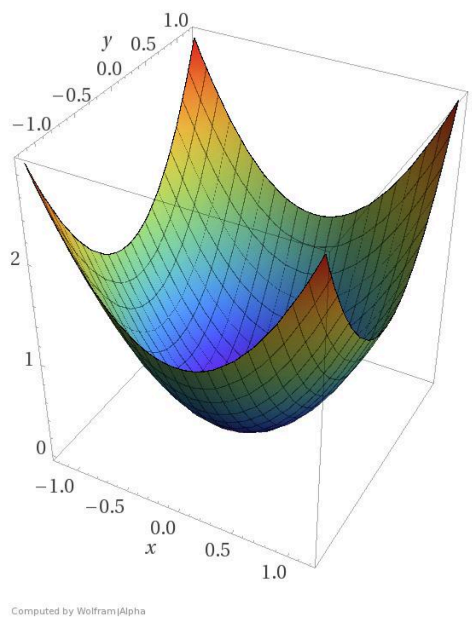
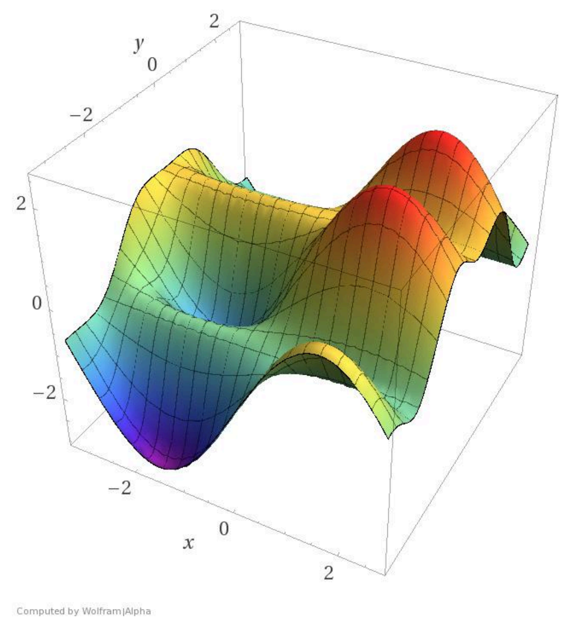
Gradient Descent

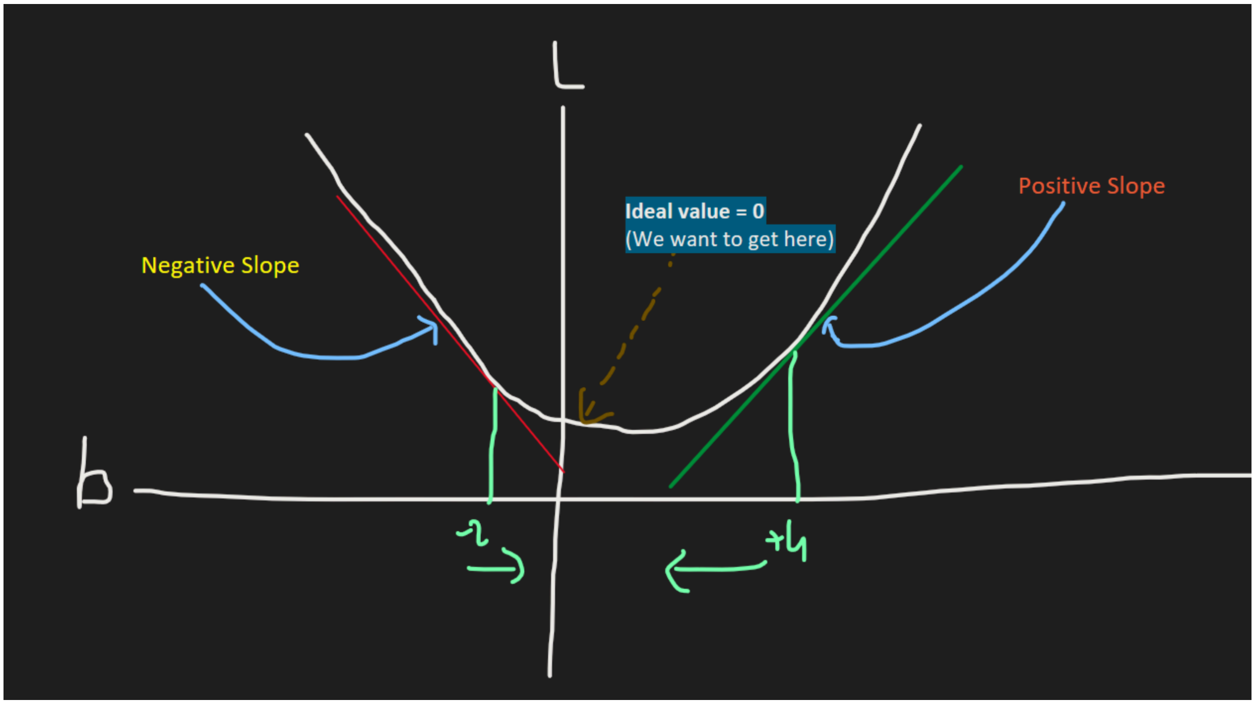




- GD is technique to calculate **Minima.**
- Backbone of deep learning
- Gradient = Derivative

Steps

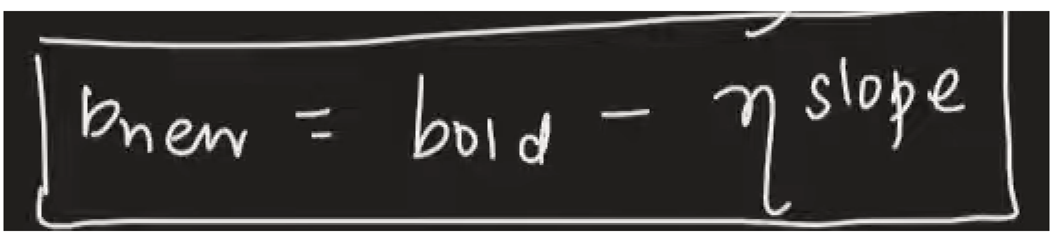
1. Select a random value of b (Y-Intercept)
2. Calculate the slope at that point with derivative
3. If slope is → +ve, increment the value of b
4. If slope is → -ve, decrease the value of b



In short,

$$b_{new} = b_{old} - slope$$

- To prevent drastic changes, you multiply the slope with learning rate.
  - It's usually equal to 0.01



n= learning rate

When to Stop?

- **Approach 1:** When $b_{new} - b_{old}$ is **close to zero (0.0001)**
- **Approach 2:** You decide a certain no. of iterations eg. 100 or 1000

- These are called **epochs**

Mathematical Formulation

- Let m be a constant to make the calculation easier
- Run a loop `for i in epochs` (epoch is no. of iterations- 100,1000,etc.)

for i in epochs:

$$b_{new} = b_{old} - \eta * slope$$

n = 0.01

$n * slope$ → Step Size

- It will pick any random value of b
- Calculate slope at that point

- This is the eq for loss function:

$$L = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

- To find out slope at point b, we have to differentiate this:

$$\frac{dL}{db} = \frac{d}{db}\left(\sum_{i=1}^{n}(y_i - \hat{y}_i)^2\right)$$

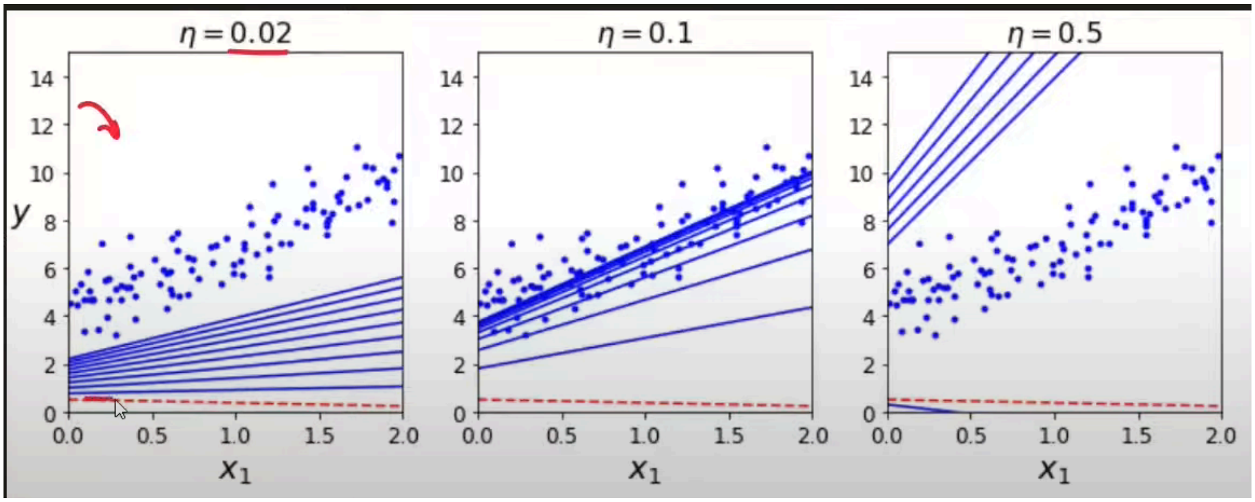
$$\frac{d}{db}\sum_{i=1}^{n}(y_i - mx_i - b)^2$$

Equation to calculate slope: 👇

$$\text{slope} = -2\sum_{i=1}^{n}(y_i - mx_i - b)$$

- Just insert the value of b, x_i, y_i

Impact of Learning Rate



Equation for GD

- When you calculate m or b keeping one thing constant → It's derivative
- But when you calculate both m & b → It's Gradient

The equation for Loss function is:

$$\sum (y_i - m x_i - b)^2$$

- We have to differentiate with wrt m & b

wrt b:

$$\frac{\partial L}{\partial b} = -2\sum \left(\underset{0}{y_i} - \underset{0}{mx_i} - \underset{-1}{b}\right)$$

$$= \boxed{-2\sum \left(y_i - mx_i - b\right)}$$

wrt m:

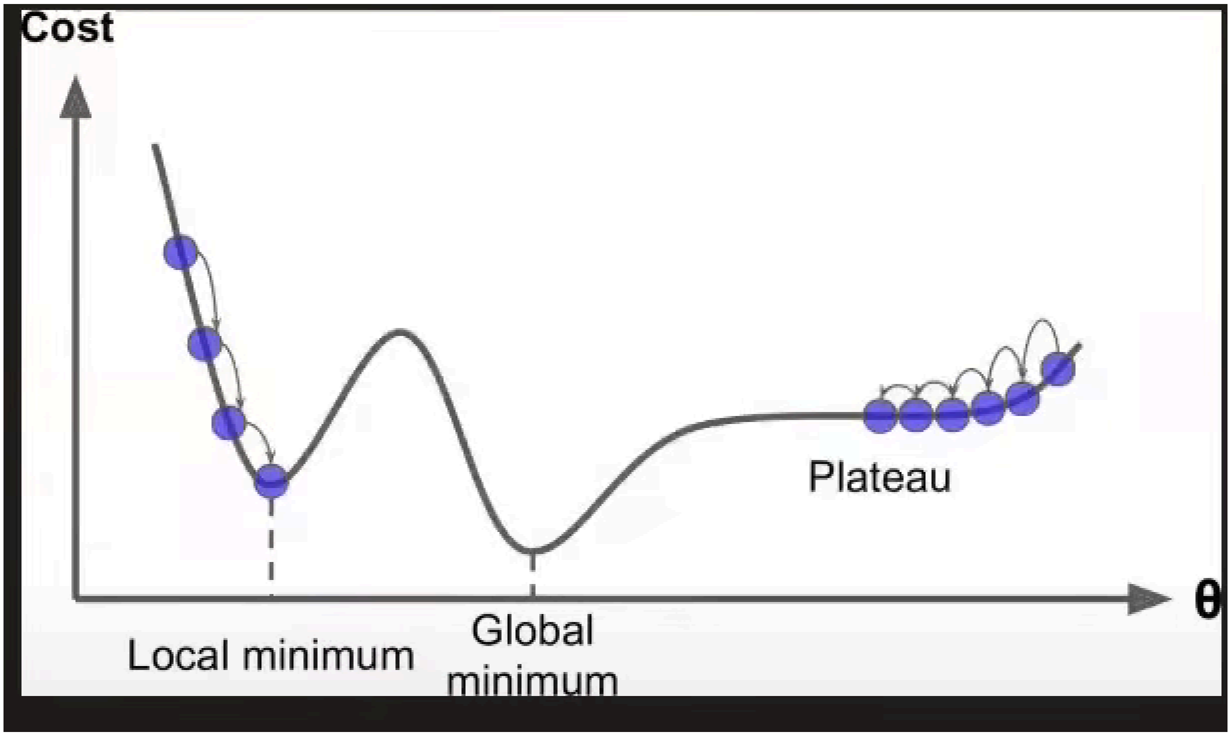
$$\frac{\partial L}{\partial m} = 2\sum \left(\underset{0}{y_i} - \underset{-x_i}{mx_i} - \underset{0}{b}\right)$$

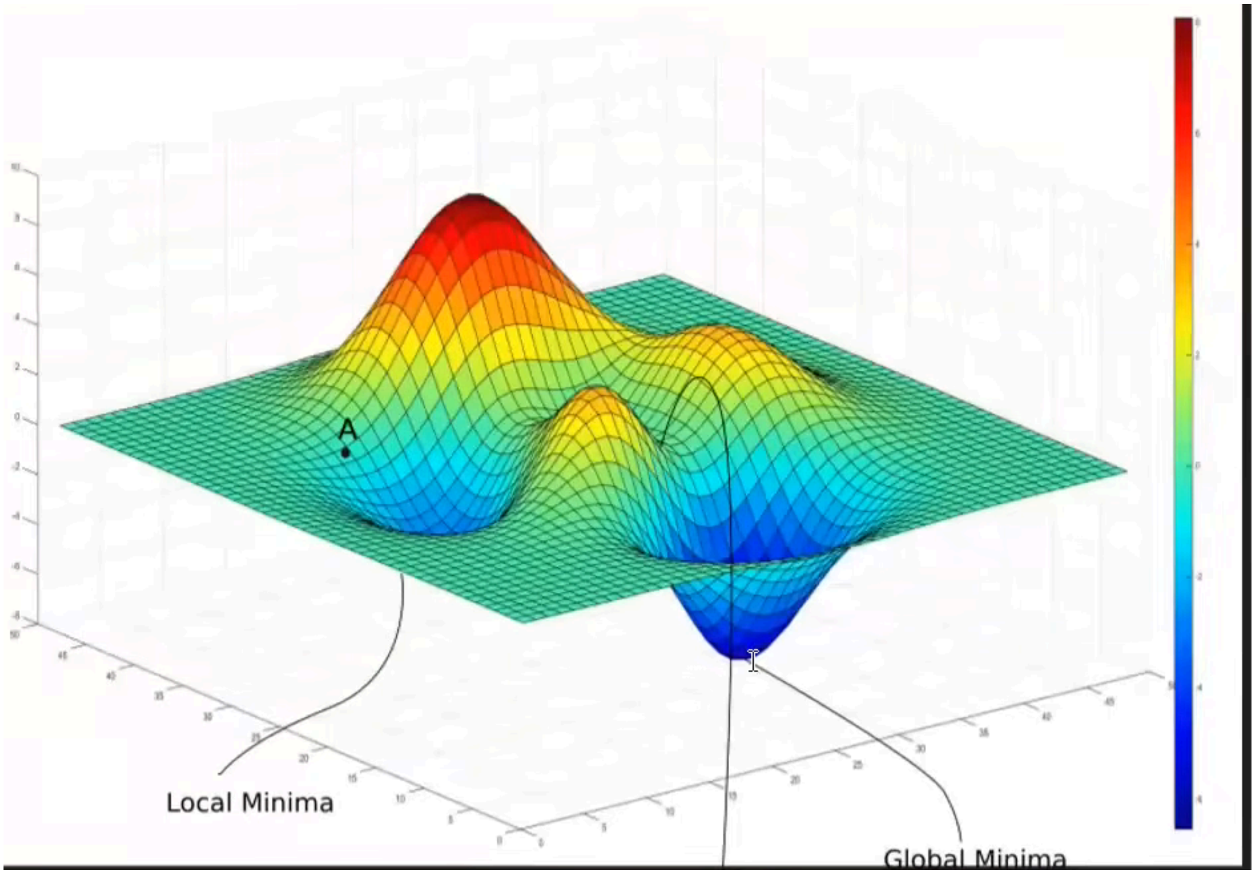
$$= -2\sum \left(y_i - mx_i - b\right) x_i$$

Effect of Loss Function

- We need a convex function so that it doesn’t cross our function
- If it’s a non-convex function, there will be more than 1 minima

- **In convex function → There's only 1 minima → Global minima**
- R2 is a convex function





Effect of Data

- If data is not on same scale, it could take a lot of time to reach to the global minima

Gradient Descent/SGDRegressor from Scratch

Steps:

1. Initiate some value of `b`, `m`
    - These represent the model's parameters (slope and intercept) which we want to optimize.

- User will provide `learning rate` & `epoch`

2. **Prediction:** For each epoch (iteration), we calculate the predicted values:

$$y_{pred} = m \times X + b$$

3. **Compute Loss**: Calculate the error between predictions and actual values.
    - The difference between the actual values (`y`) and predictions (`y_pred`) is computed as `error`.
4. **Calculate Gradients**: Find the slope of the loss function with respect to each parameter.
    - We calculate how much we need to change `m` and `b` to reduce the error

- For `m`:

$$dm = -\frac{2}{n} \sum X \times (y - y_{pred})$$

- For `b`:

$$db = -\frac{2}{n} \sum (y - y_{pred})$$

5. **Update Parameters**: Adjust parameters in the direction that reduces the loss.

$$\boxed{b_{new} = b_{old} - \eta \, slope}$$

5. **Repeat** until convergence (minimum loss is found).

```
class GDRegressor:

    def __init__(self,learning_rate,epochs):
        self.m = 100
        self.b = -120
        self.lr = learning_rate
        self.epochs = epochs

    def fit(self,X,y):
        # calcualte the b using GD
        for i in range(self.epochs):
            loss_slope_b = -2 * np.sum(y - self.m*X.ravel() - self.b)
            loss_slope_m = -2 * np.sum((y - self.m*X.ravel() - self.b)*X.ravel())

            self.b = self.b - (self.lr * loss_slope_b)
            self.m = self.m - (self.lr * loss_slope_m)
        print(self.m,self.b)

    def predict(self,X):
        return self.m * X + self.b
```