EE2211 Introduction to Machine Learning

Lecture 8

Semester 2
2024/2025

Yueming Jin
ymjin@nus.edu.sg

Electrical and Computer Engineering Department
National University of Singapore

Course Contents

- Introduction and Preliminaries (Xinchao)
  - Introduction
  - Data Engineering
  - Introduction to Probability and Statistics
- Fundamental Machine Learning Algorithms I (Yueming)
  - Systems of linear equations
  - Least squares, Linear regression
  - Ridge regression, Polynomial regression
- Fundamental Machine Learning Algorithms II (Yueming)
  - Over-fitting, bias/variance trade-off
  - Optimization, Gradient descent
  - Decision Trees, Random Forest
- Performance and More Algorithms (Xinchao)
  - Performance Issues
  - K-means Clustering
  - Neural Networks

Fundamental ML Algorithms: Optimization, Gradient Descent

Module III Contents

- Overfitting, underfitting and model complexity
- Regularization
- Bias-variance trade-off
- Loss function
- Optimization
- Gradient descent
- Decision trees
- Random forest

Review

- Supervised learning: given feature(s) x, we want to predict target y
- Most supervised learning algorithms can be formulated as the following optimization problem

$$\underset{\mathbf{w}}{\operatorname{argmin}}\, \mathbf{Data\text{-}Loss}(\mathbf{w}) + \lambda \mathbf{Regularization}(\mathbf{w})$$

- **Data-Loss(w)** quantifies fitting error to training set given parameters **w:** smaller error => better fit to training data
- **Regularization(w)** penalizes more complex models
- For example, in the case of polynomial regression (previous lectures):

$$\underset{\mathbf{w}}{\operatorname{argmin}} \underbrace{(\mathbf{P}\mathbf{w} - \mathbf{y})^T(\mathbf{P}\mathbf{w} - \mathbf{y})}_{\textbf{Data-Loss(w)}} + \underbrace{\lambda \mathbf{w}^T\mathbf{w}}_{\textbf{Reg(w)}}$$

Review

- For polynomial regression (previous lectures)

$$\underset{\mathbf{w}}{\operatorname{argmin}}\, C(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}}\, (\mathbf{P}\mathbf{w}-\mathbf{y})^T(\mathbf{P}\mathbf{w}-\mathbf{y}) + \lambda \mathbf{w}^T\mathbf{w}$$

cost function

$$= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{m} (\mathbf{p}_i^T\mathbf{w} - y_i)^2 + \lambda \mathbf{w}^T\mathbf{w}$$

$\mathbf{p}_i^T\mathbf{w}$ is prediction of i-th training sample

y_i is target of i-th training sample

(ground truth, given data)

Review

- For polynomial regression (previous lectures)

$$\underset{\mathbf{w}}{\operatorname{argmin}}\, C(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}}\, (\mathbf{P}\mathbf{w}-\mathbf{y})^T(\mathbf{P}\mathbf{w}-\mathbf{y}) + \lambda \mathbf{w}^T\mathbf{w}$$

$$= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{m} (\mathbf{p}_i^T\mathbf{w} - y_i)^2 + \lambda \mathbf{w}^T\mathbf{w}$$

- Linear regression with 2 features, $\mathbf{p}_i = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}_i$ (dimension: 3)
  - 1 ← Bias/Offset
  - x_1 ← Feature 1 of i-th sample
  - x_2 ← Feature 2 of i-th sample

  ↓ x_1, x_2

- Quadratic regression with 1 feature, $\mathbf{p}_i = \begin{bmatrix} 1 \\ x \\ x^2 \end{bmatrix}_i$ (dimension: 3)
  - 1 ← Bias/Offset
  - x, x^2 ← x is feature of i-th sample

  ↓ x_1, x_1^2

Loss Function & Learning Model

- For polynomial regression (previous lectures)

$$\begin{aligned}\underset{\mathbf{w}}{\operatorname{argmin}} C(\mathbf{w}) &= \underset{\mathbf{w}}{\operatorname{argmin}} (\mathbf{P}\mathbf{w}-\mathbf{y})^T(\mathbf{P}\mathbf{w}-\mathbf{y}) + \lambda \mathbf{w}^T\mathbf{w} \\ &= \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{m} (\mathbf{p}_i^T\mathbf{w} - y_i)^2 + \lambda \mathbf{w}^T\mathbf{w}\end{aligned}$$

- Let $f(\mathbf{x}_i, \mathbf{w})$ be the prediction of target y_i from features $\mathbf{x}_i$ for i-th training sample. For example, suppose $f(\mathbf{x}_i, \mathbf{w}) = \mathbf{p}_i^T\mathbf{w}$, then above becomes

$$\underset{\mathbf{w}}{\operatorname{argmin}} C(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{m} (f(\mathbf{x}_i, \mathbf{w}) - y_i)^2 + \lambda \mathbf{w}^T\mathbf{w}$$

y predict

- Let $L(f(\mathbf{x}_i, \mathbf{w}), y_i)$ be the penalty for predicting $f(\mathbf{x}_i, \mathbf{w})$ when true value is y_i. For example, suppose $L(f(\mathbf{x}_i, \mathbf{w}), y_i) = (f(\mathbf{x}_i, \mathbf{w}) - y_i)^2$, then above becomes

$$\underset{\mathbf{w}}{\operatorname{argmin}} C(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{m} L(f(\mathbf{x}_i, \mathbf{w}), y_i) + \lambda \mathbf{w}^T\mathbf{w}$$

y predict y true

Loss Function & Learning Model

- From previous slide

λ	Effect on Weights w_j	Effect on Model
Small λ	Weights remain large	Model fits data well, but may overfit
Moderate λ	Weights are small but meaningful	Model generalizes well
Very large λ	Weights shrink too much (close to 0)	Model is underfitting

$$\underset{\mathbf{w}}{\operatorname{argmin}}\, C(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{m} L(f(\mathbf{x}_i, \mathbf{w}), y_i) + \lambda \mathbf{w}^T \mathbf{w}$$

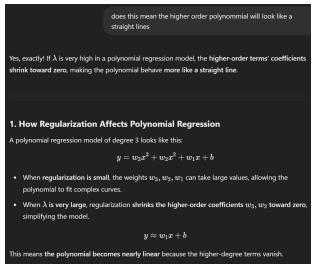
If lambda → 0, regularization term dominates.

- To make it even more general, we can write

$$\underset{\mathbf{w}}{\operatorname{argmin}}\, C(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{m} L(f(\mathbf{x}_i, \mathbf{w}), y_i) + \lambda R(\mathbf{w})$$

ypred ytrue

Cost Function Loss Function Learning Model Regularization


does this mean the higher order polynomial will look like a straight lines

Yes, exactly! If λ is very high in a polynomial regression model, the **higher-order terms' coefficients shrink toward zero**, making the polynomial behave **more like a straight line**.

1. How Regularization Affects Polynomial Regression

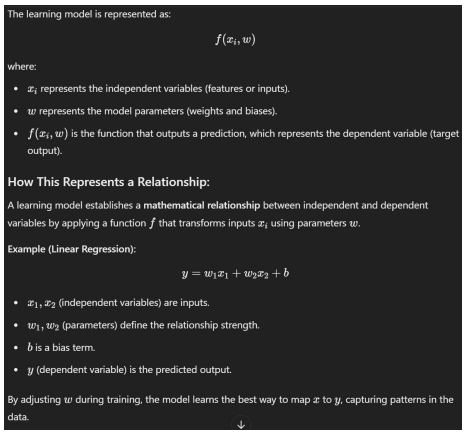
A polynomial regression model of degree 3 looks like this:

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

- When **regularization is small**, the weights w_3, w_2, w_1 can take large values, allowing the polynomial to fit complex curves.
- When λ **is very large**, regularization **shrinks the higher-order coefficients** w_3, w_2 **toward zero**, simplifying the model.

$$y \approx w_1x + b$$

This means **the polynomial becomes nearly linear** because the higher-degree terms vanish.


The learning model is represented as:

$$f(x_i, w)$$

where:

- x_i represents the independent variables (features or inputs).
- w represents the model parameters (weights and biases).
- $f(x_i, w)$ is the function that outputs a prediction, which represents the dependent variable (target output).

How This Represents a Relationship:

A learning model establishes a **mathematical relationship** between independent and dependent variables by applying a function f that transforms inputs x_i using parameters w.

Example (Linear Regression):

$$y = w_1x_1 + w_2x_2 + b$$

- x_1, x_2 (independent variables) are inputs.
- w_1, w_2 (parameters) define the relationship strength.
- b is a bias term.
- y (dependent variable) is the predicted output.

By adjusting w during training, the model learns the best way to map x to y, capturing patterns in the data.

Learning model is like an educated guess on the relationship between the dependent and independent variables.

Loss function evaluates performance of the model by comparing the prediction with the ground truth.

Regularization term reduces curvature of cost function by suppressing magnitude of the weight vector (w)

Building Blocks of ML algorithms

- From previous slide

$$\underset{\mathbf{w}}{\operatorname{argmin}}\, C(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{m} L(f(\mathbf{x}_i, \mathbf{w}), y_i) + \lambda \mathbf{w}^T \mathbf{w}$$

- To make it even more general, we can write

$$\underset{\mathbf{w}}{\operatorname{argmin}}\, C(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{m} L(f(\mathbf{x}_i, \mathbf{w}), y_i) + \lambda R(\mathbf{w})$$

- Learning model f reflects our belief about the relationship between the features $\mathbf{x}_i$ & target y_i
- Loss function L is the penalty for predicting $f(\mathbf{x}_i, \mathbf{w})$ when the true value is y_i
- Regularization R encourages less complex models
- Cost function C is the final optimization criterion we want to minimize
- Optimization routine to find solution to cost function → to find the "best" w

Motivation for Gradient Descent

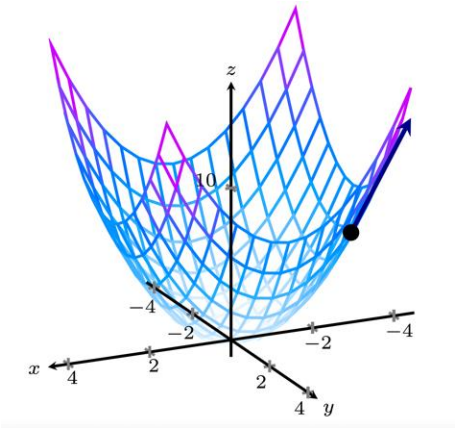
- Different learning function f, loss function L & regularization R give rise to different learning algorithms
- In polynomial regression (previous lectures), optimal $\mathbf{w}$ can be written with the following "closed-form" formula (primal solution): → Lecture 6 pg 14 (over-determined), m>d

$$\hat{\mathbf{w}} = (\mathbf{P}_{train}^T \mathbf{P}_{train} + \lambda \mathbf{I})^{-1} \mathbf{P}_{train}^T \mathbf{y}_{train}$$

- For other learning function f, loss function L & regularization R, optimizing $C(\mathbf{w})$ might not be so easy
- Usually have to estimate $\mathbf{w}$ iteratively with some algorithm
- Optimization workhorse for modern machine learning is gradient descent

Gradient Descent Algorithm

- Suppose we want to minimize $C(\mathbf{w})$ with respect to $\mathbf{w} = [w_1, \cdots, w_d]^T$
- Gradient $\nabla_{\mathbf{w}} C(\mathbf{w}) = \begin{pmatrix} \frac{\partial C}{\partial w_1} \\ \frac{\partial C}{\partial w_2} \\ \vdots \\ \frac{\partial C}{\partial w_d} \end{pmatrix}$ ∴ Differentiation of scalar value w.r.t vector gives vector of dimension $d \times 1$



  - $\nabla_{\mathbf{w}} C(\mathbf{w})$ is vector & function of $\mathbf{w}$
  - $\nabla_{\mathbf{w}} C(\mathbf{w})$ is direction at $\mathbf{w}$ where C is increasing most rapidly, so $-\nabla_{\mathbf{w}} C(\mathbf{w})$ is direction at $\mathbf{w}$ where C is decreasing most rapidly

****

- Gradient Descent:

  Initialize $\mathbf{w}_0$ and learning rate η; (0.1, 0.001, etc)
  while *true* **do**
    Compute $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \eta \nabla_{\mathbf{w}} C(\mathbf{w}_k)$
    if *converge* **then**
      return $\mathbf{w}_{k+1}$
    end
  end

the learning rate

According to multi-variable calculus, if eta is not too big, then $C(\mathbf{w}_{k+1}) < C(\mathbf{w}_k)$ => we get better $\mathbf{w}$ after each iteration

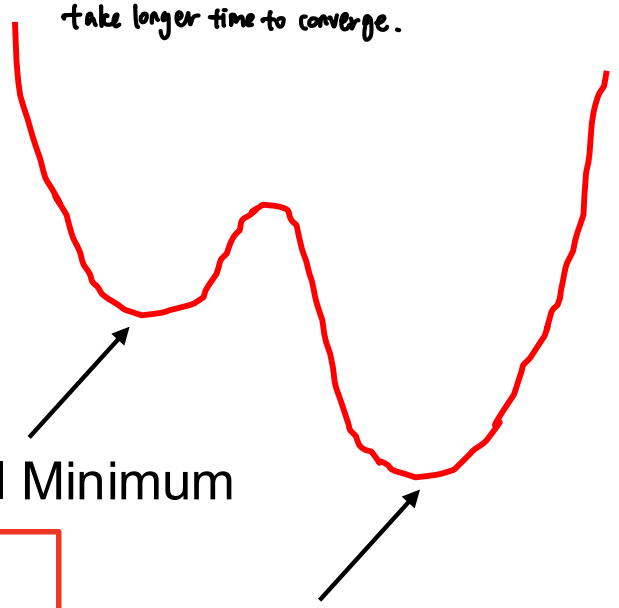
Gradient Descent Algorithm

- Gradient Descent:

Initialize $\mathbf{w}_0$ and learning rate η;
while *true* **do**
  Compute $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k - \eta \nabla_{\mathbf{w}} C(\mathbf{w}_k)$
  if *converge* **then**
    return $\mathbf{w}_{k+1}$
  end
end

- Possible convergence criteria
  - Set maximum iteration k
  - Check percentage or absolute change in C below a threshold
  - Check percentage or absolute change in $\mathbf{w}$ below a threshold
- Gradient descent can only find local minimum
  - Because gradient = 0 at local minimum, so $\mathbf{w}$ won't change after that
- Many variations of gradient descent, e.g., change how gradient is computed or learning rate η decreases with increasing k

need to choose good learning rate, if learning rate too high, it will most likely oscillate and not converge and if too small, take longer time to converge.



why need to decrease learning rate when increasing no. of iterations K?
↓
may not converge

Find x to minimize $g(x) = x^2$ ****

not bolded (scalar)

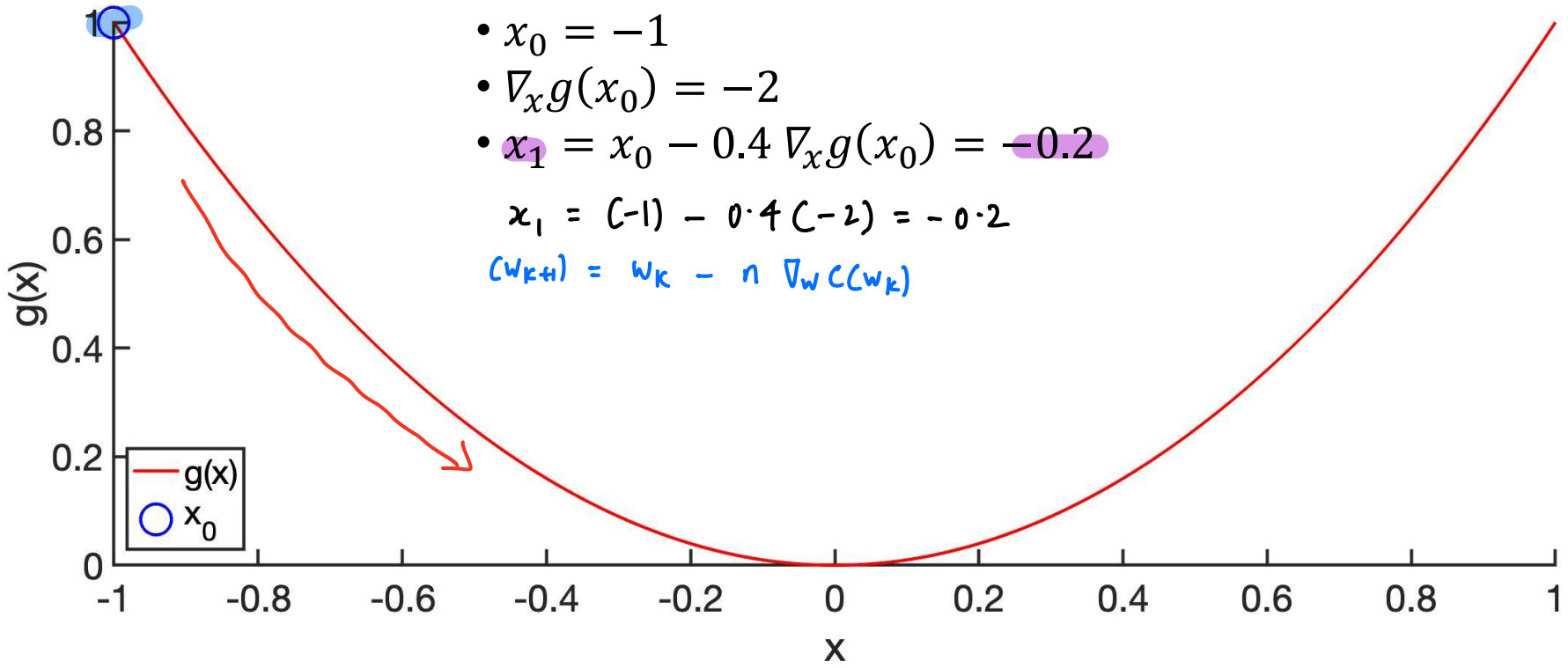
- Obviously minimum corresponds to $x = 0$, but let's do gradient descent

$\frac{\partial g(x)}{\partial x} = 2x$

① – Gradient $\nabla_x g(x) = 2x$

② – Initialize $x_0 = -1$, learning rate $\eta = 0.4$

③ – At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



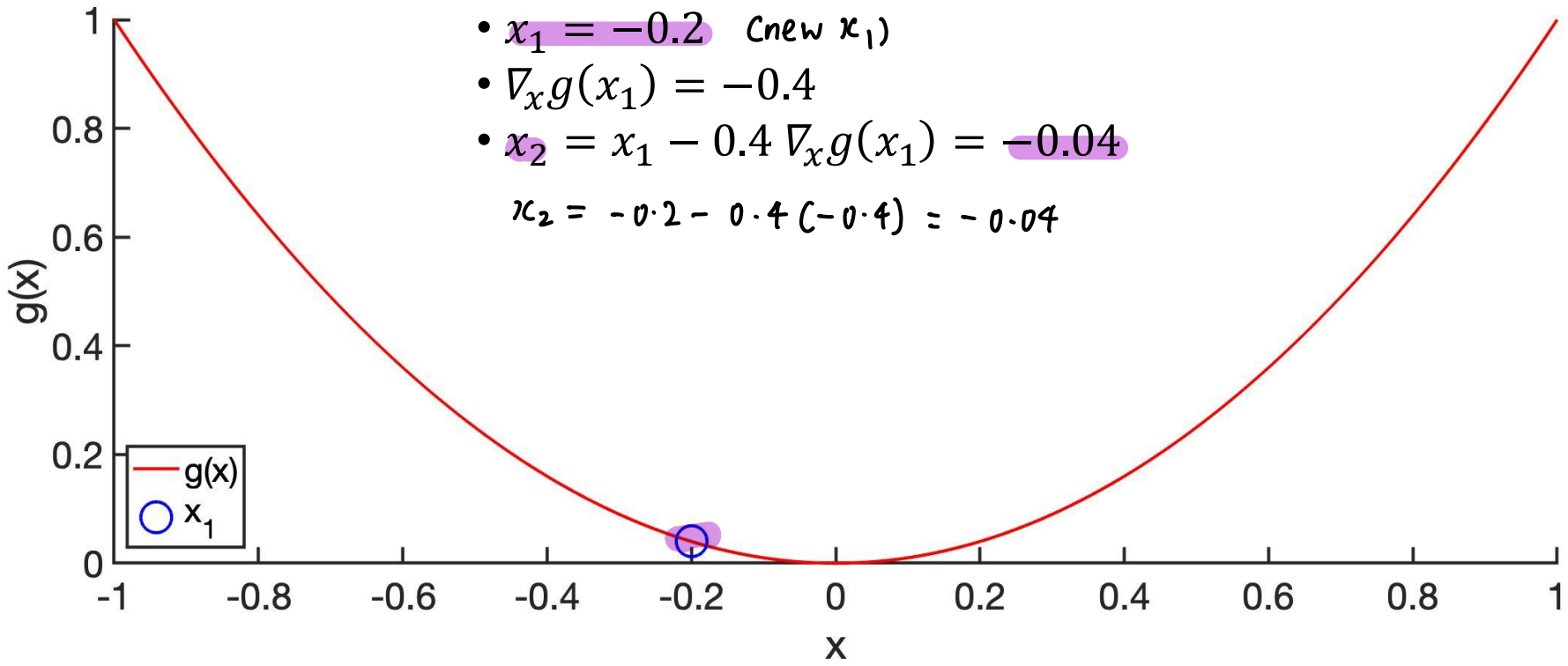
- $x_0 = -1$
- $\nabla_x g(x_0) = -2$
- $x_1 = x_0 - 0.4\,\nabla_x g(x_0) = -0.2$

$x_1 = (-1) - 0.4(-2) = -0.2$

$(w_{k+1}) = w_k - \eta \nabla_w C(w_k)$

Find x to minimize $g(x) = x^2$

- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.4$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$

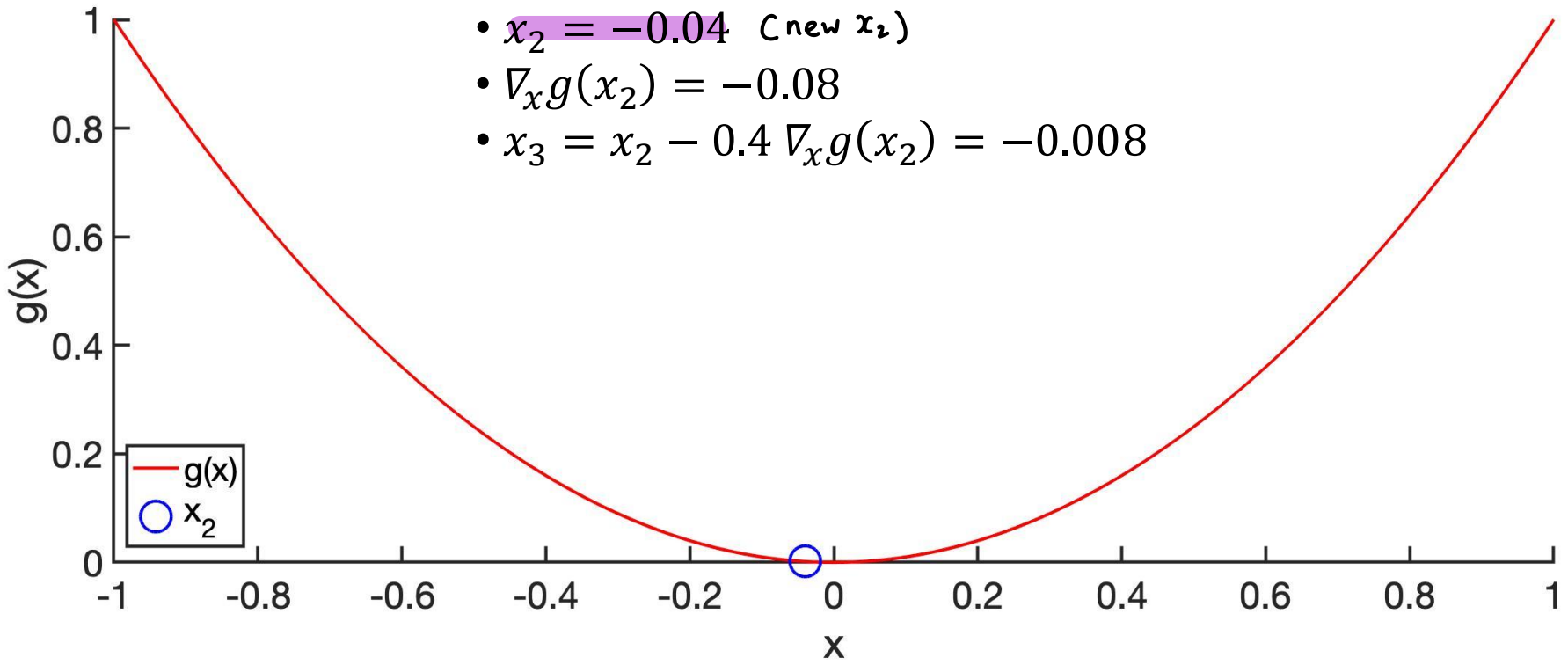


- $x_1 = -0.2$ (new x_1)
- $\nabla_x g(x_1) = -0.4$
- $x_2 = x_1 - 0.4\, \nabla_x g(x_1) = -0.04$

$x_2 = -0.2 - 0.4(-0.4) = -0.04$

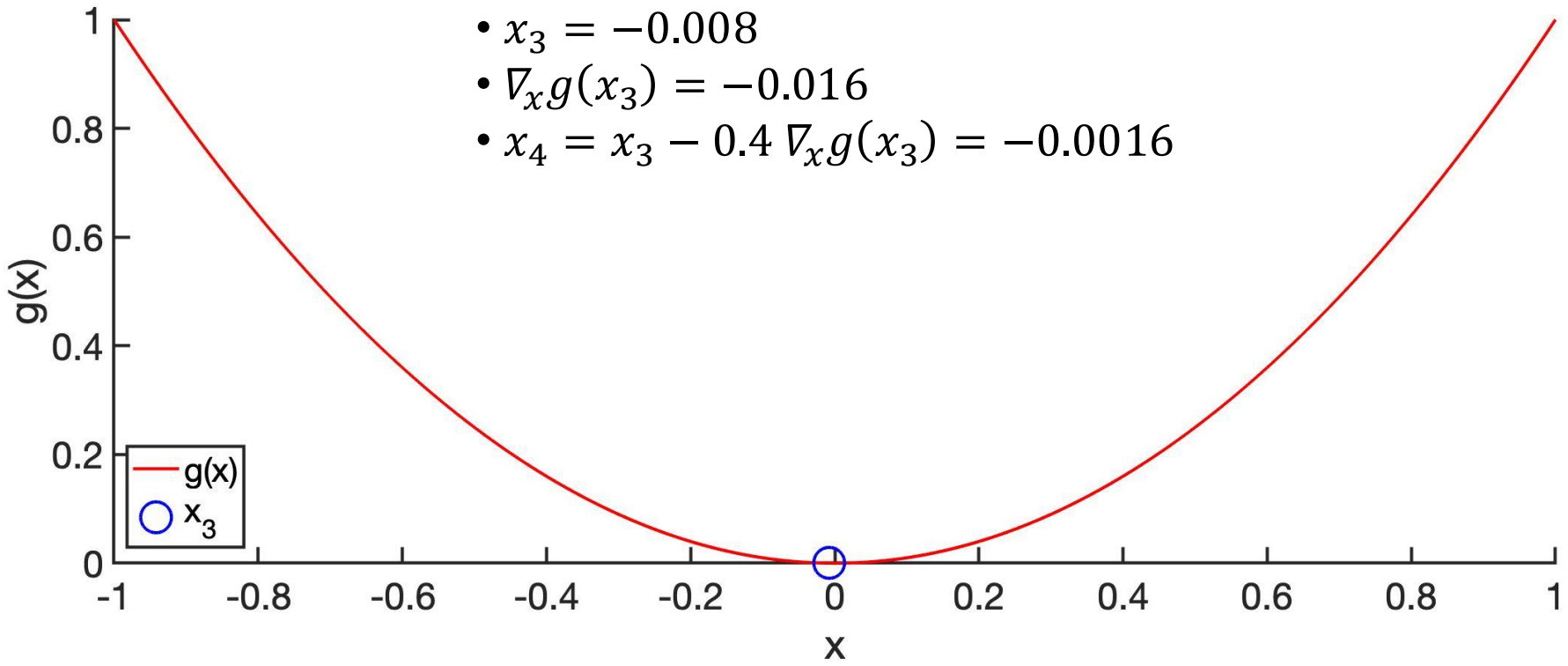
Find x to minimize $g(x) = x^2$

- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.4$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



Find x to minimize $g(x) = x^2$

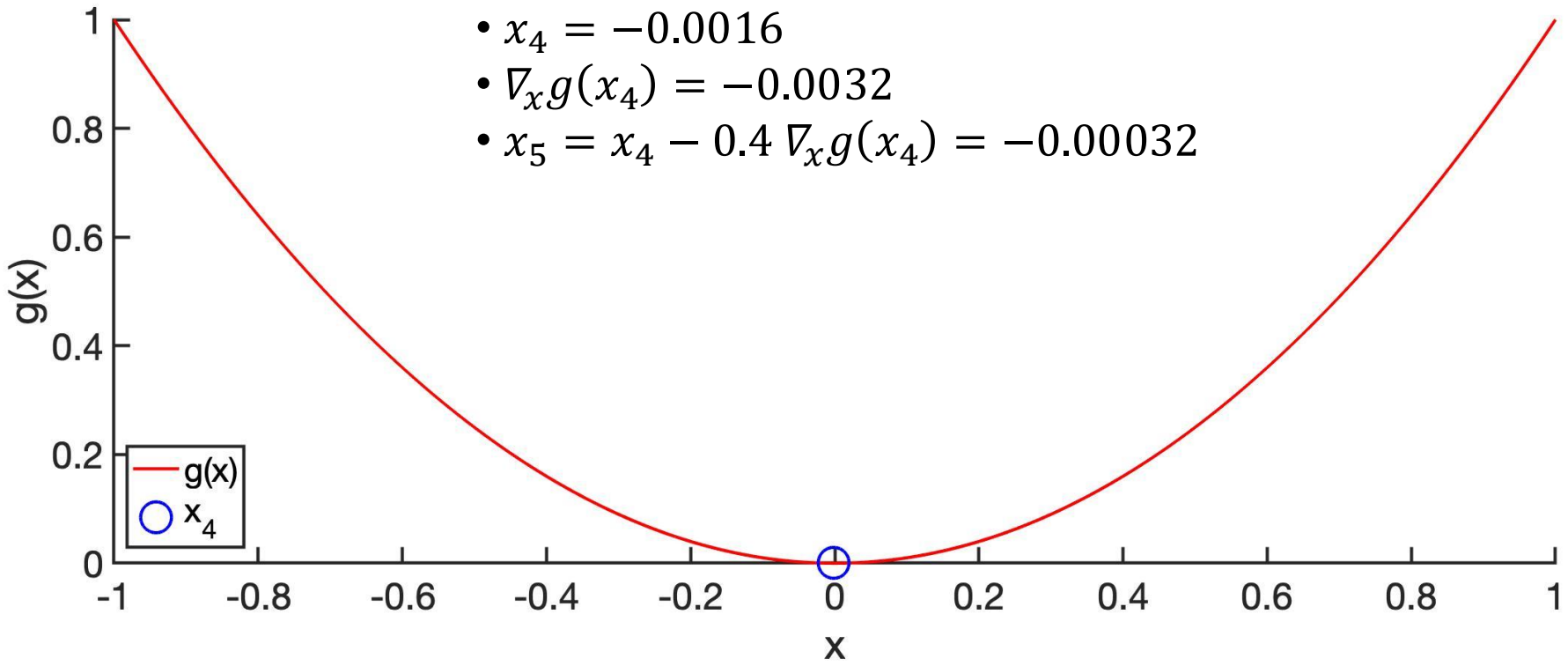
- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.4$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



- $x_3 = -0.008$
- $\nabla_x g(x_3) = -0.016$
- $x_4 = x_3 - 0.4\, \nabla_x g(x_3) = -0.0016$

Find x to minimize $g(x) = x^2$

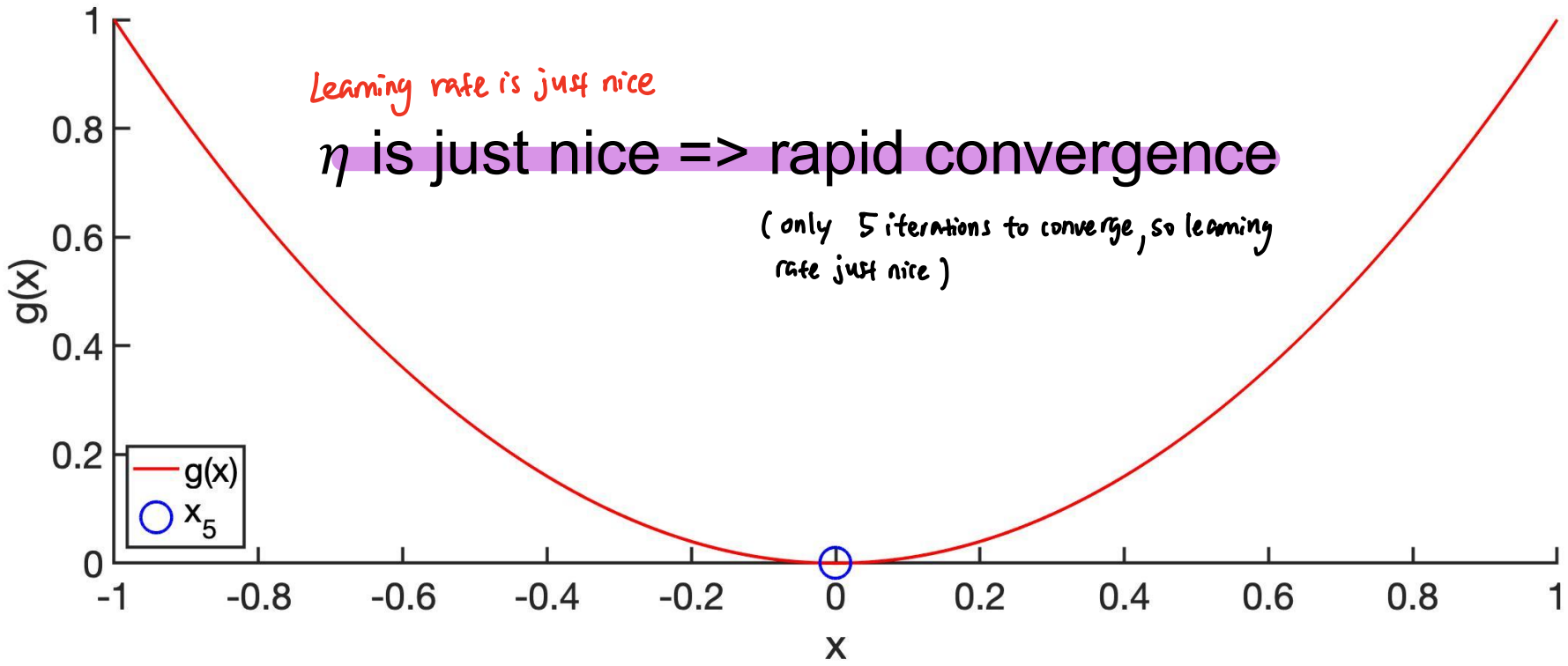
- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.4$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



- $x_4 = -0.0016$
- $\nabla_x g(x_4) = -0.0032$
- $x_5 = x_4 - 0.4\, \nabla_x g(x_4) = -0.00032$

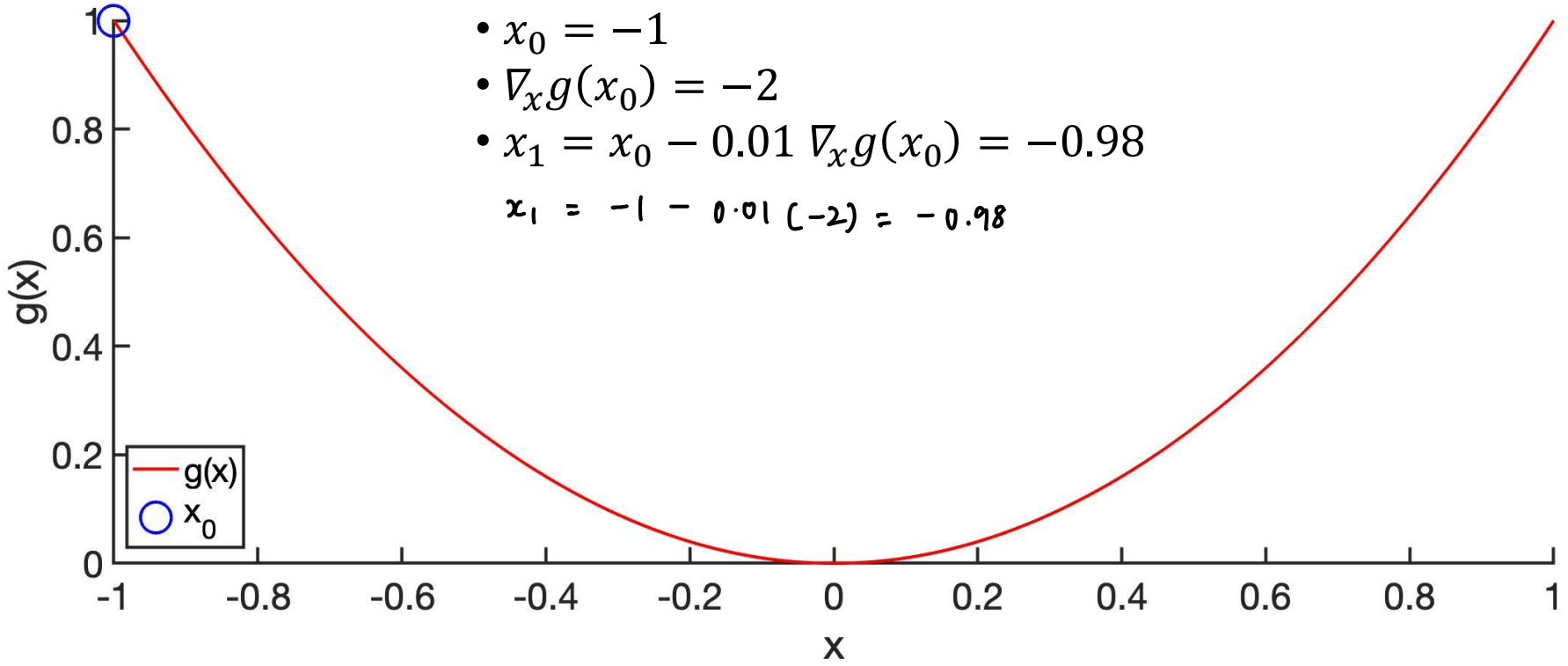
Find x to minimize $g(x) = x^2$

- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.4$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



What if learning rate η is too small?

- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.01$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$

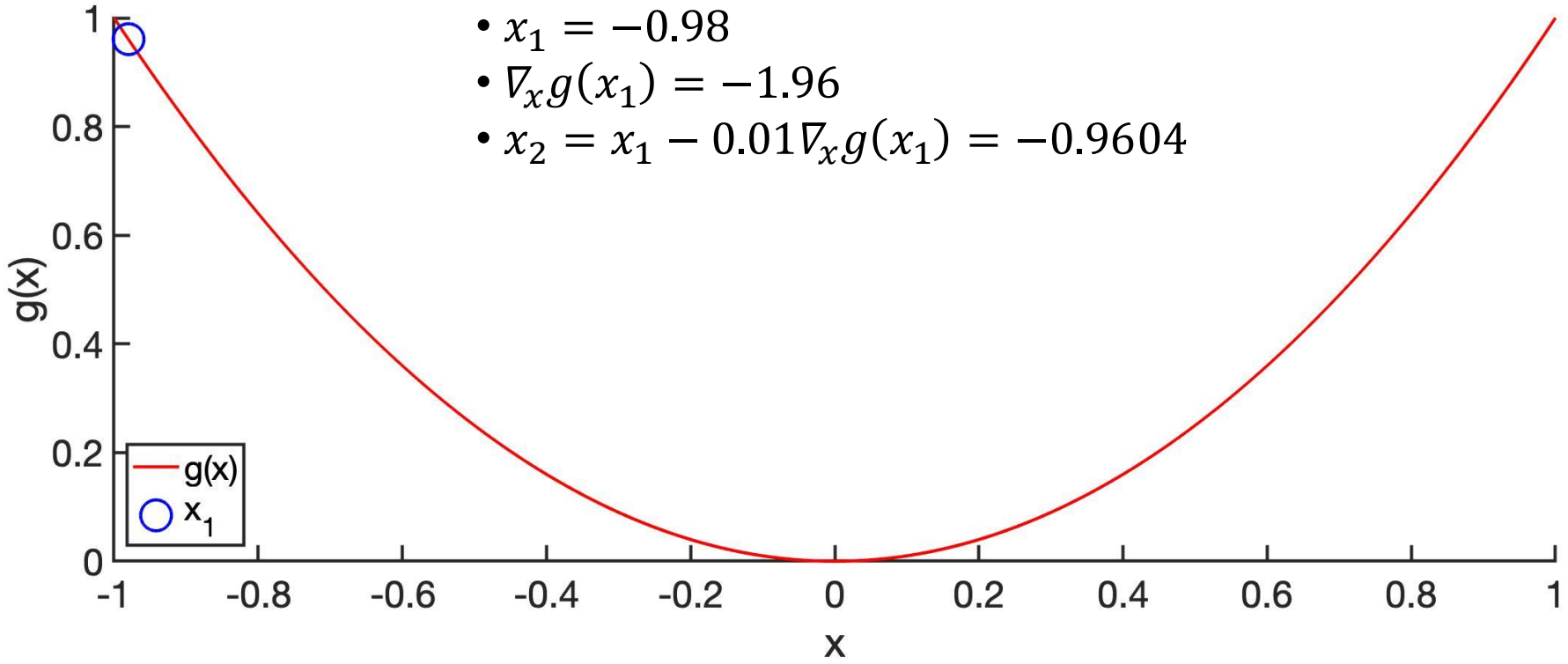


- $x_0 = -1$
- $\nabla_x g(x_0) = -2$
- $x_1 = x_0 - 0.01\, \nabla_x g(x_0) = -0.98$

$x_1 = -1 - 0.01(-2) = -0.98$

What if learning rate η is too small?

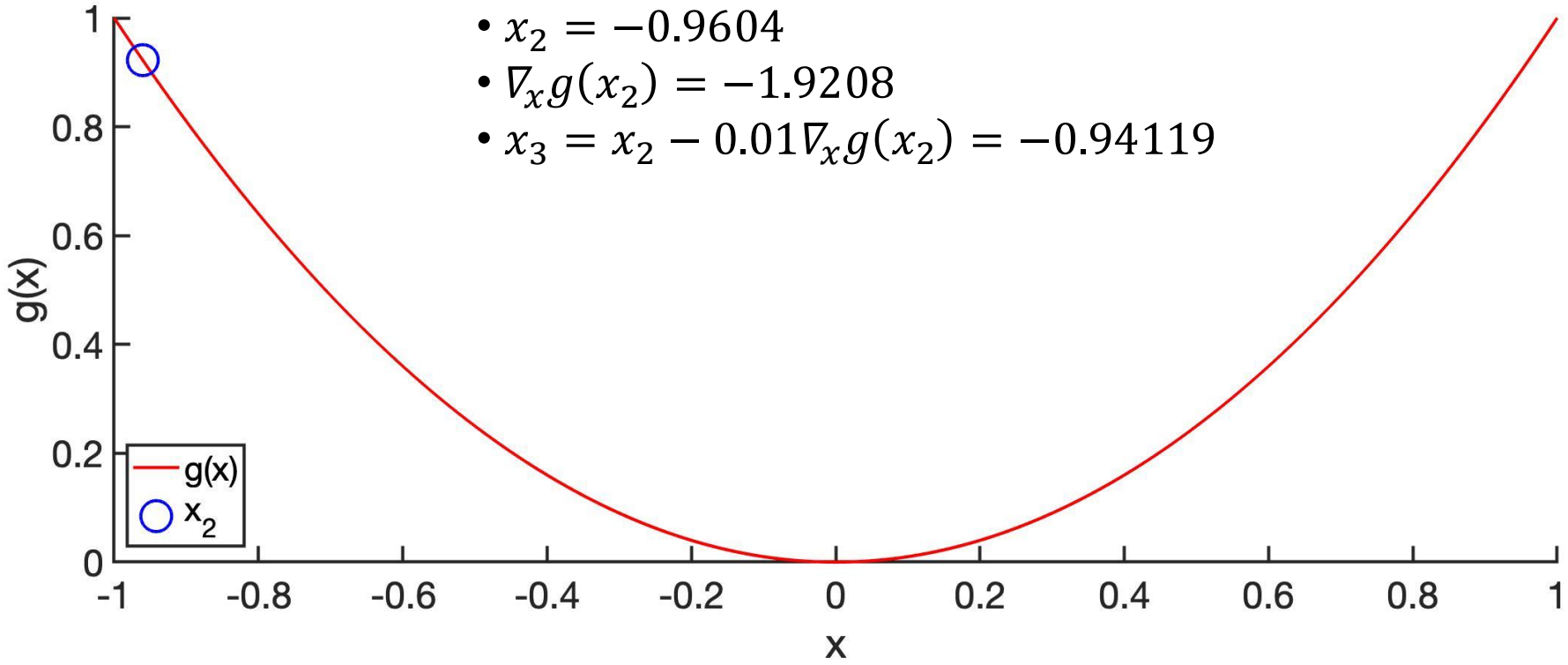
- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.01$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



- $x_1 = -0.98$
- $\nabla_x g(x_1) = -1.96$
- $x_2 = x_1 - 0.01 \nabla_x g(x_1) = -0.9604$

What if learning rate η is too small?

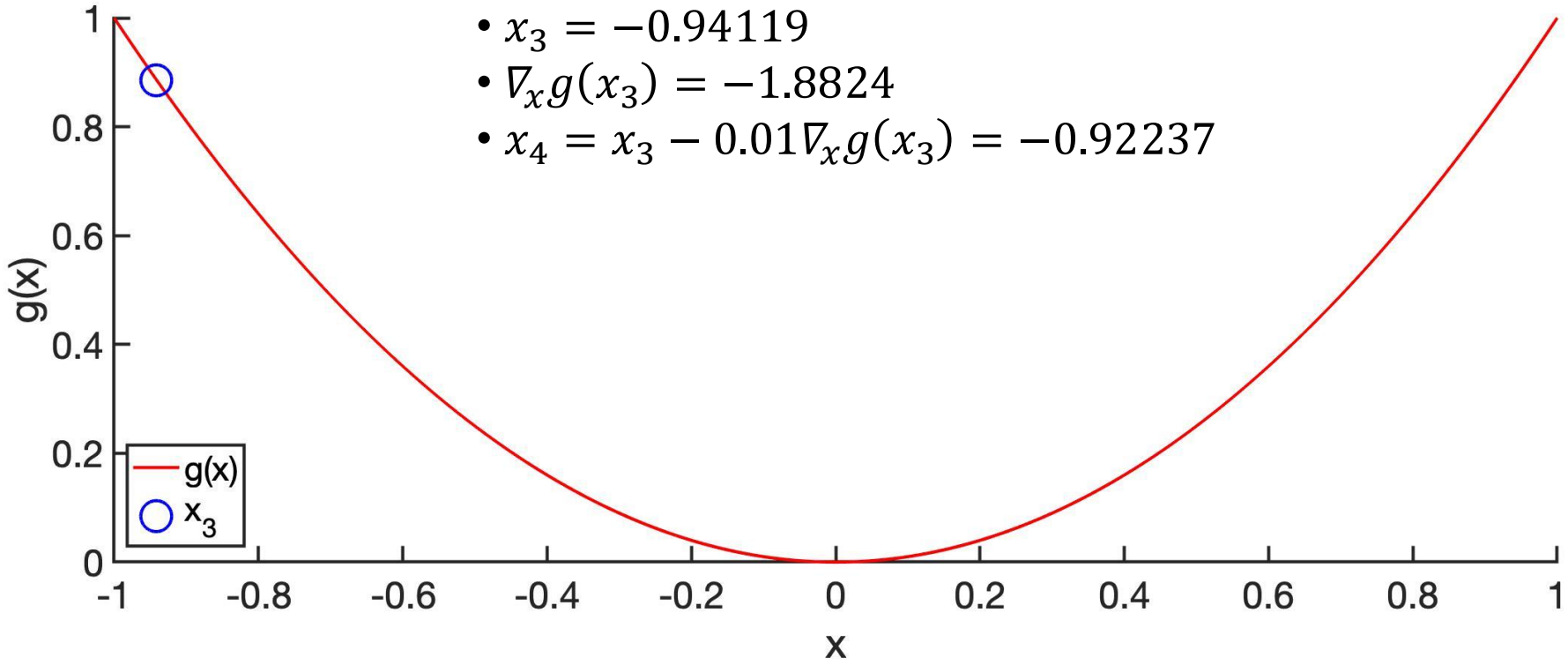
- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.01$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



- $x_2 = -0.9604$
- $\nabla_x g(x_2) = -1.9208$
- $x_3 = x_2 - 0.01 \nabla_x g(x_2) = -0.94119$

What if learning rate η is too small?

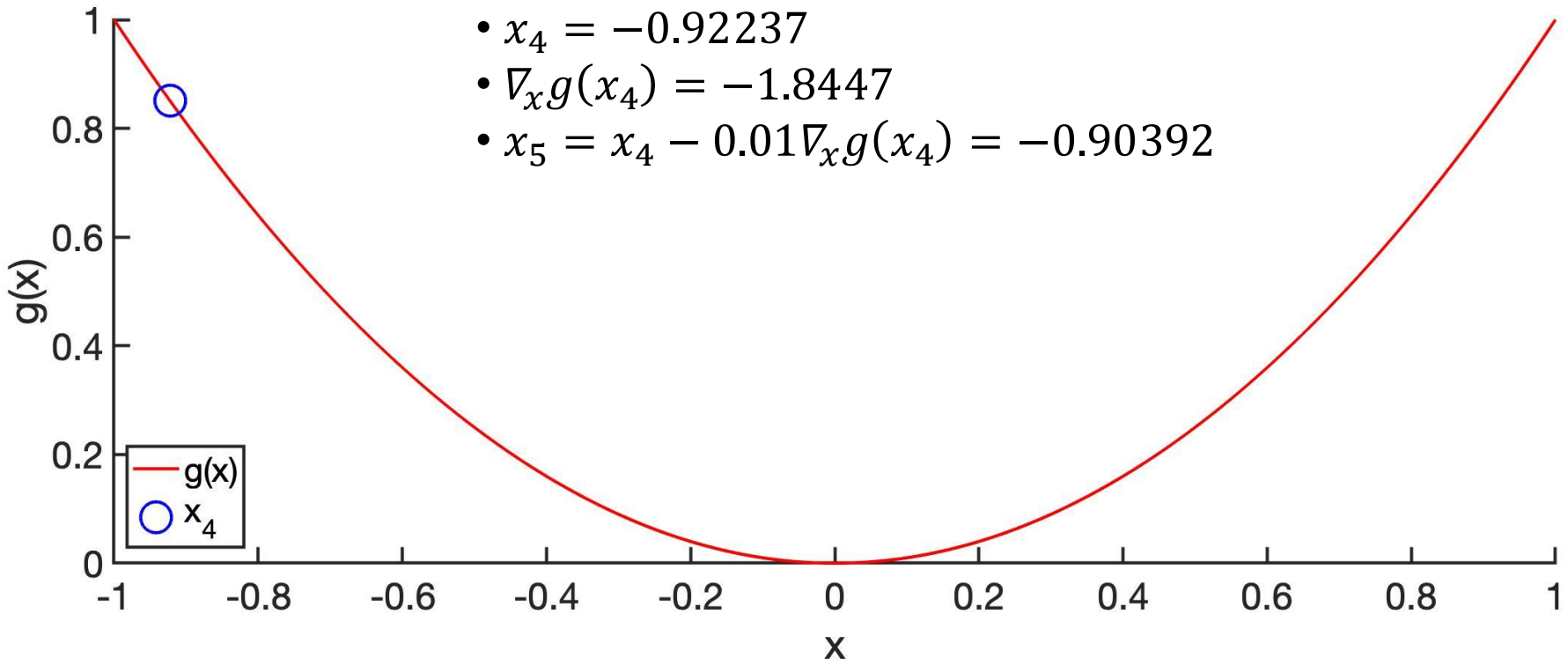
- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.01$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



- $x_3 = -0.94119$
- $\nabla_x g(x_3) = -1.8824$
- $x_4 = x_3 - 0.01 \nabla_x g(x_3) = -0.92237$

What if learning rate η is too small?

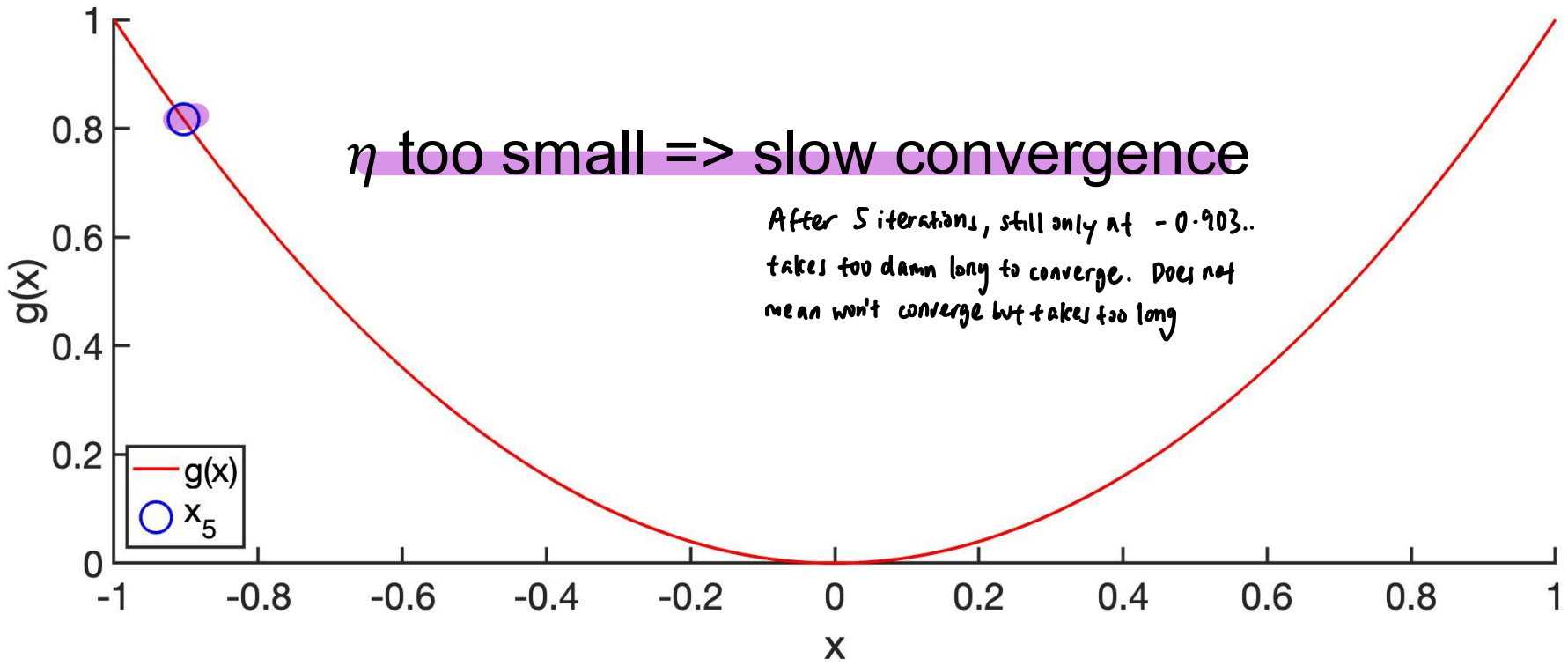
- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.01$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



- $x_4 = -0.92237$
- $\nabla_x g(x_4) = -1.8447$
- $x_5 = x_4 - 0.01 \nabla_x g(x_4) = -0.90392$

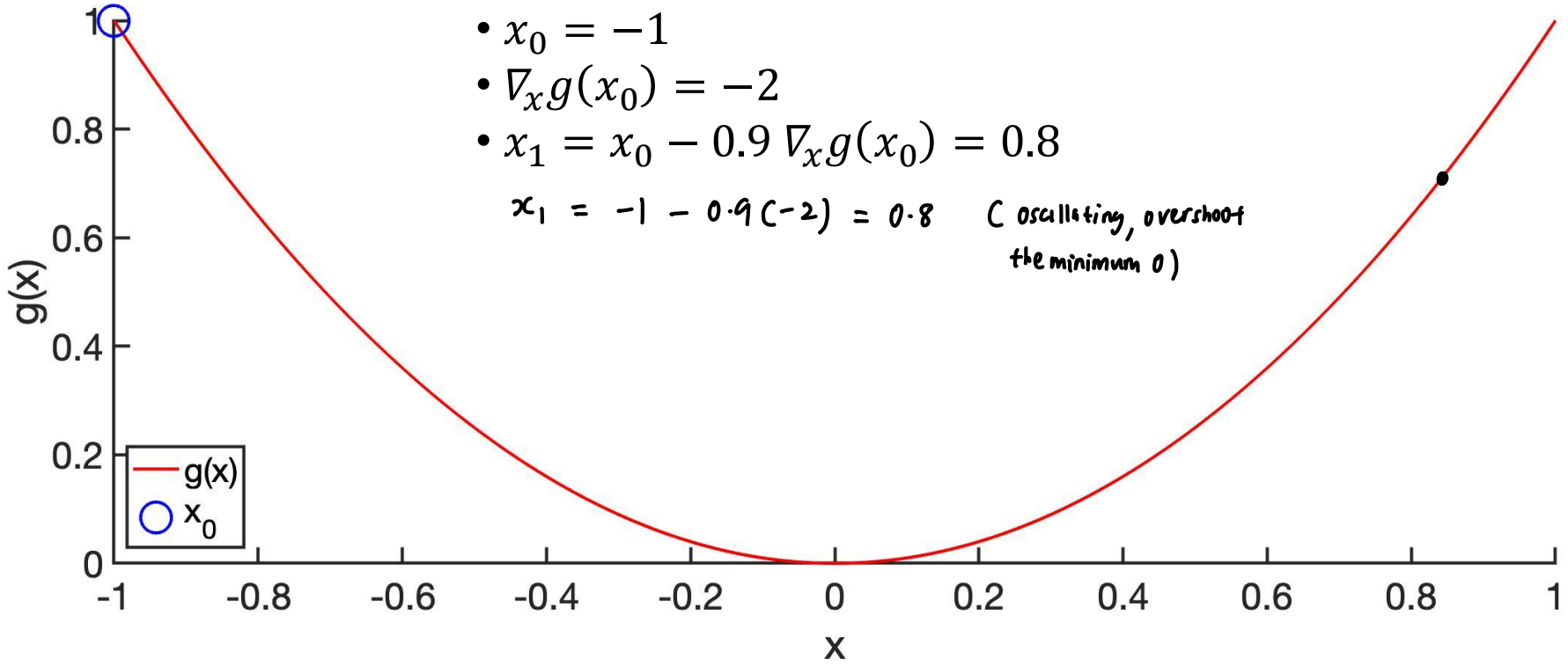
What if learning rate η is too small?

- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.01$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



What if learning rate η is too big?

- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.9$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$

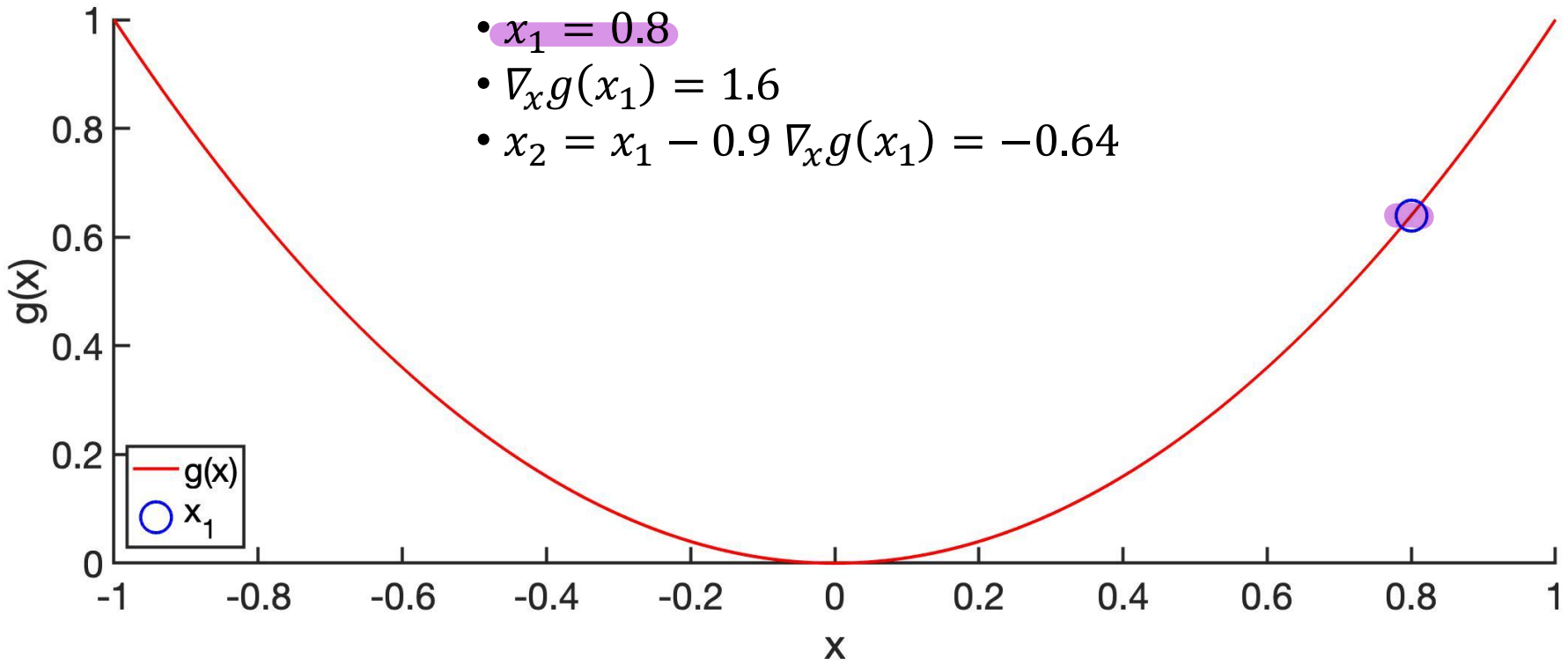


- $x_0 = -1$
- $\nabla_x g(x_0) = -2$
- $x_1 = x_0 - 0.9\,\nabla_x g(x_0) = 0.8$

$x_1 = -1 - 0.9(-2) = 0.8$ (oscillating, overshoot the minimum 0)

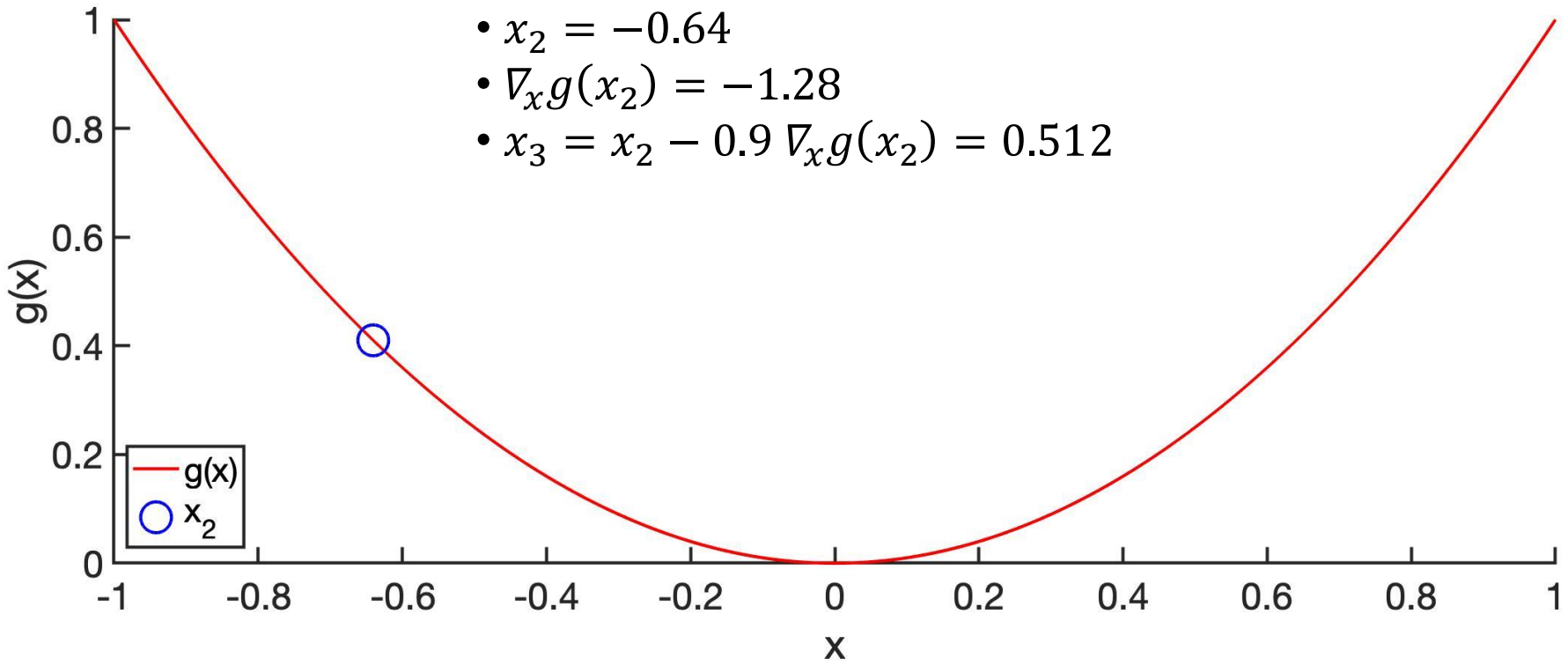
What if learning rate η is too big?

- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.9$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



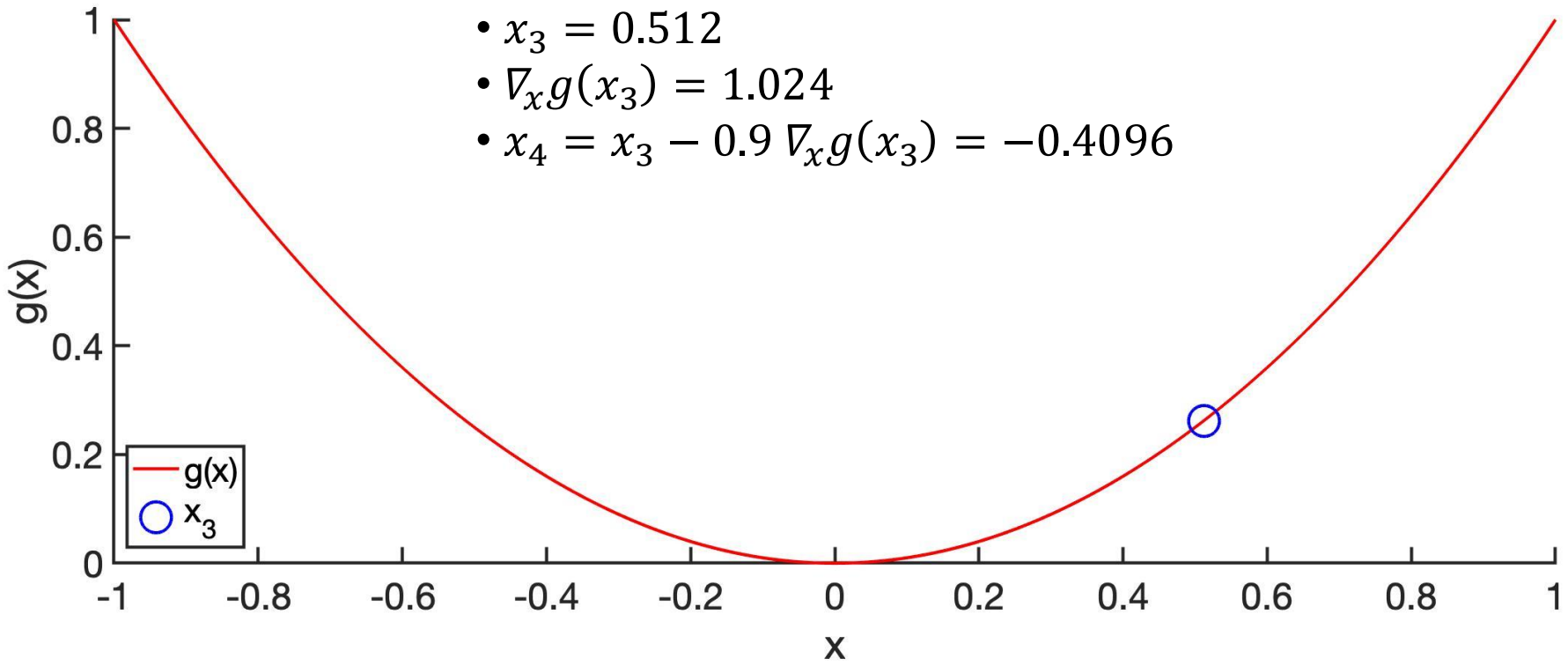
What if learning rate η is too big?

- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.9$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



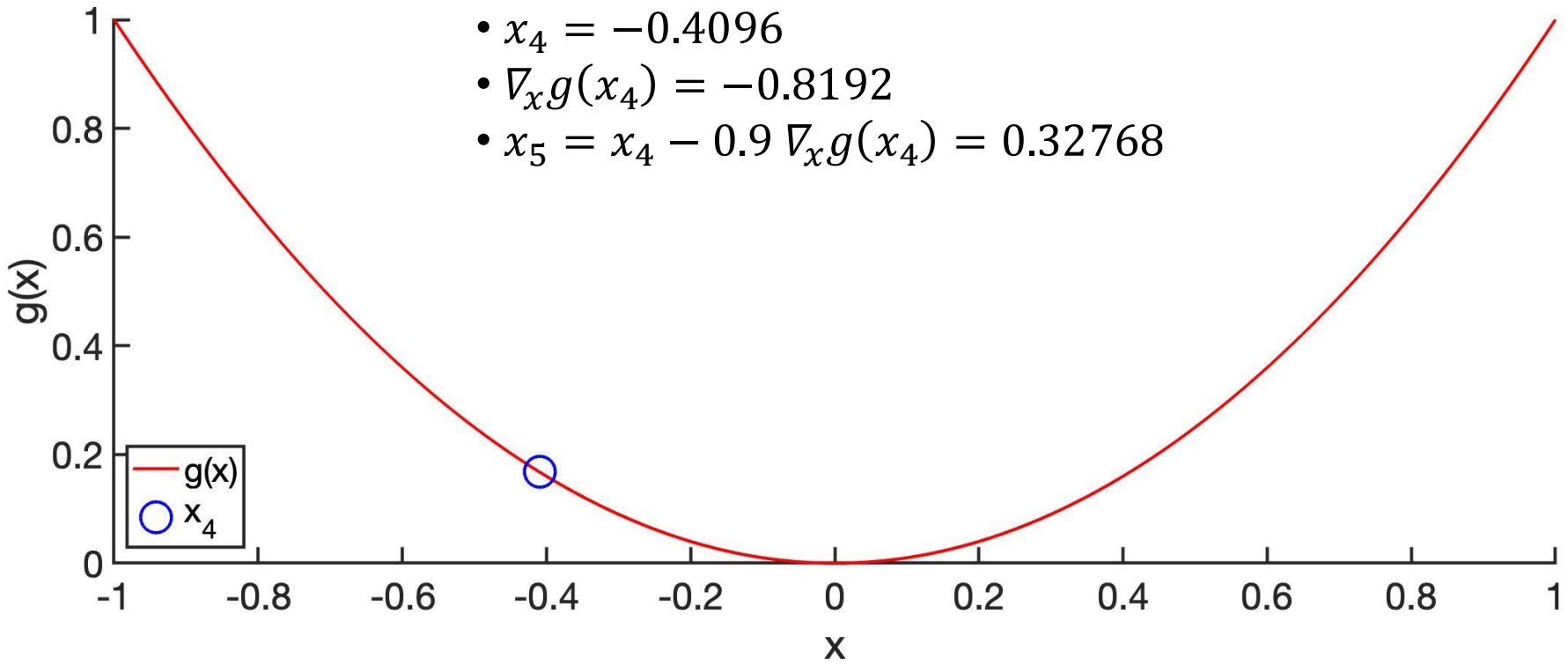
What if learning rate η is too big?

- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.9$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$



What if learning rate η is too big?

- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.9$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$

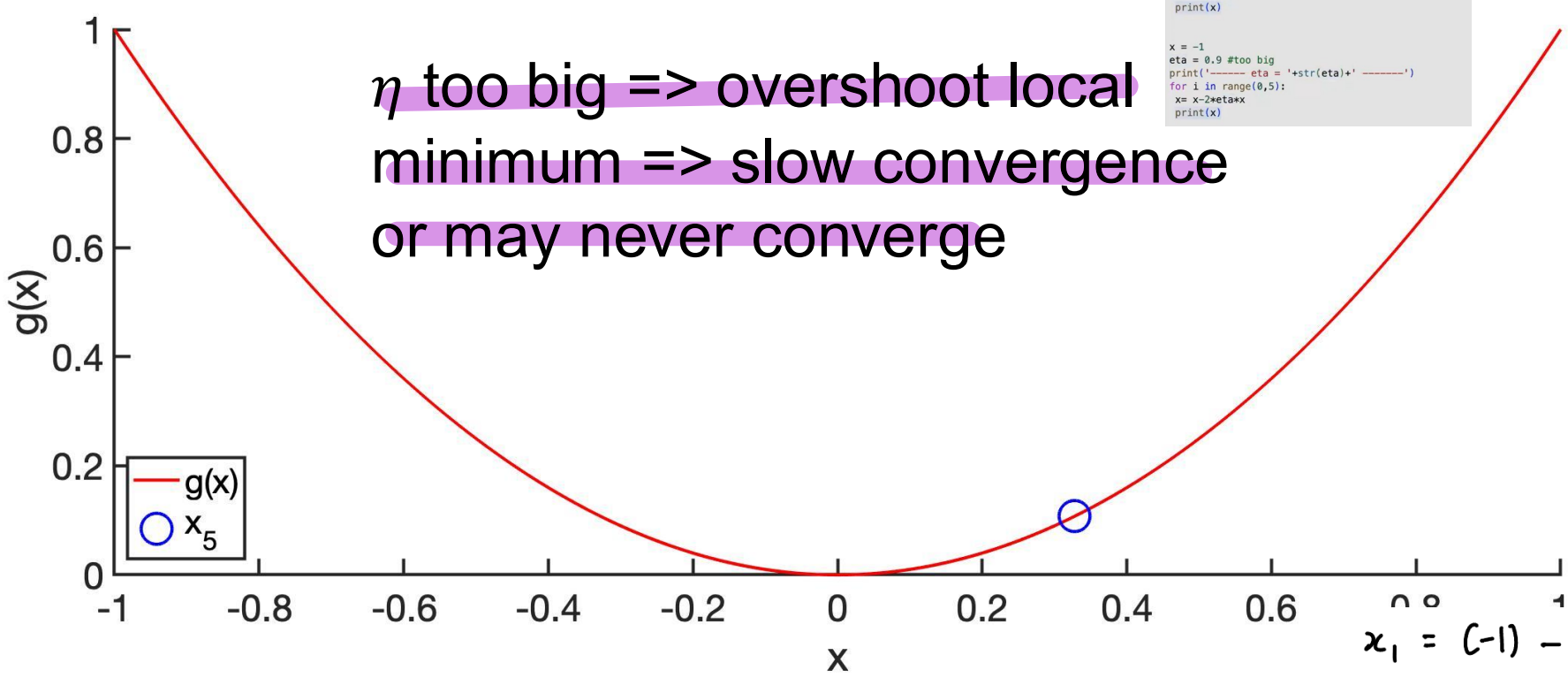


What if learning rate η is too big?

- Obviously minimum corresponds to $x = 0$, but let's do gradient descent
  - Gradient $\nabla_x g(x) = 2x$
  - Initialize $x_0 = -1$, learning rate $\eta = 0.9$
  - At each iteration, $x_{k+1} = x_k - \eta \nabla_x g(x_k)$

```python
# Lecture 8 Demo 1 - Gradient descent
# #!/usr/bin/env python3
# -*- coding: utf-8 -*-


import pandas as pd
import matplotlib.pyplot as plt
import numpy as np


# Main function is y=x**2
# Gradient is 2x
# Initialization & Parameters
x = -1
eta = 0.4 #just nice
print('------ eta = '+str(eta)+' -------')
for i in range(0,5):
 x= x-2*eta*x
 print(x)


x = -1
eta = 0.01 #too small
print('------ eta = '+str(eta)+' -------')
for i in range(0,5):
 x= x-2*eta*x
 print(x)


x = -1
eta = 0.9 #too big
print('------ eta = '+str(eta)+' -------')
for i in range(0,5):
 x= x-2*eta*x
 print(x)
```

Python Demo

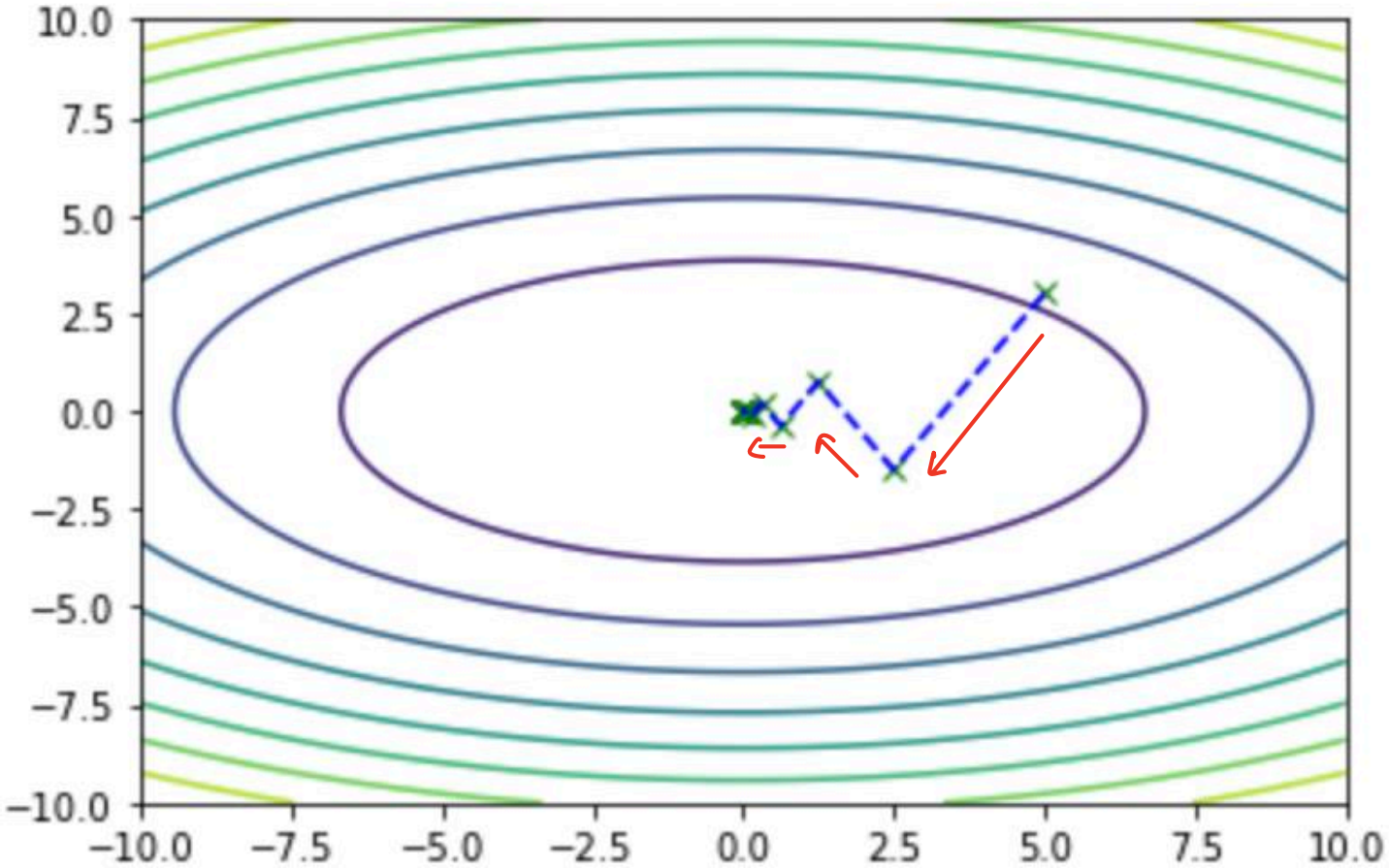


η too big => overshoot local minimum => slow convergence or may never converge

$x_1 = (-1) - 0.4(-2)$

$(w_{k+1}) = w_k - \eta \nabla_w C(w$

Gradient descent: quadratic function



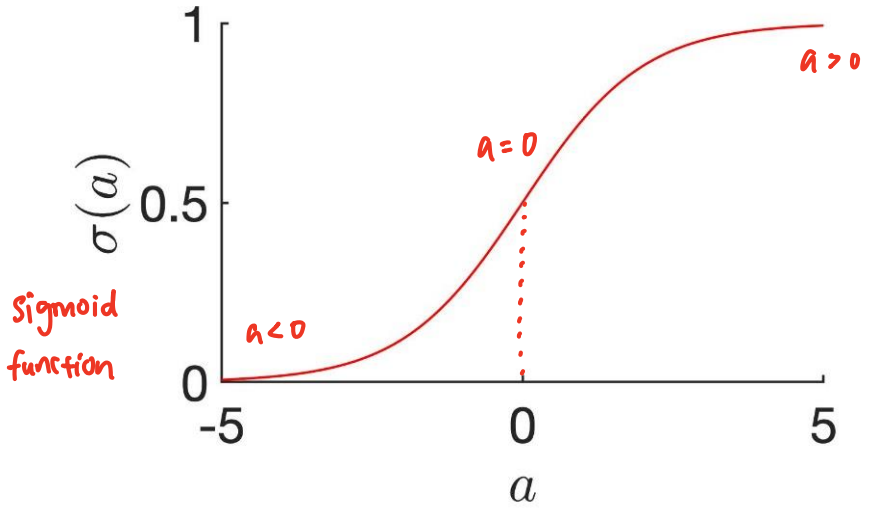
Convergence to the foot of the valley

Different Learning Models

- Different learning models $f(\mathbf{x}_i, \mathbf{w})$ reflect our beliefs about the relationship between the features $\mathbf{x}_i$ and target y_i
  - For example, $f(\mathbf{x}_i, \mathbf{w}) = \mathbf{p}_i^T \mathbf{w}$ assumes polynomial relationship between features and target
- Suppose we are performing classification (rather than regression), so y_i is class -1 or class 1
  - $\mathbf{p}_i^T \mathbf{w}$ is number between $-\infty$ to ∞.
  - Can use sigmoid function to map $\mathbf{p}_i^T \mathbf{w}$ to between 0 and 1:

$$f(\mathbf{x}_i, \mathbf{w}) = \sigma(\mathbf{p}_i^T \mathbf{w})$$

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$



Different Learning Models

- Different learning models $f(\mathbf{x}_i, \mathbf{w})$ reflect our beliefs about the relationship between the features $\mathbf{x}_i$ and target y_i
  - For example, $f(\mathbf{x}_i, \mathbf{w}) = \mathbf{p}_i^T \mathbf{w}$ assumes polynomial relationship between features and target
- Suppose we are performing classification (rather than regression), so y_i is class -1 or class 1
  - $\mathbf{p}_i^T \mathbf{w}$ is number between $-\infty$ to ∞.
  - Can use sigmoid function to map $\mathbf{p}_i^T \mathbf{w}$ to between 0 and 1:

$$f(\mathbf{x}_i, \mathbf{w}) = \sigma(\mathbf{p}_i^T \mathbf{w})$$

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$

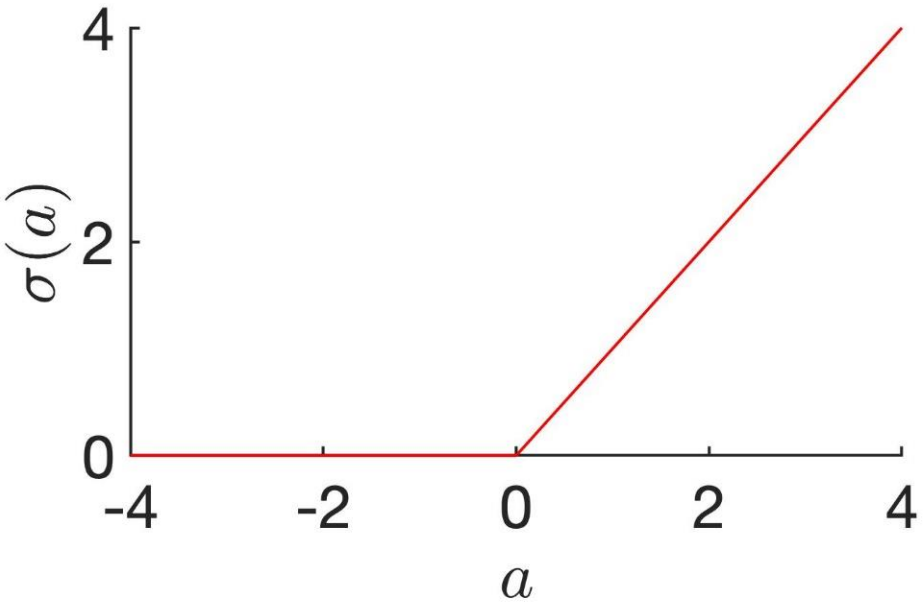
  - If $f(\mathbf{x}_i, \mathbf{w})$ is closer to 0 (or 1), we predict class -1 (or class 1)
- More generally, in one layer neural network: $f(\mathbf{x}_i, \mathbf{w}) = \sigma(\mathbf{p}_i^T \mathbf{w})$, where activation function σ can be sigmoid or some other functions & $\mathbf{p}$ is linear

activation function

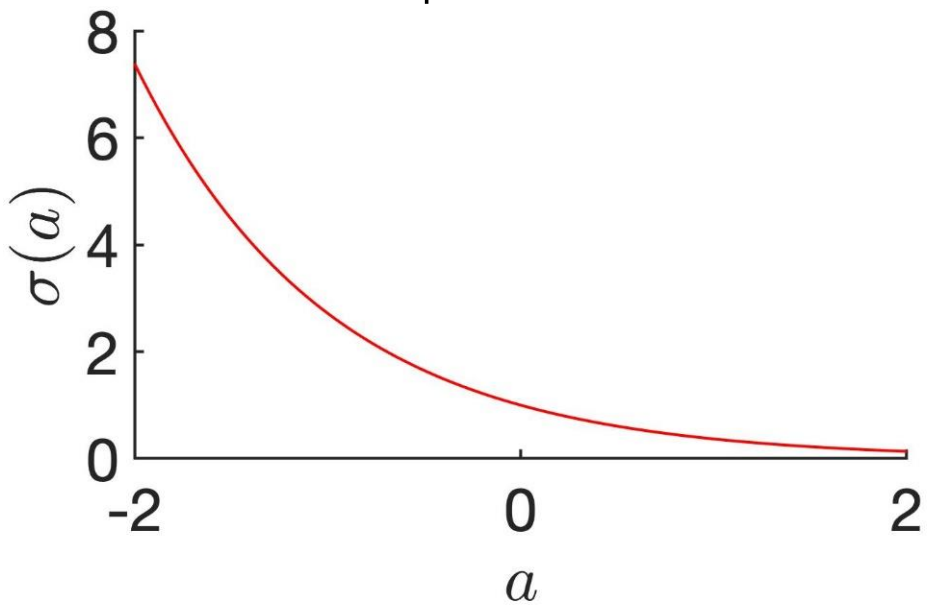
Different Learning Models

- $f(\mathbf{x}_i, \mathbf{w}) = \sigma(\mathbf{p}_i^T \mathbf{w})$, where σ can be different functions:
- Rectified linear unit (ReLU): $\sigma(a) = \max(0, a)$ → $a \le 0$: 0, $a > 0$: a
  derivative of RELU: $\frac{\partial \sigma(a)}{\partial a} = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{if } a \le 0 \end{cases}$
- Exponential: $\sigma(a) = \exp(-a)$ → $a \to \infty : \sigma \to 0$, $a \to -\infty : \sigma \to \infty$

ReLU

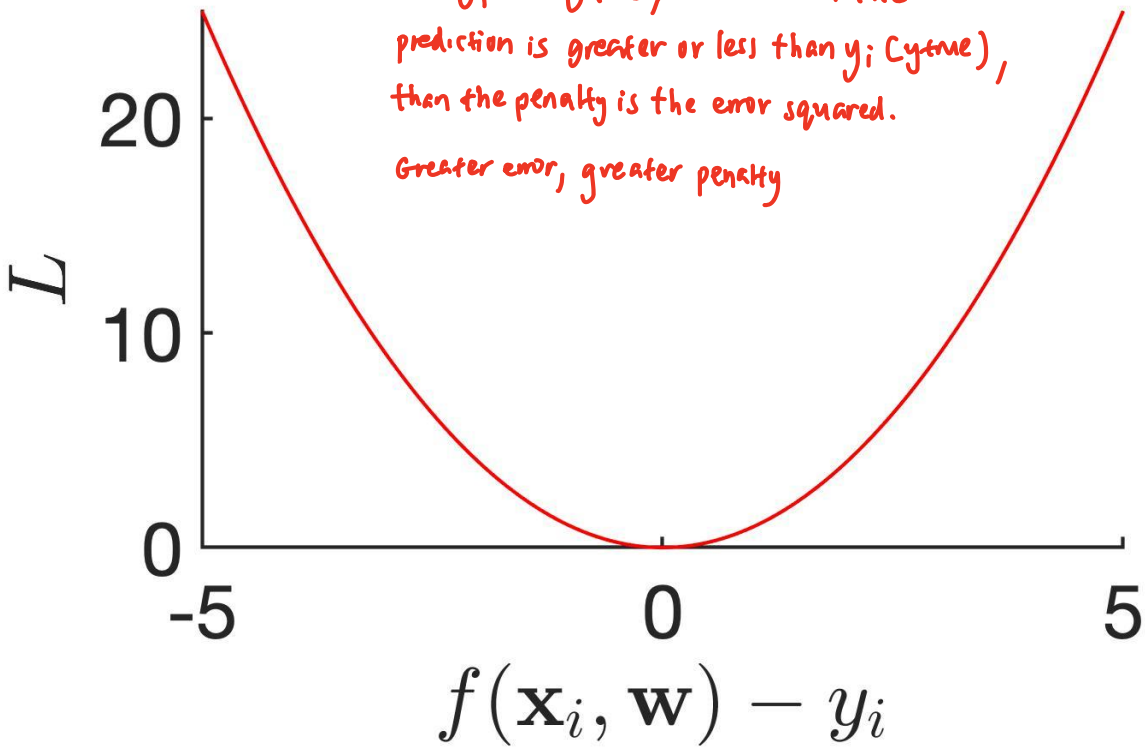


Exponential



Different Loss Functions

- Different loss functions $L(f(\mathbf{x}_i, \mathbf{w}), y_i)$ encodes the penalty when we predict $f(\mathbf{x}_i, \mathbf{w})$ but the true value is y_i
  - $L(f(\mathbf{x}_i, \mathbf{w}), y_i) = (f(\mathbf{x}_i, \mathbf{w}) - y_i)^2$ is called the square error loss



Different Loss Functions

- Different loss functions $L(f(\mathbf{x}_i, \mathbf{w}), y_i)$ encodes the penalty when we predict $f(\mathbf{x}_i, \mathbf{w})$ but the true value is y_i
  - $L(f(\mathbf{x}_i, \mathbf{w}), y_i) = (f(\mathbf{x}_i, \mathbf{w}) - y_i)^2$ is called the square error loss
- Suppose we are performing classification (rather than regression), so y_i is class -1 or class 1, then square error loss makes less sense. Instead, we can use
  - Binary loss (or 0–1 loss): $L(f(\mathbf{x}_i, \mathbf{w}), y_i) = \begin{cases} 0 & \text{if } f(\mathbf{x}_i, \mathbf{w}) = y_i \\ 1 & \text{if } f(\mathbf{x}_i, \mathbf{w}) \neq y_i \end{cases}$

    ypredict ytrue
  - In practice, hard to constrain $f(\mathbf{x}_i, \mathbf{w})$ to be exactly -1 or 1, so we can declare "victory" if $f(\mathbf{x}_i, \mathbf{w})$ & y have the same sign:

$$L(f(\mathbf{x}_i, \mathbf{w}), y_i) = \begin{cases} 0 & \text{if } f(\mathbf{x}_i, \mathbf{w}) y_i > 0 \\ 1 & \text{if } f(\mathbf{x}_i, \mathbf{w}) y_i < 0 \end{cases}$$

multiply, if same sign, if should be > 0, penalty: 0

multiply, if different sign, ie, one negative one positive, than < 0, have penalty of 1

Different Loss Functions

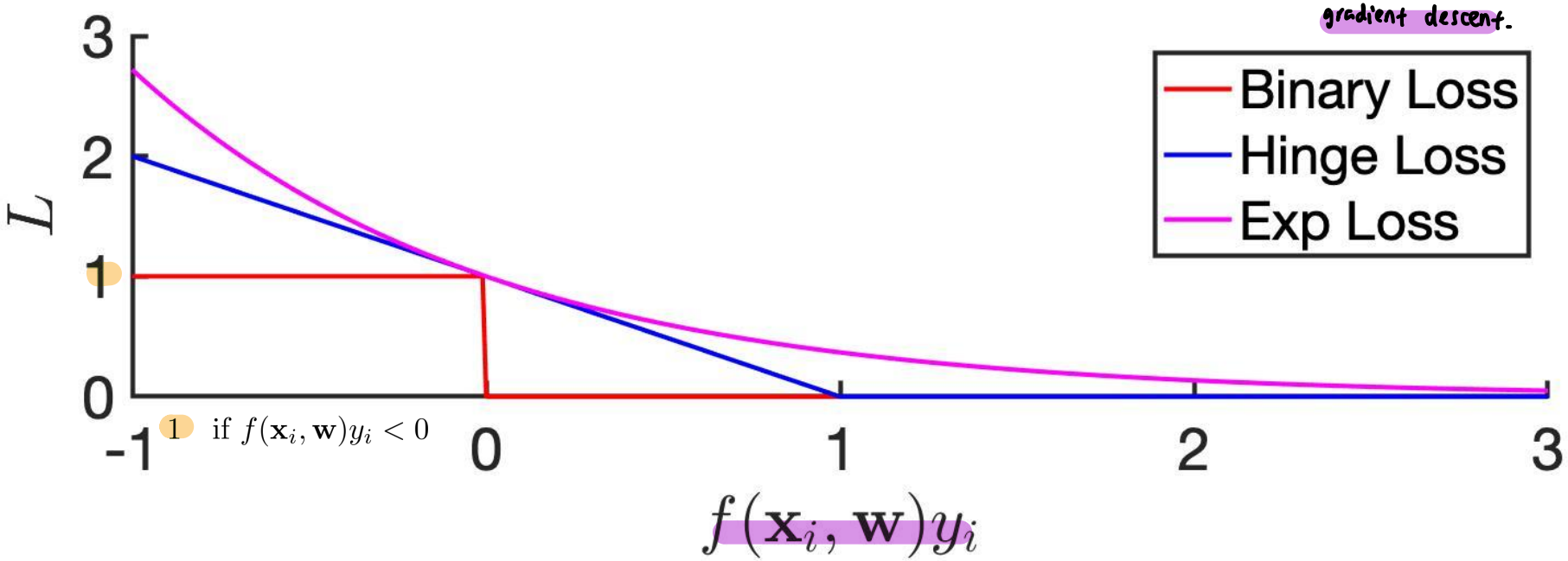
- Binary loss, where y_i is class -1 or class 1 & $f(\mathbf{x}_i, \mathbf{w})$ is a number between $-\infty$ and ∞: $L(f(\mathbf{x}_i, \mathbf{w}), y_i) = \begin{cases} 0 & \text{if } f(\mathbf{x}_i, \mathbf{w}) y_i > 0 \\ 1 & \text{if } f(\mathbf{x}_i, \mathbf{w}) y_i < 0 \end{cases}$

loss function

↗ cannot do gradient descent

- Binary loss not differentiable, so two other possibilities
  - Hinge loss: $L(f(\mathbf{x}_i, \mathbf{w}), y_i) = \max(0, 1 - f(\mathbf{x}_i, \mathbf{w}) y_i)$
  - Exponential loss: $L(f(\mathbf{x}_i, \mathbf{w}), y_i) = \exp(-f(\mathbf{x}_i, \mathbf{w}) y_i)$

← use this two loss functions instead when we do classification (+1, -1) because this two can be differentiable and we can do gradient descent.



Summary

- Building blocks of machine learning algorithms
  - Learning model: reflects our belief about relationship between features & target we want to predict
  - Loss function: penalty for wrong prediction
  - Regularization: penalizes complex models
  - Optimization routine: find minimum of overall cost function
- Gradient descent algorithm
  - At each iteration, compute gradient & update model parameters in direction opposite to gradient
  - If learning rate η is too big => may not converge
  - If learning rate η is too small => converge very slowly
- Different learning models, e.g., linear, polynomial, sigmoid, ReLU, exponential, etc
- Different loss functions, e.g., square error, binary, logistic, etc

Let us minimize f(x) = x^2. Assume learning rate = 0.1 & initialize x to be 2. What is the value of x after the 1st iteration?

Jin Yueming

0

1.2

1.4

✓ 1.6

1.8

$\frac{\partial f(x)}{\partial x} = 2x$

$w_1 = w_0 - \eta\left(\frac{\partial f(x)}{\partial x}\right)$

$= 2 - 0.1(2(2))$

$= 1.6$