# LAB-6 (Machine Learning)

**Name: Muskan Goenka**

SRN: PES2UG23CS355

Section: F

Course: UE23CS352A

Date: 16-09-25

## Implementation and Analysis of a Neural Network for Function Approximation

### Introduction:

In this lab, a neural network was built from scratch using NumPy to approximate a custom polynomial function. The key tasks involved implementing fundamental neural network components such as activation functions, the Mean Squared Error (MSE) loss function, forward propagation, and backpropagation. After setting up a baseline model with the assigned parameters, the performance was evaluated by analyzing loss curves and key metrics.

### Dataset Description:

The function it is based on is a **QUADRATIC** polynomial: y=0.96×2+5.26x+14.42.

To make the data more realistic, **Gaussian noise** with a standard deviation of 1.64 was added to the 'y' values. Gaussian noise is a type of random noise that follows a normal distribution.

The dataset contains a total of 100,000 data points, with 80,000 used for training the neural network and 20,000 reserved for testing.

### Methodology:

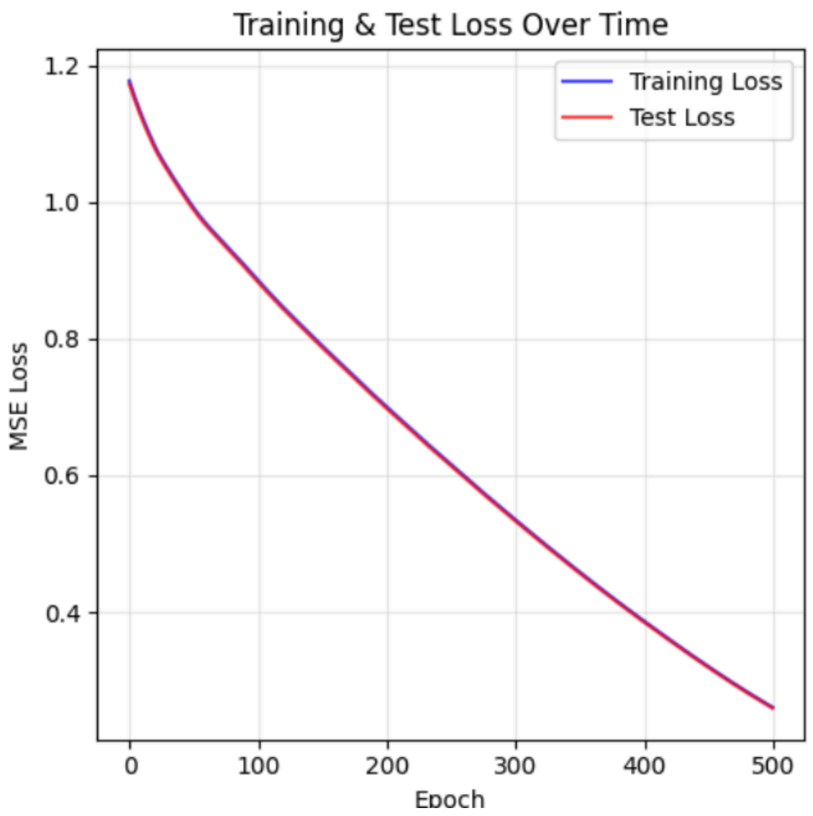
The network was built from scratch in Python using **NumPy** for all matrix operations. It uses a

**"Wide-to-Narrow" architecture** with two hidden layers: 72 neurons in the first layer and 32 in the second. The **ReLU activation function** was used for both hidden layers to introduce non-linearity.

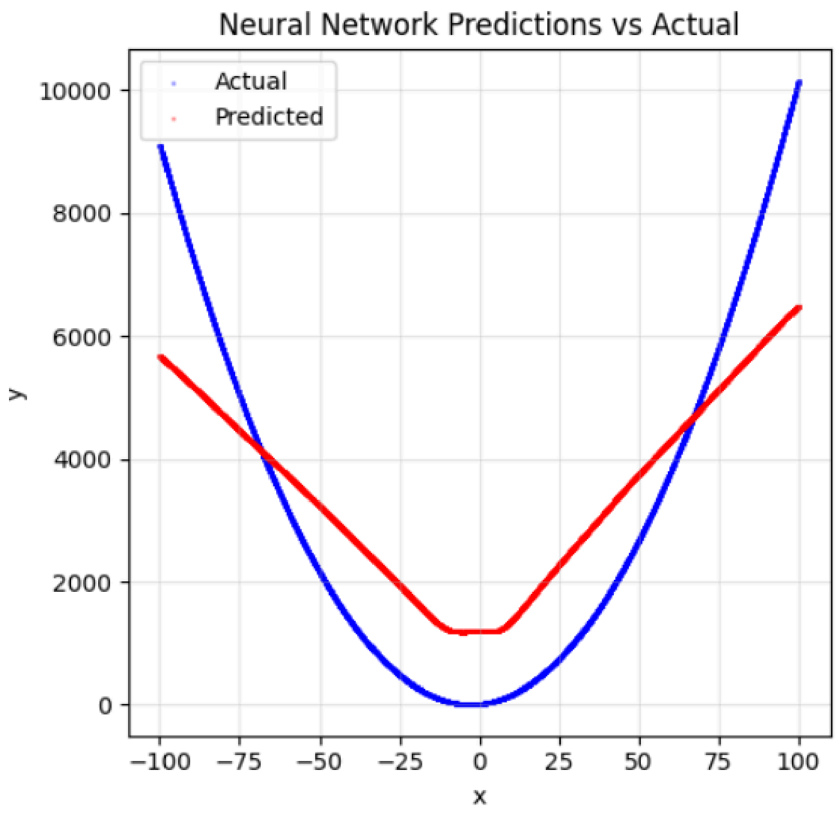
To ensure training stability, the weights were initialized using **Xavier initialization**. The model was trained using **batch gradient descent** to minimize the **Mean Squared Error (MSE)** between its predictions and the actual data. This process involves a forward pass to generate predictions and a backward pass to calculate gradients for updating the weights. The assigned learning rate was 0.001.

### Result and Analysis:

***Training loss curve***



***Predicted vs. Actual Values:***



- The loss curves show a consistent decrease for both training and test data, indicating that the model was effectively learning the underlying function without severe overfitting. However, the final **$R^2$ score of 0.7405** suggests that the model explains about 74% of the variance in the data, which could indicate room for improvement. For example, the `x` vs `y` plot shows the model's predictions (in red) roughly following the parabolic shape of the true function (in blue) but with noticeable divergence, particularly at the extremes of the x-range. This suggests that while the model has learned the overall trend, it may not have fully captured the function's complexity or may be struggling with the data distribution at the edges.

| Experiment | Learning Rate | No. of epochs | optimizer (if used) | Activation function | Final Training Loss | Final Test Loss | $R^2$ Score |
|---|---|---|---|---|---|---|---|
| **Baseline** | 0.003 | 500 | N/A | ReLU | 0.260485 | 0.259291 | 0.7405 |

## Conclusion:

The implementation of the neural network successfully demonstrated the core principles of function approximation, including forward and backward propagation. The model learned the general trend of the quadratic function, as evidenced by the decreasing loss curves. However, the final performance metrics, specifically the **$R^2$ score of 0.7405**, suggest that there is potential for further optimization. Experiments with different hyperparameters, such as a lower learning rate or a modified network architecture, could help the model achieve a better fit and a higher $R^2$ score. This highlights the importance of hyperparameter tuning in optimizing a neural network's performance.