

# Conventional Programming vs Machine Learning



conventional programming Vs machine learning



Conventional programming and machine learning differ in how problems are solved and how solutions are developed:

## Conventional Programming:

1. **Rule-Based Approach:** In conventional programming, you provide explicit instructions to the computer. Programmers define a set of rules (algorithms) for the system to follow. For example, if you want to create a program that converts Celsius to Fahrenheit, you write a formula ( $F = C * 9/5 + 32$ ).
2. **Clear Logic Flow:** The logic flow is predictable and based on conditional statements (if-else), loops, and predefined rules.
3. **Deterministic Outcomes:** The output is always determined by the code and input. Given the same input, the system will always produce the same output.
4. **Examples:** Applications like web servers, database systems, and calculators follow this rule-based approach.

## Machine Learning:

1. **Data-Driven Approach:** Instead of writing explicit rules, you provide a large amount of data and let the system learn patterns from it. The system learns the rules based on the data through algorithms like decision trees, neural networks, or support vector machines.
2. **Adaptive Logic Flow:** The system adapts over time as it trains on more data. The logic is not explicitly defined by the programmer but is inferred from data.
3. **Probabilistic Outcomes:** Machine learning systems predict outputs based on probabilities. Given the same input, the output may vary because the system relies on the patterns it has learned.
4. **Examples:** Applications like recommendation systems, voice recognition, and image classification rely on machine learning.

In summary, **conventional programming** requires explicit rules, while **machine learning** allows systems to learn from data and infer rules themselves.



Human learning vs machine learning



Human learning and machine learning share the concept of learning from experience, but they differ in how they acquire, process, and apply knowledge. Here's a breakdown of the key differences:

## Human Learning:

### 1. Experience-Based and Intuitive:

- Humans learn through experiences, interactions, and observations in the real world. Learning can be conscious (studying) or unconscious (observing).
- Intuition plays a large role in human learning, allowing people to make inferences without complete data.

### 2. Adaptability and Context Understanding:

- Humans can understand and adapt to new contexts with limited examples. We apply prior knowledge, reasoning, and creativity to solve problems.
- Humans can generalize knowledge to new, unseen problems (transfer learning), even when situations are significantly different.

### 3. Learning Over Time:

- Learning is continuous throughout life. Human brains can refine understanding through reflection, abstract thinking, and social interactions.
- Humans use various learning techniques such as memorization, understanding concepts, hands-on practice, and observation.

### 4. Emotional and Social Factors:

- Emotions, motivation, curiosity, and social influences affect how humans learn. Group dynamics and collaboration can accelerate learning.
- Humans can be self-aware, knowing when they understand something or when they need to learn more.

### 5. Knowledge Representation:

- Humans use abstract concepts, language, symbols, and visuals to represent knowledge. Learning involves both theoretical and practical understanding.

---

## Machine Learning:

### 1. Data-Driven and Algorithmic:

- Machines learn by processing large amounts of data through algorithms. They find patterns in data and make decisions based on those patterns.
- Machine learning models rely on statistical methods and mathematical optimization to improve their performance over time.

### 2. Narrow Scope and Specificity:

- Machine learning systems are designed for specific tasks, like recognizing images, translating languages, or predicting prices. They struggle with adapting to new contexts without additional training.
- Machines require large, structured datasets to learn, whereas humans can learn from fewer examples and generalize knowledge more easily.

### 3. Training and Feedback Loops:

- Machine learning involves training models through iterative processes, often using supervised learning (labeled data), unsupervised learning (finding patterns), or reinforcement learning (feedback-based learning).
- Machines improve by minimizing error rates or maximizing performance metrics over time, rather than understanding the task itself.

### 4. Objective and Data-Dependent:

- Machine learning models lack emotions, motivations, or curiosity. They purely rely on data, and their performance is limited by the quality and quantity of the data provided.
- Machines can be retrained with new data but don't possess self-awareness or reflection abilities.

### 5. Knowledge Representation:

- Machine learning systems represent knowledge in the form of numerical weights, patterns, or models. These representations are not as easily interpretable as human understanding.

## Summary of Key Differences:

Aspect	Human Learning	Machine Learning
Learning Source	Experiences, observations, interactions	Data, algorithms, feedback
Adaptability	High adaptability, can generalize across tasks	Task-specific, limited adaptability without retraining
Learning Approach	Intuitive, emotional, social	Statistical, data-driven
Speed of Learning	Can learn from small amounts of data and few examples	Requires large datasets and many iterations
Generalization	Can apply knowledge to new and unfamiliar tasks	Struggles to generalize without new training
Emotional Influence	Affected by emotions, motivations, and social context	No emotional or motivational influence
Knowledge Representation	Abstract concepts, language, symbols	Mathematical models, weights, and patterns

In essence, human learning is more flexible and nuanced, while machine learning excels at processing vast amounts of data to make predictions or decisions in narrowly defined tasks.



give me a simple math example to solve gradient descent



Let's take a simple quadratic function as an example to understand how gradient descent works.

### Problem:

We want to minimize the function  $f(x) = (x - 3)^2$ .

This is a simple convex function with a minimum at  $x = 3$ .

## Steps to Solve Gradient Descent:

### 1. Objective Function:

$$f(x) = (x - 3)^2$$

### 2. Gradient of the Function: The derivative (gradient) of $f(x)$ with respect to $x$ is:

$$f'(x) = 2(x - 3)$$

The gradient tells us the direction in which the function increases. To minimize the function, we move in the opposite direction of the gradient.

### 3. Choose a Learning Rate: Let's set a small learning rate $\alpha = 0.1$ .

### 4. Initial Guess: Suppose we start with an initial guess $x_0 = 0$ .

## Gradient Descent Algorithm:

At each iteration, we update  $x$  using the formula:

$$x_{\text{new}} = x_{\text{old}} - \alpha \cdot f'(x_{\text{old}})$$

### Iteration 1:

- $x_0 = 0$
- Gradient at  $x_0$ :  $f'(0) = 2(0 - 3) = -6$
- Update  $x_1 = x_0 - 0.1 \cdot (-6) = 0 + 0.6 = 0.6$

### Iteration 2:

- $x_1 = 0.6$
- Gradient at  $x_1$ :  $f'(0.6) = 2(0.6 - 3) = -4.8$
- Update  $x_2 = x_1 - 0.1 \cdot (-4.8) = 0.6 + 0.48 = 1.08$

### Iteration 3:

- $x_2 = 1.08$
- Gradient at  $x_2$ :  $f'(1.08) = 2(1.08 - 3) = -3.84$
- Update  $x_3 = x_2 - 0.1 \cdot (-3.84) = 1.08 + 0.384 = 1.464$

### Iteration 4:

- $x_3 = 1.464$
- Gradient at  $x_3$ :  $f'(1.464) = 2(1.464 - 3) = -3.072$
- Update  $x_4 = x_3 - 0.1 \cdot (-3.072) = 1.464 + 0.3072 = 1.7712$

As you continue this process,  $x$  will get closer and closer to the minimum, which is at  $x = 3$ .

## Conclusion:

In gradient descent, you start with an initial guess and iteratively update the value of  $x$  by subtracting the gradient multiplied by the learning rate. This allows you to gradually approach the minimum of the function.



For a collection of labelled/training examples,  $\{x_n, y_n\}$  from  $i=1$  to  $N$ . what does it mean?



In the context of machine learning, the notation  $\{x_n, y_n\}$  from  $i = 1$  to  $N$  represents a **collection of labeled training examples** used for supervised learning. Here's a breakdown of what each part means:

## Notation Breakdown:

- $\{x_n, y_n\}$ :
  - $x_n$ : This represents the input feature(s) (or data point) for the  $n$ -th example. It can be a single scalar value or a vector (if the data has multiple features).
  - $y_n$ : This represents the label or target value associated with the  $n$ -th example. In regression tasks,  $y_n$  could be a continuous value (e.g., house price), while in classification tasks, it could be a class label (e.g., 0 or 1).
- $n = 1$  to  $N$ :
  - $N$ : The total number of training examples in the dataset.
  - $n$  indexes the examples, meaning you have  $N$  pairs  $\{x_1, y_1\}, \{x_2, y_2\}, \dots, \{x_N, y_N\}$ , each representing a data point and its corresponding label.

## Meaning:

This notation signifies that you have a **training set** consisting of  $N$  examples, where:

- $x_n$  is the input data (also called features or attributes),
- $y_n$  is the corresponding output (label) that the model should predict.

## Example:

Let's say you're working on a house price prediction model:

- Each  $x_n$  could represent the features of a house (e.g., size, number of bedrooms), and
- $y_n$  could represent the price of that house.

In this context, you're training a model to learn the relationship between the features  $x_n$  and the target  $y_n$  by using these  $N$  labeled examples.

## Purpose:

The goal of machine learning is to find a function (hypothesis) that maps inputs  $x_n$  to outputs  $y_n$  as accurately as possible, based on this training data.



give me a step by step of how to go from linear regression to gradient descent



## Transition from Linear Regression to Gradient Descent

Linear regression aims to model the relationship between input features  $x_n$  and target values  $y_n$  using a linear equation. Gradient descent is an optimization algorithm that minimizes the cost function (or error) in linear regression. Here's a step-by-step guide on how to move from linear regression to applying gradient descent:

### 1. Define the Linear Regression Model

The goal in linear regression is to fit a line that minimizes the difference between the predicted values and the actual values. The linear regression model for one feature (univariate case) can be written as:

$$\hat{y}_n = \theta_0 + \theta_1 x_n$$

Where:

- $x_n$  is the input feature for the  $n$ -th data point.
- $\hat{y}_n$  is the predicted value for the  $n$ -th data point.
- $\theta_0$  is the y-intercept (bias).
- $\theta_1$  is the slope (weight or coefficient).

In the case of multiple features (multivariate), it generalizes to:

$$\hat{y}_n = \theta_0 + \theta_1 x_{n1} + \theta_2 x_{n2} + \dots + \theta_p x_{np}$$

Where  $p$  is the number of features, and  $\theta = (\theta_0, \theta_1, \dots, \theta_p)$  are the model parameters.

### 2. Define the Cost Function (Mean Squared Error)

The cost function quantifies how far off the predictions are from the actual target values. For linear regression, the most commonly used cost function is the **Mean Squared Error (MSE)**, which measures the average squared difference between predicted values  $\hat{y}_n$  and actual target values  $y_n$ :

$$J(\theta) = \frac{1}{2N} \sum_{n=1}^N (\hat{y}_n - y_n)^2$$

Where:

- $J(\theta)$  is the cost function.
- $N$  is the total number of training examples.
- $\hat{y}_n$  is the prediction for the  $n$ -th example using the current parameters  $\theta$ .
- $y_n$  is the actual label for the  $n$ -th example.

We add the factor  $\frac{1}{2}$  to make the math simpler when taking the derivative.

### 3. Compute the Gradients of the Cost Function

Gradient descent is used to minimize the cost function  $J(\theta)$  by updating the parameters  $\theta_0, \theta_1, \dots, \theta_p$  iteratively.

To do this, we need the **partial derivatives** of the cost function with respect to each parameter  $\theta_j$ :

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{N} \sum_{n=1}^N (\hat{y}_n - y_n) x_{nj}$$

For the bias term  $\theta_0$ , the derivative is:

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{N} \sum_{n=1}^N (\hat{y}_n - y_n)$$

For  $\theta_1$  (and other  $\theta_j$  in the case of multiple features):

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{N} \sum_{n=1}^N (\hat{y}_n - y_n) x_{nj}$$

These derivatives (or gradients) indicate the direction and magnitude of the change needed in the parameters to reduce the error.

### 4. Update the Parameters Using Gradient Descent

The parameters are updated iteratively using the gradient descent algorithm. The update rule for each parameter is:

$$\theta_j \leftarrow \theta_j - \alpha \cdot \frac{\partial J(\theta)}{\partial \theta_j}$$

Where:

- $\alpha$  is the **learning rate**, a small positive number that controls the step size.
- $\frac{\partial J(\theta)}{\partial \theta_j}$  is the gradient of the cost function with respect to parameter  $\theta_j$ .

The process is repeated until the cost function converges to a minimum or a maximum number of iterations is reached.

### 5. Iterative Steps (Algorithm)

Here's how the gradient descent algorithm works step-by-step:

1. **Initialize Parameters:** Start with random or zero values for  $\theta_0, \theta_1, \dots, \theta_p$ .
2. **Compute Predictions:** For each training example, compute the predicted value  $\hat{y}_n = \theta_0 + \theta_1 x_{n1} + \dots + \theta_p x_{np}$ .
3. **Compute the Cost Function:** Compute the cost function  $J(\theta)$  using the current parameters.
4. **Compute the Gradients:** Calculate the partial derivatives (gradients) of the cost function with respect to each parameter  $\theta_j$  (i.e.,  $\frac{\partial J(\theta)}{\partial \theta_j}$ ).
5. **Update the Parameters:** Update each parameter  $\theta_j$  using the gradient descent rule:
 
$$\theta_j \leftarrow \theta_j - \alpha \cdot \frac{\partial J(\theta)}{\partial \theta_j}$$
6. **Repeat:** Repeat steps 2-5 until the cost function converges (i.e., changes become small or reach a predefined number of iterations).

## 6. Stopping Criteria

The gradient descent algorithm continues updating the parameters until one of these conditions is met:

- The cost function  $J(\theta)$  decreases below a certain threshold (convergence).
- The gradient becomes very small, indicating that the algorithm has reached a local or global minimum.
- A predefined number of iterations (max iterations) is reached.

## Example Walkthrough

Let's say we have a simple dataset where we are trying to fit a line using gradient descent for linear regression.

### Dataset:

- $x = [1, 2, 3]$
- $y = [2, 2.5, 3.5]$

1. **Linear Model:**  $\hat{y} = \theta_0 + \theta_1 x$
2. **Initialize Parameters:** Start with  $\theta_0 = 0, \theta_1 = 0$
3. **Learning Rate:** Choose a learning rate  $\alpha = 0.1$

### Iteration 1:

1. **Compute Predictions:**  $\hat{y} = \theta_0 + \theta_1 x = 0$  (since both  $\theta_0$  and  $\theta_1$  are 0)
2. **Compute Cost Function:**



$$J(\theta) = \frac{1}{2N} \sum_{n=1}^N (\hat{y}_n - y_n)^2 = \frac{1}{6} ((0 - 2)^2 + (0 - 2.5)^2 + (0 - 3.5)^2) = 6.25$$

### 3. Compute Gradients:

- $\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{N} \sum (\hat{y}_n - y_n)$
- $\frac{\partial J(\theta)}{\partial \theta_1} = \frac{1}{N} \sum (\hat{y}_n - y_n) x_n$

### 4. Update Parameters:

- $\theta_0 \leftarrow \theta_0 - \alpha \cdot \frac{\partial J(\theta)}{\partial \theta_0}$
- $\theta_1 \leftarrow \theta_1 - \alpha \cdot \frac{\partial J(\theta)}{\partial \theta_1}$

Repeat these steps until the cost function converges to a minimum.

## Conclusion

By following these steps, gradient descent helps to iteratively update the model parameters  $\theta$  in linear regression, reducing the error and improving the fit of the model.



tell me the difference between  $x_n$  and  $x_{n1}$



The notations  $x_n$  and  $x_{n1}$  represent different things, typically used in the context of machine learning, data representation, and equations involving multiple features. Here's the distinction:

### 1. $x_n$ — Single Input for Example $n$

- $x_n$  refers to the **input (feature)** for the  $n$ -th training example in a dataset.
- If you have only **one feature** per example,  $x_n$  represents the input for the  $n$ -th data point. For example, in a dataset of house sizes,  $x_n$  could be the size of the  $n$ -th house.

Example (with one feature per data point):

- $x_1 = 1000$  (size of the 1st house in square feet)
- $x_2 = 1500$  (size of the 2nd house)

### 2. $x_{n1}$ — First Feature for Example $n$

- $x_{n1}$  refers to the **first feature** of the  $n$ -th example when there are **multiple features**. In cases where each data point has multiple attributes, such as house size and number of bedrooms, the features are indexed using a second subscript.

Example (with multiple features per data point):

- $x_{n1}$  is the first feature for the  $n$ -th example (e.g., house size).
- $x_{n2}$  is the second feature for the  $n$ -th example (e.g., number of bedrooms).

For example, for the  $n$ -th house:

- $x_{n1} = 1000$  (size of the  $n$ -th house in square feet)
- $x_{n2} = 3$  (number of bedrooms in the  $n$ -th house)

## Summary:

- $x_n$ : Refers to the input for the  $n$ -th example (used when there is only one feature per example).
- $x_{n1}$ : Refers to the **first feature** of the  $n$ -th example (used when there are multiple features per example).  $x_{n2}$ ,  $x_{n3}$ , etc., refer to other features for the same example.



$x_n$  is for single feature and  $x_{n1}$  is for multiple features?



Yes, that's correct! Here's a clear summary of the difference:

- $x_n$ : Refers to the input for the  $n$ -th example **when there is only a single feature**. For example, if you're using just house size to predict house prices,  $x_n$  would represent the size of the  $n$ -th house.
- $x_{n1}$ : Refers to the **first feature** of the  $n$ -th example **when there are multiple features**. For instance, if you're using both house size and the number of bedrooms,  $x_{n1}$  would represent the first feature (e.g., house size), and  $x_{n2}$  would represent the second feature (e.g., number of bedrooms) for the  $n$ -th example.

## In summary:

- **Single feature case:**  $x_n$  is used to refer to each example's input.
- **Multiple feature case:**  $x_{n1}$ ,  $x_{n2}$ , etc., are used to represent the different features of the  $n$ -th example.



stochastic vs mini-batch gradient descent



## Gradient Descent Overview

Gradient Descent is an optimization technique used to minimize the cost function by iteratively updating the model parameters. There are three main variations of gradient descent: **Stochastic**

**Gradient Descent (SGD), Mini-batch Gradient Descent, and Batch Gradient Descent.** Each one differs based on how much data is used to compute the gradient in each step.

Let's break down **Stochastic Gradient Descent** and **Mini-batch Gradient Descent**:

---

## 1. Stochastic Gradient Descent (SGD)

- **Definition:** In **Stochastic Gradient Descent**, the parameters are updated after computing the gradient from **only one training example** at a time. This means the model parameters are updated more frequently, after each individual example, instead of waiting for all examples to be processed.
- **How it Works:**
  - For each training example  $(x_n, y_n)$ , compute the gradient of the cost function  $J(\theta)$  and update the parameters.
  - Update rule:

$$\theta \leftarrow \theta - \alpha \cdot \nabla J(\theta; x_n, y_n)$$

where  $(x_n, y_n)$  is a single training example, and  $\alpha$  is the learning rate.

- **Advantages:**
    - Faster iterations because only one example is used at a time.
    - Can escape local minima due to its noisy updates.
    - Efficient for very large datasets.
  - **Disadvantages:**
    - The noisy updates can make the convergence path less smooth, leading to more fluctuations.
    - Typically requires more iterations to converge compared to batch gradient descent.
- 

## 2. Mini-batch Gradient Descent

- **Definition:** **Mini-batch Gradient Descent** is a compromise between **Stochastic Gradient Descent** and **Batch Gradient Descent**. It updates the parameters after computing the gradient from a **small random subset of the training data** (a mini-batch), rather than just one example (SGD) or the entire dataset (batch gradient descent).
- **How it Works:**
  - Divide the dataset into small **mini-batches** of size  $b$  (typically between 32 and 512 examples).
  - For each mini-batch, compute the gradient of the cost function and update the parameters.
  - Update rule:

$$\theta \leftarrow \theta - \alpha \cdot \nabla J(\theta; X_b, Y_b)$$

where  $X_b$  and  $Y_b$  are the inputs and outputs for the mini-batch.

- **Advantages:**

- Combines the benefits of both SGD (faster updates) and Batch Gradient Descent (smoother convergence).
- More computationally efficient than SGD because vectorized operations can be applied to a batch of data.
- Reduces the noise of parameter updates, leading to smoother convergence compared to SGD.

- **Disadvantages:**

- Still noisier than Batch Gradient Descent but not as noisy as Stochastic Gradient Descent.
- Choosing the right mini-batch size can require some experimentation.

## Key Differences

Aspect	Stochastic Gradient Descent (SGD)	Mini-batch Gradient Descent
Data used for update	One example at a time	A small batch of examples
Update frequency	Very frequent (after each example)	More frequent than batch, less than SGD
Noise in updates	High (because of single examples)	Lower (due to mini-batches)
Memory efficiency	Most memory efficient	More memory needed than SGD but less than full batch
Convergence path	Noisy and fluctuating	Smoother than SGD, noisier than batch
Speed per iteration	Fastest (as it's only one example)	Slower than SGD, faster than full batch
Best suited for	Large datasets, online learning	Medium to large datasets, neural networks

## Summary:

- **Stochastic Gradient Descent (SGD)** updates parameters after every individual data point, leading to fast but noisy updates.
- **Mini-batch Gradient Descent** updates parameters after processing a small batch of data, offering a balance between the efficiency of SGD and the stability of full-batch gradient descent.

Choosing between these methods depends on the size of your dataset and the computational resources available. For most practical machine learning tasks, **Mini-batch Gradient Descent** is often preferred because it strikes a good balance between convergence speed and computational efficiency.



after updating the parameters  $\theta_0$  and  $\theta_1$  until when should I keep doing these calculations?



You should keep updating the parameters  $\theta_0$  and  $\theta_1$  in your gradient descent algorithm until **one of the following conditions** is met:

## 1. Convergence of the Cost Function ( $J(\theta)$ )

- The cost function  $J(\theta)$  reaches a point where the changes in its value are very small (i.e., it's no longer decreasing significantly). This indicates that the algorithm has reached a **minimum** or an optimal solution.
- **Stopping criterion:**

$$|J(\theta_{\text{new}}) - J(\theta_{\text{old}})| < \epsilon$$

where  $\epsilon$  is a small threshold value (e.g.,  $10^{-6}$  or  $10^{-7}$ ) that defines how small the change in the cost function should be for you to consider stopping.

## 2. Gradient Near Zero (Small Gradient)

- When the **gradient** of the cost function becomes very small, it indicates that further updates to the parameters will not change the cost function significantly. This suggests that you've reached the minimum (or a local minimum in the case of non-convex functions).
- **Stopping criterion:**

$$\|\nabla J(\theta)\| < \epsilon$$

This means that the magnitude (norm) of the gradient vector is smaller than a pre-defined small number  $\epsilon$ .

## 3. Maximum Number of Iterations

- To prevent the algorithm from running indefinitely, you can set a **maximum number of iterations**. This ensures that the algorithm stops even if it hasn't fully converged. If you set a large enough maximum, it will usually stop before reaching the maximum due to convergence.
- **Stopping criterion:**

$$\text{iteration count} \geq \text{max\_iterations}$$

This is often used in practice to ensure that gradient descent doesn't run indefinitely, especially if it's slow to converge.

## 4. Tolerance in Parameter Updates

- If the **change in the parameters** ( $\theta_0$  and  $\theta_1$ ) after each iteration becomes very small, it can be an indication that the algorithm is converging.
- **Stopping criterion:**

$$|\theta_{\text{new}} - \theta_{\text{old}}| < \epsilon$$

This ensures that you stop when parameter updates no longer make a significant difference.

---

## Common Practice

In practice, you often combine **multiple criteria** to stop the algorithm, such as using both the convergence of the cost function and a maximum number of iterations.

For example:

1. Stop when the cost function changes less than  $\epsilon$ .
2. Or when the gradient becomes too small.
3. Or if the algorithm reaches the maximum number of iterations.

## Conclusion

Keep updating the parameters  $\theta_0$  and  $\theta_1$  until one of these stopping criteria is satisfied: the cost function stops decreasing, the gradient is small, or you've hit the maximum number of iterations.



give me an example to practice



Let's practice applying gradient descent to a simple linear regression problem. Here's the example:

### Problem:

You have a dataset that consists of house sizes (in square feet) and the corresponding prices (in \$1000s). You want to fit a linear model to predict house prices based on size using gradient descent.

The dataset is:

Size (sq. ft.) $x$	Price (\$1000s) $y$
1000	300
1500	450
2000	500

### Linear Model:

We are trying to fit a line using the equation:

$$\hat{y} = \theta_0 + \theta_1 x$$

where:

- $\hat{y}$  is the predicted house price.
- $\theta_0$  is the intercept.
- $\theta_1$  is the slope (how much the price changes with each square foot).

### Steps for Gradient Descent:

1. **Initialize**  $\theta_0$  and  $\theta_1$  to 0 (or some small random values).
2. Use a **learning rate**  $\alpha$  of 0.00001 (since house prices are in thousands, we use a smaller learning rate).
3. **Apply gradient descent** to minimize the cost function:

$$J(\theta_0, \theta_1) = \frac{1}{2N} \sum_{n=1}^N (\hat{y}_n - y_n)^2$$

4. Update  $\theta_0$  and  $\theta_1$  using:

$$\theta_0 \leftarrow \theta_0 - \alpha \cdot \frac{1}{N} \sum (\hat{y}_n - y_n)$$

$$\theta_1 \leftarrow \theta_1 - \alpha \cdot \frac{1}{N} \sum (\hat{y}_n - y_n) x_n$$

## Practice Task:

1. **Initialize**  $\theta_0 = 0$  and  $\theta_1 = 0$ .
2. **Calculate** the cost function for the first iteration.
3. **Compute** the gradients for  $\theta_0$  and  $\theta_1$ .
4. **Update**  $\theta_0$  and  $\theta_1$ .
5. **Repeat** for multiple iterations (3-5 should be enough to see some change).

## Example Walkthrough for the First Iteration:

1. **Initial values:**  $\theta_0 = 0$ ,  $\theta_1 = 0$ ,  $\alpha = 0.00001$
2. **Predictions:**  $\hat{y} = \theta_0 + \theta_1 x = 0$  (since both parameters are 0 initially)
  - For  $x = 1000$ ,  $\hat{y} = 0$
  - For  $x = 1500$ ,  $\hat{y} = 0$
  - For  $x = 2000$ ,  $\hat{y} = 0$
3. **Cost Function:**

$$J(\theta_0, \theta_1) = \frac{1}{2N} \sum (\hat{y} - y)^2$$

Substituting the values:

$$J(0, 0) = \frac{1}{6} ((0 - 300)^2 + (0 - 450)^2 + (0 - 500)^2) = 116,250$$

### 4. Gradients:

- Gradient w.r.t.  $\theta_0$ :

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0} = \frac{1}{N} \sum (\hat{y} - y) = \frac{1}{3} ((0 - 300) + (0 - 450) + (0 - 500)) = -416.67$$

- Gradient w.r.t.  $\theta_1$ :

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1} = \frac{1}{N} \sum (\hat{y} - y) x = \frac{1}{3} ((0 - 300) \cdot 1000 + (0 - 450) \cdot 1500 + (0 - 500) \cdot 2000) = -416,667$$

### 5. Update Parameters:

- $\theta_0 \leftarrow 0 - 0.00001 \cdot (-416.67) = 0.004167$
- $\theta_1 \leftarrow 0 - 0.00001 \cdot (-808,333.33) = 8.08333$

Now, with updated  $\theta_0 = 0.004167$  and  $\theta_1 = 8.08333$ , repeat the process for more iterations.

---

### Your Task:

1. Perform at least **3 iterations** of this process by:
  - Calculating predictions.
  - Computing the cost function.
  - Calculating the gradients.
  - Updating  $\theta_0$  and  $\theta_1$ .
2. After 3-5 iterations, see how the cost function decreases and the parameters  $\theta_0$  and  $\theta_1$  evolve.

Let me know how it goes or if you'd like to verify your steps!



after the 1st iteration do i use the values of theta0 and theta1 in the cost function again?



Yes, exactly! After the first iteration, you will use the **updated values** of  $\theta_0$  and  $\theta_1$  to calculate the cost function and the gradients for the next iteration.

### Steps after the 1st iteration:

1. **Update Parameters:** After the first iteration, you have new values for  $\theta_0$  and  $\theta_1$ .

From the first iteration in our example:

- $\theta_0 = 0.004167$
- $\theta_1 = 8.08333$

2. **Use these updated values** in the cost function:

$$\hat{y}_n = \theta_0 + \theta_1 x_n$$

Now, for the new predictions  $\hat{y}$ , you'll use:

$$\hat{y} = 0.004167 + 8.08333 \cdot x$$

For example:

- For  $x = 1000$ :  $\hat{y} = 0.004167 + 8.08333 \cdot 1000 = 8083.33$
- For  $x = 1500$ :  $\hat{y} = 0.004167 + 8.08333 \cdot 1500 = 12125$
- For  $x = 2000$ :  $\hat{y} = 0.004167 + 8.08333 \cdot 2000 = 16166.66$

3. **Calculate the cost function** using these updated predictions:



$$J(\theta_0, \theta_1) = \frac{1}{6} \sum (\hat{y}_n - y_n)^2$$

Now substitute the new predictions and actual values:

- $(\hat{y}_1 - y_1)^2 = (8083.33 - 300)^2$
- $(\hat{y}_2 - y_2)^2 = (12125 - 450)^2$
- $(\hat{y}_3 - y_3)^2 = (16166.66 - 500)^2$

4. **Compute the new gradients** using the updated  $\theta_0$  and  $\theta_1$ .

5. **Update  $\theta_0$  and  $\theta_1$  again** based on the new gradients.

## In Summary:

- After each iteration, you **use the updated parameters**  $\theta_0$  and  $\theta_1$  to calculate new predictions, the cost function, and the gradients.
- This process is repeated until the cost function stops decreasing significantly or other stopping criteria are met.

Let me know if you want to go through the calculations step-by-step for further iterations!



give me another example with small values



Sure! Let's work through an example with smaller values for easier calculations.

## Problem:

You have the following simple dataset of two points:

$x$ (input)	$y$ (output)
1	2
2	3

You want to fit a linear model using gradient descent:

$$\hat{y} = \theta_0 + \theta_1 x$$

where:

- $\theta_0$  is the intercept,
- $\theta_1$  is the slope (how much the output changes with the input).

## Steps for Gradient Descent:

1. **Initialize**  $\theta_0 = 0$  and  $\theta_1 = 0$ .

2. Use a **learning rate**  $\alpha = 0.1$ .
3. Apply **gradient descent** to minimize the cost function:

$$J(\theta_0, \theta_1) = \frac{1}{2N} \sum_{n=1}^N (\hat{y}_n - y_n)^2$$

where  $N$  is the number of training examples.

4. Update  $\theta_0$  and  $\theta_1$  using the gradient descent update rules:

$$\theta_0 \leftarrow \theta_0 - \alpha \cdot \frac{1}{N} \sum (\hat{y}_n - y_n)$$

$$\theta_1 \leftarrow \theta_1 - \alpha \cdot \frac{1}{N} \sum (\hat{y}_n - y_n) x_n$$

## 1st Iteration:

1. **Initial values:**  $\theta_0 = 0, \theta_1 = 0, \alpha = 0.1$

2. **Predictions:**

$$\hat{y}_n = \theta_0 + \theta_1 x_n = 0 + 0 \cdot x_n = 0$$

For both  $x_1 = 1$  and  $x_2 = 2$ , the prediction  $\hat{y} = 0$ .

3. **Cost Function:**

$$J(\theta_0, \theta_1) = \frac{1}{2 \cdot 2} ((0 - 2)^2 + (0 - 3)^2)$$

$$J(0, 0) = \frac{1}{4} (4 + 9) = \frac{13}{4} = 3.25$$

4. **Gradients:**

- Gradient w.r.t.  $\theta_0$ :

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0} = \frac{1}{2} ((0 - 2) + (0 - 3)) = \frac{1}{2} \cdot (-5) = -2.5$$

- Gradient w.r.t.  $\theta_1$ :

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1} = \frac{1}{2} ((0 - 2) \cdot 1 + (0 - 3) \cdot 2) = \frac{1}{2} \cdot (-2 - 6) = -4$$

5. **Update Parameters:**

- $\theta_0 \leftarrow 0 - 0.1 \cdot (-2.5) = 0.25$
- $\theta_1 \leftarrow 0 - 0.1 \cdot (-4) = 0.4$

## 2nd Iteration:

1. **Updated values:**  $\theta_0 = 0.25, \theta_1 = 0.4$

## 2. Predictions:

$$\hat{y}_1 = 0.25 + 0.4 \cdot 1 = 0.65$$

$$\hat{y}_2 = 0.25 + 0.4 \cdot 2 = 1.05$$

## 3. Cost Function:

$$J(\theta_0, \theta_1) = \frac{1}{4} ((0.65 - 2)^2 + (1.05 - 3)^2)$$

$$J(0.25, 0.4) = \frac{1}{4} (1.8225 + 3.8025) = \frac{1}{4} \cdot 5.625 = 1.40625$$

## 4. Gradients:

- Gradient w.r.t.  $\theta_0$ :

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_0} = \frac{1}{2} ((0.65 - 2) + (1.05 - 3)) = \frac{1}{2} \cdot (-3.3) = -1.65$$

- Gradient w.r.t.  $\theta_1$ :

$$\frac{\partial J(\theta_0, \theta_1)}{\partial \theta_1} = \frac{1}{2} ((0.65 - 2) \cdot 1 + (1.05 - 3) \cdot 2) = \frac{1}{2} \cdot (-1.35 - 3.9) = -2.625$$

## 5. Update Parameters:

- $\theta_0 \leftarrow 0.25 - 0.1 \cdot (-1.65) = 0.415$
- $\theta_1 \leftarrow 0.4 - 0.1 \cdot (-2.625) = 0.6625$

## Your Task:

- Continue this process for 2-3 more iterations, updating  $\theta_0$  and  $\theta_1$  each time, and observe how the cost function decreases.
- You'll notice that after each iteration, the values of  $\theta_0$  and  $\theta_1$  get closer to the optimal solution.

Let me know how it goes or if you need further clarification!



what happens in stochastic gradient descent?



In stochastic gradient descent (SGD), the key difference from traditional (batch) gradient descent is that the parameter updates are performed using only a **single training example** at a time instead of the entire dataset. This leads to several important characteristics and implications:

## Key Features of Stochastic Gradient Descent:

### 1. Single Sample Updates:

- In SGD, for each training example  $(x_n, y_n)$ , the parameters  $\theta_0$  and  $\theta_1$  are updated immediately after computing the gradient based on that single example.
- The update rule for each example is:

$$\theta_0 \leftarrow \theta_0 - \alpha \cdot \frac{\partial J(\theta_0, \theta_1; x_n, y_n)}{\partial \theta_0}$$

$$\theta_1 \leftarrow \theta_1 - \alpha \cdot \frac{\partial J(\theta_0, \theta_1; x_n, y_n)}{\partial \theta_1}$$

## 2. Faster Convergence:

- Since the model parameters are updated more frequently (after each example), SGD can lead to faster convergence, especially for large datasets.
- This frequent updating can help the model to escape local minima more effectively.

## 3. Higher Variance:

- The updates in SGD are noisier than those in batch gradient descent because they are based on a single sample rather than the average of all samples.
- This noise can cause the cost function to fluctuate instead of decreasing steadily. However, this can also help in exploring the parameter space more thoroughly.

## 4. Potentially Better Generalization:

- The stochastic nature of the updates can act as a form of regularization, which may help prevent overfitting. It encourages the model to find a more generalized solution rather than fitting the training data too closely.

## 5. Learning Rate Scheduling:

- In practice, the learning rate  $\alpha$  may be adjusted over time (e.g., decreasing as training progresses) to improve convergence and stability.

## Example:

Imagine a simple linear regression problem with a small dataset. Instead of computing the gradients based on the entire dataset, you would:

1. Shuffle the training data.
2. For each training example  $(x_n, y_n)$ :
  - Compute the predicted value  $\hat{y}_n$ .
  - Calculate the gradients for  $\theta_0$  and  $\theta_1$ .
  - Update  $\theta_0$  and  $\theta_1$  immediately after each example.

## Summary:

Stochastic gradient descent offers a more dynamic and faster approach to optimization compared to batch gradient descent. While it introduces more variance in the updates, this can be beneficial for certain types of problems, especially in large datasets or complex models.

Let me know if you'd like to dive deeper into any specific aspects of stochastic gradient descent!



give a simple example



Let's go through a simple example of Stochastic Gradient Descent (SGD) using a linear regression model with a small dataset.

## Problem:

You have the following small dataset:

$x$ (input)	$y$ (output)
1	2
2	3
3	5

You want to fit a linear model:

$$\hat{y} = \theta_0 + \theta_1 x$$

where:

- $\theta_0$  is the intercept,
- $\theta_1$  is the slope.

## Setup:

- Initialize  $\theta_0 = 0$  and  $\theta_1 = 0$ .
- Use a learning rate  $\alpha = 0.1$ .

## Stochastic Gradient Descent Steps:

### Iteration 1:

#### 1. Using first training example ( $x_1 = 1, y_1 = 2$ ):

- Prediction:

$$\hat{y}_1 = \theta_0 + \theta_1 x_1 = 0 + 0 \cdot 1 = 0$$

- Calculate the cost:

$$\text{Error} = \hat{y}_1 - y_1 = 0 - 2 = -2$$

- **Gradients:**

$$\frac{\partial J}{\partial \theta_0} = \text{Error} = -2$$

$$\frac{\partial J}{\partial \theta_1} = \text{Error} \cdot x_1 = -2 \cdot 1 = -2$$

- **Update Parameters:**

$$\theta_0 \leftarrow 0 - 0.1 \cdot (-2) = 0.2$$

$$\theta_1 \leftarrow 0 - 0.1 \cdot (-2) = 0.2$$

## 2. Using second training example ( $x_2 = 2, y_2 = 3$ ):

- Prediction:

$$\hat{y}_2 = 0.2 + 0.2 \cdot 2 = 0.6$$

- Calculate the cost:

$$\text{Error} = \hat{y}_2 - y_2 = 0.6 - 3 = -2.4$$

- **Gradients:**

$$\frac{\partial J}{\partial \theta_0} = \text{Error} = -2.4$$

$$\frac{\partial J}{\partial \theta_1} = \text{Error} \cdot x_2 = -2.4 \cdot 2 = -4.8$$

- **Update Parameters:**

$$\theta_0 \leftarrow 0.2 - 0.1 \cdot (-2.4) = 0.44$$

$$\theta_1 \leftarrow 0.2 - 0.1 \cdot (-4.8) = 0.68$$

## 3. Using third training example ( $x_3 = 3, y_3 = 5$ ):

- Prediction:

$$\hat{y}_3 = 0.44 + 0.68 \cdot 3 = 2.48$$

- Calculate the cost:

$$\text{Error} = \hat{y}_3 - y_3 = 2.48 - 5 = -2.52$$

- **Gradients:**

$$\frac{\partial J}{\partial \theta_0} = \text{Error} = -2.52$$

$$\frac{\partial J}{\partial \theta_1} = \text{Error} \cdot x_3 = -2.52 \cdot 3 = -7.56$$

- **Update Parameters:**

$$\theta_0 \leftarrow 0.44 - 0.1 \cdot (-2.52) = 0.694$$

$$\theta_1 \leftarrow 0.68 - 0.1 \cdot (-7.56) = 1.435$$

## Summary of Iteration 1:

- After processing each example, you updated the parameters  $\theta_0$  and  $\theta_1$  based on the error from each individual example. The parameters now are:
  - $\theta_0 \approx 0.694$
  - $\theta_1 \approx 1.435$

## Repeat the Process:

You would then continue this process for multiple iterations, shuffling the training examples each time to ensure a different order of examples is processed, further refining the parameters until convergence is reached.

## Conclusion:

In Stochastic Gradient Descent, updates are made after each training example, which can lead to faster convergence and better exploration of the parameter space due to the noise introduced by the single-example updates. Let me know if you need further clarification or if you'd like to explore more examples!



so the main difference is that in batch we take the summation and take the squared error but here we take a single point and take the non squared error?



Not quite! Let's clarify the differences between **batch gradient descent** and **stochastic gradient descent (SGD)**:

## Batch Gradient Descent:

- **Process:**
  - In batch gradient descent, you compute the gradients of the cost function using the entire dataset.
  - The update of the parameters is done once per iteration after calculating the average of the gradients over all training examples.
- **Cost Function:**
  - The cost function is typically the **mean squared error (MSE)**, defined as:

$$J(\theta_0, \theta_1) = \frac{1}{N} \sum_{n=1}^N (\hat{y}_n - y_n)^2$$

- Here,  $N$  is the total number of training examples, and  $(\hat{y}_n - y_n)^2$  is the squared error for each example.

- **Updates:**

- The parameters  $\theta_0$  and  $\theta_1$  are updated using the average gradients:

$$\theta_0 \leftarrow \theta_0 - \alpha \cdot \frac{1}{N} \sum_{n=1}^N \frac{\partial J}{\partial \theta_0}$$

$$\theta_1 \leftarrow \theta_1 - \alpha \cdot \frac{1}{N} \sum_{n=1}^N \frac{\partial J}{\partial \theta_1}$$

## Stochastic Gradient Descent (SGD):

- **Process:**

- In SGD, you compute the gradients using **one training example** at a time.
- The parameters are updated immediately after processing each individual example.

- **Cost Function:**

- The same **mean squared error** is used, but you evaluate it for each individual training example when calculating the gradients:

$$\text{Error} = \hat{y}_n - y_n$$

- You compute the error and gradients for just that single point, not averaging over the entire dataset.

- **Updates:**

- The parameters are updated based on the individual gradients:

$$\theta_0 \leftarrow \theta_0 - \alpha \cdot \frac{\partial J}{\partial \theta_0} \quad (\text{for one sample})$$

$$\theta_1 \leftarrow \theta_1 - \alpha \cdot \frac{\partial J}{\partial \theta_1} \quad (\text{for one sample})$$

## Summary of Differences:

- **Batch Gradient Descent:**

- Computes gradients using the entire dataset.
- Updates parameters after processing all examples, leading to smoother convergence.

- **Stochastic Gradient Descent:**

- Computes gradients using one training example at a time.
- Updates parameters immediately after each example, leading to noisier but more frequent updates.

## Conclusion:

So the main difference is **not** just about taking the squared error vs. non-squared error but rather about whether you're using the entire dataset or individual examples to compute the gradients and update the parameters. Both methods typically use the squared error for the cost function, but they handle the updates differently.



Let me know if you have more questions or need further examples!

ChatGPT can make mistakes. Check important info.