SUSTech CS324 Deep Learning
Report for Assignment 1

Student ID: 12312710
Name: 胥熠/Brighton
Date: 2025 年 10 月 17 日

目录

1 Part 1: Perceptron

1.1 Task 1:

Generate a dataset of points in R^2. To do this, define two Gaussian distributions and sample 100 points from each. Keep 80% points per distribution as the training (160 in total), 20% for the test (40 in total).

We can generate the dataset using numpy as follows:

```
import numpy as np
class1_data = np.random.multivariate_normal(mean1, cov1, n_samples)
class2_data = np.random.multivariate_normal(mean2, cov2, n_samples)
```

Where mean1 and mean2 are the means of the two Gaussian distributions, cov1 and cov2 are their covariance matrices, and n_samples is the number of samples to draw from each distribution (100 in this case).

To divide the dataset into training and test sets, we can use a simple indexing method:

```
class1_labels = np.ones(n_samples)
class2_labels = np.zeros(n_samples)
```

1.2 Task 2:

Perceptron is a simple linear classifier. It takes multiple inputs, applies weights to them, sums them up, and passes the result through an activation function (usually a step function) to produce a binary output (0 or 1).

1.2.1 Mathematical Formulation:

The perceptron decision can be mathematically formulated as follows:

$$f(x) = sign(w \times x + b) \tag{1}$$

$$sign(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases} \tag{2}$$

Where x is the input vector, w is the weight vector, b is the bias term, and sign is the activation function that outputs 1 if the input is non-negative and 0 otherwise.

1.2.2 Loss Function:

Let y_i be the true label (0 or 1) and $f(x_i)$ be the predicted label for the i-th training example. It's obvious to see that when $y_i * f(x_i) < 0$, the prediction is incorrect.

Thus, the perceptron loss function can be defined as:

$$L = -\frac{1}{\|w\|}\sum_{i=1}^{N} y_i(w \cdot x_i + b) \quad \text{when } y_i(w \cdot x_i + b) < 0 \tag{3}$$

1.2.3 Gradient Descent:

To minimize the loss function, we can use gradient descent to update the weights and bias. The gradients of the loss function with respect to the weights and bias are given by:

$$\frac{\partial L}{\partial w} = -\frac{1}{\|w\|}\sum_{i=1}^{N} y_i x_i \quad \text{when } y_i(w \cdot x_i + b) < 0 \tag{4}$$

$$\frac{\partial L}{\partial b} = -\frac{1}{\|w\|}\sum_{i=1}^{N} y_i \quad \text{when } y_i(w \cdot x_i + b) < 0 \tag{5}$$

So, we can minimize the loss function by updating the weights and bias using gradient descent:

$$w = w + \eta \sum_{i=1}^{N} y_i x_i \tag{6}$$

$$b = b + \eta \sum_{i=1}^{N} y_i \tag{7}$$

where η represents the learning rate (0.01 in this case).
So, the complete algorithm for training a perceptron can be summarized as follows:

```
Initialize weights w and bias b to small random values
For each epoch:
    For each training example (x_i, y_i):
        Compute the output f(x_i) = sign(w * x_i + b)
        If y_i * f(x_i) < 0:  # Misclassified
            Update weights: w = w + 0.01 * y_i * x_i
            Update bias: b = b + 0.01 * y_i
```

1.3 Task 3:

Now we suppose that the dataset is given as:
$dataset_1 : mean = [-2, -2], cov = [[1, 0], [0, 1]], n\_samples = 100$
$dataset_2 : mean = [2, 2], cov = [[1, 0], [0, 1]], n\_samples = 100$
Since the two dataset are very far from each other, this group are linearly separable.

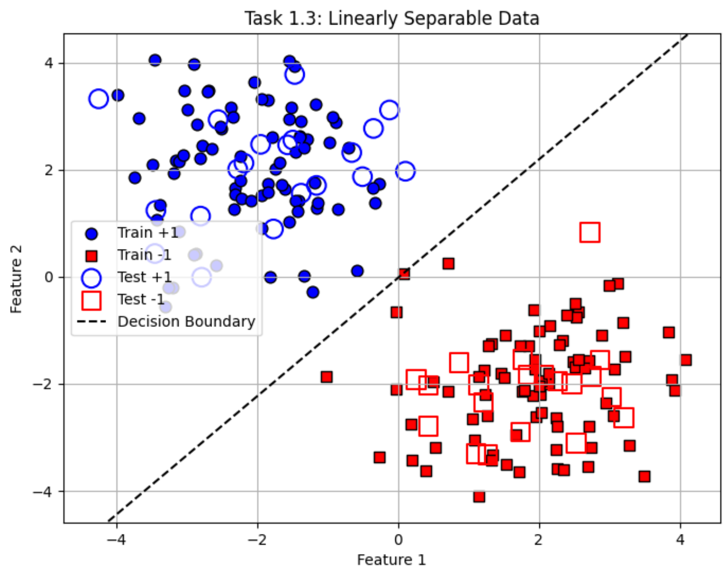

图 1: Linearly Separable Dataset & Perceptron Decision Boundary

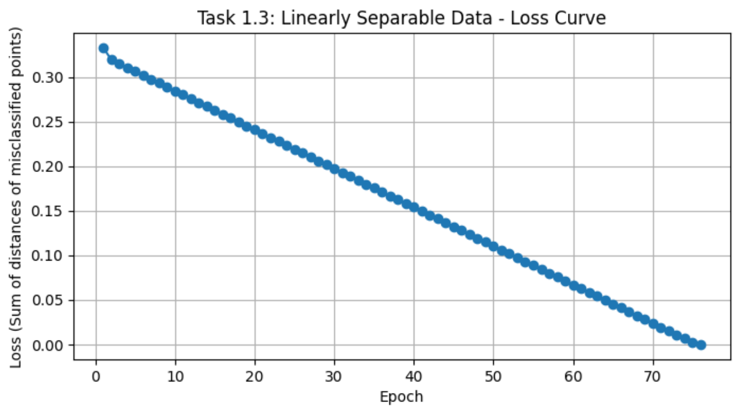

图 2: Perceptron Loss Function in Linearly Separable Dataset

For this dataset, the classification accuracy is 100%. And the loss function converges at the round of 75.

1.4 Task 4:

I design another two group:

- Group 1 (the means are too close):
  - $dataset_1 : mean = [0.5, 0.5], cov = [[1, 0.5], [0.5, 1]], n\_samples = 100$
  - $dataset_2 : mean = [-0.5, -0.5], cov = [[1, 0.5], [0.5, 1]], n\_samples = 100$
- Group 2 (the variances are too high):

- $dataset_1 : mean = [3, 3], cov = [[4, 2], [2, 4]], n\_samples = 100$
- $dataset_2 : mean = [-3, -3], cov = [[4, 2], [2, 4]], n\_samples = 100$

And here are the result:

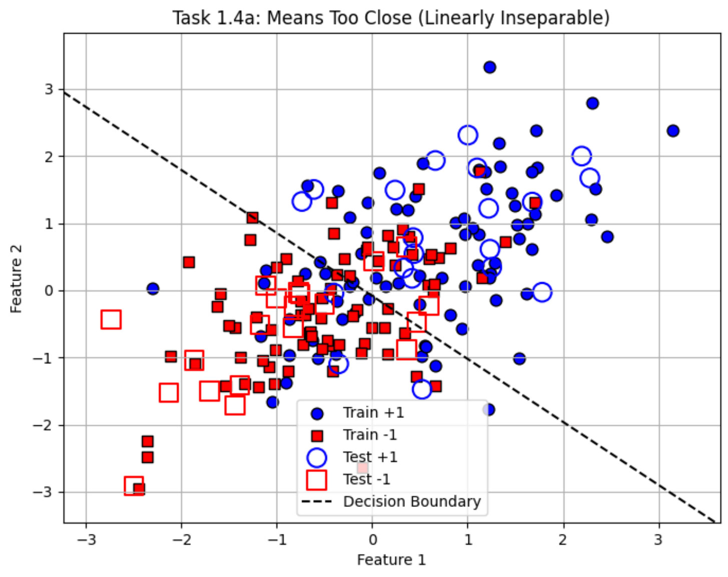

图 3: Group 1 Perceptron Decision Boundary

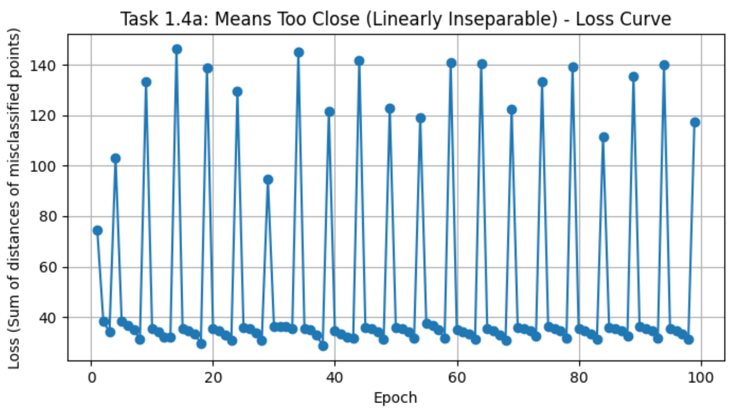

图 4: Perceptron Loss Function in Group 1

We can notice that when the means are too close, the two Gaussian distributions have significant overlap, making the dataset inherently non-linearly separable.

The decision boundary becomes more complex, and the perceptron cannot find a suitable hyperplane to separate the two classes.

The resulting loss curve exhibits high variance and fails to converge, indicating that the model continuously finds misclassified points and adjusts its weights without ever reaching a stable solution.

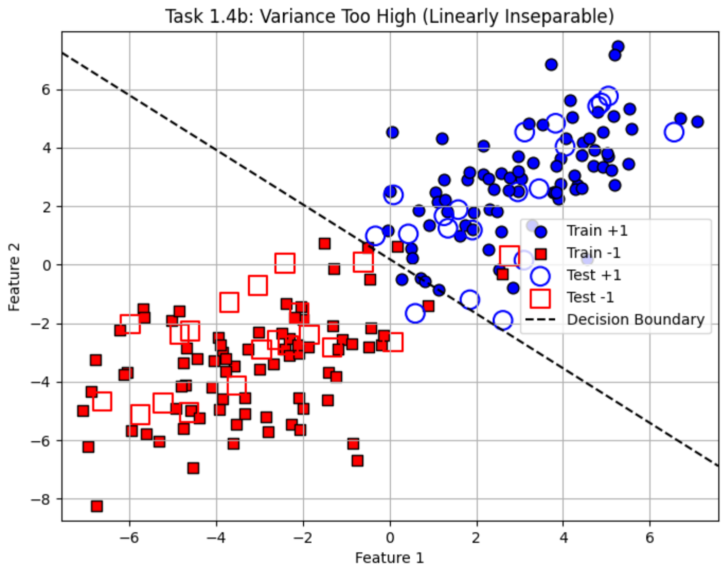

图 5: Group 2 Perceptron Decision Boundary

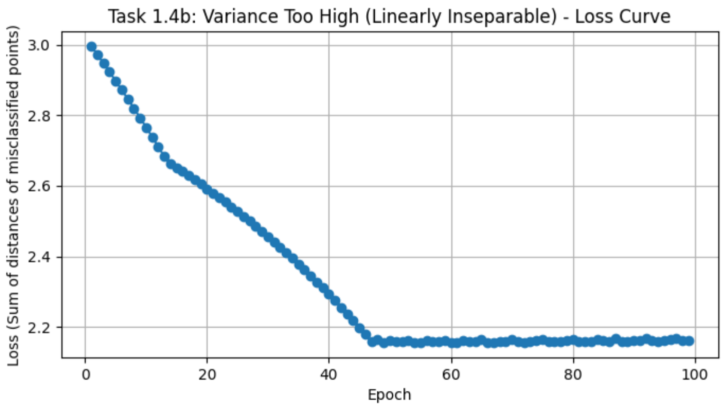

图 6: Perceptron Loss Function in Group 2

Similarly, high variance causes the distributions to spread out and overlap.

While the classes are centered far apart, outliers from each class invade the other's territory. In this case, the perceptron algorithm does converge to a stable decision boundary. However, the loss does not reach zero because a perfect linear separation is impossible. The final boundary represents a 'best effort' compromise, minimizing the number of misclassified points but never eliminating them entirely, hence the accuracy remains below 100

The core reason for the failure of the perceptron is these two cases lies in the fundamental limitation of the perceptron model itself. The perceptron is a linear classifier, which means it can only find a linear decision boundary to separate the classes. When the data is not linearly separable, as in these two cases, the perceptron cannot find an appropriate hyperplane to separate the classes effectively.

To overcome this limitation, we can introduce Multi-Layer Perceptron, which can learn non-linear decision boundaries and handle more complex datasets.

2 Part 2: MLP Implementation

2.1 Task 1:

For the MLP model, I choose the ReLU as the activation function for the hidden layers, softmax for the output layer and cross-entropy as the loss function.

For each layer $l \in \{1, \ldots, N\}$, the forward propagation can be formulated as:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)} \tag{8}$$

where $W^{(L)} \in \mathbb{R}^{n^{(l)} \times n^{(l-1)}}$ and $b^{(l)} \in \mathbb{R}^{n^{(l)}}$ are the weight matrix and bias vector for layer l.

For layer $l \in \{1, \ldots, N-1\}$ (hidden layer), the activation function is ReLU, the formula is as follows:

$$a^{(l)} = ReLU(z^{(l)}) = max(0, z^{(l)}) \tag{9}$$

For the output layer $l = N$, the activation function is softmax, the formula is:

$$a_i^{(N)} = \frac{e^{z_i^{(N)}}}{\sum_j e^{z_j^{(N)}}} \tag{10}$$

Then the cross-entropy loss function is defined as:

$$L = -\sum_{i=1}^{C} y_i \log(a_i^{(N)}) \tag{11}$$

After some experiments on the performance of different hyperparameters, I found that the following structure works well for this task:

- Input Layer: Receives input vectors as $x \in \mathbb{R}^2$
- Hidden Linear Layer 1: $W \in \mathbb{R}^{2\times 20} (Mapping\ x\ to\ R^{(20)}), b \in \mathbb{R}^{20}$, followed by a ReLU activation.
- Hidden Linear Layer 2: $W \in \mathbb{R}^{20\times 20}, b \in \mathbb{R}^{20}$, followed by a ReLU activation.
- Output Linear Layer 3: $W \in \mathbb{R}^{20\times 2}, b \in \mathbb{R}^2$
- Softmax Activation: $a_i^{(3)} = \frac{e^{z_i^{(3)}}}{\sum_j e^{z_j^{(3)}}}$
- Cross-Entropy Loss: $L = -\sum_{i=1}^{C} y_i \log(a_i^{(3)})$

In practical usage in my Python code, this can be simplified as:

```
DNN_HIDDEN_UNITS = '20,20'
```

2.2 Task 2:

When doing the BGD (batch size = full, that is, all mismatched samples), we can observe two different patterns of convergence in the loss function, depending on the random initialization of weights and biases.

If the initial weights and biases are randomly set to lucky values, the loss function could converge into zero and eventually achieve 100% accuracy.

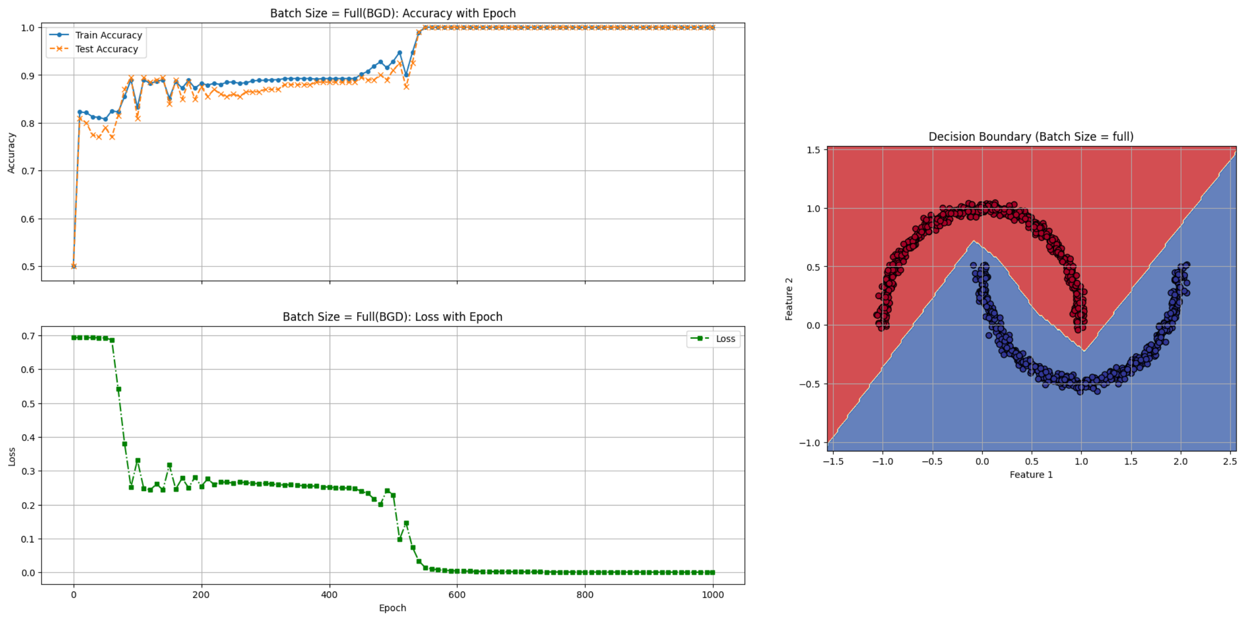

图 7: BGD with Good Initialization

However, if the initial weights and biases are set to unlucky values, the loss function may converge to a small value but not zero, which means that it stuck in a local minimum and cannot achieve 100% accuracy.

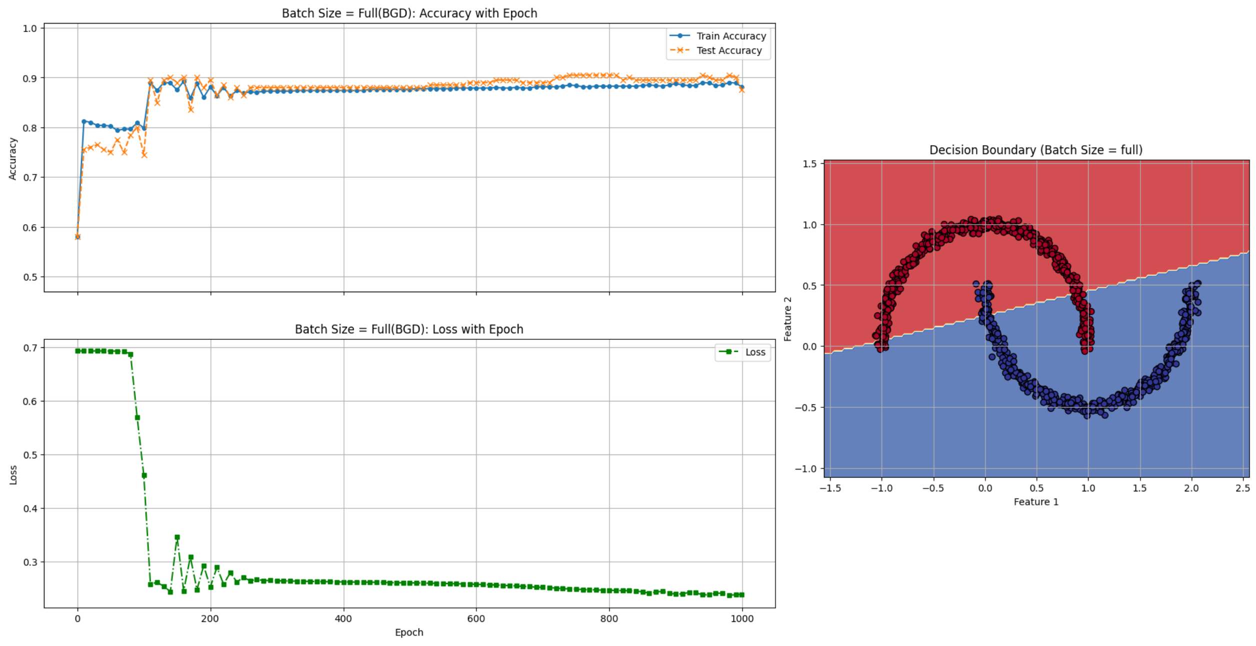

图 8: BGD with Bad Initialization

Obviously, from the decision boundary, we can observe that the model failed to find the pattern of the dataset, it just linearly separated the two classes and rejected to study further.

3 Part 3: Mini-batch SGD Experiment

3.1 Task 1:

In this part of the experiment, I tested several different batch sizes for mini-batch SGD, including batchsize = 1 (SGD), 16, 64.

Also, for the smaller batch sizes, the learning rate should be smaller to ensure the stability of the training process.Here are the learning rates I used for different batch sizes:

$$\begin{bmatrix} \text{Batch Size} & \text{Learning Rate} \\ 1 & 10^{-2} \\ 16 & 10^{-1} \\ 64 & 4 \times 10^{-1} \\ \text{full} & 1 \end{bmatrix} \tag{12}$$

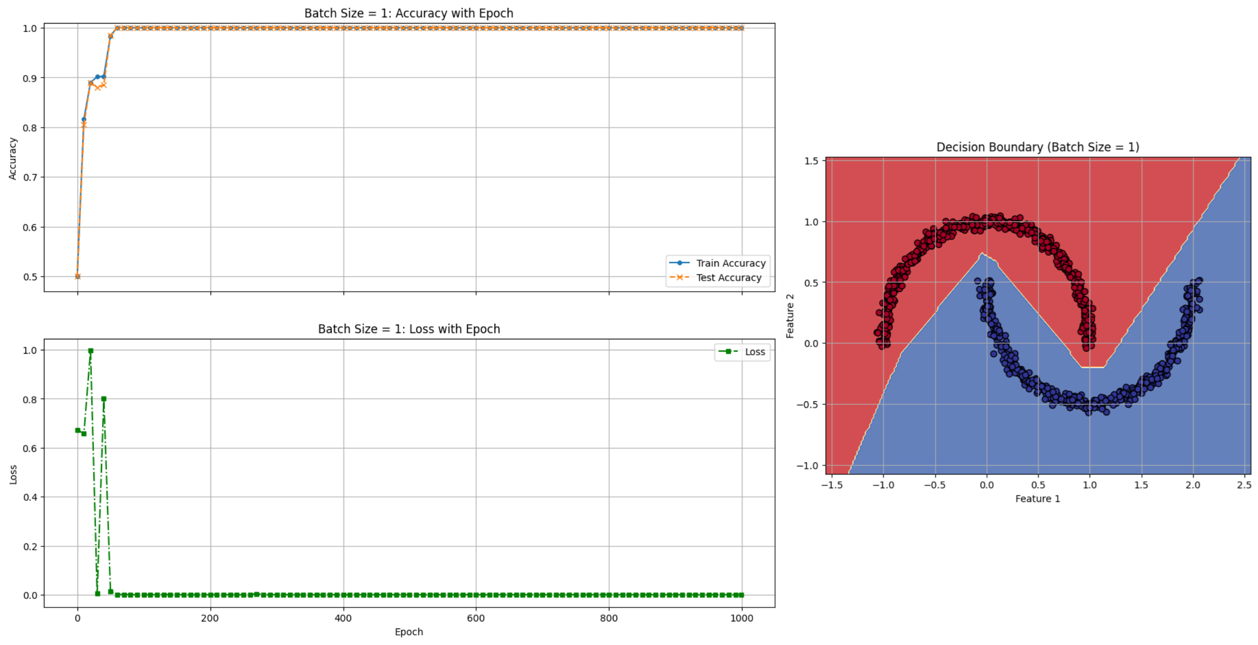

图 9: SGD (batch size = 1)

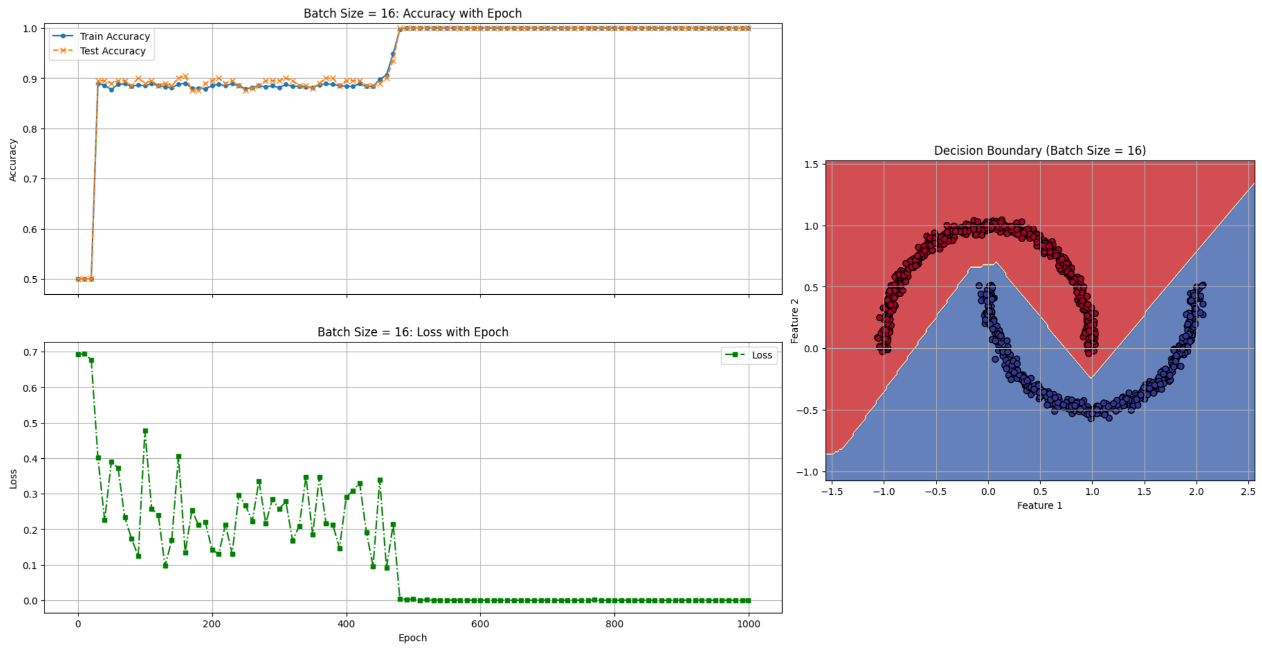

图 10: Mini-batch BGD (batch size = 16)

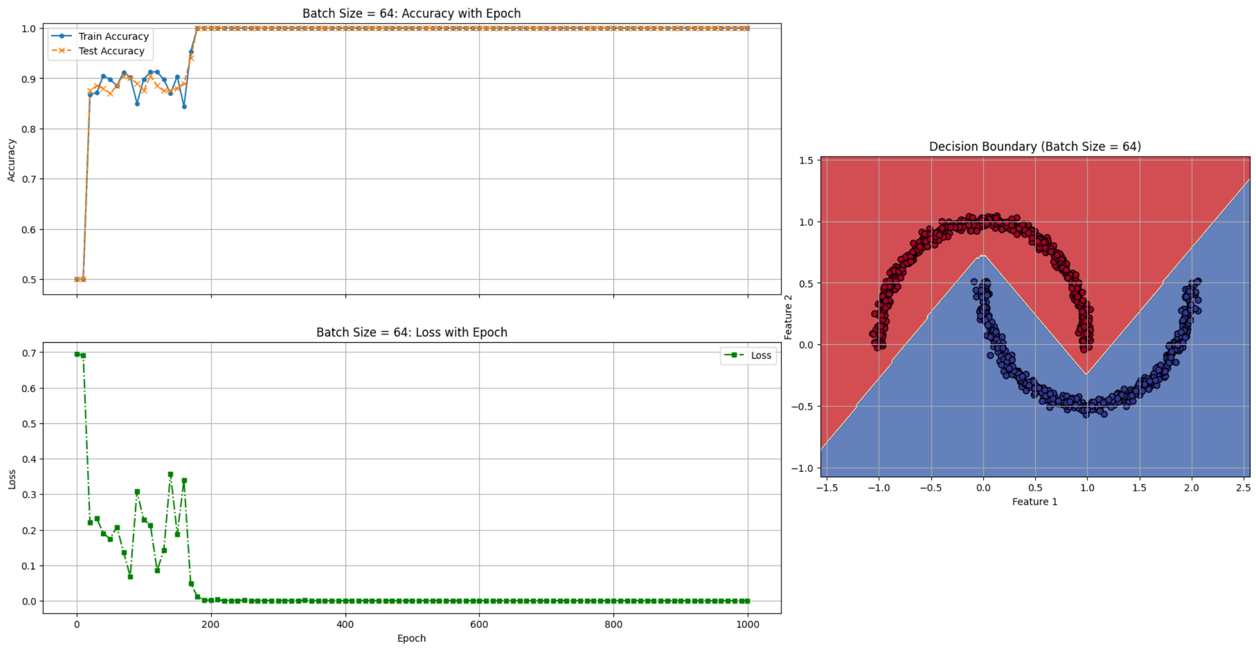

图 11: Mini-batch BGD (batch size = 64)

We can observe that with smaller batch sizes and proper learning rates, the model can achieve 100% accuracy and the loss function converges to zero in a short epoches round.

3.2 Conclusion

During the experiments, by experiences and observations, I found several important factors that influence the performance of mini-batch BGD:

- When Batch Size is smaller, or even SGD, the model has more possibility to escape from local minima, since it introduces more randomness into the training process, thus more chances to find the global minimum.
- Properly tuning the learning rate is crucial for the convergence of mini-batch BGD. Smaller batch sizes often require smaller learning rates to ensure stable training. If the learning rate is too high, it can lead to divergence or oscillations in the loss function.
- The structure of the MLP model matters. I tried different architectures and found that: for this simple task, a small layer with 2 hidden layers and 20 neurons each works well. Increasing the number of layers or neurons computes extremely slow and might lead to overfitting.

3.3 How to replicate the experiments above

For perceptron part, run the $'Part\_1\backslash perceptron.py'$, and all the graph will be shown directly.

For MLP part, find the $'Part\_2\backslash NewTask.ipynb'$. This jupiter notebook is carefully blocked, you can simply 'run all' or run each cell step by step to see the results. You cannot use some commands like "python train_mlp_numpy.py" since the data is not prepared.

4 Part 4: Extra Experiments

Notice: The Following experiments are not required in the assignment, and I do not provide the way of replicating them. However, I think they are worth discussing.

However, the 'train' function is designed explicitly, which you can try the following things in a very easy way.

4.1 Introduction of Momentum

The gradient descent can be fined by adding momentum, which helps both accelerate convergence and somehow avoid local minima. However, does it really work?

The Mathematical formulation of GD with momentum is as follows:

$$v_t = \beta v_{t-1} + \eta \nabla L(w_{t-1}) \tag{13}$$

$$w_t = w_{t-1} - \eta v_t \tag{14}$$

Define the experiment config as:

$$\begin{bmatrix} \text{Batch Size} & \text{Learning Rate} & \text{Momentum} \\ 1 & 10^{-2} & 0.9 \\ 1 & 10^{-2} & 0.7 \\ 1 & 10^{-2} & 0.4 \\ 1 & 10^{-2} & 0 \\ 64 & 4 \times 10^{-1} & 0.9 \\ 64 & 4 \times 10^{-1} & 0.7 \\ 64 & 4 \times 10^{-1} & 0.4 \\ 64 & 4 \times 10^{-1} & 0 \end{bmatrix} \tag{15}$$

And then, we conduct 5 times of each, calculate the mean round needed to reach 100% accuracy and here are the results:

$$\begin{bmatrix} \text{Batch Size} & \text{Momentum} & \text{Average Epoches to Converge} \\ 1 & 0.9 & \text{Some cases failed to converge} \\ 1 & 0.7 & 66 \\ 1 & 0.4 & 130 \\ 1 & 0 & 155 \\ 64 & 0.9 & 27 \\ 64 & 0.7 & 84 \\ 64 & 0.4 & 173 \\ 64 & 0 & 142 \end{bmatrix} \tag{16}$$

From the above statistics, we can observe some rules:

- Choosing the proper momentum is very important, unsuitable Momentum might result in slower convergence or even failure of convergence.
- If the momentum is proper, it will greatly fasten the iteration epoch!
- From this experiment, we cannot figure out whether GD with momentum can help escaping from local minimas.

4.2 ReLU, but Leaky

From the Part 2, we see that in some cases, batch_size='full' don't always converge. Is it due to the poor initialization causing 'dead' neurons? Can Leaky ReLU solve it? Let's try it out.

Leaky ReLU is just slightly different from ReLU, the formula is given as:

$$a^{(l)} = \mathit{LeakyReLU}(z^{(l)}) = \begin{cases} z^{(l)} & \text{if } z^{(l)} > 0 \\ \alpha z^{(l)} & \text{if } z^{(l)} \leq 0 \end{cases} \quad \alpha \text{ is usually a small number like 0.01} \tag{17}$$

We conduct 200 experiences on full batch size and see how many times can these 2 differenct ReLU converge within 1000 epoches.

Here are the results:(You can replicate them by *Part_2\Leaky.ipynb*!)

- Experiment 1: For the standard ReLU: 73.0%; For the $\alpha = 0.01$ Leaky ReLU: 79.5%.
- Experiment 2: For the standard ReLU: 71.50%; For the $\alpha = 0.05$ Leaky ReLU: 81.00%.
- Experiment 3: For the standard ReLU: 72.50%; For the $\alpha = 0.1$ Leaky ReLU: 80.50%.

The results consistently show that Leaky ReLU variants achieve a higher convergence rate than the standard ReLU across multiple experiments.

Although the improvement is medium (around 6-10 percentage points), it is consistent anyway.

This provides evidence supporting the hypothesis that Leaky ReLU can mitigate the 'dying ReLU' problem. By allowing a small, non-zero gradient to flow through for negative inputs, it prevents neurons from becoming permanently inactive, making the network more robust to unfortunate weight initializations, especially in the context of full-batch gradient descent where gradient diversity is limited.