Generalizing Gradient Descent

Gradient Descent **in Higher Dimensions**

1. Reminder: Gradient Descent in 1D

- We want to minimize some function J(θ).
- Update rule:

  $\theta := \theta - \alpha \frac{dJ}{d\theta}$
- Here:
  - α = learning rate (step size).
  - $\frac{dJ}{d\theta}$ = slope of the curve at θ.
- Idea: Move in the **opposite direction of the slope** because that leads downhill.

---

2. Generalizing to Higher Dimensions

Suppose θ is not just a scalar but a vector:

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} \in \mathbb{R}^{d+1}$$

The cost function is:

$J(\theta) : \mathbb{R}^{d+1} \to \mathbb{R}$

---

3. Gradient in Higher Dimensions

Instead of a single derivative, we take **partial derivatives** for each parameter:

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \frac{\partial J(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_d} \end{bmatrix}$$

This vector is called the **gradient** of J.

- Each component tells us how much the cost changes if we move a little in that direction.

---

4. Update Rule in Higher Dimensions

For each parameter θ_j (where j=0,1,2,...,d):

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

Or, in vector form:

$$\theta := \theta - \alpha \nabla_\theta J(\theta)$$

---

5. Intuition

- The gradient $\nabla_\theta J(\theta)$ points in the direction of **steepest ascent** (the fastest increase of J).
- Moving in the **negative gradient direction** gives us the direction of **steepest descent** (the fastest decrease of J).
- That's why gradient descent works: every step tries to lower the cost as quickly as possible.

✅ **Summary:**

In higher dimensions, gradient descent works by computing the gradient vector (all partial derivatives), then updating all parameters simultaneously by stepping in the opposite direction of the gradient. This ensures we move toward the valley (minimum) of the cost function.

Stochastic Gradient Descent (SGD)

🔷 1. Reminder: Batch Gradient Descent

- In **batch gradient descent**, we compute the gradient of the cost function using **all training examples**:

$$\theta := \theta - \alpha \cdot \nabla_\theta J(\theta)$$

- Very accurate update, but **slow** when the dataset is huge (millions of samples).

🔷 2. Stochastic Gradient Descent (SGD)

- Instead of computing the gradient on the entire dataset, **SGD updates parameters using only one data point at a time**.
- Update rule for the i-th training example:

$$\theta := \theta + \alpha\big(y^{(i)} - h_\theta(x^{(i)})\big)x^{(i)}$$

where:

- $y^{(i)}$ = true label of example i
- $h_\theta(x^{(i)})$ = prediction using current θ
- $x^{(i)}$ = input vector for example i
- α = learning rate

👉 Each update is **fast** (since it uses just one example).

🔷 3. How it Looks

- Loop: **for i = 1 to n**
  - Update θ immediately using the i-th example.
- Example with n=2:
  - First update with data point 1 → θ_1
  - Then update with data point 2 → θ_2
  - Then cycle back to point 1, and so on.

So θ keeps moving after **every single data point**.

🔷 4. Characteristics of SGD

- **Pros**:
  - Very fast for large datasets.
  - Can escape local minima (updates are noisy, which helps).
- **Cons**:
  - Updates are noisy → cost function curve looks "jittery."
  - Doesn't converge as smoothly as batch gradient descent.

🔷 5. Mini-batch Gradient Descent

- A compromise between **batch** and **stochastic**.
- Instead of **1 example** or **all examples**, we use a **small batch** (say 32, 64, 100 examples).
- Update rule:

$$\theta := \theta - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta J^{(i)}(\theta)$$

- **Most commonly used in practice** (especially in deep learning).

✅ **Summary:**

- **Batch GD**: Accurate, but slow (uses all data each step).
- **SGD**: Fast, noisy (uses 1 point each step).
- **Mini-batch GD**: Best of both worlds, widely used in practice.