Deep Learning Assignment 1

Junyou Su 12110911

October 14, 2024

1 Instructions on how to run the code

File Structure Overview

- **Assignment1/**
  - **part1/**
    - datasets/ *(Folder containing part1 datasets)*
    - **generate_dataset.py** *(Script to generate datasets)*
    - **loss.png** *(Image file showing loss visualization)*
    - **perceptron_tutorial.pdf** *(PDF tutorial on perceptron)*
    - **perceptron.py** *(Python script implementing a perceptron)*
    - **show_datasets.py** *(Script to visualize dataset)*
    - **train.py** *(Training script for the perceptron model)*
    - **visualization_train_test.png** *(Image visualizing training vs. testing)*
  - **part2/**
    - datasets/ *(Folder containing part2 datasets)*
    - **accuracy.png** *(Image showing model accuracy)*
    - **generate_dataset.py** *(Script to generate datasets)*
    - **loss.png** *(Loss visualization image)*
    - **mlp_numpy.py** *(MLP implementation using numpy)*
    - **modules.py** *(Base Module for MLP)*
    - **readme.md** *(Introdution of this part)*
    - **show_datasets.py** *(Script to show datasets)*
    - **task3.ipynb** *(Jupyter notebook include train Process)*
    - **train_mlp_numpy.py** *(Script to train MLP using numpy)*
    - **visualization_test.png** *(Test Dataset visualization image)*
    - **visualization_train.png** *(Training Dataset visualization image)*
  - **part3/**
    - datasets/ *(Folder containing part3 datasets)*
    - fig/ *(Folder containing loss and accuracy figures in different batch size)*
    - **generate_dataset.py** *(Dataset generation script)*
    - **mlp_numpy.py** *(MLP implementation using numpy)*
    - **modules.py** *(Base Module for MLP)*
    - **part3.ipynb** *(Jupyter notebook include train code in different batch size)*
    - **show_datasets.py** *(Dataset display script)*
    - **train_mlp_numpy.py** *(MLP training script)*
  - **assignment1.pdf** *(Assignment 1 PDF)*
  - **assignment1_report.pdf** *(This assignment 1 Report)*

1.1 Run the Code

1.1.1 Part1

"python train.py"

1.1.2 Part2

"python train_mlp_numpy.py –dnn_hidden_units 20 –learning_rate 0.01 –max_steps 1500 –eval_freq 10"

1.1.3 Part3

"python train_mlp_numpy.py –dnn_hidden_units 20 –learning_rate 0.01 –max_steps 1500 –eval_freq 10 –batch_size 100 –sgd"

2 Theoretical Analysis of the Learning Process of the Perceptron

The perceptron, introduced by Frank Rosenblatt in 1957, is a fundamental building block for neural networks. It mimics the behavior of biological neurons, and its primary goal is to classify input data into one of two categories by learning a decision boundary based on the input features.

2.1 Mathematical Model

The perceptron uses the following formula to make decisions:

$$f(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b)$$

where:

- $\mathbf{x}$ is the input vector,
- $\mathbf{w}$ is the weight vector,
- b is the bias term,
- $\text{sign}(\cdot)$ is the activation function, which outputs $+1$ if the argument is greater than or equal to 0, and -1 otherwise.

2.2 Geometric Interpretation

The decision boundary of the perceptron is defined by the hyperplane:

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

This boundary divides the input space into two regions: one where the perceptron predicts $+1$ and the other where it predicts -1. The vector $\mathbf{w}$ is orthogonal to the decision boundary, and the bias b shifts the boundary to better fit the data.

2.3 Learning Process

The perceptron is trained using a supervised learning algorithm. The goal is to find the optimal weights $\mathbf{w}$ and bias b such that all the training examples are correctly classified. The training procedure is as follows:

1. Initialize the weight vector $\mathbf{w} = 0$ and the bias $b = 0$.
2. For each training example $(\mathbf{x}_i, y_i)$, where y_i is the true label:
    - Compute the prediction: $\hat{y}_i = \text{sign}(\mathbf{w} \cdot \mathbf{x}_i + b)$.

- If the prediction is incorrect, update the weights and bias:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \cdot y_i \cdot \mathbf{x}_i$$

$$b \leftarrow b + \eta \cdot y_i$$

where η is the learning rate.

2.4 Loss Function and Gradient Descent

The perceptron algorithm minimizes the number of misclassified points. However, the sign function is not differentiable, so a modified loss function is used based on the total distance of misclassified points from the decision boundary. The loss function can be written as:

$$L(\mathbf{w}) = \sum_{i:\hat{y}_i \neq y_i} \frac{-y_i(\mathbf{w} \cdot \mathbf{x}_i + b)}{\|\mathbf{w}\|}$$

The perceptron algorithm uses gradient descent to minimize this loss, iteratively updating the weights in the direction of the steepest descent:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L(\mathbf{w})$$

2.5 Types of Gradient Descent

The perceptron training algorithm can be implemented using different types of gradient descent:

- **Batch Gradient Descent**: Updates the weights after evaluating all the training examples.
- **Stochastic Gradient Descent (SGD)**: Updates the weights after evaluating each training example individually.
- **Mini-batch Gradient Descent**: A compromise between batch and stochastic gradient descent, updating the weights after evaluating a subset of the training examples.

2.6 The "Standard" Algorithm

The standard algorithm for training a perceptron can be summarized as follows:

Algorithm 1 Perceptron Learning Algorithm

1: **Input:** Training set $D = \{(\mathbf{x}_i, y_i)\}$, where $y_i \in \{-1, 1\}$
2: Initialize $\mathbf{w} = 0$
3: **for** epoch = 1 to T **do**
4: Compute predictions for the entire training set
5: Compute the gradient of the loss function with respect to $\mathbf{w}$:

$$\text{gradient} = -\frac{1}{N} \sum_{i:\hat{y}_i \neq y_i} y_i \mathbf{x}_i$$

where $\hat{y}_i$ is the predicted label.
6: Update $\mathbf{w} \leftarrow \mathbf{w} - \text{lr} \times \text{gradient}$
7: **end for**
8: **Output:** Weight vector $\mathbf{w}$

3 Theoretical Analysis of Forward and Backward Propagation in Multi-layer Perceptrons

3.1 Forward Propagation

Forward propagation in a multi-layer perceptron is the process through which input data is passed through the network to generate an output. Each layer of the MLP consists of neurons, and each

neuron receives a weighted sum of the inputs from the previous layer, which is then passed through an activation function. This process is repeated layer by layer until the output layer is reached.

Mathematically, the output $\mathbf{a}^{(l)}$ of layer l is computed as follows:

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)}), \quad \mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$$

where:

- $\mathbf{W}^{(l)}$ is the weight matrix for layer l,
- $\mathbf{b}^{(l)}$ is the bias vector for layer l,
- $\mathbf{a}^{(l-1)}$ is the output from the previous layer,
- f is the activation function (e.g., sigmoid, ReLU, or tanh).

The forward pass continues through each layer of the MLP, ultimately producing an output $\mathbf{y}$, which can be used for predictions.

3.2 Backward Propagation

Backward propagation, or backpropagation, is the process used to compute gradients of the loss function with respect to the weights of the network. These gradients are used in gradient descent or other optimization methods to update the weights and minimize the loss function.

1. Loss Calculation: First, the loss function $\mathcal{L}$ is calculated, which quantifies the error between the network's predicted output $\hat{\mathbf{y}}$ and the actual target $\mathbf{y}$. For example, with cross-entropy loss for classification tasks:

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

2. Gradient of the Output Layer: The error in the output layer $\delta^{(L)}$ is computed as the derivative of the loss with respect to the activation of the output layer:

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(L)}} \odot f'(\mathbf{z}^{(L)})$$

where $\odot$ denotes element-wise multiplication, and f' is the derivative of the activation function.

3. Backpropagating the Error: The error is propagated back through the network to update the weights of the preceding layers. For each layer l, the error $\delta^{(l)}$ is computed as:

$$\delta^{(l)} = (\mathbf{W}^{(l+1)})^\top \delta^{(l+1)} \odot f'(\mathbf{z}^{(l)})$$

4. Weight Updates: The gradients of the weights are computed as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} (\mathbf{a}^{(l-1)})^\top$$

The weights and biases are updated using gradient descent:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}$$

where η is the learning rate.

3.3 Linear Layer

The linear layer performs a fully connected transformation of the input, where each output neuron is a weighted sum of the input neurons plus a bias term. Mathematically, the forward pass is represented as:

$$\mathbf{y} = \mathbf{x}\mathbf{W} + \mathbf{b}$$

where:

- $\mathbf{x}$ is the input of shape $(N, \text{in\_features})$, where N is the number of samples,
- $\mathbf{W}$ is the weight matrix of shape $(\text{in\_features}, \text{out\_features})$,
- $\mathbf{b}$ is the bias vector of shape $(1, \text{out\_features})$.

The backward pass computes the gradients with respect to both the weights and the input. The gradients are:

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{1}{N}\mathbf{x}^\top \mathbf{dout}, \quad \frac{\partial L}{\partial \mathbf{b}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{dout}_i$$

where $\mathbf{dout}$ is the gradient from the next layer, and L is the loss function.

3.4 ReLU Layer

The ReLU (Rectified Linear Unit) layer applies the ReLU activation function element-wise to the input, which is defined as:

$$\text{ReLU}(\mathbf{x}) = \max(0, \mathbf{x})$$

In the forward pass, ReLU outputs the input if it is positive, and zero otherwise. The backward pass computes the gradient of the loss with respect to the input. The gradient is:

$$\frac{\partial L}{\partial \mathbf{x}} = \mathbf{dout} \odot (\mathbf{x} > 0)$$

where $\odot$ is the element-wise multiplication, and $(\mathbf{x} > 0)$ is a mask that sets the gradient to zero where the input was negative.

3.5 Softmax Layer

The softmax layer converts the raw output logits from the previous layer into probabilities. The softmax function is defined as:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

for each element x_i of the input vector $\mathbf{x}$. To ensure numerical stability, the softmax function can be computed using the "Max Trick," which involves subtracting the maximum value from the input before applying the exponential function.

In the forward pass, softmax produces a probability distribution over the output classes. The backward pass for softmax is often combined with the cross-entropy loss for simplicity.

3.6 Cross-entropy Layer

The cross-entropy layer computes the loss between the predicted probability distribution and the true labels. The cross-entropy loss is defined as:

$$L = -\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic} \log(p_{ic})$$

where:

- N is the number of samples,
- C is the number of classes,
- y_{ic} is the one-hot encoded label for class c of sample i,
- p_{ic} is the predicted probability for class c of sample i.

In the backward pass, the gradient of the cross-entropy loss with respect to the softmax output simplifies to:

$$\frac{\partial L}{\partial \mathbf{x}} = \mathbf{p} - \mathbf{y}$$

where $\mathbf{p}$ is the predicted probability distribution from softmax, and $\mathbf{y}$ is the true label.

4 Loss curve of training and testing

4.1 Loss of Perceptron

The perceptron model used in this experiment was configured with the following parameters:

- **Number of Inputs (n_inputs)**: 2
- **Maximum Epochs**: 1000
- **Learning Rate**: 0.01
- **Datasets Details**: see Section 5.1.1

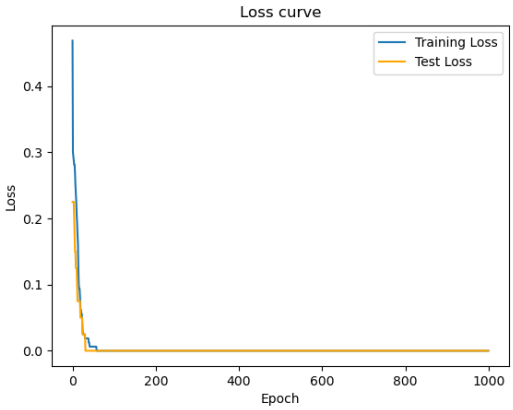


Figure 1: **The Train Loss and Test Loss of Preceptron**

4.2 Loss of MLP

The MLP model used in this experiment was configured with the following parameters:

- **Hidden Units**: 20
- **Learning Rate**: 1e-2
- **Maximum Epochs**: 1500
- **Evaluation Frequency**: Every 10 epochs
- **Datasets Details**: see Section 5.2.1

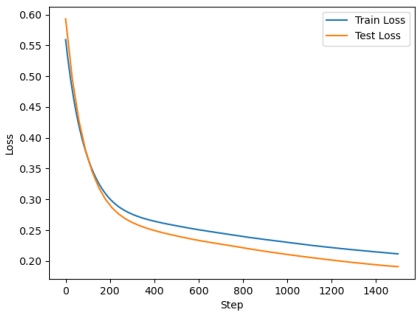


Figure 2: **The Train Loss and Test Loss of MLP**

4.3 SGD

Experiment Setup

- **Hidden Units**: 20
- **Learning Rate**: 1e-2
- **Maximum Epochs**: 1500
- **Evaluation Frequency**: Every 10 epochs
- **Datasets Details**: see Section 5.2.1

4.3.1 Batch Size: 1 (Stochastic gradient descent) & 10

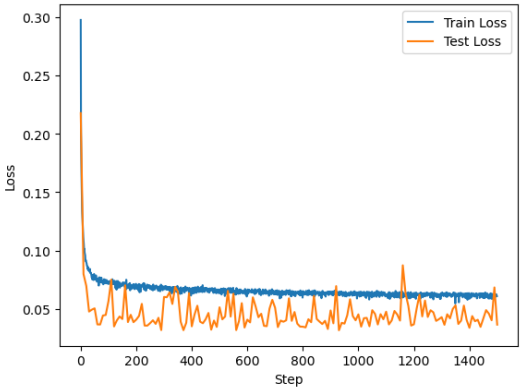


Figure 3: **The Train Loss and Test Loss of MLP (Batch Size: 1)**

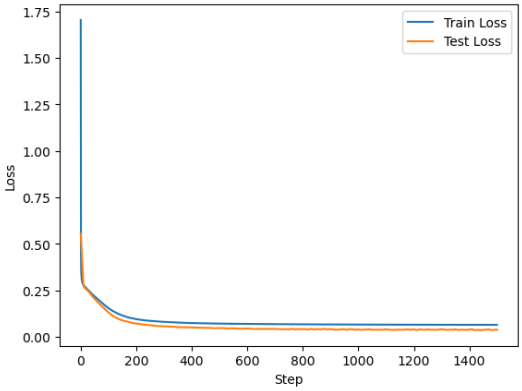


Figure 4: **The Train Loss and Test Loss of MLP (Batch Size: 10)**

4.3.2 Batch Size: 100 & 200

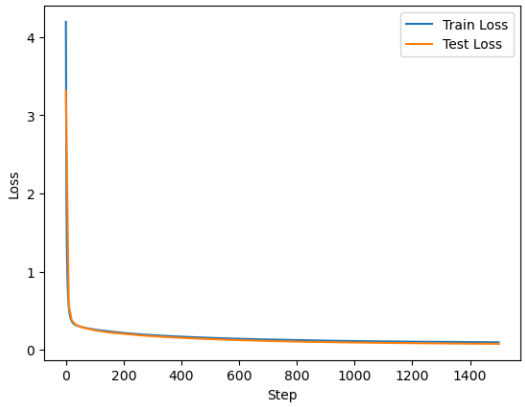


Figure 5: **The Train Loss and Test Loss of MLP (Batch Size: 100)**

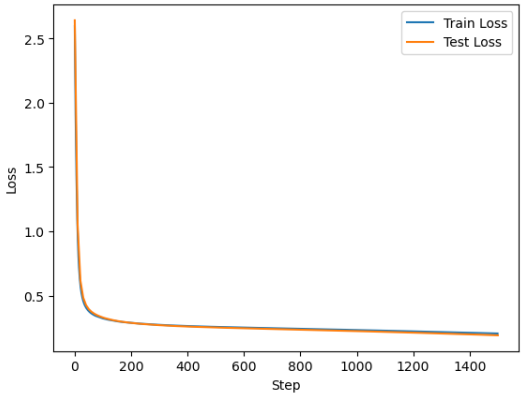


Figure 6: **The Train Loss and Test Loss of MLP (Batch Size: 200)**

4.3.3 Batch Size: 400 & 800

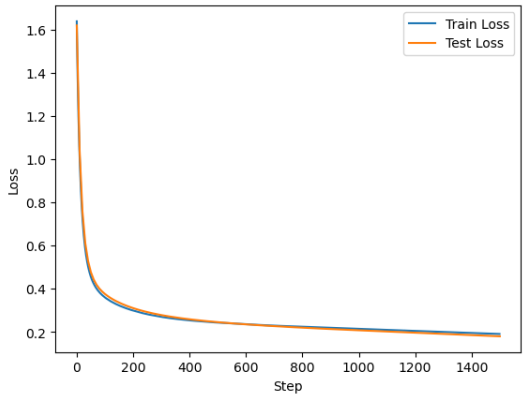


Figure 7: **The Train Loss and Test Loss of MLP (Batch Size: 400)**

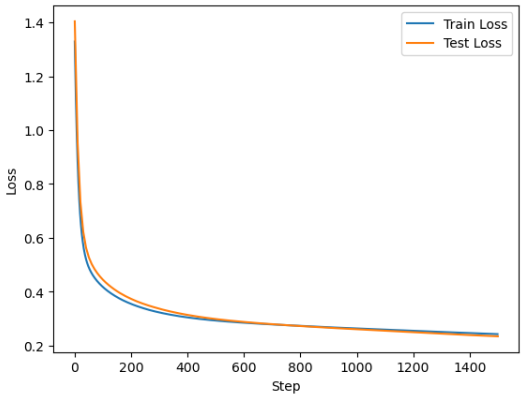


Figure 8: **The Train Loss and Test Loss of MLP (Batch Size: 800)**

5 Analysis of results

5.1 Perceptron

5.1.1 Task1

In this task, I generated a dataset of points in $\mathbb{R}^2$. The dataset consists of two Gaussian distributions, with 100 points sampled from each distribution. The first distribution is centered at $(0, 0)$ with a standard deviation of 1, while the second distribution is centered at $(10, 10)$, also with a standard deviation of 1.

A total of 200 points were generated: 100 from the first Gaussian and 100 from the second. Each point was labeled according to its distribution, with points from the first distribution labeled as $+1$ and points from the second distribution labeled as -1. After generating the dataset, it was randomly shuffled and split into training and test sets.

The training set contains 80 points from each distribution (160 points in total), and the test set contains 20 points from each distribution (40 points in total).

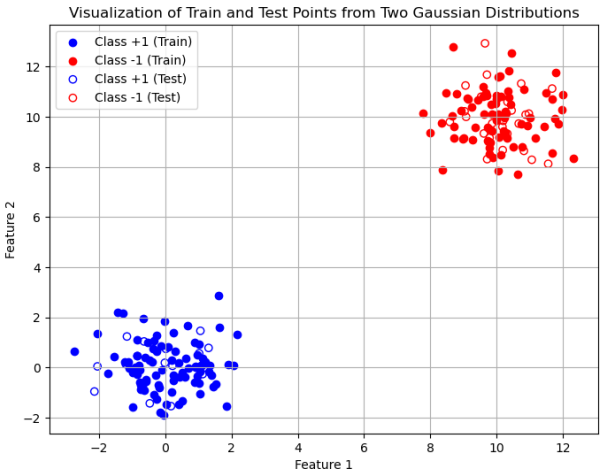


Figure 9: **The Visualization of Default Train and Test Dataset of Perceptron**

5.1.2 Task 2 & Task 3

For these two tasks, I implemented the perceptron in perceptron.py using the standard algorithm and completed the training process in train.py. The perceptron was trained on the dataset generated in Task 1 (see Section 5.1.1). The model achieved a classification accuracy of **100%** on this dataset.

5.1.3 Task4

The perceptron model used in this experiment was configured with the following parameters:

- **Number of Inputs (n_inputs)**: 2
- **Maximum Epochs**: 1000
- **Learning Rate**: 0.01

mean1	sigma1	mean2	sigma2	accuracy
[10, 10]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	100%
[5, 5]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	100%
[3, 3]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	100%
[2, 2]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	85.0%
[1, 1]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	47.5%
[0.5, 0.5]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	50.0%
[0, 0]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	57.5%

Table 1: Accuracy comparison with varying mean and constant sigma

mean1	sigma1	mean2	sigma2	accuracy
[10, 10]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	100%
[5, 5]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	100%
[3, 3]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	100%
[2, 2]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	85.0%
[1, 1]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	47.5%
[0.5, 0.5]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	50.0%
[0, 0]	[[1, 0], [0, 1]]	[0, 0]	[[1, 0], [0, 1]]	57.5%

Table 2: Accuracy comparison with varying mean and constant sigma

The experiment demonstrates how varying the means and variances of two Gaussian distributions influences the performance of a perceptron in a classification task. Here's a breakdown of the key observations:

Effect of Varying Means (Constant Variance) As seen in Table 1, when the means of the two Gaussian distributions are relatively far apart, such as $[10, 10]$ and $[0, 0]$, the perceptron achieves 100% accuracy. This result is expected since the two classes are linearly separable, allowing the perceptron to easily find a decision boundary between the two sets of points.

However, as the means of the two distributions get closer to each other (e.g., $[2, 2]$ vs. $[0, 0]$), the accuracy drops to 85%, indicating that the distributions overlap, making it harder for the perceptron to separate the classes correctly. When the means are very close or the same (e.g., $[0.5, 0.5]$ or $[0, 0]$ vs. $[0, 0]$), the accuracy drops significantly (47.5% - 57.5%), as the perceptron struggles to differentiate between the points. When the mean very close, the accuracy should fluctuate around 50%, these changes are normal.

Effect of High Variance When the variance of the Gaussian distributions increases significantly, as explored in later experiments, the spread of the data increases, resulting in more overlap between the two distributions. This overlap decreases the accuracy of the perceptron, as the decision boundary becomes less clear. For instance, when the variance is large ($[50, 50]$ or greater), the perceptron achieves much lower accuracy (e.g., 42.5%).

Conclusion

- **Close means**: When the means of the two distributions are too close, the decision boundary between the two classes becomes unclear, leading to a significant drop in accuracy.
- **High variance**: High variance increases the spread of the data, creating overlap between classes, and reducing the perceptron's ability to classify points correctly.

This experiment shows the limitations of the perceptron model in scenarios where the data distributions have significant overlap due to small mean differences or high variance.

5.2 MLP

5.2.1 Task1 & Task2

In Task1, I implement the MLP architecture by completing the files mlp numpy.py and modules.py.

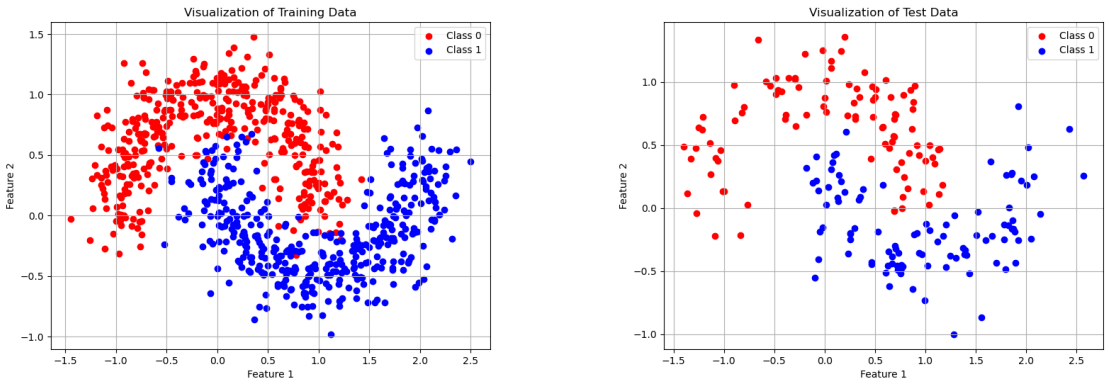


Figure 10: **The Visualization of Train and Test Dataset of MLP**

In Task2, I developed a training and testing script within `train_mlp_numpy.py`. The goal of this task is to train the MLP model on the moon-shaped dataset generated using

the `make_moons` function from the `sklearn.datasets` module. The dataset consists of 1000 samples, with 80% of the data reserved for training and 20% for testing.

5.2.2 Task3

The train loss and test loss of MLP showed in Figure 2 before.

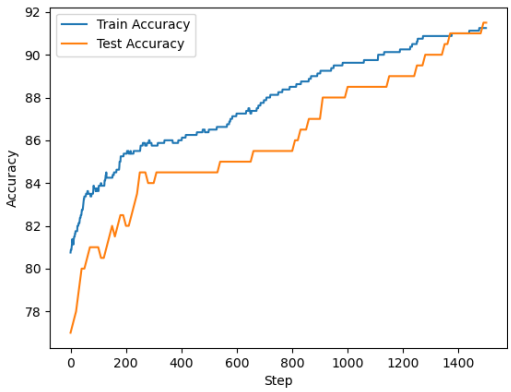


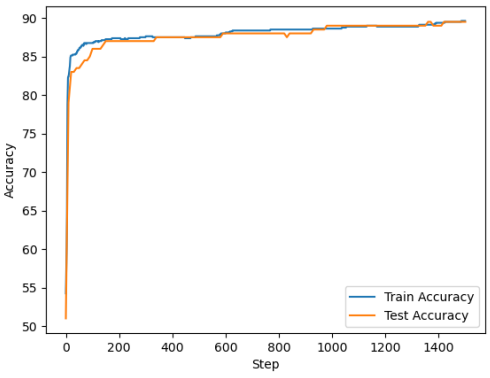
Figure 11: **The Train Loss and Test Loss of MLP**

From the accuracy curve is Figure 11, we observe that the training and test accuracy initially start around **77%** and progressively rise to about **92%** throughout the training process. This suggests that the model is effectively learning to classify the training examples and generalizing well to unseen data. The relatively small gap between the training and test accuracy curves further indicates no significant overfitting, implying a satisfactory performance overall.

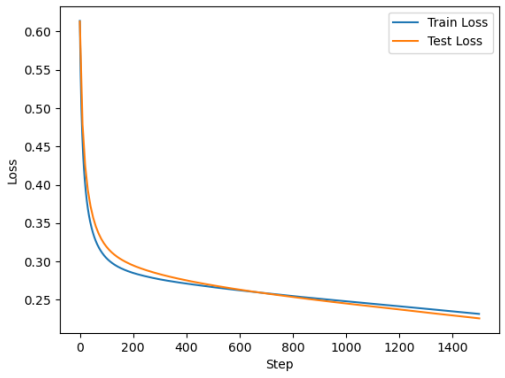
At the early stages of training, there is a noticeable plateau in accuracy, which may suggest that the model has reached a local optimum where the loss function is minimized, but not at the global minimum.

5.2.3 More Explore

I noticed that the initial test accuracy was significantly higher than anticipated. To investigate, I changed the random seed for parameter initialization from 42 to 4. As a result, the initial accuracy was around 50%, as shown in Figure 12a for accuracy and Figure 12b for loss.



(a) The Accuracy of MLP(Random Seed 4)



(b) The Loss of Preceptron(Random Seed 4)

5.3 SGD

5.3.1 Task 1 & Task 2

In this task, we modified the `train` method to accept a parameter that allows the user to specify whether to perform training using batch gradient descent (which we implemented in Part II) or stochastic gradient descent (SGD) with a batch size of 1.

5.4 Experimental Setup

We used the default parameter values and conducted experiments with varying batch sizes. The batch sizes ranged from 1 to the max value 800. For each case, we recorded the accuracy curves for both training and testing data. The Loss Figure I showed in Figure 345678

5.5 Impact of Batch Size on Stochastic Gradient Descent

5.5.1 Stochastic Gradient Descent (Batch Size = 1)

Stochastic Gradient Descent (SGD) operates by updating the model parameters after each training example, introducing a level of stochasticity or randomness into the training process. This inherent noise can be beneficial in several ways:

- **Escaping Sharp Minima**: The random fluctuations in gradient estimates allow the model to escape from sharp minima, which are often associated with poor generalization. This characteristic can help the model explore the loss landscape more effectively, potentially leading to better overall optima.
- **Noisy Training Process**: However, the high variance in gradient updates can make the training process volatile and less stable. As depicted in the training graphs, this noise manifests as multiple small oscillations in the accuracy curves, reflecting the erratic nature of the updates. While this noise can facilitate exploration, it may also hinder convergence towards a stable solution.
- **Computational Efficiency**: From a computational standpoint, SGD is generally inefficient compared to batch or mini-batch gradient descent. Each update requires processing a single training example, resulting in frequent model parameter updates. Consequently, this can lead to increased training time, making SGD the most time-consuming approach among the three batch sizes considered in our experiments.

5.5.2 Small Batch Size

Utilizing a small batch size strikes an advantageous balance between the benefits of larger batch sizes and the inherent noise associated with SGD. Here are some key aspects:

- **Variance Reduction**: When gradients are computed over a small subset of the training data, the variance of the gradient estimates is reduced compared to SGD. This averaging process mitigates the noise introduced by individual examples, resulting in more stable updates.
- **Trade-off in Convergence**: The small batch size allows the model to benefit from some level of stochasticity while maintaining a degree of stability in convergence. This balance can facilitate smoother convergence towards a minimum, as the model can explore the loss landscape without being overly influenced by any single training example.
- **Ability to Escape Sharp Minima**: Small batches still retain the ability to escape sharp minima effectively, as the randomness from the small subset introduces variability in the gradient estimates. This characteristic enables the model to explore more of the loss landscape while also converging steadily.

5.5.3 Large Batch Size

In contrast, large batch sizes present different dynamics during the training process:

- **Stable Updates**: When using large batch sizes, gradients are computed over a substantial portion of the training data. This leads to more stable and consistent parameter updates, as the averages of the gradients are less susceptible to fluctuations caused by outliers or noise.
- **Faster Convergence**: The stability of the gradients often results in faster convergence during training, as the model receives more reliable updates that reflect the general direction of the loss landscape. Consequently, the training process tends to exhibit smoother accuracy curves with less variability.
- **Risk of Sharp Minima**: However, large batch sizes can have drawbacks, particularly the tendency to converge towards sharp minima. While the stability of updates can be advantageous, it may also lead to the model settling into less optimal regions of the loss landscape, resulting in poor generalization to unseen data.
- **Computational Considerations**: While large batch sizes can speed up computation due to fewer updates per epoch, they also come with higher memory demands. Processing the entire batch at once requires more resources, which may become a limiting factor, particularly with larger datasets or limited hardware capabilities.

5.5.4 Conclusion

The experiments clearly demonstrate that the batch size significantly impacts the training effectiveness of the model. Smaller batches (e.g., 1 or 10) can provide faster convergence but may introduce instability. Medium batch sizes (like 100 or 200) offer a good balance between convergence speed and stability. Larger batch sizes (such as 400 or 800) may lead to slow convergence but tend to provide better generalization capability.

Based on the experimental results, we recommend selecting an appropriate batch size according to the characteristics of the dataset and the requirements of the model in practical applications.