DEEP LEARNING (CS324)
ASSIGNMENT 1 REPORT

Southern University of Science and Technology

Instructor: Prof. Zhang Jianguo

Student Name: THA Sreyny

Student ID: 12113053

October 15, 2024

1. INTRODUCTION

A neural network is a key concept in deep learning, designed to mimic the way the human brain processes information. Composed of layers of interconnected nodes (neurons), neural networks transform input data through a series of mathematical operations to produce an output. Each layer in the network extracts increasingly complex features from the input, allowing the model to learn patterns and make predictions. Neural networks are highly effective in tasks like image recognition, natural language processing, and decision-making. Through techniques like backpropagation and optimization, they adjust weights to minimize errors and improve accuracy in learning tasks.

A Perceptron is the simplest type of neural network, consisting of a single neuron that classifies input data by computing a weighted sum and applying an activation function. It is ideal for binary classification problems. However, its simplicity limits its ability to handle complex or non-linear datasets that are not linearly separable. To overcome this, the Multilayer Perceptron (MLP) extends the Perceptron by adding multiple hidden layers between the input and output using forward propagation to calculate the error and backpropagation to adjust the weight and bias of the input to make the performance.

In this assignment, I will implement both Perceptron and MLP using methods such as forward propagation, backpropagation, and gradient descent. Forward propagation will be used to calculate the output of the network, while backpropagation will adjust the weights based on the error between predicted and actual outputs. Gradient descent will optimize these weights to minimize the overall error and improve model performance.

2. MOTIVATION

The primary objectives of this project are as follows:

1. Evaluate and compare the loss and accuracy of a single-layer perceptron using Gaussian distributions with varying mean and variance.
2. Assess the accuracy of training and test data using a multilayer perceptron with default parameter values.
3. Examine the performance of the multilayer perceptron model when trained using stochastic gradient descent.
4. Analyze the impact of batch size, ranging from 1 to a relatively large value, on the multilayer perceptron model when using batch gradient descent.

3. METHODOLOGY

■ Part I: The Perceptron

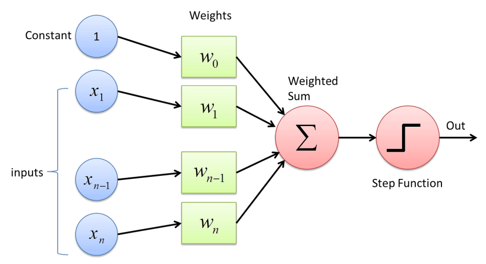
The perceptron is a binary classifier that maps input data into one of two possible outputs (1, -1) based on a linear decision boundary. The tasks involved generating a dataset, training the perceptron, and evaluating its performance on varying mean and variance of the training and testing data.

Perceptron Implementation Process

1. **Mathematical Model:** The perceptron decision is based on these formulas:

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

Figura 1. Basic Component of Perceptron



Source: SAGAR SHARMA

where the activation function is given by:

$$sign(x) = \begin{cases} 1 & \text{if } x >= 0, \\ -1 & \text{otherwise.} \end{cases}$$

Here, w represents the weights, x represents the input vector, and b is the bias.

2. **How to train a Perceptron:**

Loss Function

Since we want to classify all points correctly, a natural idea is to directly use the total number of misclassified points as the loss function:

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

Where y_i is the true label, and $f(x_i)$ represents the predicted output of the perceptron.

Gradient of the Loss Function

To update the weights and bias, we need to compute the gradient of the loss function with respect to the weights w and the bias b.

$$\nabla_w L(w, b) = -\sum_{i=1}^{N} y_i * x_i$$
$$\nabla_b L(w, b) = -\sum_{i=1}^{N} -y_i$$

Weight and Bias Update Rule

Using the gradients, the weights and bias can be updated using the following update rules:

$$w \leftarrow w - \eta \cdot \nabla_w L(w, b)$$
$$b \leftarrow b - \eta \cdot \nabla_b L(w, b)$$

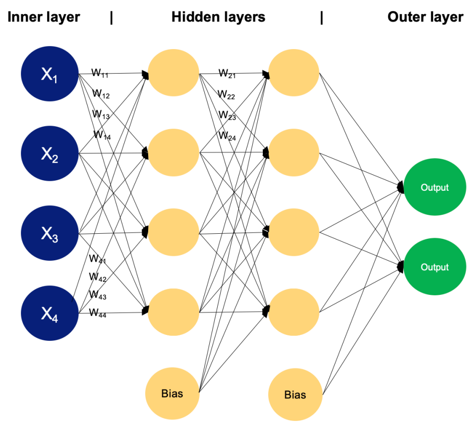
Where η is the learning rate. These updates are applied iteratively until a set number of epochs have passed. This ensures that the perceptron adjusts the weights and bias to reduce the classification error on the training data.

3. **Training the Perceptron:** The perceptron is trained by iterating over and over the training dataset. For each point, the weights and bias are updated until a set number of iterations (epochs) is reached.
4. **Testing the Perceptron:** Once trained, the perceptron is tested on a separate test set. For each test point, the perceptron computes the output and compares it to the true label. The classification accuracy is calculated as the percentage of correctly classified points.

This implementation process ensures that the perceptron can learn a linear decision boundary that separates two classes in a dataset, adjusting its weights based on the classification error.

■ Part II: Multiple Layers Perceptron (MLP)

Figura 2. Basic Component of MLP



Source: Campus Ambassador

- Each layer $l = 1, \dots, N$ 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):

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

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

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

$$L(x^{(N)}, t) = -\sum_{k=1}^{d_N} t_k \log x_k^{(N)}$$

4. EXPERIMENT

Perceptron

Step 1: Generate a Dataset of Points in $\mathbb{R}^2$ To generate the dataset, we define two Gaussian distributions and sample 100 points from each distribution. This results in a dataset containing 200 points in total, 100 from each distribution.

The dataset is then split as follows:

- 80 points per distribution are used for training (160 points in total).
- 20 points per distribution are used for testing (40 points in total).

No.	Distribution 1		Distribution 2	
	mean	var	mean	var
1	2	1	4	1
2	2	1	4	2
3	2	1	2.1	1
4	2	30	3	30

Tabela 1. Experiment Datasets

Step 2: implemented a single layer of perception with 100 epochs and 0.01 learning rate.

Step 3: Train the dataset and test dataset with the trained model

Step 4: Evaluate loss and accuracy of training and testing set

Result of Perceptron

No.	Training Accuracy		Testing Accuracy	
	Min	Max	Min	Max
1	0.74	0.95	0.76	0.93
2	0.69	0.88	0.63	0.85
3	0.5	0.57	0.43	0.6
4	0.44	0.53	0.4	0.53

Tabela 2. Experiment Datasets

Multiple Layers Perceptron

Step 1: Data Preparation

In Step 1, a dataset comprising 1,000 two-dimensional points is generated using the `make_moons` method from the `scikit-learn` library. This dataset contains two classes, identified as class 0 and class 1. The dataset is subsequently split into a training set comprising 20% of the samples and a testing set containing the remaining 80%. The labels are encoded using one-hot encoding, resulting in a matrix of dimensions $M \times C$, where M represents the number of samples and C denotes the number of classes. Datasets for

experiment: Case 1: make moons(n samples=1000, shuffle=True, noise=0.2, random state=42) Case 2: make moons(n samples=1000, shuffle=True, noise=0.4, random state=42)

Step 2: Train and test the MLP model using following default hyper-parameters:

- `dnn_hidden_units` $= 20$
- `learning rate` $= 1 \times 10^{-2}$
- `max_steps` $= 1500$
- `eval_freq` $= 10$
- `batch_size_default` $= 1000$

Step 3: The MLP model is trained and tested again with the same hyper-parameters, but this time the batch size = 1, which using stochastic gradient descent

Step 4: The MLP model is trained and tested again with the same hyper-parameters, but this time the batch size = 100

Step 5: The MLP model is trained and tested again with the same hyper-parameters, but this time the batch size = 400

MLP Model Performance Results

The results of the MLP model for different batch sizes are summarized in Table 3.

Step	BS	TR_A	TR_L	TS_A	TS_L
1	1000	89.62%	23%	87.5%	24%
2	1	97.75%	5.0%	98.0%	5.0%
3	10	89.00%	30.0%	87.0%	31.0%
4	50	89.25%	24.0%	87.0%	23.0%
5	100	89.5%	23.0%	86.5%	25.0%
6	200	83.75%	28.0%	85.5%	28.0%
7	400	89.25%	23.0%	88.5%	22.0%

Tabela 3. Results of MLP Model with Different Batch Sizes

Note:
BS = Batch Size,
TR_A = Training Accuracy,
TR_L = Training Loss,
TS_A = Testing Accuracy,
TS_L = Testing Loss.

5. ANYLYSIS RESULT

5.1. Percepton

Scenario 1 (No. 1) In this scenario, the two distributions have similar means and variances:

- Distribution 1: mean = 2, variance = 1
- Distribution 2: mean = 4, variance = 1

The similar variances and small difference in means indicate that the two distributions are relatively close to each other in the feature space. This makes the data more linearly separable, which aligns with the strong performance of the perceptron model in this scenario. The training accuracy range of 0.74 to 0.95 and the testing accuracy range of 0.76 to 0.93 suggest that the perceptron was able to learn an effective linear decision boundary to separate the two classes.

Scenario 2 (No. 2) In this scenario, the two distributions have different means and variances:

- Distribution 1: mean = 2, variance = 1
- Distribution 2: mean = 4, variance = 2

The difference in variances introduces more overlap between the distributions, making the data less linearly separable. This is reflected in the performance of the perceptron model, with a training accuracy range of 0.69 to 0.88 and a testing accuracy range of 0.63 to 0.85. The model still performs reasonably well, but the higher complexity of the data distribution likely leads to some difficulty in generalization, as seen in the lower testing accuracy compared to Scenario 1.

Scenario 3 (No. 3) In this scenario, the two distributions have similar means and variances:

- Distribution 1: mean = 2, variance = 1
- Distribution 2: mean = 2.1, variance = 1

The close means and similar variances result in a high overlap between the two distributions, making the data non-linearly separable. This is a challenging scenario for the perceptron model, which is limited to learning linear decision boundaries. The training accuracy range of 0.5 to 0.57 and the testing accuracy range of 0.43 to 0.6 indicate that the perceptron struggles to perform well on this type of dataset.

Scenario 4 (No. 4) In this scenario, the two distributions have different means and variances:

- Distribution 1: mean = 2, variance = 30
- Distribution 2: mean = 3, variance = 30

The large difference in means and the significantly different variances result in a complex data distribution that is not easily separable by a linear decision boundary. This is reflected in the poor performance of the perceptron model, with a training accuracy range of 0.44 to 0.53 and a testing accuracy range of 0.4 to 0.53. The perceptron is unable to learn an effective decision boundary for this type of non-linear data distribution.

Conclusion The analysis of the mean and variance of the experiment datasets provides valuable insights into the performance of the perceptron model. The perceptron excels when the data is linearly separable, as in Scenario 1, but struggles when the data distributions become more complex, as seen in Scenarios 2, 3, and 4. This highlights the inherent limitations of the perceptron model and the need to consider more advanced machine learning techniques when dealing with non-linearly separable data.

5.2. MLP

The performance of the MLP model shows notable variation across different batch sizes (BS) in terms of both training and testing accuracy, as well as loss:

- **Batch Size = 1000 (Step 1):**
  The model achieves moderate accuracy both

on the training set (89.62%) and the testing set (87.5%), with relatively high loss values (23% for training, 24% for testing). This suggests that the model is learning but with some inefficiency in optimization, possibly due to the large batch size, which can slow convergence.

- **Batch Size = 1 (Step 2):**
The model performs exceptionally well with near-perfect accuracy on both the training set (97.75%) and testing set (98.0%). The corresponding loss is also very low (5.0%). This indicates that with stochastic gradient descent (small batch size), the model learns more efficiently and generalizes well to unseen data.
- **Batch Size = 10 (Step 3):**
With a smaller batch size, the model's performance drops (training accuracy: 89%, testing accuracy: 87%), and the loss values increase significantly (30% for training, 31% for testing). This might suggest that such a small batch size doesn't capture enough variance to effectively train the model.
- **Batch Size = 50 (Step 4):**
The model's accuracy is slightly improved (89.25% for training, 87% for testing), and the loss decreases compared to batch size 10, showing more stable learning with smaller loss values (24% for training, 23% for testing).
- **Batch Size = 100 (Step 5):**
Similar performance to batch size 50, with moderate accuracy (training: 89.5%, testing: 86.5%) and loss around 23-25%. The model maintains steady performance but doesn't benefit much from the smaller batch.
- **Batch Size = 200 (Step 6):**
The model's performance drops significantly, with the lowest training accuracy (83.75%) and high loss values (28% for both training and testing). This shows that the model struggles with this batch size.
- **Batch Size = 400 (Step 7):**
The model's accuracy returns to moderate levels (training: 89.25%, testing: 88.5%), and the loss is lower (23% for training, 22% for testing), indicating better performance compared to larger and smaller batch sizes, possibly due to a balance between learning speed and variance captured.

Conclusion: The MLP model performs best with a very small batch size (BS = 1), achieving the highest accuracy and lowest loss. Large batch sizes (BS = 1000) and mid-range sizes (BS = 200) tend to result in slower learning and poorer performance.

6. CONCLUSION

Experiments with single-layer and multi-layer perceptron models revealed key insights into their capabilities and limitations. The single-layer perceptron functioned well as a linear classifier for linearly separable distributions but struggled with increased complexity, particularly with overlapping classes.

Stochastic Gradient Descent (SGD) and Batch Gradient Descent are two commonly used optimization techniques to minimize the loss function in machine learning. While they differ in their approach to computing gradients, their performance can vary based on the specific problem and dataset:

- **Batch Gradient Descent** computes the gradient using the entire dataset, ensuring a smooth and stable descent towards the global minimum. Although traditionally seen as computationally expensive for large datasets, it can sometimes perform more efficiently in practice, as observed in this case, where it outperforms SGD in speed.
- **Stochastic Gradient Descent (SGD)** updates the model parameters after evaluating the gradient on each individual sample. Though generally considered faster for large datasets, it can introduce high variance, leading to less stable convergence. In this testing, SGD was observed to be slower than batch gradient descent, possibly due to the additional noise and more frequent updates.

A. APPENDIX

A.1. Perceptron

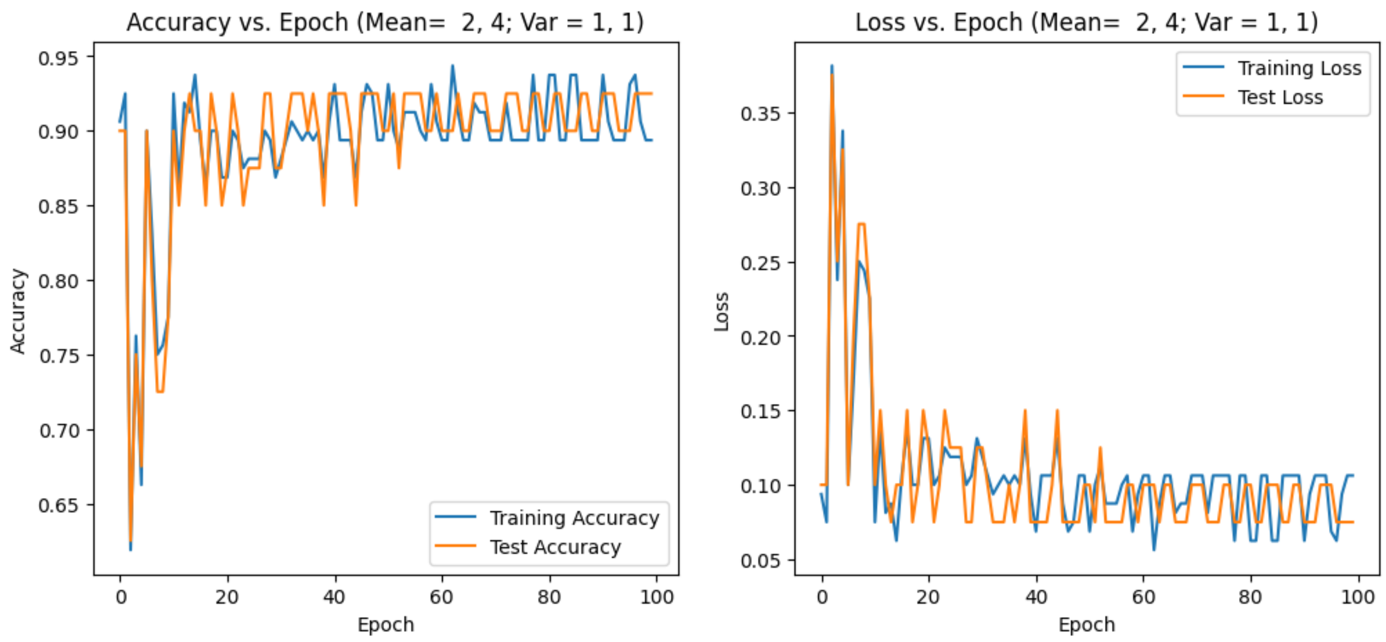


Figura 3. Case 1

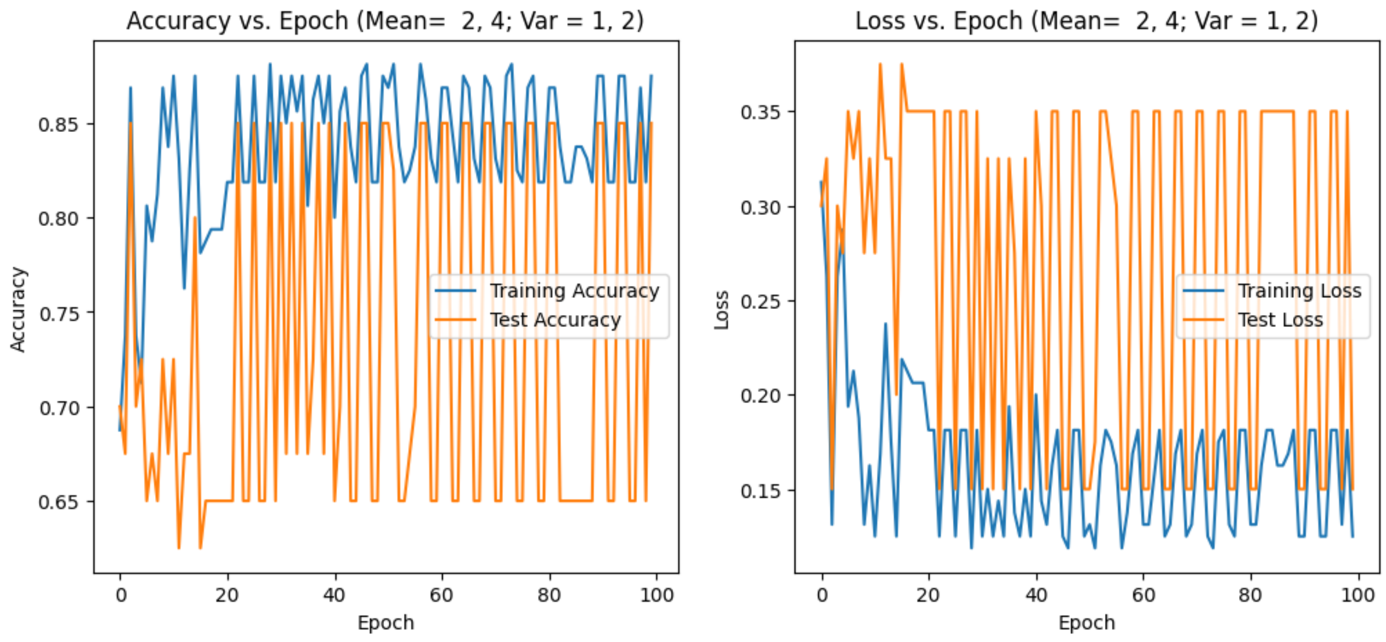
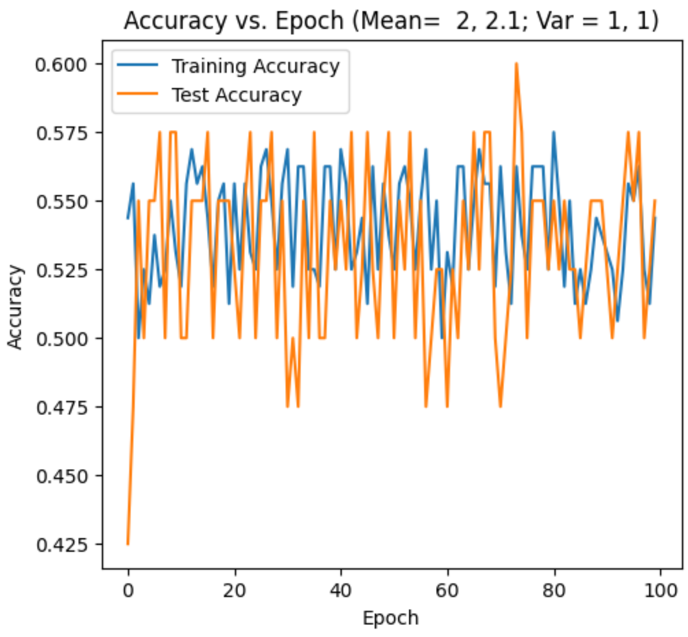
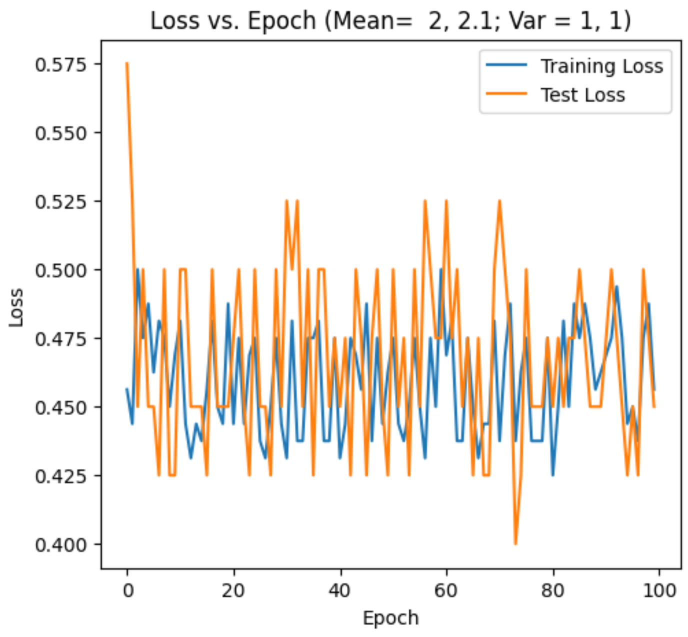


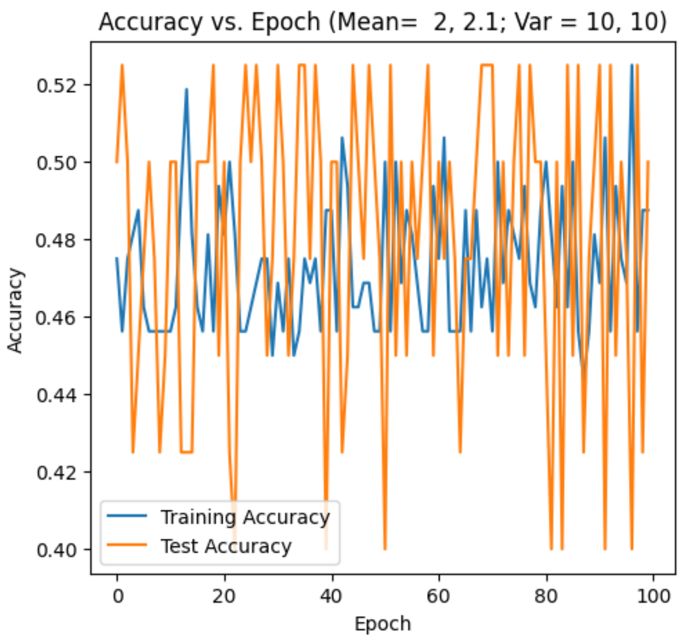
Figura 4. Description of image 2



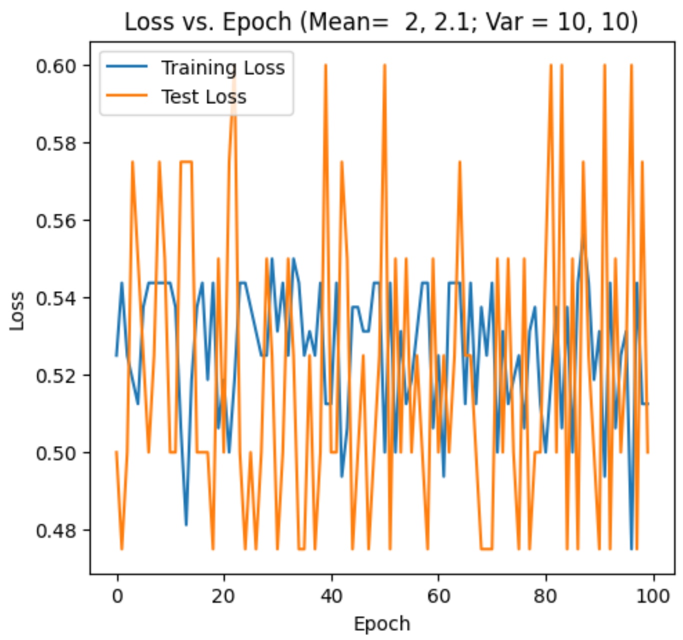
Accuracy vs. Epoch (Mean= 2, 2.1; Var = 1, 1)
Training Accuracy
Test Accuracy
Accuracy
Epoch



Loss vs. Epoch (Mean= 2, 2.1; Var = 1, 1)
Training Loss
Test Loss
Loss
Epoch

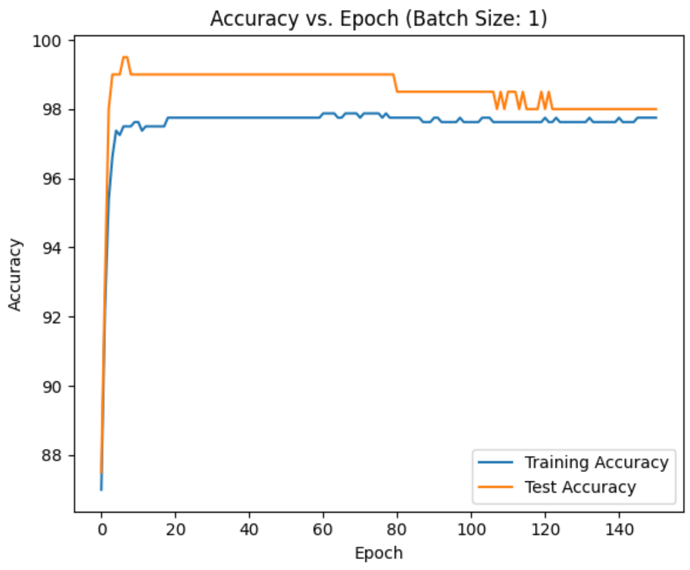


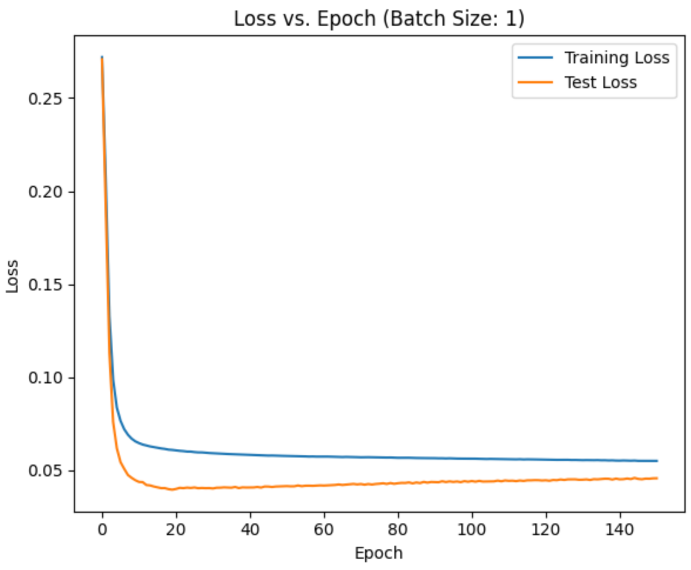
Accuracy vs. Epoch (Mean= 2, 2.1; Var = 10, 10)
Training Accuracy
Test Accuracy
Accuracy
Epoch

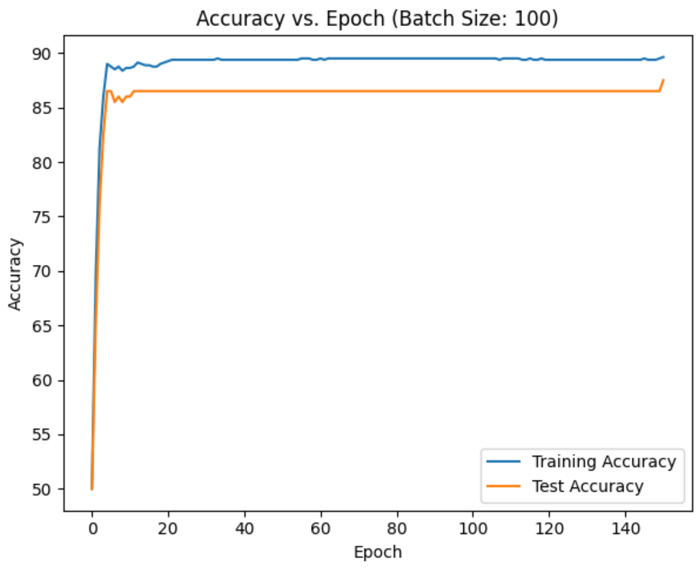


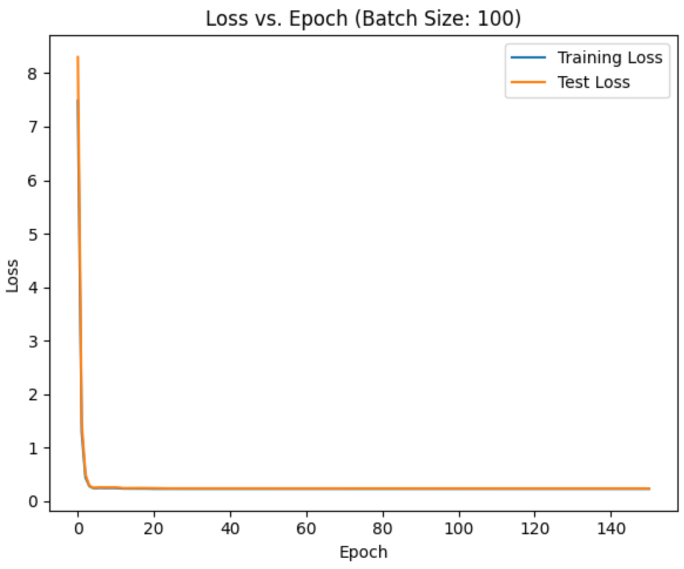
Loss vs. Epoch (Mean= 2, 2.1; Var = 10, 10)
Training Loss
Test Loss
Loss
Epoch

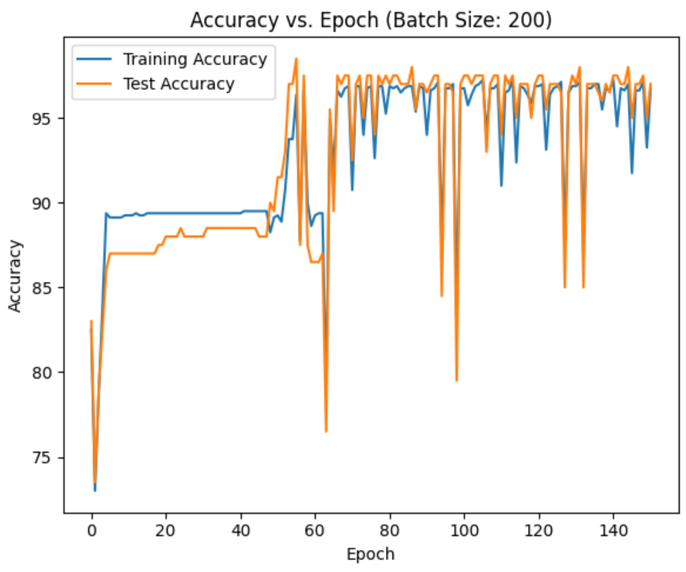
A.2. MLP

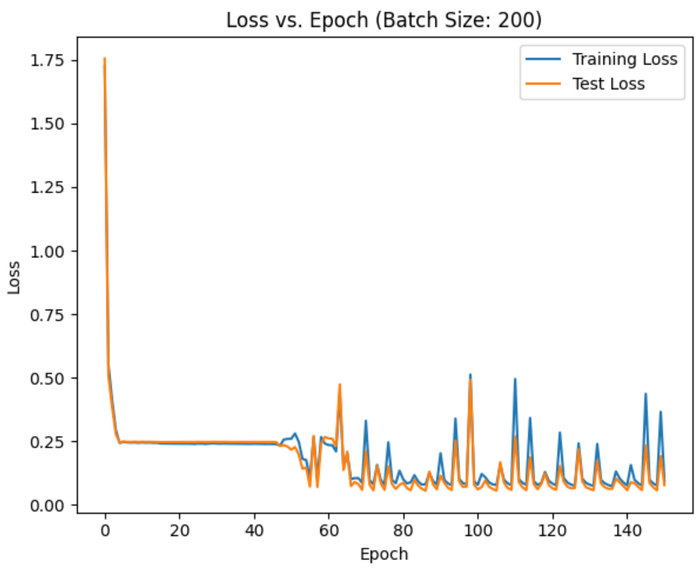

Accuracy vs. Epoch (Batch Size: 1)
Accuracy
Epoch
Training Accuracy
Test Accuracy

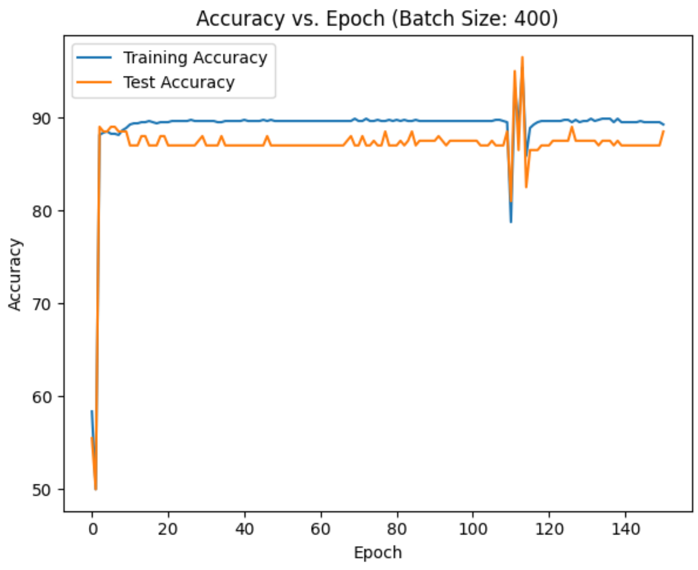

Loss vs. Epoch (Batch Size: 1)
Loss
Epoch
Training Loss
Test Loss

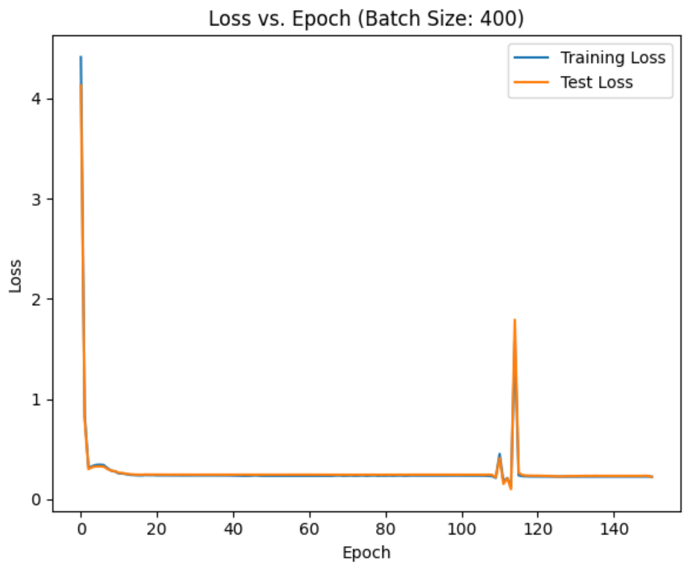

Accuracy vs. Epoch (Batch Size: 100)
Accuracy
Epoch
Training Accuracy
Test Accuracy

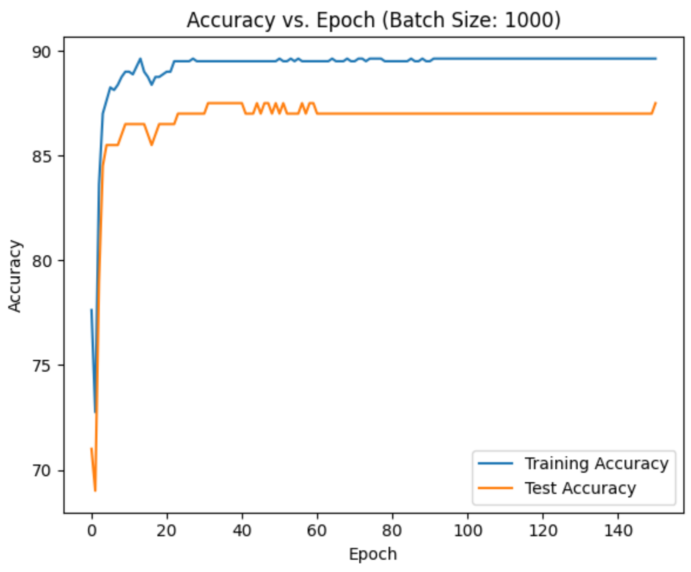

Loss vs. Epoch (Batch Size: 100)
Loss
Epoch
Training Loss
Test Loss

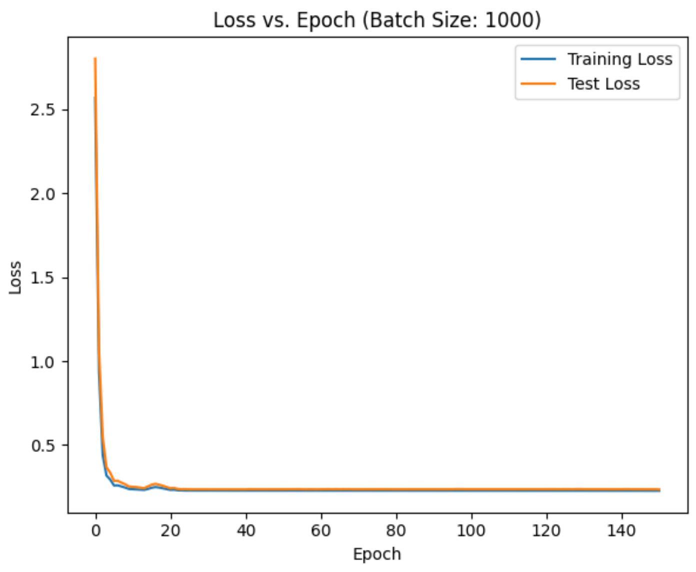

Accuracy vs. Epoch (Batch Size: 200)
Training Accuracy
Test Accuracy
Accuracy
Epoch


Loss vs. Epoch (Batch Size: 200)
Training Loss
Test Loss
Loss
Epoch


Accuracy vs. Epoch (Batch Size: 400)
Training Accuracy
Test Accuracy
Accuracy
Epoch


Loss vs. Epoch (Batch Size: 400)
Training Loss
Test Loss
Loss
Epoch


Accuracy vs. Epoch (Batch Size: 1000)
Accuracy
Epoch
90
85
80
75
70
0
20
40
60
80
100
120
140
Training Accuracy
Test Accuracy


Loss vs. Epoch (Batch Size: 1000)
Loss
Epoch
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100
120
140
Training Loss
Test Loss