IN402

Machine Learning

Chapter

1

Linear Regression and Optimization

Author: Abbas El-Hajj Youssef

University: Lebanese University

Department: Computer Science Department

These notes extend course materials taught by Prof. Ahmad Faour with additional content from textbooks and supplementary resources.

Disclaimer: *This is not an official course document.*

Contents

1 Introduction to Regression

Supervised learning forms one of the cornerstones of machine learning, distinguished by a fundamental characteristic: we are given a dataset where we already know what the correct output should look like. This prior knowledge of correct outputs guides the learning process, allowing algorithms to learn patterns and relationships from labeled examples.

Regression vs. Classification

Within supervised learning, problems naturally divide into two categories based on the nature of their outputs:

- **Regression**: Predicts *continuous, real-valued* outputs–quantities that can take any value within a range, such as house prices measured in dollars, temperatures in degrees, or stock prices.
- **Classification**: Predicts *discrete, categorical* outputs–distinct labels like spam or not spam, disease present or absent, or which of several categories an item belongs to.

This chapter focuses exclusively on regression, and more specifically on **linear regression**, which makes a simplifying but powerful assumption: that the relationship between input features and the output can be approximated by a linear function. While this might seem restrictive, linear regression serves as both a practical tool for many real-world problems and a foundational concept upon which more sophisticated methods are built.

Linear Regression

A supervised learning task that models the relationship between input variables and a continuous target variable using a linear function. Given a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$ with $x^{(i)} \in \mathbb{R}^n$ and $y^{(i)} \in \mathbb{R}$, the goal is to learn a linear function h_θ that predicts $\hat{y} = h_\theta(x)$ accurately on new, previously unseen inputs.

It is worth noting from the outset that linear regression's applicability extends beyond strictly linear relationships. Through a technique called *feature engineering*, we can use linear regression to model nonlinear relationships by creating polynomial features (such as x^2, x^3) or other transformations of the original inputs. The term "linear" refers to linearity in the *parameters* θ, not necessarily in the input features themselves–a distinction we will explore in depth later in this chapter.

1.1 A Practical Example: Housing Prices

To make these concepts concrete, consider one of the classic problems in regression: predicting house prices. Imagine you are a real estate analyst tasked with estimating the market value of homes. You have historical data showing the sizes of houses (measured in square feet) and their corresponding sale prices (in thousands of dollars). Your goal is to learn a relationship between size and price that will allow you to estimate the value of a new house based on its size.

This is precisely the type of problem linear regression is designed to solve. We treat the house size as our *input* (often denoted x) and the price as our *output* or *target* (denoted y). From

a collection of past sales–our training set–we learn a hypothesis function h_θ that captures the relationship between size and price. Once learned, this function can estimate the price of any house given its size.

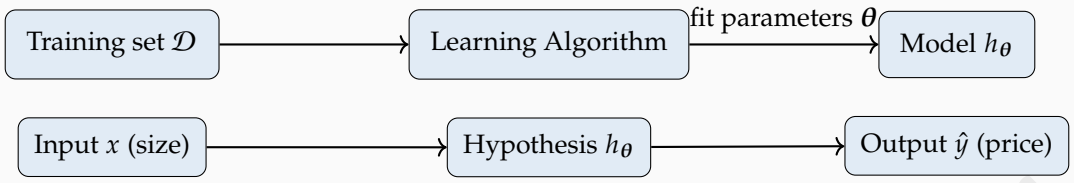


Figure 1: *The machine learning pipeline: training data flows through a learning algorithm to produce a model with learned parameters θ. During inference, new inputs are mapped through the hypothesis function to produce predictions.*

The workflow consists of two distinct phases. During *training*, we feed our historical data into a learning algorithm that automatically discovers the best values for the parameters θ that define our model. During *inference* or *prediction*, we apply the learned model to new inputs to generate estimated outputs. This separation between learning and prediction is fundamental to supervised learning.

2 Problem Setup and Notation

Before diving into the mathematics of linear regression, we need to establish clear notation and terminology. Precision in notation is not mere pedantry–it enables clear thinking, correct implementation, and effective communication of ideas.

2.1 Understanding the Training Set

Let's begin with the simplest scenario: **univariate linear regression**, where we predict an output based on a single input feature. While real-world problems often involve many features, starting with one feature allows us to visualize the problem and build intuition before generalizing.

Key Notation Elements

We denote:

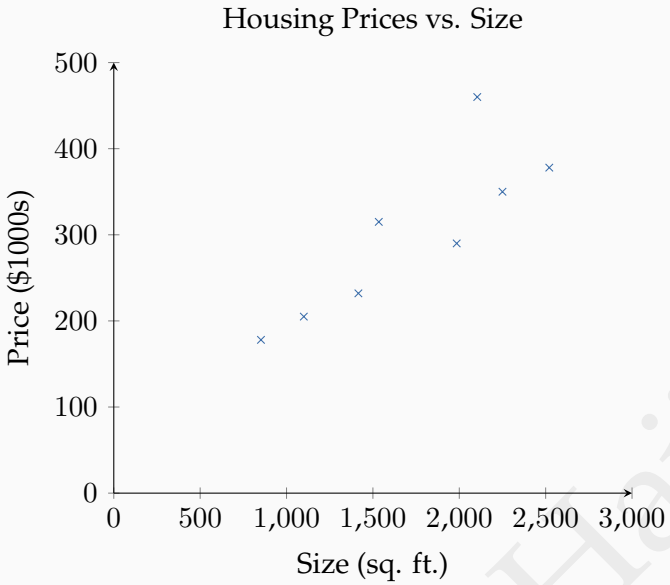
- m The number of training examples in our dataset
- n The number of features (for univariate regression, $n = 1$)
- x The input variable or feature (e.g., house size in square feet)
- y The output variable or target (e.g., house price in thousands of dollars)
- $(x^{(i)}, y^{(i)})$ The i-th training example, where the superscript (i) is an *index* into the training set, not an exponent
- $h_\theta(x)$ The hypothesis function–our model that makes predictions

A complete training set consists of m such pairs: $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$. Each pair represents one observation from which we can learn.

The **hypothesis function** $h_{\boldsymbol{\theta}}(x)$ is our model–the mathematical function that takes an input x and produces a predicted output. For univariate linear regression, this takes the form:

$$h_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x$$

where θ_0 and θ_1 are **parameters** (also called *weights*) that determine the specific line. We collect these parameters into a vector $\boldsymbol{\theta} = (\theta_0, \theta_1)^\top$. The parameter θ_0 is the *intercept* (where the line crosses the vertical axis when $x = 0$), and θ_1 is the *slope* (how much y changes for each unit increase in x).



Size (x)	Price (y)
2104 ($x^{(1)}$)	460 ($y^{(1)}$)
1416 ($x^{(2)}$)	232 ($y^{(2)}$)
852 ($x^{(3)}$)	178 ($y^{(3)}$)
⋮	⋮

Each row is one training example. For instance, $(x^{(2)}, y^{(2)}) = (1416, 232)$ means a house with 1416 square feet sold for \$232,000.

Figure 2: *Scatter plot of training data: each point represents a house with its size (horizontal axis) and price (vertical axis). Our goal is to find a line that best fits these points.*

2.2 The Intercept Term Convention

As we transition from single-feature to multi-feature models, a notational convention will greatly simplify our mathematics. We introduce a dummy feature x_0 that always equals 1 for every training example. This allows us to treat the intercept θ_0 uniformly with other parameters.

With this convention, we rewrite our feature vector and parameter vector as:

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ x \end{bmatrix} \in \mathbb{R}^2, \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \in \mathbb{R}^2$$

Now the hypothesis becomes a compact dot product:

$$h_{\boldsymbol{\theta}}(x) = \theta_0 \cdot 1 + \theta_1 \cdot x = \theta_0 x_0 + \theta_1 x_1 = \sum_{j=0}^{1} \theta_j x_j = \boldsymbol{\theta}^\top \boldsymbol{x}$$

This vectorized form $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{\theta}^\top \boldsymbol{x}$ will prove invaluable as we scale to multiple features and implement efficient algorithms.

3 Vectorization and the Design Matrix

Modern machine learning relies heavily on **vectorization**–expressing operations in terms of vector and matrix computations rather than explicit loops. This is not merely a matter of elegant notation; it has profound practical implications.

Why Vectorization Matters

- **Computational speed**: Vectorized code is typically orders of magnitude faster because it leverages highly optimized linear algebra libraries (BLAS, LAPACK) that exploit parallel computation and specialized hardware instructions
- **Readability**: A single line like $\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha X^\top (X\boldsymbol{\theta} - \boldsymbol{y})$ replaces dozens of nested loops
- **Scalability**: The same code works whether you have 1 feature or 1000 features, 100 examples or 100 million
- **Fewer bugs**: Eliminating explicit loops reduces opportunities for indexing errors

3.1 The Design Matrix

When we have multiple training examples, we organize them into a **design matrix** X. This matrix has one row for each training example and one column for each feature (including the intercept term $x_0 = 1$).

For m training examples with n features (plus intercept), the design matrix is:

$$X = \begin{bmatrix} (\boldsymbol{x}^{(1)})^\top \\ (\boldsymbol{x}^{(2)})^\top \\ \vdots \\ (\boldsymbol{x}^{(m)})^\top \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} \in \mathbb{R}^{m \times (n+1)}$$

Similarly, we collect all target values into a vector:

$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \in \mathbb{R}^m$$

With these definitions, computing predictions for all training examples simultaneously becomes a single matrix-vector multiplication: $X\boldsymbol{\theta}$ produces a vector of predicted values for all m examples.

4 The Learning Objective: Minimizing Prediction Error

Learning in supervised learning is fundamentally an optimization problem. We seek parameter values that make our model's predictions as close as possible to the true target values in our training set. But what does "as close as possible" mean mathematically? We need a **cost function** (also called a *loss function* or *objective function*) that quantifies the quality of a given set of parameters.

4.1 The Mean Squared Error Cost Function

For linear regression, the most widely used cost function is the **Mean Squared Error (MSE)**. For a given parameter vector $\boldsymbol{\theta}$, it measures the average squared difference between predictions and true values:

Mean Squared Error Cost Function

$$J(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_{\boldsymbol{\theta}}(x^{(i)}) - y^{(i)} \right)^2 = \frac{1}{2m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)} - y^{(i)} \right)^2$$

Our learning objective is to find the parameters that minimize this cost:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

The factor $\frac{1}{2m}$ warrants explanation. The division by m ensures the cost represents an average per training example, making it comparable across datasets of different sizes. The factor of $\frac{1}{2}$ is purely for mathematical convenience–it will cancel with the factor of 2 produced when we differentiate the squared term, yielding cleaner gradient expressions. Some texts use $\frac{1}{m}$ instead; both conventions lead to the same optimal parameters since scaling the cost function doesn't change the location of its minimum.

Why Square the Errors?

Several reasons justify squaring the prediction errors:
Mathematical tractability: The squared function is differentiable everywhere and has well-behaved derivatives, essential for gradient-based optimization.
Penalization of large errors: Squaring amplifies larger errors. A prediction error of 10 contributes 100 to the cost, while an error of 1 contributes only 1. This makes the model particularly sensitive to outliers and large mistakes.
Statistical foundations: Under the assumption that prediction errors follow a Gaussian (normal) distribution, minimizing the MSE is equivalent to maximum likelihood estimation–finding the parameters most likely to have generated our observed data.
Geometric interpretation: For univariate regression, the cost is the sum of squared vertical distances from data points to the fitted line.

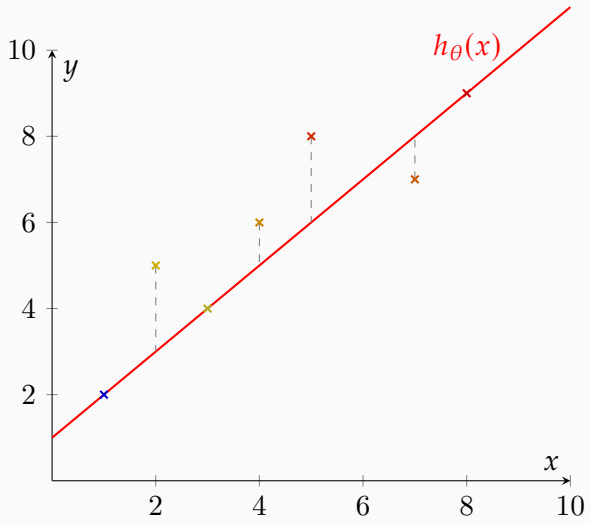


Figure 3: *Visualizing the cost function: dashed lines show the residuals (errors) for several points. The cost function sums the squares of these vertical distances.*

The MSE cost function emerges naturally from a probabilistic model of the data. Suppose we model the relationship between inputs and outputs as:

$$y^{(i)} = \boldsymbol{\theta}^\top \boldsymbol{x}^{(i)} + \epsilon^{(i)}$$

where $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$ are independent, identically distributed Gaussian noise terms with zero mean and variance σ^2. This says that the true target value equals the linear prediction plus some random noise. Under this model, finding the parameters that maximize the probability of observing our training data (maximum likelihood estimation) turns out to be mathematically equivalent to minimizing the MSE. This provides a principled statistical justification: MSE is the "right" cost function when we believe our data contains Gaussian noise–a reasonable assumption for many real-world phenomena.

Convexity: A Guarantee of Success

A crucial property distinguishes the MSE cost function for linear regression: it is **convex**. Geometrically, convexity means the cost function has a bowl-like shape with a single global minimum and no local minima.
For optimization, convexity is extraordinarily valuable. It guarantees that:

- Any local minimum is also a global minimum
- Gradient descent, starting from any initial parameter values, will converge to the optimal solution (given appropriate learning rate)
- We don't need to worry about getting stuck in suboptimal solutions

This stands in stark contrast to non-convex optimization problems, such as training neural networks, where convergence to the global minimum is not guaranteed.

5 Building Intuition: Visualizing the Cost Function

Abstract mathematical definitions become clearer when accompanied by concrete examples and visualizations. Let's develop intuition for how the cost function behaves by examining

specific cases.

5.1 The Simplest Case: One Parameter

Consider a deliberately simplified scenario: the dataset $\{(1,1),(2,2),(3,3)\}$ and fix $\theta_0 = 0$, leaving only θ_1 free. Our hypothesis becomes $h_\theta(x) = \theta_1 x$–a line through the origin.

Testing Different Parameter Values

For different values of θ_1, we can compute the cost:
When $\theta_1 = 1$: The hypothesis is $h(x) = x$, giving predictions $(1,2,3)$ that exactly match the targets $(1,2,3)$. All errors are zero, so:

$$J(0,1) = \frac{1}{6}[(1-1)^2 + (2-2)^2 + (3-3)^2] = 0$$

When $\theta_1 = 0.5$: The hypothesis is $h(x) = 0.5x$, giving predictions $(0.5, 1, 1.5)$. Now we have errors:

$$J(0,0.5) = \frac{1}{6}[(0.5-1)^2 + (1-2)^2 + (1.5-3)^2] = \frac{1}{6}[0.25 + 1 + 2.25] = \frac{3.5}{6} \approx 0.58$$

When $\theta_1 = 0$: The hypothesis is $h(x) = 0$, predicting zero for all inputs:

$$J(0,0) = \frac{1}{6}[1^2 + 2^2 + 3^2] = \frac{14}{6} \approx 2.33$$

As we vary θ_1 continuously, $J(0,\theta_1)$ traces out a parabola with its minimum at $\theta_1 = 1$–exactly where our hypothesis perfectly fits the data.

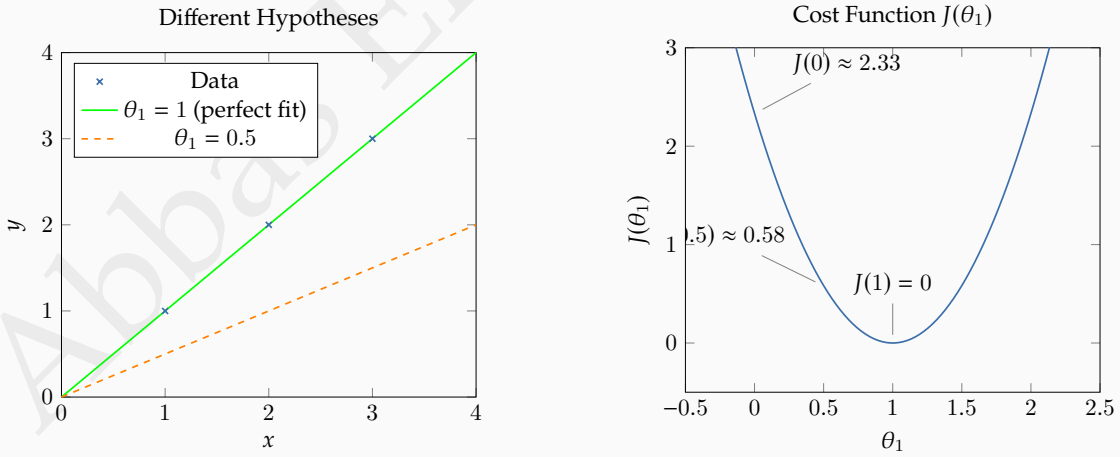


Figure 4: *Left: Different hypotheses with varying slopes. Right: The corresponding cost function is a parabola, minimized when the hypothesis perfectly fits the data.*

5.2 The General Case: Two Parameters

In practice, both θ_0 and θ_1 are free to vary. Now the cost function $J(\theta_0, \theta_1)$ is a surface in three-dimensional space. For the convex MSE cost, this surface resembles a bowl or paraboloid.

While 3D surface plots can be informative, **contour plots** often provide clearer insight. A

contour plot shows curves of constant cost value in the (θ_0, θ_1) plane–analogous to elevation contours on a topographic map. Each contour line connects parameter values that yield the same cost. Contours closer to the center represent lower cost values, with the very center marking the optimal parameters.

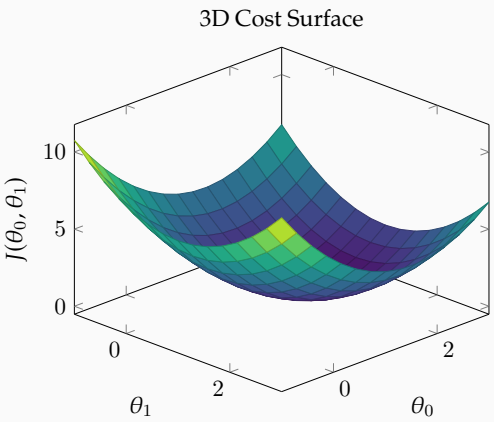


Figure 5: *The cost function forms a bowl-shaped surface with a unique global minimum at the bottom.*

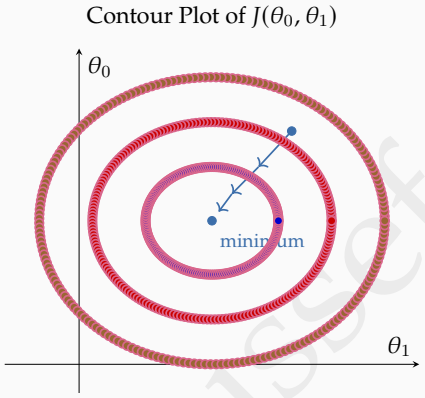


Figure 6: *Contour plot showing curves of equal cost. The trajectory illustrates how gradient descent navigates toward the minimum at the center.*

These visualizations reveal why optimization is necessary. For simple problems we might guess reasonable parameter values, but for higher-dimensional problems (with many features), systematically finding the minimum requires algorithmic approaches.

6 Gradient Descent: Navigating to the Minimum

We now turn to the central algorithmic question: how do we actually find the parameter values that minimize the cost function? While we will later discuss a direct analytical solution (the normal equation), the iterative algorithm called **gradient descent** is more general, scalable, and fundamental to modern machine learning.

6.1 The Core Idea

Imagine standing in a mountainous landscape while blindfolded, trying to reach the valley floor. A sensible strategy is to feel the slope of the ground beneath your feet and take a step in the direction of steepest descent. Repeat this process, and you'll eventually reach a local minimum.

Gradient descent formalizes this intuition. At each step, we:

1. Compute the *gradient* $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$–a vector of partial derivatives indicating the direction of steepest ascent
2. Take a step in the *opposite* direction (steepest descent)
3. The step size is controlled by the *learning rate* α

Gradient Descent – General Form

Initialize parameters $\boldsymbol{\theta}$ (e.g., to small random values or zeros)
Repeat until convergence:

1. Compute the gradient: $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. Update parameters: $\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3. Check convergence criteria

The update rule for individual parameters is:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}), \quad \text{for all } j$$

Critical: Simultaneous Updates

All parameters must be updated *simultaneously*. This means you first compute all the new parameter values using the *current* parameters, then update them all at once. Updating θ_0 first and using its new value to compute the update for θ_1 would be incorrect and can prevent convergence.

6.2 Computing the Gradient for Linear Regression

To apply gradient descent to linear regression with MSE cost, we need the partial derivatives of $J(\boldsymbol{\theta})$. Let's derive these for the univariate case, then generalize.

Starting with $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$ where $h_\theta(x) = \theta_0 + \theta_1 x$:

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{\partial}{\partial \theta_j} \left[\frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2 \right]$$

Applying the chain rule:

$$= \frac{1}{2m} \sum_{i=1}^{m} 2 \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right) \cdot \frac{\partial}{\partial \theta_j} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)$$

$$= \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot \frac{\partial}{\partial \theta_j} \left(\theta_0 + \theta_1 x^{(i)} \right)$$

For θ_0 (the intercept): $\frac{\partial}{\partial \theta_0}(\theta_0 + \theta_1 x^{(i)}) = 1$

$$\frac{\partial}{\partial \theta_0} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$

For θ_1 (the slope): $\frac{\partial}{\partial \theta_1}(\theta_0 + \theta_1 x^{(i)}) = x^{(i)}$

$$\frac{\partial}{\partial \theta_1} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

This gives us the complete **batch gradient descent** update rule:

Batch Gradient Descent for Linear Regression

Repeat until convergence: {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot x^{(i)}$$

}

The term "batch" refers to using all m training examples in each update step–we compute the gradient as an average over the entire "batch" of data.

6.3 When to Stop: Convergence Criteria

How do we know when gradient descent has "converged"–when it has found a good enough solution that we can stop iterating? Several practical criteria exist:

Common Convergence Criteria

1. **Fixed iterations**: Simply run for a predetermined number of steps (e.g., 1000 iterations). Simple but risks stopping too early or wasting computation by running too long.
2. **Cost tolerance**: Stop when the cost changes very little between iterations:

$$|J(\boldsymbol{\theta}^{(t)}) - J(\boldsymbol{\theta}^{(t-1)})| < \epsilon$$

   where ϵ is a small threshold like 10^{-6}. This adapts to the problem–it stops when progress becomes negligible.
3. **Gradient norm tolerance**: Stop when the gradient becomes very small:

$$\|\nabla J(\boldsymbol{\theta})\| < \epsilon$$

   Since the gradient is zero at the minimum, a very small gradient indicates we're close.
4. **Parameter change tolerance**: Stop when parameter updates become tiny:

$$\|\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)}\| < \epsilon$$

In practice, combining multiple criteria (e.g., a maximum iteration limit plus a cost tolerance) provides robustness.

6.4 Choosing the Learning Rate

The **learning rate** α controls how large a step we take at each iteration. Its choice profoundly affects both whether gradient descent converges and how quickly.

If α is too small, gradient descent will eventually reach the minimum, but it will take an extremely long time–thousands or millions of iterations that could have been avoided.

If α is too large, the updates overshoot the minimum. Parameters oscillate wildly or diverge, with the cost actually *increasing* over iterations rather than decreasing.

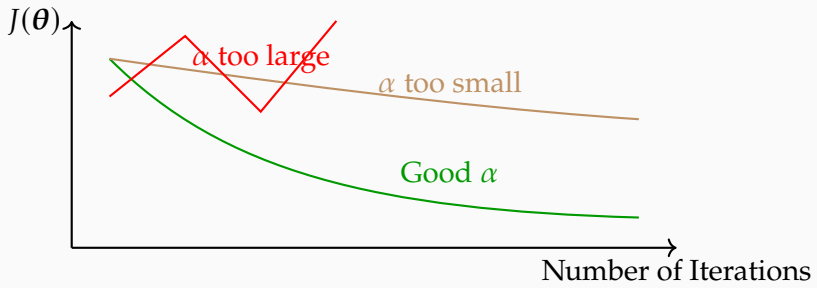


Figure 7: *Cost vs. iterations for different learning rates. A well-chosen α leads to steady decrease. Too small causes slow progress; too large causes instability.*

The primary debugging tool is a plot of $J(\boldsymbol{\theta})$ versus iteration number. For correctly implemented gradient descent with an appropriate learning rate, this plot should show a monotonic decrease, with the cost eventually flattening as it approaches the minimum.

A practical approach to selecting α: try a range of values on a logarithmic scale:

$$\ldots, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, \ldots$$

Start small, observe the cost plot, and increase by factors of roughly 3 until finding a value that converges quickly without diverging.

6.5 Variants of Gradient Descent

The "batch" version we've described uses all m training examples to compute each gradient. This is one of three main variants that differ in how many examples are used per update.

Three Flavors of Gradient Descent

Batch Gradient Descent (BGD) uses the entire dataset:

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \left(h_{\boldsymbol{\theta}}(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

Advantage: Stable, direct path toward minimum with guaranteed convergence for convex problems
Disadvantage: Computationally expensive for large datasets

Stochastic Gradient Descent (SGD) uses a single randomly chosen training example:

$$\theta_j := \theta_j - \alpha \left(h_{\boldsymbol{\theta}}(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

Advantage: Extremely fast per update; can escape shallow local minima
Disadvantage: Noisy updates that bounce around the minimum rather than converging smoothly

Mini-Batch Gradient Descent uses a small random subset (mini-batch) of size b (typically 32, 64, or 128):

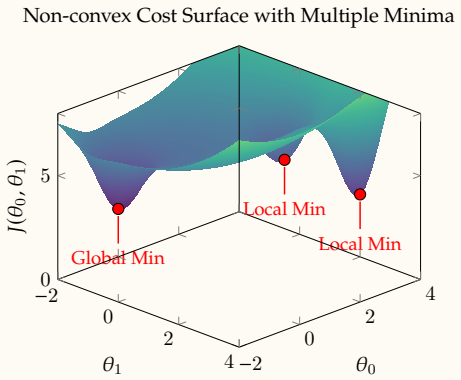
$$\theta_j := \theta_j - \alpha \frac{1}{b} \sum_{k=1}^{b} \left(h_{\boldsymbol{\theta}}(x^{(k)}) - y^{(k)} \right) x_j^{(k)}$$

Advantage: Balances stability and speed; works well with GPU acceleration
Disadvantage: Adds batch size as another hyperparameter to tune

Mini-batch GD combines advantages of both extremes and is the most common choice in modern machine learning, especially when combined with hardware acceleration (GPUs excel at processing mini-batches).

Escaping Local Minima

In non-convex problems, the cost function may have multiple local minima. Batch GD can get trapped in shallow ones, but the stochastic updates in SGD introduce noise that allows the algorithm to escape and continue searching for better solutions. This is particularly useful in neural networks, where the cost surface is highly non-convex. (For linear regression, this is irrelevant since the MSE cost is convex with only one global minimum.)



7 Walking Through an Example

Theory solidifies through concrete application. Let's carefully work through gradient descent on a simple dataset, computing each step by hand to see exactly how the algorithm operates.

Predicting Exam Score from Study Hours

Suppose we want to predict a student's exam score based on hours spent studying:

Hours (x)	Score (y)
1	2
2	4
3	6

Our hypothesis is $h_\theta(x) = \theta_0 + \theta_1 x$, and we'll use learning rate $\alpha = 0.1$.
Initialization: Start with $\theta_0 = 0, \theta_1 = 0$.
Initial hypothesis: $h_\theta(x) = 0$ (predicting zero regardless of study hours)
Initial predictions: For all three examples, $\hat{y} = 0$
Errors (residuals): $[0-2, 0-4, 0-6] = [-2, -4, -6]$
Initial cost: $J(0,0) = \frac{1}{6}[4+16+36] = \frac{56}{6} \approx 9.33$
Iteration 1: Apply the update rules.
For θ_0:

$$\theta_0^{new} = 0 - 0.1 \times \frac{1}{3}[(-2) + (-4) + (-6)]$$
$$= 0 - 0.1 \times \frac{-12}{3} = 0 - 0.1 \times (-4) = 0.4$$

For θ_1:

$$\theta_1^{new} = 0 - 0.1 \times \frac{1}{3}[(-2)(1) + (-4)(2) + (-6)(3)]$$

$$= 0 - 0.1 \times \frac{-28}{3} = 0 - 0.1 \times (-9.33) \approx 0.933$$

After iteration 1: $\theta_0 = 0.4, \theta_1 = 0.933$
Iteration 2: New hypothesis is $h_\theta(x) = 0.4 + 0.933x$
Predictions:

$$\begin{aligned} x = 1: \quad \hat{y} &= 0.4 + 0.933(1) = 1.333 \\ x = 2: \quad \hat{y} &= 0.4 + 0.933(2) = 2.266 \\ x = 3: \quad \hat{y} &= 0.4 + 0.933(3) = 3.199 \end{aligned}$$

Errors: $[1.333 - 2, 2.266 - 4, 3.199 - 6] = [-0.667, -1.734, -2.801]$
Cost: $J(0.4, 0.933) = \frac{1}{6}[0.445 + 3.006 + 7.846] \approx 1.88$
Updates:

$$\theta_0^{new} = 0.4 - 0.1 \times \frac{1}{3}(-5.202) \approx 0.573$$
$$\theta_1^{new} = 0.933 - 0.1 \times \frac{1}{3}(-12.538) \approx 1.351$$

After iteration 2: $\theta_0 = 0.573, \theta_1 = 1.351$
The pattern is clear: with each iteration, parameters move closer to the true values ($\theta_0 = 0, \theta_1 = 2$) and the cost decreases. Continuing this process would eventually converge to the optimal solution.

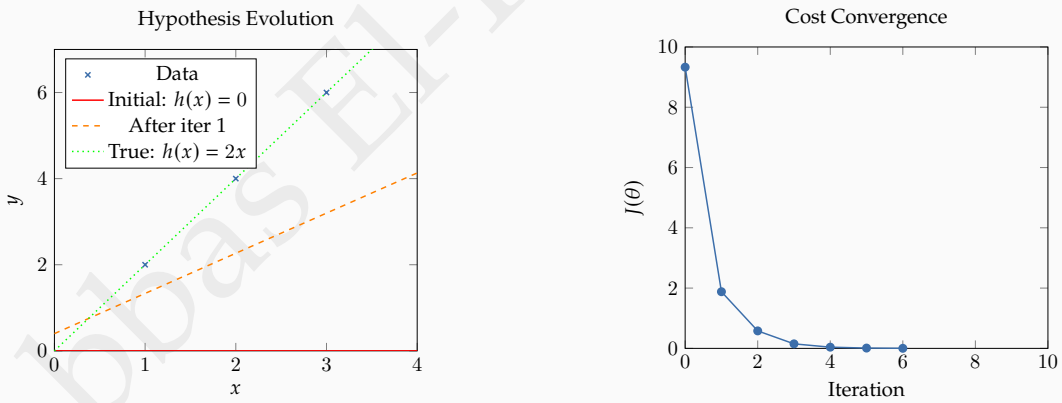


Figure 8: *Left: The hypothesis line evolves toward the data with each iteration. Right: The cost decreases exponentially, confirming convergence.*

8 Scaling to Multiple Features

Real-world prediction problems rarely depend on a single variable. House prices depend on size, but also on the number of bedrooms, number of bathrooms, age, location, and numerous other factors. This is **multivariate** or **multiple** linear regression.

8.1 Extended Notation

We now generalize our notation to handle n features. The hypothesis generalizes to a linear combination of all features:

$$h_\theta(\boldsymbol{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Each parameter θ_j (for $j \geq 1$) represents the expected change in the output for a one-unit increase in feature j, holding all other features constant. The intercept θ_0 represents the baseline prediction when all features are zero.

Interpreting Parameters in Context

For house price prediction with multiple features:

- θ_0: Baseline price of a house
- θ_1: Price change per square foot (if x_1 is size)
- θ_2: Price change per bedroom (if x_2 is number of bedrooms)
- θ_3: Price change per year of age (if x_3 is age)

For example, if $\theta_1 = 0.15$ (with price in thousands of dollars), each additional square foot increases the predicted price by \$150.

8.2 Vectorized Formulation

As before, we introduce $x_0 = 1$ for all examples, allowing us to write the hypothesis compactly:

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1}, \qquad \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

$$h_\theta(\boldsymbol{x}) = \boldsymbol{\theta}^\top \boldsymbol{x} = \sum_{j=0}^{n} \theta_j x_j$$

The cost function remains structurally identical:

$$J(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} \left(\boldsymbol{\theta}^\top \boldsymbol{x}^{(i)} - y^{(i)} \right)^2$$

8.3 Gradient Descent with Multiple Features

The gradient descent update rule extends naturally–we now update each of $n+1$ parameters:

Multivariate Batch Gradient Descent

Repeat until convergence: {

$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(\boldsymbol{x}^{(i)}) - y^{(i)} \right) \cdot x_j^{(i)}$ for $j = 0, 1, \ldots, n$

}

Vectorized Implementation:
Define the design matrix $X \in \mathbb{R}^{m\times(n+1)}$ with rows $(\boldsymbol{x}^{(i)})^\top$, and target vector $\boldsymbol{y} \in \mathbb{R}^m$.
The gradient is: $\nabla J(\boldsymbol{\theta}) = \frac{1}{m} X^\top (X\boldsymbol{\theta} - \boldsymbol{y})$
The vectorized update is: $\boldsymbol{\theta} := \boldsymbol{\theta} - \frac{\alpha}{m} X^\top (X\boldsymbol{\theta} - \boldsymbol{y})$

This single line replaces what would otherwise be nested loops over features and training examples–a dramatic simplification enabled by vectorization.

8.4 Concrete Example

House Price Prediction with Two Features

Consider predicting house prices with two features: size and number of bedrooms.

Size (x_1)	Bedrooms (x_2)	Price (y)
1000	2	200
1500	3	300
2000	4	400

The design matrix and target vector are:

$$X = \begin{bmatrix} 1 & 1000 & 2 \\ 1 & 1500 & 3 \\ 1 & 2000 & 4 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} 200 \\ 300 \\ 400 \end{bmatrix}$$

Starting from $\boldsymbol{\theta} = [0, 0, 0]^\top$ with $\alpha = 0.0001$:
Prediction errors:

$$X\boldsymbol{\theta} - \boldsymbol{y} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 200 \\ 300 \\ 400 \end{bmatrix} = \begin{bmatrix} -200 \\ -300 \\ -400 \end{bmatrix}$$

Gradient:

$$\nabla J(\boldsymbol{\theta}) = \frac{1}{3} X^\top \begin{bmatrix} -200 \\ -300 \\ -400 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \\ 1000 & 1500 & 2000 \\ 2 & 3 & 4 \end{bmatrix} \begin{bmatrix} -200 \\ -300 \\ -400 \end{bmatrix} = \begin{bmatrix} -300 \\ -483333.33 \\ -966.67 \end{bmatrix}$$

Updated parameters:

$$\boldsymbol{\theta}^{new} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} - 0.0001 \times \begin{bmatrix} -300 \\ -483333.33 \\ -966.67 \end{bmatrix} = \begin{bmatrix} 0.03 \\ 48.33 \\ 0.097 \end{bmatrix}$$

Notice the vastly different magnitudes in the gradient components (-300 vs -483333). This disparity stems from the different scales of the features (bedrooms are small integers while size is in thousands). Such scale differences can severely hamper gradient descent–a problem we address next.

9 Feature Scaling: Accelerating Convergence

When features live on vastly different scales–say, house size ranging from 500 to 5000 square feet while number of bedrooms ranges from 1 to 5–gradient descent can become painfully slow. Understanding why requires examining the geometry of the cost function.

9.1 The Problem: Elongated Contours

When features have different scales, the cost function's contours become highly elongated ellipses rather than circles. Parameters associated with large-scale features need small adjustments (since small changes in those parameters cause large changes in predictions), while parameters for small-scale features need large adjustments.

This mismatch forces gradient descent to zigzag back and forth across narrow valleys rather than proceeding directly toward the minimum. What might take dozens of iterations with scaled features can require thousands without scaling.

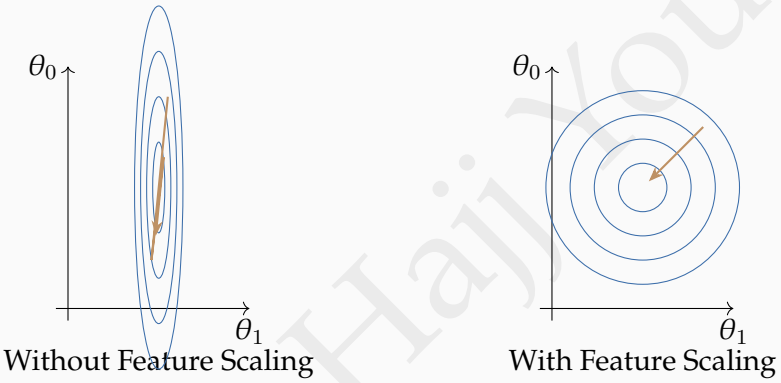


Figure 9: *Left: Unscaled features create elongated contours, causing gradient descent to take an inefficient zigzag path. Right: Scaled features produce circular contours, enabling direct descent to the minimum.*

9.2 Scaling Techniques

The goal of feature scaling is to transform all features to approximately the same range, typically around $[-1, 1]$ or $[0, 1]$.

Common Scaling Methods

Min-Max Normalization scales features to the range $[0, 1]$:

$$x_j := \frac{x_j - \min(x_j)}{\max(x_j) - \min(x_j)}$$

This preserves the original distribution's shape while compressing it into a fixed range.
Standardization (Z-score Normalization) transforms features to have zero mean and unit variance:

$$x_j := \frac{x_j - \mu_j}{\sigma_j}$$

where $\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$ is the mean and $\sigma_j = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2}$ is the standard deviation. This is the most commonly used technique in machine learning.

Mean Normalization centers features around zero:

$$x_j := \frac{x_j - \mu_j}{\max(x_j) - \min(x_j)}$$

Critical: Scaling Workflow

Scaling parameters (mean, standard deviation, min, max) must be computed using *only the training data*, then applied consistently to validation and test sets. The correct workflow is:

1. Compute scaling parameters (μ_j, σ_j) from training data
2. Transform training data using these parameters
3. Transform validation and test data using the *same* parameters
4. Never scale the intercept term $x_0 = 1$

Computing separate scaling parameters for test data would "leak" information and give an unrealistic assessment of model performance.

10 Understanding Training Dynamics: Epochs and Iterations

When discussing training, precise terminology helps avoid confusion. An **epoch** is one complete pass through the entire training dataset. The number of parameter updates per epoch depends on the gradient descent variant used.

Epochs Across Gradient Descent Variants

- **Batch GD**: One epoch = one parameter update (gradient computed over all examples)
- **Stochastic GD**: One epoch = m parameter updates (one per example)
- **Mini-batch GD**: One epoch = m/b parameter updates (one per mini-batch of size b)

Table 1: *Relationship between gradient descent variants and updates per epoch*

Variant	Updates per Epoch	Examples per Update
Batch GD	1	All m
Stochastic GD	m	1
Mini-batch GD	m/b	b

Training typically requires many epochs–parameters rarely converge after seeing the data just once. In SGD and mini-batch GD, it's standard practice to shuffle the dataset at the start of each epoch, ensuring that mini-batches are different each time through the data. This randomization helps prevent the algorithm from getting stuck and generally improves convergence.

11 The Normal Equation: A Direct Solution

Gradient descent is iterative–it approaches the minimum through a sequence of steps. For linear regression, there also exists a *direct*, analytical solution that computes the optimal parameters in one step: the **Normal Equation**.

11.1 Deriving the Closed-Form Solution

Recall that we can write the cost function in matrix form as:

$$J(\boldsymbol{\theta}) = \frac{1}{2m}(X\boldsymbol{\theta} - \boldsymbol{y})^\top (X\boldsymbol{\theta} - \boldsymbol{y})$$

where $X \in \mathbb{R}^{m\times(n+1)}$ is the design matrix and $\boldsymbol{y} \in \mathbb{R}^m$ is the target vector.
Expanding this quadratic form:

$$J(\boldsymbol{\theta}) = \frac{1}{2m}(\boldsymbol{\theta}^\top X^\top X\boldsymbol{\theta} - 2\boldsymbol{\theta}^\top X^\top \boldsymbol{y} + \boldsymbol{y}^\top \boldsymbol{y})$$

Taking the gradient with respect to $\boldsymbol{\theta}$ and applying matrix calculus rules:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}(X^\top X\boldsymbol{\theta} - X^\top \boldsymbol{y})$$

At the minimum, the gradient must equal zero:

$$X^\top X\boldsymbol{\theta} - X^\top \boldsymbol{y} = 0$$

Solving for $\boldsymbol{\theta}$:

$$X^\top X\boldsymbol{\theta} = X^\top \boldsymbol{y}$$

The Normal Equation

Provided that $X^\top X$ is invertible, the optimal parameters are:

$$\boldsymbol{\theta}^* = (X^\top X)^{-1} X^\top \boldsymbol{y}$$

This formula directly computes the optimal parameters without any iterations, learning rate choices, or convergence criteria.

11.2 Gradient Descent vs. Normal Equation

Why would we ever use gradient descent when this direct formula exists? The answer lies in computational complexity and generality.

Table 2: *Comparing gradient descent and the normal equation*

Gradient Descent	Normal Equation
Requires choosing learning rate α	No hyperparameters
Requires many iterations	Computes answer in one step
Works well even for very large n	Slow when $n > 10{,}000$
Complexity $O(mn)$ per iteration	Complexity $O(n^3)$
Generalizes to other models and cost functions	Only works for linear regression with MSE

Computing $(X^\top X)^{-1}$ involves matrix inversion, which has time complexity $O(n^3)$ where n is the number of features. For small n (say, $n < 10{,}000$), this is perfectly tractable. But for high-dimensional problems with tens or hundreds of thousands of features, the normal equation becomes prohibitively expensive.

Gradient descent, by contrast, has complexity $O(mn)$ per iteration. While many iterations may be needed, each is fast. For large n, this scales much better than $O(n^3)$.

Additionally, the normal equation is specific to least squares regression. It doesn't extend to other cost functions (like logistic regression's cross-entropy) or to more complex models (like neural networks). Gradient descent, being a general-purpose optimization algorithm, works across all these domains.

11.3 When $X^\top X$ is Non-Invertible

The normal equation assumes $X^\top X$ is invertible. This can fail in two common scenarios:

- **Redundant features**: If features are linearly dependent (e.g., having both size in square feet and size in square meters), $X^\top X$ becomes singular.
- **Too many features**: When $n \geq m$ (more features than examples), $X^\top X$ cannot have full rank.

Practical solutions include:

- Delete redundant features through careful data analysis
- Use *regularization* (covered in later chapters), which adds a term to ensure invertibility
- Compute the *pseudoinverse* X^+, which provides a numerically stable solution even when $X^\top X$ is singular

Most numerical libraries offer robust pseudoinverse implementations (e.g., `numpy.linalg.pinv` in Python) that handle these edge cases automatically.

12 Polynomial Regression: Capturing Nonlinear Relationships

Linear regression assumes the relationship between features and target is linear–a straight line in 2D, a plane in 3D, a hyperplane in higher dimensions. But many real phenomena are inherently nonlinear: growth rates that accelerate or decelerate, relationships that peak and decline, curves rather than lines.

Polynomial regression extends linear regression to capture these patterns while preserving its mathematical structure and algorithmic simplicity. The key insight: we transform the feature space by creating polynomial terms, then apply standard linear regression to these transformed features.

12.1 The Mathematical Framework

For a single feature x, a polynomial of degree d takes the form:

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \cdots + \theta_d x^d$$

Polynomial Regression

A form of regression analysis in which the relationship between an independent variable x and a dependent variable y is modeled as an n-th-degree polynomial. Despite modeling a non-linear curve, it is considered a special case of multiple linear regression because the model is *linear in its parameters* $\boldsymbol{\theta}$.
By defining new features:

$$x_1 = x, \quad x_2 = x^2, \quad x_3 = x^3, \quad \ldots, \quad x_d = x^d$$

we convert the problem to standard multivariate linear regression:

$$h_\theta(\boldsymbol{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_d x_d = \boldsymbol{\theta}^\top \boldsymbol{x}$$

All our previous machinery–gradient descent, normal equation, cost function–applies without modification.

Fuel Efficiency Modeling

Consider modeling a car's fuel efficiency as a function of speed. Physical principles suggest that efficiency increases with speed up to an optimal point, then decreases–a relationship poorly captured by linear models. A quadratic model provides a natural fit:

$$\text{FuelEfficiency} = \theta_0 + \theta_1 \times \text{speed} + \theta_2 \times \text{speed}^2$$

This captures the characteristic inverted parabola shape, with $\theta_2 < 0$ ensuring efficiency peaks at moderate speeds and declines at very high speeds.

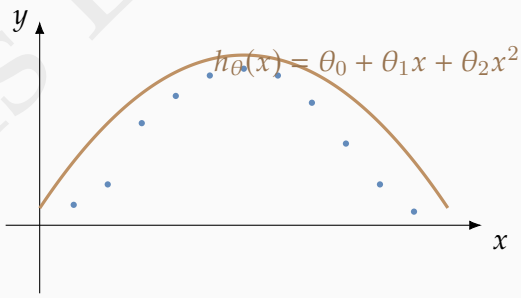


Figure 10: *A quadratic polynomial capturing a non-linear relationship while remaining linear in parameters* $\boldsymbol{\theta}$.

12.2 Mathematical Formulation

For m training examples with polynomial degree d, we construct the design matrix $X \in \mathbb{R}^{m \times (d+1)}$:

$$X = \begin{bmatrix} 1 & x^{(1)} & (x^{(1)})^2 & \cdots & (x^{(1)})^d \\ 1 & x^{(2)} & (x^{(2)})^2 & \cdots & (x^{(2)})^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x^{(m)} & (x^{(m)})^2 & \cdots & (x^{(m)})^d \end{bmatrix}$$

The optimization problem remains identical to linear regression, with the mean squared error cost function:

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\|X\boldsymbol{\theta} - \boldsymbol{y}\|_2^2 = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

12.3 Solution Methods

Since polynomial regression reduces to linear regression on transformed features, we can employ our standard solution techniques.

The Normal Equation: For datasets where the number of features is moderate (typically $d < 10{,}000$), the normal equation provides a direct solution:

$$\boldsymbol{\theta} = (X^\top X)^{-1}X^\top \boldsymbol{y}$$

Gradient Descent: For large-scale problems or high-degree polynomials, gradient descent offers better scalability. The vectorized update is:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \frac{\alpha}{m}X^\top(X\boldsymbol{\theta} - \boldsymbol{y})$$

12.4 Critical Considerations

Feature Scaling: Absolutely Essential

Feature scaling becomes **critically important** in polynomial regression. Without scaling, if the original feature x ranges from $[1, 1000]$, the polynomial features will have vastly different scales:

- $x \in [1, 10^3]$
- $x^2 \in [1, 10^6]$
- $x^3 \in [1, 10^9]$

These dramatically different scales cause:

1. **Numerical instability**: Risk of overflow or catastrophic cancellation
2. **Slow convergence**: Gradient descent may require millions of iterations
3. **Ill-conditioning**: The matrix $X^\top X$ becomes nearly singular

Solution: Always normalize features *before* creating polynomial terms using standardization:

$$x_{\text{scaled}} = \frac{x - \mu_x}{\sigma_x}$$

12.5 Model Selection and Best Practices

Practical Guidelines for Polynomial Regression

1. **Start Simple**: Begin with degree 2 or 3 before exploring higher degrees. This often provides a good balance between model flexibility and simplicity.
2. **Feature Engineering**: Consider domain knowledge when selecting polynomial terms.

Not all features may benefit from polynomial expansion.

3. **Choosing the Degree**: The polynomial degree d is a critical hyperparameter. A low degree may *underfit* the data, while a very high degree can lead to *overfitting*–where the model fits training noise but fails to generalize. Use cross-validation or a validation set to select the optimal degree.
4. **Monitor Convergence**: When using gradient descent, always plot the cost $J(\boldsymbol{\theta})$ versus iterations. The cost should decrease monotonically. If it increases or oscillates, reduce the learning rate α.
5. **Regularization**: For high-degree polynomials ($d > 3$) that are prone to overfitting, consider adding L2 (Ridge) or L1 (Lasso) regularization to penalize large parameter values and improve generalization.
6. **Computational Efficiency**: Cache computed polynomial features rather than recomputing them at each iteration to improve performance.

Polynomial regression demonstrates the power of feature engineering: by transforming the input space rather than modifying the learning algorithm, we can capture complex non-linear relationships while preserving the computational efficiency and theoretical elegance of linear methods. The key is to balance model complexity with generalization ability, guided by validation performance and domain expertise.

13 Chapter Summary and Key Takeaways

This chapter introduced linear regression, a foundational supervised learning algorithm that predicts continuous outputs by learning a linear relationship between features and targets. We covered the complete lifecycle: from problem formulation through cost function definition, algorithmic optimization via gradient descent, and analytical solution via the normal equation, concluding with extensions to multiple features and polynomial models.

Essential Concepts and Formulas

The Hypothesis Function: Our predictive model

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{\theta}^\top \boldsymbol{x} = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n$$

The Cost Function: Quantifies prediction error

$$J(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)} \right)^2$$

Gradient Descent: Iteratively minimizes cost

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \frac{\alpha}{m} X^\top (X\boldsymbol{\theta} - \boldsymbol{y})$$

The Normal Equation: Direct analytical solution

$$\boldsymbol{\theta}^* = (X^\top X)^{-1} X^\top \boldsymbol{y}$$

Core Insights:

- Linear regression models linear relationships between features and continuous outputs, forming the foundation for more advanced techniques
- The MSE cost function measures average squared prediction error and has desirable mathematical properties including convexity and statistical justification
- Gradient descent iteratively improves parameters by following the negative gradient; its success depends critically on learning rate selection and feature scaling
- Convexity of the MSE cost guarantees gradient descent converges to the global minimum regardless of initialization
- Feature scaling transforms features to similar ranges, dramatically accelerating convergence by preventing elongated cost contours
- Polynomial regression extends linear models to nonlinear relationships through feature transformation while preserving linear-in-parameters structure
- The choice between gradient descent and the normal equation depends on problem scale: normal equation for small n, gradient descent for large n
- Vectorization enables efficient implementation, scalability, and readable code by expressing operations as matrix computations

Linear regression serves as more than just a practical algorithm–it provides conceptual foundations that permeate machine learning. The interplay between cost functions and optimization, the bias-variance tradeoff implicit in model complexity choices, the crucial role of feature engineering, and the power of vectorization all reappear throughout the field. Master these concepts with linear regression, and you have built a solid foundation for understanding neural networks, support vector machines, and the full spectrum of modern machine learning.

A Notation Reference

For quick reference, this table summarizes the notation used throughout the chapter:

Table 3: *Complete notation guide*

Symbol	Definition
m	Number of training examples
n	Number of features
x	Input variable or feature (scalar)
$\boldsymbol{x}$	Feature vector $\in \mathbb{R}^{n+1}$ (includes $x_0 = 1$)
X	Design matrix $\in \mathbb{R}^{m \times (n+1)}$
y	Target variable (scalar)
$\boldsymbol{y}$	Target vector $\in \mathbb{R}^m$
$\hat{y}$	Predicted value
$(x^{(i)}, y^{(i)})$	i-th training example
$\boldsymbol{x}^{(i)}$	Feature vector for i-th example
$x_j^{(i)}$	Value of feature j in example i
θ_j	Parameter for feature j
$\boldsymbol{\theta}$	Parameter vector $\in \mathbb{R}^{n+1}$
$h_{\boldsymbol{\theta}}(\boldsymbol{x})$	Hypothesis function (model)
$J(\boldsymbol{\theta})$	Cost function (objective)
$\nabla J(\boldsymbol{\theta})$	Gradient of cost function
α	Learning rate (step size)
ϵ	Convergence tolerance threshold
b	Mini-batch size
$\mathcal{D}$	Training dataset
μ_j	Mean of feature j
σ_j	Standard deviation of feature j
$\mathcal{N}(0, \sigma^2)$	Normal distribution, mean 0, variance σ^2