Deep Learning Assignment 1 Report

姓名：邱逸伦

学号：12013006

Part 1 Perceptron

1.1 Task 1

For this task, I generate the dataset using `np.random.multivariate_normal` function. This method takes three arguments: `mean`, `cov`, and `size`. The `mean` parameter represents the mean values of the distribution, while `cov` denotes the covariance matrix of the distribution. The `size` parameter specifies the number of the data points in the dataset. After generating the data points, they are combined into training and testing datasets. These two distributions are labeled 1 and -1, respectively.

```
mean1 = [0, 0]
cov1 = [[1, 0], [0, 1]]
mean2 = [3, 3]
cov2 = [[1, 0], [0, 1]]

data1 = np.random.multivariate_normal(mean1, cov1, 100)
data2 = np.random.multivariate_normal(mean2, cov2, 100)

train_data = np.concatenate((data1[:80], data2[:80]))
train_labels = np.concatenate((np.ones(80), -np.ones(80)))
test_data = np.concatenate((data1[80:], data2[80:]))
test_labels = np.concatenate((np.ones(20), -np.ones(20)))
```

1.2 Task 2

```
class Perceptron(object):

    def __init__(self, n_inputs, max_epochs=1e2, learning_rate=1e-2):
        self.n_inputs = n_inputs
        self.weights = np.random.normal(0, 1, n_inputs + 1)
        self.max_epochs = max_epochs
        self.lr = learning_rate

    def forward(self, input):
        label = np.dot(self.weights[1:], input) + self.weights[0]
        return label

    def train(self, training_inputs, labels):
        for epoch in range(int(self.max_epochs)):
            for x, y in zip(training_inputs, labels):
                pred = self.forward(x)
                mask = y * pred <= 0
```

```
                self.weights[0] += self.lr * np.sum(y[mask])
                self.weights[1:] += self.lr * np.dot(y[mask], x[mask])

    def test(self, testing_inputs, labels):
        cnt = 0
        for x, y in zip(testing_inputs, labels):
            pred = self.forward(x)
            pred = np.where(pred >= 0, 1, -1)
            if int(pred) == int(y):
                cnt += 1
        return cnt
```

The `Perceptron` class defines four methods:

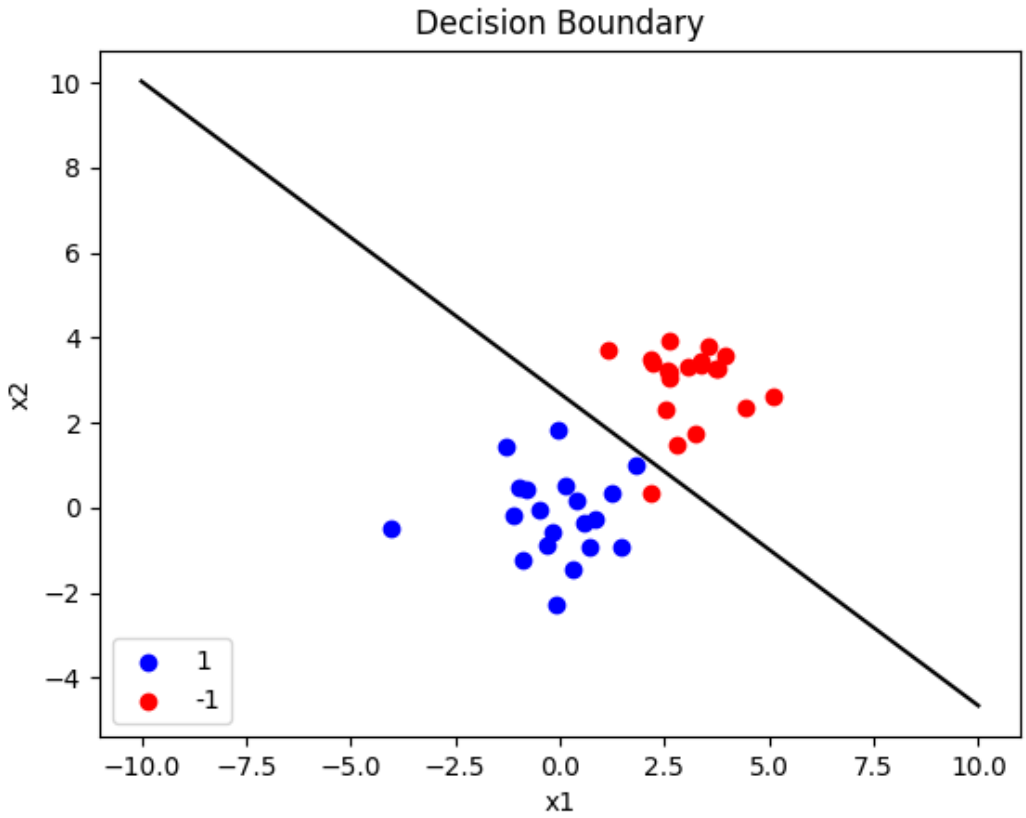
- `__init__`: initialize the class by setting the values of the number of inputs, the maximum epoch, and the learning rate.
- `forward`: compute the predicted result using the formula $\hat{y} = w^T x + b$, w is the weight of the perceptron, x is the input to the perceptron, and b is the bias.
- `train`: train the model by updating w and b based on their previous values and computed results.
- `test`: test the results by computing the number of the correct prediction.

1.3 Task 3

Based on the codes above, I tried to compute the classification accuracy on the test set 10 times. The numbers of correct predictions are 39, 40, 39, 40, 38, 37, 40, 40, 38, and 39, respectively. The average classification accuracy on the test set is **97.75%**.

```
TEST 1:
The number of data points is 40, the number of correct predictions is 39, the classification accuracy is 0.975
TEST 2:
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 3:
The number of data points is 40, the number of correct predictions is 39, the classification accuracy is 0.975
TEST 4:
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 5:
The number of data points is 40, the number of correct predictions is 38, the classification accuracy is 0.95
TEST 6:
The number of data points is 40, the number of correct predictions is 37, the classification accuracy is 0.925
TEST 7:
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 8:
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 9:
The number of data points is 40, the number of correct predictions is 38, the classification accuracy is 0.95
TEST 10:
The number of data points is 40, the number of correct predictions is 39, the classification accuracy is 0.975
```

The result shows a high classification accuracy for our perceptron on the test set.



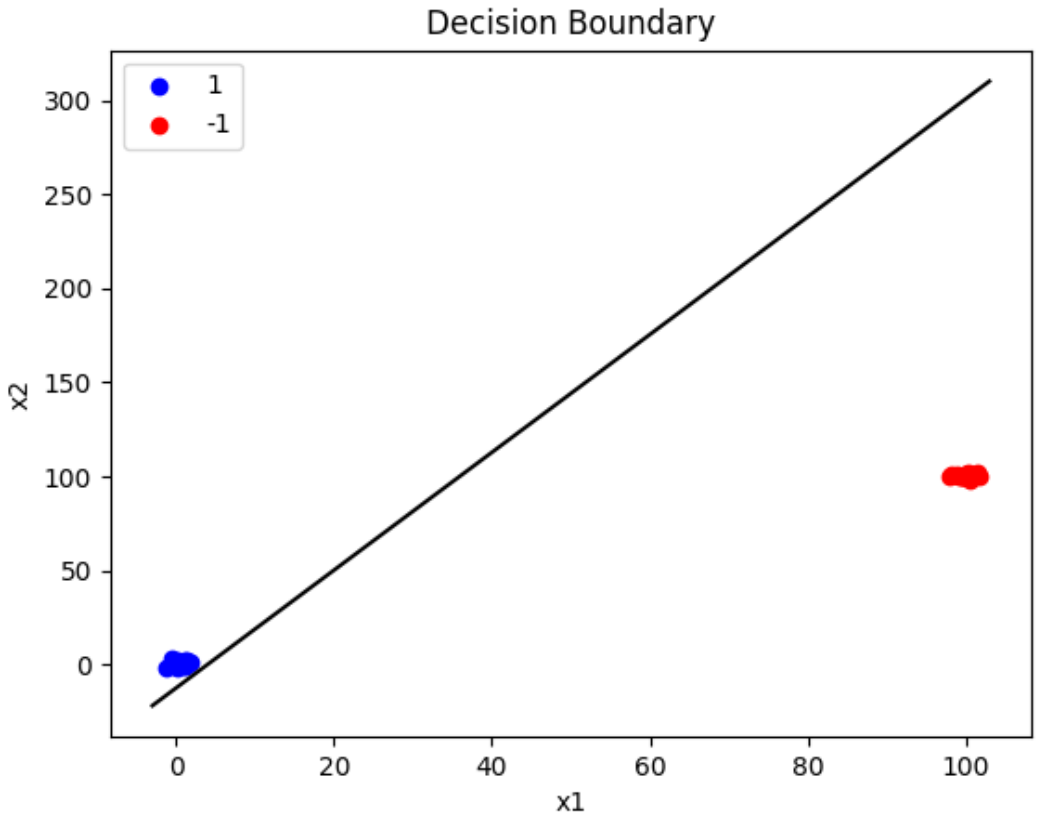
1.4 Task 4

Experiment 1: Mean: [0, 0] and [100, 100] | Covariance Matrix: [[1, 0], [0, 1]] and [[1, 0], [0, 1]]

Average Classification Accuracy: **99.5%**

```
TEST 1
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 2
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 3
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 4
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 5
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 6
The number of data points is 40, the number of correct predictions is 39, the classification accuracy is 0.975
TEST 7
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 8
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 9
The number of data points is 40, the number of correct predictions is 39, the classification accuracy is 0.975
TEST 10
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
```

The classification accuracy is higher than the baseline. This is because these two distributions have very different mean values, so they can be easier distinguished by the perceptron since the data points belonging to a distribution will be far away from the data points belonging to another distribution.

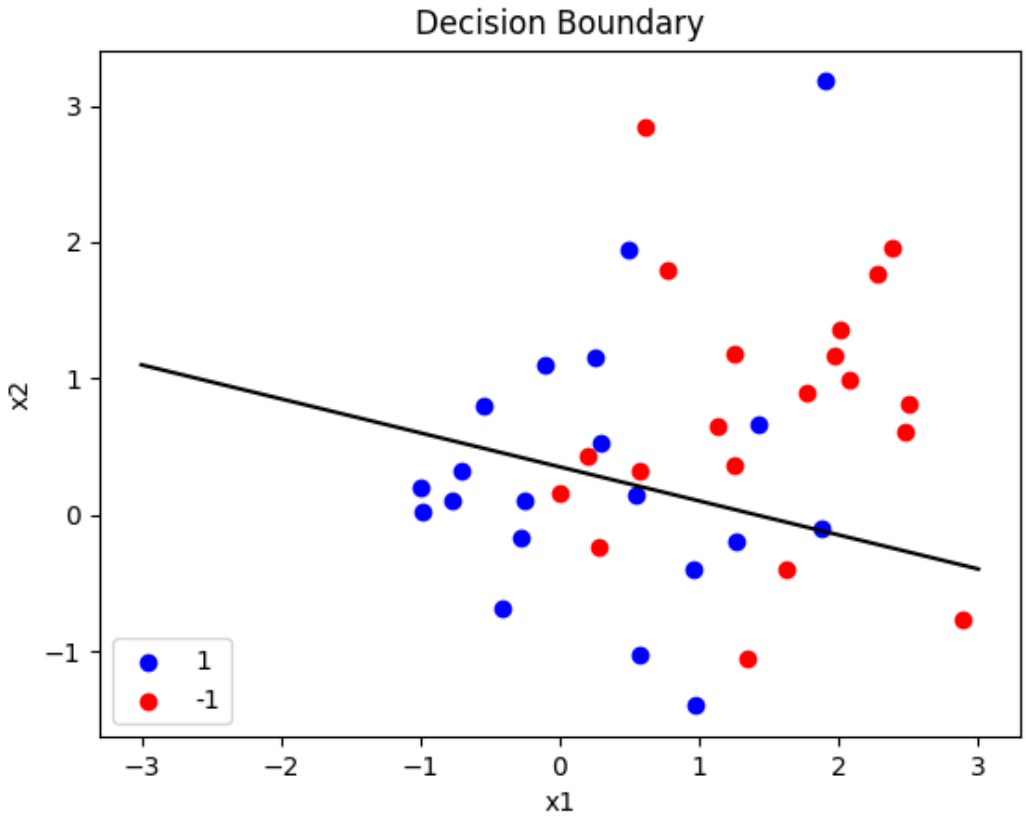


Experiment 2: Mean: [0, 0] and [1, 1] | Covariance Matrix: [[1, 0], [0, 1]] and [[1, 0], [0, 1]]

Average Classification Accuracy: **64%**

```
TEST 1
The number of data points is 40, the number of correct predictions is 33, the classification accuracy is 0.825
TEST 2
The number of data points is 40, the number of correct predictions is 29, the classification accuracy is 0.725
TEST 3
The number of data points is 40, the number of correct predictions is 27, the classification accuracy is 0.675
TEST 4
The number of data points is 40, the number of correct predictions is 33, the classification accuracy is 0.825
TEST 5
The number of data points is 40, the number of correct predictions is 25, the classification accuracy is 0.625
TEST 6
The number of data points is 40, the number of correct predictions is 10, the classification accuracy is 0.25
TEST 7
The number of data points is 40, the number of correct predictions is 28, the classification accuracy is 0.7
TEST 8
The number of data points is 40, the number of correct predictions is 28, the classification accuracy is 0.7
TEST 9
The number of data points is 40, the number of correct predictions is 19, the classification accuracy is 0.475
TEST 10
The number of data points is 40, the number of correct predictions is 24, the classification accuracy is 0.6
```

The classification accuracy is lower than the baseline. This is because the means of the two distributions are too close to determine a curve in dimension 2 to distinguish the data points between these distributions. The figure can also show the reason of such case.

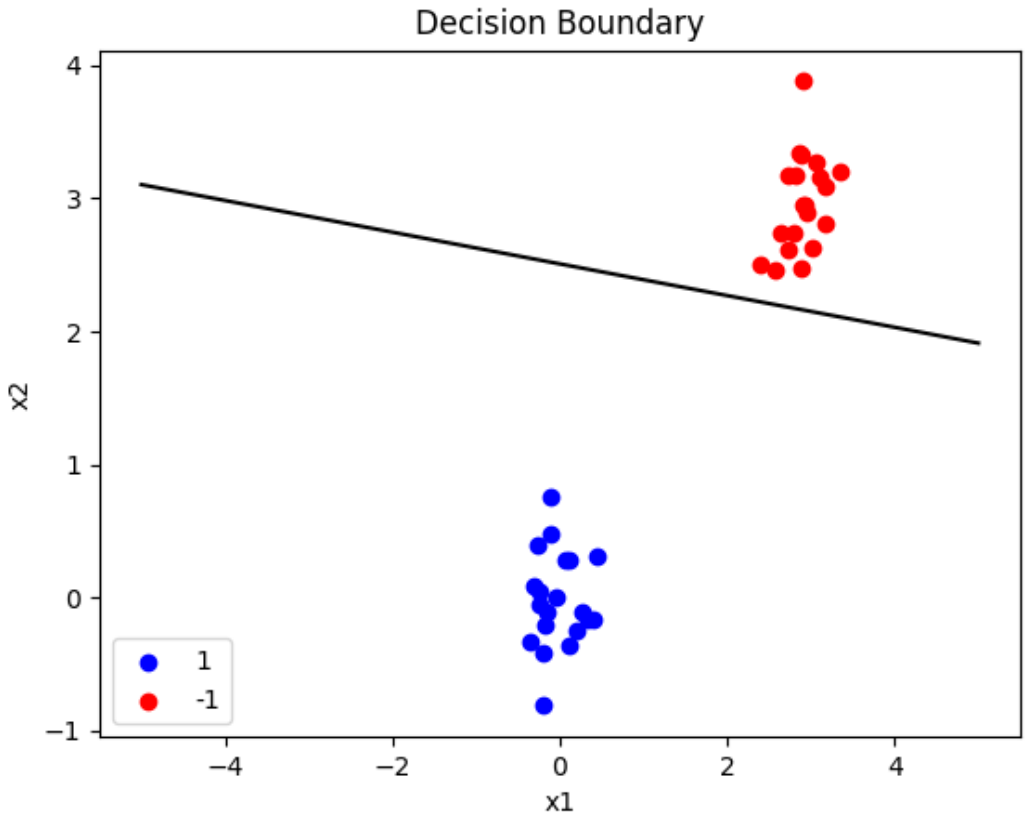


Experiment 3: Mean: [0, 0] and [3, 3] | Covariance Matrix: [[0.1, 0], [0, 0.1]] and [[0.1, 0], [0, 0.1]]

Average Classification Accuracy: **99.5%**

```
TEST 1
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 2
The number of data points is 40, the number of correct predictions is 39, the classification accuracy is 0.975
TEST 3
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 4
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 5
The number of data points is 40, the number of correct predictions is 39, the classification accuracy is 0.975
TEST 6
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 7
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 8
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 9
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
TEST 10
The number of data points is 40, the number of correct predictions is 40, the classification accuracy is 1.0
```

The classification accuracy is higher than the baseline. This is because the distribution shows a higher degree of aggregation of the data points. The more they converge, the easier that a curve can distinguish the set.

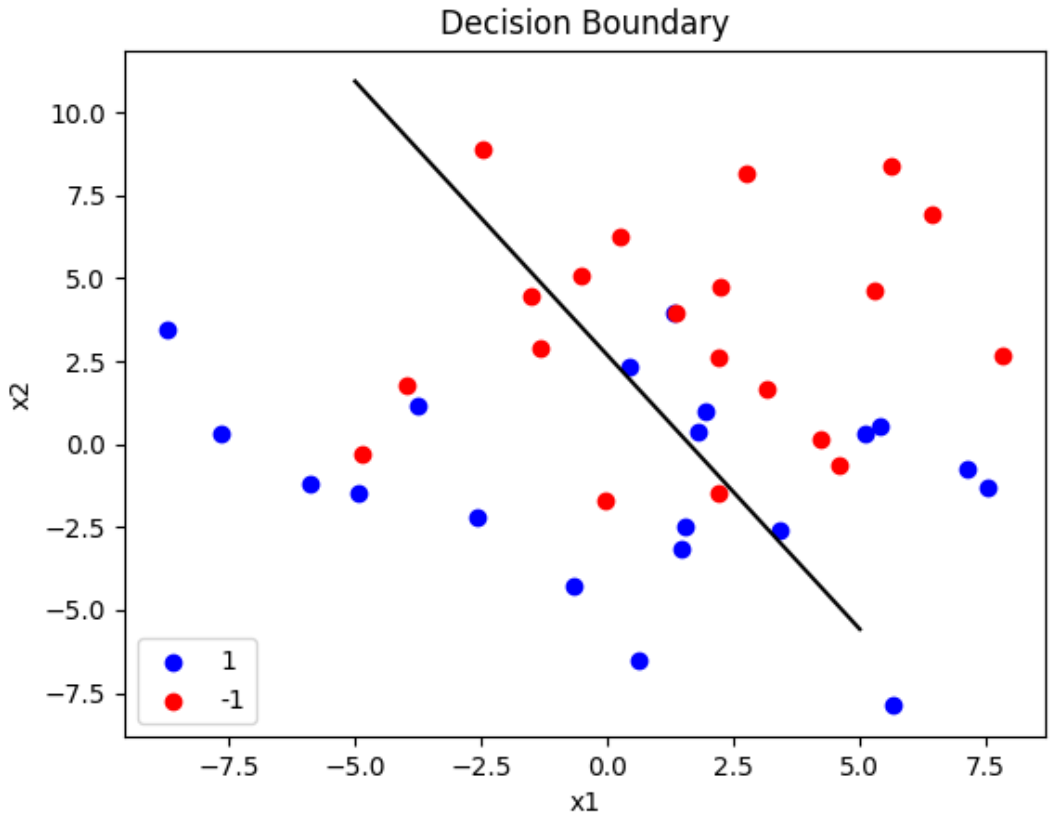


Experiment 4: Mean: [0, 0] and [3, 3] | Covariance Matrix: [[10, 0], [0, 10]] and [[10, 0], [0, 10]]

Average Classification Accuracy: **61%**

```
TEST 1
The number of data points is 40, the number of correct predictions is 25, the classification accuracy is 0.625
TEST 2
The number of data points is 40, the number of correct predictions is 30, the classification accuracy is 0.75
TEST 3
The number of data points is 40, the number of correct predictions is 24, the classification accuracy is 0.6
TEST 4
The number of data points is 40, the number of correct predictions is 27, the classification accuracy is 0.675
TEST 5
The number of data points is 40, the number of correct predictions is 30, the classification accuracy is 0.75
TEST 6
The number of data points is 40, the number of correct predictions is 21, the classification accuracy is 0.525
TEST 7
The number of data points is 40, the number of correct predictions is 22, the classification accuracy is 0.55
TEST 8
The number of data points is 40, the number of correct predictions is 28, the classification accuracy is 0.7
TEST 9
The number of data points is 40, the number of correct predictions is 17, the classification accuracy is 0.425
TEST 10
The number of data points is 40, the number of correct predictions is 20, the classification accuracy is 0.5
```

The classification accuracy is lower than the baseline. This is because a high variance will result in some outliers. They will enhance the difficulty for the curve to distinguish the data points. The figure below shows that the data points are less regular due to high variance.



Part 2 Multi-Layer Perceptron

2.1 Task 1

The classes in the `modules.py` will have both `forward` and `backward` methods, the size of the input to the `forward` method is (batch_size, input_size) and the size of the output to the `backward` method is (batch_size, output_size).

```
class Linear(object):
    def __init__(self, in_features, out_features):
        self.params = {'weight': np.random.normal(loc=0, scale=0.0001, size=
(in_features, out_features)),
                       'bias': np.zeros(out_features)}
        self.grads = {'weight': np.zeros_like(self.params['weight']),
                      'bias': np.zeros_like(self.params['bias'])}
        self.x = None

    def forward(self, x):
        out = np.dot(x, self.params['weight']) + self.params['bias']
        self.x = x
        return out

    def backward(self, dout):
        n = dout.shape[0]
        dx = np.dot(dout, self.params['weight'].T)
        self.grads['weight'] = np.dot(self.x.T, dout) / n
        self.grads['bias'] = np.sum(dout, axis=0) / n
```

```
        return dx
```

The `Linear` class has a `params` parameter to store the values of weight and bias and a `grads` parameter to store the gradients of weight and bias. Parameter `x` is updated in the `forward` method and used in the `backward` method to compute the gradients.

```
class ReLU(object):
    def __init__(self):
        self.x = None

    def forward(self, x):
        self.x = x
        out = np.maximum(0, x)
        return out

    def backward(self, dout):
        dx = dout * (self.x > 0)
        return dx
```

The `ReLU` class acts as an activation function, it will compute the output result using $y = max(0, x)$ in the `forward` method and store the input in the parameter `x`. Then `x` is used in the `backward` method to obtain the gradient.

```
class SoftMax(object):
    def __init__(self):
        self.out = None

    def forward(self, x):
        exp_vals = np.exp(x - np.max(x, axis=1, keepdims=True))
        self.out = exp_vals / np.sum(exp_vals, axis=1, keepdims=True)
        return self.out

    def backward(self, dout):
        dx = -self.out[:, :, None] * self.out[:, None, :]
        dx[:, np.arange(self.out.shape[1]), np.arange(self.out.shape[1])] =
self.out * (1 - self.out)
        d_result = np.matmul(dout[:, None, :], dx)
        d_result = np.squeeze(d_result, axis=0)
        return d_result
```

The `Softmax` class acts as an activation function in the output layer to obtain the probability space of the output by the `forward` method using $softmax(z_i) = \frac{e^{z_i}}{\Sigma_{c=1}^{C} e^{z_c}}$, and the output is recorded in the `output` parameter. In the `backward` method, the gradient is computed by the gradient `dout` passed by the loss function and `output`.

```
class CrossEntropy(object):
    def __init__(self):
        self.pd = None
        self.gt = None
        self.batch_size = None

    def forward(self, pd, gt):
```

```
        self.batch_size = pd.shape[0]
        self.pd = pd
        self.gt = gt
        eps = 1e-9
        loss = - np.sum(np.multiply(gt, np.log(pd + eps))) / self.batch_size
        return loss

    def backward(self):
        eps = 1e-9
        d_loss = - self.gt / (self.pd + eps)
        return d_loss
```

The `CrossEntropy` class acts as a loss function computer the error between the prediction and the ground truth. The loss is computed in the `forward` method using $L = \frac{1}{N}\Sigma_i - [y_i * log(p_i) + (1 - y_i) * log(1 - p_i)]$, also the values of prediction and the ground truth are stored in `pd` and `gt`, respectively. The `backward` method computes the gradient of the loss function using `pd` and `gt`, a `eps` parameter is also applied.

```
class MLP(object):

    def __init__(self, n_inputs, n_hidden, n_classes):
        self.layers = []
        in_features = n_inputs
        for out_features in n_hidden:
            self.layers.append(Linear(in_features, out_features))
            self.layers.append(ReLU())
            in_features = out_features
        self.layers.append(Linear(in_features, n_classes))
        self.layers.append(SoftMax())

    def forward(self, x):
        out = x
        for layer in self.layers:
            out = layer.forward(out)
        return out

    def backward(self, dout):
        for layer in reversed(self.layers):
            dout = layer.backward(dout)

    def update(self, lr):
        for layer in self.layers:
            if isinstance(layer, Linear):
                layer.params['weight'] -= lr * layer.grads['weight']
                layer.params['bias'] -= lr * layer.grads['bias']
                layer.grads['weight'] = np.zeros_like(layer.params['weight'])
                layer.grads['bias'] = np.zeros_like(layer.params['bias'])
```

The `MLP` class consists of `Linear` layers, `ReLU` layers, and `SoftMax` layer. All these layers are appended to the list `layers`. The `forward` method is used to compute the result obtained by each layer and finally get the prediction, and the `backward` method is used to compute the gradients and store all the gradients in each layer class. The `update` method is used to update

the weight and bias in each `Linear` class using the learning rate and the gradients computed in the `backward` method.

2.2 Task 2

The `train` method is defined below.

```
def train():
    dnn_hidden_units = FLAGS.dnn_hidden_units
    n_hidden = dnn_hidden_units.split(",")
    n_hidden = [int(x) for x in n_hidden]
    learning_rate = FLAGS.learning_rate
    max_steps = FLAGS.max_steps
    eval_freq = FLAGS.eval_freq
    optimizer = FLAGS.optimizer

    inputs, labels = datasets.make_moons(n_samples=1000, shuffle=True)

    mlp = MLP(2, n_hidden, 2)
    if optimizer.lower() == 'bgd':
        optimizer = BGD(mlp)
    elif optimizer.lower() == 'sgd':
        optimizer = SGD(mlp)
    else:
        raise TypeError
    optimizer.optimize(inputs, labels, learning_rate, max_steps, eval_freq)
```

In this method, I first obtain the parameters from `FLAGS` and do something with them. After that, I generate the data using `make_moons` method in `scikit-learn` library to get 1000 data points. Finally, the program will choose to use BGD (Batch Gradient Descent) or SGD (Stochastic Gradient Descent) to optimize the model.

The `BGD` and `SGD` are both defined in `optimizer.py`. The implementation of `BGD` class is shown below.

```
class BGD(object): # Eliminate some print info
    def __init__(self, mlp):
        self.mlp = mlp

    def optimize(self, inputs, labels, lr, max_epochs, eval_freq):
        batch_size = len(inputs)
        x_train, x_test, y_train, y_test =
model_selection.train_test_split(inputs, labels, shuffle=True, test_size=0.2)
        y_train, y_test = one_hot(y_train), one_hot(y_test)
        num_batches = int(np.ceil(len(x_train) / batch_size))
        criterion = CrossEntropy()
        for epoch in range(max_epochs):
            for i in range(num_batches):
                start_idx = i * batch_size
                end_idx = min((i + 1) * batch_size, len(x_train))
                x_batch = x_train[start_idx:end_idx]
                y_batch = y_train[start_idx:end_idx]
                pd_batch = self.mlp.forward(x_batch)
                loss = criterion.forward(pd_batch, y_batch)
```

```
                d_loss = criterion.backward()
                self.mlp.backward(d_loss)
                self.mlp.update(lr)
                acc = accuracy(pd_batch, y_batch)
            pd_test = self.mlp.forward(x_test)
            test_acc = accuracy(pd_test, y_test)
```

The `BGD` class is initialized using the `MLP` model. The `optiimize` method requires some parameters including `inputs` which is the input to the model, `labels` which is the label to each input, `lr` which is the learning rate, `max_epochs` which is the maximum epochs the optimizer can have, and `eval_freq` which denotes the frequency to print related information. The `batch_size` is the size of the batch during training, the default value of the batch size is the size of the whole training set(the method can also support the MiniBatch method). In the `optimize` method, I first split the whole dataset into a training dataset and a testing dataset using the `train_test_split` method in `scikit-learn` library. Then the `y_train` and the `y_test` will be converted into one-hot vector form using the `one_hot` method. Then `num_batches` is computed as the number of batches in one epoch. The `criterion` is the loss function, `CrossEntropy` is used here. Then the program will loop until the `epoch` reaches the `max_epochs`. In each epoch, the program will construct a batch of inputs `x_batch` and labels `y_batch`. Then the `x_batch` will be passed to the `mlp.forward` method to get a batch of prediction `pd_batch`. Then the prediction `pd_batch` and the ground truth `y_batch` will be passed into the `criterion.backward` method to obtain the loss, then the program will call the `criterion.backward` method to get the gradient `d_loss` of the loss function. Then the `d_loss` is passed into the method `mlp.backward` to obtain the gradients of all layers in the `MLP`. Then the program calls the method `mlp.update` to update the weights and biases. Finally, I can obtain the accuracy between `pd_batch` and `y_batch`. For testing, I obtain the `y_test` from `x_test`, and the accuracy in testing can also be computed.

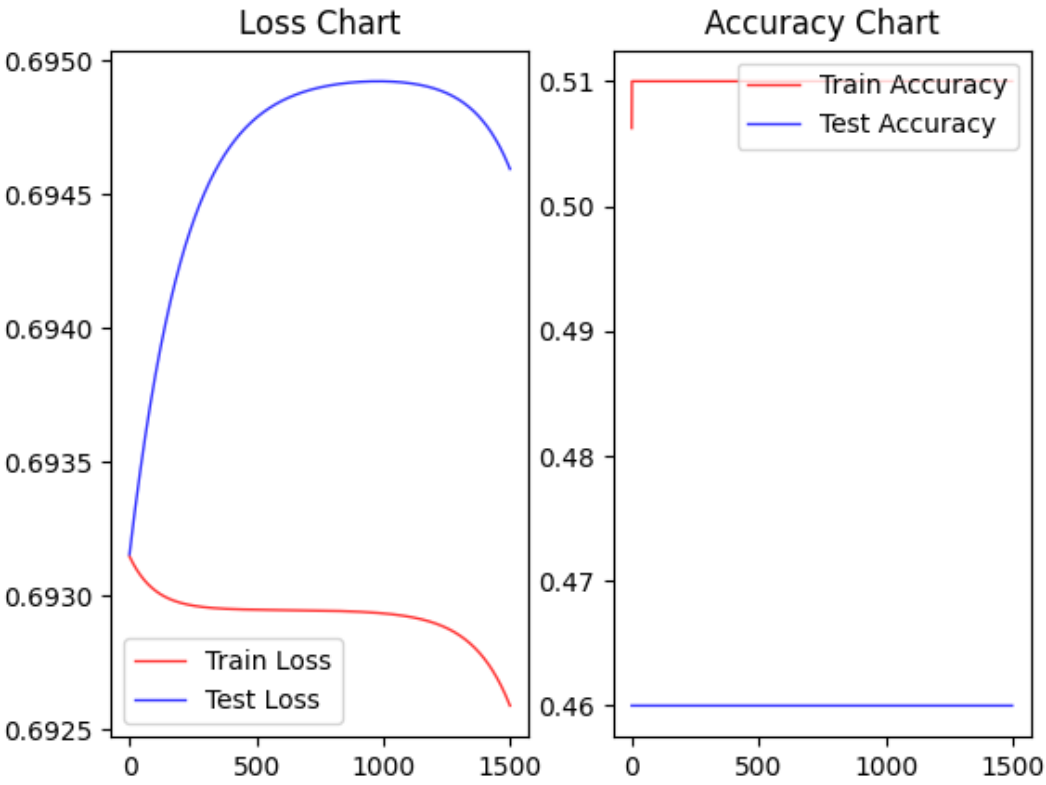
The method implementations of `one_not` and `accuracy` are shown below.

```
def one_hot(input, num_classes=2):
    return np.eye(num_classes)[input]
```

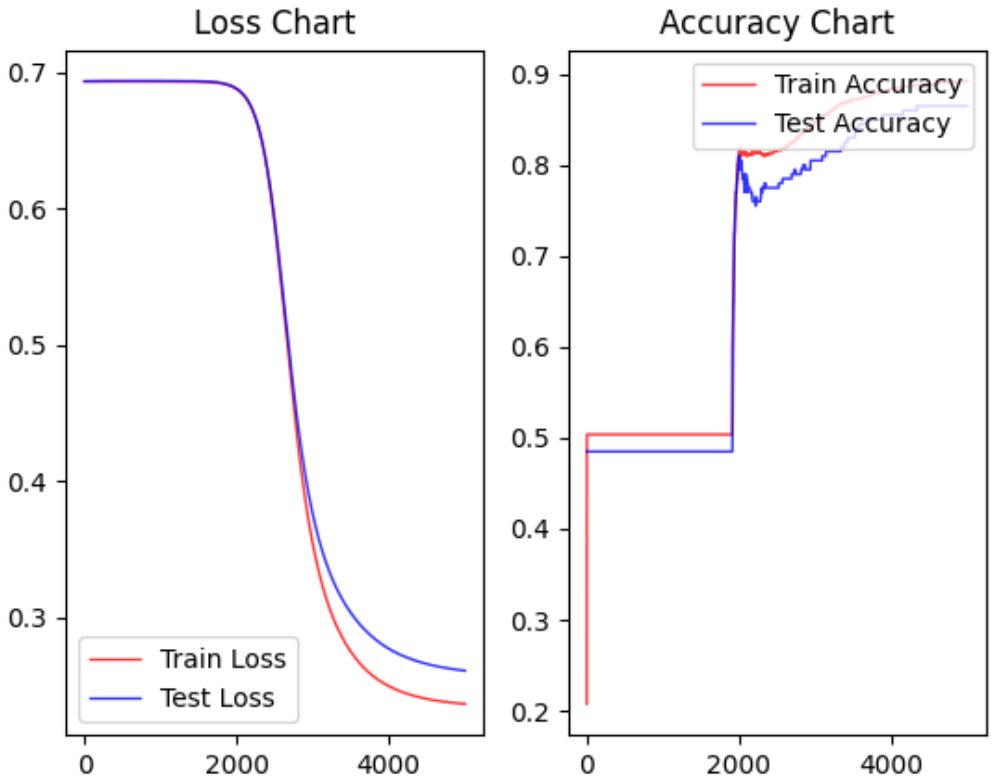
```
def accuracy(predictions, targets):
    predicted_labels = np.argmax(predictions, axis=1)
    true_labels = np.argmax(targets, axis=1)
    return np.mean(predicted_labels == true_labels)
```

2.3 Task 3

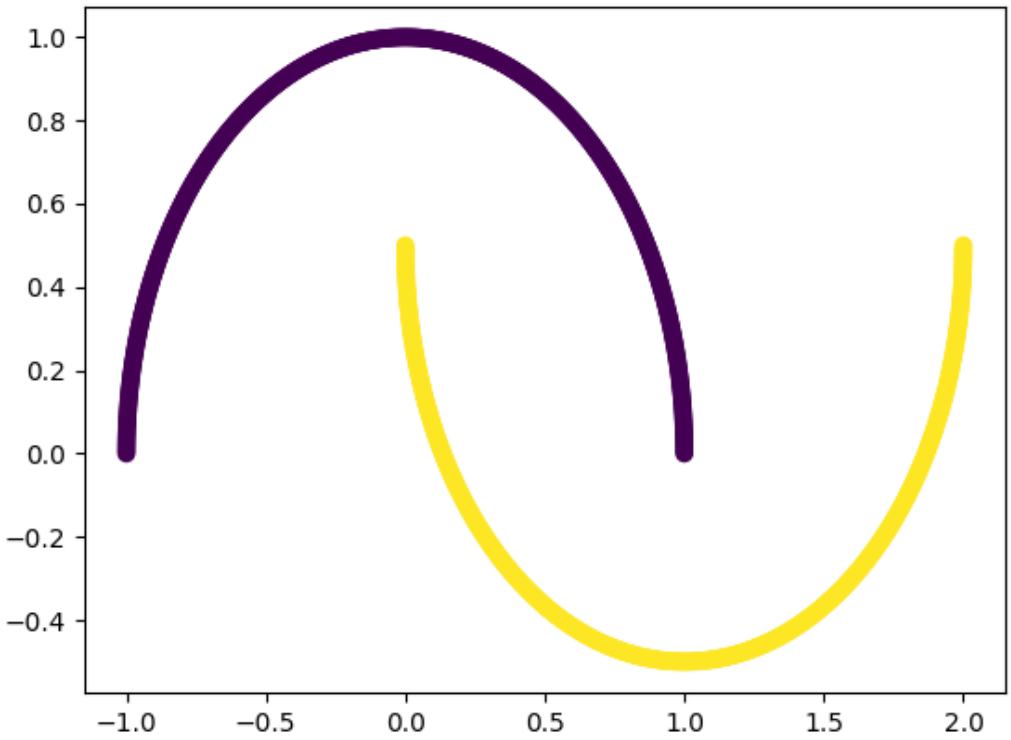
Since I don't set some breakpoints during training, the program will loop completely 1500 times using the default values of the parameters. However, the result is not good enough, the training accuracy and the testing accuracy are almost unchanged and low.



After some analysis, I think using the whole training dataset will result in a very small change in both weight and bias in the `Linear` layer, so the predicted result will be almost unchanged. But if I increase the number of epochs to 5000 during training, the accuracy will have a sudden rise in about epoch 2000 as shown below.

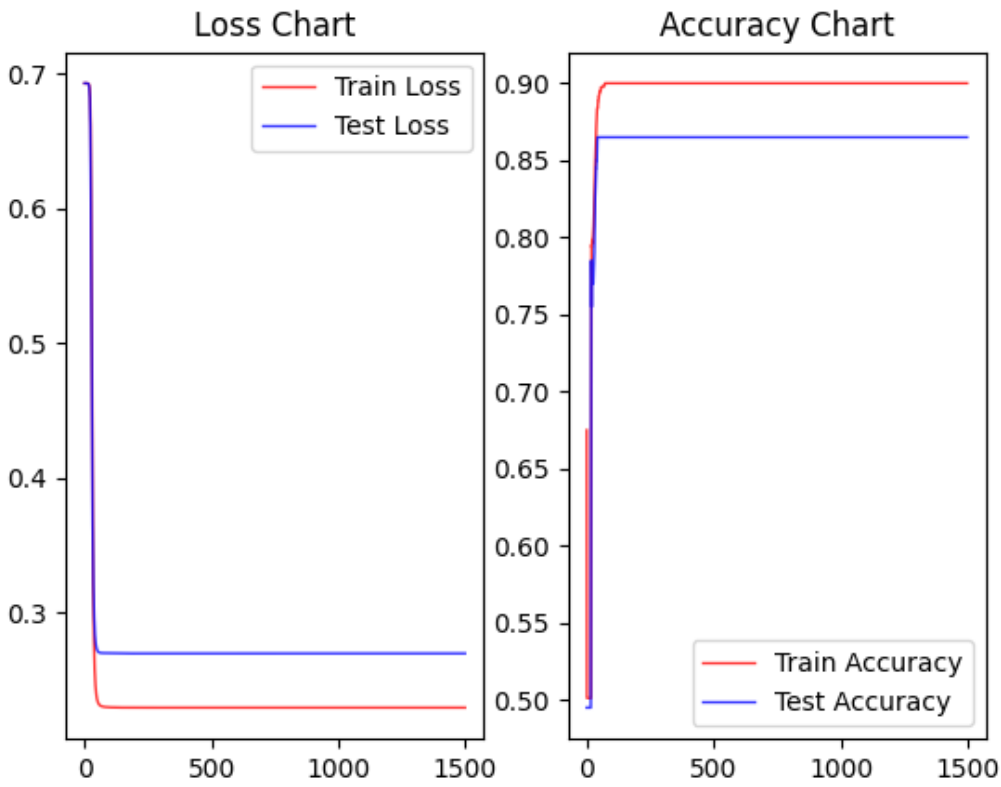


I think this is because there are only several data points that will contribute to the prediction result since the distribution of the dataset is shown below.

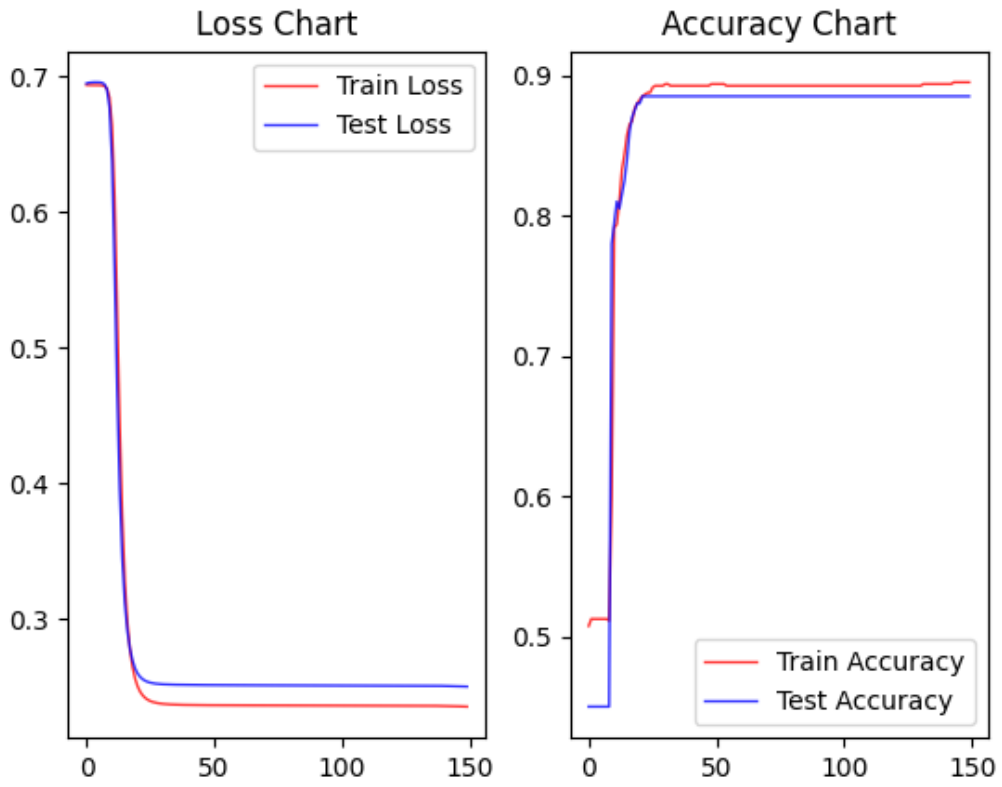


The change needs to be accumulated to a threshold to perform better.

Increasing the learning rate to 1 or other values can also make the same effect to perform better as shown below.



However, the MiniBatch method can perform better with the same learning rate and fewer epochs. I think this is because the data points that contribute to the classification will likely take control during training in a low epoch. The figure below shows the result with batch size 4 and epoch 150. The best train loss equals 0.23568745523087017, the best train accuracy equals 0.895 the best test loss equals 0.2504730121080941, and the best test accuracy equals 0.875.



Part 3 Stochastic Gradient Descent

3.1 Task 1

The choice between `SGD` and `BGD` is determined by a command line argument `optimizer`. If the value of the `optimizer` is "SGD", the program will use SGD optimizer; if the value is "BGD", the program will use BGD optimizer; otherwise, the program will raise a `TypeError`.

The implementation of `SGD` class is shown below.

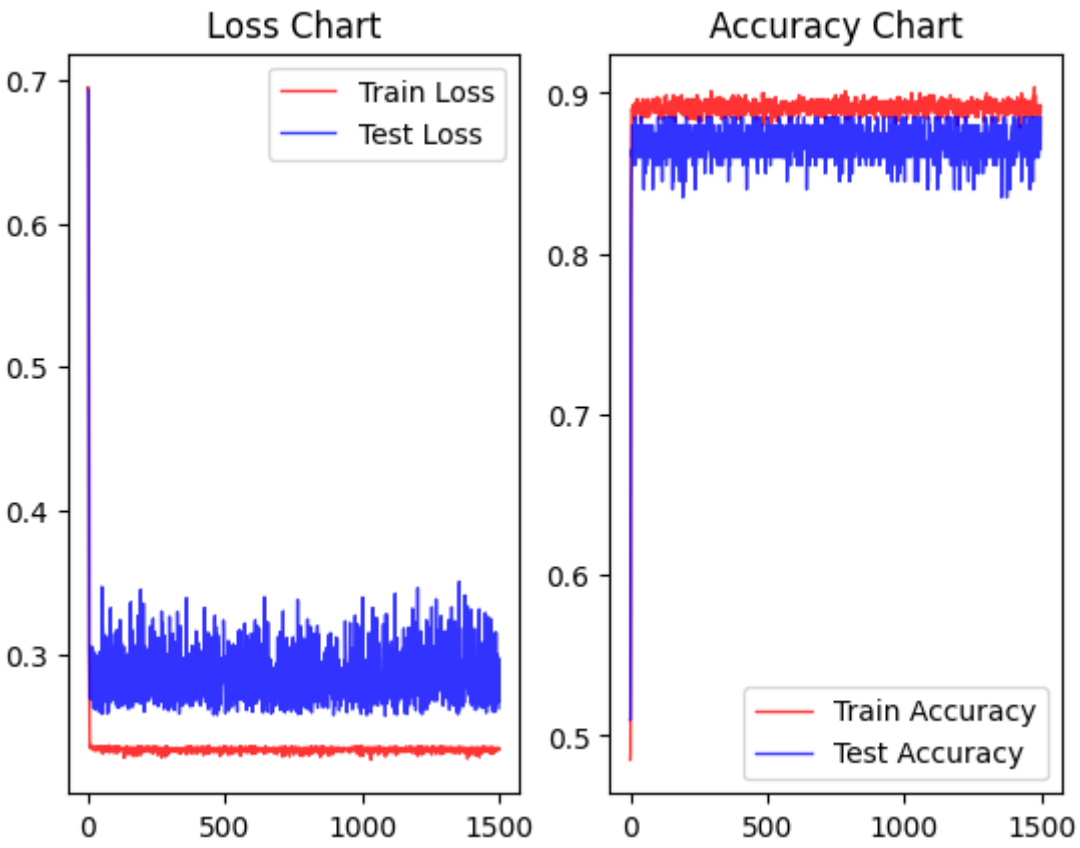
```
class SGD(object):
    def __init__(self, mlp):
        self.mlp = mlp

    def optimize(self, inputs, labels, lr, max_epochs, eval_freq):
        x_train, x_test, y_train, y_test =
model_selection.train_test_split(inputs, labels, shuffle=True, test_size=0.2)
        y_train, y_test = one_hot(y_train), one_hot(y_test)
        criterion = CrossEntropy()
        n_samples = x_train.shape[0]
        for epoch in range(max_epochs):
            shuffled_indices = np.random.permutation(n_samples)
            for i in range(n_samples):
```

```
                idx = shuffled_indices[i]
                x = x_train[idx]
                y = y_train[idx]
                x, y = x[np.newaxis, :], y[np.newaxis, :]
                pd = self.mlp.forward(x)
                loss = criterion.forward(pd, y)
                d_loss = criterion.backward()
                self.mlp.backward(d_loss)
                self.mlp.update(lr)
                acc = accuracy(pd, y)
            pd_test = self.mlp.forward(x_test)
            test_acc = accuracy(pd_test, y_test)
```

The `optimize` method applies the same operation to the dataset as `BGD`. But the indexes are shuffled by `np.random.permutation`. After that, the program will choose only one data point and use it to train the model and update the weight and the bias.

3.2 Task 2

Using the default values of the parameters, the model converges quickly, but the results may slowly deviate over time.



Best Train Loss: 0.22639556090128463 | Best Train Accuracy: 0.90375
Best Test Loss: 0.2569323720542476 | Best Test Accuracy: 0.885

Part 4 Instruction

The Jupyter Notebook should be run relying on `optimizer.py` and `mlp_numpy.py`. Other `.py` files can be run directly. Some parameters can be changed directly to obtain different results.