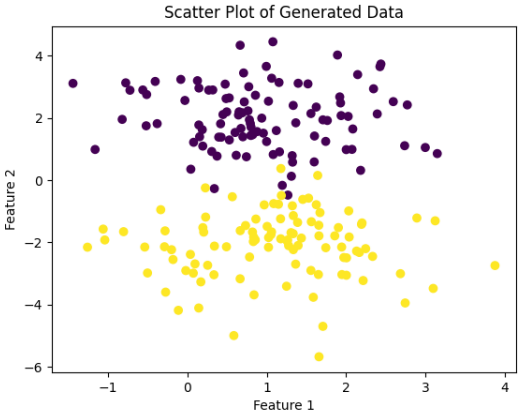
Part A

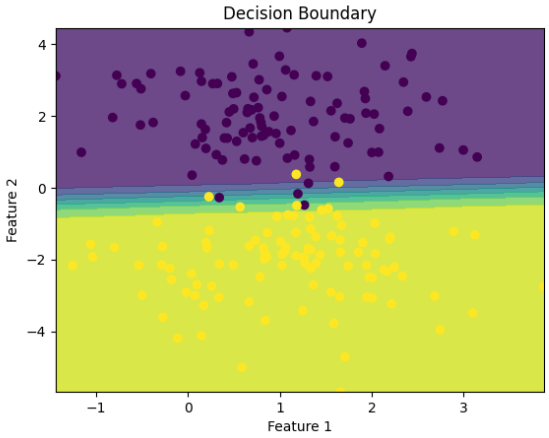


Dataset:

- **Purple Points**: These points are primarily located in the upper part of the plot, indicating one class of data with a mean of (1, 2).
- **Yellow Points**: These points are mainly found in the lower part of the plot, representing the other class with a mean of (1, -2).

This dataset will be used to train the model.

The results for different batch sizes with Epochs: 100 and Learning Rate: 0.1:



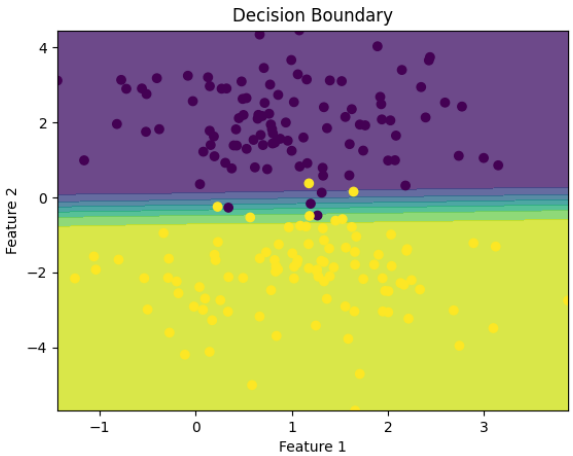
Batch Size: 1

Learning Time:
1.3126747608184814 Second

Testing Time:
0.005120754241943359 Second

Average Loss: 0.04622569119507577

Accuracy: 97.50%



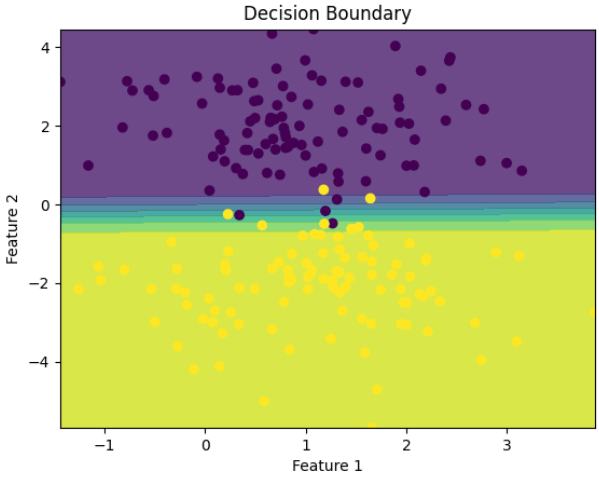
Batch Size: 2

Learning Time:
0.5515632629394531 Second

Testing Time:
0.00534510612487793 Second

Average Loss: 0.04538037149057856

Accuracy: 98.00%



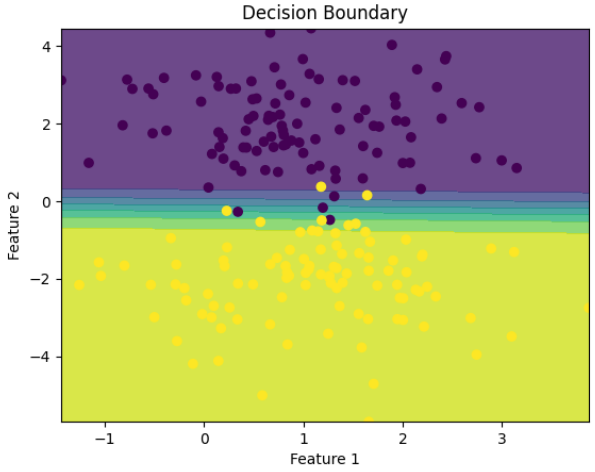
Batch Size: 4

Learning Time: 0.4141840934753418 Second

Testing Time: 0.006634235382080078 Second

Average Loss: 0.04568184200909669

Accuracy: 98.00%



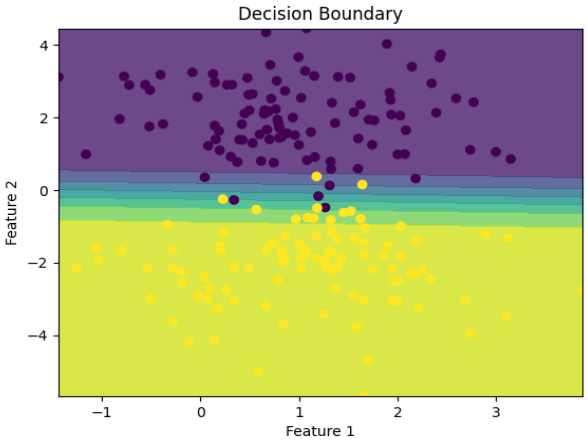
Batch Size: 16

Learning Time: 0.09339785575866699 Second

Testing Time: 0.010871171951293945 Second

Average Loss: 0.05046599760987784

Accuracy: 97.50%



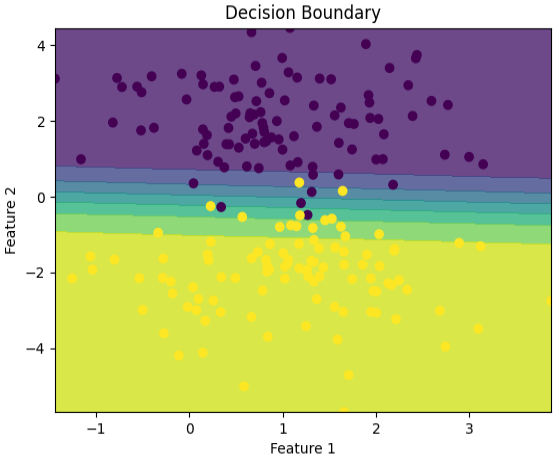
Batch Size: 32

Learning Time: 0.05539441108703613 Second

Testing Time: 0.006998300552368164 Second

Average Loss: 0.061971976771600434

Accuracy: 97.50%



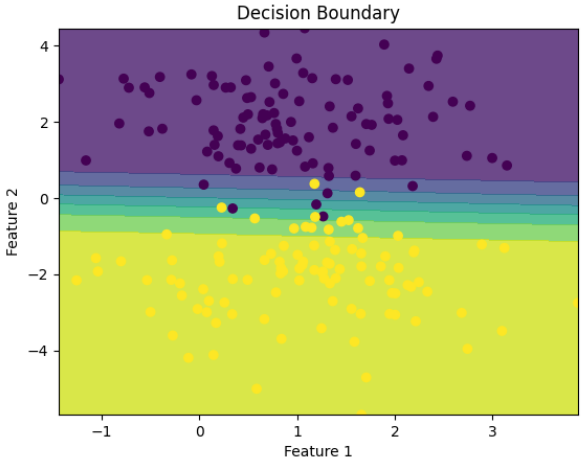
Batch Size: 64

Learning Time:
0.02043008804321289 Second

Testing Time:
0.0060040950775146484 Second

Average Loss: 0.06237504248913877

Accuracy: 97.50%



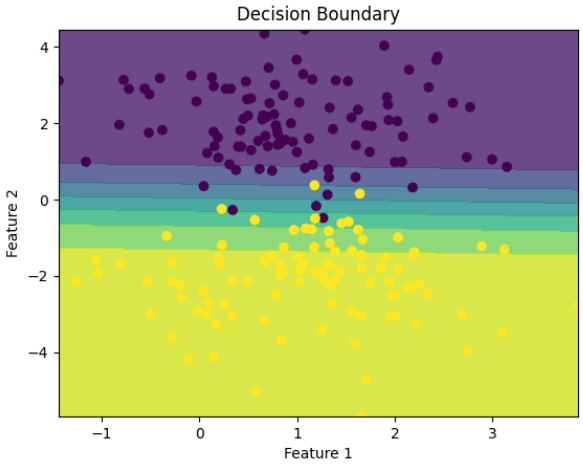
Batch Size: 128

Learning Time:
0.02241659164428711 Second

Testing Time:
0.007299184799194336 Second

Average Loss: 0.06965356451742241

Accuracy: 97.50%



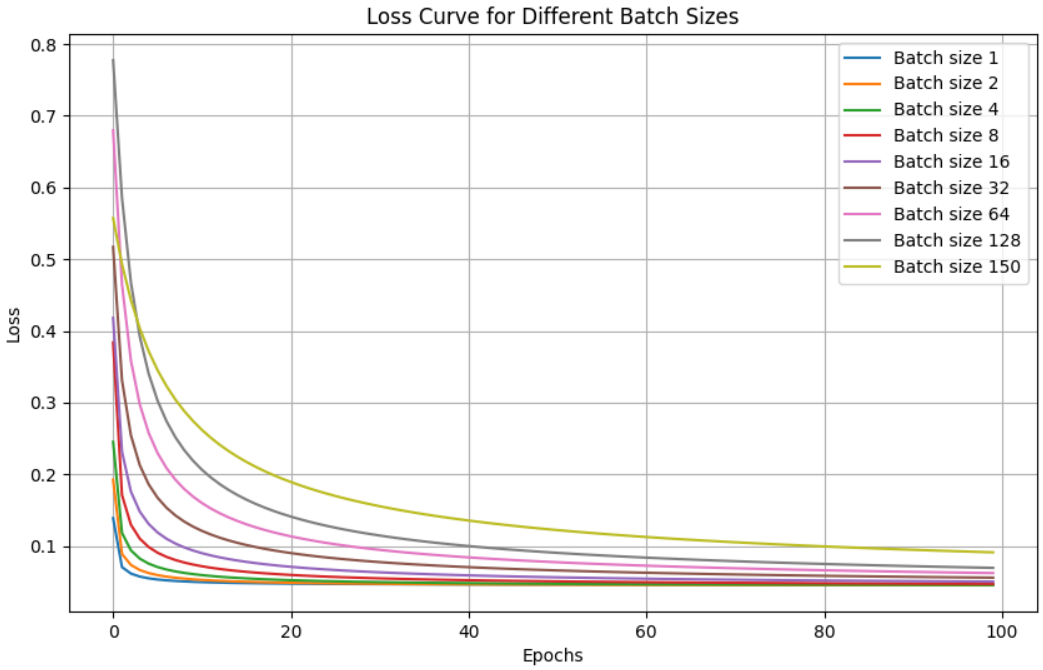
Batch Size: 150

Learning Time:
0.015277862548828125 Second

Testing Time:
0.007996797561645508 Second

Average Loss:
0.09128170862754151

Accuracy: 97.50%



1. **Learning Speed and Batch Size**:
    - It is evident that as the batch size increases, the learning process becomes faster. This is because larger batches allow the model to process more data in each iteration, reducing the number of updates required to complete an epoch.
    - With larger batch sizes, the model takes bigger steps in the optimization process, leading to faster convergence. This means that the model can reach a lower loss value in fewer epochs.
2. **Accuracy and Batch Size**:
    - The accuracy of the model tends to decrease as the batch size increases. This can be clearly seen from the decision boundary plots and accuracy metrics.
    - Smaller batch sizes provide a noisier but more frequent update to the model parameters, which can help the model escape local minima and find a better overall solution. Larger batches, while more stable, may miss these opportunities for fine-tuning, leading to slightly lower accuracy.

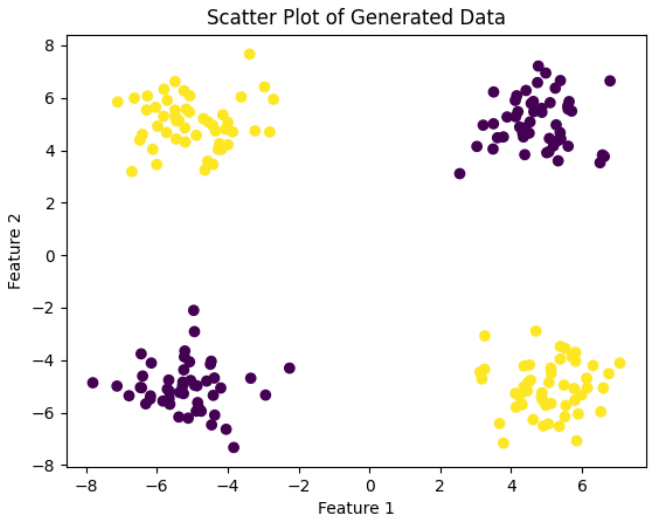
3. **Decision Boundary and Batch Size**:
    - The decision boundary becomes less precise with larger batch sizes. This is reflected in the plots where the area between points of gradual color becomes larger as batch size increases.
    - The decision boundary is a critical indicator of how well the model is classifying the data points. With larger batch sizes, the model might generalize more broadly, resulting in a less accurate boundary. The gradual color area between points indicates the regions where the model is less certain about the classification, and this uncertainty increases with larger batch sizes.

Part B

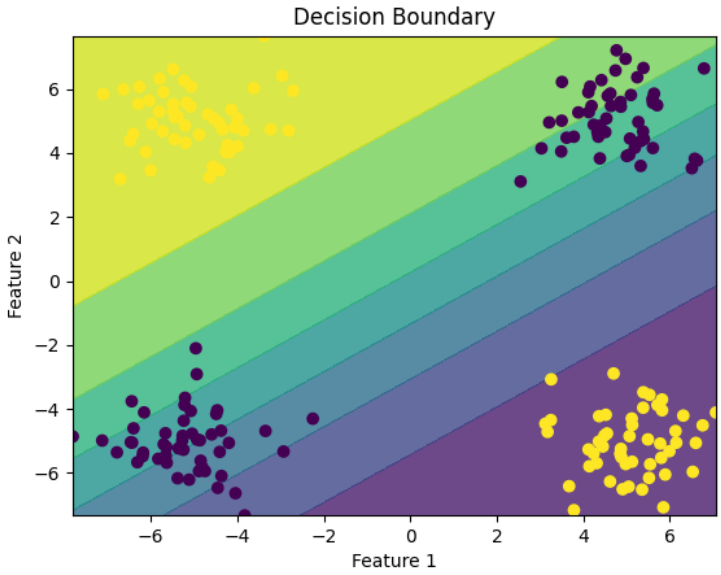
Dataset:

- Purple Points: These points are primarily located in the upper part of the plot, indicating one class of data with means of (5, 5) and (-5, -5).
- Yellow Points: These points are mainly found in the lower part of the plot, representing the other class with means of (5, -5) and (-5, 5).

This dataset will be used to train the model. We chose these means to ensure that the problem cannot be solved linearly, presenting a more challenging task for the model.

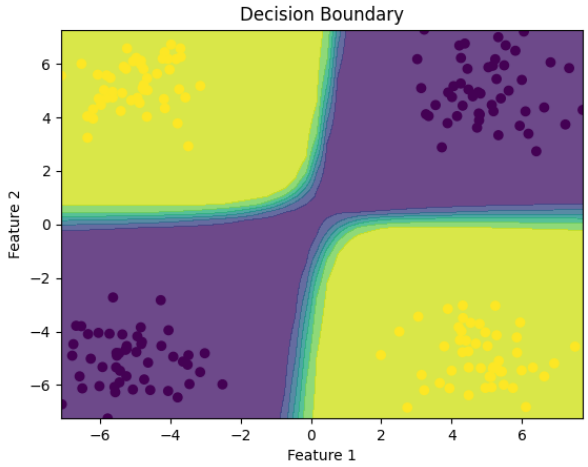


This section presents the results of applying logistic regression with the same parameters used in part A to the newly generated dataset.



After confirming that logistic regression cannot handle non-linear problems, we turned to using a Multi-Layer Perceptron (MLP). The MLP incorporates the sigmoid function as its activation function, effectively adding non-linearity to the model. This allows the MLP to learn and classify complex patterns that logistic regression cannot manage.

And here the result after use the same dataset

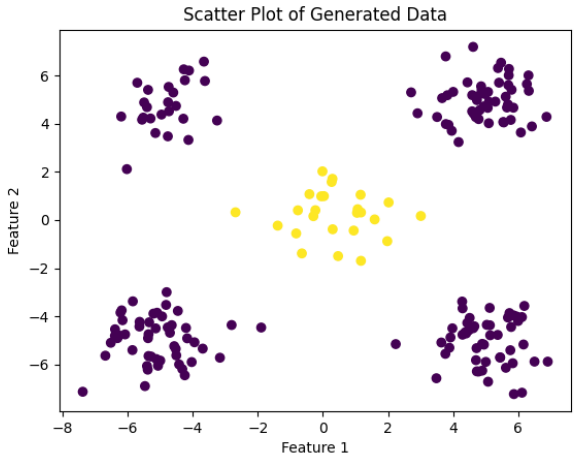


Epochs: 100, Learning Rate: 0.1,
Batch Size: 1.

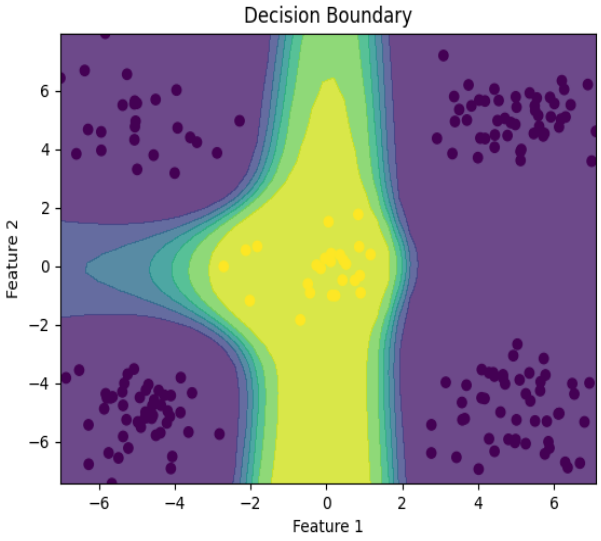
Learning Time: 1.58 Second,
Testing Time: 0.01 Second.

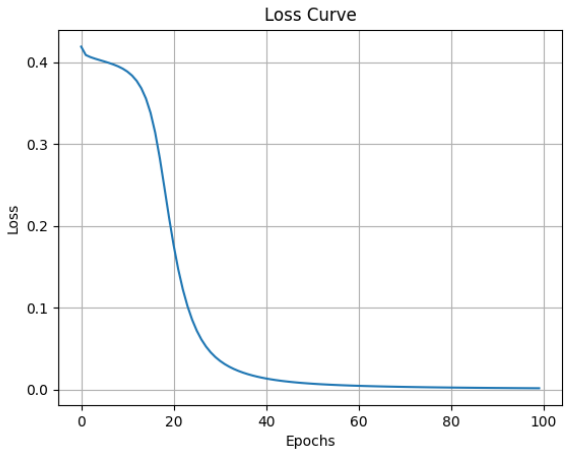
Average Loss: 0.001531581607997364,
Accuracy: 100.00%.

To improve the model's performance, we added a Relu activation function to the first layer. This modification led to a slight enhancement in performance, resulting in sharper decision boundaries as shown in the results. Encouraged by this improvement, we decided to experiment with a more complex dataset to further evaluate the model's capabilities.



Without Relu:



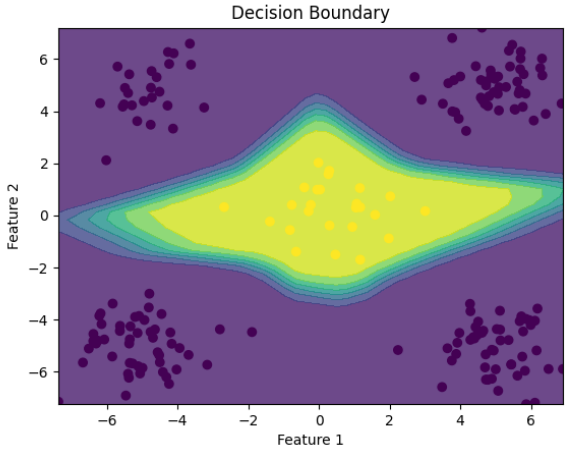


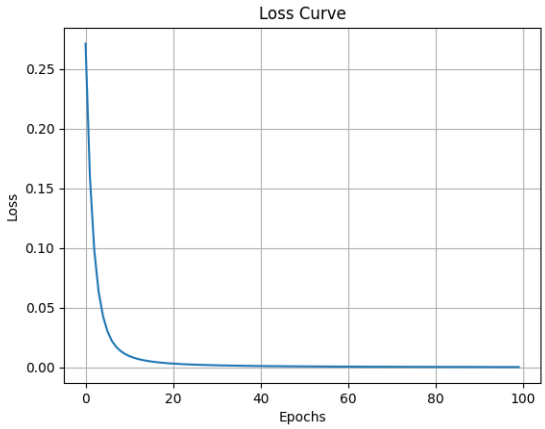
Epochs: 100, Learning Rate: 0.1, Batch Size: 1

Learning Time: 1.51 Second, Testing Time: 0.01 Second

Average Loss: 0.0012586080867374686, Accuracy: 100.00%

With ReLU after first layer:





Epochs: 100, Learning Rate: 0.1, Batch Size: 1

Learning Time: 3.71 Second, Testing Time: 0.01 Second

Average Loss 0.0004145173863278762, Accuracy: 100.00%

To further improve our approach, we automated the graph generation process, making it easier to implement and compare different models. This automation streamlines the workflow and enhances the efficiency of experimenting with various model configurations.

```
def topological_sort(node, visited=None, graph=None):
    if not visited:
        visited = set()
        graph = []
    if node in visited:
        return
    visited.add(node)
    for input_node in node.inputs:
        graph = topological_sort(input_node, visited, graph)
    graph.append(node)
    return graph
```

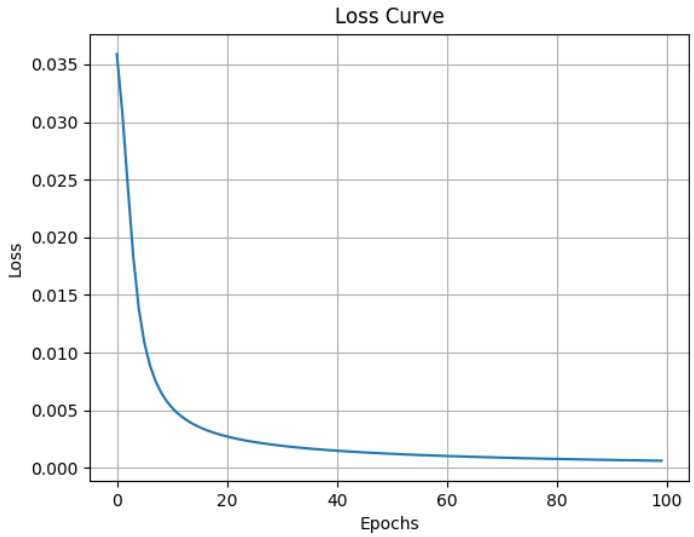
```
trainable = [i for i in graph if isinstance(i, Parameter)]
```

Now, we can easily edit the model to learn a simplified version of the MNIST dataset. This flexibility allows us to experiment with different architectures and optimizations to improve model performance. Here the results.

Epochs: 100, Learning Rate: 0.01, Batch Size: 64

Learning Time: 0.44 Second, Testing Time: 0.06 Second

Average Loss: 0.0006216799812872168, Accuracy: 98.22%



1. **Logistic Regression Limitation**:
   - Logistic regression cannot learn from non-linear datasets due to its inherently linear nature.
   - It is restricted to finding linear decision boundaries, making it unsuitable for datasets where classes are not linearly separable (e.g., XOR problem).

2. **Solution with MLP**:
   - Multi-Layer Perceptron (MLP) is employed to handle non-linear data.
   - By introducing non-linearity through activation functions like the sigmoid function, MLP can learn complex patterns and relationships within the data.
   - MLP effectively solves the non-linear problem with high performance, providing accurate classifications and well-defined decision boundaries.

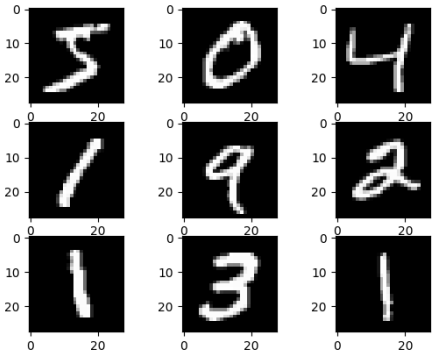
3. **Improvement with ReLU**:
    - Adding a ReLU (Rectified Linear Unit) activation function to the first layer of the MLP significantly improved model accuracy.
    - ReLU sharpens the precision, leading to more accurate and distinct decision boundaries.

4. **Purpose of Automation**:
    - Automating the graph generation process facilitates easy modifications to the model.
    - This automation allows for quick adjustments and comparisons of different models, enhancing our ability to optimize the model's performance efficiently.

5. **MNIST Dataset**:
    - The model is easily adjustable to learn a simplified version of the MNIST dataset.
    - This flexibility allows for experimentation with different architectures and optimizations, helping to further improve model performance on more complex datasets.

Part C

In this part, we will use the dataset created in Task One with the same model from Part B. However, before using the data, we need to flatten it. Below are some pictures of the data:

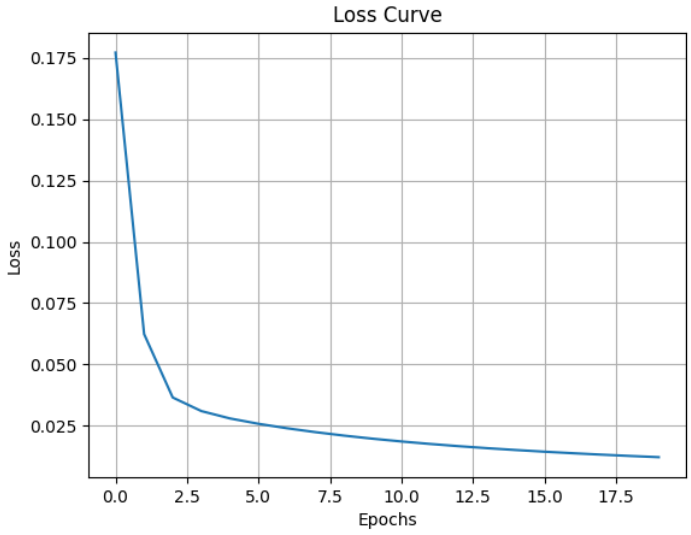


The result:

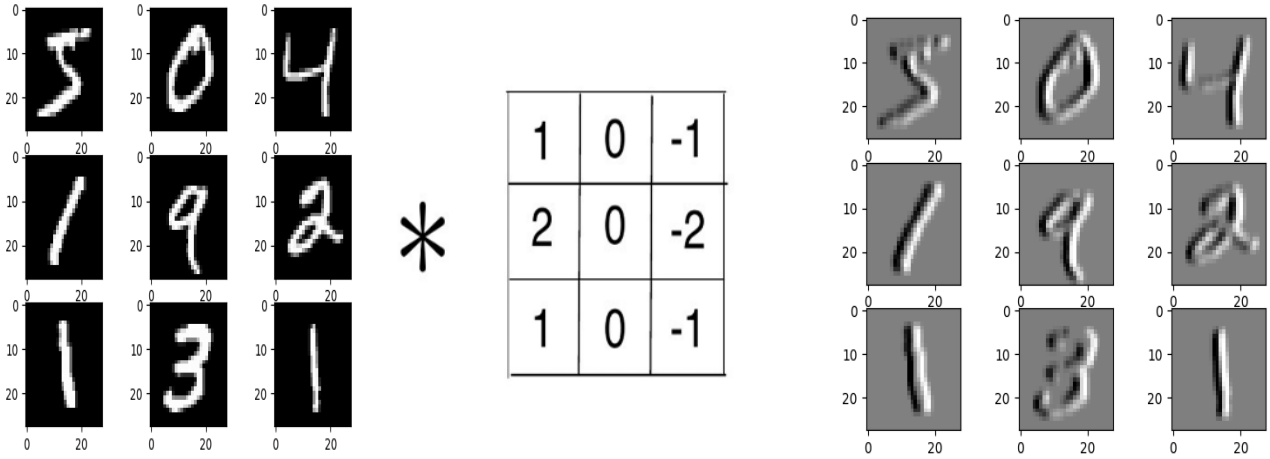
Epochs: 20, Learning Rate: 0.5, Batch Size: 64

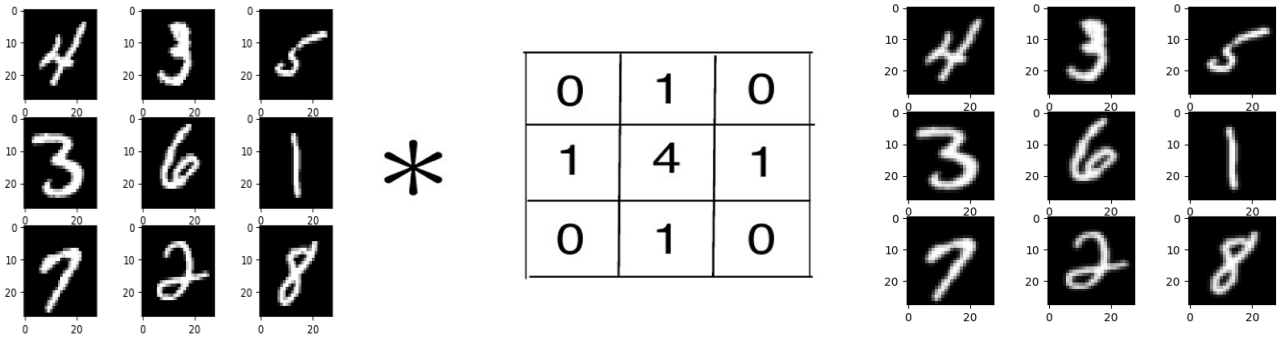
Learning Time: 1.69 Minute, Testing Time: 0.57 Second

Average Loss: 0.002044941965803771, Accuracy: 96.17%



Now, we have built the max-pooling and convolution functions to use in the next task. Below, you can see the effects of the convolution on our dataset:





Now, we will construct the Convolutional Neural Network (CNN). Before doing so, we need to build a Flatten class to flatten the data before it goes to the linear layers. This preprocessing step is crucial for ensuring the data is in the correct format for the fully connected layers of the CNN.

Epoch 1, Loss: 0.007952755527399725

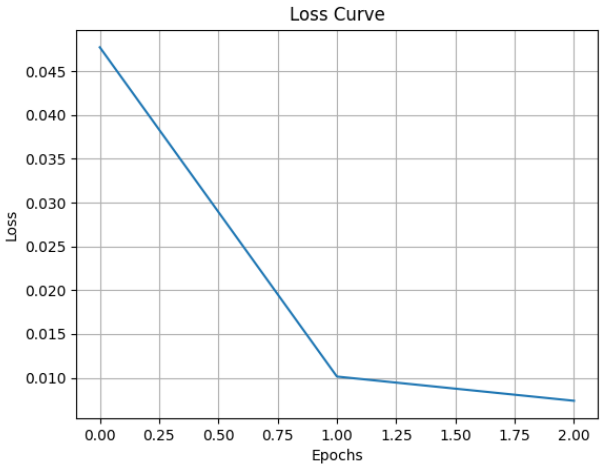
Epoch 2, Loss: 0.001691926333229384

Epoch 3, Loss: 0.001231438534453772

Epochs: 3, Learning Rate:0.1, Batch Size: 64

Learning Time: 9.15 Minute, Testing Time: 2.69 Minute

Average Loss: 0.001231438534453772, Accuracy: 98.05%



1. **Improvement with Full MNIST Dataset**:
- We expanded our work from the previous sections by utilizing the full MNIST dataset. This provided a more comprehensive and challenging dataset for evaluating the performance of different models.

2. **Building Max-Pooling and Convolutional Layers**:
- We constructed max-pooling and convolutional functions to enhance feature extraction capabilities.
- The convolutional layers were particularly effective in highlighting important features within the data, as demonstrated by the visual results.

3. **Constructing and Running the CNN**:
- We built a CNN and trained it on the same dataset used in previous sections.
- The CNN showed remarkable performance improvements, achieving significantly lower loss values compared to the Multi-Layer Perceptron (MLP).
- Specifically, the CNN reached a lower loss in just two epochs, outperforming the MLP, which took 20 epochs to achieve similar results.

4. **Performance and Efficiency**:
- The CNN not only improved performance metrics but also used fewer weights, demonstrating efficient learning of the data.
- However, it is important to note that the CNN required more computational time compared to the MLP. This trade-off between performance and computational time is a critical consideration when selecting models for specific tasks.