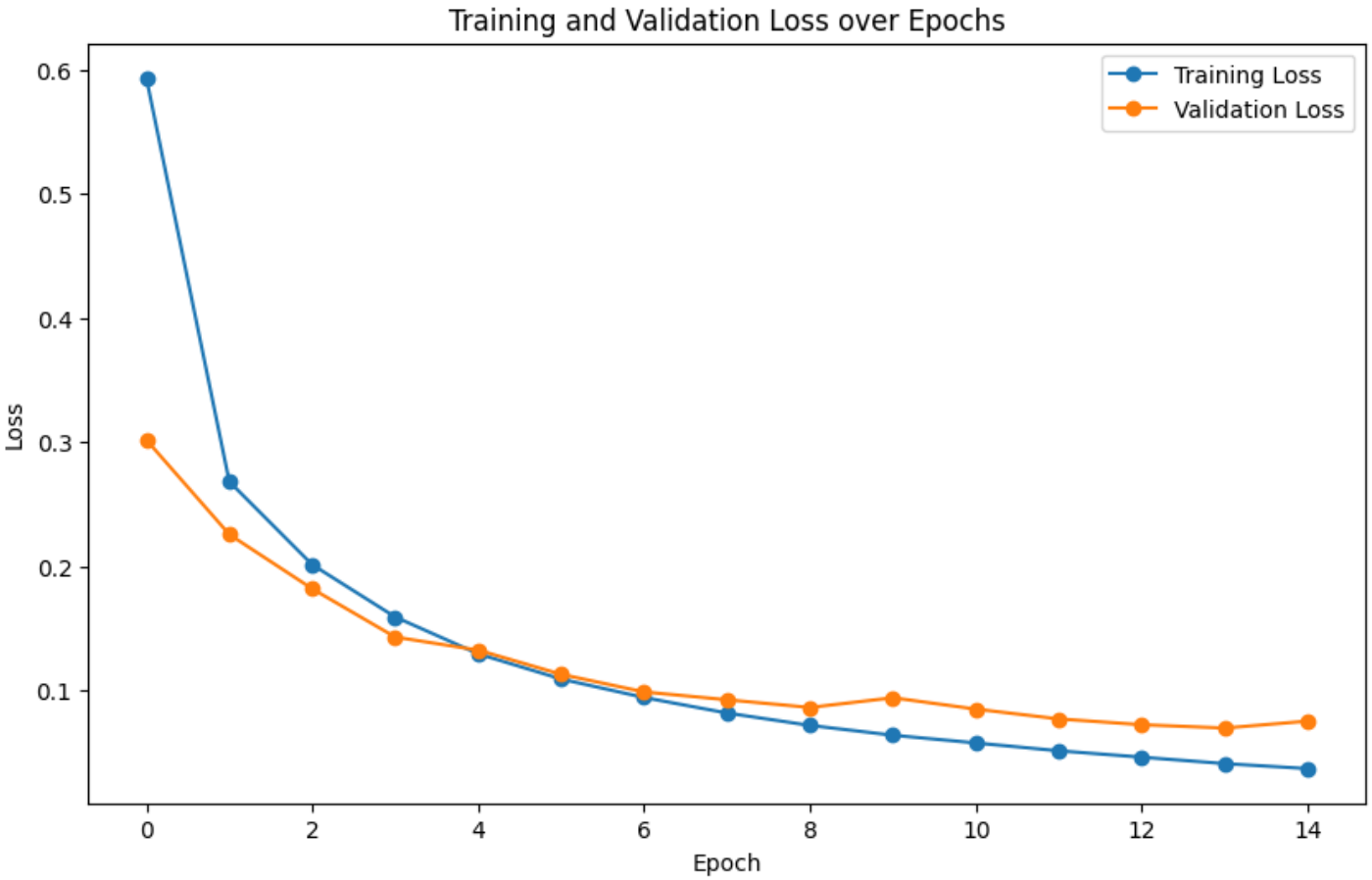
# Homework 1 CSCE 790: Neural Networks and Their Applications

## Problem 1

### c

To assess whether the neural network model is trained properly, we can plot the training and validation loss over epochs. A properly trained model should show a decreasing training loss and a stable or slightly increasing validation loss. This indicates that the model is learning without overfitting.

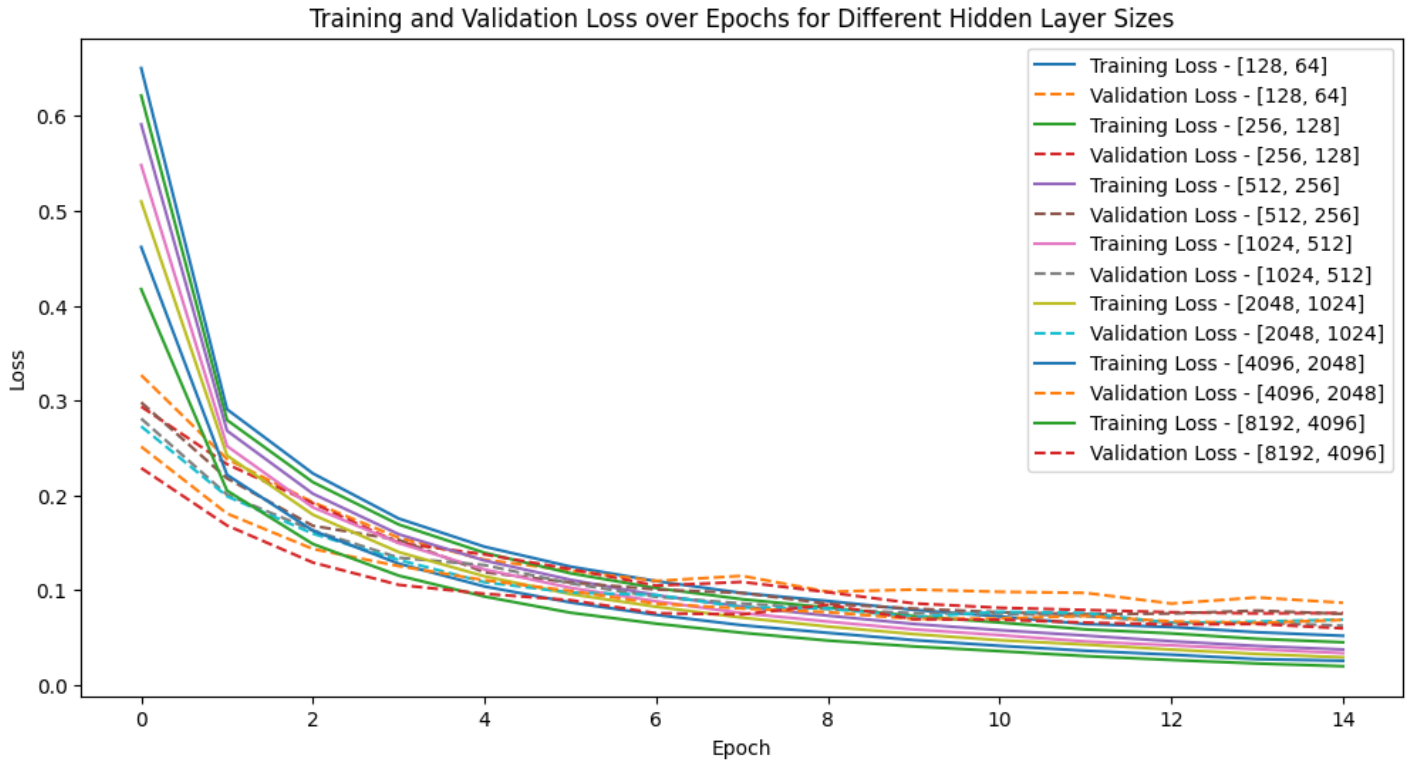



### d

The codes with comments can be found on handwritten_digit_recognition.ipynb

### e

Increasing the number of neurons in the hidden layers can have different effects depending on the complexity of the dataset and the task at hand. In some cases, it might improve the performance by allowing the network to learn more complex representations. However, it could also lead to overfitting, especially when the dataset is small, as a larger model has more parameters and, hence, more capacity to memorize the training data. In this case, I observerd no siginificant improvement. The figure below shows my findings.



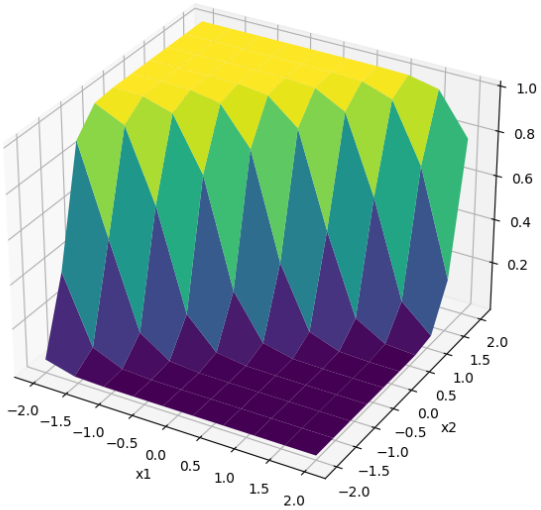

```
Final Accuracy for [128, 64] hidden neurons: 0.9731
Final Accuracy for [256, 128] hidden neurons: 0.9769
Final Accuracy for [512, 256] hidden neurons: 0.9755
Final Accuracy for [1024, 512] hidden neurons: 0.9797
Final Accuracy for [2048, 1024] hidden neurons: 0.9776
Final Accuracy for [4096, 2048] hidden neurons: 0.9782
Final Accuracy for [8192, 4096] hidden neurons: 0.981
```

# Problem 2

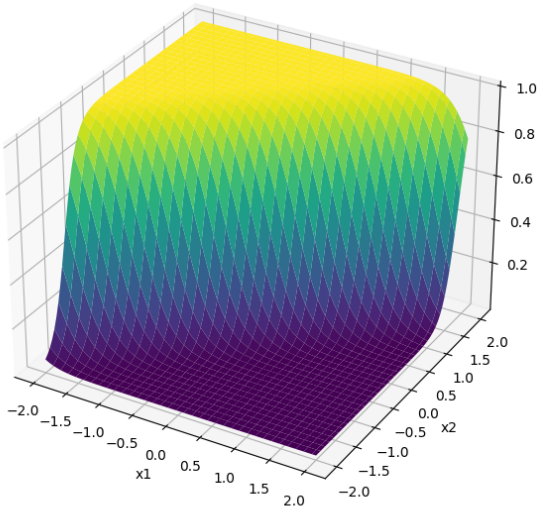
The code for this problem can be found on perceptron.ipnyb

The results are shown in the figures below:

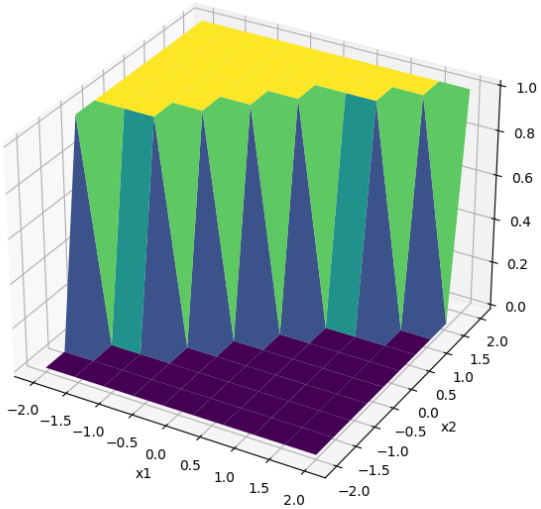
Sigmoid Activation Function - 100 Sample Points



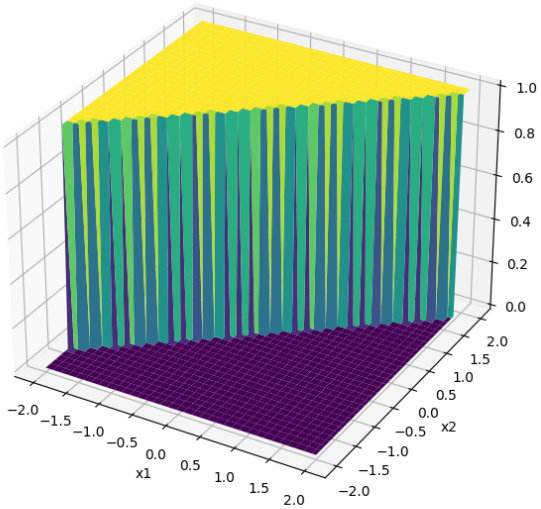

Sigmoid Activation Function - 5000 Sample Points



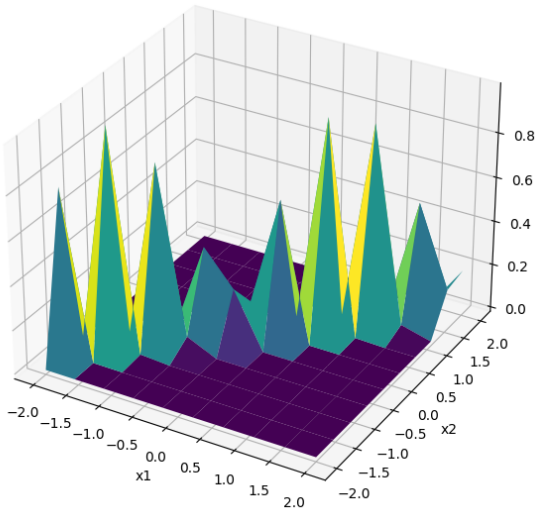

Hard Limit Activation Function - 100 Sample Points



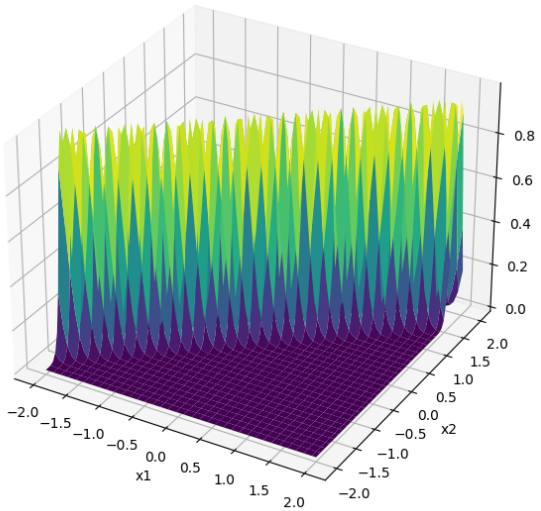

Hard Limit Activation Function - 5000 Sample Points



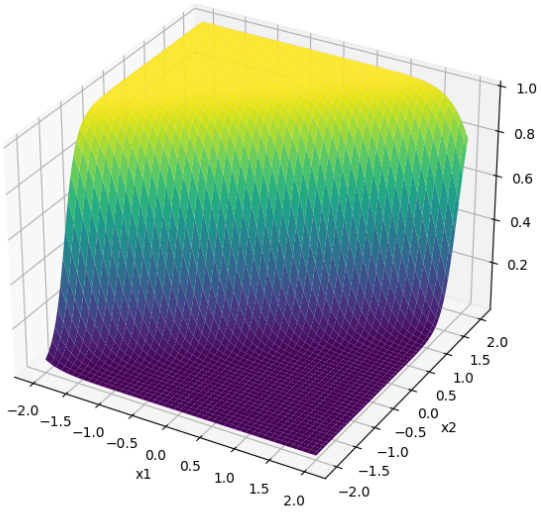

Radial Basis Function (RBF) - 100 Sample Points



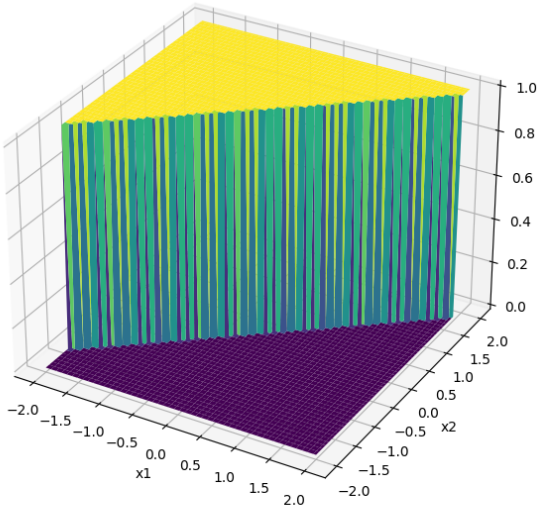

Radial Basis Function (RBF) - 5000 Sample Points



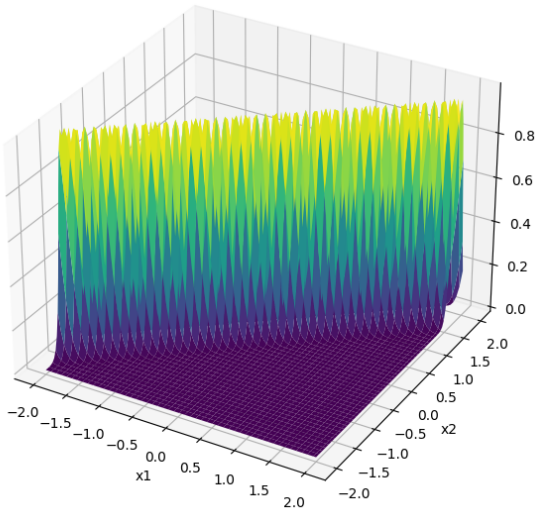
Sigmoid Activation Function - 10000 Sample Points




Hard Limit Activation Function - 10000 Sample Points



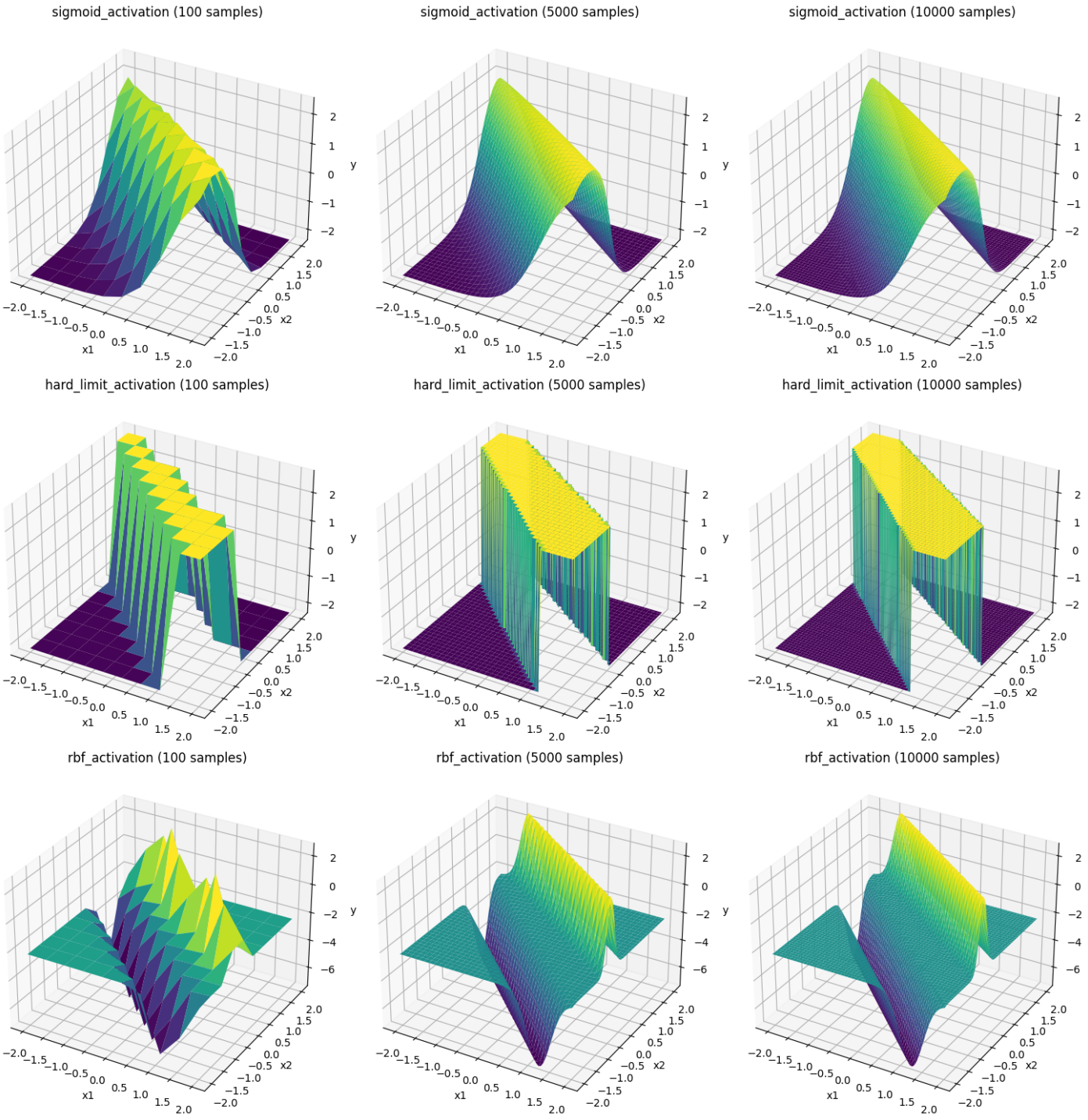

Radial Basis Function (RBF) - 10000 Sample Points




# Problem 3

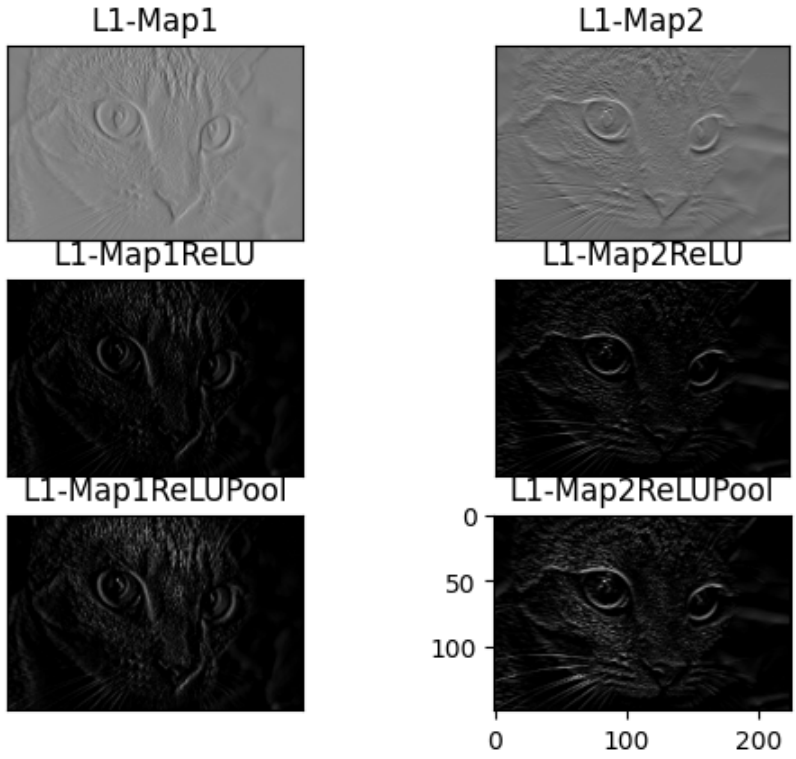
The code for this problem can be found on [two_layer_NN.ipnyb]

The results are shown in the figures below:

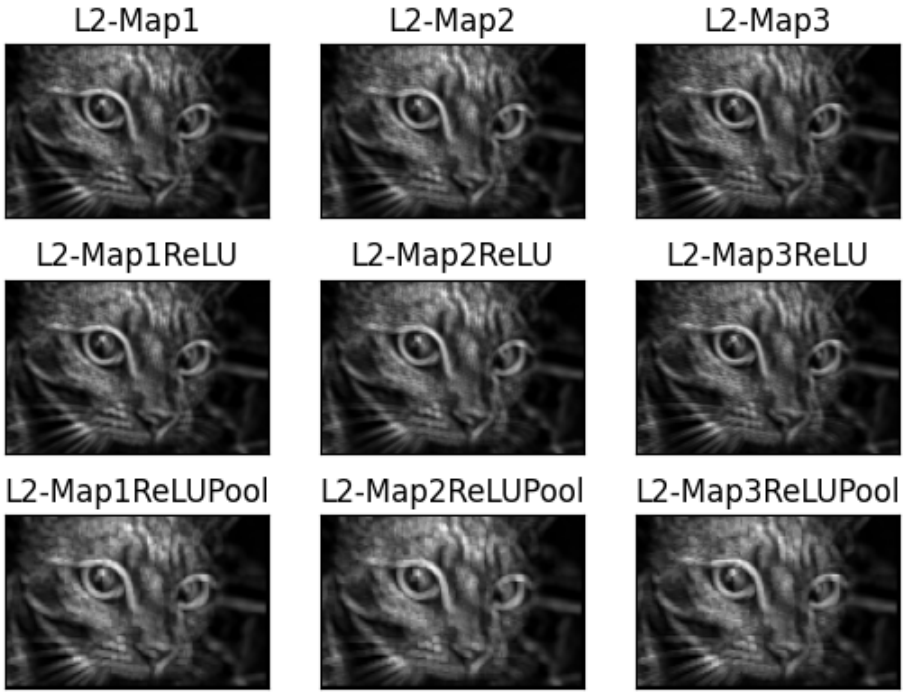



# Problem 4

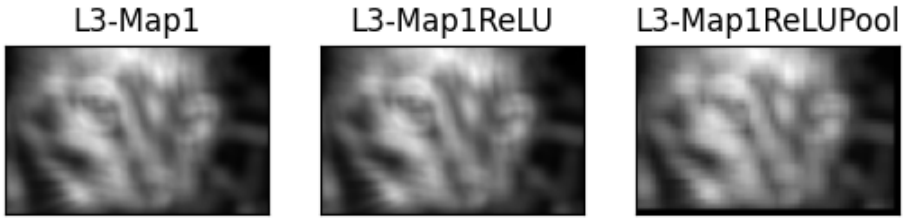
The results:




Layer 1 results




Layer 2 results




Layer 3 results

The codes can be found on [cnn_numpy_scratch.ipnyb]

This Python code is implementing a basic form of a Convolutional Neural Network (CNN) from scratch using NumPy.

## 1. Import Libraries

```
import numpy
import sys
import skimage.data
import matplotlib.pyplot
```

These lines import the necessary libraries: NumPy for numerical operations, sys for system-specific parameters and functions, skimage for accessing sample images, and matplotlib for plotting.

## 2. Convolution Function: `conv_`

```
def conv_(img, conv_filter):
```

This function performs the convolution operation on an input image `img` with a given filter `conv_filter`. It calculates the feature map by sliding the filter over the image and performing element-wise multiplication followed by summation.

## 3. Convolution Layer Function: `conv`

```
def conv(img, conv_filter):
```

This function handles the application of multiple filters and channels. It checks the dimensions of the image and the filters, then applies the convolution to each filter and sums the results if there are multiple channels. It returns the feature maps.

## 4. Pooling Function: `pooling`

```
def pooling(feature_map, size=2, stride=2):
```

This function applies max pooling to the input feature map. It slides a window of a given `size` over the feature map with a certain `stride` and takes the maximum value within the window.

## 5. ReLU Activation Function: `relu`

```
def relu(feature_map):
```

This function applies the Rectified Linear Unit (ReLU) activation function to the input feature map. It replaces all negative pixel values in the feature map with zero.

## 6. Load and Preprocess the Image

```
img = skimage.data.chelsea()
img = skimage.color.rgb2gray(img)
```

This code loads a sample image of a Chelsea apartment from the `skimage` library and converts it to grayscale.

## 7. First Convolutional Layer

```
l1_filter = numpy.zeros((2,3,3))
...
l1_feature_map = conv(img, l1_filter)
l1_feature_map_relu = relu(l1_feature_map)
l1_feature_map_relu_pool = pooling(l1_feature_map_relu, 2, 2)
```

Here, the first convolutional layer is defined with two filters of size 3x3. The layer is then applied to the image, followed by ReLU activation and pooling.

### 8. Second Convolutional Layer

```
l2_filter = numpy.random.rand(3, 5, 5, l1_feature_map_relu_pool.shape[-1])
...
l2_feature_map = conv(l1_feature_map_relu_pool, l2_filter)
l2_feature_map_relu = relu(l2_feature_map)
l2_feature_map_relu_pool = pooling(l2_feature_map_relu, 2, 2)
```

The second convolutional layer is defined with three randomly initialized filters of size 5x5. The layer is applied to the output of the previous layer, followed by ReLU activation and pooling.

### 9. Third Convolutional Layer

```
l3_filter = numpy.random.rand(1, 7, 7, l2_feature_map_relu_pool.shape[-1])
...
l3_feature_map = conv(l2_feature_map_relu_pool, l3_filter)
l3_feature_map_relu = relu(l3_feature_map)
l3_feature_map_relu_pool = pooling(l3_feature_map_relu, 2, 2)
```

The third convolutional layer is defined with one randomly initialized filter of size 7x7. The layer is applied to the output of the second layer, followed by ReLU activation and pooling.

### 10. Results

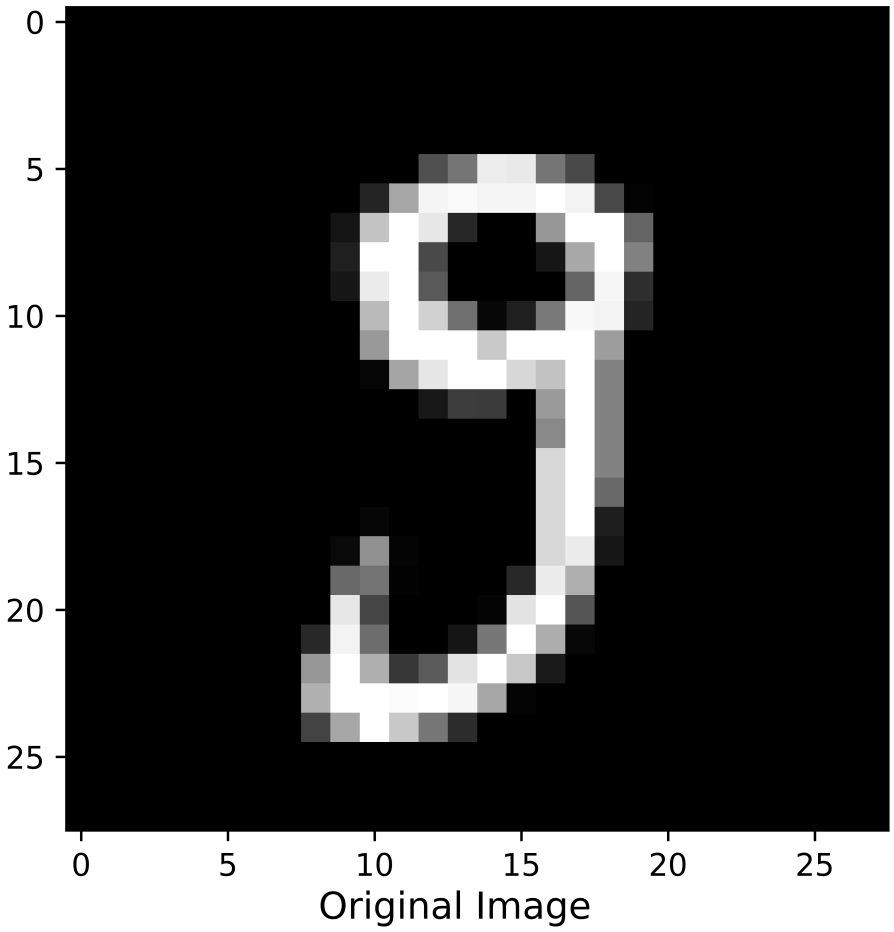
The resulting feature maps after each layer are stored in the variables `l1_feature_map_relu_pool`, `l2_feature_map_relu_pool`, and `l3_feature_map_relu_pool`.

### Notes:

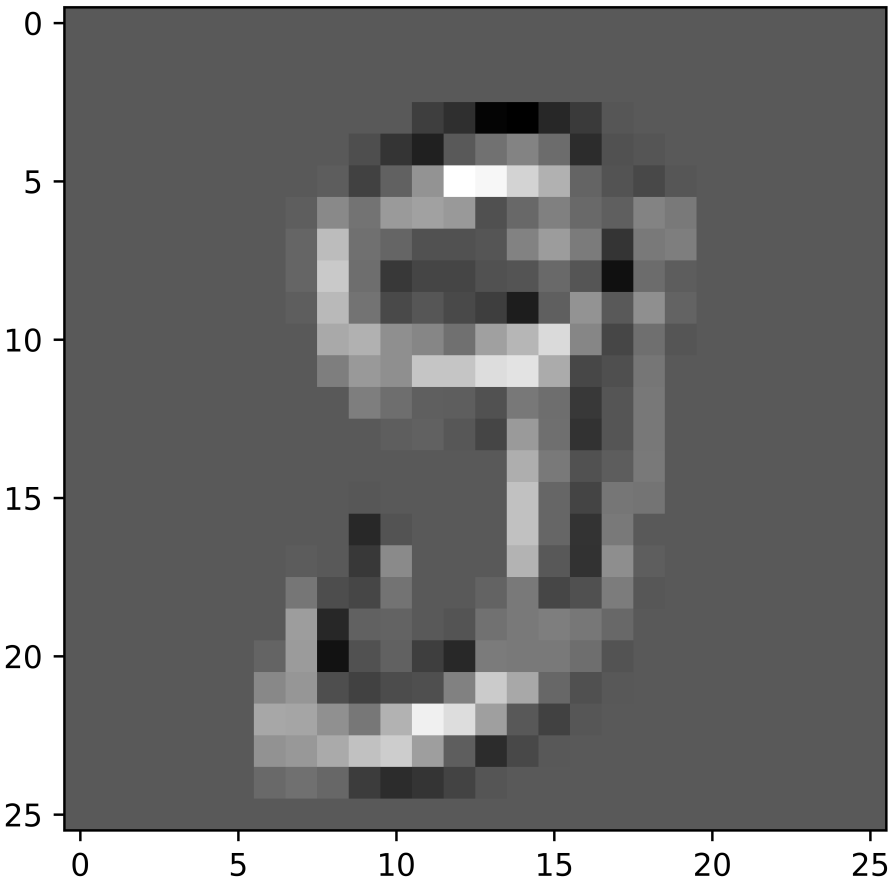
- The code does not involve any learning; it's a forward pass of an image through a CNN with predefined or randomly initialized filters.

# Problem 5

The results:




Original input image




The image after applying convolution

The codes can be found on cnn_numpy_scratch.ipnyb

## 1. ConvolutionLayer Class

- **Initialization (** `__init__` **)**:
  - Initializes the number of kernels and their sizes.
  - Generates random kernel weights, normalized by the square of the kernel size.
- **Patches Generator (** `patches_generator` **)**:
  - Iterates over the input image, generating patches and their coordinates for the convolution operation.
- **Forward Propagation (** `forward_prop` **)**:
  - Initializes an output volume to store the convolution results.
  - Iterates over the patches, performs the convolution operation for each patch with each kernel, and stores the result in the output volume.
- **Backward Propagation (** `back_prop` **)**:
  - Computes the gradient of the loss function with respect to the kernel weights.
  - Updates the kernel weights using the computed gradient and a learning rate ( `alpha` ).

## 2. MaxPoolingLayer Class

- **Initialization (** `__init__` **)**:
  - Initializes the pooling kernel size.

- **Patches Generator ( `patches_generator` )**:
  - Iterates over the input volume, generating patches and their coordinates for the pooling operation.
- **Forward Propagation ( `forward_prop` )**:
  - Initializes an output volume to store the pooling results.
  - Iterates over the patches, performs max pooling for each patch, and stores the result in the output volume.
- **Backward Propagation ( `back_prop` )**:
  - Computes the gradient of the loss function with respect to the input of the pooling layer.
  - There are no weights to update in this layer, but the computed gradient is used to update the weights of the preceding convolutional layer.

### 3. SoftmaxLayer Class

- **Initialization ( `__init__` )**:
  - Initializes the weights and biases with dimensions corresponding to the input and output units.
- **Forward Propagation ( `forward_prop` )**:
  - Flattens the input volume.
  - Performs matrix multiplication with the weights, adds the biases, and applies the softmax activation function.
- **Backward Propagation ( `back_prop` )**:
  - Computes the gradient of the loss function with respect to the weights, biases, and input.
  - Updates the weights and biases using the computed gradients and a learning rate ( `alpha` ).

### 4. CNN Forward and Backward Propagation Functions

- **`CNN_forward`**: Takes an image and label, performs forward propagation through the layers, computes the loss and accuracy, and returns the output, loss, and accuracy.
- **`CNN_backprop`**: Takes the initial gradient and performs backward propagation through the layers, updating the weights.
- **`CNN_training`**: Calls `CNN_forward` and `CNN_backprop`, returns the loss and accuracy.

### 5. Comparison with problem 4

### Defining CNN Layers:

- In the first algorithm, the convolutional layers were defined using functions and arrays, with each layer applying convolution, ReLU activation, and pooling sequentially. The filters for the convolutional layers were either predefined or randomly initialized.
- In the second algorithm, an object-oriented approach is used to define each layer as a class with its methods for forward and backward propagation. The convolutional layer is defined with random kernel weights, the max-pooling layer with a specified pooling size, and the softmax layer with initialized weights and biases based on input and output units.

### Pooling:

- Both algorithms use max-pooling, but the implementation details differ. In the first algorithm, the pooling function is separate and applies pooling to the feature maps. In the second algorithm, the max-pooling layer is a class with its forward and backward propagation methods.

### Softmax Layer:

- The first algorithm does not include a softmax layer, as it does not perform classification and is not trained on a dataset.
- The second algorithm includes a softmax layer as the final layer of the CNN for classification purposes. The softmax layer computes the probability distribution of the classes and is part of the training process.

### Backpropagation and Weight Updates:

- The first algorithm does not include backpropagation or weight updates since it only demonstrates a forward pass.
- The second algorithm includes methods for backward propagation in each layer class, computes gradients, and updates the weights of the convolutional and softmax layers during training.

# Problem 6

The results:

**The MLP model**

```
Train accuracy: 97.14%
Val accuracy:   54.60%
Test accuracy:  60.60%
```

**The GNN model**

```
Train accuracy: 100.00%
Val accuracy:   78.60%
Test accuracy:  82.40%
```

**The GraphConv model**

```
Train performance: 93.28%
Test performance:  92.11%
```

The codes can be found on t7_gnn.ipnyb

# Graph Neural Networks Tutorial Overview

# 1. Graph Representation

## Concept:

The tutorial initiates by laying down the basics of graph representation. It explains that a graph, denoted as G, is made up of nodes (V) and edges (E). The relationships or connections between the nodes are represented using an adjacency matrix A. In this matrix, the element in the i-th row and j-th column is 1 if there is a connection between nodes i and j, and 0 otherwise.

## Code:

In the code section, the notebook demonstrates this concept by defining an example graph with nodes and edges and subsequently forming its adjacency matrix. Python arrays or lists are used to create the adjacency matrix manually, where rows and columns symbolize nodes, and the binary elements indicate the presence or absence of edges between the corresponding nodes.

# 2. Graph Convolutional Networks (GCNs)

## Concept:

GCNs operate by enabling nodes in a graph to exchange information with their neighboring nodes. In a GCN layer, the features of each node at the next layer are calculated by applying a transformation to the aggregated features of the neighboring nodes and the node itself. The transformation includes a weight matrix, and the results are passed through an activation function. The process includes adding self-connections to nodes and normalizing the features by the number of neighbors.

### Code:

The implementation in PyTorch involves initializing the node features and the adjacency matrix, adding self-connections, normalizing by the number of neighbors, and applying the GCN layer to compute the next layer's features. The weight matrix is typically initialized as an identity matrix for simplicity.

## 3. Graph Attention Networks (GATs)

### Concept:

GATs utilize attention mechanisms to dynamically assign different importances to the neighboring nodes when aggregating their features. The attention mechanism computes attention weights based on the features of the nodes and applies a non-linear transformation (LeakyReLU) before normalization (softmax).

### Code:

This part demonstrates the detailed construction of a GAT layer in PyTorch, specifying the computation of attention weights and their utilization for aggregating features. The implementation showcases the use of multiple attention heads for enhanced model expressiveness and diversity.

## 4. PyTorch Geometric

### Concept:

PyTorch Geometric is introduced as a specialized library for GNNs, offering a variety of graph layers, efficient handling of graph-structured data, and a collection of datasets for experimentation.

### Code:

This part guides users through the installation of PyTorch Geometric, loading graph datasets, representing graphs using `Data` objects, and accessing various graph attributes such as edge indices and node features.

## 5. Node-level Tasks and Experiments

### Concept:

The tutorial explores node-level tasks, specifically using the Cora dataset for semi-supervised node classification. Each publication in the dataset is represented by a bag-of-words vector and categorized into one of seven classes.

### Code:

The code segments involve constructing a GNN model using defined graph layers, incorporating ReLU activation, and applying dropout for regularization. An MLP baseline model is also implemented for comparison. The models are integrated into a PyTorch Lightning module, with code provided for training, validation, testing, and performance evaluation on the Cora dataset.

# Problem 7

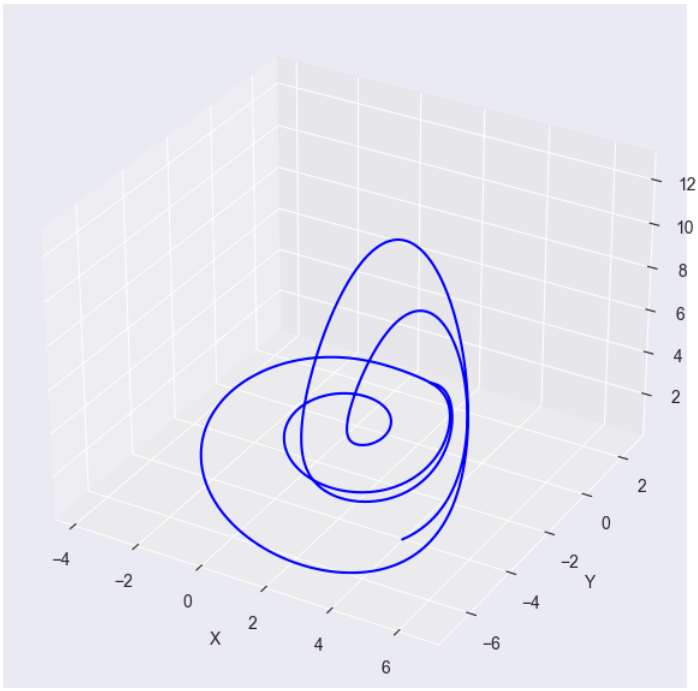
For this part, I got help from the following links:

1. https://medium.com/codex/python-and-physics-lorenz-and-rossler-systems-65735791f5a2
2. https://github.com/hukenovs/chaospy
3. https://thebrickinthesky.wordpress.com/2013/02/23/maths-with-python-2-rossler-system/

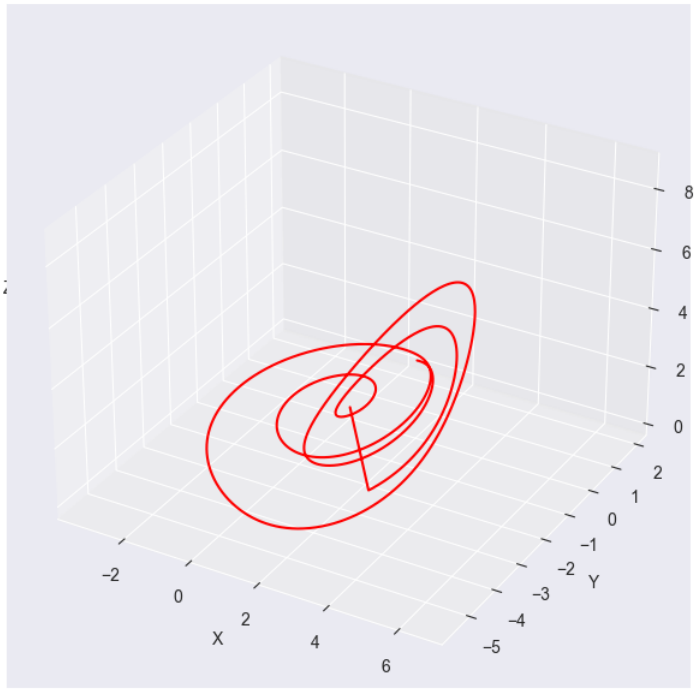
The codes can be found on reservoir_observers.ipnyb

The results:

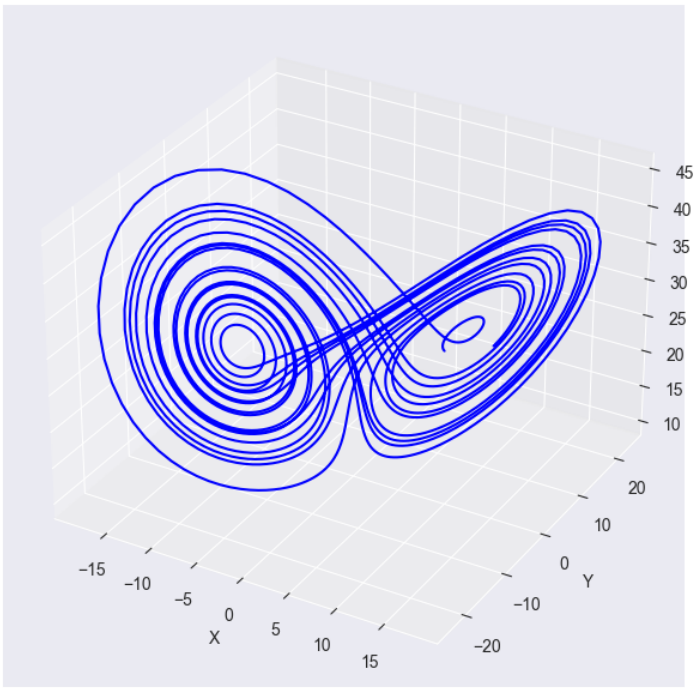
Actual Rossler System



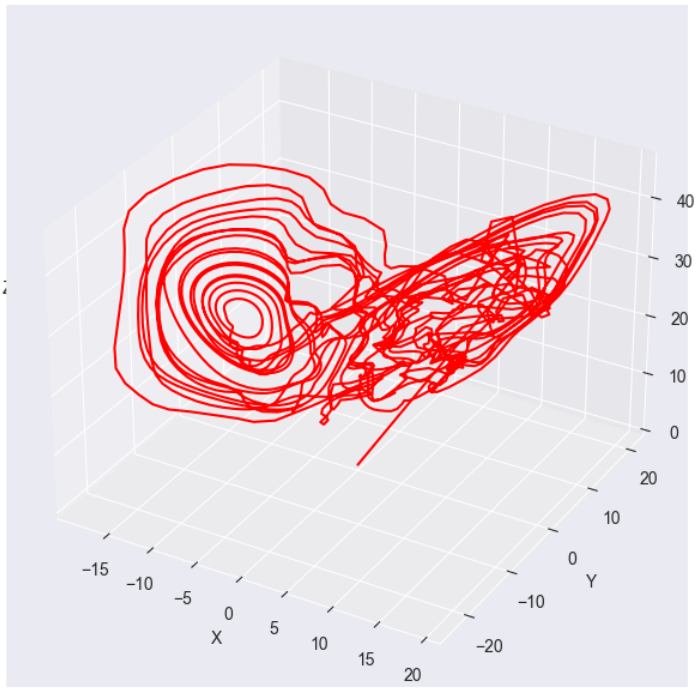

Predicted Rossler System

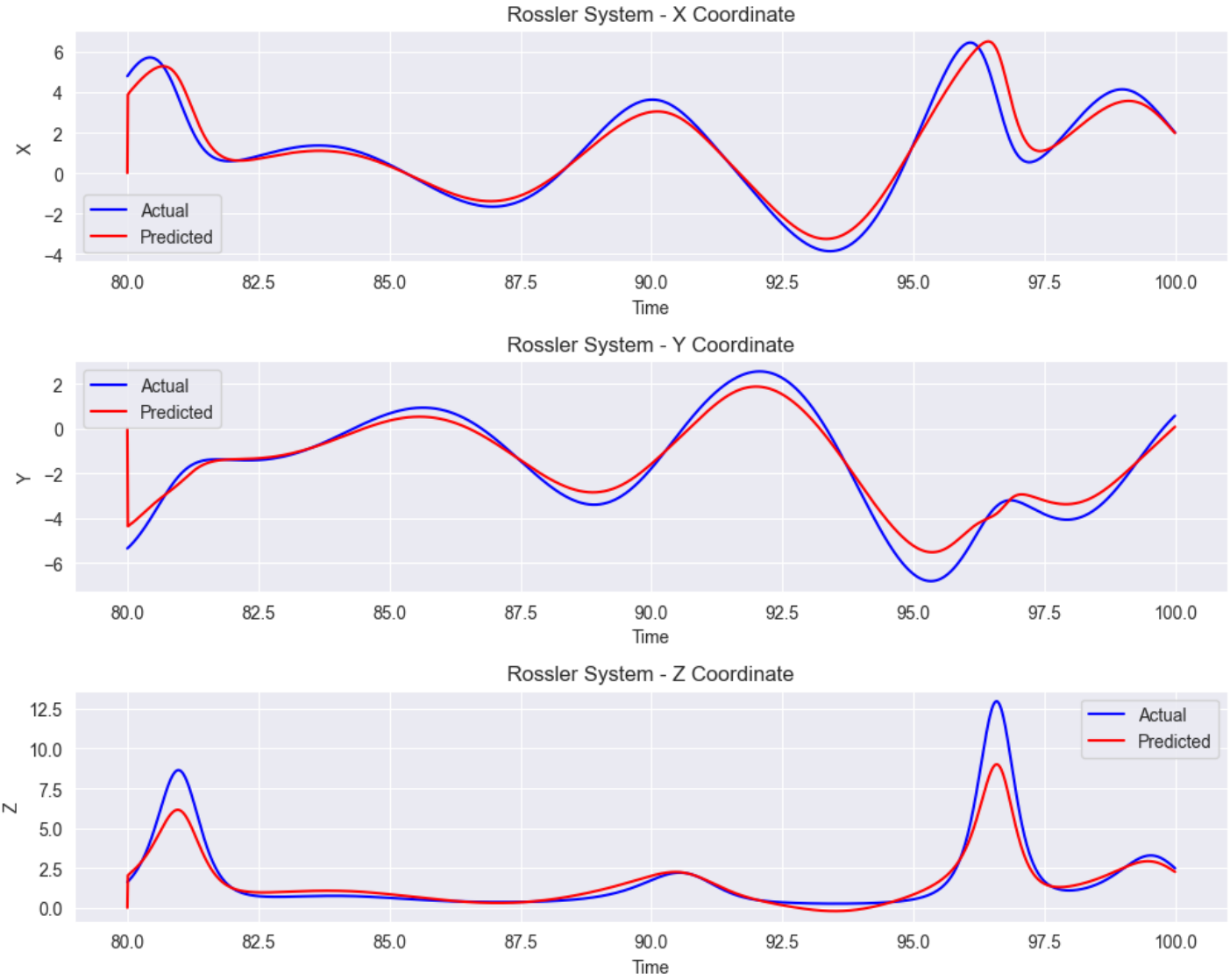


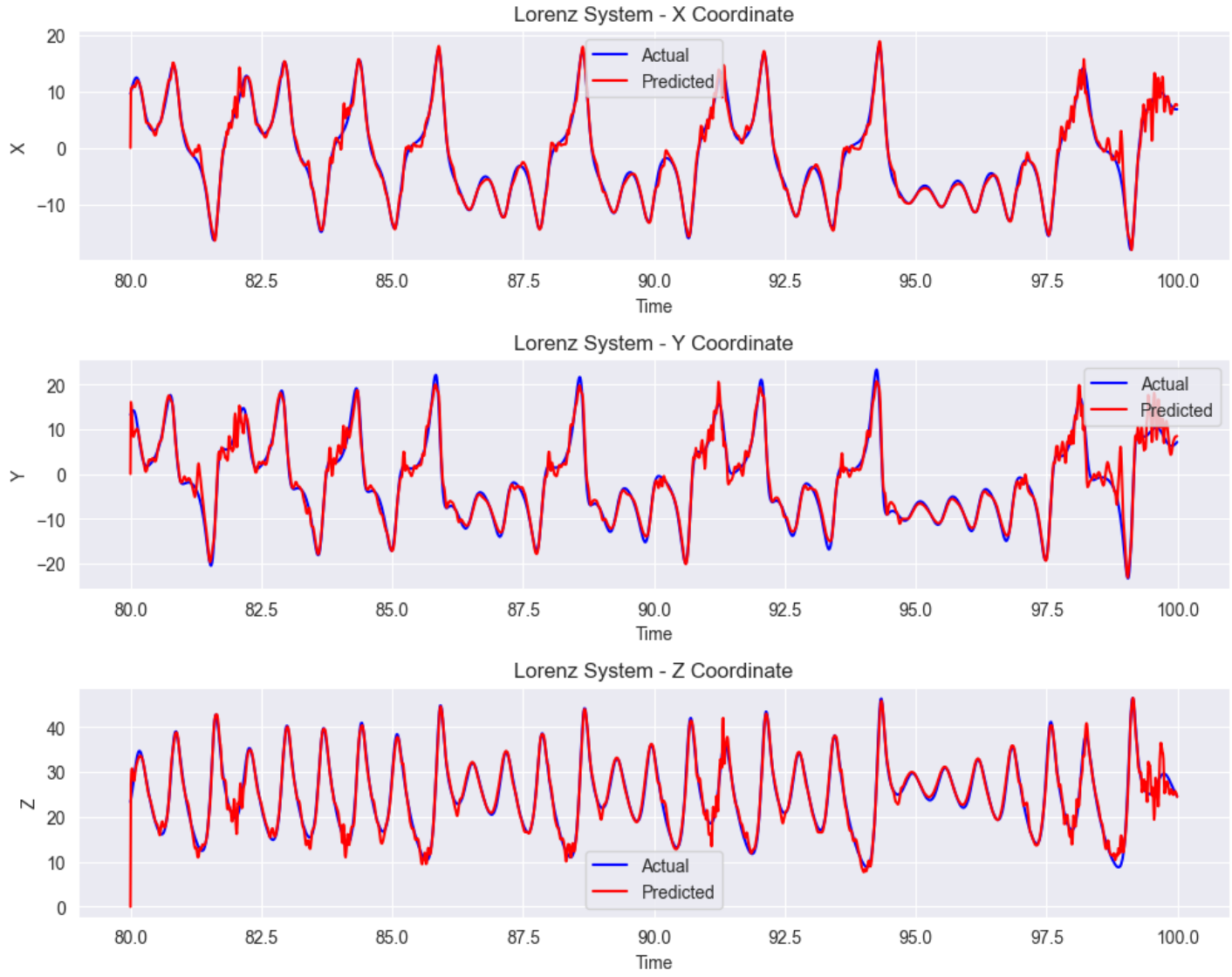

Actual Lorenz System




Predicted Lorenz System




Rossler System - X Coordinate
X
Time
Actual
Predicted
Rossler System - Y Coordinate
Y
Time
Actual
Predicted
Rossler System - Z Coordinate
Z
Time
Actual
Predicted




# 1. Simulation of Dynamical Systems

## 1.1 Rossler System

- **Equations**:

$$[\frac{dx}{dt} = -y - z], [\frac{dy}{dt} = x + 0.5y], [\frac{dz}{dt} = 2.0 + z(x - 4.0)] \tag{1}$$

- **Method**: Used the first-order Euler approximation with a time step of `0.01` and simulated for a total time of `100` units.
- **Data Generation**: Generated a time series of 8000 samples for training and 2000 samples for testing.

## 1.2 Lorenz System

- **Equations**:

$$[\frac{dx}{dt} = 10(y - x)], [\frac{dy}{dt} = x(28 - z) - y], [\frac{dz}{dt} = xy - \frac{8}{3}z] \tag{2}$$

- **Method**: Similar to the Rossler system, used the Euler approximation for simulation.
- **Data Generation**: Produced training and testing datasets with the same number of samples as the Rossler system.

# 2. Implementation of Reservoir Computing

- **Model Structure**: The reservoir computing model consisted of three layers: an input layer, a reservoir layer with `100` nonlinear nodes, and a linear output layer.

- **Dynamics**: The reservoir dynamics were defined as

$$r(t+\delta t) = (1-a)r(t) + a\tanh(Ar(t) + W_{\text{in}}u(t)) \tag{3}$$

  where (a=0.5), (A) is the adjacency matrix, W is the input weight matrix, and (u(t)) is the input signal.
- **Training**: The model was trained using ridge regression with a regularization parameter of `1e-6`. The reservoir states were updated and stored, forming the feature matrix for regression, and the output weights were learned to map the reservoir states to the target outputs.

## 3. Evaluation and Results

- **Testing**: The trained models were tested on unseen data from the Rossler and Lorenz systems. The reservoir states were updated using the test input, and predictions were made using the learned output weights.
- **Performance Metrics**: The mean squared error (MSE) was calculated to quantify the difference between the predicted and actual states.
  - Rossler system: MSE ≈ 4.24
  - Lorenz system: MSE ≈ 1.89

## 4. Visualizations

- **3D Trajectory Plots**: The plots showed the actual and predicted trajectories in the phase space for both the Rossler and Lorenz systems, illustrating the ability of the models to capture the chaotic dynamics.
- **Time Series Plots**: The time series for each coordinate (X, Y, Z) were plotted for both systems, demonstrating the close alignment between the predicted and actual time series over time.