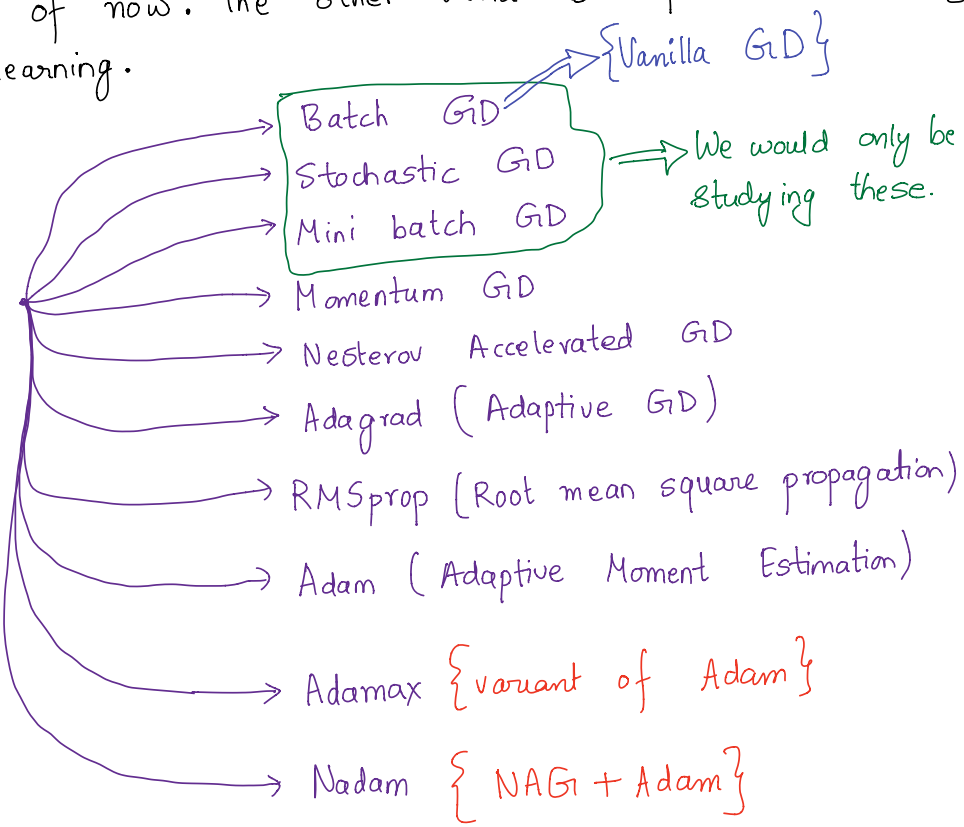
① {Types of gradient descent}

We basically have 10 types of GD. However, we would only be studying 3 as of now. The other variations of GD are mostly used in deep learning.



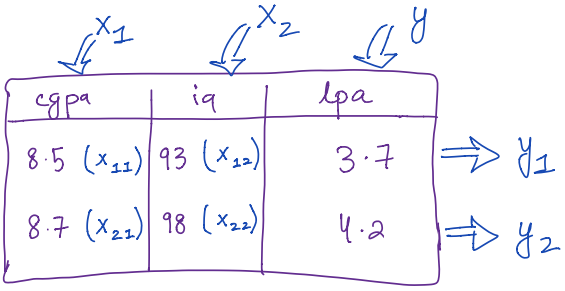
The only difference between the these 3 types of GD is the # of data points required for single updation of the coefficients

The Batch Gradient descent is just the normal GD in which the coefficients are updated once after every epoch. So if we 1000 epochs, we would be updating the coefficients only 1000 times.

② {Understanding Batch GD with simple example}

Consider the dataset having 2 input features namely, CGPA and

IQ, and one output feature, namely package (lpa). For the sake of simplicity, we will only consider 2 data points i.e., only 2 rows.



We need to find 3 coefficients β_0, β_1 and β_2 in order to train our linear regression model.

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

Thus,
$$\hat{y}_1 = \beta_0 + \beta_1 x_{11} + \beta_2 x_{12}$$
$$\hat{y}_2 = \beta_0 + \beta_1 x_{21} + \beta_2 x_{22}$$

Suppose we have m columns (predictors), then the above equation could be written as:-

$$\hat{y}_1 = \beta_0 + \beta_1 x_{11} + \beta_2 x_{12} + \ldots + \beta_m x_{1m}$$
$$\hat{y}_2 = \beta_0 + \beta_1 x_{21} + \beta_2 x_{22} + \ldots + \beta_m x_{2m}$$

Here, we have (m+1) coefficients to be found namely $\langle \beta_0, \beta_1, \ldots, \beta_m \rangle$ where $m \triangleq$ # of predictors

Now further suppose that we have n data points in our dataset. Then we would have n $\hat{y}$'s.

$$\hat{y}_1 = \beta_0 + \beta_1 x_{11} + \beta_2 x_{12} + \dots + \beta_m x_{1m}$$

$$\vdots$$

$$\hat{y}_n = \beta_0 + \beta_1 x_{n1} + \beta_2 x_{n2} + \dots + \beta_m x_{nm}$$

This could also be written as.

$$\hat{y}_i = \beta_0 + \sum_{j=1}^{m} \beta_j x_{ij}$$

The error that our model makes on each input is

$$E = \hat{y}_i - y_i$$

So, the Loss function (cost function) can be written as

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

This is a function of $(m+1)$ variables. Thus $L: \mathbb{R}^{m+1} \to \mathbb{R}$ can be viewed as a function of $\beta_0, \beta_1, \dots, \beta_m$.

$$L(\beta_0, \dots, \beta_m) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$

$$L(\beta_0, \dots, \beta_m) = \frac{1}{n} \sum_{i=1}^{n} \left(\beta_0 + \sum_{j=1}^{m} \beta_j x_{ij} - y_i\right)^2$$

In order to find the optimal value of $\beta_0, \ldots, \beta_m$ for which this function L takes minimum value, we need to use the method of GD which is a first order linear approximation technique. The new values of $\beta_0, \ldots, \beta_m$ are calculated by using the learning rate η and the value of gradient of L at the old values of $\beta_0, \ldots, \beta_m$.

Thus,

$$(\beta_0, \ldots, \beta_m)_{new} = (\beta_0, \ldots, \beta_m)_{old} - \eta \left(\nabla L\right)_{\text{evaluated at } (\beta_0, \ldots, \beta_m)_{old}}$$

For calculating ∇L at $(\beta_0, \ldots, \beta_m)_{old}$, we need to calculate

$$\frac{\partial L}{\partial \beta_0}, \ldots, \frac{\partial L}{\partial \beta_m}$$

③ { Calculating $\frac{\partial L}{\partial \beta_0}, \frac{\partial L}{\partial \beta_1}, \ldots, \frac{\partial L}{\partial \beta_m}$ }

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

Now $$\frac{\partial L}{\partial \beta_0} = \frac{1}{n} \sum_{i=1}^{n} 2(\hat{y}_i - y_i)\left(\frac{\partial \hat{y}_i}{\partial \beta_0}\right)$$ { as $\hat{y}_i$ is a function of $\beta_0, \ldots, \beta_m$ }

Silly, $$\frac{\partial L}{\partial \beta_m} = \frac{1}{n} \sum_{i=1}^{n} 2(\hat{y}_i - y_i)\left(\frac{\partial \hat{y}_i}{\partial \beta_m}\right)$$

These derivatives could be enclosed in a vector

$$\left(\frac{\partial L}{\partial \beta_0}, \dots, \frac{\partial L}{\partial \beta_m} \right) = \left(\frac{1}{n} \sum_{i=1}^{n} 2(\hat{y}_i - y_i) \frac{\partial \hat{y}_i}{\partial \beta_j} \right)_{j=0}^{m}$$

Thus, $$\frac{\partial L}{\partial \beta_j} = \frac{1}{n} \sum_{i=1}^{n} 2(\hat{y}_i - y_i) \frac{\partial \hat{y}_i}{\partial \beta_j}, \quad \forall\, j \in \{0, \dots, m\}$$

Now since $$\hat{y}_i = \beta_0 + \sum_{j=1}^{m} \beta_j x_{ij}$$

$$\Rightarrow \frac{\partial \hat{y}_i}{\partial \beta_j} = \begin{cases} 1, & \text{if } j = 0 \\ x_{ij}, & \text{if } j \neq 0 \end{cases}$$

Thus, $$\frac{\partial L}{\partial \beta_0} = \frac{1}{n} \sum_{i=1}^{n} 2(\hat{y}_i - y_i) \frac{\partial \hat{y}_i}{\partial \beta_0} \Rightarrow 1$$

$$\boxed{\frac{\partial L}{\partial \beta_0} = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)} \quad \text{— (1)}$$

and $$\frac{\partial L}{\partial \beta_j} = \frac{1}{n} \sum_{i=1}^{n} 2(\hat{y}_i - y_i) \frac{\partial \hat{y}_i}{\partial \beta_j} \Rightarrow x_{ij} \text{ where } j \neq 0$$

$$\Rightarrow \frac{\partial L}{\partial \beta_j} = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) x_{ij}, \quad \boxed{j \neq 0}$$

Summarizing the above equations, we get

$$\frac{\partial L}{\partial \beta_j} = \begin{cases} \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) , & j = 0 \\ \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) x_{ij} , & j \neq 0 \end{cases}$$

and thus now the coefficients $\beta_0, \ldots, \beta_m$ can be updated as

$$(\beta_0, \ldots, \beta_m)_{new} = (\beta_0, \ldots, \beta_m)_{old} - \eta \left(\frac{\partial L}{\partial \beta_0}, \ldots, \frac{\partial L}{\partial \beta_m} \right)_{\substack{\text{evaluated at} \\ (\beta_0, \ldots, \beta_m)_{old}}}$$

It is very clear from the above equation, that for getting the new values of β's we need to evaluate $\frac{\partial L}{\partial \beta}$'s at the old value of β's. Each $\frac{\partial L}{\partial \beta}$ has a summation which is running from 1 to n. Thus, while calculating each derivative in each epoch, we need to make use of all the data points of our dataset which makes the overall process of updation slower.

④ { code for batch gradient descent }

we will make our own class of batch GD. In this we shall be making 2 methods. The `__init__()` constructor will take two parameters namely, learning_rate and epochs

```
def __init__(self, learning_rate, epochs):
    self.coef_ = None
    self.intercept_ = None
    self.lr = learning_rate
    self.epochs = epochs
```

The predict() method is fairly easy and takes in only 1 parameter namely X_test. Further, let us assume that we have n data points and m predictors. Thus,

$$X \in \mathbb{R}^{n \times m} \qquad X\_test \in \mathbb{R}^{n_1 \times m}$$

(n_1 : # of rows in X_test)

$$y \in \mathbb{R}^{1 \times n}$$

$$self.coef\_ \in \mathbb{R}^{1 \times m} \triangleq (\beta_1, \ldots, \beta_m)$$

```
def predict(self, X_test):
    return X_test @ self.coef_.T + self.intercept_
```

$\Downarrow$

$X\_test \in \mathbb{R}^{n_1 \times m}$ and $self.coef\_ \in \mathbb{R}^{1 \times m}$. So, $self.coef\_.T \in \mathbb{R}^{m \times 1}$. Thus, $X\_test \,@\, self.coef\_.T \in \mathbb{R}^{n_1 \times 1}$.

The final and the most important method is fit(). This is the most important method and takes 2 parameters, X_train

and y_train.

$$X\_train \in \mathbb{R}^{n_0 \times m}$$

$$y\_train \in \mathbb{R}^{1 \times n}$$

```
def fit(self, X_train, y_train):
    self.coef_ = np.ones(X_train.shape[1])   # initialize β1,...,βm = 1
    self.intercept_ = 0                      # initialize β0 = 0

    for _ in range(epochs):
        y_pred = self.intercept_ + X_train @ self.coef_
        del_beta_not = [2 * np.sum(y_pred - y)] / X_train.shape[0]
        others = 2 * ((y_pred - y) @ X_train) / X_train.shape[0]
        self.intercept_ = self.intercept_ - self.lr * del_beta_not
        self.coef_ = self.coef_ - self.lr * others
```

β_0

$$\# \frac{\partial L}{\partial \beta_0} = \frac{2}{n_0} \sum_{i=1}^{n} (\hat{y}_i - y_i)$$

n_0

$n_0 \times 1$ $\quad$ $n_0 \times m$ $\quad$ n_0

$$\left(\frac{\partial L}{\partial \beta_1}, \cdots, \frac{\partial L}{\partial \beta_m} \right) = \frac{2}{n_0} (\hat{y} - y) X\_train$$

⑤ {Problems with Batch gradient descent}

ⓐ {Computational Expanse}

For large datasets, calculating the gradient of the cost function over the entire dataset can be computationally expansive and slow.

ⓑ {Memory Usage}

It requires loading the entire dataset into memory, which can be impractical for very large datasets.

ⓒ {Convergence Issues}

It may converge slowly if the cost function has many local minima or saddle points

ⓓ {Stuck in local minima}

In case of convex functions, the batch GD works fine. However, when we study deep learning, the loss function is often NOT a simple convex function. It consists of multiple local minimas and global minima. The batch GD algorithm is very likely to be trapped inside a local minim compared to other algorithms like Stochastic GD or mini batch gradient descent.