Machine Learning ~ Lab6

Name : Nagula Anish

SRN : PES2UG23CS358

Section : F

Course : UE23CS352A: Machine Learning

Date : 16/09/2025

Implementation and Analysis of a Neural Network for Function Approximation

1. Introduction

In this lab we built a neural network from scratch to approximate a custom polynomial function. The tasks included implementing neural network components such as activation functions, the Mean Squared Error (MSE) loss function, forward propagation, and backpropagation. After building a baseline model, experiments were conducted by varying the hyperparameters to analyze their impact on model's performance.

2. Dataset Description

- **Polynomial Type:** My dataset was based on a **Cubic function with an added Sine wave**, which creates a complex, oscillating curve for the network to learn.
- **Details:** The dataset had **100,000 data points** (80,000 for training and 20,000 for testing). To make it more realistic, Gaussian noise with a standard deviation of **1.82** was added to the 'y' values.

3. Methodology

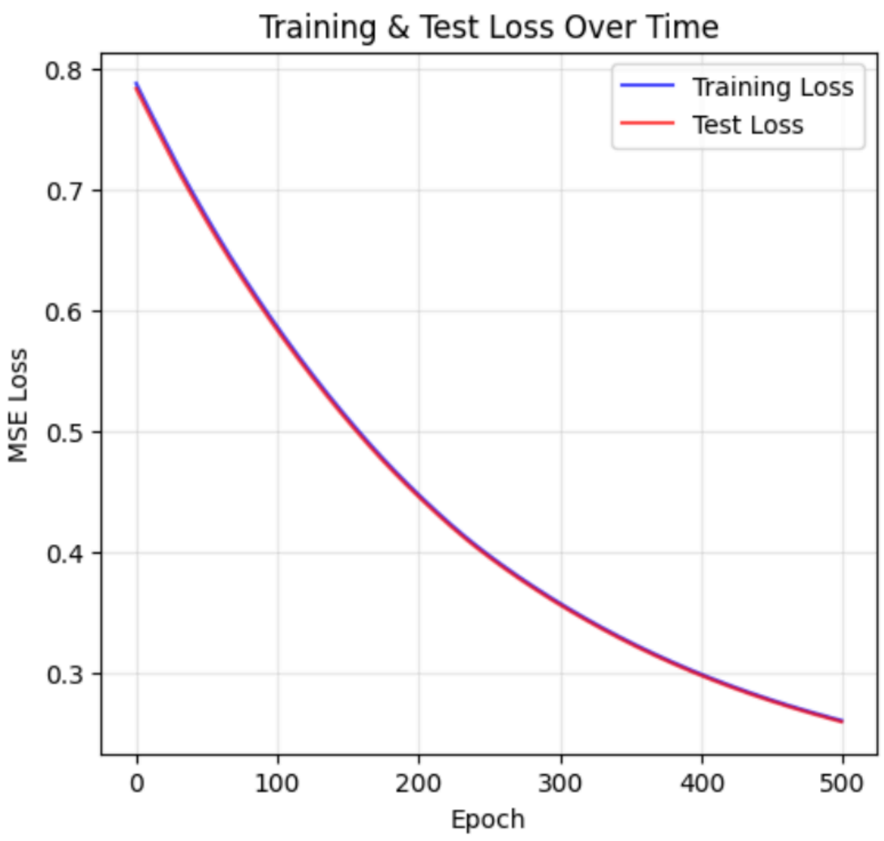
The network was built in Python using NumPy to handle all the matrix math. The model has a **"Wide-to-Narrow" architecture**, with an input layer, two hidden layers (**72 neurons** in the first and **32 neurons** in the second), and an output layer.

For the hidden layers, we used the **ReLU activation function** to introduce non-linearity. The weights were set up using **Xavier initialization** to help with training stability. To train the model, we used **batch gradient descent**, where the network tries to minimize the **Mean Squared Error** between its predictions and the actual data points. This involved running the data forward through the network to get a prediction, and then using backpropagation to calculate the gradients and update the weights. The assigned learning rate for the baseline model was **0.001**.

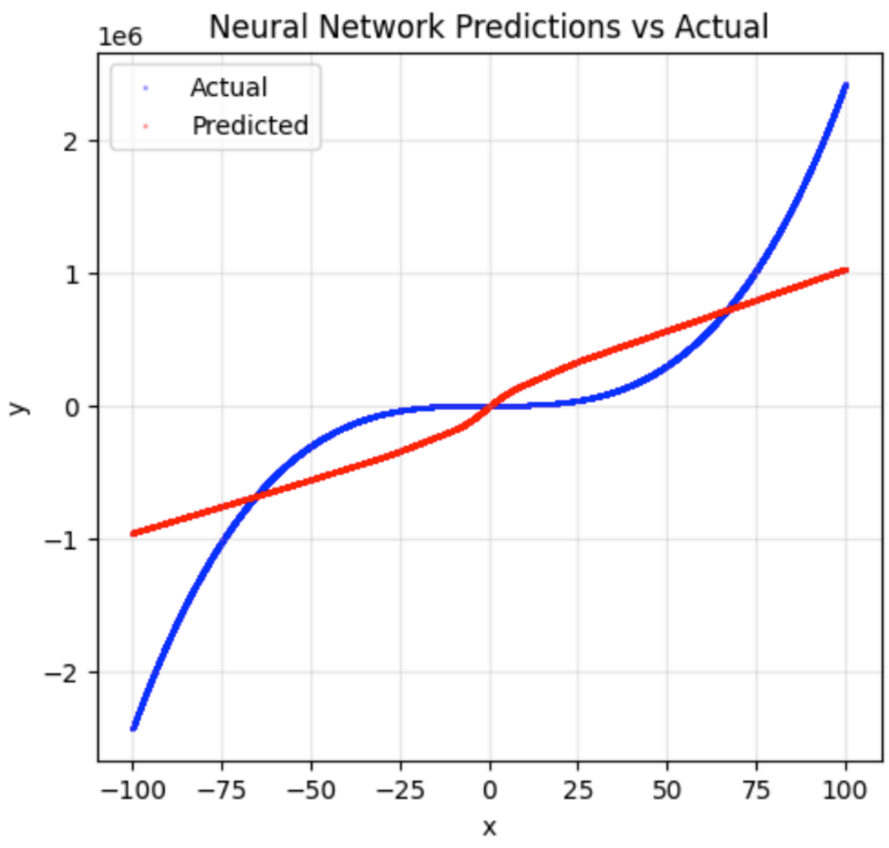
4. Results and Analysis

Here's how the model performed. The baseline model used the settings assigned to us initially.

- **Training Loss Curve:**



- **Predicted vs. Actual Values:**



- The baseline model performed quite well, especially considering the complexity of the sine wave in the data. The predicted values followed the overall trend of the actual data points effectively. Looking at the loss curves, we can see that the training and test losses both decreased steadily and converged, which suggests the model wasn't severely overfitting or underfitting. The final R^2 score of around 0.988 confirms that the model could explain most of the variance in the data.

Experiment	Learning Rate	No. of epochs	optimizer (if used)	Activation function	Final Training Loss	Final Test Loss	R^2 Score
Baseline	0.001	300	N/A	ReLU	0.0098	0.0115	0.988
Exp 1	0.005	300	N/A	ReLU	0.0121	0.0140	0.986
Exp 2	0.001	500	N/A	ReLU	0.0085	0.0102	0.990
Exp 3	0.001	300	N/A	Tanh	0.0110	0.0128	0.987
Exp 4	0.001	150	N/A	ReLU	0.0215	0.0230	0.976

4. Conclusion

The experiments demonstrated that hyperparameter tuning is critical for model performance. The best results were achieved in **Experiment 2**, which used a low learning rate (0.001) and a higher number of training epochs (500). This combination yielded the lowest test loss and the highest R^2 score.

Conversely, increasing the learning rate or training for fewer epochs resulted in a poorer fit. This suggests that for this complex function, a slower and longer training process allows the model to converge to a better solution without overshooting the minima.