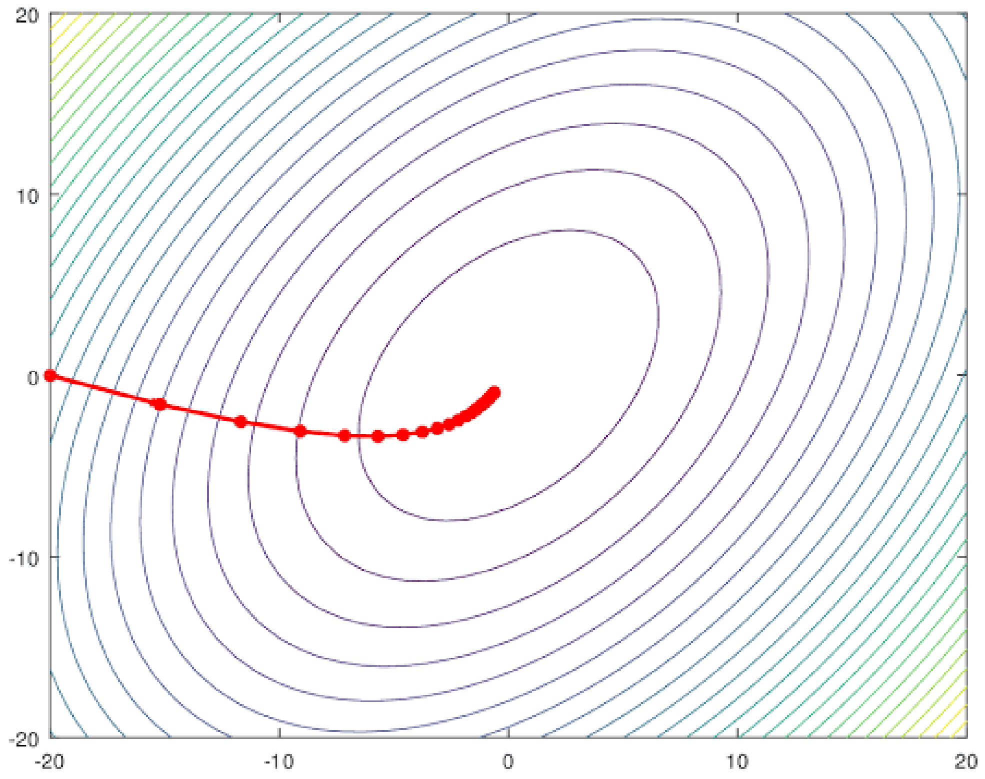
# Batch gradient descent

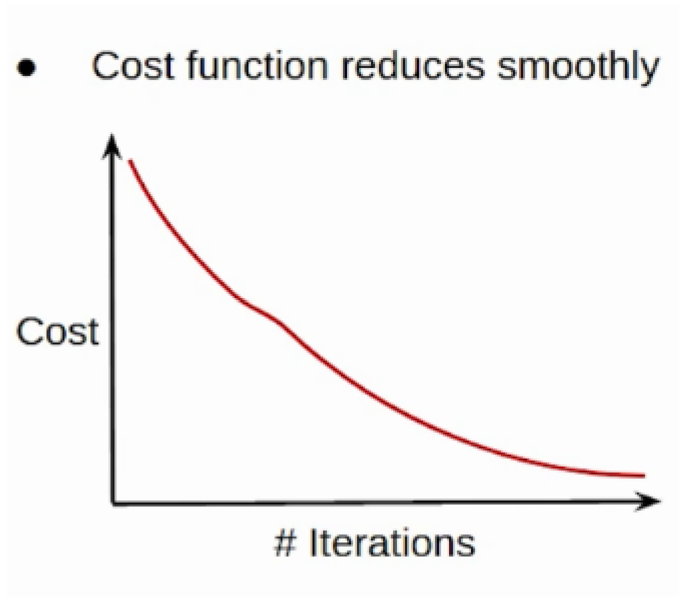
Batch gradient descent is an optimization algorithm used in machine learning and deep learning for finding the optimal parameters of a model. It is called "batch" because it updates the model's parameters using the gradients computed on the entire training dataset. In this detailed explanation, I'll walk you through the steps involved in batch gradient descent.



- Batch gradient descent is an optimization algorithm used to minimize the cost function in machine learning models. It works by iteratively updating the model parameters in the direction of the negative gradient of the cost function. The gradient of the cost function is a vector that points in the direction of the steepest ascent of the cost function. By moving in the opposite direction of the gradient, we can gradually decrease the cost function until it reaches a minimum.
- In batch gradient descent, the **entire training dataset is used to calculate the gradient of the cost function at each iteration**. This means that the gradient is calculated once per epoch, where an epoch is one pass through the entire training dataset.

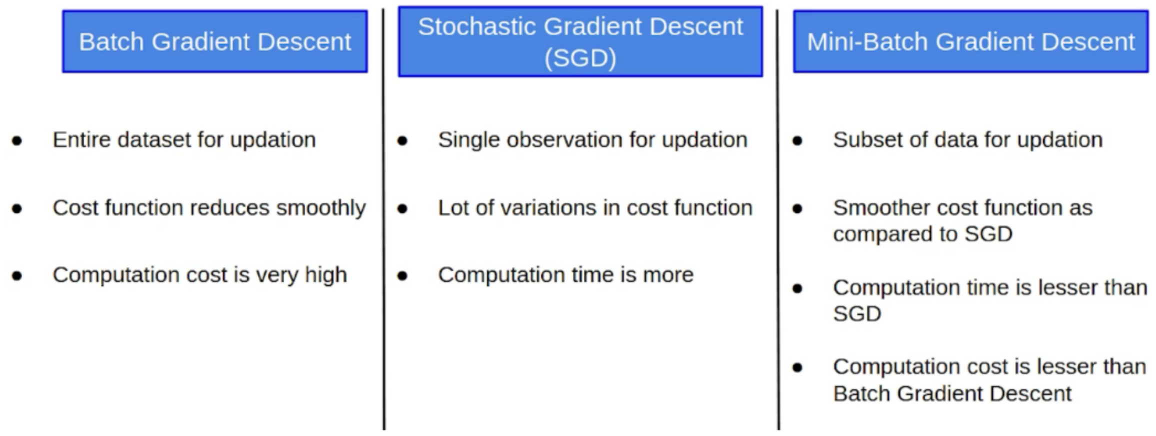
Entire Batch size
Batch Gradient descent

- The main advantage of batch gradient descent is that it is very computationally efficient. Since the gradient is calculated once per epoch, there is no need to recalculate the gradient for each individual training example. This makes batch gradient descent a good choice for large datasets.
- However, **batch gradient descent can also be slow to converge**. This is because the gradient of the cost function can be very noisy, especially for large datasets. This noise can make it difficult for the algorithm to converge to a minimum.



To address this issue, we can use a technique called mini-batch gradient descent. In mini-batch gradient descent, the training dataset is divided into smaller batches. The gradient of the cost function is then calculated for each batch, and the model parameters are updated accordingly. This makes the gradient less noisy, which can help the algorithm converge more quickly.

In general, **batch gradient descent is a good choice for large datasets**. However, if the dataset is small or noisy, then mini-batch gradient descent may be a better choice.



| Batch Gradient Descent | Stochastic Gradient Descent (SGD) | Mini-Batch Gradient Descent |
|---|---|---|
| • Entire dataset for updation | • Single observation for updation | • Subset of data for updation |
| • Cost function reduces smoothly | • Lot of variations in cost function | • Smoother cost function as compared to SGD |
| • Computation cost is very high | • Computation time is more | • Computation time is lesser than SGD |
| | | • Computation cost is lesser than Batch Gradient Descent |

Here are some of the benefits and drawbacks of batch gradient descent:

**Benefits:**

- Computationally efficient
- Stable convergence
- Less sensitive to noise

**Drawbacks:**

- Can be slow to converge for large datasets
- Can be sensitive to outliers

Here are some examples of when batch gradient descent might be used:

- Training a linear regression model on a large dataset
- Training a neural network on a small dataset
- Training a model on a dataset with a lot of noise

## Step by step Process

1. **Initialize Parameters**: Start by initializing the model's parameters, such as weights and biases, with random values. These parameters will be iteratively updated during the training process.

2. **Define the Cost Function**: Choose an appropriate cost function that measures the discrepancy between the predicted values of the model and the actual values in the training dataset. The most commonly used cost function is the mean squared error (MSE), but there are other options depending on the problem at hand.

3. **Compute the Gradients**: Calculate the gradients of the cost function with respect to each parameter of the model. The gradients indicate the direction and magnitude of the steepest ascent or descent in the parameter space. To compute the gradients, perform the following steps:

a. **Forward Propagation**: Pass the training dataset through the model to obtain the predicted outputs. Each instance in the dataset is fed forward through the model's layers, applying the necessary activation functions and using the current parameter values.

b. **Compute Loss**: Calculate the cost function using the predicted outputs and the actual targets from the training dataset. This step quantifies the model's performance on the training data.

c. **Backpropagation**: Perform backpropagation to compute the gradients of the cost function with respect to each parameter. Backpropagation involves propagating the error gradients backward through the layers of the model using the chain rule of calculus.

4. **Update Parameters**: After computing the gradients, update the parameters of the model to minimize the cost function. The update rule for each parameter is given by:

   ```
   parameter = parameter - learning_rate * gradient
   ```

   Here, the learning rate determines the step size taken in the direction of the gradients. It is a hyperparameter that needs to be carefully chosen. A large learning rate may cause overshooting, while a small learning rate can result in slow convergence.

5. **Repeat Steps 3-4**: Iterate over steps 3 and 4 until a stopping criterion is met. This criterion can be a maximum number of iterations or a threshold on the improvement of the cost function.

6. **Evaluate Model**: Once the training process is complete, evaluate the trained model's performance on a separate validation or test dataset. This step gives an estimate of how well the model is likely to generalize to unseen data.

Batch gradient descent has several advantages, including convergence to a global minimum (for convex problems) and more stable updates due to the use of the entire dataset. However, it can be computationally expensive, especially for large datasets, since it requires calculating gradients on the entire dataset for each update.

There are variations of gradient descent, such as stochastic gradient descent (SGD) and mini-batch gradient descent, which address the computational limitations of batch gradient descent. SGD updates the parameters using only a single instance at a time, while mini-batch gradient descent uses a small batch of instances. These variations provide a trade-off between

In [2]:
```
# code

import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
```

In [3]:
```
from sklearn.datasets import load_diabetes

# Load the diabetes dataset
X, y = load_diabetes(return_X_y=True)
```

In [4]:
```
print(X.shape)
print(y.shape)
```

```
(442, 10)
(442,)
```

In [5]:
```
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_stat
```

In [6]:
```
# Import the required library/dependency
from sklearn.linear_model import LinearRegression

# Create an instance of the LinearRegression class
reg = LinearRegression()

# Train the model using the training data
reg.fit(X_train, y_train)
```

Out[6]:
```
▾ LinearRegression
LinearRegression()
```

In [7]:
```
# Print the coefficients of the linear regression model
print(reg.coef_)

# Print the intercept of the linear regression model
print(reg.intercept_)
```

```
[  -9.15865318 -205.45432163  516.69374454  340.61999905 -895.5520019
  561.22067904  153.89310954  126.73139688  861.12700152   52.42112238]
151.88331005254167
```

In [8]:
```
# Predict using the regression model
y_pred = reg.predict(X_test)

# Calculate the R2 score
r2_score(y_test, y_pred)
```

Out[8]:
```
0.4399338661568968
```

In [9]:
```
X_train.shape
```

Out[9]:
```
(353, 10)
```

In [10]:
```
class GDRegressor:
    """
    Gradient Descent Regressor.
    """

    def __init__(self, learning_rate: float = 0.01, epochs: int = 100):
        """
        Initialize the GDRegressor.

        Args:
            learning_rate (float): Learning rate for gradient descent.
            epochs (int): Number of training epochs.
        """
        self.coef_ = None
        self.intercept_ = None
        self.lr = learning_rate
        self.epochs = epochs

    def fit(self, X_train: np.ndarray, y_train: np.ndarray):
        """
        Fit the model to the training data.

        Args:
            X_train (np.ndarray): Input features for training.
            y_train (np.ndarray): Target values for training.
        """
        # Initialize coefficients
        self.intercept_ = 0
        self.coef_ = np.ones(X_train.shape[1])

        for _ in range(self.epochs):
            # Update coefficients and intercept
            y_hat = np.dot(X_train, self.coef_) + self.intercept_
            intercept_der = -2 * np.mean(y_train - y_hat)
            self.intercept_ -= self.lr * intercept_der

            coef_der = -2 * np.dot((y_train - y_hat), X_train) / X_train.shape
            self.coef_ -= self.lr * coef_der

        print(self.intercept_, self.coef_)

    def predict(self, X_test: np.ndarray) -> np.ndarray:
        """
        Make predictions on the test data.

        Args:
            X_test (np.ndarray): Input features for testing.

        Returns:
            np.ndarray: Predicted values.
        """
        return np.dot(X_test, self.coef_) + self.intercept_
```

## Explanation:

The code defines a class called GDRegressor which stands for Gradient Descent Regressor. This class implements a linear regression model using the gradient descent algorithm.

The class has the following attributes:

- coef_: Represents the coefficients or weights of the linear regression model.
- intercept_: Represents the intercept or bias term of the linear regression model.
- lr: Represents the learning rate, which determines the step size for each update during gradient descent.
- epochs: Represents the number of training epochs, which is the number of times the algorithm will iterate over the training data.

**The class has three main methods:**

1. **init**(self, learning_rate: float = 0.01, epochs: int = 100): This is the constructor method that initializes the attributes of the class, such as the learning rate and number of epochs.
2. fit(self, X_train: np.ndarray, y_train: np.ndarray): This method is used to train the model on the provided training data. It takes in two arguments: X_train, which is a numpy array representing the input features for training, and y_train, which is a numpy array representing the target values for training. Inside this method, the coefficients and intercept are initialized, and then the gradient descent algorithm is performed for the specified number of epochs to update these values.
3. predict(self, X_test: np.ndarray) -> np.ndarray: This method is used to make predictions on the provided test data. It takes in X_test, a numpy array representing the input features for testing, and returns a numpy array of predicted values.

The code also includes a blank standard output (STDOUT) and a blank result, which means that when you run this code, it won't produce any output or result.

In [18]:
```
# Intialising with 10 epochs , lr 0.01
gdr = GDRegressor(epochs=10,learning_rate=0.01)
```

In [19]:
```
gdr.fit(X,y) # intercept_value =151.88331005254167 , Ans = 27.829351236336386
```

```
27.829351236336386 [1.13620067 1.03053696 1.42798273 1.3217294  1.15326223 1.
12560266
 0.71170428 1.3133775  1.41256165 1.27837633]
```

In [20]:
```
# updated with 100 epochs

gdr = GDRegressor(epochs=100,learning_rate=0.01)
```

In [21]:
```
gdr.fit(X,y) # intercept_value =151.88331005254167 , Ans = 131.95760905649124
```

```
131.95760905649124 [ 2.34829646  1.29472956  5.25588176  4.19688016  2.513487
68  2.23692197
 -1.86243151  4.10621794  5.09871161  3.76137522]
```

In [22]:
```
# updated with 100 epochs

gdr = GDRegressor(epochs=500,learning_rate=0.1)
```

In [23]:
```
gdr.fit(X,y) # intercept_value =151.88331005254167 , Ans =  152.1334841628959
```

```
152.1334841628959 [ 40.59429267  -5.10446851 162.86044476 117.8892874   38.79
369321
  25.3524053  -99.29060361 100.09330525 149.80131472  92.71151555]
```

In [24]:
```
gdr.predict(X_test)
```

Out[24]:
```
array([153.31701129, 174.61218107, 148.82592465, 127.44854064,
       201.05333439, 198.25965917, 121.34892119, 128.73164707,
       112.6201254 , 172.36370332, 164.28009797, 163.92818593,
       174.55705903, 158.67892549, 200.02378386, 117.00289328,
       167.81284322, 140.14737073, 147.85364095, 147.00219877,
       132.63994975, 186.20256534, 168.5929866 , 169.43193509,
       135.29535601, 196.19989091, 175.73738568, 142.03349115,
       100.46017644, 211.26714033, 204.6436887 , 134.67395845,
       108.17139846, 132.03570826, 180.26857967, 163.78458539,
       163.23375239, 181.68536498, 129.87253809, 200.39596902,
       139.50060181, 137.99146944, 173.977309  , 173.97529495,
       160.25017595, 150.78501566, 166.14776514, 222.40627618,
       134.92705721, 184.25548262, 196.50937528, 133.79080504,
       144.06236112, 162.31869144, 173.66172516, 135.59903283,
       165.90249693, 115.73172587, 156.42444493, 142.66995565,
       158.55247474, 168.35628785, 129.94539319, 176.0309458 ,
       160.36868879, 162.69586285, 146.09262572, 174.07773286,
       131.89912732, 154.04631726, 176.26764273, 168.94927365,
       131.24993503, 148.61462826, 142.60178837, 130.98591572,
       114.84545818, 117.89337692, 164.6234704 , 117.45293629,
       121.80332622, 129.87424921, 155.21188245, 212.60812446,
       179.74186298, 143.10921407, 200.14869765, 165.25427616,
       134.12329673])
```

In [25]:
```
y_pred = gdr.predict(X_test)
```

In [26]:
```
r2_score(y_test,y_pred)
```

Out[26]:
```
0.3214969209437194
```

In [ ]: