SUSTech CS324 2024 Spring Assignment 1 Report

12110304 Chunhui XU

April 12, 2024

1 Part 1

In this part, I need to create a simple perceptron to complete a two-classification problem. But before start, I will provide some information about perceptron in this Part **refer from our tutorial materials**.

1.1 Problem Analysis

1.1.1 Simple Introduction

A perceptron takes multiple input values, each multiplied by a weight, sums them up, and produces a single binary output based on whether the sum is above a certain threshold.

In my code implementation, I simply insert a new column as the bias value.

1.1.2 Mathematical Model

The perceptron decision is based on these formulas:

$$f(x) = sign(w \cdot x + b)$$

$$sign(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Components of a Perceptron:

- Inputs (x): The features or data points provided to the perceptron.
- Weights (w): Coefficients determining the importance of each input.
- Bias (b): An extra input (always 1) with its own weight, allowing the activation function to shift, fitting the data better.
- Activation Function(sign function): Decides whether the neuron should activate, typically a step function for basic perceptrons

1.1.3 Loss Function

We need to design a loss function to compute the model loss,

$$
\begin{aligned}
L_1(w,b) &= \sum_{i=1}^{N} -y_i * f(x_i) \ \textbf{when} \ y_i * f(x_1) < 0 \\
&= \sum_{i=1}^{N} -y_i * sign(w * x_i + b) \ \textbf{when} \ y_i * sign(w * x_i + b < 0 \\
L_2(w,b) &= \sum_{i=1}^{N} \frac{1}{||w||} |w * x_i + b| \ \textbf{when} \ y_i * (w * x_i + b) < 0 \\
&= -\frac{1}{||w||} \sum_{i=1}^{N} y_i * (w * x_i + b) \ \textbf{when} \ y_i * (w * x_i + b) < 0
\end{aligned}
$$

Obviously, the first function just compute the number of misclassified points, it cannot be differentiated, so I choose L_2 as my loss function, which expresses the total distance from the misclassified point to the hyperplane S.

And we can ignore this coefficient $\frac{1}{||w||}$, the loss function is:

$$L_3(w,b) = -\sum_{i=1}^{N} y_i * (w * x_i + b) \ \text{ when } y_i * (w * x_i + b) < 0$$

In this way, the derivation of Loss function is:

$$
\begin{aligned}
\nabla_w L(w,b) &= -\sum_{x_i \in M} y_i x_i \\
\nabla_b L(w,b) &= -\sum_{x_i \in M} y_i
\end{aligned}
$$

1.1.4 Gradient Descent

We need to get the gradient descent equation, that is

$$b = a - \gamma \nabla f(a)$$

where γ is the learning rate (usually 0.01).

1.1.5 Types of Gradient Descent

- **Batch Gradient Descent (BGD)**: In batch gradient descent, model parameters are updated after computing the gradient of the cost function with respect to the entire training data set. It involves computing the gradient of the cost function over the entire dataset at each iteration, which can be computationally expensive for large datasets.

- **Stochastic Gradient Descent (SGD)**: Unlike batch gradient descent, stochastic gradient descent updates model parameters after computing the gradient of the cost function for only one randomly selected training example at a time. Compared to batch gradient descent, this method is computationally cheaper, especially for large datasets, since it only requires computing the gradient for a single example in each iteration.

- **Mini-Batch Gradient Descent**: Mini-batch gradient descent combines the advantages of batch gradient descent and SGD, updating model parameters in each iteration based on a small random subset of the training data set. This method strikes a balance between the computational efficiency of SGD and the stability of batch gradient descent.

1.1.6 The "standard" algorithm

Given a training set $D = \{(x_i, y_i)\}, x_i \in \mathbb{R}^N, y_i \in \{-1, 1\}$

1. Initialize $w = 0 \in \mathbb{R}^n$
2. For epoch $= 1 \cdots T$:
   (a) Compute the predictions of Perceptron of the whole training set
   (b) Compute the gradient of the loss function with respect to w:
   $$gradient = -\frac{1}{N}\sum(x_i \cdot y_i), for\ sample_i : p_i * y_i < 0$$
   where p_i is i th prediction, y_i is the related ground truth of sample i, N is the number of misclassified points
   (c) Update $w \leftarrow w - lr * gradient$
3. Return w

1.2 Results Display

In Part1, I insert a bias column, so it's no need to generate symmetric mean value.

1.2.1 Data Feature

I generate 3 test example, the Table 1 show the detailed information. They are the test ID, the mean of positive and negative samples, and the covariance of positive and negative samples.

Table 1: Data Set Feature

ID	Mean Pos	Mean Neg	Cov Pos	Cov Neg
1	[3, 3]	[15, 15]	[[2,0], [0,2]]	[[2,0], [0,2]]
2	[3, 3]	[6, 6]	[[10,0], [0,10]]	[[10,0],[0,10]]
3	[3, 3]	[4, 4]	[[2,0], [0,2]]	[[2,0], [0,2]]

1.2.2 Default Parameters

- `n_inputs`: 2, number of inputs
- `max_epochs`: 100, maximum number of training cycles
- `learning_rate`: 0.01, magnitude of weight changes at each training cycle

1.2.3 Data Visualization

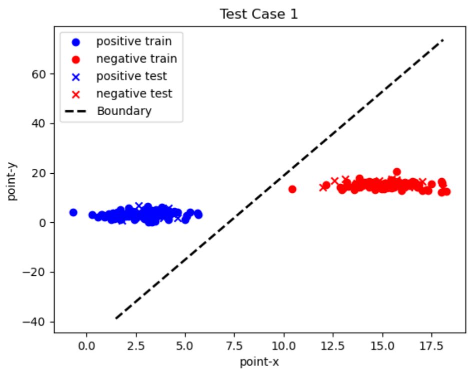
Figure 1, show us an example of data visualization for the data and train.
Also, you can check Figure 7, Figure 8 in Appendix section.

1.3 Results Analysis

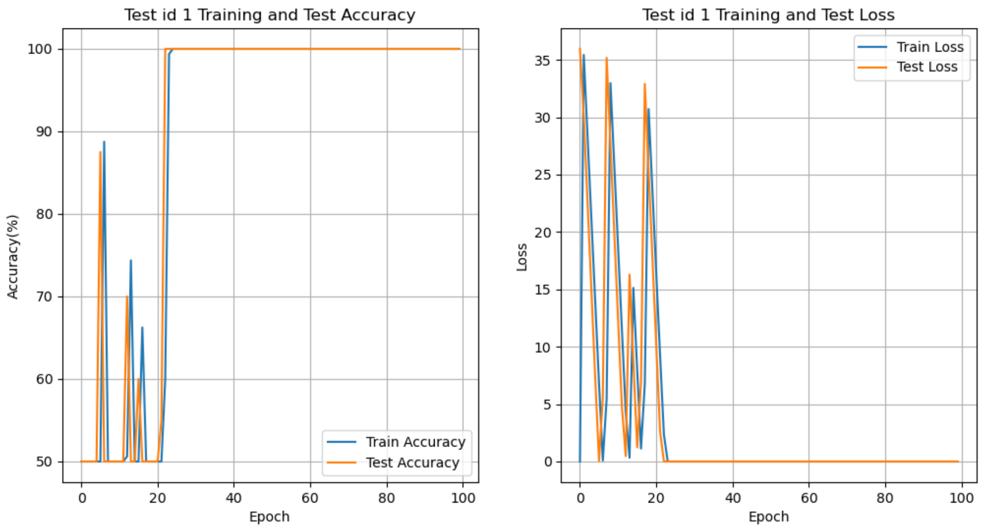
1.3.1 Data Feature Analysis

Actually, the 3 test case corresponding to 3 different situations:

1. The mean difference is large, and the respective variances are small.
2. The variances are both large.
3. The mean difference is small.



(a) Test case 1 Data



(b) Test case 1 Accuracy and Loss

Figure 1: Test Case 1

1.3.2 Training Loss and Accuracy Analysis

By comparing the three pictures, we can find that when the mean difference between positive and negative samples is large and their respective variances are small, the classification is more accurate and the training results are better.

The latter two case lead to two different situation:

- **The variances are both large**: Although the difference between the means may be large enough, excessive variance will cause the data to deviate more from the mean. This will lead to a possible situation: positive and negative samples are each generated near the mean of the other party.
- **The mean difference is small**: In this case, the points of the two types of samples will be too close. In this case, the points of the two types of samples will be mixed together, making it difficult to find a suitable way to classify them accurately.

In both cases, one type of data acts like a ”spy” masquerading as the other’s sample points. it is easy to be predicted by Perceptron as the other party’s sample point during training, conflicting with its own label, causing training confusion and leading to problems.

Intuitively, this is because the latter two case will cause the possible areas of positive and negative sample points to overlap too high, making it difficult (or even impossible) to find a decision boundary to completely separate the two samples.

As training data continues to increase, it does not improve Perceptron's training efficiency. Instead, it continues to bring confusing and misleading information.

This results in Perceptron constantly adjusting its weight and bias to fit the training data during the training process, and making it difficult to converge to a stable value. So we can see that, the loss value and accuracy fluctuates wildly in case 2 and 3. Because the model cannot converge in a better direction during the training process, but may become worse due to misleading data.

Only data like 1 that can find effective decision boundary can be trained to achieve higher accuracy. Understanding from the images means that a boundary can be found to completely separate the two types of data.

1.4 Simple Conclusion

In this part, I implemented a Perceptron. Starting with some relatively simple methods of forward propagation, back propagation and gradient descent, I gained a basic understanding of the deep learning methodology.

2 Part 2

2.1 Problem Analysis

In this part, I need to create a a multi-layer perceptron(MLP) by Numpy, generate a given dataset and do the classification problem correctly.

2.1.1 Simple Introduction

In the second part, I use scikit-learn package and generate by `make_moon` function. Then, I create a dataset of 1000 two-dimensional points. Let S denote the dataset, i.e., the set of tuples $\{(x^{(0),s}, t^s)\}_{s=1}^{S}$, where $x^{(0),s}$ is the s-th element of the dataset and t^s is its label. And I need to use `OneHotEncoder` to encode the label.

After that, I need to implement the MLP and make correct classification for this problem.

2.1.2 MLP Structure

Let d_0 be the dimension of the input space and d_n the dimension of the output space. The network I build have N layers (including the output layer). In particular, the structure will be as follows:

- Each layer $l = 1, \cdots, N$ first applies the affine mapping

$$\tilde{x}^{(l)} = W^{(l)} x^{(l-1)} + b^{(l)}$$

  where $W^{(l)} \in \mathbb{R}^{d_l \times d_{(l-1)}}$ is the matrix of the weight parameters and $b^{(l)} \in \mathbb{R}^{d_l}$ is the vector of biases. Given $\tilde{x}^{(l)}$, the activation of the l-th layer is computed using a ReLU unit

$$x^{(l)} = \max(0, \tilde{x}^{(l)})$$

- The output layer (i.e., the N-th layer) first applies the affine mapping

$$\tilde{x}^{(N)} = W^{(N)} x^{(N-1)} + b^{(N)}$$

  and then uses the softmax activation function (instead of the ReLU of the previous layers) to compute a valid probability mass function (pmf)

$$x^{(N)} = \text{softmax}(\tilde{x}^{(N)}) = \frac{\exp(\tilde{x}^{(N)})}{\sum_{i=1}^{d_N} \exp(\tilde{x}^{(N)})_i}$$

  Note that both max and exp are element-wise operations.

- Finally, compute the cross entropy loss L between the predicted and the actual label,

$$L(x^{(N)}, t) = -\sum_i t_i \log x_i^{(N)}$$

Particularly, in this assignment, there is only one hidden layer, the final structure is:

1. 2 input
2. 2 $\rightarrow$ 20 Linear Layer
3. 20 $\rightarrow$ 20 ReLU Layer
4. 20 $\rightarrow$ 2 Linear Layer
5. 2 $\rightarrow$ 2 Softmax Layer
6. Output and Cross Entropy loss function

2.1.3 Default Parameters

- `dnn_hidden_units`: [20], comma separated list of number of units in each hidden layer
- `learning_rate`: 1e^{-2}, learning rate for optimization
- `max_steps`: 1500, number of epochs to run trainer
- `eval_freq`: 10, frequency of evaluation on the test set

2.2 Forward and Backward Propagation Functions

2.2.1 Linear Layer

Forward propagation:

$$z = Wx + b$$

- W: weight matrix
- x: input vector
- b: bias vector

Backward propagation:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z} x^T$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}$$
$$\frac{\partial L}{\partial x} = W^T \frac{\partial L}{\partial z}$$

- L: The Loss function
- $\frac{\partial L}{\partial z}$: gradient of the loss with respect to the linear layer output

2.2.2 ReLU Layer

Forward propagation:

$$y = \max(0, x)$$

- x: input vector

Backward propagation:

$$grad(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot grad(x)$$

2.2.3 Softmax Layer

Forward propagation:

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- z_i: i-th element of the input vector
- K: number of classes, that is 2

Backward propagation:
The backward pass for softmax is often directly integrated with CrossEntropy for simplicity.

2.2.4 Cross-entropy Loss

Forward propagation:

$$L = -\sum_{i=1}^{K} y_i \log(\hat{y}_i)$$

- y_i: true label (one-hot encoded)
- $\hat{y}_i$: predicted probability for class i

Backward propagation:

$$\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i$$

- y_i: true label (one-hot encoded)

2.3 Results and Display

2.3.1 Data Visualization

Figure 2 show the visualization of the whole data set. 1000 positive data and 1000 negative data.

2.3.2 My Additional parameters

- `batch_size`: Positive value for Batch Gradient Descent and negative value for Mini-BGD (absolute value as batch size).

2.3.3 Test Figure

In our training, epoch is given and fixed, which means that the smaller the batch size, the more iterations are required, and large batches of parallel calculations are lost, which may take longer time.

Figure 3, 4, 5, 6 show SGD, batch size 10, 100, 800 for 1500 epoch respectively.

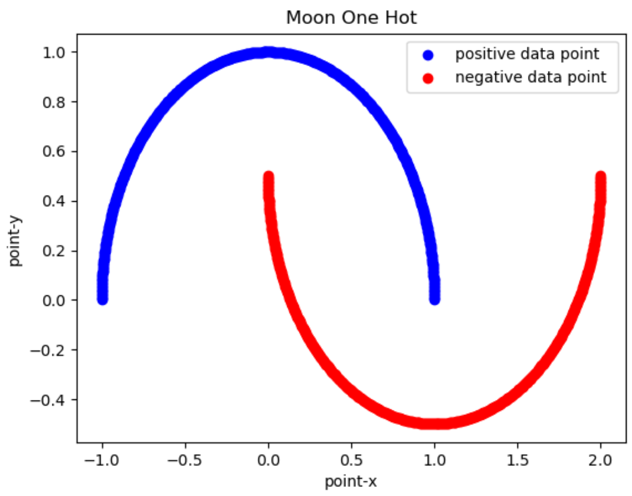


Figure 2: Moon Sample Data

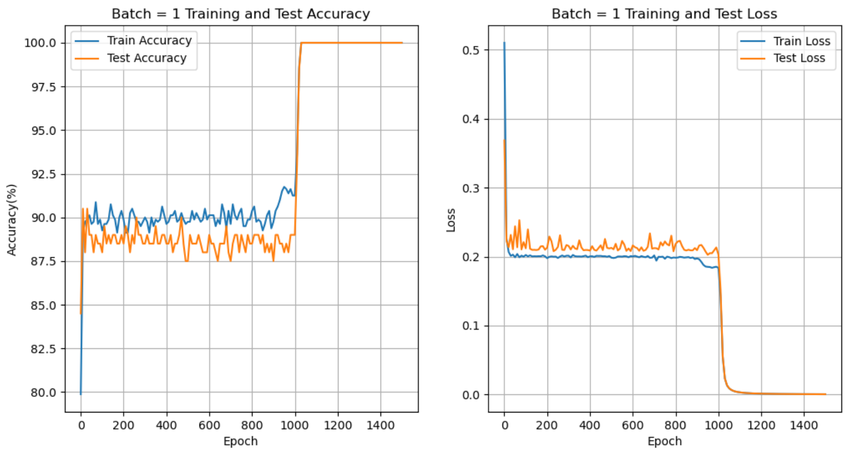


Figure 3: SGD

2.4 Results and Analysis

Because I mainly focused on the analysis of SGD and BGD, I did not consider changing other default parameters in this part. They maintain consistent values.

2.4.1 Characteristic Analysis

In general, SGD and smaller batche sizes mean faster training: less data is involved in each iteration, making iterations faster. But in this Assignment, epoch is given, which means that although the data in each iteration becomes less, we need more iterations to complete the traversal of the entire sample. requirements are reduced, this will lead to the loss of parallel computing opportunities for large matrices, a decrease in the overall utilization of the computing core, and a significant increase in time consumption.

2.4.2 SGD Analysis

For SGD, it declines very quickly in each epoch: in each epoch, it iterates 800 times and updates the weights 800 times. At first glance, it has more opportunities for decline than large batches. However,

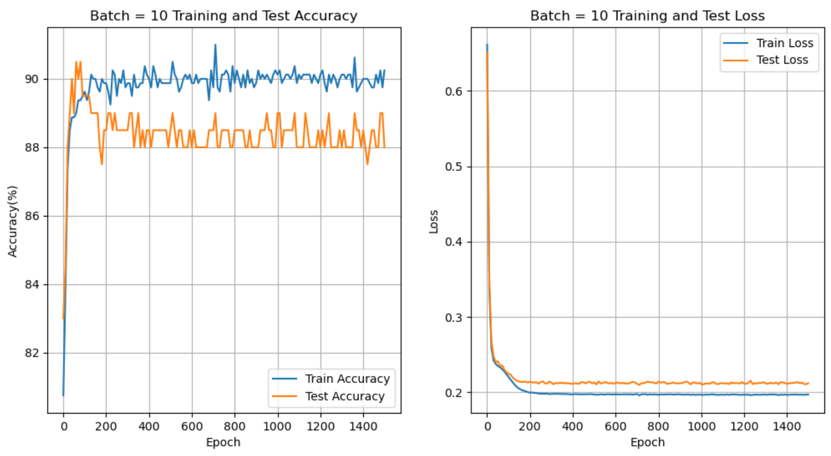


Figure 4: BGD size 10

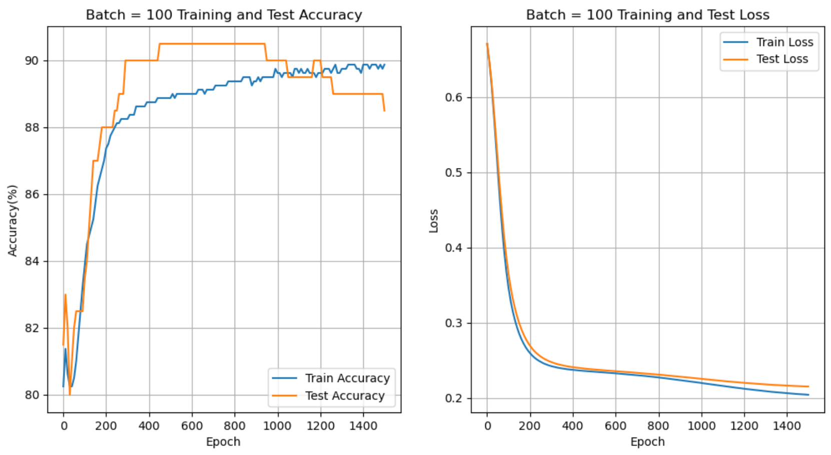


Figure 5: BGD size 100

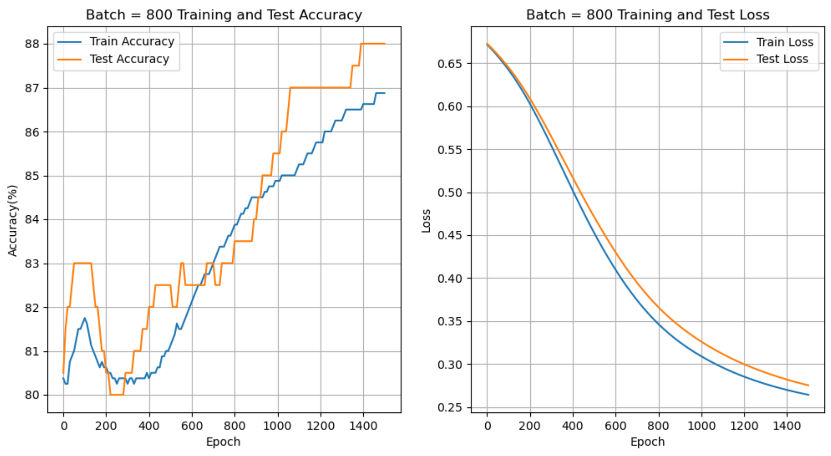


Figure 6: BGD size 800

we can see that after experiencing a rapid decline in the initial stage, its loss curve has experienced great fluctuations.

This is because each time gradient descent is performed on a single sample, it is easy to fall into a local optimal. In the fitting of a single sample, the individual samples may interfere with each other (e.g, the previous one sample updates a certain weight value by +1, and the latter one then updates this value by -1), resulting in a loss that cannot be reduce at each epoch, even lead to an increase in loss.

However, under the large number of iterations, SGD finally found an opportunity to rapidly increase the model fitting degree. Therefore, after two rapid fittings to the training set, the loss of the model has been greatly reduced and the accuracy has reached a relatively good state.

2.4.3 BGD Analysis

Different from the overview in the previous section, in this section I will highlight the impact of batch size on training characteristics in several aspects.

Attention: everything is based on the epoch is given.

- **Training speed**: As the batch size continues to increase, the number of iterations required begins to decrease, which makes the training speed significantly faster.
- **Loss**: As the batch size increases, it is easier for the model to find the global optimal point in each iteration update. Although the loss decreases less for each epoch (because the number of updates becomes less), the model can easily find the global optimal point of gradient descent, which makes the loss decrease more smoothly with almost no fluctuations.
- **Accuracy**: As the batch size becomes larger, the final accuracy limit is not as good as SGD because the number of iterations becomes smaller. However, the training accuracy of the model is still rising steadily. If there are more iteration opportunities, I believe it can reach a higher level.
- **Overfitting**: This is one of my observations: although the training accuracy of BGD with large batch size is constantly getting higher, its accuracy on the test set is not necessarily the same. BGD has lower generalization ability than SGD, and the risk of overfitting to the training set will be higher.

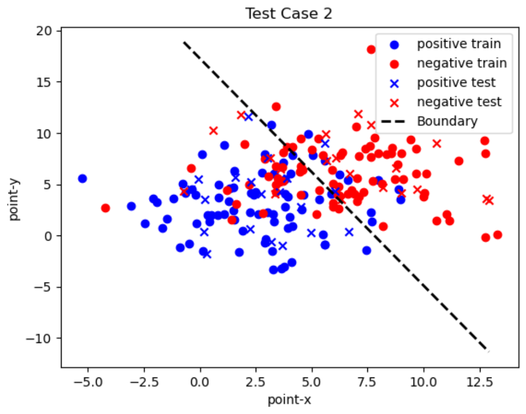
2.4.4 Additional Analysis

The appendix includes more mini-BGD test data and is updated with each iteration. I can learn more about the characteristics of each iteration of SGD and mini-BGD: the overall convergence speed is fast, but the fluctuation range is really large.

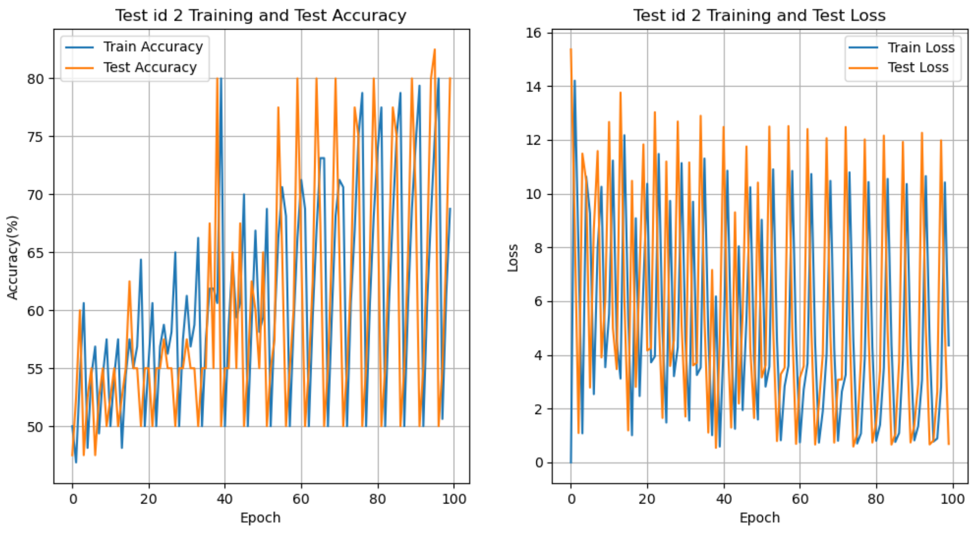
2.5 Simple Conclusion

In this part, I implemented a MLP. I learned about the hidden layer, softmax layer and cross-entropy loss function, and their corresponding forward propagation and back propagation formulas. In the comparison of different batch sizes, we learned about the various characteristics and principles hidden in training.

A Extra Pictures

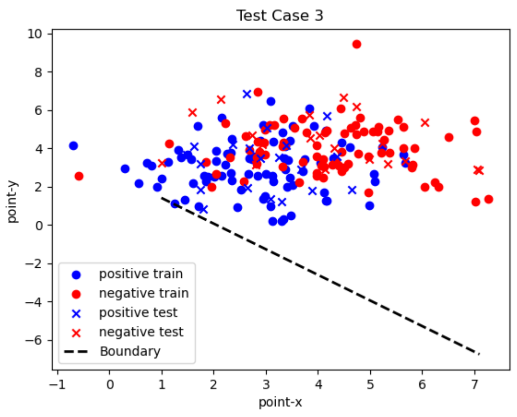


(a) Test case 2 Data

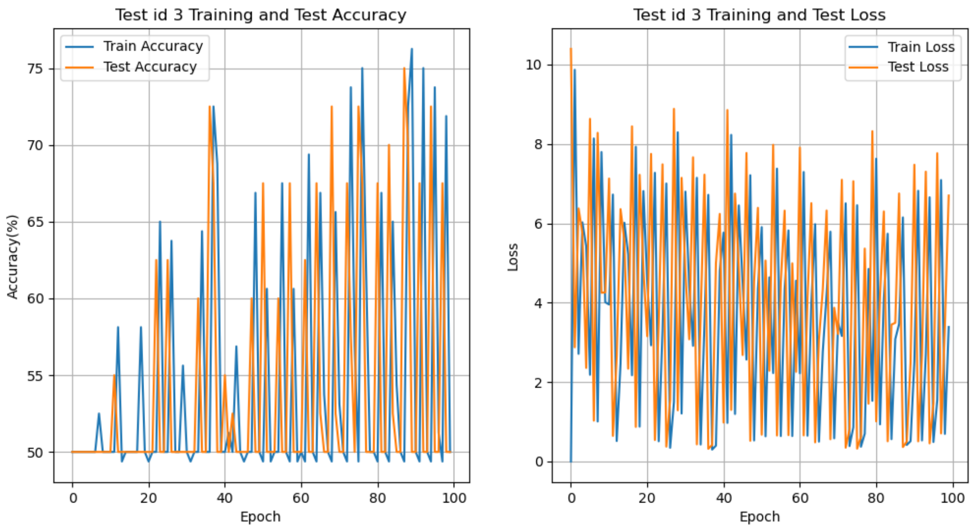


(b) Test case 2 Accuracy and Loss

Figure 7: Part 1 Test Case 2



(a) Test case 3 Data



(b) Test case 3 Accuracy and Loss

Figure 8: Part 1 Test Case 3

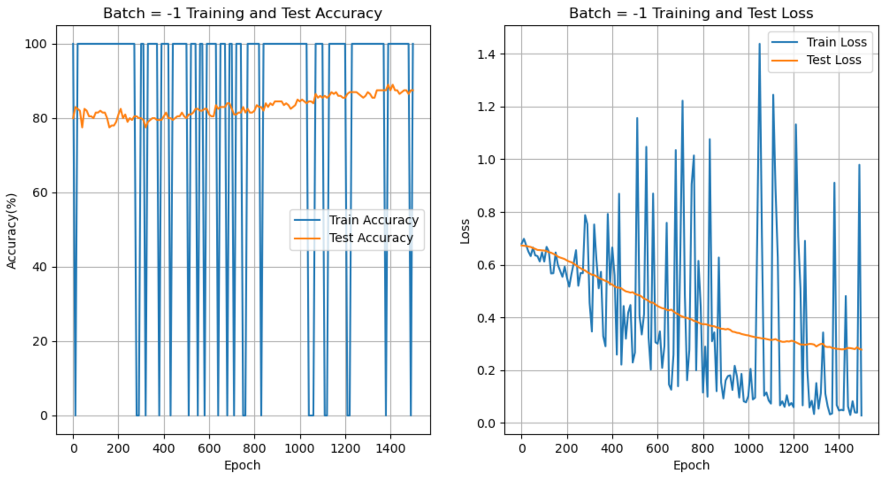


Figure 9: Mini Batch Size 1 with 1500 Iterations

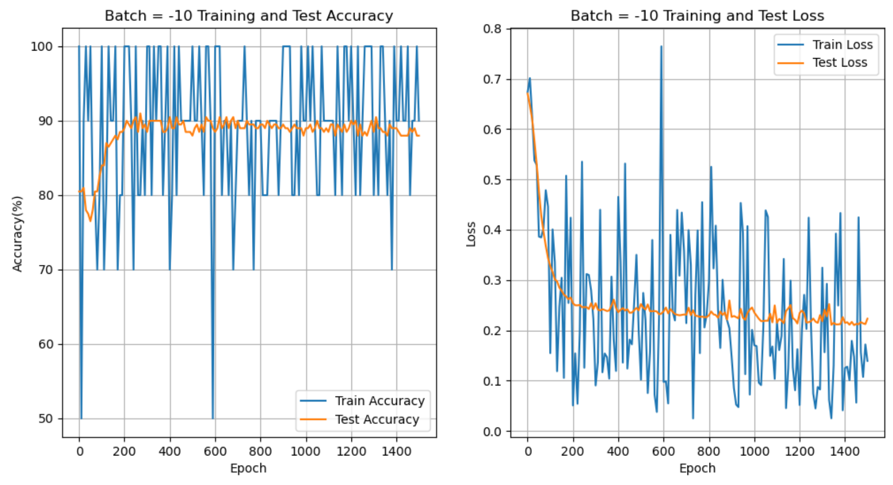


Figure 10: Mini Batch Size 10 with 1500 Iterations

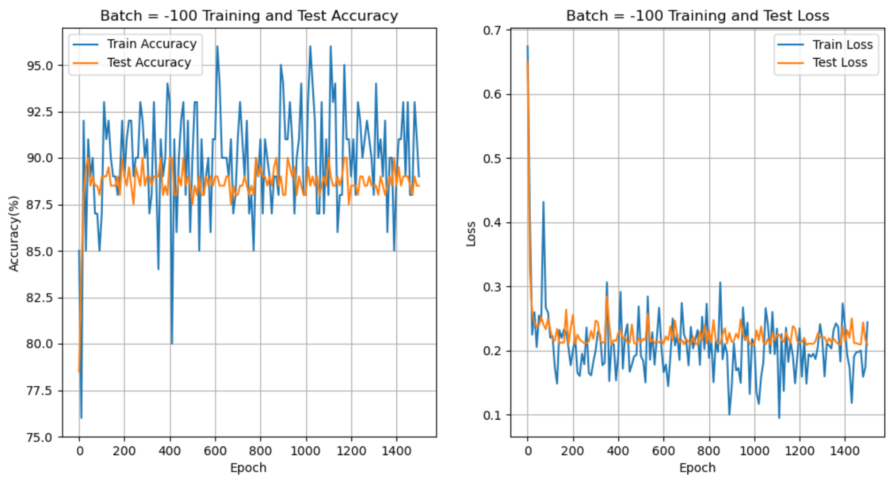


Figure 11: Mini Batch Size 100 with 1500 Iterations