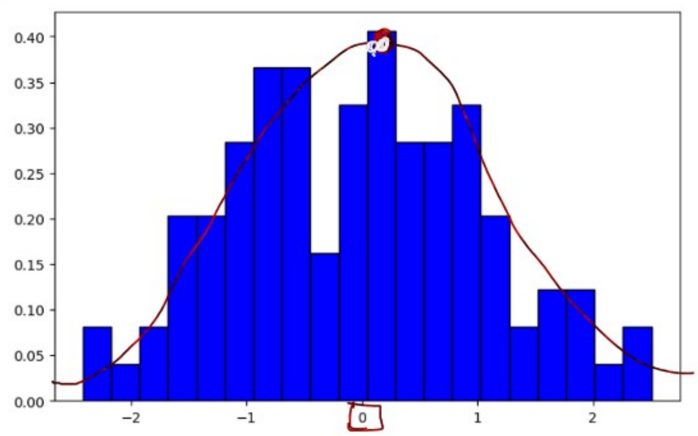
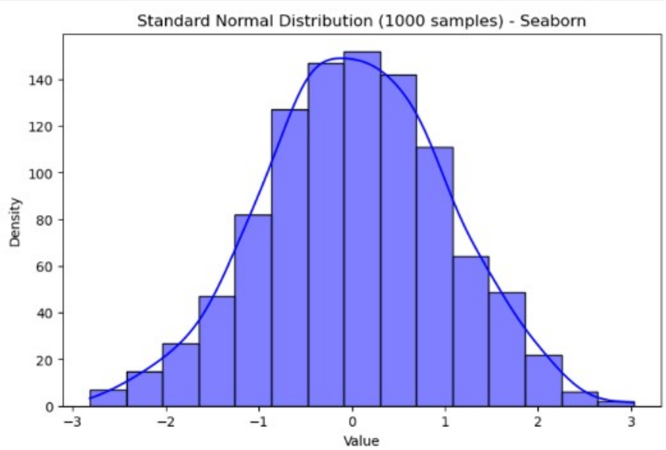
GDA Code Explanation

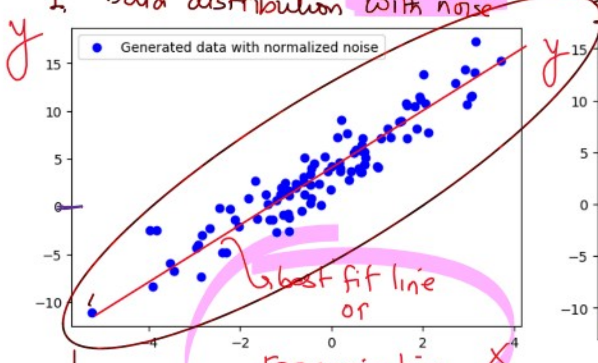
27 September 2025 11:55

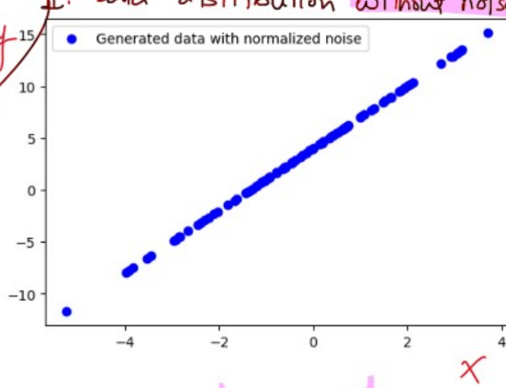




I. Data distribution with noise

II. Data distribution without noise





best fit line or regression line

x

need GDA

do you need GDA

No.

since the distribution is not around y=x hence definitely some bias.

number of data points to be generated

```python
def generate_data( n_samples = 100, noise = 0.1, seed =42):
    """
    Generate random samples (synthetic) linear data: y=4 + 3*X + noise
    """

    np.random.seed(seed) #to ensure the reproducibility of the same random data -->to fix the random numbers generated
    X = 2 * np.random.randn(n_samples, 1) #generates random numbers from `standard normal distribution`
    y = 4 + 3*X + noise*np.random.randn(n_samples, 1) # Creates a random distribution around a line having some random noise added to it as well

    return X, y
```

ensures reproducibility

constant noise → 0.1 (default)

0.1 * (100 random values coming from std. normal distribution)

2 * random std. normal values

y → 4 + 3*X + noise

4 + ( ) + ( )

100 values

$y = 4 + 3x$

4: intercept (bias)

3: coefficient / slope (weight)

random values — same

```python
    X = 2 * np.random.randn(n_samples, 1) #generates random numbers from `standard normal distribution`
    y = 4 + 3*X + noise*np.random.randn(n_samples, 1) # Creates a random distribution around a line having sc

    return X, y
```

```python
generate_data( n_samples = 10, noise = 0, seed =42)
```

```
(array([[ 0.99342831],
        [-0.2765286 ],
        [ 1.29537708],
        [ 3.04605971],
        [-0.46830675],
        [-0.46827391],
        [ 3.15842563],
        [ 1.53486946],
        [-0.93894877],
        [ 1.08512009]]),
 array([[ 6.98028492],
```

random (same) — X

$x_1 = 0.9934$

$y_1 = 4 + 3 * 0.9934 + 0$

y

$4 + 3*0.99342831 = 6.98028493$

Mean Squared Error - Cost Function for regression problems

$\frac{2}{2m}[\ \ ]$

For m training examples:

weight bias error ①

$$\underbrace{J}_{\text{cost function}}(w,b) = \frac{1}{2m}\sum_{i=1}^{m}\left(\underbrace{\hat{y}^{(i)}}_{\text{Predicted}} - \underbrace{y^{(i)}}_{\text{actual}}\right)^2$$

Mean Squared Error

③ ← ② ← ①

$(P-A)^2$

where:

- $y^{(i)}$ = actual output for sample i.
- $\hat{y}^{(i)}$ = predicted output.
- w, b = parameters (weights, bias).
- m = number of samples.
- i → ith row/sample

$(\text{error})^2$ → why square ??

- ⇒ to prevent positive and negative errors cancelling/nullifying each other.
- ⇒ squaring makes the error +ve
- ⇒ penalizes large errors more strongly than small errors

→ Division by m gives the mean/avg. making it independent of dataset size
↓
to get the avg. model error.

→ Factor $\frac{1}{2}$ is to simplify the derivative output (2 cancels when differentiating)

$y = x^2$	$y = \frac{1}{2}x^2$
$\frac{dy}{dx} = 2x$	$\frac{dy}{dx} = \frac{1}{\cancel{2}}(\cancel{2}x) = x$

Task

Do the below derivations:

$$\frac{\partial J}{\partial w} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)\cdot x^{(i)}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)$$

} gradient

[Cost Function] → MSE

Cost Function → MSE

$\hat{y} = \hat{\beta_0} + \hat{\beta_1} \mathbf{x}$ → $\hat{y} = \hat{\beta_0} \cdot x^0 + \hat{\beta_1} x$

(Predicted)

$\hat{y} = \begin{bmatrix} \hat{\beta_0} & \hat{\beta_1} \end{bmatrix}_{1\times2} \times \begin{bmatrix} 1 \\ x \end{bmatrix}_{2\times1}$

$\hat{y} = X \cdot \theta$

$$J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)} \right)^2$$

$\begin{bmatrix} \hat{\beta_0} & \hat{\beta_1} & \hat{\beta_2} & \cdots \end{bmatrix} \to \theta \to$ Parameters

```python
def compute_cost(X, y, theta):
    """
    Compute the mean squared error cost function
    """
    m = len(y) #no. of rows in the data
    return np.sum((X.dot(theta) - y)**2)/(2*m)
```

Linear Algebra Videos:

https://www.khanacademy.org/math/linear-algebra

$y = f(x)$

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\underbrace{h_\theta(x^{(i)})}_{P} - \underbrace{y^{(i)}}_{A})^2$$

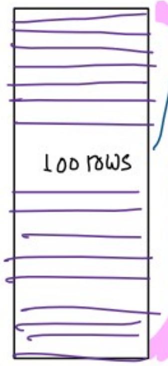
$$\boxed{h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1}$$

$\begin{bmatrix} \hat{\beta_0} & \hat{\beta_1} \end{bmatrix}$

BATCH GRADIENT DESCENT ALGORITHM (BGD)

→ (Vanilla Gradient Descent)

→ by default → BGD



Batch Gradient Descent is an optimization algorithm that is used to minimize the cost function, updating gradients

↓

updating weights and biases ONLY ONCE for the entire training dataset.

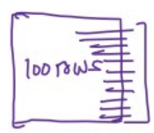
→ refers to the fact that the gradient is computed using the entire training dataset at each iteration (ONE EPOCH)

What is an Epoch?

An epoch → one complete pass through the entire training dataset by the model.

During one epoch, every training sample has been used once to update the model's parameters

↙ weights ↘ biases

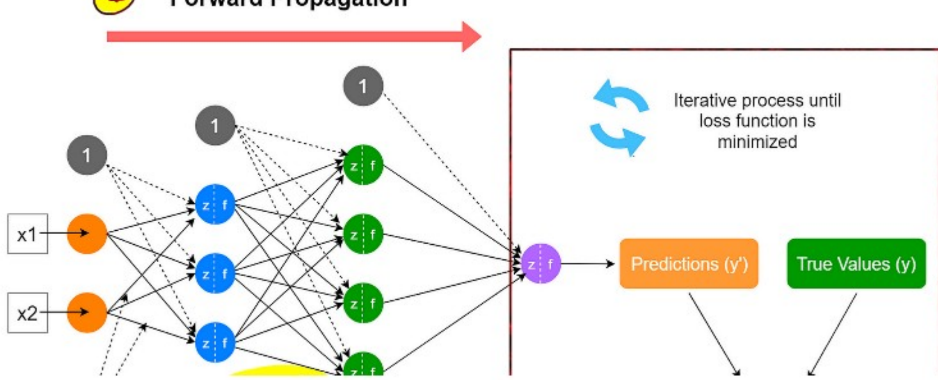


1 epoch.

** Pro-tip

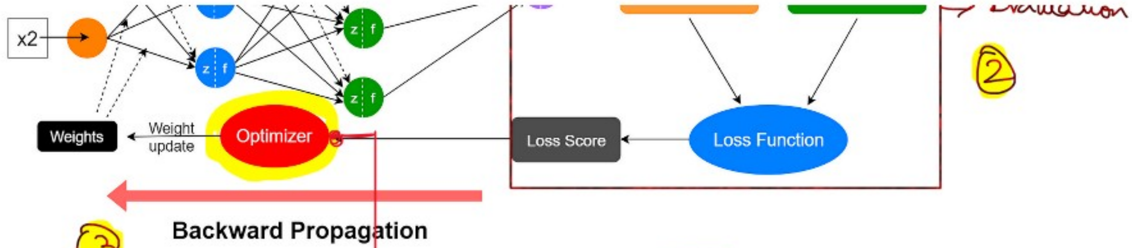
Batch Gradient Descent: → uses all training samples | rows → the entire training dataset to compute the gradient of the cost function.
(loss)

1



→ Model Evaluation

2



② ③

→ Batch Gradient Descent
→ Mini-Batch Gradient Descent
→ Stochastic Gradient descent

* Adam Optimizer.
↳ will be discussed in module #3.

Flowchart : Batch Gradient Descent (from scratch)

Start

→ Load the training dataset
& the validation / testing dataset.

→ Initialize parameters → weights (W), biases (b) → randomly → (small values)] θ_0 θ_1 → thetas

→ Set the hyperparameters

- Learning Rate α or η
- No. of epochs (N)

parameters not hyperparameters
↓
learn and adjust automatically

© apc

(Hyper) overrides →

→ Parameter
↓
when something goes wrong while driving who gets HYPER.

(Instructor)   (You)

[Parameters vs Hyperparameters → (HPT) → spend 10 mins]

For each epoch in range(N)
↳ # epochs

- Forward Pass

  weight matrix

  $y\_pred = (WX + b)$

  ↳ bias

- Compute loss & model evaluation

  $L = loss(y\_pred, y)$

  Regression # MSE

  Classification # cross-entropy

- Backward Pass

  $dW = \nabla W$ Loss - gradient of loss w.r.t. weight

  $db = \nabla b$ Loss - gradient of loss w.r.t. bias

- Update weights and biases

  $W_{new} = W_{old} - \underbrace{\alpha * dW}_{\text{step size}}$

  $b_{new} = b_{old} - \alpha * db$

- Plotting the loss function vs epoch

  accuracy vs epoch

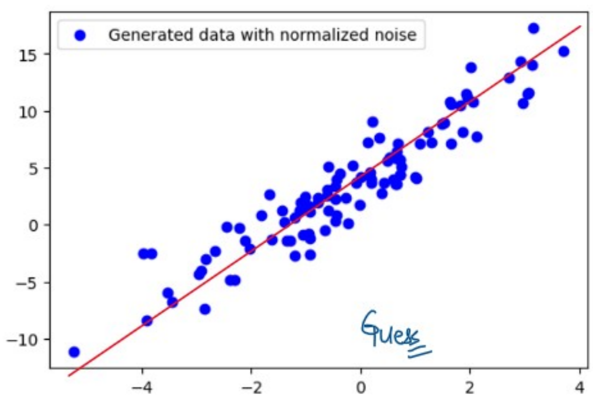
  error vs epoch.

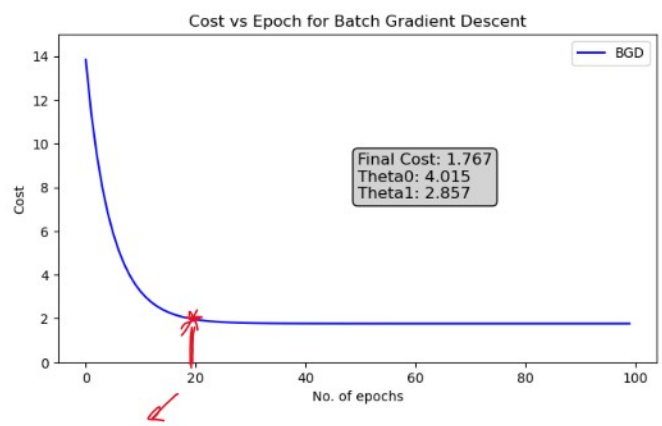
  Best fit line on the distribution

as many times as many epochs.

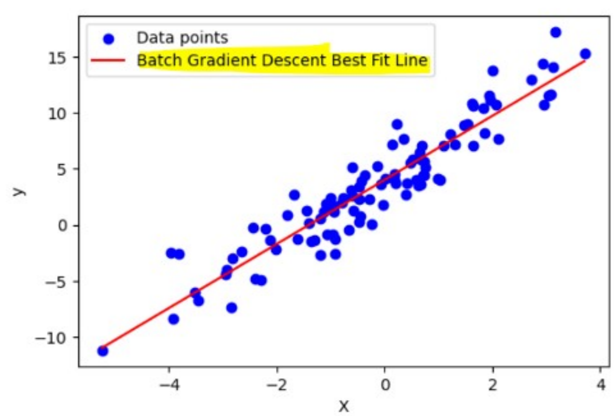
$$J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)} \right)^{2}$$

```python
def compute_cost(X, y, theta):
    """
    Compute the mean squared error cost function
    """
    m = len(y) #no. of rows in the data
    return np.sum((X.dot(theta) - y)**2)/(2*m)
```





model has stabilized near epoch 20



https://github.com/scikit-learn/scikit-learn/tree/main/sklearn

https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/neural_network/_stochastic_optimizers.py

TASK: Create the documentation for the BGD concept