

AI and Machine Learning, Homework5

Author: Ying Yiwen
Number: 12210159

Contents

1	Develop a Multilayer Perceptron Model	2
2	Implement Mini-batch and Stochastic Gradient Descent Updates	4
3	Cross-Validation Implementation	5
4	Generate the Dataset	6
5	The Result of Nonlinear Dataset by MBGD Method	7
6	The Result of Nonlinear Dataset by SGD Method	8
7	The Result of Classifier Dataset by MBGD Method	9
8	The Result of Classifier Dataset by SGD Method	11
9	Appendix	13

1 Develop a Multilayer Perceptron Model

In multilayer perceptron, we use the formula (1) to do forward propagation.

$$h_j(x) = f(v_{j0} + \sum_{i=1}^D x_i v_{ji}) \quad (1)$$

```
1 def forward(self, inputs):
2     # Forward pass through the network
3     self.activations = [inputs] # Store activations of each layer
4     for i in range(self.num_layers - 1):
5         # Compute the weighted sum and apply activation function
6         z = np.dot(self.activations[i], self.weights[i]) + self.biases[i]
7         activation = self.activation(z)
8         self.activations.append(activation)
9     return self.activations[-1]
```

In multilayer perceptron, we use the formula (2) to do backward propagation.

$$W \leftarrow W - \eta \sum_{n=1}^N \frac{\partial E(o^{(n)}, t^{(n)})}{\partial W} \quad (2)$$

```
1 def backward(self, inputs, targets, learning_rate):
2     # Backpropagation process
3     output_errors = targets - self.activations[-1]
4     output_delta = output_errors * self.activation_derivative(self.activations[-1])
5
6     # Update weights and biases for the output layer
7     self.weights[-1] += self.activations[-1].T.dot(output_delta) * learning_rate
8     self.biases[-1] += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
9
10    # Calculate errors and update weights for hidden layers
11    hidden_errors = output_delta.dot(self.weights[-1].T) # initialize hidden_errors
12    with output_delta for the last layer
13    for i in range(self.num_layers - 3, -1, -1):
14        # Calculate hidden delta for the current layer
15        hidden_delta = hidden_errors * self.activation_derivative(self.activations[i +
16            1])
17
18        # Update weights and biases for the current hidden layer
19        self.weights[i] += self.activations[i].T.dot(hidden_delta) * learning_rate
20        self.biases[i] += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate
21
22        # Update hidden_errors for the next layer's calculation
23        hidden_errors = hidden_delta.dot(self.weights[i].T)
```

For the activation function, we use formula (3) to be activation function, and its derivative is formula (4).

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \quad (3)$$

$$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z)) \quad (4)$$

The sigma activation function has a limit of output from 0 to 1, so we need min-max normalization to put the target into the range, like,

$$x' = \frac{x - \min}{x - \max} \quad (5)$$

With forward propagation and backward propagation, we can estimate the weights of the layers. Then we can create a model like this:

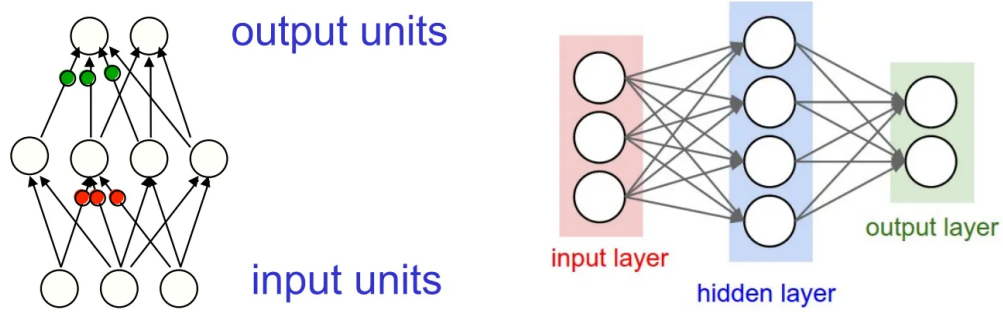


Fig. 1: Schematic of MLP

To achieve implementation of an MLP capable of handling any number of layers and units per layer, the model is able to initialize the layers by input layer number and neuron number.

```

1 def __init__(self, layer_sizes, dataset_type):
2     # layer_sizes is a list containing the number of nodes in each layer
3     self.layer_sizes = layer_sizes
4     self.num_layers = len(layer_sizes)
5
6     # Initialize weights and biases
7     self.weights = []
8     self.biases = []
9
10    # Initialize weights and biases for each layer
11    for i in range(self.num_layers - 1):
12        weight = np.random.randn(layer_sizes[i], layer_sizes[i + 1])
13        bias = np.zeros((1, layer_sizes[i + 1]))
14        self.weights.append(weight)
15        self.biases.append(bias)

```

We also achieve an auto increasing strategy to get the best choice of layers.

```

1 for layers in range(initial_layers, max_layers + 1):
2     layer_sizes = [inputs.shape[1]] + self.get_neurons_for_layers(layers) + [1]

```

The function `get_neurons_for_layers` use the sequence of predefined numbers. The numbers are set by test. The output is like: [2, 6, 14, 26, 38, 19, 13, 7, 3, 1].

There should be an early stopping judgement to reduce the running time. If the performance doesn't go well above the tolerance in the patience turns, we can stop the training.

```

1 # Early stopping mechanism
2 if average_accuracy >= best_score:

```

```

3     if average_accuracy - best_score > tolerance:
4         no_improve_count = 0 # Reset counter
5         best_score = average_accuracy
6         best_model = model
7     else:
8         no_improve_count += 1 # Increment counter if no significant improvement
9
10    # Stop if no significant improvement for consecutive 'patience' times
11    if no_improve_count >= patience:
12        print("Early stopping triggered")
13        break

```

2 Implement Mini-batch and Stochastic Gradient Descent Updates

There are several update methods, like mini-batch gradient descent, batch gradient descent and stochastic gradient descent. The principle is almost the same as those in the past assignments.

```

1 def mbgd_train(self, inputs, targets, learning_rate, epochs, batch_size=32):
2     # Train the model over specified epochs with mini-batch gradient descent
3     for epoch in range(epochs):
4         # Shuffle training data for each epoch to improve generalization
5         indices = np.arange(inputs.shape[0])
6         np.random.shuffle(indices)
7         inputs_train = inputs[indices]
8         targets_train = targets[indices]
9
10        # Process each mini-batch
11        for start in range(0, len(inputs_train), batch_size):
12            end = min(start + batch_size, len(inputs_train))
13            batch_inputs = inputs_train[start:end]
14            batch_targets = targets_train[start:end]
15
16            # Forward pass and get predictions
17            predictions = self.forward(batch_inputs)
18            # Backward pass for the mini-batch
19            self.backward(batch_inputs, batch_targets, learning_rate)
20
21            # Calculate loss for the current mini-batch
22            batch_loss = self.mean_squared_error(predictions, batch_targets)

```

```

1 def sgd_train(self, inputs, targets, learning_rate, epochs):
2     # Train the model over specified epochs using stochastic gradient descent
3     for epoch in range(epochs):
4         # Shuffle training data for each epoch to improve generalization
5         indices = np.arange(len(inputs))
6         np.random.shuffle(indices)
7         inputs_train = inputs[indices]
8         targets_train = targets[indices]
9
10        # Iterate over each sample in the training set
11        for i in range(len(inputs_train)):
12            # Forward pass for a single sample
13            predictions = self.forward(inputs_train[i:i+1])

```

```

14     # Backward pass for the same sample
15     self.backward(inputs_train[i:i+1], targets_train[i:i+1], learning_rate)
16
17     # Calculate loss for the current sample
18     batch_loss = self.mean_squared_error(predictions, targets_train[i:i+1])

```

3 Cross-Validation Implementation

In each test of layer size, we use k-fold cross-validation to test the performance of the model. According to the common sense, we choose k to be 5.

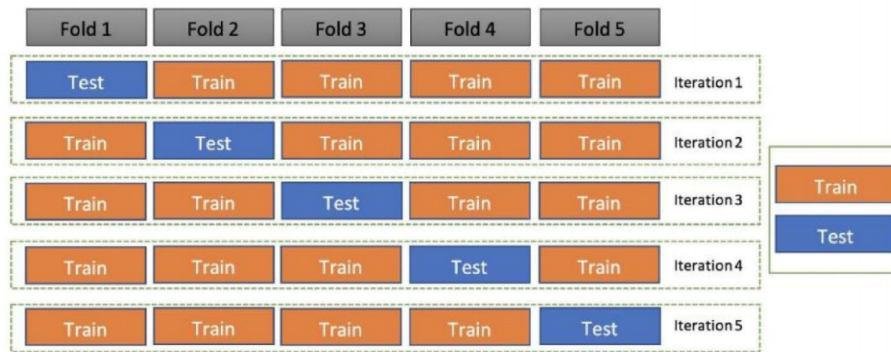


Fig. 2: Schematic of k-fold cross-validation

```

1 def k_fold_split(self, length, k):
2     indices = np.arange(length)
3     np.random.shuffle(indices) # shuffle the indices
4     folds = np.array_split(indices, k) # split the data into k folds
5     # yield the train and validation indices
6     for i in range(k):
7         train_indices = np.concatenate(folds[:i] + folds[i+1:]) # train set
8         val_indices = folds[i] # validation set
9         yield train_indices, val_indices

```

```

1 for train_indices, val_indices in self.k_fold_split(len(inputs), k):
2     model = MultiLayerPerceptron(layer_sizes, dataset_type='classify')
3
4     # Train on training set
5     model.train(inputs[train_indices], targets[train_indices], learning_rate=
6         learning_rate, epochs=epochs, method=method)
7
8     # Validate on validation set
9     accuracy, precision, recall, f1 = model.evaluate(inputs[val_indices], targets[
10         val_indices])
11     accuracy_scores.append(accuracy)
12     precision_scores.append(precision)
13     recall_scores.append(recall)
14     f1_scores.append(f1)

```

```

14 print(f"Layers: {layer_sizes}, Fold Accuracy: {accuracy:.3f}, Fold Precision: {
    precision:.3f}, Fold Recall: {recall:.3f}, Fold F1 score: {f1:.3f}")
15 model.visualize_predictions_classifier(inputs[val_indices], targets[val_indices
    ])
16
17 average_accuracy = np.mean(accuracy_scores)
18 average_precision = np.mean(precision_scores)
19 average_recall = np.mean(recall_scores)
20 average_f1 = np.mean(f1_scores)
21 print(f"In this layer test: {layer_sizes}, Average Accuracy: {average_accuracy},
    Average Precision: {average_precision}, Average Recall: {average_recall},
    Average F1 score: {average_f1}")
22 print("_____")

```

4 Generate the Dataset

To test the model, we use to dataset to train it.

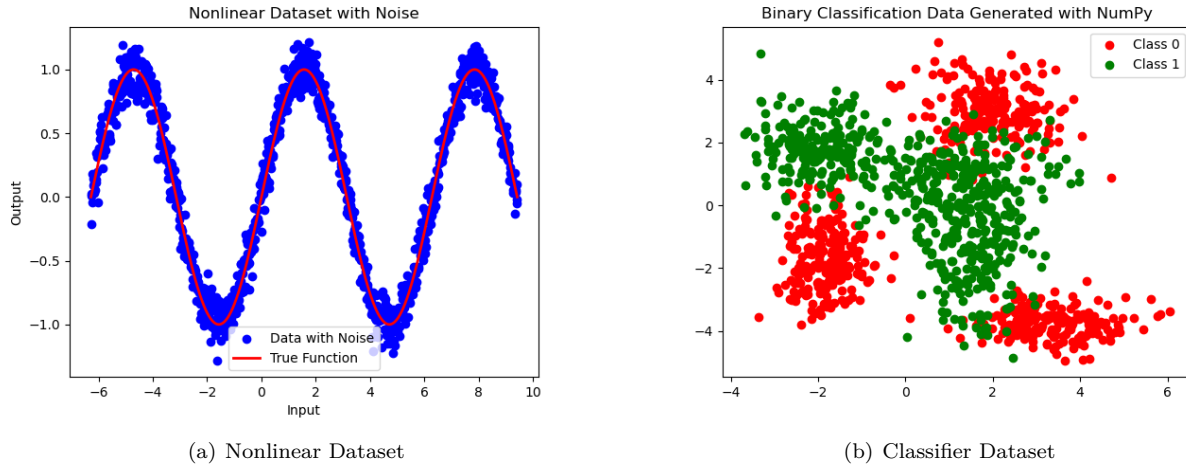


Fig. 3: Generate the Dataset

The nonlinear dataset has 1 input and 1 output to do regression. We use stuff like this:

```

1 # generate input features
2 X = np.linspace(-2 * np.pi, 3 * np.pi, n_samples).reshape(-1, 1)
3 noise_level = 0.1 # Adjust this value to control the noise level
4 # generate output labels
5 y = np.sin(X) + np.random.normal(0, noise_level, X.shape)

```

The classifier dataset has 2 input and 1 output to do classification. We use stuff like this:

```

1 # 0 class data
2 x0 = np.random.multivariate_normal(mean=[-1.8, -1.5], cov=[[0.3, 0], [0, 1.2]]) +
    np.random.normal(0, 0.1, 2)
3 X.append(x0)
4 y.append(0)
5 # 1 class data
6 x1 = np.random.multivariate_normal(mean=[1.2, 0.9], cov=[[1.3, 0], [0, 0.5]]) + np
    .random.normal(0, 0.1, 2)

```

```

7 | X.append(x1)
8 | y.append(1)

```

5 The Result of Nonlinear Dataset by MBGD Method

MBGD method takes advantage of SGD and BGD, it's quickly and effectively. As linear dataset can be easily done regression problem, nonlinear dataset can better prove the performance of our model.

I take sin function to be the dataset. Those sample points that are far from the origin require very many iterations to fit gradually. After many attempts, we chose the following parameters:

learning rate: 0.1, epochs: 10000, method: mbgd

We do the choices in the main function:

```

1 | dataset = 'nonlinear'
2 | n_samples = 1000
3 | learning_rate, epochs = 0.1, 10000
4 | method = 'mbgd'
5 | k = 5, tolerance = 1e-4, patience = 3

```

The training courses is as follows:

In this layer test: [1, 9, 1], Average MSE: 0.0036760063793220125

In this layer test: [1, 9, 6, 1], Average MSE: 0.003466735412638111

In this layer test: [1, 9, 21, 6, 1], Average MSE: 0.0019875823724081034

In this layer test: [1, 9, 21, 14, 6, 1], Average MSE: 0.00209688022635737

In this layer test: [1, 9, 21, 39, 14, 6, 1], Average MSE: 0.0019867735818585924

In this layer test: [1, 9, 21, 39, 26, 14, 6, 1], Average MSE: 0.002136578738297147

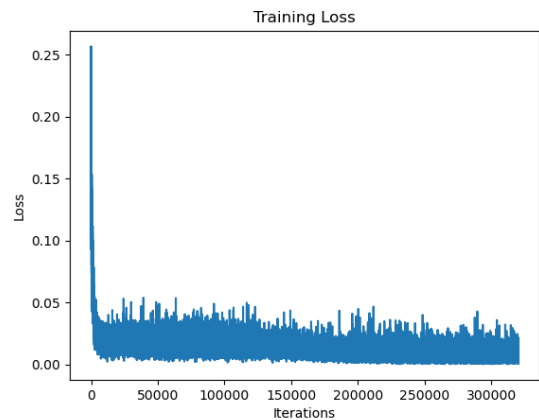
Then it triggers early stopping. Take all samples into account, Best Model - MSE: 0.38383448075092513

```

0:\study\study\2024-2025fall\AI\人工智能与机器学习\hahad\python\hah.py
Training Nonlinear, learning rate: 0.1, epochs: 10000, method: mbgd
Layers: [1, 9, 1], Fold MSE: 0.0036760063793220125
Layers: [1, 9, 1], Fold MSE: 0.0036760063793220125
Layers: [1, 9, 1], Fold MSE: 0.0036760063793220125
Layers: [1, 9, 1], Fold MSE: 0.0036760063793220125
Layers: [1, 9, 1], Fold MSE: 0.0036760063793220125
In this layer test: [1, 9, 1], Average MSE: 0.0036760063793220125
Layers: [1, 9, 6, 1], Fold MSE: 0.003466735412638111
Layers: [1, 9, 6, 1], Fold MSE: 0.003466735412638111
Layers: [1, 9, 6, 1], Fold MSE: 0.003466735412638111
Layers: [1, 9, 6, 1], Fold MSE: 0.003466735412638111
Layers: [1, 9, 6, 1], Fold MSE: 0.003466735412638111
In this layer test: [1, 9, 6, 1], Average MSE: 0.003466735412638111
Layers: [1, 9, 21, 6, 1], Fold MSE: 0.0019875823724081034
Layers: [1, 9, 21, 6, 1], Fold MSE: 0.0019875823724081034
Layers: [1, 9, 21, 6, 1], Fold MSE: 0.0019875823724081034
Layers: [1, 9, 21, 6, 1], Fold MSE: 0.0019875823724081034
Layers: [1, 9, 21, 6, 1], Fold MSE: 0.0019875823724081034
In this layer test: [1, 9, 21, 6, 1], Average MSE: 0.0019875823724081034
Layers: [1, 9, 21, 14, 6, 1], Fold MSE: 0.00209688022635737
Layers: [1, 9, 21, 14, 6, 1], Fold MSE: 0.00209688022635737
Layers: [1, 9, 21, 14, 6, 1], Fold MSE: 0.00209688022635737
Layers: [1, 9, 21, 14, 6, 1], Fold MSE: 0.00209688022635737
Layers: [1, 9, 21, 14, 6, 1], Fold MSE: 0.00209688022635737
In this layer test: [1, 9, 21, 14, 6, 1], Average MSE: 0.00209688022635737
Layers: [1, 9, 21, 39, 14, 6, 1], Fold MSE: 0.0019867735818585924
Layers: [1, 9, 21, 39, 14, 6, 1], Fold MSE: 0.0019867735818585924
Layers: [1, 9, 21, 39, 14, 6, 1], Fold MSE: 0.0019867735818585924
Layers: [1, 9, 21, 39, 14, 6, 1], Fold MSE: 0.0019867735818585924
Layers: [1, 9, 21, 39, 14, 6, 1], Fold MSE: 0.0019867735818585924
In this layer test: [1, 9, 21, 39, 14, 6, 1], Average MSE: 0.0019867735818585924
Layers: [1, 9, 21, 39, 26, 14, 6, 1], Fold MSE: 0.002136578738297147
Layers: [1, 9, 21, 39, 26, 14, 6, 1], Fold MSE: 0.002136578738297147
Layers: [1, 9, 21, 39, 26, 14, 6, 1], Fold MSE: 0.002136578738297147
Layers: [1, 9, 21, 39, 26, 14, 6, 1], Fold MSE: 0.002136578738297147
Layers: [1, 9, 21, 39, 26, 14, 6, 1], Fold MSE: 0.002136578738297147
In this layer test: [1, 9, 21, 39, 26, 14, 6, 1], Average MSE: 0.002136578738297147
Early stopping triggered
Best Model - MSE: 0.38383448075092513

```

(a) Command Output

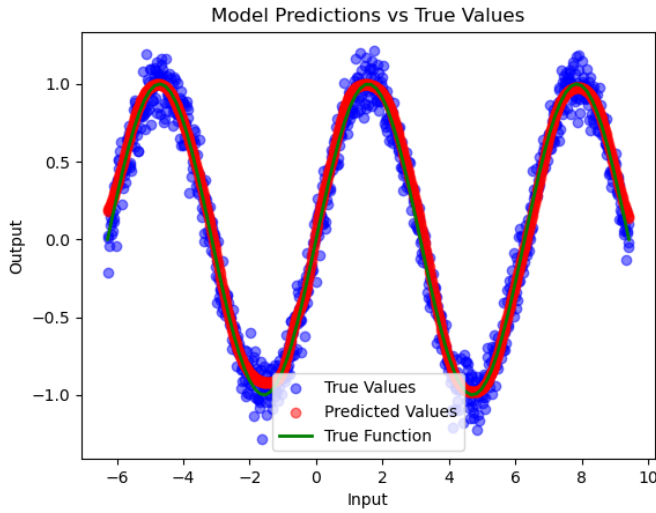


(b) Loss Trend

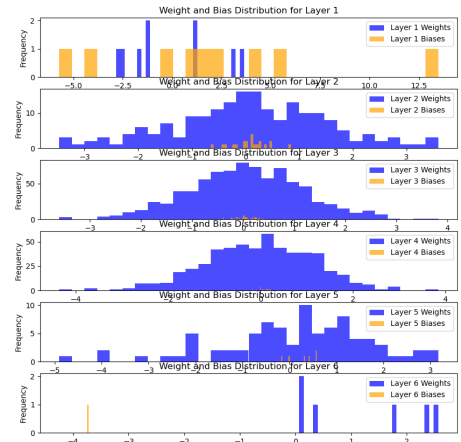
Fig. 4: The Training Course of Nonlinear Function by MBGD Method

The model can perfectly fit the input data.

We can visualize the predictions and true labels. At the same time, we visualize the distribution of weights and bias.



(a) Fitting Result



(b) Weights and Bias Graph

Fig. 5: Nonlinear Dataset - Using MBGD

6 The Result of Nonlinear Dataset by SGD Method

SGD uses less memory, thus can deal with large amounts of samples, but it isn't capable of parallel processing, so it works slowly. As linear dataset can be easily done regression problem, nonlinear dataset can better prove the performance of our model.

```

D:\wendsy\study\2020-2022Fall\人工智能与机器学习\theUnderpython\hnd.py
Training Nonlinear, Learning rate: 0.1, epochs: 1000, method: sgd
Layers: [1, 9, 1], Fold MSE: 0.002782409758560894
Layers: [1, 9, 1], Fold MSE: 0.00278090392811914
Layers: [1, 9, 1], Fold MSE: 0.005289074733375334
Layers: [1, 9, 1], Fold MSE: 0.00369076269602201
Layers: [1, 9, 1], Fold MSE: 0.004670832334190486
In this layer test: [1, 9, 1], Average MSE: 0.0038816708299745093

Layers: [1, 9, 6, 1], Fold MSE: 0.003858779262651775
Layers: [1, 9, 6, 1], Fold MSE: 0.00333509236639315
Layers: [1, 9, 6, 1], Fold MSE: 0.00233902296291162
Layers: [1, 9, 6, 1], Fold MSE: 0.0040823180189818009
Layers: [1, 9, 6, 1], Fold MSE: 0.003923101598786024
In this layer test: [1, 9, 6, 1], Average MSE: 0.00388538316689595

Layers: [1, 9, 21, 6, 1], Fold MSE: 0.00582414569827102
Layers: [1, 9, 21, 6, 1], Fold MSE: 0.002022208318739166
Layers: [1, 9, 21, 6, 1], Fold MSE: 0.002212377764071475
Layers: [1, 9, 21, 6, 1], Fold MSE: 0.0026056618106131146
Layers: [1, 9, 21, 6, 1], Fold MSE: 0.003938616239940524
In this layer test: [1, 9, 21, 6, 1], Average MSE: 0.003324311312526211

Layers: [1, 9, 21, 14, 6, 1], Fold MSE: 0.0016931405681656695
Layers: [1, 9, 21, 14, 6, 1], Fold MSE: 0.0021371739390181976
Layers: [1, 9, 21, 14, 6, 1], Fold MSE: 0.0023605565638083817
Layers: [1, 9, 21, 14, 6, 1], Fold MSE: 0.0023212463752369883
Layers: [1, 9, 21, 14, 6, 1], Fold MSE: 0.0030942483907561
In this layer test: [1, 9, 21, 14, 6, 1], Average MSE: 0.0021204979970729674

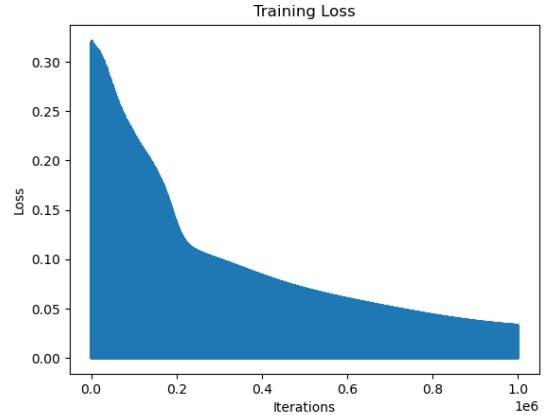
Layers: [1, 9, 21, 39, 14, 6, 1], Fold MSE: 0.0018383589107744194
Layers: [1, 9, 21, 39, 14, 6, 1], Fold MSE: 0.00168151568955657
Layers: [1, 9, 21, 39, 14, 6, 1], Fold MSE: 0.0018040254607926076
Layers: [1, 9, 21, 39, 14, 6, 1], Fold MSE: 0.0019401068835207124
Layers: [1, 9, 21, 39, 14, 6, 1], Fold MSE: 0.002100764338842627
In this layer test: [1, 9, 21, 39, 14, 6, 1], Average MSE: 0.001803642782962017

Layers: [1, 9, 21, 39, 26, 14, 6, 1], Fold MSE: 0.0017315862642783745
Layers: [1, 9, 21, 39, 26, 14, 6, 1], Fold MSE: 0.002086273231618147
Layers: [1, 9, 21, 39, 26, 14, 6, 1], Fold MSE: 0.0021626266995214063
Layers: [1, 9, 21, 39, 26, 14, 6, 1], Fold MSE: 0.001802007730777885
Layers: [1, 9, 21, 39, 26, 14, 6, 1], Fold MSE: 0.0016495228212580833
In this layer test: [1, 9, 21, 39, 26, 14, 6, 1], Average MSE: 0.0018559472358267603

Early stopping triggered
Best Model - MSE: 0.35278046157892876

```

(a) Command Output



(b) Loss Trend

Fig. 6: The Training Course of Nonlinear Function by SGD Method

I take sin function to be the dataset. Those sample points that are far from the origin require very many iterations to fit gradually. After many attempts, we chose the following parameters:

learning rate: 0.1, epochs: 1000, method: sgd

We do the choices in the main function:

```
1 dataset = 'nonlinear'
```



```

2 | n_samples = 1000
3 | learning_rate, epochs = 0.1, 1000
4 | method = 'sgd'
5 | k = 5, tolerance = 1e-3, patience = 2

```

The training courses is as follows:

In this layer test: [1, 9, 1], Average MSE: 0.0038816768299745593

In this layer test: [1, 9, 6, 1], Average MSE: 0.0030885383816689595

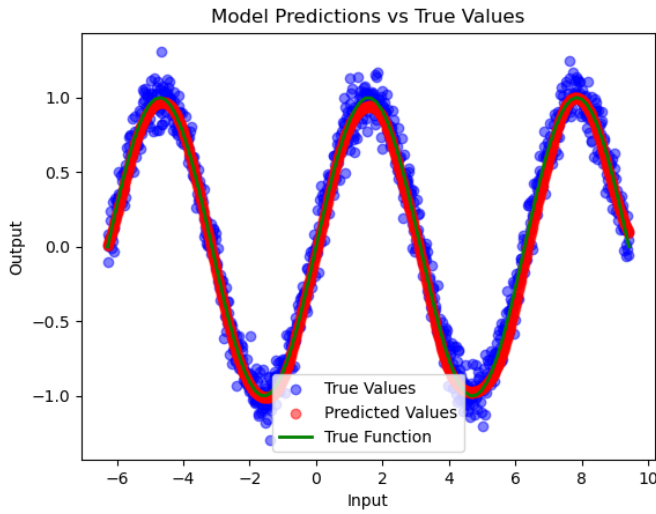
In this layer test: [1, 9, 21, 6, 1], Average MSE: 0.003324313132526211

In this layer test: [1, 9, 21, 14, 6, 1], Average MSE: 0.0021204979970729674

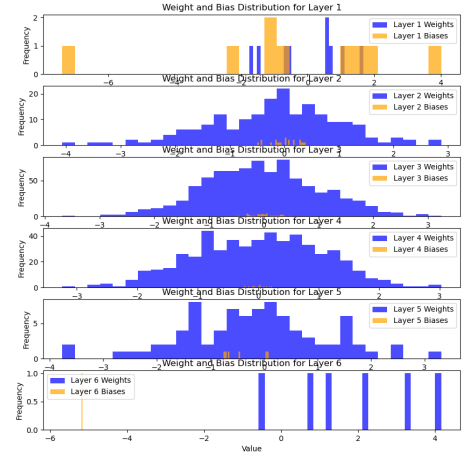
In this layer test: [1, 9, 21, 39, 14, 6, 1], Average MSE: 0.0018803642782962017

In this layer test: [1, 9, 21, 39, 26, 14, 6, 1], Average MSE: 0.0018859472358267603

Then it triggers early stopping. Take all samples into account, Best Model - MSE: 0.35270846157802876



(a) Fitting Result



(b) Weights and Bias Graph

Fig. 7: Nonlinear Dataset - Using SGD

The model can perfectly fit the input data.

We can visualize the predictions and true labels. At the same time, we visualize the distribution of weights and bias.

7 The Result of Classifier Dataset by MBGD Method

MBGD method takes advantage of SGD and BGD, it's quickly and effectively. MLP model can also do classification problem, especially for those without linear bounds.

I put in six clusters of data labeled 0 and 1. The different categories have various intersections and overlaps, so it is a very challenging task. In that case, we can use MLP to validate our model. After many attempts, we chose the following parameters:

learning rate: 0.01, epochs: 500, method: mbgd

We do the choices in the main function:

```

1 | dataset = 'classifier'
2 | n_samples = 1200

```

```

3 learning_rate, epochs = 0.01, 500
4 method = 'mbgd'
5 k = 5, tolerance = 1e-4, patience = 3

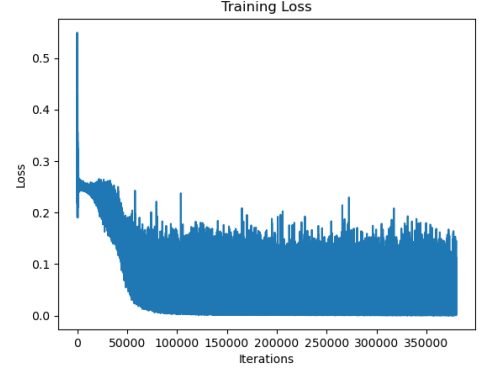
```

```

0:\study\2024-02\fall\1.1 机器学习\08\python\mbgd.py
Training Classifier, Learning rate: 0.01, Epochs: 500, method: mbgd
Layers: [2, 6, 1], Field Accuracy: 0.867, Field Precision: 0.863, Field Recall: 0.862, Field F1 score: 0.866
Layers: [2, 6, 1], Field Accuracy: 0.837, Field Precision: 0.843, Field Recall: 0.876, Field F1 score: 0.824
Layers: [2, 6, 1], Field Accuracy: 0.866, Field Precision: 0.866, Field Recall: 0.867, Field F1 score: 0.866
Layers: [2, 6, 1], Field Accuracy: 0.829, Field Precision: 0.837, Field Recall: 0.866, Field F1 score: 0.828
Layers: [2, 6, 1], Field Accuracy: 0.893, Field Precision: 0.878, Field Recall: 0.911, Field F1 score: 0.904
In this layer test: [2, 6, 1], Average Accuracy: 0.865, Average Precision: 0.8551023622047245, Average Recall: 0.8826159321021665, Average F1 score: 0.8680747668228099
Layers: [2, 6, 3, 1], Field Accuracy: 0.921, Field Precision: 0.907, Field Recall: 0.904, Field F1 score: 0.923
Layers: [2, 6, 3, 1], Field Accuracy: 0.907, Field Precision: 0.900, Field Recall: 0.879, Field F1 score: 0.911
Layers: [2, 6, 3, 1], Field Accuracy: 0.896, Field Precision: 0.912, Field Recall: 0.906, Field F1 score: 0.903
Layers: [2, 6, 3, 1], Field Accuracy: 0.933, Field Precision: 0.904, Field Recall: 0.926, Field F1 score: 0.908
Layers: [2, 6, 3, 1], Field Accuracy: 0.929, Field Precision: 0.909, Field Recall: 0.927, Field F1 score: 0.927
In this layer test: [2, 6, 3, 1], Average Accuracy: 0.9261333333333333, Average Precision: 0.9265513546765902, Average Recall: 0.9253511746089229, Average F1 score: 0.9235595010559691
Layers: [2, 6, 14, 3, 1], Field Accuracy: 0.931, Field Precision: 0.934, Field Recall: 0.922, Field F1 score: 0.923
Layers: [2, 6, 14, 3, 1], Field Accuracy: 0.909, Field Precision: 0.922, Field Recall: 0.879, Field F1 score: 0.918
Layers: [2, 6, 14, 3, 1], Field Accuracy: 0.931, Field Precision: 0.916, Field Recall: 0.922, Field F1 score: 0.920
Layers: [2, 6, 14, 3, 1], Field Accuracy: 0.965, Field Precision: 0.965, Field Recall: 0.967, Field F1 score: 0.966
Layers: [2, 6, 14, 3, 1], Field Accuracy: 0.972, Field Precision: 0.964, Field Recall: 0.969, Field F1 score: 0.970
In this layer test: [2, 6, 14, 3, 1], Average Accuracy: 0.9608333333333333, Average Precision: 0.955330760642248, Average Recall: 0.913485629346499, Average F1 score: 0.8768421015396651
Layers: [2, 6, 14, 7, 3, 1], Field Accuracy: 0.962, Field Precision: 0.961, Field Recall: 0.961, Field F1 score: 0.961
Layers: [2, 6, 14, 7, 3, 1], Field Accuracy: 0.969, Field Precision: 0.969, Field Recall: 0.966, Field F1 score: 0.968
Layers: [2, 6, 14, 7, 3, 1], Field Accuracy: 0.967, Field Precision: 0.967, Field Recall: 0.966, Field F1 score: 0.966
Layers: [2, 6, 14, 7, 3, 1], Field Accuracy: 0.972, Field Precision: 0.973, Field Recall: 0.966, Field F1 score: 0.968
Layers: [2, 6, 14, 7, 3, 1], Field Accuracy: 0.939, Field Precision: 0.938, Field Recall: 0.929, Field F1 score: 0.931
In this layer test: [2, 6, 14, 7, 3, 1], Average Accuracy: 0.93889962269575, Average Precision: 0.9327203948829395, Average Recall: 0.9266889062269575, Average F1 score: 0.9327203948829395
Layers: [2, 6, 14, 26, 7, 3, 1], Field Accuracy: 0.933, Field Precision: 0.922, Field Recall: 0.922, Field F1 score: 0.922
Layers: [2, 6, 14, 26, 7, 3, 1], Field Accuracy: 0.907, Field Precision: 0.902, Field Recall: 0.907, Field F1 score: 0.906
Layers: [2, 6, 14, 26, 7, 3, 1], Field Accuracy: 0.933, Field Precision: 0.907, Field Recall: 0.907, Field F1 score: 0.916
Layers: [2, 6, 14, 26, 7, 3, 1], Field Accuracy: 0.933, Field Precision: 0.907, Field Recall: 0.907, Field F1 score: 0.912
Layers: [2, 6, 14, 26, 7, 3, 1], Field Accuracy: 0.925, Field Precision: 0.909, Field Recall: 0.902, Field F1 score: 0.920
In this layer test: [2, 6, 14, 26, 7, 3, 1], Average Accuracy: 0.9316666666666666, Average Precision: 0.9474269769031529, Average Recall: 0.9136512259186576, Average F1 score: 0.9300985673022767
Layers: [2, 6, 14, 26, 13, 7, 3, 1], Field Accuracy: 0.866, Field Precision: 0.866, Field Recall: 1.000, Field F1 score: 0.817
Layers: [2, 6, 14, 26, 13, 7, 3, 1], Field Accuracy: 0.929, Field Precision: 0.921, Field Recall: 0.906, Field F1 score: 0.920
Layers: [2, 6, 14, 26, 13, 7, 3, 1], Field Accuracy: 0.929, Field Precision: 0.906, Field Recall: 0.902, Field F1 score: 0.912
Layers: [2, 6, 14, 26, 13, 7, 3, 1], Field Accuracy: 0.925, Field Precision: 0.929, Field Recall: 0.922, Field F1 score: 0.921
In this layer test: [2, 6, 14, 26, 13, 7, 3, 1], Average Accuracy: 0.9256666666666666, Average Precision: 0.8501455736898842, Average Recall: 0.832650671550671, Average F1 score: 0.8660485083425306
Layers: [2, 6, 14, 26, 38, 13, 7, 3, 1], Field Accuracy: 0.907, Field Precision: 0.889, Field Recall: 0.888, Field F1 score: 0.888
Layers: [2, 6, 14, 26, 38, 13, 7, 3, 1], Field Accuracy: 0.917, Field Precision: 0.901, Field Recall: 0.881, Field F1 score: 0.900
Layers: [2, 6, 14, 26, 38, 13, 7, 3, 1], Field Accuracy: 0.919, Field Precision: 0.863, Field Recall: 0.897, Field F1 score: 0.924
Layers: [2, 6, 14, 26, 38, 13, 7, 3, 1], Field Accuracy: 0.921, Field Precision: 0.941, Field Recall: 0.960, Field F1 score: 0.922
Layers: [2, 6, 14, 26, 38, 13, 7, 3, 1], Field Accuracy: 0.907, Field Precision: 0.859, Field Recall: 0.886, Field F1 score: 0.904
In this layer test: [2, 6, 14, 26, 38, 13, 7, 3, 1], Average Accuracy: 0.9087720465890182, Average Precision: 0.9352159468438538, Average Recall: 0.9383333333333334, Average F1 score: 0.9367720465890182
Early stopping triggered
Best Model - Accuracy: 0.9366666666666666, Precision: 0.9352159468438538, Recall: 0.9383333333333334, F1 score: 0.9367720465890182

```

(a) Command Output



(b) Loss Trend

Fig. 8: The Training Course of Classifier Problem by MBGD Method

The training courses are as follows:

In this layer test: [2, 6, 1], Average Accuracy: 0.865, Average Precision: 0.8551023622047245, Average Recall: 0.8826159321021665, Average F1 score: 0.8680747668228099

In this layer test: [2, 6, 3, 1], Average Accuracy: 0.9258333333333333, Average Precision: 0.9265513546765902, Average Recall: 0.9253511746089229, Average F1 score: 0.9253595010559691

In this layer test: [2, 6, 14, 3, 1], Average Accuracy: 0.8608333333333332, Average Precision: 0.855330760642248, Average Recall: 0.913485629346499, Average F1 score: 0.8768421015396651

In this layer test: [2, 6, 14, 7, 3, 1], Average Accuracy: 0.9333333333333333, Average Precision: 0.9389368161485347, Average Recall: 0.9266889062269575, Average F1 score: 0.9327203948829395

In this layer test: [2, 6, 14, 26, 7, 3, 1], Average Accuracy: 0.9316666666666666, Average Precision: 0.9474269769031529, Average Recall: 0.9136512259186576, Average F1 score: 0.9300985673022767

In this layer test: [2, 6, 14, 26, 13, 7, 3, 1], Average Accuracy: 0.8316666666666667, Average Precision: 0.8501455736898842, Average Recall: 0.9255550671550671, Average F1 score: 0.8660485083425306

In this layer test: [2, 6, 14, 26, 38, 13, 7, 3, 1], Average Accuracy: 0.925, Average Precision: 0.9401063212625675, Average Recall: 0.906767460717532, Average F1 score: 0.922524412210004

Then it triggers early stopping. Take all samples into account, Best Model - Accuracy: 0.9366666666666666, Precision: 0.9352159468438538, Recall: 0.9383333333333334, F1 score: 0.9367720465890182

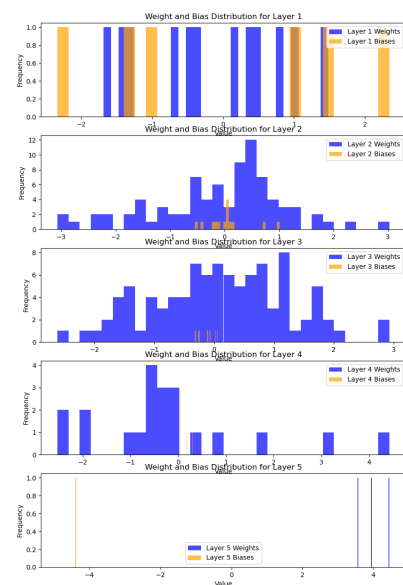
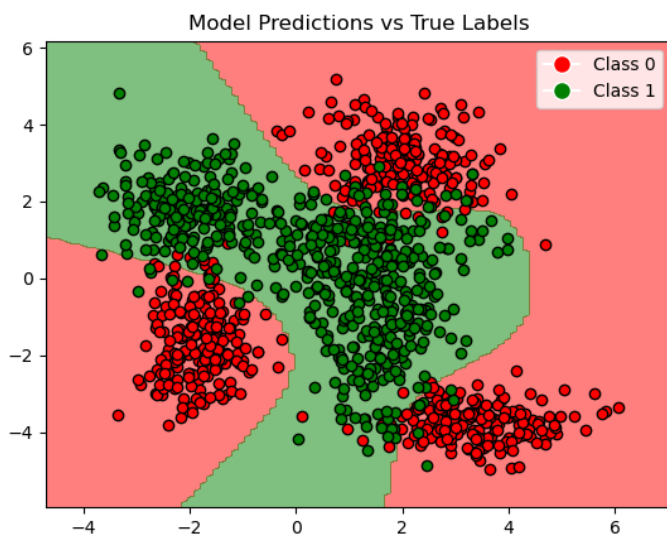


Fig. 9: Classification Dataset - Using MBGD

The model can perfectly fit the input data.

We can visualize the predictions and true labels. At the same time, we visualize the distribution of weights and bias.

8 The Result of Classifier Dataset by SGD Method

SGD uses less memory, thus can deal with large amounts of samples, but it isn't capable of parallel processing, so it works slowly. MLP model can also do classification problem, especially for those without linear bounds.

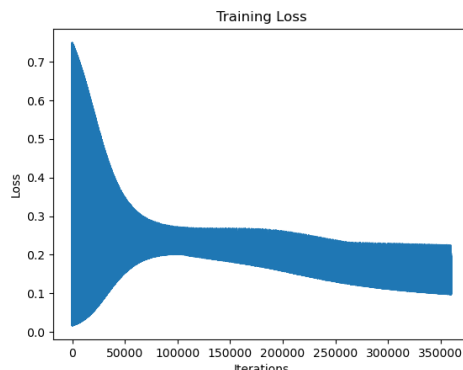
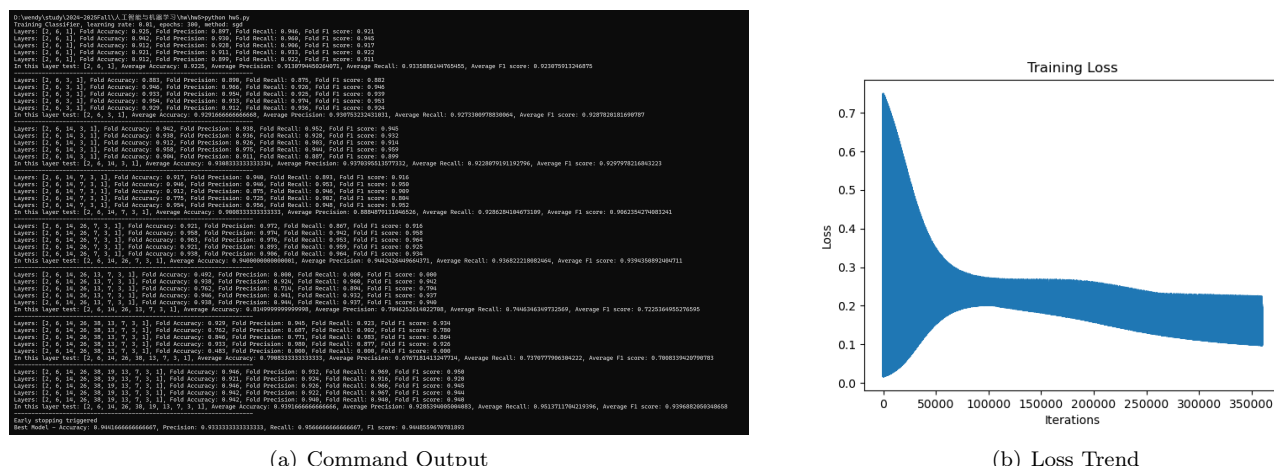


Fig. 10: The Training Course of Classifier Problem by SGD Method

I put in six clusters of data labeled 0 and 1. The different categories have various intersections and overlaps, so it is

a very challenging task. In that case, we can use MLP to validate our model. After many attempts, we chose the following parameters:

learning rate: 0.01, epochs: 300, method: sgd

We do the choices in the main function:

```

1 dataset = 'classifier'
2 n_samples = 1200
3 learning_rate, epochs = 0.01, 300
4 method = 'sgd'
5 k = 5, tolerance = 1e-4, patience = 3

```

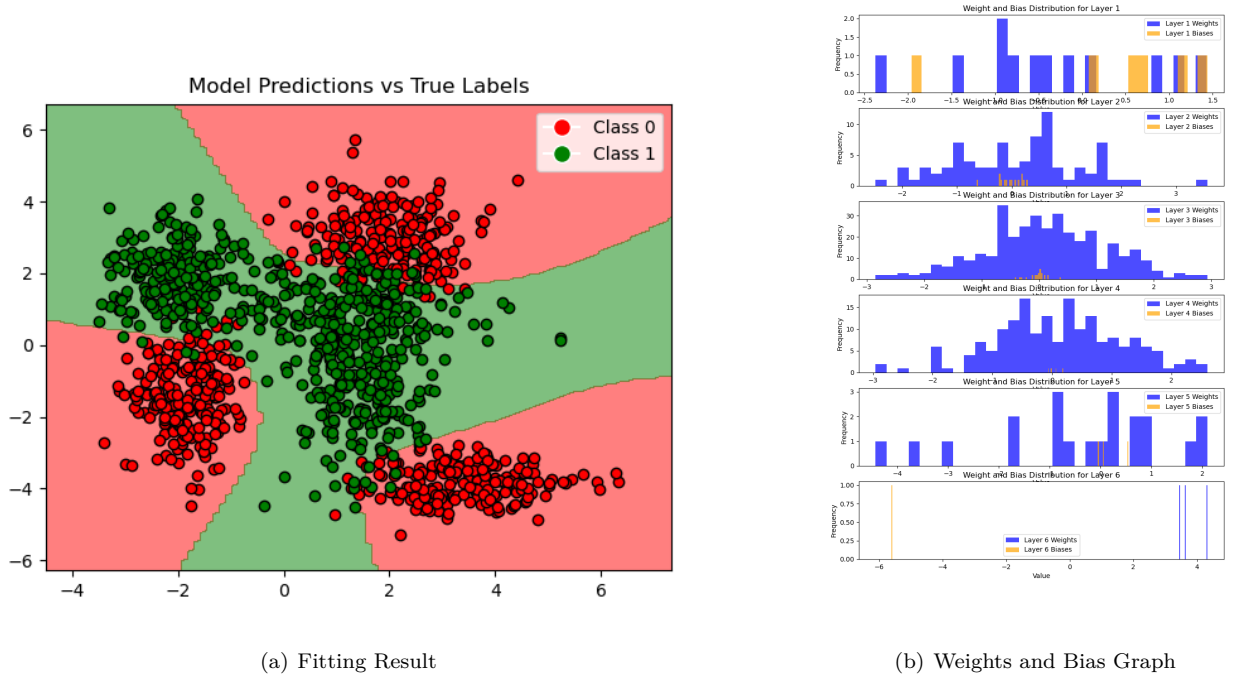


Fig. 11: Classification Dataset - Using SGD

The traning courses is as follows:

In this layer test: [2, 6, 1], Average Accuracy: 0.9225, Average Precision: 0.9130794450264071, Average Recall: 0.9335886144765455, Average F1 score: 0.923075913246875

In this layer test: [2, 6, 3, 1], Average Accuracy: 0.9291666666666668, Average Precision: 0.930753232431031, Average Recall: 0.9273300978830064, Average F1 score: 0.9287820181690787

In this layer test: [2, 6, 14, 3, 1], Average Accuracy: 0.9308333333333334, Average Precision: 0.9370395513577332, Average Recall: 0.9228079191192796, Average F1 score: 0.9297978216843223

In this layer test: [2, 6, 14, 7, 3, 1], Average Accuracy: 0.9008333333333333, Average Precision: 0.8884879131046526, Average Recall: 0.9286284104673109, Average F1 score: 0.9062354274083241

In this layer test: [2, 6, 14, 26, 7, 3, 1], Average Accuracy: 0.9400000000000001, Average Precision: 0.9442426449664371, Average Recall: 0.936822218082464, Average F1 score: 0.9394350892404711

In this layer test: [2, 6, 14, 26, 13, 7, 3, 1], Average Accuracy: 0.8149999999999998, Average Precision: 0.7046252614022708, Average Recall: 0.7446346349732569, Average F1 score: 0.7225364955276595

In this layer test: [2, 6, 14, 26, 38, 13, 7, 3, 1], Average Accuracy: 0.7908333333333333, Average Precision: 0.6767181413247714, Average Recall: 0.7370777906304222, Average F1 score: 0.7008339420790783

In this layer test: [2, 6, 14, 26, 38, 19, 13, 7, 3, 1], Average Accuracy: 0.9391666666666666, Average Precision: 0.9285394005004083, Average Recall: 0.9513711704219396, Average F1 score: 0.9396882050348658
Then it triggers early stopping. Take all samples into account, Best Model - Accuracy: 0.9441666666666667, Precision: 0.9333333333333333, Recall: 0.9566666666666667, F1 score: 0.9448559670781893

The model can perfectly fit the input data.

We can visualize the predictions and true labels. At the same time, we visualize the distribution of weights and bias.

9 Appendix

The code achieve the implementation of multi-layer perceptron. Like the main function tells:

```

1 dataset = 'nonlinear' # 'classify' or 'nonlinear'
2
3 if dataset == 'classify':
4     inputs, targets = generate_data_classifier(n_samples=200)
5     method = 'mbgd'
6     learning_rate, epochs = 0.01, 1000
7     print(f"Training Classifier, learning rate: {learning_rate}, epochs: {epochs},
8           method: {method}")
9     initial_layers = 1
10    max_layers = 10
11    mlp = MultiLayerPerceptron(layer_sizes=[inputs.shape[1]], dataset_type='classify'
12                                ')
13    best_model = mlp.incremental_model_training_classifier(inputs, targets,
14                                                            initial_layers, max_layers, learning_rate, epochs, method=method)
15
16    # Evaluate the best model
17    accuracy, precision, recall, f1 = best_model.evaluate(inputs, targets)
18    print(f"Best Model - Accuracy: {accuracy}, Precision: {precision}, Recall: {
19          recall}, F1 score: {f1}")
20    best_model.visualize_predictions_classifier(inputs, targets)
21    best_model.plot_weights_and_biases()
22
23 elif dataset == 'nonlinear':
24    inputs, targets = generate_nonlinear_dataset(n_samples=1000)
25    learning_rate, epochs = 0.1, 10000
26    method = 'mbgd'
27    print(f"Training Nonlinear, learning rate: {learning_rate}, epochs: {epochs},
28          method: {method}")
29    initial_layers = 1
30    max_layers = 10
31    mlp = MultiLayerPerceptron(layer_sizes=[1], dataset_type='nonlinear')
32    best_model = mlp.incremental_model_training_nonlinear(inputs, targets,
33                                                            initial_layers, max_layers, learning_rate, epochs, method=method)
34
35    # Visualize the predictions
36    mse = best_model.mean_squared_error(best_model.predict(inputs), targets)
37    print(f"Best Model - MSE: {mse}")
38    best_model.visualize_predictions_nonlinear(inputs, targets)
39    best_model.plot_weights_and_biases()

```