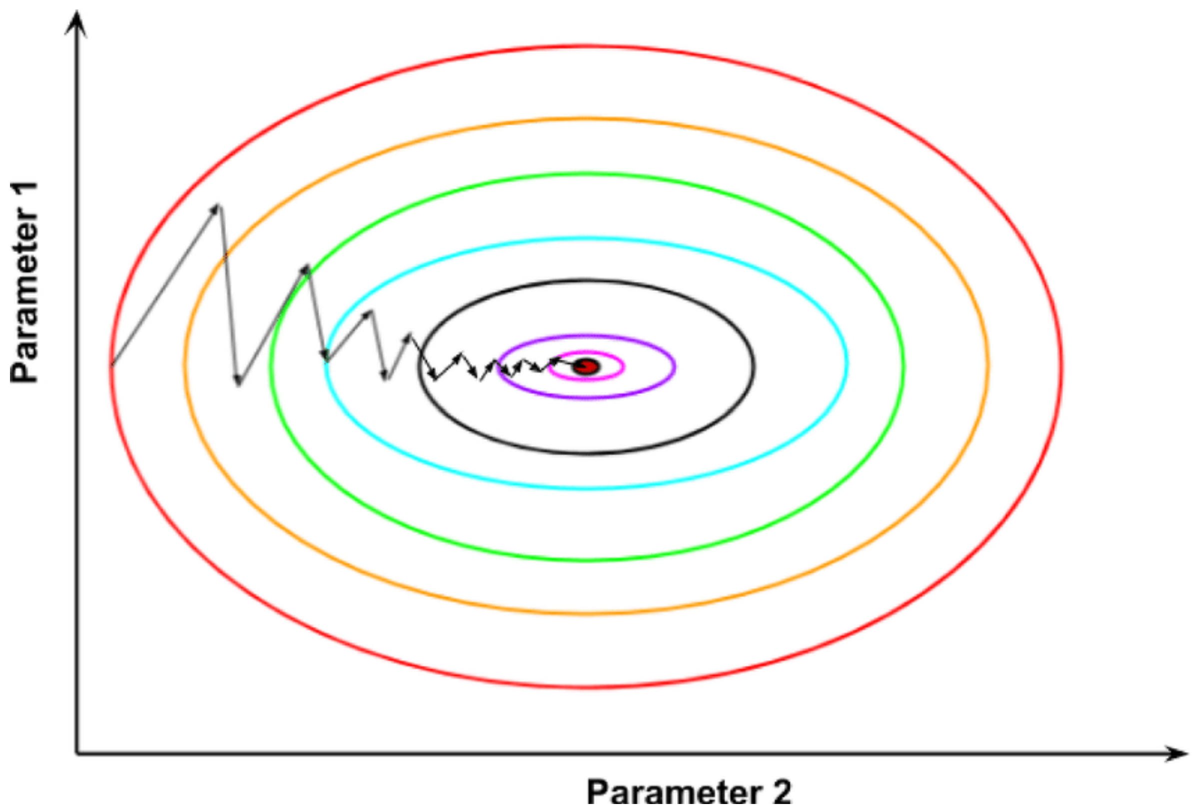
Mini-Batch Gradient Descent

Mini-Batch Gradient Descent is a variation of the gradient descent optimization algorithm that updates the model parameters using a **mini-batch of samples instead of the entire dataset (as in batch gradient descent)** or a **single sample (as in stochastic gradient descent)**. It strikes a balance between the efficiency of stochastic gradient descent and the stability of batch gradient descent.



Mini-batch gradient descent is a variation of gradient descent that divides the training dataset 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**.

Mini-batch gradient descent is a compromise between batch gradient descent and stochastic gradient descent.

- Batch gradient descent uses the entire training dataset to calculate the gradient, which can be computationally expensive for large datasets.
- Stochastic gradient descent uses a single training example to calculate the gradient, which can be less accurate. Mini-batch gradient descent falls somewhere in between, using a small subset of the training dataset to calculate the gradient.

Here are some of the benefits of using mini-batch gradient descent:

- **Reduced noise:** The gradient of the cost function is less noisy than in stochastic gradient descent, which can help the algorithm converge more quickly.

- **Computational efficiency:** Mini-batch gradient descent is more computationally efficient than batch gradient descent, especially for large datasets.

- **Better generalization:** Mini-batch gradient descent can sometimes generalize better to unseen data than stochastic gradient descent.

Here are some of the drawbacks of using mini-batch gradient descent:

- **May not converge as quickly as stochastic gradient descent:** If the batch size is too large, the gradient of the cost function may still be noisy.
- **May be more sensitive to the choice of learning rate:** The learning rate must be tuned carefully to avoid overfitting or underfitting.

Overall, mini-batch gradient descent is a good compromise between batch gradient descent and stochastic gradient descent. It is more computationally efficient than batch gradient descent, and it can sometimes generalize better to unseen data than stochastic gradient descent. However, the choice of batch size and learning rate can be important for achieving good results.

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

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

Step by step Process

Here's an overview of the steps involved in Mini-Batch Gradient Descent:

1. **Initialize Parameters**: Initialize the model parameters, such as weights and biases, with random values or predetermined values.

2. **Specify Hyperparameters**: Define hyperparameters, including the learning rate, number of epochs, batch size, and any regularization parameters.

3. **Partition the Dataset**: Divide the training dataset into mini-batches of size `batch_size`. The number of mini-batches will be `total_samples / batch_size`.

4. **Training Loop**: Iterate over the mini-batches for the specified number of epochs. In each epoch, perform the following steps:

   a. **Forward Propagation**: Pass a mini-batch through the model to compute the predictions and calculate the loss.

   b. **Backward Propagation**: Compute the gradients of the loss with respect to the model parameters using backpropagation.

c. **Parameter Update**: Update the model parameters using the computed gradients and the learning rate. The update step typically follows the equation: `parameter = parameter - learning_rate * gradient` , where `parameter` represents a model parameter (e.g., weight or bias) and `gradient` is the corresponding gradient.

5. **Repeat**: Repeat steps 4a-4c for the specified number of epochs or until convergence criteria are met.

By updating the model parameters based on mini-batches of samples, Mini-Batch Gradient Descent achieves a balance between the stability of batch gradient descent and the computational efficiency of stochastic gradient descent. It can provide better convergence speed and generalization performance compared to both batch and stochastic gradient descent.

You can customize the batch size according to your computational resources and dataset characteristics. Larger batch sizes provide more stable updates but require more memory and computational power, while smaller batch sizes introduce more noise in the parameter updates but can converge faster. It's a hyperparameter that needs to be tuned based on your specific problem.

In [1]:
```
# 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 [2]:
```
from sklearn.datasets import load_diabetes

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

In [3]:
```
print(X.shape)
print(y.shape)
```

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

In [4]:
```
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_stat
```

In [5]:
```
# 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[5]:
```
▾ LinearRegression
LinearRegression()
```

In [6]:
```
# 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 [7]:
```
# Predict using the regression model
y_pred = reg.predict(X_test)

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

Out[7]:
```
0.4399338661568968
```

In [8]:
```
X_train.shape
```

Out[8]:
```
(353, 10)
```

In [9]:
```
import random
import numpy as np

class MBGDRegressor:
    """
    Mini-Batch Gradient Descent Regressor.
    """

    def __init__(self, batch_size, learning_rate=0.01, epochs=100):
        """
        Initialize the regressor.

        Args:
            batch_size (int): Size of each mini-batch.
            learning_rate (float): Learning rate for gradient descent. Default
            epochs (int): Number of epochs for training. Default is 100.
        """
        self.coef_ = None
        self.intercept_ = None
        self.lr = learning_rate
        self.epochs = epochs
        self.batch_size = batch_size

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

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

        for i in range(self.epochs):
            for j in range(int(X_train.shape[0] / self.batch_size)):
                idx = random.sample(range(X_train.shape[0]), self.batch_size)

                y_hat = np.dot(X_train[idx], self.coef_) + self.intercept_
                intercept_der = -2 * np.mean(y_train[idx] - y_hat)
                self.intercept_ = self.intercept_ - (self.lr * intercept_der)

                coef_der = -2 * np.dot((y_train[idx] - y_hat), X_train[idx])
                self.coef_ = self.coef_ - (self.lr * coef_der)

        print(self.intercept_, self.coef_)

    def predict(self, X_test):
        """
        Predict the target values for the test data.

        Args:
            X_test (numpy.ndarray): Input features of the test data.

        Returns:
            numpy.ndarray: Predicted target values.
```

```
        """
        return np.dot(X_test, self.coef_) + self.intercept_
```

Explanation

The code defines a class called MBGDRegressor, which implements a mini-batch gradient descent regressor. The regressor is designed to perform linear regression on a given dataset using mini-batches of samples.

The MBGDRegressor class has the following attributes:

1. **coef_**: Represents the coefficients of the linear regression model.
2. **intercept_**: Represents the intercept of the linear regression model.
3. **lr**: Represents the learning rate for gradient descent.
4. **epochs**: Represents the number of epochs (iterations) for training.
5. **batch_size**: Represents the size of each mini-batch.

The class has the following methods:

- **init(self, batch_size, learning_rate=0.01, epochs=100)**: This is the constructor method that initializes the attributes of the MBGDRegressor object. It takes the batch size, learning rate, and number of epochs as arguments and assigns them to the corresponding attributes.

- **fit(self, X_train, y_train)**: This method fits the regressor to the training data. It takes the input features X_train and target values y_train as arguments. Inside the method, the coefficients and intercept are initialized. Then, for each epoch, mini-batches of samples are randomly selected from the training data. The predicted values are computed using the current coefficients and intercept. The derivatives of the intercept and coefficients are calculated based on the mini-batch and used to update the intercept and coefficients using the gradient descent update rule.

- **predict(self, X_test)**: This method predicts the target values for the test data. It takes the input features X_test as an argument. The predicted values are computed using the learned coefficients and intercept, and returned as an array.

In [10]:
```
mbr = MBGDRegressor(batch_size=int(X_train.shape[0]/50),learning_rate=0.01,epc
```

In [11]:
```
mbr.fit(X_train,y_train)
```

```
153.35711553151856 [  24.17908944 -147.04876678  451.77312141  304.76151116
-18.16248615
  -85.26718225 -192.99705711  119.0185062   410.11740003  108.68184731]
```

In [12]:
```
mbr.predict(X_test)
```

Out[12]:
```
array([153.95709106, 198.61469128, 132.30480308, 106.79781886,
       260.79121167, 249.93931487, 111.30139208, 116.24711583,
        95.05583758, 188.8338829 , 151.17721305, 173.82786417,
       182.84417036, 142.83310199, 282.66069211,  88.89733394,
       199.37920096, 147.28470231, 135.548083  , 132.81687419,
       143.99988233, 180.25115874, 158.26649547, 176.93346867,
       128.03741994, 224.7374577 , 200.07187742, 109.25629618,
        57.65420251, 242.30906679, 245.66883389, 116.23574723,
        71.12132108, 101.25187241, 204.50108997, 167.19814895,
       164.113367  , 195.29868418, 114.81379816, 239.58749739,
       140.34690611, 122.81744962, 189.57558016, 188.58996215,
       174.35610668, 145.6715616 , 171.68621429, 295.58363854,
       109.98218973, 177.96424581, 250.58978497, 140.16438131,
       150.97646661, 133.08302395, 192.18381556, 101.36041194,
       139.72606278,  79.83517116, 160.10710955, 154.10473272,
       164.40725574, 166.01460652, 105.84694802, 222.95979643,
       151.52513461, 139.29518652, 158.08579744, 194.01392509,
       123.42738715, 133.36996047, 214.81006688, 197.57078543,
       123.14754195, 153.78402285, 145.35706414, 114.48387701,
        83.30322614,  81.59347922, 171.25753228,  84.29349806,
        99.62404002, 102.52274788, 176.49146225, 273.52470786,
       206.44378891, 146.27527718, 275.29739681, 198.88088073,
       103.79902042])
```

In [13]:
```
y_pred = mbr.predict(X_test)
```

In [14]:
```
r2_score(y_test,y_pred)
```

Out[14]:
```
0.45427609715442907
```

Sklearn

In [15]:
```
from sklearn.linear_model import SGDRegressor
```

In [16]:
```
sgd = SGDRegressor(learning_rate='constant',eta0=0.1)
```

In [17]:
```
batch_size = 35

for i in range(100):

    idx = random.sample(range(X_train.shape[0]),batch_size)
    sgd.partial_fit(X_train[idx],y_train[idx])
```

In [18]:
```
sgd.coef_
```

Out[18]:
```
array([  53.19666751,  -75.41438544,  358.40379846,  246.11441738,
          7.8070073 ,  -36.54168457, -181.34352289,  124.1249008 ,
        324.31309961,  124.2752621 ])
```

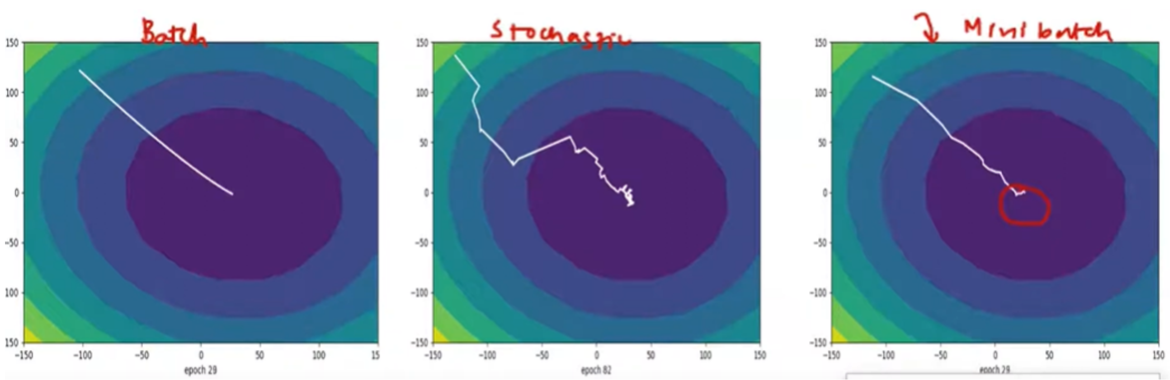
In [19]:
```
sgd.intercept_
```

Out[19]:
```
array([143.18856217])
```

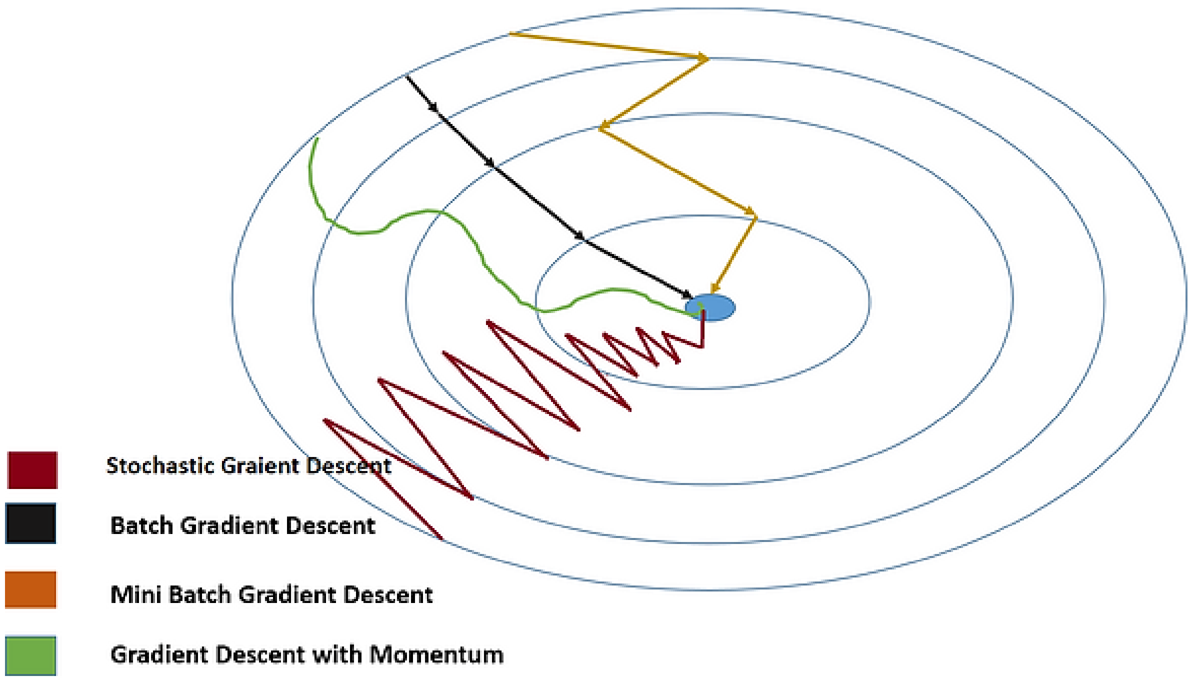
In [20]:
```
y_pred = sgd.predict(X_test)
```

In [21]:
```
r2_score(y_test,y_pred)
```

Out[21]:
```
0.422946154899145
```



Differences between batch gradient, stochastic gradient, and mini-batch gradient descent



here are the differences between batch gradient, stochastic gradient, and mini-batch gradient descent:

Gradient Descent Type	How it works	Advantages	Disadvantages
Batch gradient descent	Uses the entire training dataset to calculate the gradient of the cost function at each iteration.	- More accurate than stochastic gradient descent. - Less likely to get stuck in local minima. - Can be more efficient if the dataset is small.	- Can be slow to converge for large datasets. - Can be sensitive to outliers.
Stochastic gradient descent	Uses a single training example to calculate the gradient of the cost function at each iteration.	- More efficient than batch gradient descent for large datasets. - Less likely to get stuck in local minima.	- Less accurate than batch gradient descent. - More likely to oscillate around the minimum.
Mini-batch gradient descent	Uses a small subset of the training dataset to calculate the gradient of the cost function at each iteration.	- Combines the advantages of batch gradient descent and stochastic gradient descent. - More accurate than stochastic gradient descent. - Less likely to get stuck in local minima. - More efficient than batch gradient descent for large datasets.	- Can be more sensitive to the choice of batch size. - May not converge as quickly as stochastic gradient descent.

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

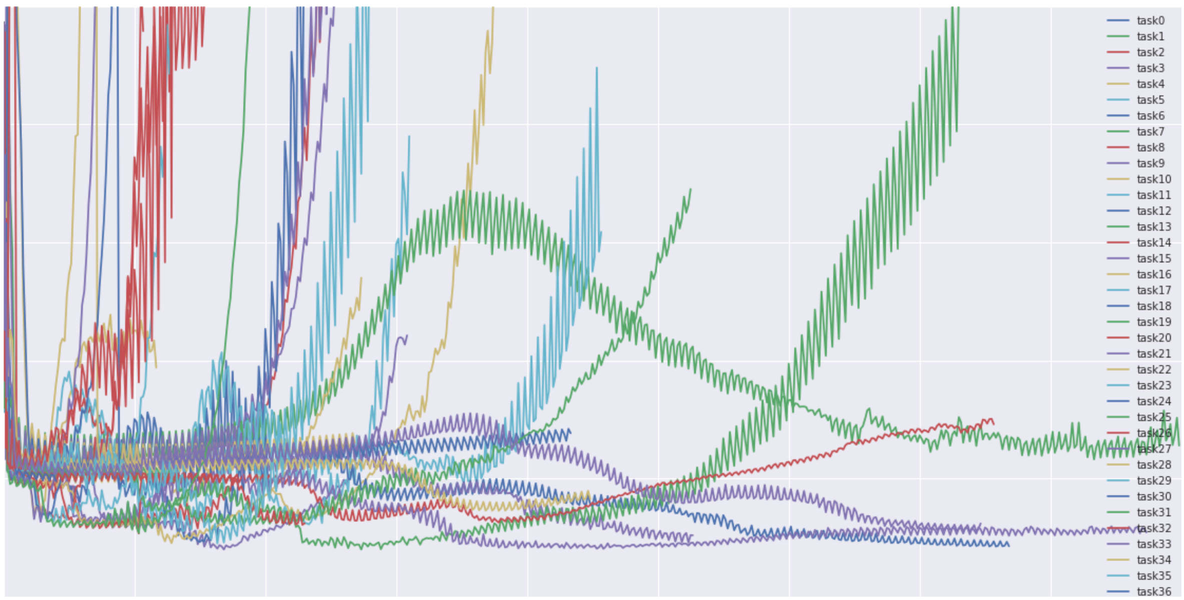
- **Batch gradient descent:** This could be used to train a linear regression model on a small dataset.

- **Stochastic gradient descent:** This could be used to train a neural network on a large dataset.

- **Mini-batch gradient descent:** This could be used to train a deep learning model on a very large dataset.

Ultimately, the best type of gradient descent to use depends on the specific problem that you are trying to solve. If you are not sure which type to use, you can try different approaches and see what works best for you.

Detailed Explanation



Here is more detailed information about the differences between batch gradient, stochastic gradient, and mini-batch gradient descent:

Batch gradient descent

- **How it works:** Batch gradient descent uses the entire training dataset 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.
- **Advantages:**
  - More accurate than stochastic gradient descent. This is because the gradient is calculated using the entire training dataset, which gives a more accurate estimate of the direction of the steepest descent.
  - Less likely to get stuck in local minima. This is because the gradient is calculated using the entire training dataset, which helps to prevent the algorithm from getting stuck in a local minimum.
  - Can be more efficient if the dataset is small. This is because the gradient only needs to be calculated once per epoch, which can be faster than calculating the gradient for each individual training example.
- **Disadvantages:**

  - Can be slow to converge for large datasets. This is because the gradient needs to be calculated for the entire training dataset, which can be computationally expensive for large datasets.
  - Can be sensitive to outliers. This is because the gradient is calculated using the entire training dataset, which means that outliers can have a significant impact on the

Stochastic gradient descent

- **How it works:** Stochastic gradient descent uses a single training example to calculate the gradient of the cost function at each iteration. This means that the gradient is calculated much more frequently than with batch gradient descent, which can help the algorithm to converge more quickly.
- **Advantages:**
  - More efficient than batch gradient descent for large datasets. This is because the gradient only needs to be calculated for a single training example, which can be much faster than calculating the gradient for the entire training dataset.
  - Less likely to get stuck in local minima. This is because the gradient is calculated for a single training example, which helps to prevent the algorithm from getting stuck in a local minimum.
- **Disadvantages:**
  - Less accurate than batch gradient descent. This is because the gradient is calculated using a single training example, which gives a less accurate estimate of the direction of the steepest descent.
  - More likely to oscillate around the minimum. This is because the gradient is calculated for a single training example, which can cause the algorithm to oscillate around the minimum.

Mini-batch gradient descent

- **How it works:** Mini-batch gradient descent uses a small subset of the training dataset to calculate the gradient of the cost function at each iteration. This is a compromise between batch gradient descent and stochastic gradient descent, and it can often achieve the best of both worlds.
- **Advantages:**
  - Combines the advantages of batch gradient descent and stochastic gradient descent. This means that it is more accurate than stochastic gradient descent, and it is less likely to get stuck in local minima than batch gradient descent.
  - More efficient than batch gradient descent for large datasets. This is because the gradient only needs to be calculated for a small subset of the training dataset, which can be much faster than calculating the gradient for the entire training dataset.
- **Disadvantages:**
  - Can be more sensitive to the choice of batch size. This is because the gradient is calculated using a small subset of the training dataset, which means that the batch size can have a significant impact on the performance of the algorithm.
  - May not converge as quickly as stochastic gradient descent. This is because the gradient is calculated using a small subset of the training dataset, which can slow down the convergence of the algorithm.

Ultimately, the best type of gradient descent to use depends on the specific problem that you are trying to solve. If you are not sure which type to use, you can try different approaches and