AI and Machine Learning, Homework5

Author: Ying Yiwen
Number: 12210159

Contents

1 Develop a Multilayer Perceptron Model

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

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

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

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

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

```python
def backward(self, inputs, targets, learning_rate):
  # Backpropagation process
  output_errors = targets - self.activations[-1]
  output_delta = output_errors * self.activation_derivative(self.activations[-1])

  # Update weights and biases for the output layer
  self.weights[-1] += self.activations[-1].T.dot(output_delta) * learning_rate
  self.biases[-1] += np.sum(output_delta, axis=0, keepdims=True) * learning_rate

  # Calculate errors and update weights for hidden layers
  hidden_errors = output_delta.dot(self.weights[-1].T) # initialize hidden_errors with output_delta for the last layer
  for i in range(self.num_layers - 3, -1, -1):
    # Calculate hidden delta for the current layer
    hidden_delta = hidden_errors * self.activation_derivative(self.activations[i + 1])

    # Update weights and biases for the current hidden layer
    self.weights[i] += self.activations[i].T.dot(hidden_delta) * learning_rate
    self.biases[i] += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate

    # Update hidden_errors for the next layer's calculation
    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).

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

$$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z)) \tag{4}$$

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,

$$x' = \frac{x - min}{x - max} \tag{5}$$

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

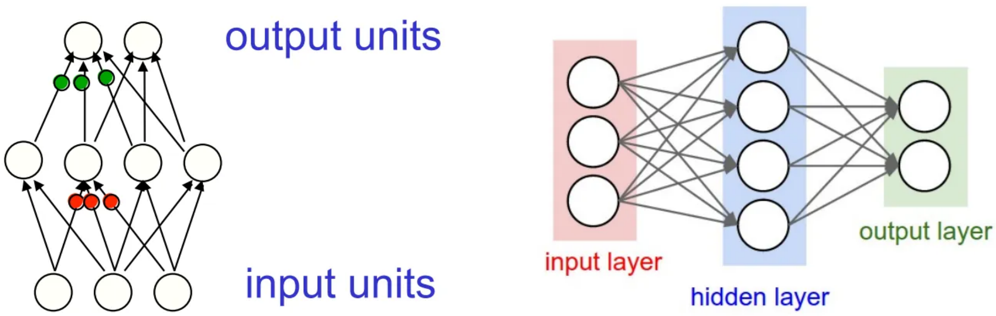


Fig. 1: Schematic of MLP

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.

```
def __init__(self, layer_sizes, dataset_type):
  # layer_sizes is a list containing the number of nodes in each layer
   self.layer_sizes = layer_sizes
   self.num_layers = len(layer_sizes)

  # Initialize weights and biases
   self.weights = []
   self.biases = []

  # Initialize weights and biases for each layer
   for i in range(self.num_layers - 1):
   weight = np.random.randn(layer_sizes[i], layer_sizes[i + 1])
   bias = np.zeros((1, layer_sizes[i + 1]))
   self.weights.append(weight)
   self.biases.append(bias)
```

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

```
for layers in range(initial_layers, max_layers + 1):
   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.

```
# Early stopping mechanism
if average_accuracy >= best_score:
```

```
  if average_accuracy - best_score > tolerance:
    no_improve_count = 0  # Reset counter
  best_score = average_accuracy
  best_model = model
else:
  no_improve_count += 1  # Increment counter if no significant improvement

# Stop if no significant improvement for consecutive `patience` times
if no_improve_count >= patience:
  print("Early stopping triggered")
  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.

```
def mbgd_train(self, inputs, targets, learning_rate, epochs, batch_size=32):
  # Train the model over specified epochs with mini-batch gradient descent
  for epoch in range(epochs):
    # Shuffle training data for each epoch to improve generalization
    indices = np.arange(inputs.shape[0])
    np.random.shuffle(indices)
    inputs_train = inputs[indices]
    targets_train = targets[indices]

    # Process each mini-batch
    for start in range(0, len(inputs_train), batch_size):
      end = min(start + batch_size, len(inputs_train))
      batch_inputs = inputs_train[start:end]
      batch_targets = targets_train[start:end]

      # Forward pass and get predictions
      predictions = self.forward(batch_inputs)
      # Backward pass for the mini-batch
      self.backward(batch_inputs, batch_targets, learning_rate)

      # Calculate loss for the current mini-batch
      batch_loss = self.mean_squared_error(predictions, batch_targets)
```

```
def sgd_train(self, inputs, targets, learning_rate, epochs):
  # Train the model over specified epochs using stochastic gradient descent
  for epoch in range(epochs):
    # Shuffle training data for each epoch to improve generalization
    indices = np.arange(len(inputs))
    np.random.shuffle(indices)
    inputs_train = inputs[indices]
    targets_train = targets[indices]

    # Iterate over each sample in the training set
    for i in range(len(inputs_train)):
      # Forward pass for a single sample
      predictions = self.forward(inputs_train[i:i+1])
```

```
        # Backward pass for the same sample
        self.backward(inputs_train[i:i+1], targets_train[i:i+1], learning_rate)

        # Calculate loss for the current sample
        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.

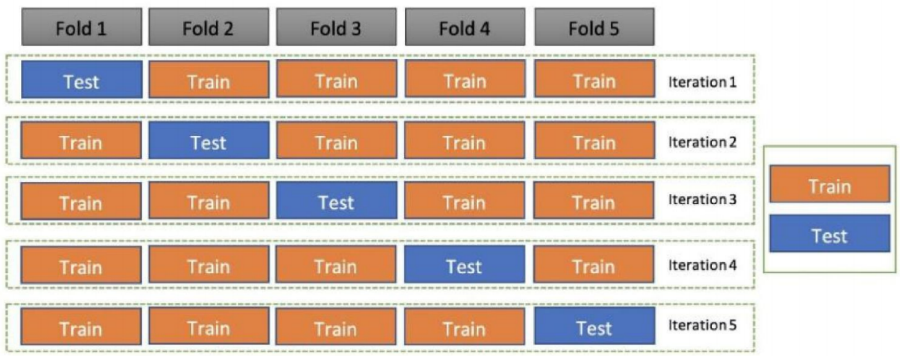


Fig. 2: Schematic of k-fold cross-validation

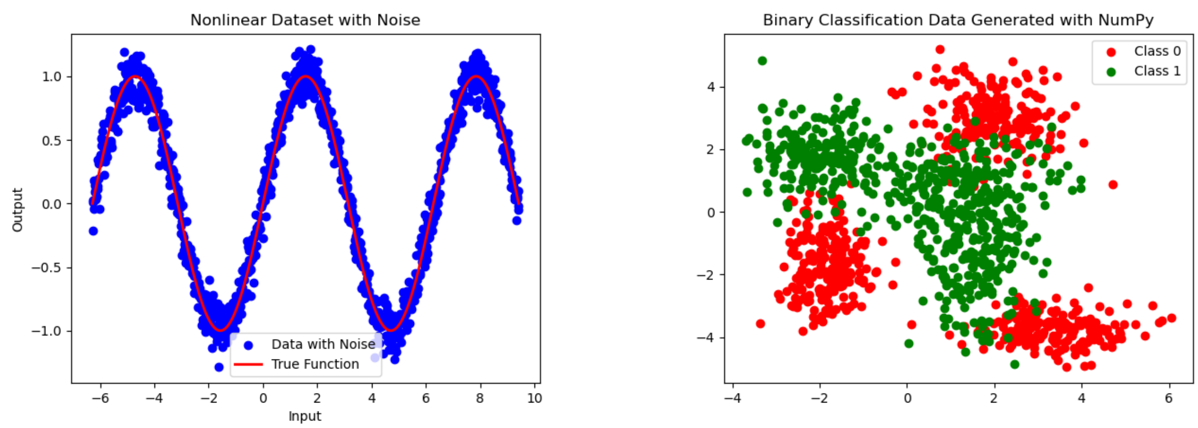
```python
def k_fold_split(self, length, k):
  indices = np.arange(length)
  np.random.shuffle(indices)  # shuffle the indices
  folds = np.array_split(indices, k)  # split the data into k folds
  # yield the train and validation indices
  for i in range(k):
    train_indices = np.concatenate(folds[:i] + folds[i+1:])  # train set
    val_indices = folds[i]  # validation set
    yield train_indices, val_indices
```

```python
for train_indices, val_indices in self.k_fold_split(len(inputs), k):
  model = MultiLayerPerceptron(layer_sizes, dataset_type='classify')

  # Train on training set
  model.train(inputs[train_indices], targets[train_indices], learning_rate=
      learning_rate, epochs=epochs, method=method)

  # Validate on validation set
  accuracy, precision, recall, f1 = model.evaluate(inputs[val_indices], targets[
      val_indices])
  accuracy_scores.append(accuracy)
  precision_scores.append(precision)
  recall_scores.append(recall)
  f1_scores.append(f1)

```

```
    print(f"Layers: {layer_sizes}, Fold Accuracy: {accuracy:.3f}, Fold Precision: {precision:.3f}, Fold Recall: {recall:.3f}, Fold F1 score: {f1:.3f}")
    model.visualize_predictions_classifier(inputs[val_indices], targets[val_indices])

average_accuracy = np.mean(accuracy_scores)
average_precision = np.mean(precision_scores)
average_recall = np.mean(recall_scores)
average_f1 = np.mean(f1_scores)
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}")
print("--------------------------------------------------------------------------------")
```

4 Generate the Dataset

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



(a) Nonlinear Dataset

(b) Classifier Dataset

Fig. 3: Generate the Dataset

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

```
# generate input features
X = np.linspace(-2 * np.pi, 3 * np.pi, n_samples).reshape(-1, 1)
noise_level = 0.1  # Adjust this value to control the noise level
# generate output labels
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:

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

```
X.append(x1)
y.append(1)
```

5 The Result of Nonlinear Dataset by MBGD Method

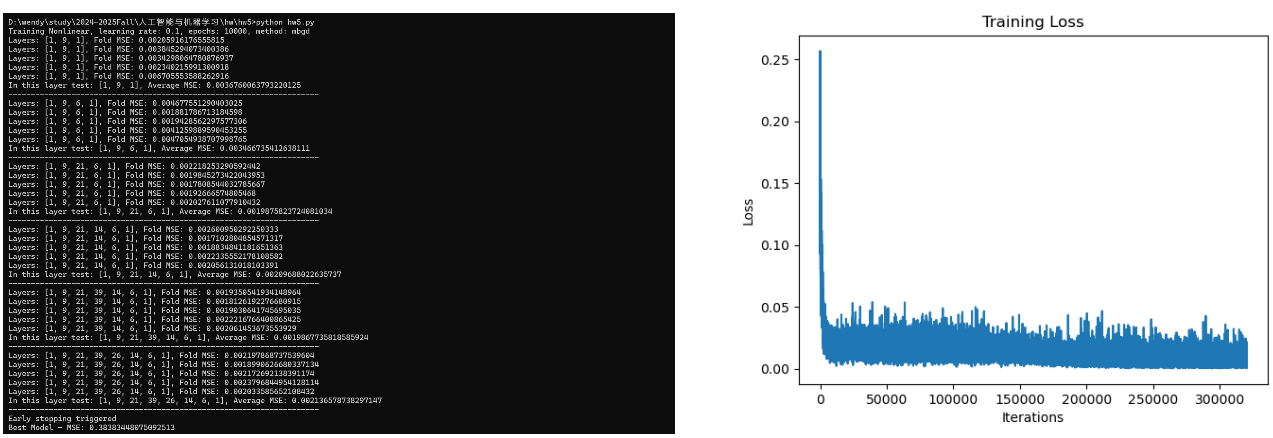
MBGD method takes advantage of SGD and BGD, it's quickly and effectively. As linear dataset can be easily done regression problem, nonlinear dataset can better prove the performance of our model.

I take sin funcion 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:

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

The traning 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

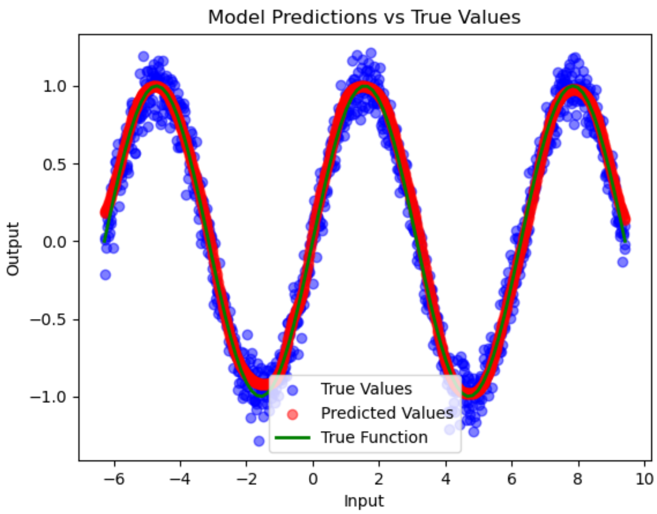


(a) Command Output

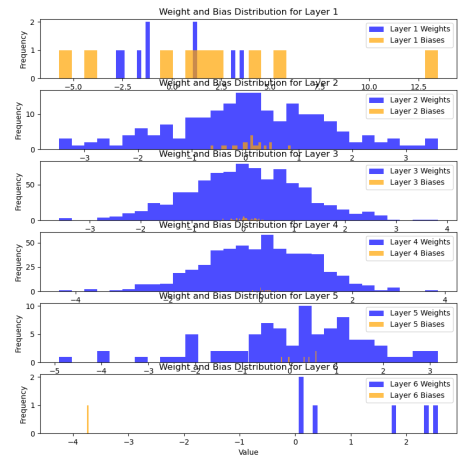
(b) Loss Trend

Fig. 4: The Traning 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

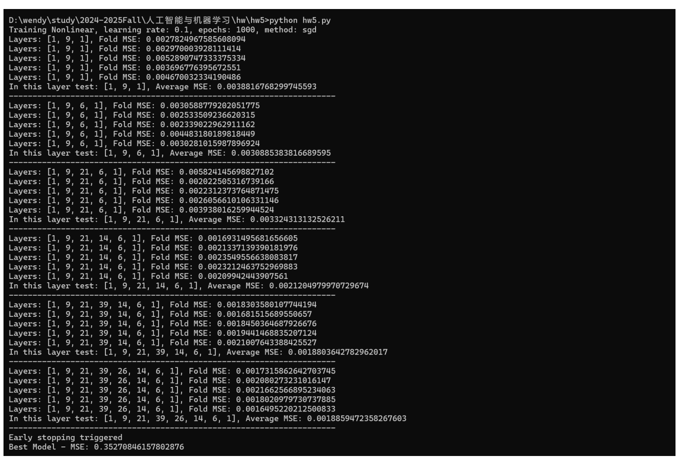


(b) Weights and Bias Graph

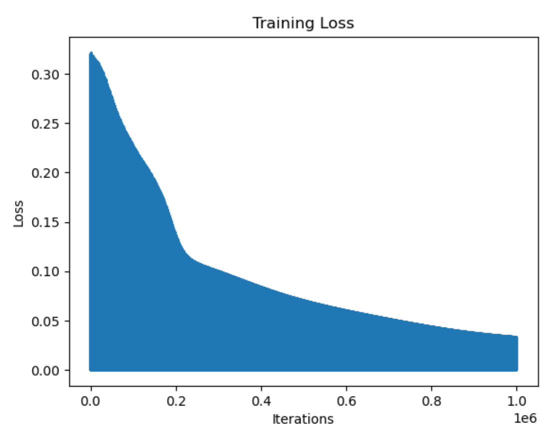
Fig. 5: Nonlinear Dataset - Using MBGD

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.



(a) Command Output



(b) Loss Trend

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

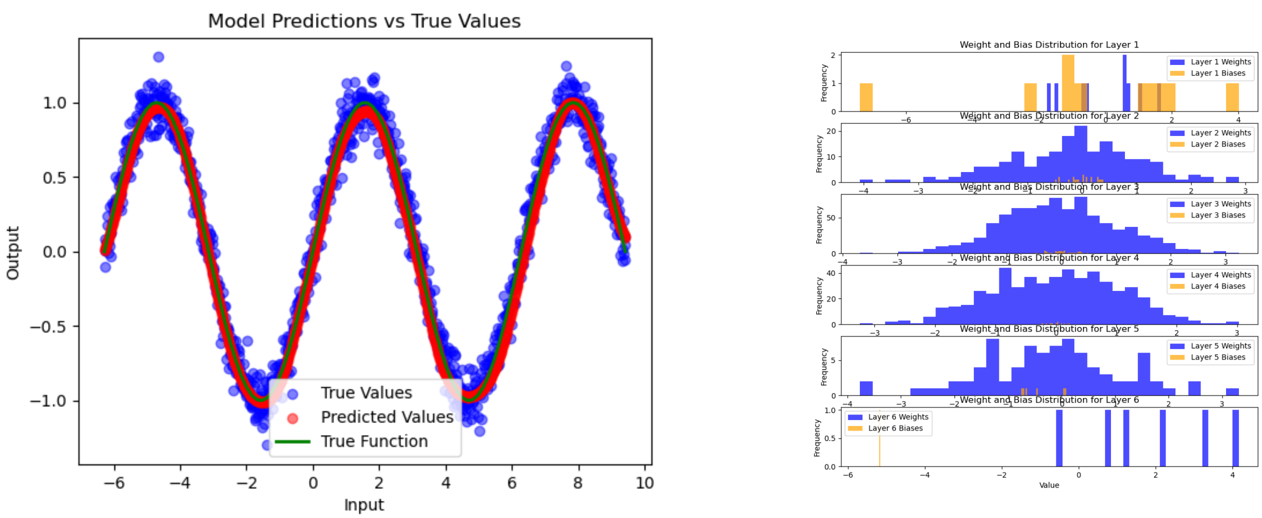
I take sin funcion to be the dataset. Those sample points that are far from the origin require very many iterations to fit gradually. After many attempts, we chose the following parameters:
learning rate: 0.1, epochs: 1000, method: sgd

We do the choices in the main function:

```
dataset = 'nonlinear'
```

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

The traning 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

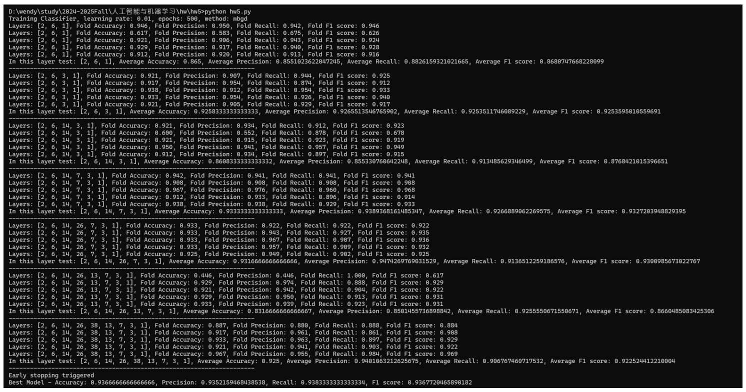
MBGD method takes advantage of SGD and BGD, it's quickly and effectively. MLP model can also do classification problem, especially for those without linear bounds.

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

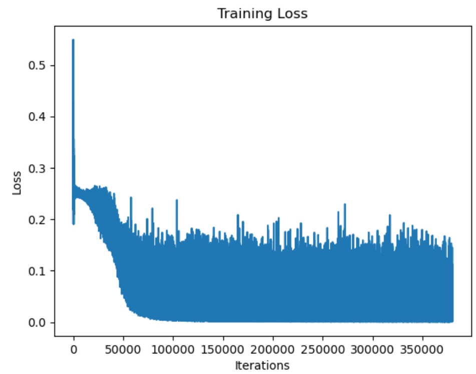
We do the choices in the main function:

```
dataset = 'classifier'
n_samples = 1200
```

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



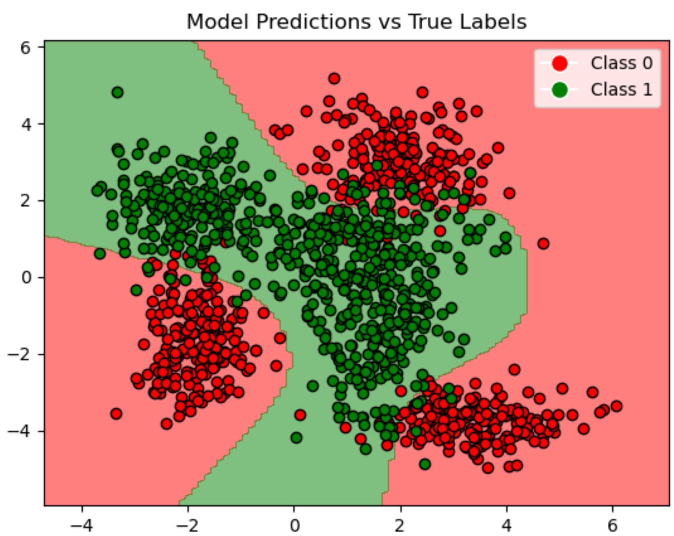
(a) Command Output



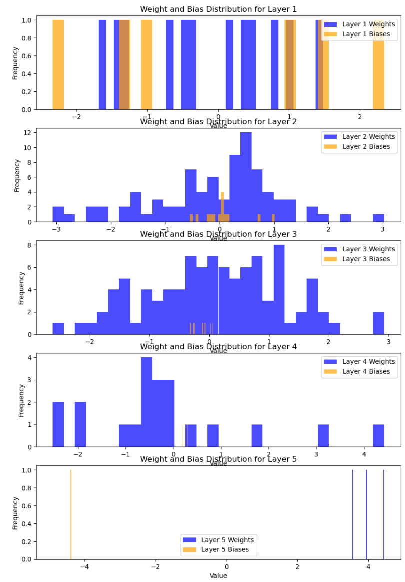
(b) Loss Trend

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

The traning courses is 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



(a) Fitting Result



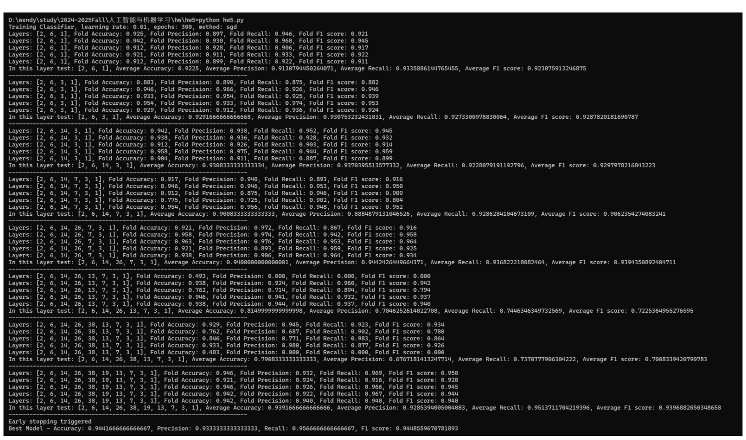
(b) Weights and Bias Graph

Fig. 9: Classification Dataset - Using MBGD

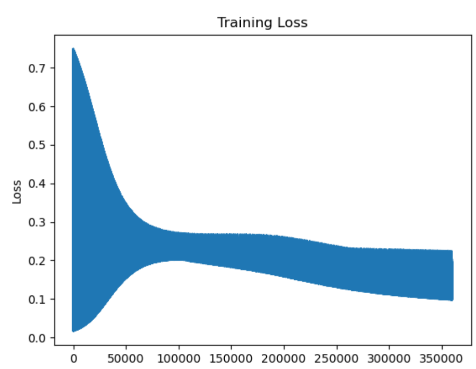
The model can perfectly fit the input data.
We can visualize the predictions and true labels. At the same time, we visualize the distribution of weights and bias.

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.



(a) Command Output



(b) Loss Trend

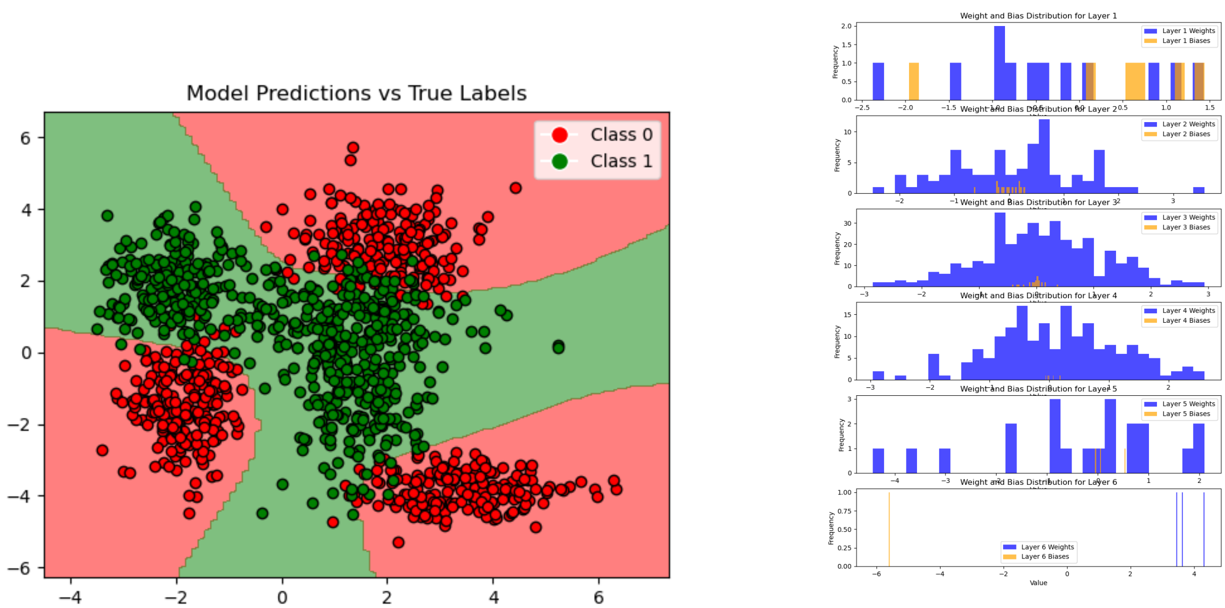
Fig. 10: The Traning Course of Classifier Problem by SGD Method

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:

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



(a) Fitting Result

(b) Weights and Bias Graph

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:

```python
dataset = 'nonlinear'  # 'classify' or 'nonlinear'

if dataset == 'classify':
  inputs, targets = generate_data_classifier(n_samples=200)
  method = 'mbgd'
  learning_rate, epochs = 0.01, 1000
  print(f"Training Classifier, learning rate: {learning_rate}, epochs: {epochs}, method: {method}")
  initial_layers = 1
  max_layers = 10
  mlp = MultiLayerPerceptron(layer_sizes=[inputs.shape[1]], dataset_type='classify')
  best_model = mlp.incremental_model_training_classifier(inputs, targets, initial_layers, max_layers, learning_rate, epochs, method=method)

  # Evaluate the best model
  accuracy, precision, recall, f1 = best_model.evaluate(inputs, targets)
  print(f"Best Model - Accuracy: {accuracy}, Precision: {precision}, Recall: {recall}, F1 score: {f1}")
  best_model.visualize_predictions_classifier(inputs, targets)
  best_model.plot_weights_and_biases()

elif dataset == 'nonlinear':
  inputs, targets = generate_nonlinear_dataset(n_samples=1000)
  learning_rate, epochs = 0.1, 10000
  method = 'mbgd'
  print(f"Training Nonlinear, learning rate: {learning_rate}, epochs: {epochs}, method: {method}")
  initial_layers = 1
  max_layers = 10
  mlp = MultiLayerPerceptron(layer_sizes=[1], dataset_type='nonlinear')
  best_model = mlp.incremental_model_training_nonlinear(inputs, targets, initial_layers, max_layers, learning_rate, epochs, method=method)

  # Visualize the predictions
  mse = best_model.mean_squared_error(best_model.predict(inputs), targets)
  print(f"Best Model - MSE: {mse}")
  best_model.visualize_predictions_nonlinear(inputs, targets)
  best_model.plot_weights_and_biases()
```