CS324: Deep Learning
Assignment 1

Wai Yan Kyaw (12312638)

1 Part I: The Perceptron

1.1 Theoretical Analysis

1.1.1 Mathematical Model

The mathematical expression for the perceptron is:

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

It takes an input x (a vector of features), then calculates a linear combination: $z = w \cdot x + b$, where w is the weight vector and b is the bias. The activation function (sign function) is applied as:

$$f(x) = \text{sign}(z)$$

to output a prediction:

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

1.2 Loss Function

To train the model, we need a loss function. A simple approach is to directly count the number of misclassified points. The corresponding loss function is:

$$\begin{aligned} L_1(w, b) &= \sum_{i=1}^{N} -y_i \cdot f(x_i) \quad (\text{when } y_i \cdot f(x_i) < 0) \\ &= \sum_{i=1}^{N} -y_i \cdot \text{sign}(w \cdot x_i + b) \quad (\text{when } y_i \cdot \text{sign}(w \cdot x_i + b) < 0) \end{aligned} \tag{1}$$

However, the sign function is not differentiable. Therefore, we choose a loss function that is the algorithm choosing the total distance from the misclassification point to the hyperplane S:

$$\begin{aligned} L_2(w, b) &= \sum_{i=1}^{N} \frac{1}{\|w\|} |w \cdot x_i + b| \quad (\text{when } y_i \cdot (w \cdot x_i + b) < 0) \\ &= -\frac{1}{\|w\|} \sum_{i=1}^{N} y_i \cdot (w \cdot x_i + b) \quad (\text{when } y_i \cdot (w \cdot x_i + b) < 0) \end{aligned} \tag{2}$$

And we can remove a normalized vector, $\frac{1}{\|w\|}$ for the purpose of optimization and simplified math without changing the core of the problem. Therefore, the final loss function is:

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

1.2.1 Derivation of the Loss Function

To perform gradient descent, we need the derivative of the loss function. We use final loss L_3:

$$L(w, b) = -\sum_{x_i \in M} y_i(w \cdot x_i + b)$$

where M is the set of all misclassified points.
The gradient with respect to w is:

$$\nabla_w L(w, b) = -\sum_{x_i \in M} y_i x_i$$

The gradient with respect to b is:

$$\nabla_b L(w, b) = -\sum_{x_i \in M} y_i$$

1.3 Implementation and Results

1.3.1 Task 1

My implementation for the dataset is as follows:

```
            def generate_dataset(mean_pos, mean_neg, cov_scale=0.5):
                cov = [[cov_scale, 0], [0, cov_scale]]

                pos_points = np.random.multivariate_normal(mean_pos, cov,
                   100)
                neg_points = np.random.multivariate_normal(mean_neg, cov,
                   100)

                pos_labels = np.ones(100)
                neg_labels = -np.ones(100)

                all_points = np.vstack((pos_points, neg_points))
                all_labels = np.hstack((pos_labels, neg_labels))

                indices = np.random.permutation(200)
                all_points = all_points[indices]
                all_labels = all_labels[indices]

                train_points = all_points[:160]
                train_labels = all_labels[:160]
                test_points = all_points[160:]
                test_labels = all_labels[160:]

                return train_points, train_labels, test_points,
                   test_labels
```

Listing 1: Dataset Generation Function

I use `np.random.multivariate_normal()` method to create two Gaussian distributions (100 points for positive labels and 100 points for negative labels), then combine to be `all_points` and `all_labels`, shuffle, and split dataset into 160 points for the training and 40 points for test.

1.3.2 Task 2

I implemented the perceptron by following this pseudocode given by the perceptron tutorial pdf: 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 $= 1 \dots T$:
    a. Compute the predictions of Perceptron of the whole training set.
    b. Compute the gradient of the loss function with respect to $w$:
       $\nabla = -\frac{1}{N} \sum (x_i y_i), \quad \text{for sample } i\text{: } p_i y_i < 0$
       where $p_i$ is the $i$th prediction, $y_i$ is the related ground truth of sample $i$, $N$ is the number of misclassification points.
    c. Update $w \leftarrow w - \text{lr} \cdot \nabla$
3. Return $w$
```

My code implementation for forward method is:

```python
                    def forward(self, input_vec):
                        N = input_vec.shape[0]
                        X_with_bias = np.c_[input_vec, np.ones((N, 1))]
                        z = np.dot(X_with_bias, self.weights)
                        return np.where(z >= 0, 1., -1.)
```

Listing 2: Perceptron Forward Pass

The forward function takes an input vector, concatenates a bias term by using `np.c`, and computes the weighted sum. Then, it uses a sign function which outputs 1 if the sum is non-negative and -1 otherwise by using `np.where`.

My code implementation for train method is:

```python
                    def train(self, training_inputs, labels):
                        training_inputs = np.array(training_inputs)
                        N = training_inputs.shape[0]
                        X_with_bias = np.c_[training_inputs, np.ones((N, 1))]
                        for epoch in range(self.max_epochs):
                            preds = self.forward(training_inputs)
                            misclassified = (preds * labels < 0)
                            if not np.any(misclassified):
                                print(f"Converged at epoch {epoch}")
                                break
                            mis_X = X_with_bias[misclassified]
                            mis_y = labels[misclassified]
                            num_mis = len(mis_y)

                            gradient = - (1 / num_mis) * np.sum(mis_y[:, np.
                               newaxis] * mis_X, axis=0)
                            self.weights -= self.learning_rate * gradient
                        else:
                            print("Reached max(100) epochs")
```

Listing 3: Perceptron Training (Batch Gradient Descent)

For training, I use batch gradient descent approach to calculate an average gradient from all misclassified samples. The algorithm stops training when there is no misclassifications, indicating convergence. Otherwise, it calculates a gradient. Then I update the weights by `weights -= learning_rate * gradient`.

1.3.3 Task 3 & 4

Table 1: Data Set Feature

ID	Mean Pos	Mean Neg	Cov Pos	Cov Neg	Description
1	$[2, 2]$	$[-2, -2]$	$[[0.5, 0], [0, 0.5]]$	$[[0.5, 0], [0, 0.5]]$	Separable
2	$[2, 2]$	$[2.5, 2.5]$	$[[0.5, 0], [0, 0.5]]$	$[[0.5, 0], [0, 0.5]]$	Close Means
3	$[2, 2]$	$[-2, -2]$	$[[8.0, 0], [0, 8.0]]$	$[[8.0, 0], [0, 8.0]]$	High Variance
4	$[2, 2]$	$[2.5, 2.5]$	$[[8.0, 0], [0, 8.0]]$	$[[8.0, 0], [0, 8.0]]$	Close Means + High Variance

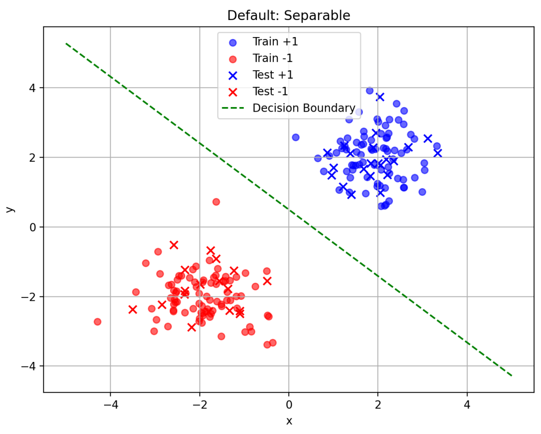
I generate 4 different sets of data to evaluate the training performance. The test cases are:

1. Separable
2. Close Means
3. High Variance
4. Close Means + High Variance

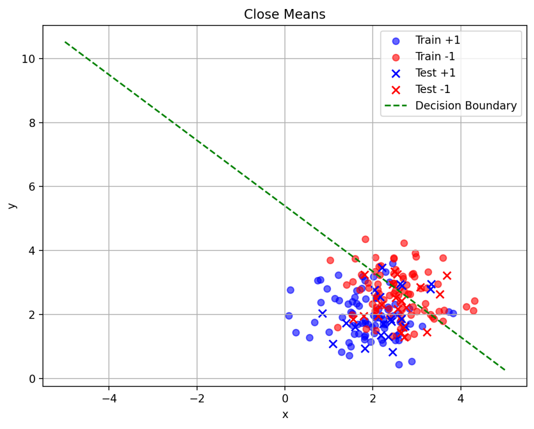
1.3.4 Analysis of the Results

- **TestCase 1:** Since mean positive and mean negative samples are separable with no overlapping, our algorithm finds the decision boundary easily, converging at epoch 1 with 1.000 test accuracy.
- **TestCase 2:** When means are too close, significant overlapping occurs between the two classes, resulting in 0.525 test accuracy even after 100 epochs.
- **TestCase 3:** Using the same separable means as TestCase 1 but with higher variance, some overlapping points appear, reducing accuracy to 0.850 after 100 epochs.
- **TestCase 4:** Using the same close means as TestCase 2 but with higher variance, the increased spread further reduces accuracy to 0.375 after 100 epochs.

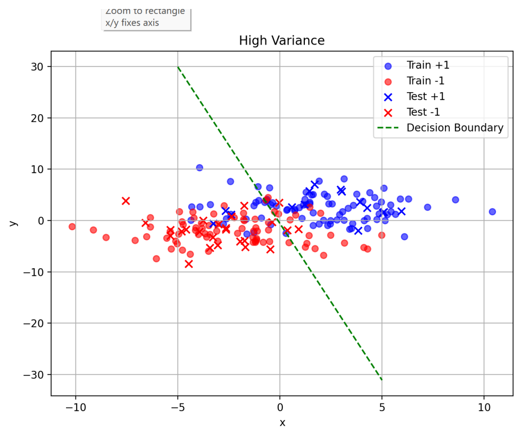
Note: All training and test points are visible in figure 1. For simplicity, only test accuracy is reported in this analysis.



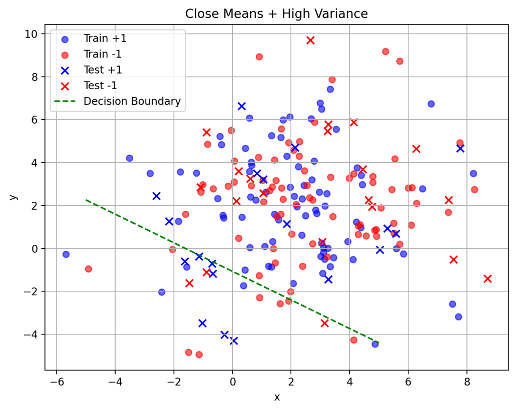
(a) TestCase 1



(b) TestCase 2



(c) TestCase 3



(d) TestCase 4

Figure 1: Data distributions for four test cases

2 Part II: The Multi-layer Perceptron

2.1 Theoretical Analysis

For hidden layers ($l = 1$ to $N - 1$), the mathematical model for MLP in the forward process is as follows:

2.1.1 Forward Process

1. **Linear Layer**

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

   where $W^{(l)}$ is the weight matrix and $b^{(l)}$ is the bias vector. Each layer l takes the output from the previous layer ($x^{(l-1)}$) and performs a linear transformation.

2. **ReLU Activation**

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

If we use a linear activation function, the neural network would simply combine inputs and output them. Therefore, we need a non-linear activation function (here, ReLU) to learn complex patterns. It outputs the input if positive and zero otherwise.

3. **Output Layer (Softmax)**

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

The softmax function converts the raw output scores ($\tilde{x}^{(N)}$) into a probability distribution:

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

Each output value lies between 0 and 1, and all values sum to 1. This works effectively with the Cross Entropy loss.

4. **Computing Loss (Cross Entropy)**

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

where t is the one-hot encoded true label. We use Cross Entropy Loss to measure the error between predicted and true labels.

2.1.2 Backward Process

To update parameters using gradient descent, we compute the gradient of the loss function with respect to all parameters via backpropagation:

1. **Loss and Softmax Layer**

$$\delta^{(N)} = \frac{\partial L}{\partial \tilde{x}^{(N)}} = x^{(N)} - t$$

2. **ReLU Layer**

$$\frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \text{diag}(1_{\{\tilde{x}^{(l)}>0\}})$$

where $1_{\{\tilde{x}^{(l)}>0\}}$ returns 1 where $\tilde{x}^{(l)} > 0$ and 0 otherwise.

3. **Linear Layer**

$$\frac{\partial L}{\partial W^{(l)}} = \delta^{(l)} (x^{(l-1)})^\top, \quad \frac{\partial L}{\partial b^{(l)}} = \delta^{(l)}$$

where

$$\delta^{(l)} = (W^{(l+1)})^\top \delta^{(l+1)} \odot 1_{\{\tilde{x}^{(l)}>0\}}$$

derived from:

$$\delta^{(l)} = \frac{\partial L}{\partial \tilde{x}^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}}$$

$$\delta^{(l)} = \delta^{(l+1)} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}}$$

$$\frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} = (W^{(l+1)})^\top$$

2.2 Implementation

2.2.1 Task 1

I implemented the forward process and backward process as follows:
1. Linear Layer Implementation
Forward

```
            def forward(self, x):
                self.x = x
                return x @ self.params['weight'] + self.params['bias']
```

Related math formula is:

$$\tilde{x} = x\overline{W} + b$$

Backward

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

Related math formula is:

$$\frac{\partial L}{\partial W} = x^{\top}\delta \tag{3}$$

$$\frac{\partial L}{\partial b} = \sum_{batch} \delta \tag{4}$$

$$\frac{\partial L}{\partial x} = \delta W^{\top} \tag{5}$$

where $\delta = \frac{\partial L}{\partial \tilde{x}}$ is the incoming gradient.
2. ReLU Implementation
Forward

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

Related math formula is:

$$x_{out} = \max(0, x_{in})$$

Backward

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

Related math formula is:

$$\frac{\partial L}{\partial x_{in}} = \delta \odot 1_{\{x_{in}>0\}}$$

3. SoftMax Activation Implementation
Forward

```python
            def forward(self, x):
                x_shifted = x - np.max(x, axis=1, keepdims=True)
                exp_x = np.exp(x_shifted)
                self.output = exp_x / np.sum(exp_x, axis=1, keepdims=True)
                return self.output
```

Related math formula is:

$$x_{shifted} = x - \max(x)$$

$$\text{softmax}(x_i) = \frac{\exp(x_{shifted,i})}{\sum_j \exp(x_{shifted,j})}$$

4. CrossEntropy Loss Implementation
Forward

```python
            def forward(self, x, y):
                self.pred = x
                self.true = y

                epsilon = 1e-15
                x_clipped = np.clip(x, epsilon, 1 - epsilon)

                loss = -np.sum(y * np.log(x_clipped)) / x.shape[0]
                return loss
```

Related math formula is:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{i,j}\log(p_{i,j})$$

where N is batch size and C is number of classes.
Backward

```python
            def backward(self):
                batch_size = self.pred.shape[0]
                return (self.pred - self.true) / batch_size
```

$$\frac{\partial L}{\partial z} = p - y$$

This elegant simplification occurs when SoftMax is combined with CrossEntropy loss.

`mlp_numpy.py` is basically the implementation of the combination of the layers in `modules.py` and update the parameters.

2.2.2 Task 2

I created 2D non-linearly separable data using the `make_moons` dataset with 0.2 noise. You can see data distributions in `figure 2`. Then, I reshaped `y` to one-hot encoding. After that, I used the `train_test_split` function from `scikit-learn` to split our dataset into 80% for training data and 20% for test data.

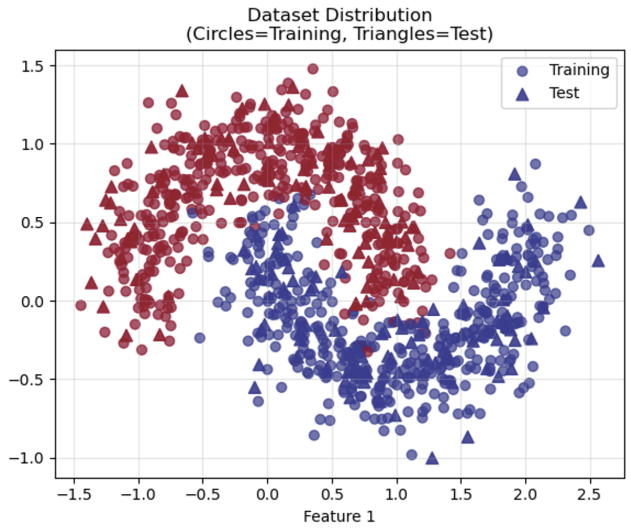
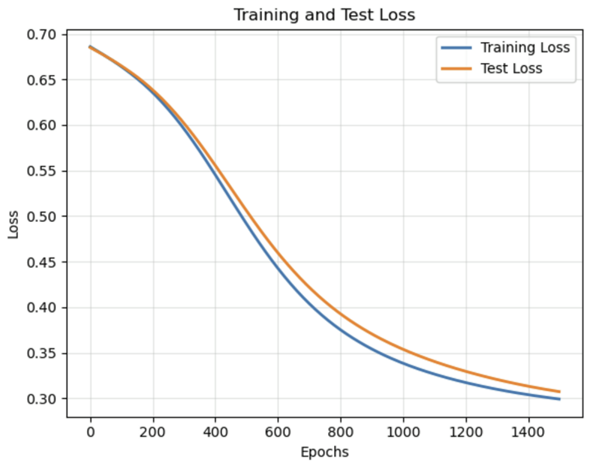


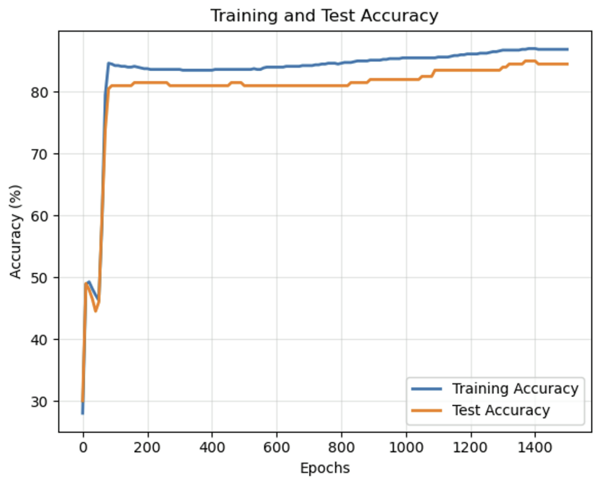
Figure 2: Moon Dataset Distribution (Training and Test Data)

2.2.3 Task 3

I created `mlp_visualization.ipynb` notebook and when you run the code block, the results are as follows:



(a) Training and Test Loss



(b) Training and Test Accuracy

Figure 3: MLP with BGD Results

2.2.4 Analysis of Model Performance

Loss Convergence The architecture for MLP is one hidden layer with 20 neurons. By using default parameters with batch gradient descent, in both training and test data, we can see the loss curves with smooth convergence, starting from 0.6994 at step 0 due to random weight initialization, to 0.3037 by step 1499, which is a 56.5% overall reduction. The final training accuracy is 0.3123 and final test accuracy is 0.3037, which indicates no signs of overfitting.

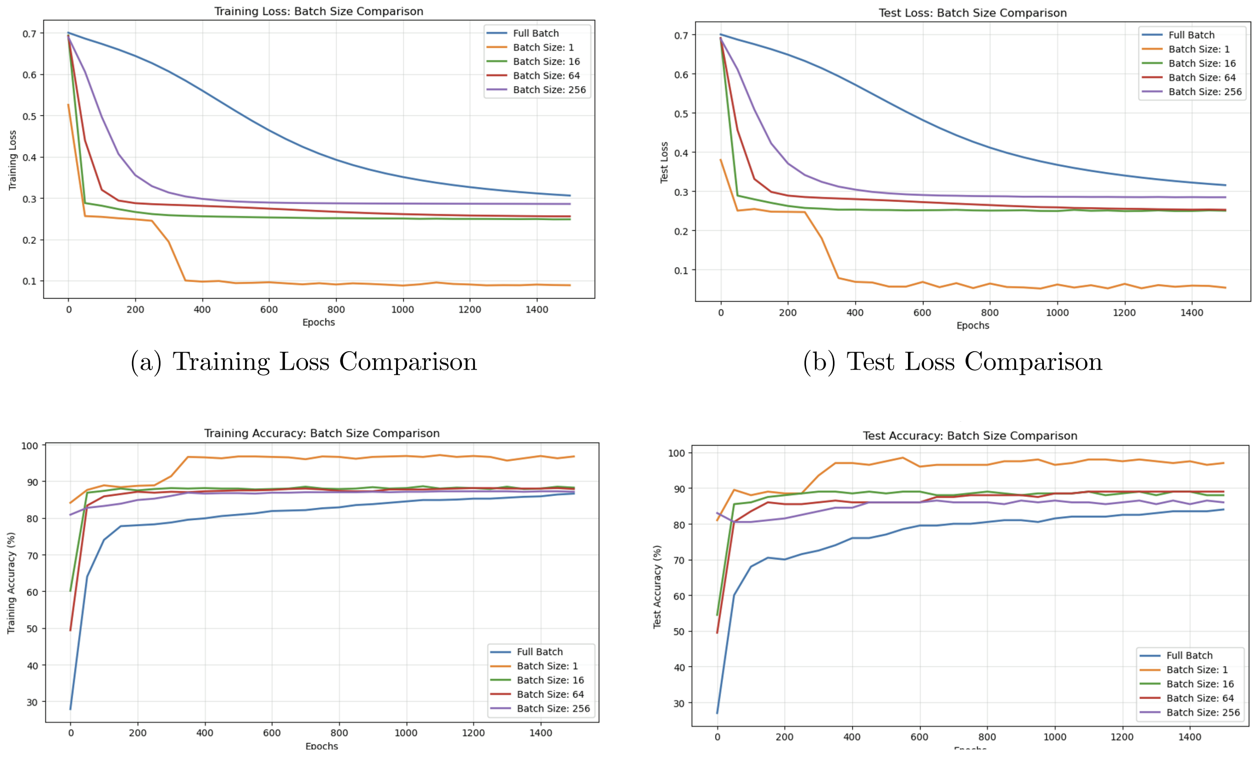
Accuracy Progression The accuracy percantage for training and testing starts at low values(step 0: train 28.00%, test 30.00%) and shows significant rise to approximately 80% after 80 steps, which is followed by a plateau with small increases to 86.88% training and 84.50% test accuracy. This pattern shows that the model captures the moons dataset's non-linear decision boundary in early phases where large error corrections occur beacuse the network needs to adjust the weights to separate the interleaved clusters effectively, but subsequent refinements are constrained by diminishing gradients and the noise in the dataset, which leads to stepwise improvements and minor fluctuations. The 2% gap between the final accuracy of training and test data shows there is only minimal overfitting.

3 Part III: stochastic gradient descent

3.1 Task 1

In `train_mlp_numpy.py`, in train function, in addition to the batch gradient descent I implemented in Part 2, I also implemented to accept the batch number from user input, if it's 1 , obviously it's stochastic gradient descent. If it's other number, it's mini batch gradient descent.

3.2 Task 2



(a) Training Loss Comparison

(b) Test Loss Comparison

(c) Training Accuracy Comparison

(d) Test Accuracy Comparison

Figure 4: Batch Size Comparison: Stochastic vs Mini-Batch vs Batch Gradient Descent

3.2.1 Analysis of Batch Size Effects

Loss Convergence for Different Batch Sizes The loss decreases slowly and smoothly from approximately 0.7 to 0.3 over 1500 epochs for a full batch because it uses the entire dataset for each update. That causes accurate gradients with infrequent updates, which brings about steady but slow convergence. For batch size 1(stochastic gradient), the loss drops rapidly in the beginning up to under 0.1 with significant volatility. Since each update is based on a single data, we see noisy and highly variable gradients which achieve lower loss faster. For mini-batch 16, 64 and 256 sizes, the loss decreases faster than the full batch but the curve is smoother than stochastic one. Smaller mini-batches(16 and 64) introduces good noise for better exploration while larger ones gives stability which is similar to full batch but with more frequent updates. The conclusion is that small batches(1 or 16) makes quicker learning through varied gradients which can reduce loss significantly. In contrast, large batches(256 or full) converge slower because they average gradients excessively.

Accuracy Progression for Different Batch Sizes The full batch starts low at 28% and increases slowly to 86% training and 84% test accuracy because of its stable updates which learn patterns steadily without major jumps. This is a good generalization but slow adaptation to complex moons shape. Batch size 1 grows rapidly to over 96% training and 97% test accuracy although it has some oscillations. For mini-batches, size 16 reaches 88% for both training and test, size 64 reaches 88% for training and 89% for test and 87% training and 86% training for size 256. Medium batch size like 64 perform optimally because it combines noise for faster improvement with stability to avoid larger errors. Small batches explore the parameter space more thoroughly which boost accuracy by adapting to data. Large batches generalize well but can plateau at lower accuracy if updates are overly averaged.

4 Instructions to Run Code

4.1 Perceptron

Run the following command in terminal:

```
python perceptron.py
```

4.2 Multi-Layer Perceptron

Ensure the following packages are installed: `numpy`, `matplotlib`, and `scikit-learn`.

4.2.1 Command Line Execution

Run the following command in terminal:

```
python train_mlp_numpy.py --batch_size <BATCH_SIZE>
```

where `<BATCH_SIZE>` can be:

- 1 for Stochastic Gradient Descent (SGD)
- 16, 64, 256 for Mini-Batch Gradient Descent
- Omitted for Full Batch Gradient Descent (default)

4.2.2 Jupyter Notebook Execution

Run `mlp_visualization.ipynb` in Jupyter Notebook to view MLP training visualizations.