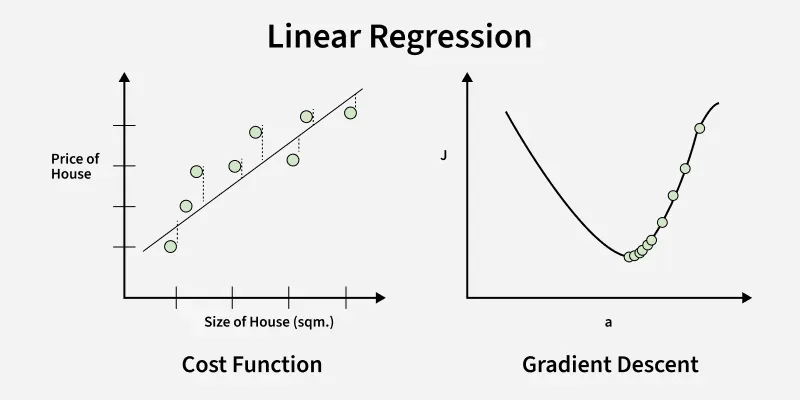
**Gradient Descent in Linear Regression:**

[**https://www.geeksforgeeks.org/gradient-descent-in-linear-regression/**](https://www.geeksforgeeks.org/gradient-descent-in-linear-regression/)

**Gradient descent** is a optimization algorithm used in **linear regression** to find the best fit line to the data. It works by gradually by adjusting the line’s slope and intercept to reduce the difference between actual and predicted values. This process helps the model make accurate predictions by minimizing errors step by step. In this article we will see more about Gradient Descent and its core concepts in detail.



Gradient Descent in Linear Regression

Above image shows two graphs, left one plots house prices against size to show errors measured by the **cost function** while right one shows how **gradient descent** moves downhill on the cost curve to minimize error by updating parameters step by step.

**Why Use Gradient Descent for Linear Regression?**

[Linear regression](https://www.geeksforgeeks.org/ml-linear-regression/) finds the **best-fit line** for a dataset by minimizing the **error** between the actual and predicted values. This error is measured using the [cost function](https://www.geeksforgeeks.org/what-is-the-cost-function-in-linear-regression/) usually Mean Squared Error (MSE). The goal is to find the model parameters i.e. the **slope m** and the **intercept b** that minimize this cost function.

For simple linear regression, we can use formulas like [Normal Equation](https://www.geeksforgeeks.org/ml-normal-equation-in-linear-regression/) to find parameters directly. However for **large datasets** or **high-dimensional data** these methods become computationally expensive due to:

* Large matrix computations.
* Memory limitations.

In models like [polynomial regression](https://www.geeksforgeeks.org/python-implementation-of-polynomial-regression/), the cost function becomes highly complex and non-linear, so analytical solutions are not available. That’s where **gradient descent** plays an important role even for:

* Large datasets.
* Complex, high-dimensional problems.

**scikit-learn (sklearn)** does support **Gradient Descent**, but not directly in the LinearRegression class. Instead:

**✅ scikit-learn's LinearRegression:**

* Uses the **Ordinary Least Squares (OLS)** analytical solution.
* It **does not use Gradient Descent**.

**✅ For Gradient Descent in sklearn:**

You can use:

* sklearn.linear\_model.SGDRegressor

This class uses **Stochastic Gradient Descent (SGD)** for regression.

**✅ Code Example: Linear Regression Using Gradient Descent (SGDRegressor)**

python

CopyEdit

from sklearn.linear\_model import SGDRegressor

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error, r2\_score

import numpy as np

import matplotlib.pyplot as plt

# 1. Sample dataset

X = np.array([[1], [2], [3], [4], [5]])

y = np.array([2, 4, 5, 4, 5])

# 2. Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 3. Define SGD Regressor

model = SGDRegressor(max\_iter=1000, learning\_rate='invscaling', eta0=0.01)

# 4. Fit model

model.fit(X\_train, y\_train)

# 5. Predict

y\_pred = model.predict(X\_test)

# 6. Evaluation

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f"Model Coefficients (theta1): {model.coef\_[0]:.4f}")

print(f"Model Intercept (theta0): {model.intercept\_[0]:.4f}")

print(f"Mean Squared Error: {mse:.4f}")

print(f"R² Score: {r2:.4f}")

# 7. Plot

plt.scatter(X, y, color='blue', label='Original Data')

plt.plot(X, model.predict(X), color='red', label='Fitted Line (SGD)')

plt.xlabel("X")

plt.ylabel("y")

plt.title("Linear Regression with Gradient Descent (SGD)")

plt.legend()

plt.grid(True)

plt.show()

**🔍 Notes:**

* SGDRegressor uses **stochastic gradient descent**, which is suitable for large datasets.
* eta0 is the **initial learning rate**.
* learning\_rate='invscaling' changes the learning rate over time.
* You can use loss='squared\_error' (default) for standard linear regression.

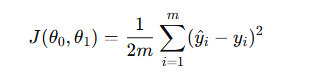
**Internal working explanation**

In **gradient descent for linear regression**, the formulas for updating the model parameters θ0\theta\_0θ0​ (intercept) and θ1\theta\_1θ1​ (slope) are derived from **minimizing the loss function** — specifically, the **Mean Squared Error (MSE)**.

Let’s break this down step by step:

**🔶 1. Loss Function Used**

The **loss function** (also called cost function) used in linear regression with gradient descent is:



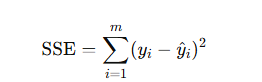
Where:

* y^​i​=θ0 ​+ θ1​xi​
* yi​ is the actual value.
* m is the number of training samples.
* This is the **Mean Squared Error (MSE)** cost function (scaled by (1/2)​ for convenience in derivatives).

**🔶 2. Gradient Descent — Why Not SSE or SSR Directly?**

**✅ SSE:**

SSE is the **Sum of Squared Errors**:

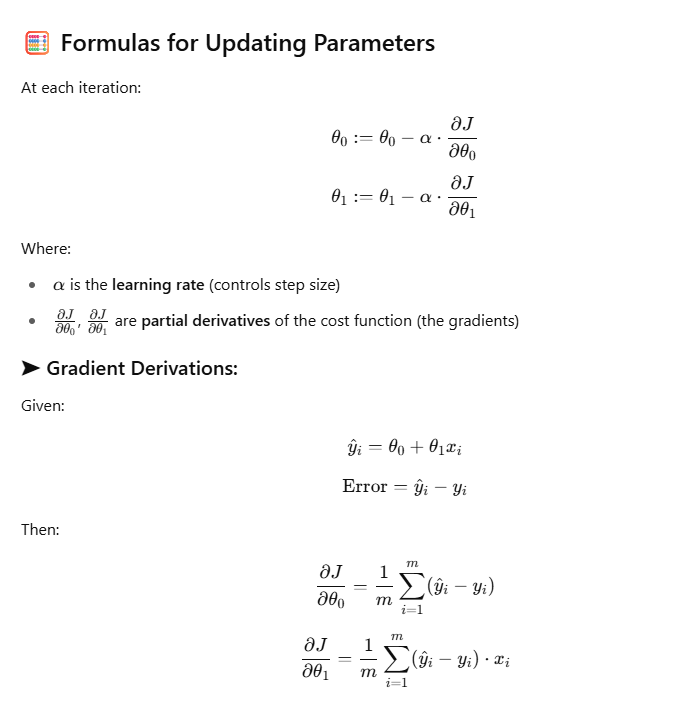


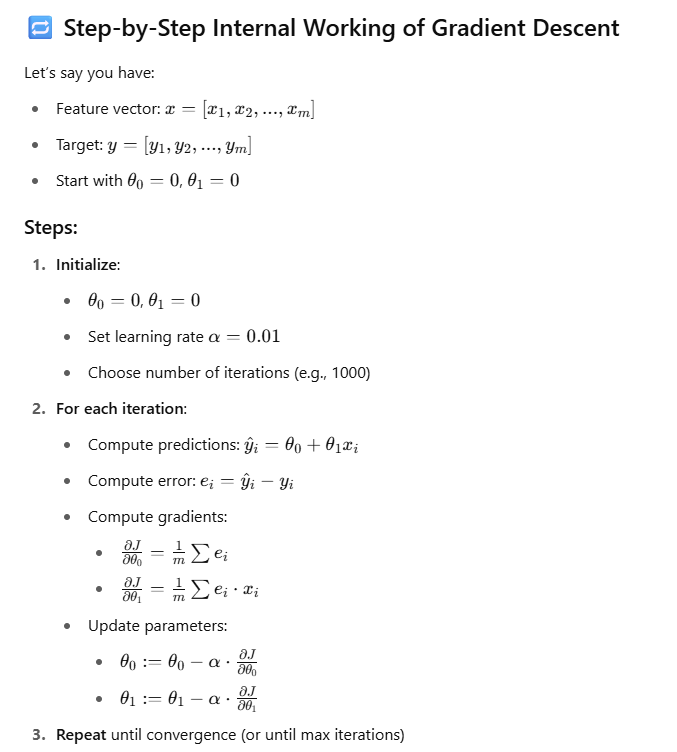
It is the **total error** we're trying to minimize.

✅ **The formula used in gradient descent is based on the derivative of SSE** (or MSE, which is SSE divided by mmm). So, SSE *is* being used — just indirectly, through **calculus**.

**❌ SSR:**

SSR (Sum of Squares due to Regression) measures the explained variance by the model, not the error. It's used more in **model evaluation (R² score)**, not in optimization.





**Python code: -**

import matplotlib.pyplot as plt

import numpy as np

import matplotlib.animation as animation

# Sample dataset

x = np.array([1, 2, 3])

y = np.array([1, 2, 3])

m = len(x)

# Learning rate

alpha = 0.1

# Initialize theta

theta\_0 = 0

theta\_1 = 0

# Store theta history

theta\_history = [(theta\_0, theta\_1)]

# Perform 3 iterations of gradient descent

for \_ in range(3):

y\_pred = theta\_0 + theta\_1 \* x

error = y\_pred - y

d\_theta\_0 = (1 / m) \* np.sum(error)

d\_theta\_1 = (1 / m) \* np.sum(error \* x)

theta\_0 = theta\_0 - alpha \* d\_theta\_0

theta\_1 = theta\_1 - alpha \* d\_theta\_1

theta\_history.append((theta\_0, theta\_1))

# Prepare plot

fig, ax = plt.subplots()

line, = ax.plot([], [], 'r-', lw=2, label='Fitted Line')

scatter = ax.scatter(x, y, color='blue', label='Data Points')

title = ax.set\_title('')

ax.set\_xlim(0, 4)

ax.set\_ylim(0, 4)

ax.set\_xlabel("x")

ax.set\_ylabel("y")

ax.legend()

# Animation function

def animate(i):

theta\_0, theta\_1 = theta\_history[i]

y\_pred\_line = theta\_0 + theta\_1 \* x

line.set\_data(x, y\_pred\_line)

title.set\_text(f"Iteration {i}: y = {theta\_0:.2f} + {theta\_1:.2f}x")

return line, title

ani = animation.FuncAnimation(fig, animate, frames=len(theta\_history), interval=1000, blit=False, repeat=True)

plt.close(fig)

ani

**Final Notes**

* The result is a line y^​=θ0​+θ1​x that **best fits the data** (minimizes the MSE).
* Learning rate (α) is critical: too large → overshooting; too small → slow convergence.
* You can use **Batch**, **Stochastic**, or **Mini-batch Gradient Descent** depending on data size.