3$

---

$$y = w_0 + w_1x + w_2x^2 + w_3x^3$$

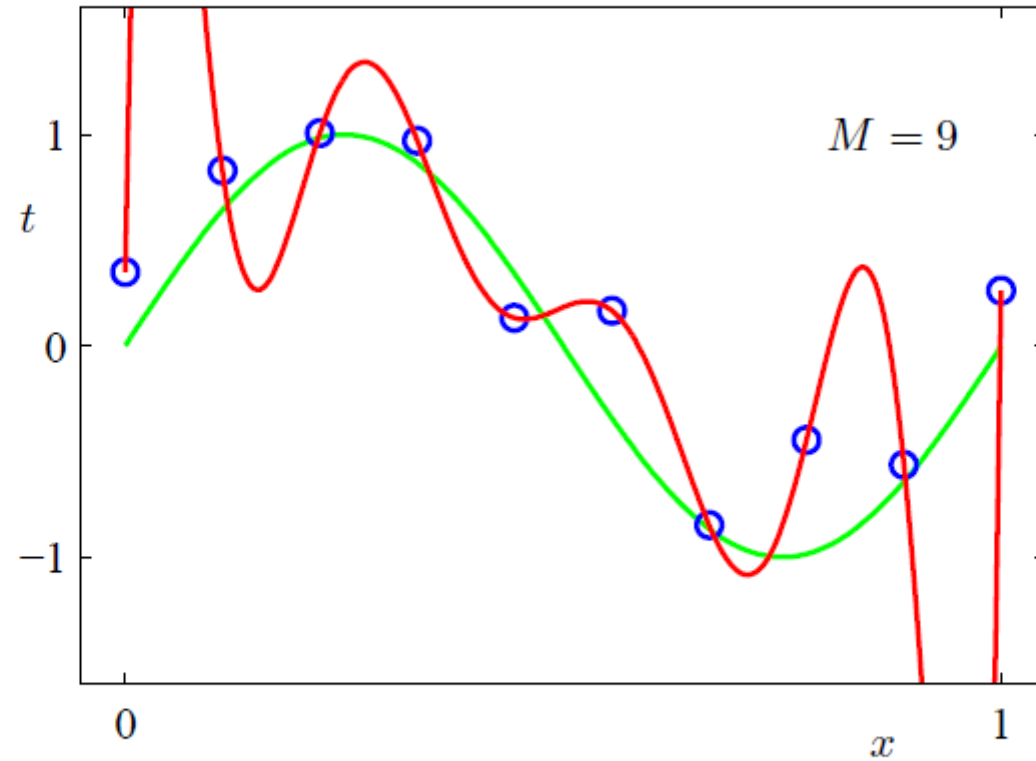


-Pattern Recognition and Machine Learning, Christopher Bishop.

# Polynomial Feature Mapping with $M = 9$

---

$$y = w_0 + w_1x + w_2x^2 + w_3x^3 + \dots + w_9x^9$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

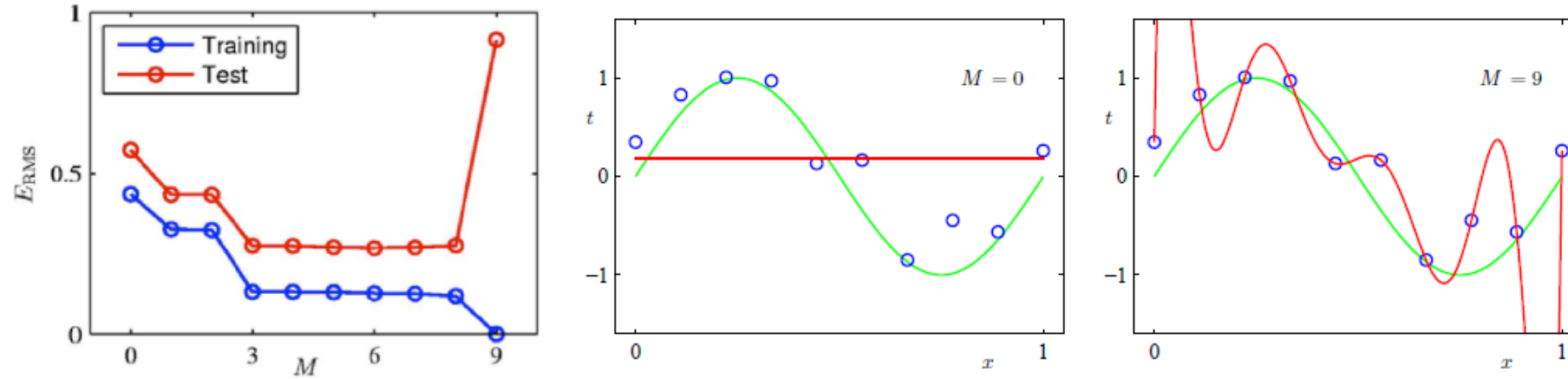


# Model Complexity and Generalization

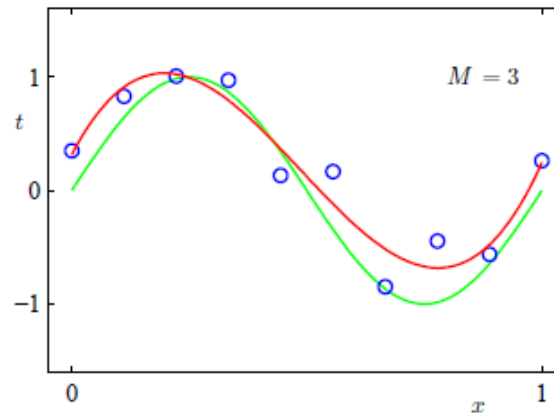
---

Underfitting ( $M=0$ ): model is too simple — does not fit the data.

Overfitting ( $M=9$ ): model is too complex — fits perfectly.



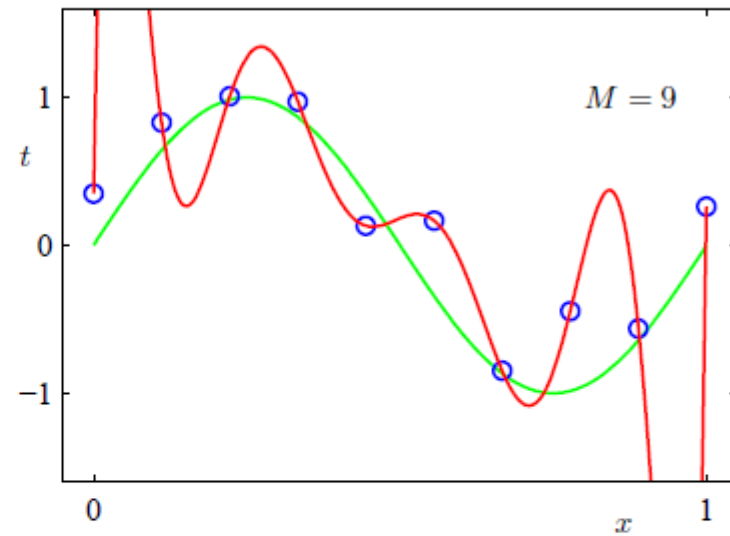
Good model ( $M=3$ ): Achieves small test error (generalizes well).



# Model Complexity and Generalization

---

	$M = 0$	$M = 1$	$M = 3$	$M = 9$
$w_0^*$	0.19	0.82	0.31	0.35
$w_1^*$		-1.27	7.99	232.37
$w_2^*$			-25.43	-5321.83
$w_3^*$			17.37	48568.31
$w_4^*$				-231639.30
$w_5^*$				640042.26
$w_6^*$				-1061800.52
$w_7^*$				1042400.18
$w_8^*$				-557682.99
$w_9^*$				125201.43



- As  $M$  increases, the magnitude of coefficients gets larger.
- For  $M = 9$ , the coefficients have become finely tuned to the data.
- Between data points, the function exhibits large oscillations.

# Regularization

---

- The degree  $M$  of the polynomial controls the model's complexity.
- The value of  $M$  is a hyperparameter for polynomial expansion, just like  $k$  in KNN. We can tune it using a validation set.
- Restricting the number of parameters / basis functions ( $M$ ) is a crude approach to controlling the model complexity.
- Another approach: keep the model large, but **regularize** it
  - ▶ **Regularizer**: a function that quantifies how much we prefer one hypothesis vs. another

## L2 Regularization

---

- We can encourage the weights to be small by choosing as our regularizer the  $L^2$  penalty.

$$\mathcal{R}(\mathbf{w}) = \frac{1}{2} \|\mathbf{w}\|_2^2 = \frac{1}{2} \sum_j w_j^2.$$

- ▶ Note: To be precise, the  $L^2$  norm is Euclidean distance, so we're regularizing the *squared*  $L^2$  norm.
- The regularized cost function makes a tradeoff between fit to the data and the norm of the weights.

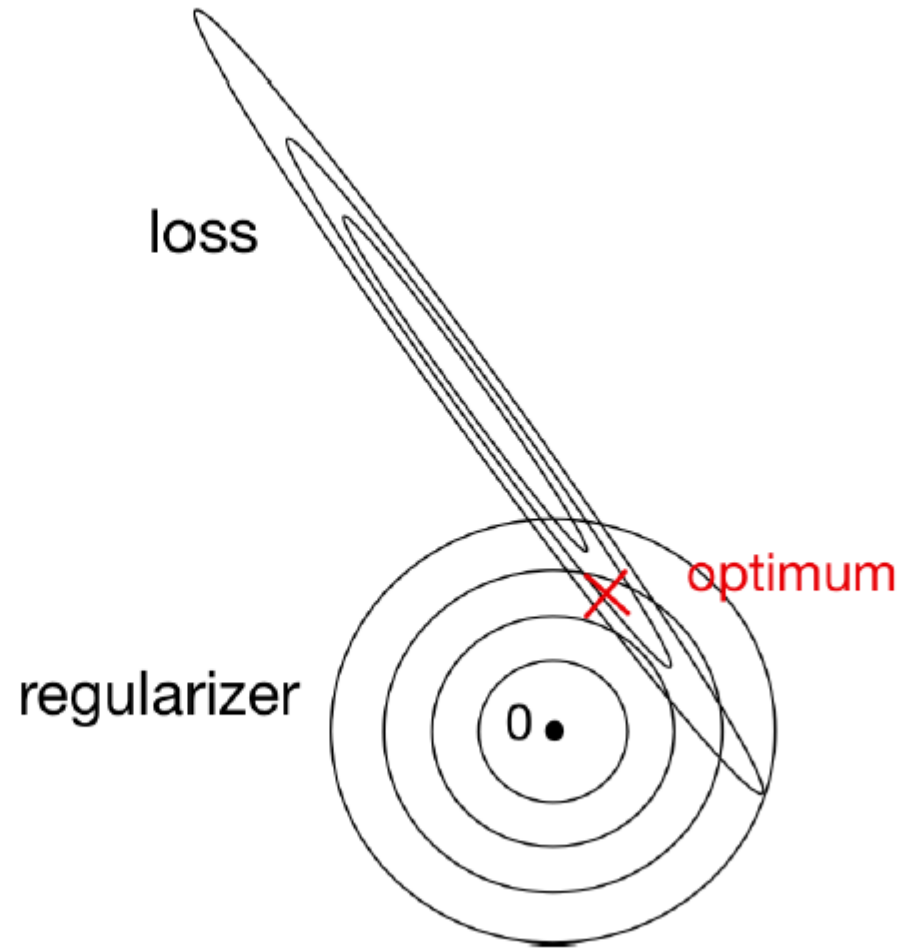
$$\mathcal{J}_{\text{reg}}(\mathbf{w}) = \mathcal{J}(\mathbf{w}) + \lambda \mathcal{R}(\mathbf{w}) = \mathcal{J}(\mathbf{w}) + \frac{\lambda}{2} \sum_j w_j^2$$

- If you fit training data poorly,  $\mathcal{J}$  is large. If your optimal weights have high values,  $\mathcal{R}$  is large.
- Large  $\lambda$  penalizes weight values more.
- Like  $M$ ,  $\lambda$  is a hyperparameter we can tune with a validation set.

# L2 Regularization

---

- The geometric picture:



## L2 Regularized Least Squares: Ridge regression

---

For the least squares problem, we have  $\mathcal{J}(\mathbf{w}) = \frac{1}{2N} \|\mathbf{X}\mathbf{w} - \mathbf{t}\|^2$ .

- When  $\lambda > 0$  (with regularization), regularized cost gives

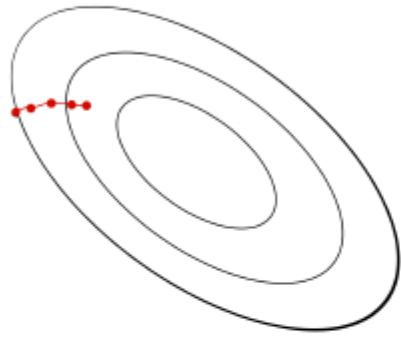
$$\begin{aligned}\mathbf{w}_{\lambda}^{\text{Ridge}} &= \underset{\mathbf{w}}{\operatorname{argmin}} \mathcal{J}_{\text{reg}}(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{2N} \|\mathbf{X}\mathbf{w} - \mathbf{t}\|_2^2 + \frac{\lambda}{2} \|\mathbf{w}\|_2^2 \\ &= (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{t}\end{aligned}$$

- The case  $\lambda = 0$  (no regularization) reduces to least squares solution!
- Note that it is also common to formulate this problem as  $\underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{2} \|\mathbf{X}\mathbf{w} - \mathbf{t}\|_2^2 + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$  in which case the solution is  $\mathbf{w}_{\lambda}^{\text{Ridge}} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{t}$ .

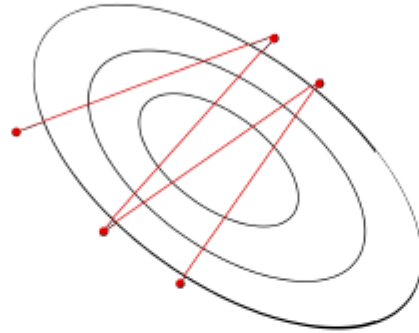
## Conclusion so far

---

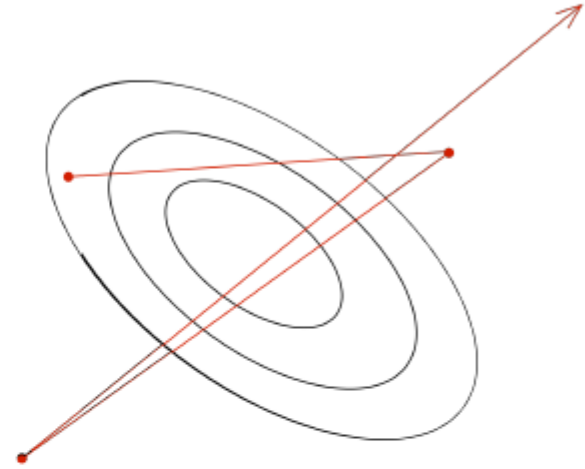
- In gradient descent, the learning rate  $\alpha$  is a hyperparameter we need to tune. Here are some things that can go wrong:



$\alpha$  too small:  
slow progress



$\alpha$  too large:  
oscillations



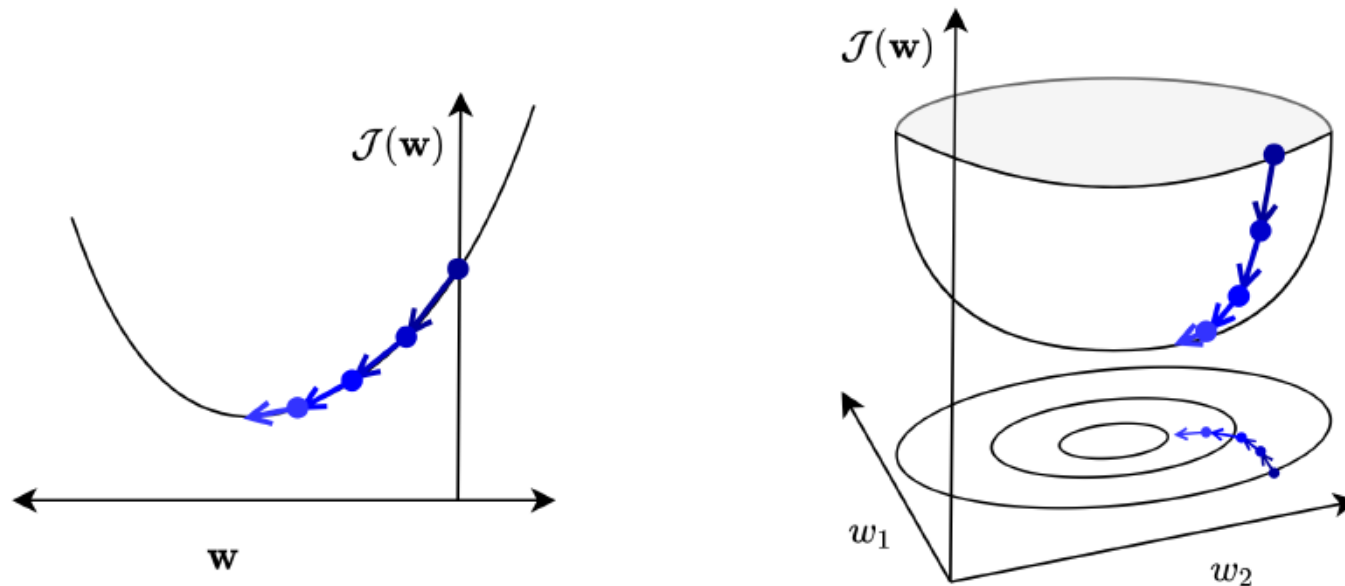
$\alpha$  much too large:  
instability

- Good values are typically between 0.001 and 0.1. You should do a grid search if you want good performance (i.e. try 0.1, 0.03, 0.01, ...).

# Gradient Descent

---

- Now let's see a second way to minimize the cost function which is more broadly applicable: **gradient descent**.
- Many times, we do not have a direct solution: Taking derivatives of  $\mathcal{J}$  w.r.t  $\mathbf{w}$  and setting them to 0 doesn't have an explicit solution.
- Gradient descent is an **iterative algorithm**, which means we apply an update repeatedly until some criterion is met.
- We **initialize** the weights to something reasonable (e.g. all zeros) and repeatedly adjust them in the **direction of steepest descent**.





# Gradient Descent

---

- Observe:
  - ▶ if  $\partial \mathcal{J} / \partial w_j > 0$ , then increasing  $w_j$  increases  $\mathcal{J}$ .
  - ▶ if  $\partial \mathcal{J} / \partial w_j < 0$ , then increasing  $w_j$  decreases  $\mathcal{J}$ .
- The following update always decreases the cost function for small enough  $\alpha$  (unless  $\partial \mathcal{J} / \partial w_j = 0$ ):

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j}$$

- $\alpha > 0$  is a **learning rate** (or step size). The larger it is, the faster  $\mathbf{w}$  changes.
  - ▶ We'll see later how to tune the learning rate, but values are typically small, e.g. 0.01 or 0.0001.
  - ▶ If cost is the sum of  $N$  individual losses rather than their average, smaller learning rate will be needed ( $\alpha' = \alpha/N$ ).

# Gradient Descent

---

- This gets its name from the [gradient](#):

$$\nabla_{\mathbf{w}} \mathcal{J} = \frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix}$$

- This is the direction of fastest increase in  $\mathcal{J}$ .
- Update rule in vector form:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}$$

And for linear regression we have:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) \mathbf{x}^{(i)}$$

- So gradient descent updates  $\mathbf{w}$  in the direction of fastest *decrease*.
- Observe that once it converges, we get a critical point, i.e.  $\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \mathbf{0}$ .

# Gradient Descent for Linear Regression

---

- The squared error loss of linear regression is a convex function.
- Even for linear regression, where there is a direct solution, we sometimes need to use GD.
- Why gradient descent, if we can find the optimum directly?
  - ▶ GD can be applied to a much broader set of models
  - ▶ GD can be easier to implement than direct solutions
  - ▶ For regression in high-dimensional space, GD is more efficient than direct solution
    - ▶ Linear regression solution:  $(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{t}$
    - ▶ Matrix inversion is an  $\mathcal{O}(D^3)$  algorithm
    - ▶ Each GD update costs  $\mathcal{O}(ND)$
    - ▶ Or less with stochastic GD (SGD, in a few slides)
    - ▶ Huge difference if  $D \gg 1$

# Gradient Descent under the L2 Regularization

---

- Gradient descent update to minimize  $\mathcal{J}$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial}{\partial \mathbf{w}} \mathcal{J}$$

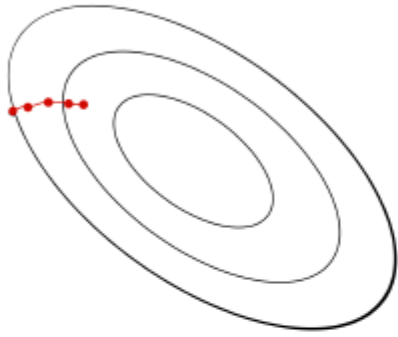
- The gradient descent update to minimize the  $L^2$  regularized cost  $\mathcal{J} + \lambda \mathcal{R}$  results in **weight decay**:

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \alpha \frac{\partial}{\partial \mathbf{w}} (\mathcal{J} + \lambda \mathcal{R}) \\ &= \mathbf{w} - \alpha \left( \frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \mathbf{w}} \right) \\ &= \mathbf{w} - \alpha \left( \frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \mathbf{w} \right) \\ &= (1 - \alpha \lambda) \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}} \end{aligned}$$

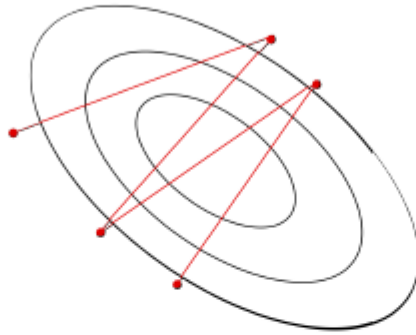
# Learning Rate (Step Size)

---

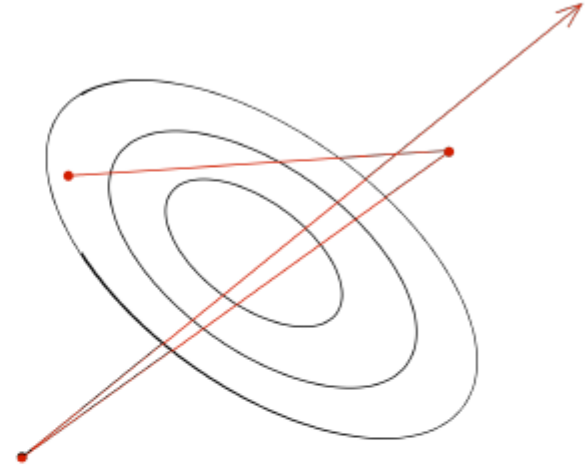
- In gradient descent, the learning rate  $\alpha$  is a hyperparameter we need to tune. Here are some things that can go wrong:



$\alpha$  too small:  
slow progress



$\alpha$  too large:  
oscillations



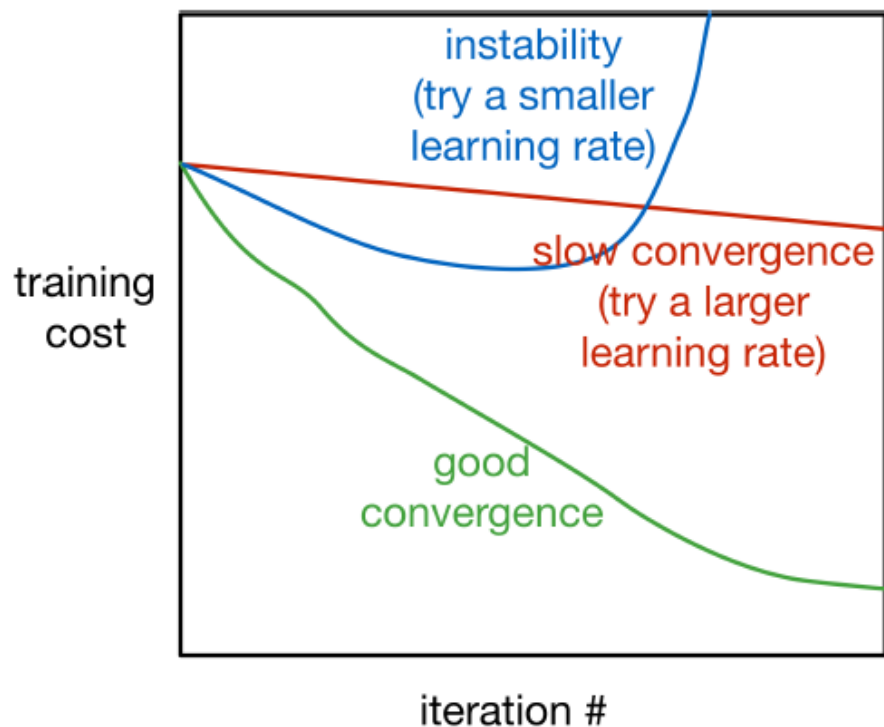
$\alpha$  much too large:  
instability

- Good values are typically between 0.001 and 0.1. You should do a grid search if you want good performance (i.e. try 0.1, 0.03, 0.01, ...).

# Training Curves

---

- To diagnose optimization problems, it's useful to look at **training curves**: plot the training cost as a function of iteration.



- Warning: in general, it's very hard to tell from the training curves whether an optimizer has converged. They can reveal major problems, but they can't guarantee convergence.

# Stochastic Gradient Descent

---

- So far, the cost function  $\mathcal{J}$  has been the average loss over the training examples:

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^N \mathcal{L}^{(i)} = \frac{1}{N} \sum_{i=1}^N \mathcal{L}(y(\mathbf{x}^{(i)}, \boldsymbol{\theta}), t^{(i)}).$$

( $\boldsymbol{\theta}$  denotes the parameters; e.g., in linear regression,  $\boldsymbol{\theta} = (\mathbf{w}, b)$ )

- By linearity,

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}} = \frac{1}{N} \sum_{i=1}^N \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}.$$

- Computing the gradient requires summing over *all* of the training examples. This is known as **batch training**.
- Batch training is impractical if you have a large dataset  $N \gg 1$  (e.g. millions of training examples)!



# Stochastic Gradient Descent

---

- **Stochastic gradient descent (SGD)**: update the parameters based on the gradient for a single training example,

1— Choose  $i$  uniformly at random,

2— 
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}$$

- Cost of each SGD update is independent of  $N$ !
- SGD can make significant progress before even seeing all the data!
- Mathematical justification: if you sample a training example uniformly at random, the stochastic gradient is an **unbiased estimate** of the batch gradient:

$$\mathbb{E} \left[ \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}} \right] = \frac{1}{N} \sum_{i=1}^N \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}}.$$



# Stochastic Gradient Descent

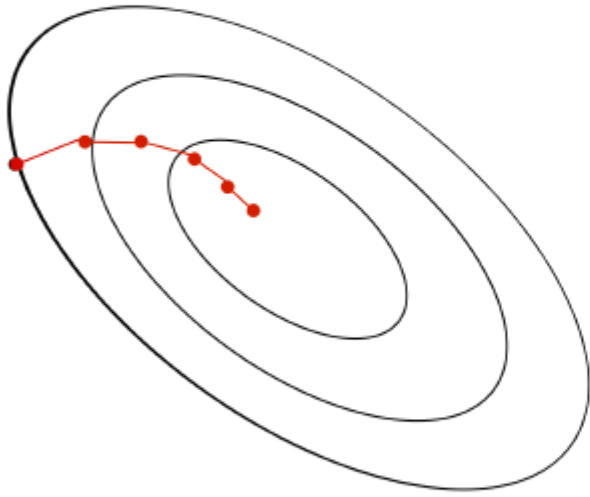
---

- Problems with using single training example to estimate gradient:
  - ▶ Variance in the estimate may be high
  - ▶ We can't exploit efficient vectorized operations
- Compromise approach:
  - ▶ compute the gradients on a randomly chosen medium-sized set of training examples  $\mathcal{M} \subset \{1, \dots, N\}$ , called a **mini-batch**.
- Stochastic gradients computed on larger mini-batches have smaller variance.
- The mini-batch size  $|\mathcal{M}|$  is a hyperparameter that needs to be set.
  - ▶ Too large: requires more compute; e.g., it takes more memory to store the activations, and longer to compute each gradient update
  - ▶ Too small: can't exploit vectorization, has high variance
  - ▶ A reasonable value might be  $|\mathcal{M}| = 100$ .

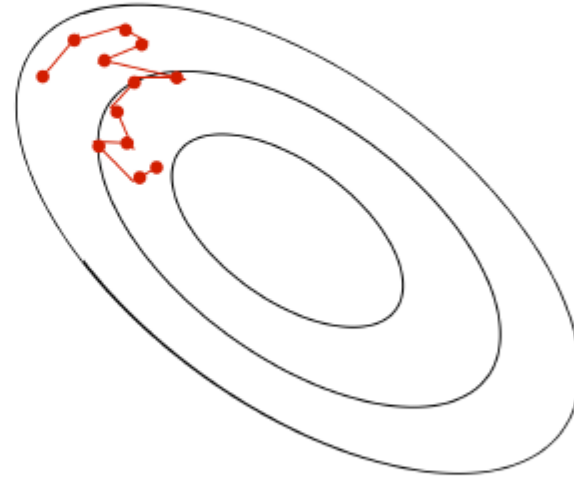
# Stochastic Gradient Descent

---

- Batch gradient descent moves directly downhill (locally speaking).
- SGD takes steps in a noisy direction, but moves downhill on average.



batch gradient descent



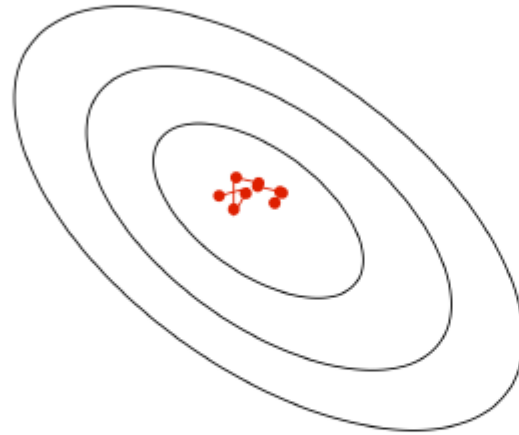
stochastic gradient descent

# SGD Learning Rate

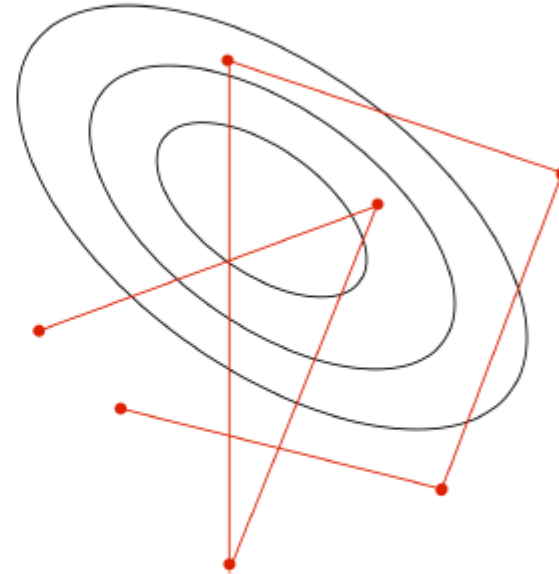
---

- In stochastic training, the learning rate also influences the **fluctuations** due to the stochasticity of the gradients.

small learning rate



large learning rate



- Typical strategy:
  - ▶ Use a large learning rate early in training so you can get close to the optimum
  - ▶ Gradually decay the learning rate to reduce the fluctuations

# Conclusion

---

- In this lecture, we looked at linear regression, which exemplifies a modular approach that will be used throughout this course:
  - ▶ choose a **model** describing the relationships between variables of interest (**linear**)
  - ▶ define a **loss function** quantifying how bad the fit to the data is (**squared error**)
  - ▶ choose a **regularizer** to control the model complexity/overfitting ( $L^2$ ,  $L^p$  **regularization**)
  - ▶ fit/optimize the model (**gradient descent**, **stochastic gradient descent**, **convexity**)
- By mixing and matching these modular components, we can obtain new ML methods.
- Next lecture: apply this framework to classification