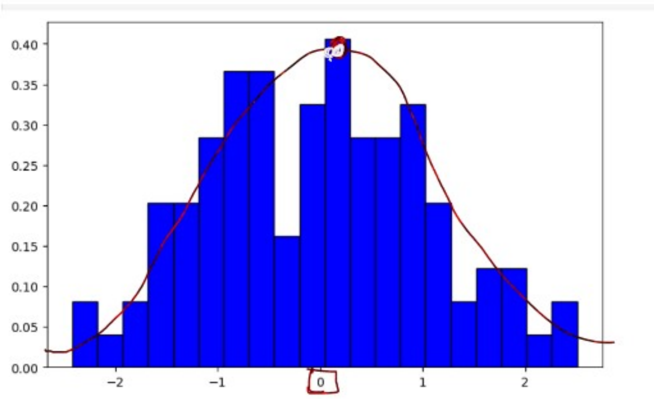
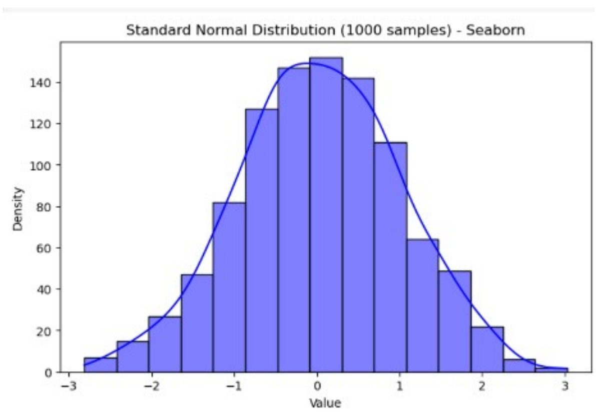
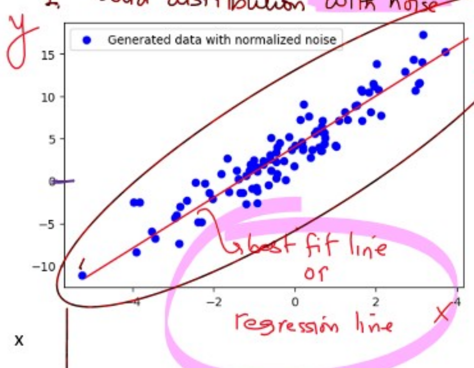
GDA Code Explanation

27 September 2025 11:55

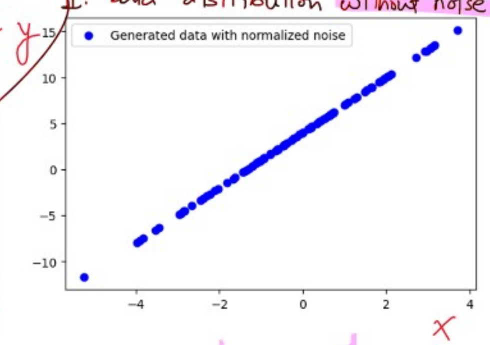




I. Data distribution with noise



II. data distribution without noise



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

ensures reproducibility

constant noise

0.1 (default)

```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
```

```python
"""
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
```

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

2 * random std. normal values

y → 4 + 3 * X + noise

→ 4 + ( ) + ( )

100 values

$y = 4 + 3x$

intercept (bias)

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

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

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

Mean Squared Error

③ ← ② ← ①

cost function

Predicted

actual

$(P-A)^2$

where:

- $y^{(i)}$ = actual output for sample i.
- $\hat{y}^{(i)}$ = predicted output.

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

$(P-A)^2$

$(\text{error})^2$ → 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)

$$y = x^2 \qquad y = \frac{1}{2}x^2$$

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

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

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

(Predicted)

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

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

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

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

$[\hat{\beta_0}\ \hat{\beta_1}\ \hat{\beta_2} \ldots] \rightarrow \theta \rightarrow$ 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

$y = f(x)$

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

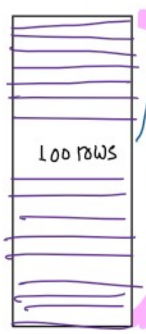
$$\boxed{h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1}$$

$[\hat{\beta_0}\ \hat{\beta_1}]$

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

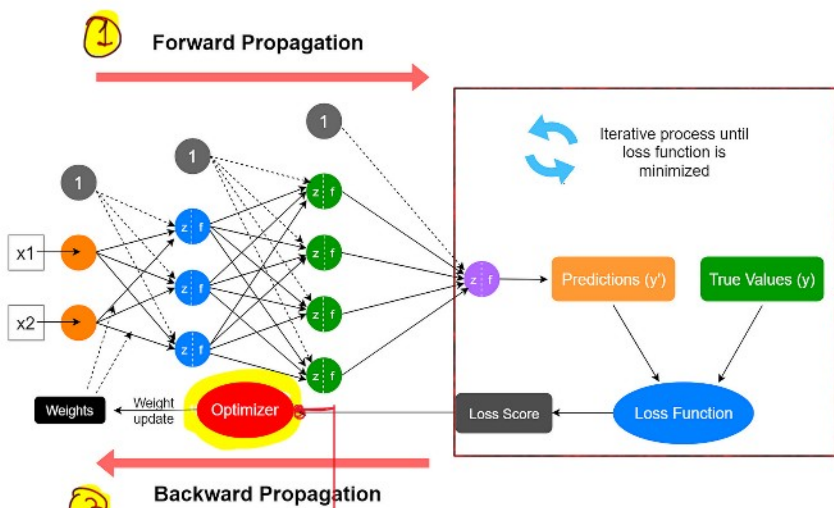
- weights
- biases

100 rows → 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 Forward Propagation



→ Model Evaluation ②

③ Backward Propagation

- → 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 hyperpara

→ 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 →

(Instructor) (You) → Parameter → when something goes wrong while driving who gets HYPER.

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

→ For each epoch in range(N) → # epochs

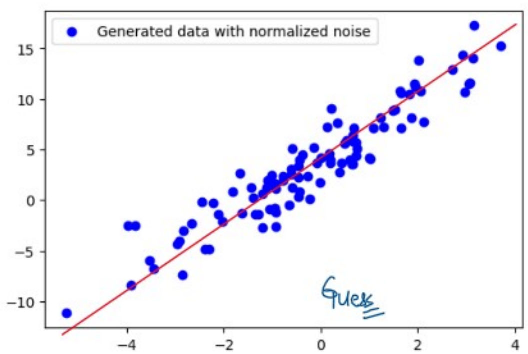
- Forward Pass
  $y\_pred = (WX + b)$ ← weight matrix, → bias
- Compute loss & model evaluation
  $L = loss(y\_pred, y)$ → Regression # MSE, Classification # cross-entropy
- Backward Pass
  $dW = \nabla W$ Loss – gradient of loss w.r.t. weight
  $db = \nabla b$ Loss – gradient of loss w.r.t. bias
- Update 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.

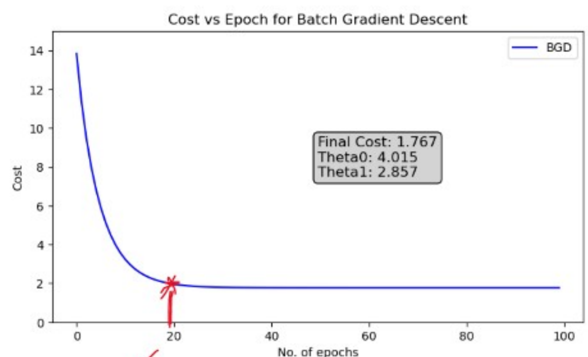
as many times as many epochs.

accuracy vs epoch
error vs epoch.
Best fit line on the distribution

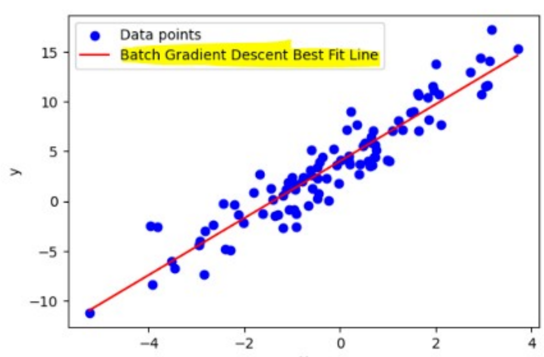
$$J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)} \right)^{2}$$

```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

COOKBOOK
MODELING BOOK - ***shared from my end***

MINI-BATCH GRADIENT DESCENT (MBGD)

MBGD is a trade-off | balance between BGD and SGD *

MBGD → (mini-batch gradient descent) → divide the training dataset into (small chunks of equal size) called mini-batch. → (mini-batches)

BGD → (Batch Gradient Descent) → [all the rows or entire training data]

SGD → (Stochastic Gradient Descent) → [one row randomly at a time]

Standard size: ($n = 32$ rows | training samples or examples. → (one mini-batch)

For ex: Training dataset size = 100 rows
std. batch size: $n = 32$

How many mini-batches ??

$$\frac{100}{32} =$$ 100/32=**3.125**

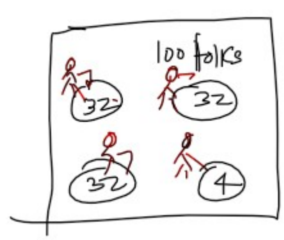
↓ take ceiling function

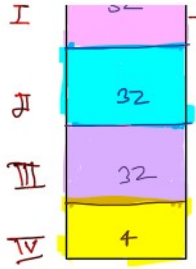
$$\left\lceil \frac{100}{32} \right\rceil = \lceil 3.125 \rceil = 4 \rightarrow 4 \text{ mini-batches}$$

In one epoch, how many gradient updates would happen for the above training dataset.

I | 32 | → one gradient update | mini-batch

II | 32 |





→ one gradient update | mini-batch

4 times

(one epoch)

32 4

Gaurav - Epoch is one full traverse of whole data or one full pass thru the entire training data

STOCHASTIC GRADIENT DESCENT (SGD)

→ refers to systems or processes that are random or probabilistic in nature

→ (uncertainty)

To predict: 1) In Bangalore, what's the chance of rain today evening?

(58% - probability of rainfall)

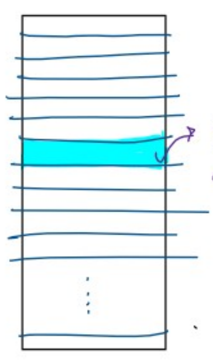
IMD: Indian Meteorological Deptt.

2) What's going to be traffic on ORR route around 9AM tmrw??

3) opening price of a company's stock everyday?

Uncertain

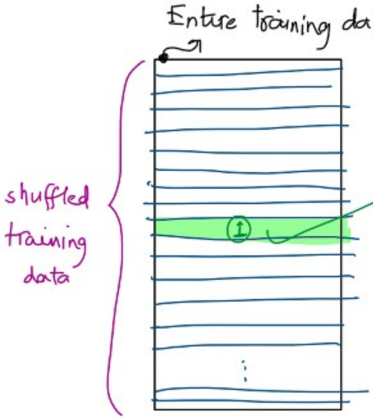
In ML/DL, stochastic describes algorithms or models that incorporate randomness in their operations.



→ picking one row randomly after shuffling.

SGD: It is an optimization technique (algorithm) used to minimize the loss function. To do so, SGD computes the gradient for a single row, randomly chosen from the shuffled training dataset ...

the gradient for a single row, randomly chosen from the shuffled training dataset at each iteration



Leveraging SGD, the gradient is computed for each and every sampled row from the shuffled training dataset (random sampling)

↓

and update bias & weight for each sample row one by one.

Training dataset: m = 100 → 100 rows in the training dataset

\# Epochs = 100

n = 32 (batch-size)

How many gradient updates or iteration would happen in:

BGD : 100 updates $\xleftarrow{\times 100}$ 1 | epoch

MBGD: 400 updates $\xleftarrow{\times 100}$ 4 | epoch

SGD : 10,000 updates $\xleftarrow{\times 100}$ 100 | epoch

Conclusion

	Training Rows = 100	100 epochs
BGD	1 gradient update / epoch	1 X 100 = 100 gradient updates
MBGD*	4 gradient updates / epoch	4 X 100 = 400 gradient updates
SGD	100 gradient updates / epoch	100 X 100 = 10,000 gradient updates

*: No. of Training rows = 100
std. Batch size = 32 : [default standard batch size decided by top DL researchers]

Batch Gradient Descent - Code Explanation

- For linear regression problem, we use `mean-squared error` (MSE)

```
[ ]: def compute_cost(X, y, theta):

    """
    Compute the mean squared error cost function

    """
    m = len(y)
    return np.sum((X.dot(theta) - y)**2)/(2*m)
```

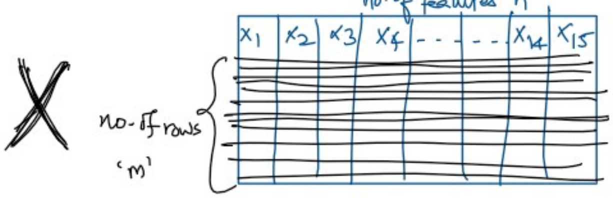
$$\frac{\sum(\text{Prediction} - \text{Actual})^2}{2m}$$

In BGD

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

where 'X' is the feature matrix of shape (m, n) → no. of features

no. of rows



→ feature matrix

θ : is the parameter array (weights & biases)

[Cost Function] → MSE

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

(Predicted)

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

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

Gradient of cost Function

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(X\cdot\theta - y)^2$$

let us find the gradient of cost function

$$\frac{\partial}{\partial\theta}J(\theta) = \frac{\partial}{\partial\theta}\left[\frac{1}{2m}\sum_{i=1}^{m}(X\cdot\theta - y)^2\right]$$

$$= \frac{1}{\cancel{2}m}\times\cancel{2}\sum(X\cdot\theta - y)\cdot\frac{\partial}{\partial\theta}(X\cdot\theta - y)$$

$$= \frac{1}{m}\sum(X\cdot\theta - y)\cdot X$$

$$\frac{\partial}{\partial\theta}J(\theta) = \frac{1}{m}\left[\sum_{i=1}^{m}X\cdot(X\cdot\theta - y)\right]$$

$$\frac{\partial}{\partial x}\underbrace{(x\cdot\theta - y)}_{X}{}^2 \qquad \frac{\partial}{\partial x}X^2 = 2X\cdot\frac{\partial X}{\partial x}$$

$$= 2(x\cdot\theta - y)\times\frac{\partial}{\partial x}(x\cdot\theta - y)$$

$$= 2(x\cdot\theta - y)\cdot\theta$$

$$= 2\theta(x\cdot\theta - y)$$

$$= \frac{d}{dx}(K\cdot x - y)$$

$$= \frac{d}{dx}Kx - \cancel{\frac{d}{dx}(y)}^{\,0}$$

$$= K$$

AI generated

Derivation of Gradient of Cost Function (Linear Regression)

We start with the **Mean Squared Error (MSE) cost function:**

ith row|sample.

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(x^{(i)}\theta - y^{(i)})^2$$

Step 1: Differentiate with respect to θ

$$\frac{\partial J(\theta)}{\partial\theta} = \frac{\partial}{\partial\theta}\left[\frac{1}{2m}\sum_{i=1}^{m}(x^{(i)}\theta - y^{(i)})^2\right] \quad ①$$

Step 2: Apply the power rule

$$= \frac{1}{2m}\sum_{i=1}^{m}2(x^{(i)}\theta - y^{(i)})\cdot\frac{\partial(x^{(i)}\theta - y^{(i)})}{\partial\theta} \quad ②$$

Step 3: Simplify (the 2 cancels with 1/2)

$$= \frac{1}{m}\sum_{i=1}^{m}(x^{(i)}\theta - y^{(i)})\cdot x^{(i)} \quad ③$$

Step 4: Write in vectorized form

$$\nabla_\theta J(\theta) = \frac{1}{m}X^T(X\theta - y) \quad ④$$

code snippet

```
# Compute gradient
gradients = (X.T.dot(X.dot(theta) - y)) / m
```

(5)

✓ code snippet

```python
    # Compute gradient
    gradients = (X.T.dot(X.dot(theta) - y)) / m
```

(5)

└ (Transpose)

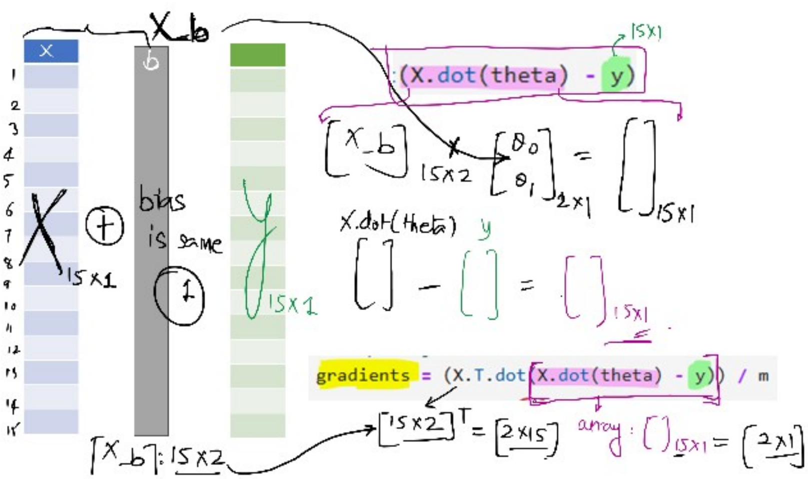
```python
def bgd(X, y, theta, learning_rate=0.01, epochs=100):
    """
    Batch Gradient Descent (Vanilla) using the entire training dataset
    X = Array of X with the added bias (X_b)
    y = Vector of y
    theta: Array of weight & bias parameters randomly assigned
    learning_rate: alpha value set to default 0.01
    epochs: number of times model will run through the entire training dataset
    """
    m = len(y)  # number of training rows

    # Arrays to track cost and theta history
    cost_history = np.zeros(epochs)                   # 1D array to store cost after each epoch
    theta_history = np.zeros((epochs, theta.shape[0]))  # 2D array to store parameter values

    for epoch in range(epochs):
        # Compute gradient
        gradients = (X.T.dot(X.dot(theta) - y)) / m
```

(Prediction - Actual) = Error

$$\text{gradients} = \frac{\text{Sum}(X * \text{Error})}{m}$$



100

```python
for epoch in range(epochs):
    # Compute gradient
    gradients = (X.T.dot(X.dot(theta) - y)) / m

    # Update parameters
    theta = theta - (learning_rate * gradients)

    # Compute cost
    cost = compute_cost(X, y, theta)
```

→ calculate gradient for each epoch

→ update θ values → weight → bias

→ compute cost

→ bias

```
    # Compute cost
    cost = compute_cost(X, y, theta)  → compute cost

    # Store history
    cost_history[epoch] = cost
    theta_history[epoch, :] = theta.T

return theta, cost_history, theta_history
```

TASK: BGD code as reference → MBGD / → SGD

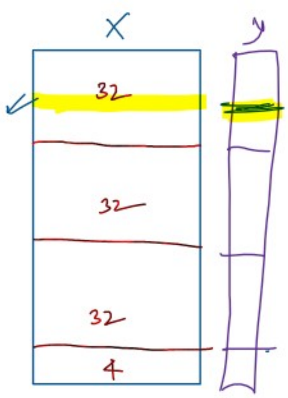
MBGD  mini-batch selected and then GDA.

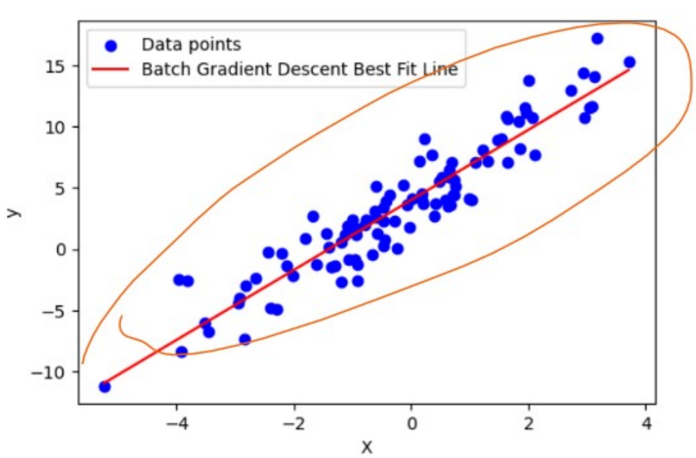
SGD  random row selected and then GDA

As aligned, we can discuss it on 12th oct

Mini-Batch Gradient Descent (MBGD)

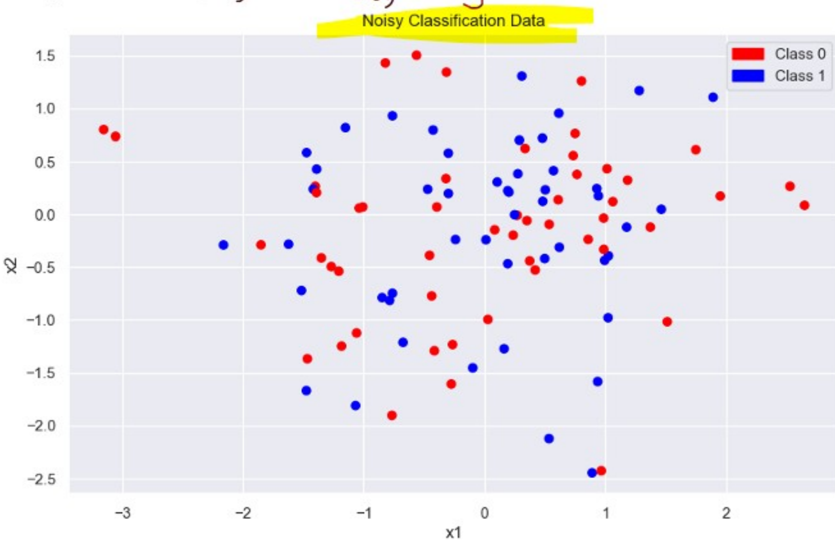
MBGD Code Explanation

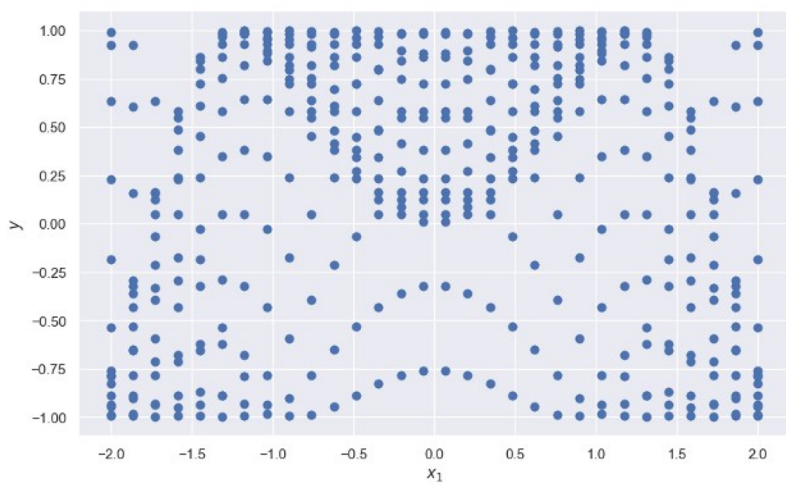




[Example distribution for a noisy data]

[Example distribution for a noisy data]





Comparison between BGD, SGD, MBGD

Characteristic	Batch Gradient Descent (BGD)	Stochastic Gradient Descent (SGD)	Mini-batch Gradient Descent (MBGD)
	BGD	SGD	MBGD
Update Frequency → (weights & biases)	After processing entire dataset	After each training example → every row	After each mini-batch (subset of data)
Memory Requirement	High (entire dataset)	Low (one example)	Medium (mini-batch)
Speed per Update	Slow (needs entire dataset)	Fast (one example at a time)	Medium (mini-batch size)
Convergence	Smooth, more stable	Noisy, can be erratic	Balanced, smoother than SGD
Handling Large Datasets	Inefficient	Efficient	Efficient (most popular choice)
Jumping out of Local Minima	Difficult	Easier (due to noise)	Easier than BGD
Application	Small datasets	Large datasets, online learning	Deep learning, large datasets

BGD

SGD
– avoid this!

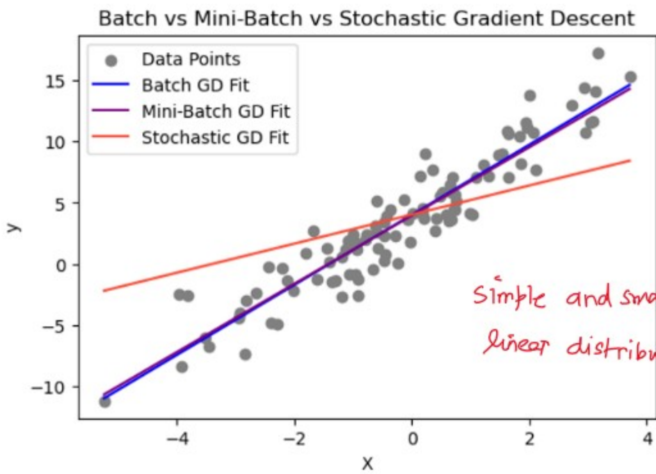
MBGD

MBGD >> BGD >>>>>>> SGD

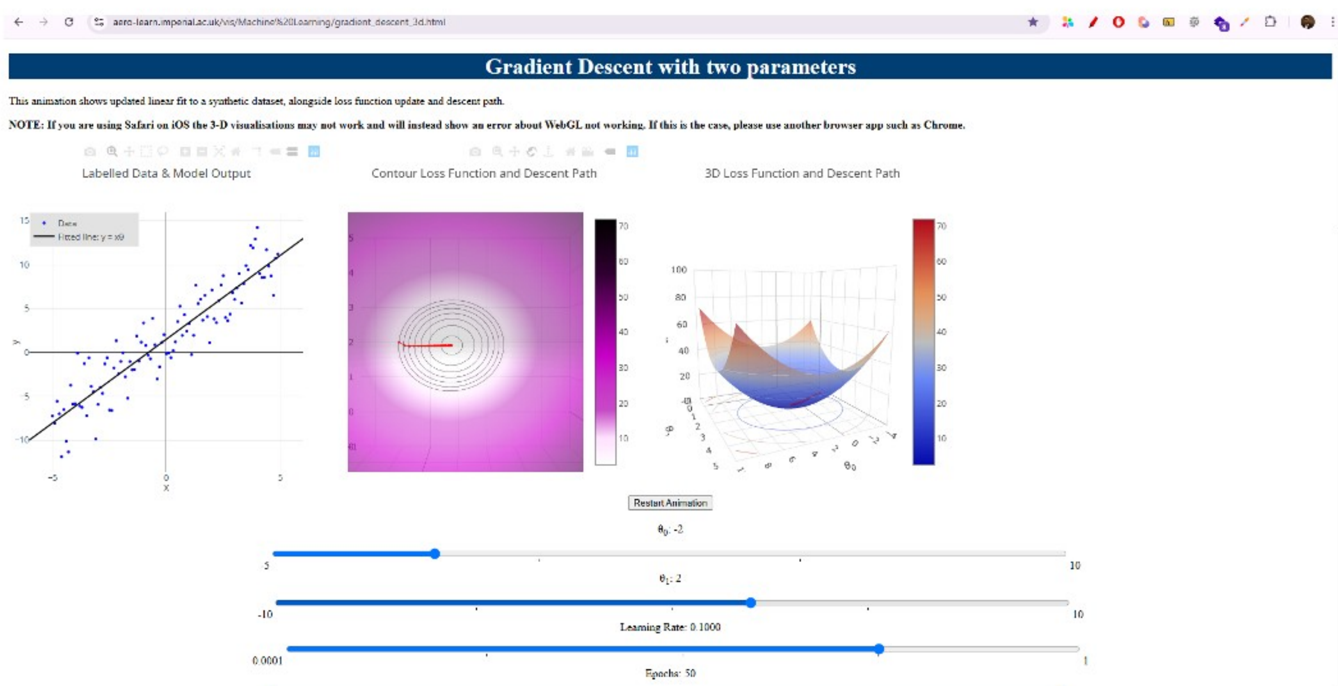
least

Most | Highly preferred.

least preferred



Simple and small linear distribution



https://aero-learn.imperial.ac.uk/vis/Machine%20Learning/gradient_descent_3d.html