AI and Machine Learning Homework1 Report

Student: 纪浩正 / 12311405

jihz2023@mail.sustech.edu.cn

Linear Regression

Gradient

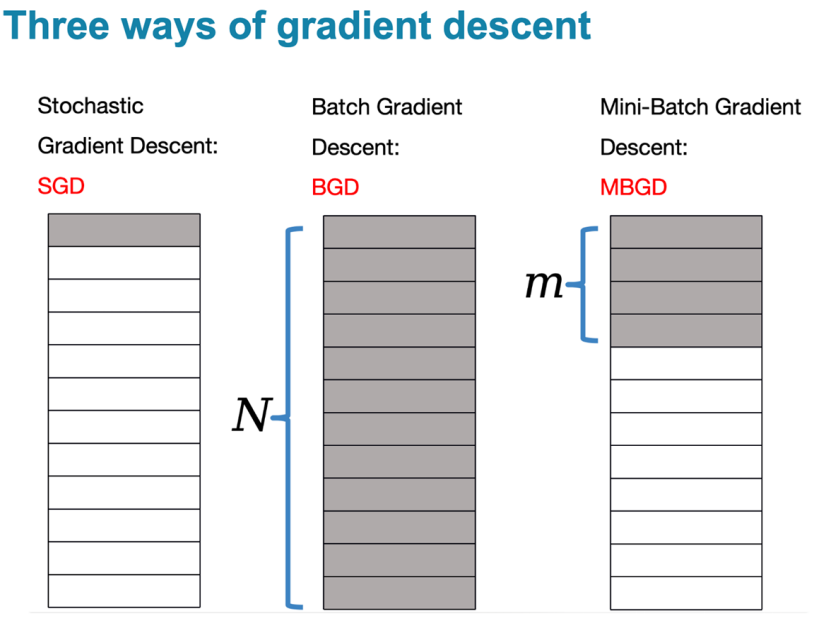
In linear regression, we aim to find the optimal parameters **w** that minimize the mean squared error (MSE) loss function. Three gradient descent variants are implemented:

The linear model with bias term is:

$$y = w_0 + \mathbf{x}^T\mathbf{w} = [\mathbf{1}, \mathbf{x}]^T[\mathbf{w}_0, \mathbf{w}]^T$$

The MSE loss function is:

$$L(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\mathbf{w})^2$$



- **Batch Gradient Descent (BGD):**

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = -\frac{1}{n}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\mathbf{w})$$

- **Stochastic Gradient Descent (SGD):**

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = -(y_i - \mathbf{x}_i^T\mathbf{w})\mathbf{x}_i$$

- **Mini-batch Gradient Descent (MBGD):**

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = -\frac{1}{|B|} \sum_{i \in B} (y_i - \mathbf{x}_i^T \mathbf{w}) \mathbf{x}_i$$

- **Gradient Normalization:**

$$\mathbf{g}_{norm} = \frac{\mathbf{g}}{||\mathbf{g}||_2}$$

Normalization

Two normalization methods are implemented:

- **Min-Max Normalization**

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

- **Standardization (Mean Normalization)**

$$x_{norm} = \frac{x - \mu}{\sigma}$$

where μ and σ are the mean and standard deviation respectively.

Data Fitting

The fitting process follows these steps:

1. Normalize input features $\mathbf{X}$ using the specified method
2. Add bias column: $\mathbf{X}_{augmented} = [\mathbf{1}, \mathbf{X}_{norm}]$
3. Initialize weights $\mathbf{w} = \mathbf{0}$
4. For each epoch, apply the selected gradient descent method
5. Optionally apply gradient normalization: $\mathbf{dw} = \frac{\mathbf{dw}}{||\mathbf{dw}||}$
6. Update weights: $\mathbf{w} = \mathbf{w} - lr \cdot \mathbf{dw}$

Data Prediction

For prediction:

1. Apply the same normalization parameters from training to test data
2. Add bias column to normalized test data

3. Compute predictions: $\hat{\mathbf{y}} = \mathbf{X}_{test,augmented}\mathbf{w}$

Dataset Description

Training Set:

- We fixed a random seed to ensure reproducibility.
- The input feature values are integers from 0 to 99:

$$X_{\text{train}} = \{0, 1, 2, \ldots, 99\}$$

- The underlying ground truth function is linear:

$$y = ax + b, \quad a = 1,\ b = 10$$

- To simulate measurement noise in real-world data, Gaussian noise $\mathcal{N}(0, 5^2)$ was added to each sample:

$$y_{\text{train}} = ax + b + \epsilon, \quad \epsilon \sim \mathcal{N}(0, 25)$$

Thus, the training set consists of noisy linear data.

Test Set:

- The test set uses the same input range:

$$X_{\text{test}} = \{0, 1, 2, \ldots, 99\}$$

- However, the labels are generated **without noise**, directly from the true function:

$$y_{\text{true}} = ax + b = x + 10$$

- The trained model then outputs predictions:

$$y_{\text{pred}} = f_{\text{model}}(X_{\text{test}})$$

Summary:

- The **training set** is noisy linear data, used to fit the regression model.
- The **test set** is exact and noise-free, generated from the true function to evaluate model accuracy.

- This design allows assessment of whether the model can recover the underlying linear relation despite noise in the training data.

Code Implementation

Calculate Error

```
def mean_squared_error(self, true, pred):
    squared_error = np.square(true - pred)
    sum_squared_error = np.sum(squared_error)
    mse_loss = sum_squared_error / true.size
    return mse_loss
    ```

Normalize

```python
def normalize(self, X, norm_mode=None):
    self.norm_mode = norm_mode
    X_norm = X.copy()
    if norm_mode == "minmax":
        self.x_min = X.min(axis=0)
        self.x_max = X.max(axis=0)
        X_norm = (X - self.x_min) / (self.x_max - self.x_min)
    elif norm_mode == "mean":
        self.x_mean = X.mean(axis=0)
        self.x_std = X.std(axis=0)
        X_norm = (X - self.x_mean) / self.x_std
    elif norm_mode == "none":
        pass
    else:
        raise ValueError("norm_mode must be 'minmax', 'mean' or None")
    return X_norm
```

Normalize for prediction

```
def _normalize(self, X, norm_mode=None):
    if self.norm_mode == "minmax":
        return (X - self.x_min) / (self.x_max - self.x_min)
    elif self.norm_mode == "mean":
        return (X - self.x_mean) / self.x_std
    else:
        return X
```

Fit

```

def fit(self, X_train, y_train, method="bgd", epochs=100, lr=0.01,
        norm_mode=None, batch_size=32, grad_norm=None):
    self.nsample, self.ndim = X_train.shape
```

```
    X_norm = self.normalize(X_train, norm_mode)
    b = np.ones((X_norm.shape[0], 1))
    X_norm = np.hstack([b, X_norm])
    self.w = np.zeros(((self.ndim + 1)))
    losses = []
    for epoch in range(epochs):
        if method == "bgd":
            y_pred = X_norm @ self.w
            dw = -X_norm.T @ (y_train - y_pred) / self.nsample
            if grad_norm == 1:
                dw = dw / np.sqrt(np.sum(np.square(dw)))
            self.w = self.w - lr * dw
            loss = self.mean_squared_error(y_train, y_pred)
            losses.append(loss)
        elif method == "sgd":
            idx = np.arange(self.nsample)
            np.random.shuffle(idx)
            for ii in idx:
                xi = X_norm[ii]
                yi = y_train[ii]
                y_pred = xi @ self.w
                dw = -(yi - y_pred) * xi
                if grad_norm == 1:
                    dw = dw / np.sqrt(np.sum(np.square(dw)))
                self.w = self.w - lr * dw
            y_pred = X_norm @ self.w
            loss = self.mean_squared_error(y_train, y_pred)
            losses.append(loss)
        elif method == "mbgd":
            idx = np.arange(self.nsample)
            np.random.shuffle(idx)
            for s_idx in range(0, self.nsample, batch_size):
                e_idx = s_idx + batch_size
                batch_idx = idx[s_idx:e_idx]
                X_batch = X_norm[batch_idx]
                y_batch = y_train[batch_idx]
                y_pred = X_batch @ self.w
                dw = -X_batch.T @ (y_batch - y_pred) / X_batch.shape[0]
                if grad_norm == 1:
                    dw = dw / np.sqrt(np.sum(np.square(dw)))
                self.w = self.w - lr * dw
            y_pred = X_norm @ self.w
            loss = self.mean_squared_error(y_train, y_pred)
            losses.append(loss)
        else:
            raise ValueError("Gradient descent must be sgd, bgd, mbgd")
    return losses
```

Predict

```
def predict(self, X_pred):
    X_norm = self._normalize(X_pred, norm_mode=None)
```

```
    b = np.ones((X_norm.shape[0], 1))
    X_norm = np.hstack((b, X_norm))
    return X_norm @ self.w
```

My Mistakes in the Past

During the implementation process, I encountered several critical issues that affected the correctness of my linear regression implementation:

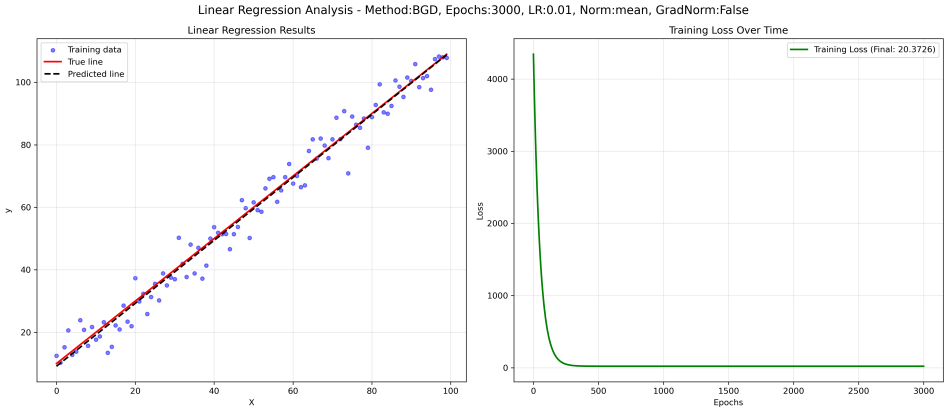
- **Training vs. Testing Set Confusion:**
  I did not originally differentiate between the **training set** and the **testing set**. I mistakenly retrained parameters on the test set. Later, I fixed this issue by generating a **separate noise-free test set** from the true function $y = ax + b$ for proper evaluation.
- **Bias Column Normalization Error:**
  Initially, I normalized the entire feature matrix **including the bias column**. This caused the bias values (which should be all ones) to be incorrectly scaled close to zero during prediction, leading to errors. The fix was to **normalize features first**, then add the bias column afterwards.
- **Inconsistent Bias Handling:**
  During training, the bias column is fixed at `1`, so during prediction I must also manually ensure the bias column remains as ones. Otherwise, the normalization parameters for the bias would not match.
- **Statistical Parameter Storage Issue:**
  In an earlier version, I mistakenly stored **mean and standard deviation for the bias column** (`x_mean[0] = 1`, `x_std[0] = 0`). When reusing these stats during prediction, the algorithm tried to normalize the bias column again, which is mathematically incorrect. This was solved by excluding the bias before normalization.

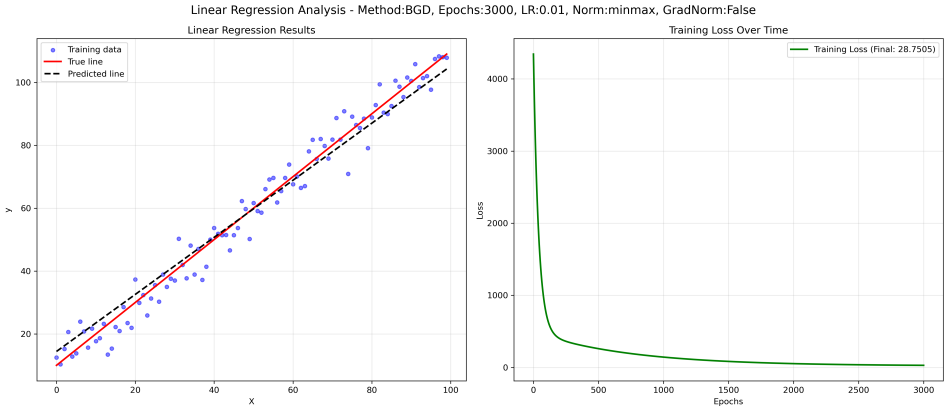
My Exploration

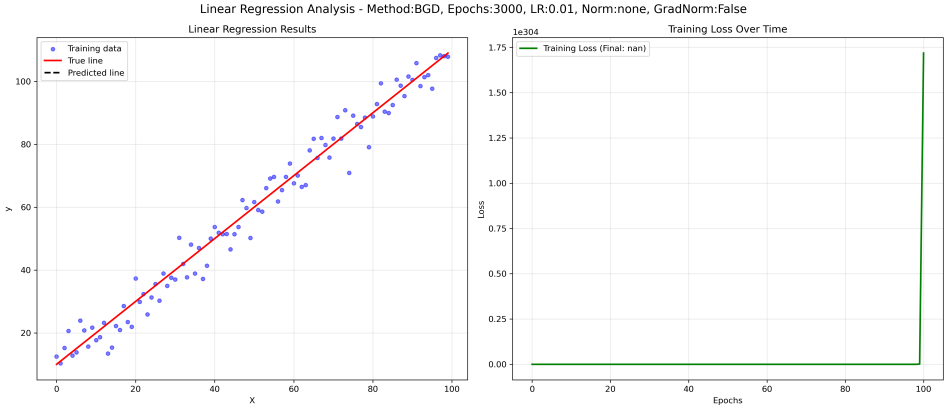
Batch Gradient Descent (BGD)

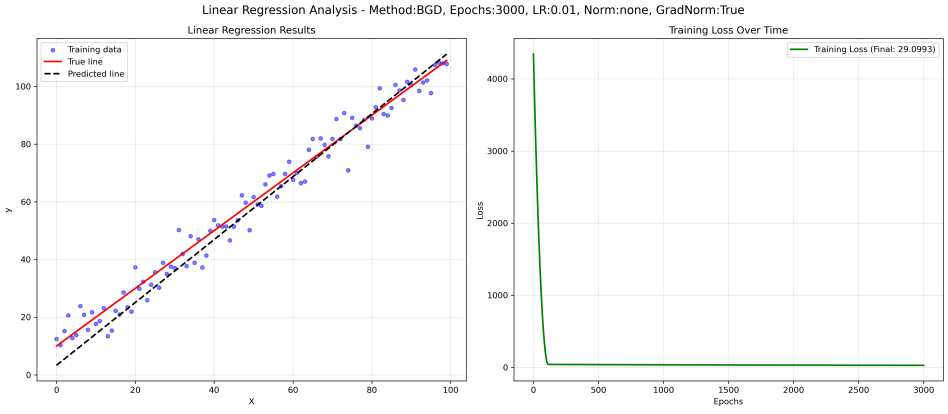
Observation:

- With **Mean or MinMax normalization**, the loss decreased smoothly and convergence was stable.
- With **no normalization**, training diverged quickly, unless I enabled gradient normalization which stabilized it.







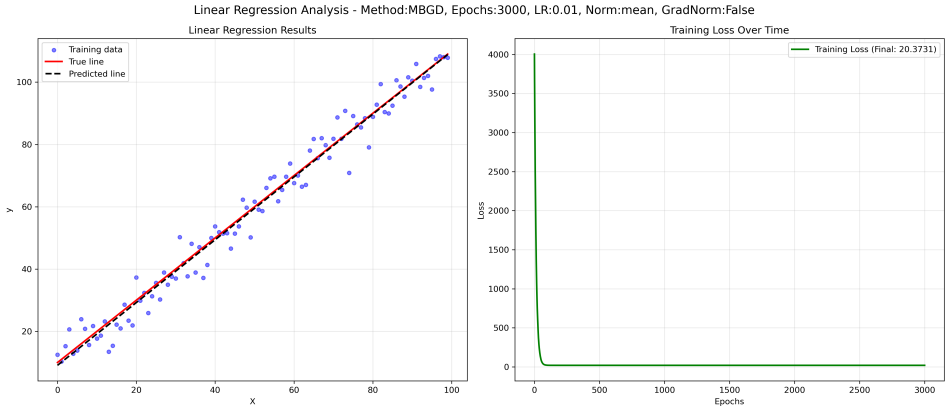


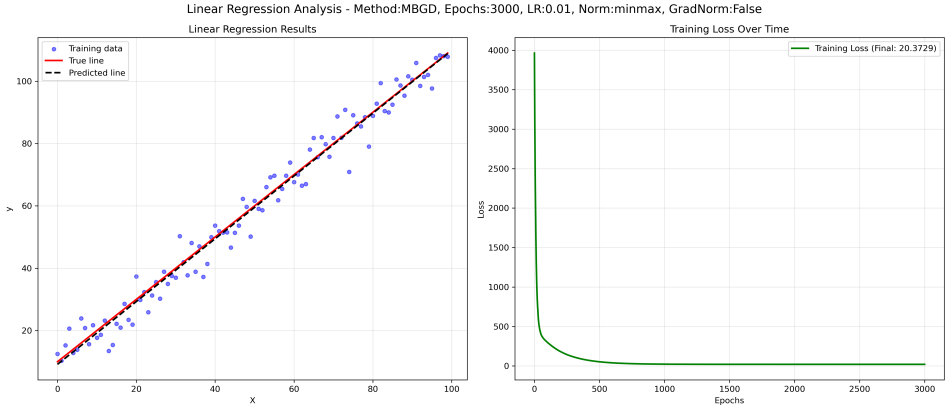
Mini-batch Gradient Descent (MBGD)

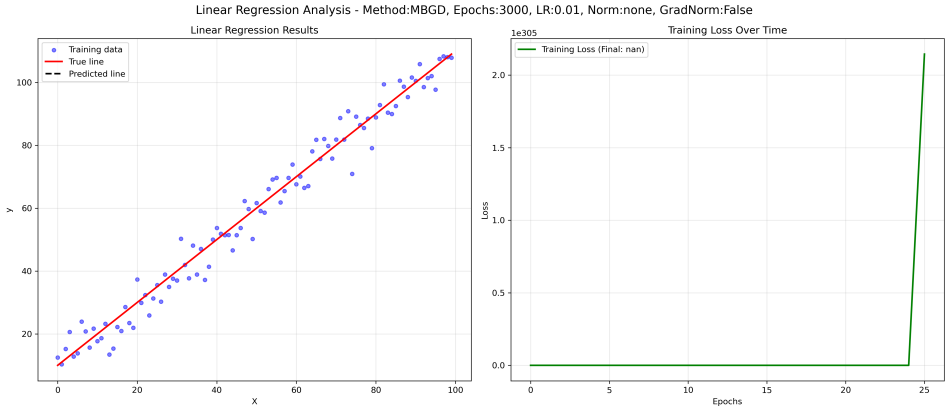
Observation:

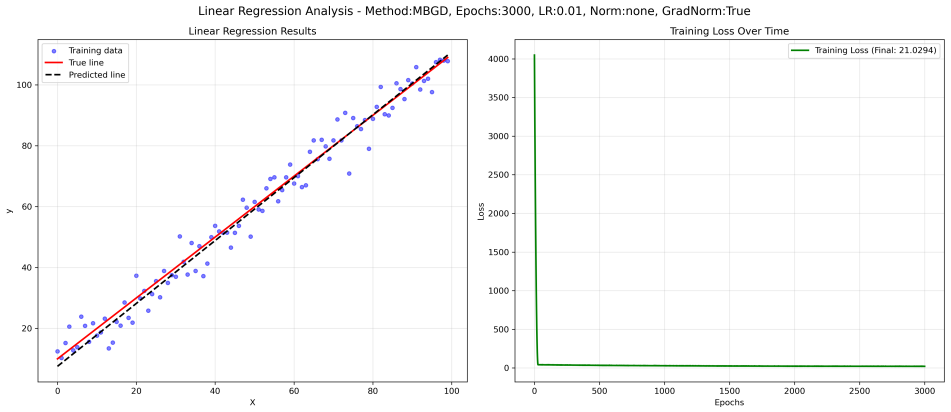
- MBGD gave smoother convergence than SGD but was faster than full BGD in per-iteration cost.
- Again, normalization methods made convergence more stable.

- When **no normalization** was used, gradient normalization (`grad_norm=1`) significantly improved stability.







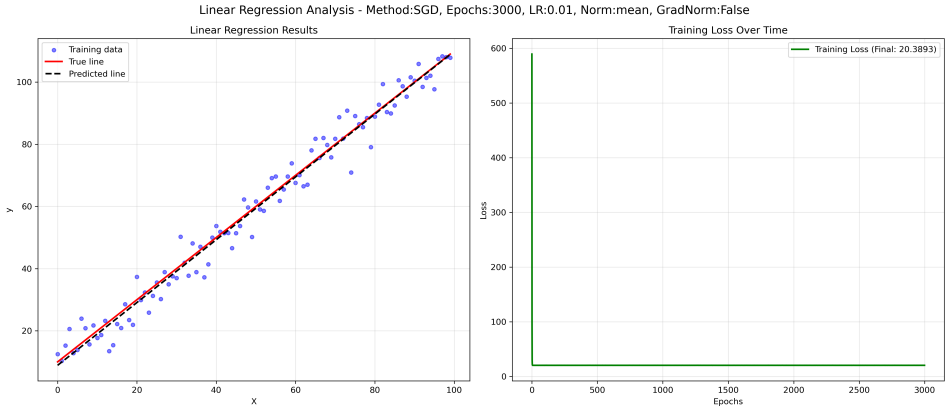


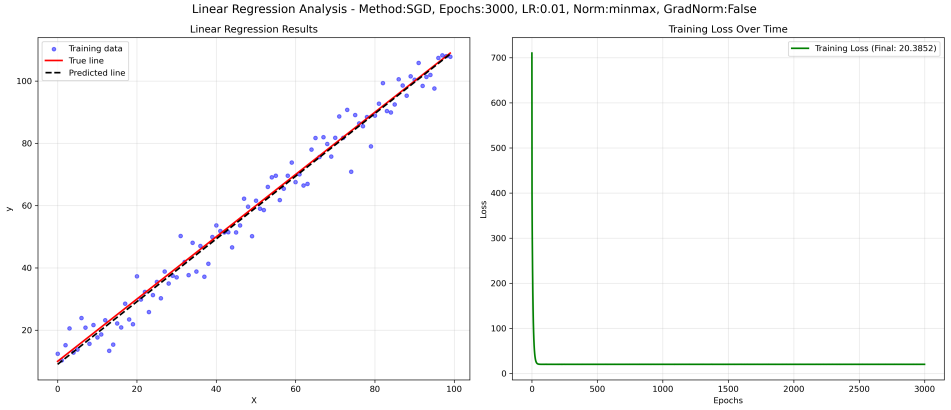
Stochastic Gradient Descent (SGD)

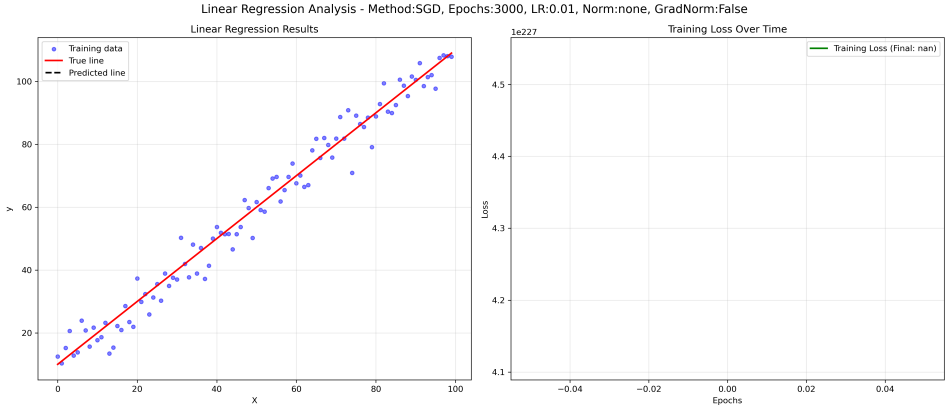
Observation:

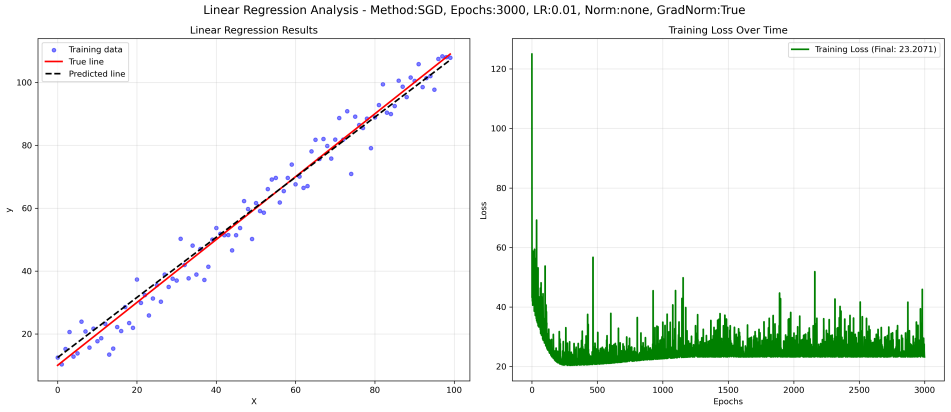
- SGD provided fast updates but the loss curve fluctuated strongly.

- Normalization reduced the noise in updates, and the final results were more stable.
- With **no normalization**, only gradient normalization could prevent divergence.









Additional Comparison: Convergence with 1200 Epochs

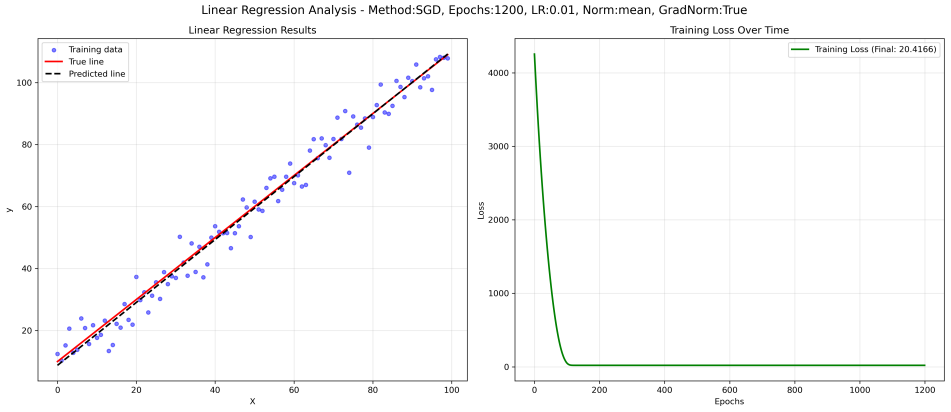
In order to compare the **convergence speed and stability** across the three gradient descent variants under the same setting (**Mean Normalization + Gradient**

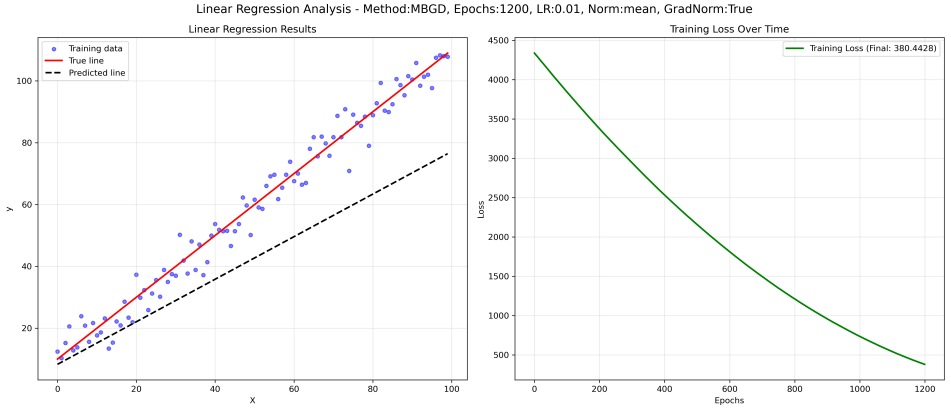
Normalization enabled), I trained each method for 1200 epochs.

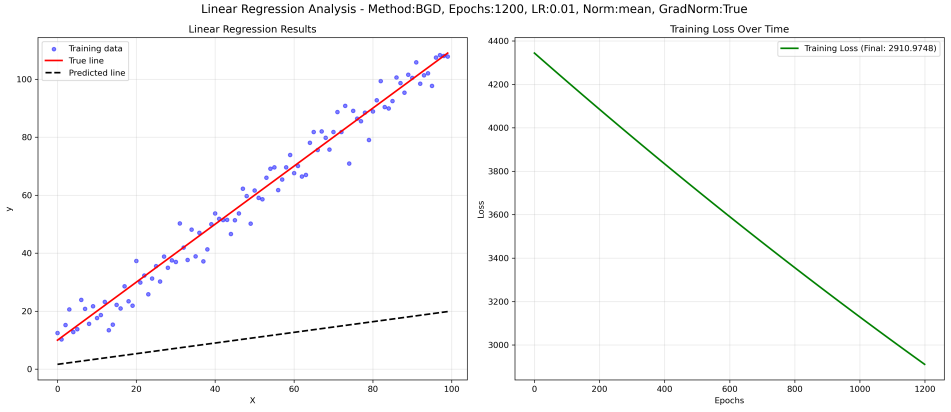
Observation:

- **SGD** shows the fastest convergence to a low loss value.
- **MBGD** balances speed and stability, converging smoothly.
- **BGD** is the worst performer: although stable, it converges the slowest and requires more epochs to reach comparable accuracy.

Hence, **SGD demonstrates the best convergence behavior**, while **BGD performs the worst** under the same settings.







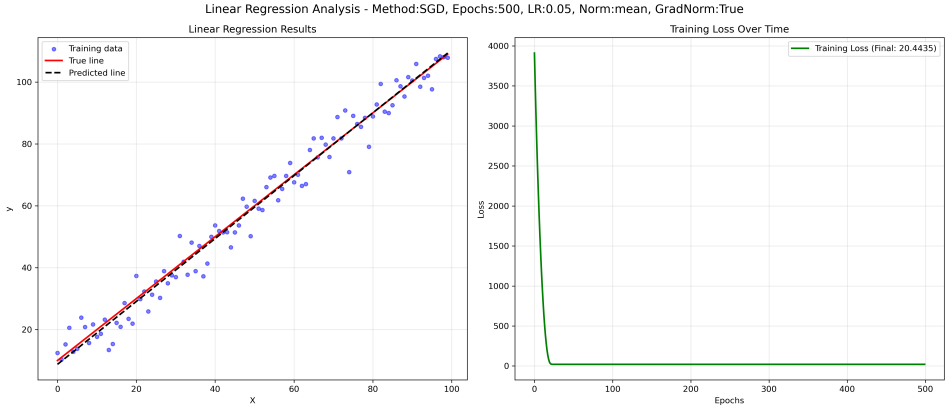
Short-term Training: Only 500 Epochs

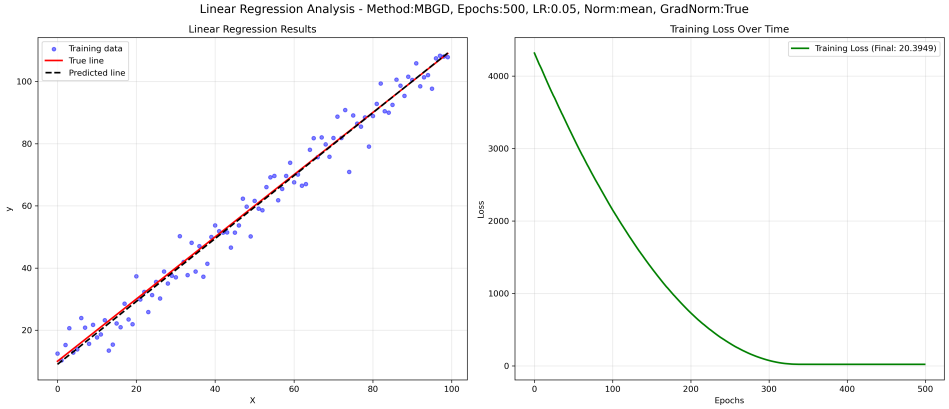
To further illustrate the performance difference in the **early training stage (500 epochs)** with learning rate = 0.05:

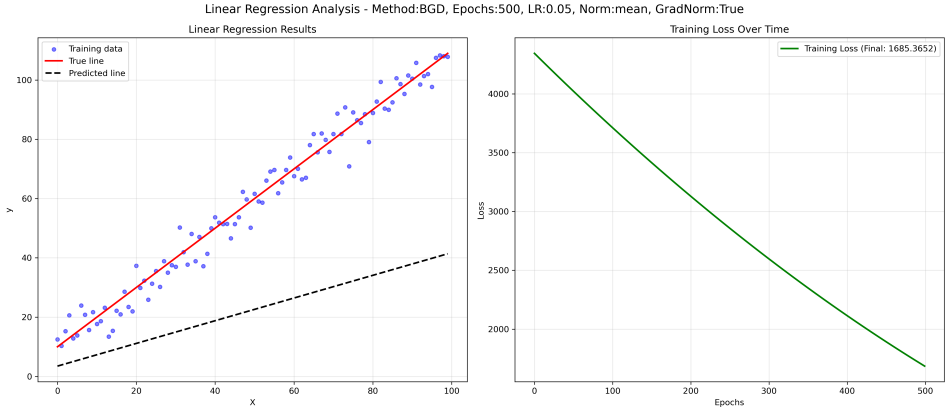
Observation:

- **SGD** already converges very close to the true line with minimal bias.
- **MBGD** also shows a relatively stable trajectory approaching the target.
- **BGD** exhibits significant **bias (shift)** after only 500 epochs, meaning it lags behind the other two algorithms in early convergence.

This confirms that **BGD not only converges slower in the long run, but also shows larger deviation in short-term training** compared to SGD and MBGD.







Summary of Exploration

1. **Normalization is necessary.** Without normalization, the loss can diverge or explode.
2. **Gradient normalization** helps especially when raw features are used without preprocessing.

3. **BGD (Batch Gradient Descent)** converges the slowest, shows larger bias in the short term, and overall performs the worst.
4. **SGD (Stochastic Gradient Descent)** updates very fast and achieves the best convergence, especially when combined with normalization and gradient normalization but introduces noise when use gradient normalization.
5. **MBGD (Mini-batch Gradient Descent)** provides the best trade-off between stability and efficiency, combining the advantages of BGD and SGD.