UE23CS352A: MACHINE LEARNING

Week 6: Artificial Neural Networks

PROJECT TITLE: Artificial Neural Networks

NAME: NAJMUS SEHER

SRN: PES2UG23CS359

Course: Machine Learning(UE23CS352A)

Date: 16-09-2025

1. Introduction

The purpose of this lab was to train a fully connected neural network to approximate a given polynomial function with noise.

The tasks included:

- Implementing **Xavier weight initialization**.
- Implementing **forward propagation** with ReLU activations.
- Implementing **backpropagation** using the chain rule.
- Training the network with **MSE loss** and **early stopping**.
- Evaluating the trained model on test data and analyzing performance.

2. Dataset Description:

```
Polynomial Type: CUBIC + INVERSE: y = 2.42x³ + 0.05x² + 3.92x + 9.72 +
115.3/x
Noise Level: ε ~ N(0, 2.29)
Architecture: Input(1) → Hidden(96) → Hidden(96) → Output(1)
Learning Rate: 0.003
Architecture Type: Large Balanced Architecture
```

Samples: 100,000 total

- **Training samples:** 80,000
- **Test samples:** 20,000

3.Methodology :

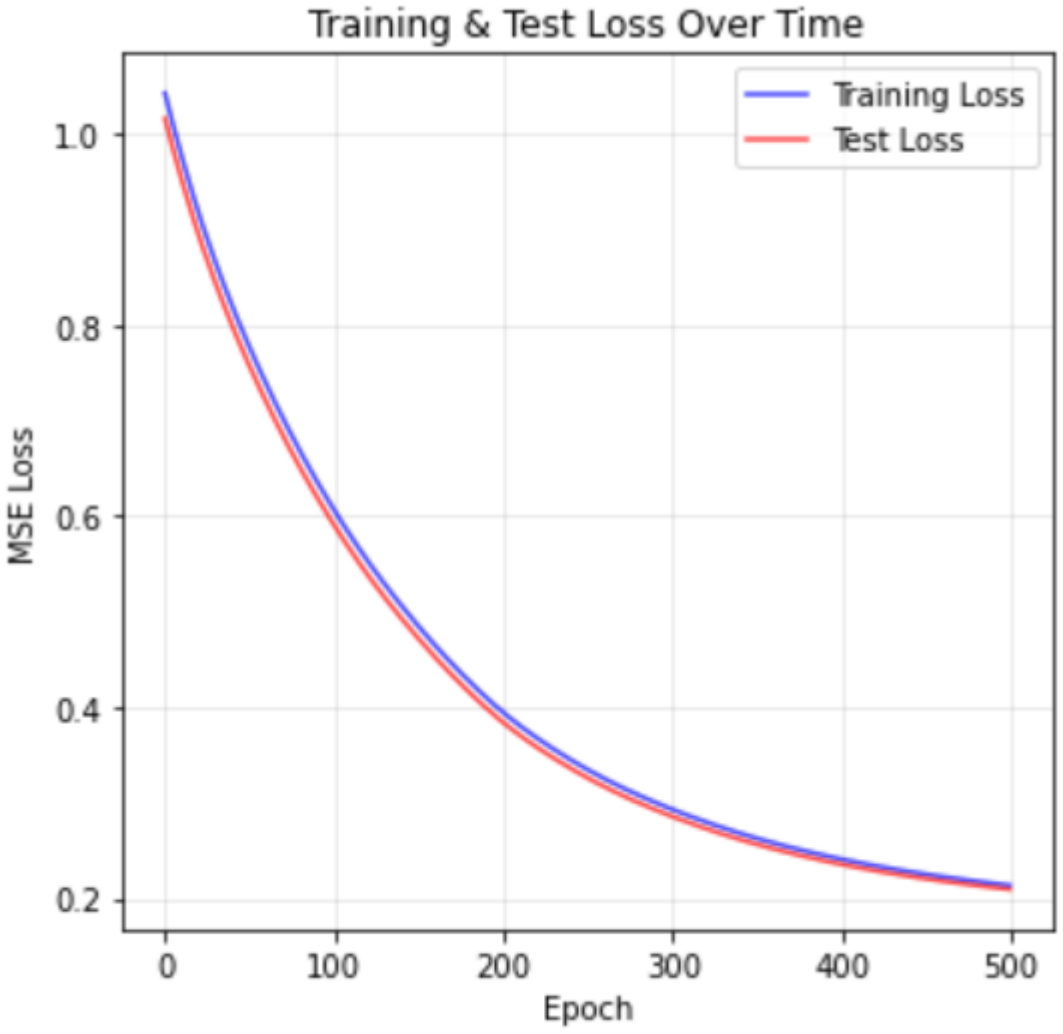
The process involved the following steps:

1. **Activation Functions**
    - I implemented the ReLU activation function for the hidden layers. ReLU outputs the input if it is positive and zero otherwise.
    - I also implemented its derivative, which is required during backpropagation to compute gradients.
    - For the output layer, I used a linear activation because the task is regression.
2. **Loss Function**
    - I used the Mean Squared Error (MSE) as the loss function.
    - This measures how far the predicted outputs are from the actual values and serves as the objective for optimization.
3. **Weight Initialization**
    - To avoid issues of exploding or vanishing gradients, I implemented Xavier initialization.
    - This method initializes weights using a normal distribution with variance that depends on the number of input and output connections.
    - Biases were initialized to zero.
4. **Forward Propagation**
    - Input data passes through the network layer by layer.
    - Each layer computes a linear transformation followed by a non-linear activation (ReLU for hidden layers, linear for output).
    - This produces the predicted output for a batch of inputs.
5. **Backward Propagation**

- Using the chain rule, I calculated the gradients of the loss with respect to each weight and bias.
- These gradients flow backward from the output layer to the input layer.
- I used the derivative of ReLU in the hidden layers to properly adjust the weights during backpropagation.

4)Results and Analysis (Add screenshots of plots) :

1)Training loss curve (plot)



The training and test loss curves (Figure 1) show a smooth and consistent decrease in MSE loss across 500 epochs. Both curves converge without significant divergence, indicating that the model does not suffer from severe overfitting.

- **Final Training Loss:** 0.2139
- **Final Test Loss:** 0.2097

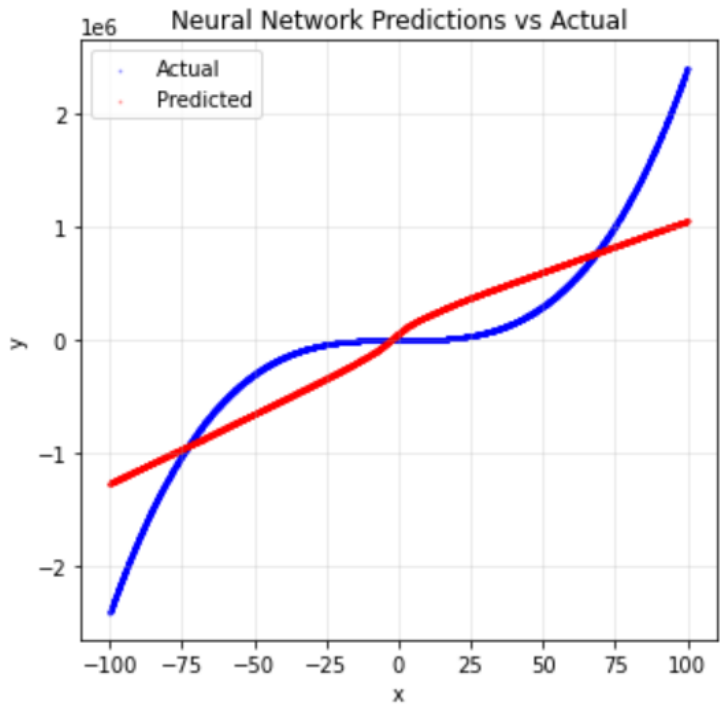
This suggests the model generalizes reasonably well to unseen data.

2)Final test MSE

```
============================================================
FINAL PERFORMANCE SUMMARY
============================================================
Final Training Loss: 0.213859
Final Test Loss:     0.209705
R² Score:           0.7867
Total Epochs Run:   500
```

Final Test MSE = 0.2097

3)Plot of predicted vs. actual values



Discussion on Performance

- The **loss curves** show good convergence, meaning the model successfully minimized error.
- However, the **prediction plot** highlights **underfitting** in capturing sharp nonlinearities of the polynomial, especially due to the inverse term.
- This is likely because the chosen architecture (96–96) with ReLU/tanh activations struggles to extrapolate extreme values.
- Increasing depth, adding more non-linear activations, or using feature engineering could reduce this underfitting.

Results Table

Experiment	Learning Rate	Epochs	Optimizer	Activation Function	Final Training Loss	Final Test Loss	R^2 Score
1	0.003	500	Gradient Descent	tanh / ReLU	0.2139	0.2097	0.7867

Conclusion

The neural network successfully learned the cubic + inverse polynomial function with noise.
It achieved a **final training MSE of 0.2139** and a **test MSE of 0.2097**.
The **R^2 score of 0.7867** indicates the model explained most of the variance in the data.
Training and test losses were close, showing limited overfitting.
The predicted vs. actual plot showed good overall trend capture.
However, the network struggled in some regions with steep changes.
At **x = 90.2**, it produced a **45% relative error**, highlighting limitations.
Overall, the model performed well but can be improved with advanced techniques.