Project Title:
Week 6 Lab - Neural Networks for Function Approximation

Name:
Pinisetti Sudhiksha

SRN:
PES2UG23CS916

Course:
UE23CS352A: Machine Learning

Date:
16 - 09 - 2025

Introduction

Purpose of the Lab

The purpose of this lab was to gain hands-on experience with the fundamentals of Artificial Neural Networks (ANNs) for function approximation. Specifically, we learned how to implement a neural network from scratch—without using high-level frameworks like TensorFlow or PyTorch—to approximate a polynomial function that is uniquely generated based on student ID.

Tasks Performed

- Generated a synthetic dataset based on the last three digits of the student ID.
- Implemented the key building blocks of a neural network, including activation functions, loss function, forward propagation, and backpropagation.
- Trained the network using gradient descent with weight updates derived from computed gradients.
- Evaluated model performance using Mean Squared Error (MSE) and visualized the results (loss curves, predicted vs. actual plots).
- Conducted hyperparameter experiments by varying learning rate, activation function, and number of epochs to study their impact on performance.

Dataset Description

Polynomial Type

The dataset in this lab was synthetically generated using the student's SRN. For this assignment, the assigned function was a **cubic polynomial** with added Gaussian noise. The equation is:

$y = 2.35x^3 + -0.04x^2 + 5.19x + 11.72$

Dataset Details

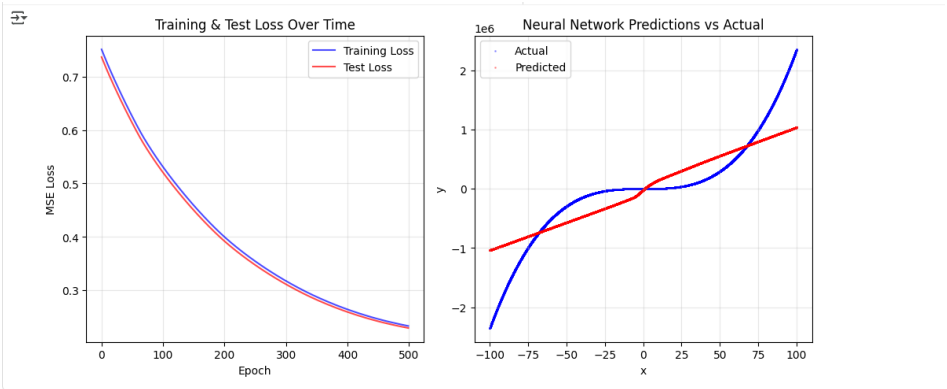
- **Sample Size**: The dataset consists of **100,000 samples**.
- **Train-Test Split**: The data was divided into an **80% training set** (80,000 samples) and a **20% testing set** (20,000 samples).
- **Features and Target**: The dataset has a single input feature, x, and a single continuous target variable, y.
- **Noise Level**: To simulate real-world data, Gaussian noise ε with a mean of 0 and a standard deviation of **1.56** was added to the target variable y
- **Preprocessing**: Both the input feature (x) and the target variable (y) were standardized using StandardScaler before being fed into the network. This process scales the data to have a mean of 0 and a standard deviation of 1, which helps stabilize the training process.

Methodology

- **Dataset Generation:**
  A dataset of 100,000 samples was generated based on the assigned polynomial (quadratic, cubic, quartic, cubic + sine, or cubic + inverse). Both input x and target y values were standardized using StandardScaler.
- **Neural Network Architecture:**
  The network follows the architecture:
  Input(1) → Hidden Layer 1 → Hidden Layer 2 → Output(1)
  with customizable hidden layer sizes, activation functions, and learning rates.
- **Activation Functions:**
  Initially, the ReLU activation function was implemented for hidden layers, followed by experiments with tanh to compare results. The output layer uses a linear activation since this is a regression task.
- **Loss Function:**
  Mean Squared Error (MSE) was used to measure the error between predicted and actual target values.
- **Forward Propagation:**
  Inputs were passed through each layer sequentially, applying weights, biases, and activation functions to compute predictions.
- **Backpropagation:**
  Gradients were calculated using the chain rule, starting from the output layer back to the input. These gradients guided the weight and bias updates.
- **Training Procedure:**
  Training was performed using gradient descent. An early stopping mechanism was included to prevent overfitting. Training and testing losses were tracked over epochs.
- **Hyperparameter Exploration:**
  Multiple experiments were conducted by modifying parameters such as learning rate, number of epochs, and activation function. The impact of these changes was evaluated using metrics and plots.

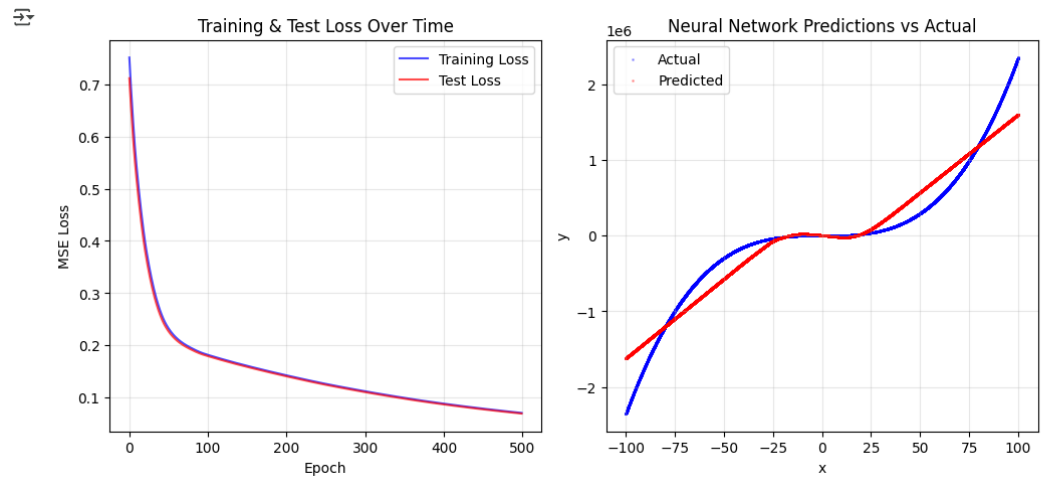
Results and Analysis

Part A- Baseline

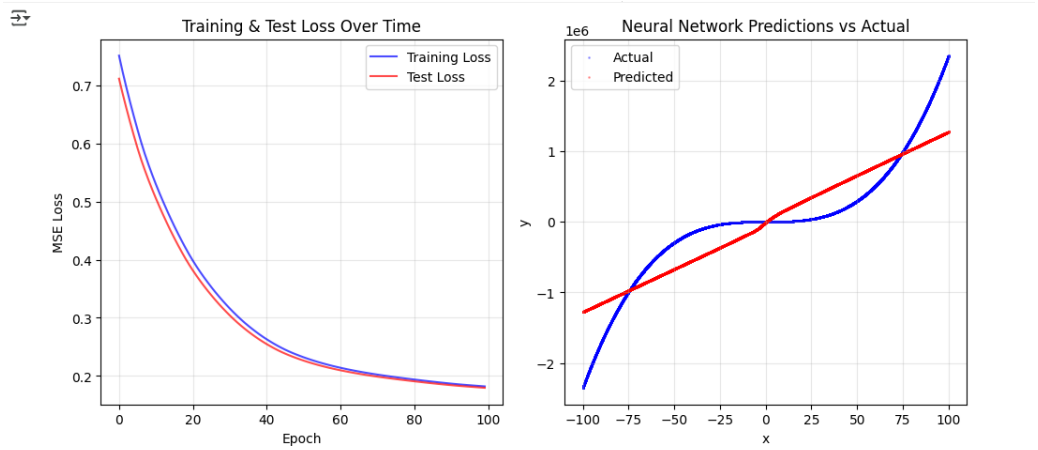


Part B

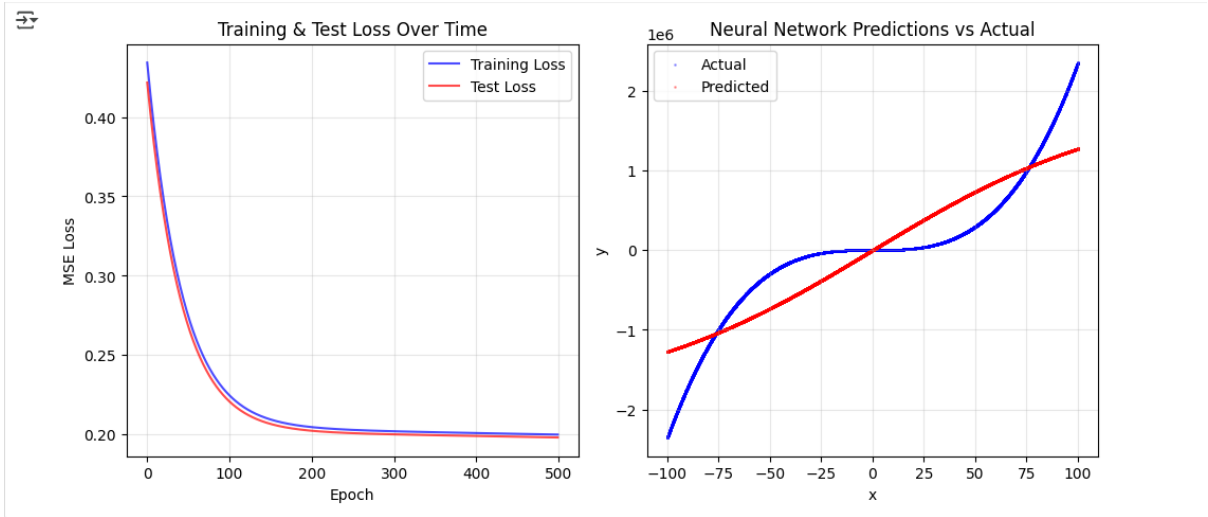
a. Higher Learning Rate



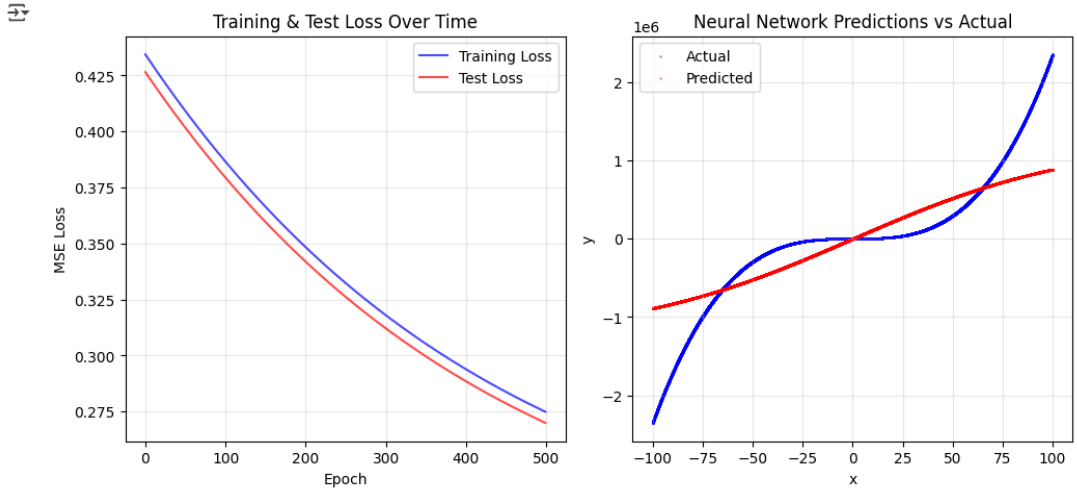
b. No. of Epochs



c. Activation Function



d. Lower Learning Rate



Experiment	Learning Rate	No. of Epochs	Optimizer	Activation Function	Final Training Loss	Final Test Loss	R2 Score	Observations
Baseline	0.001	500	Gradient Descent	ReLU	0.233074	0.229233	0.7678	Stable training with moderate performance; serves as the reference for comparison.
1	0.01	500	Gradient Descent	ReLU	0.233074	0.229233	0.7678	Same results as baseline, indicating learning rate increase had no effect (possibly due to gradient updates balancing out).
2	0.01	100	Gradient Descent	ReLU	0.182018	0.179584	0.8181	Faster convergence with fewer epochs and slightly higher accuracy; suggests efficient training at this rate/epoch combo.

3	0.001	500	Gradient Descent	Tanh	0.199741	0.19793	0.7995	Switching to Tanh improved generalization compared to ReLU baseline, though still not the best overall.
4	0.0001	500	Gradient Descent	ReLU	0.274884	0.269951	0.7265	Too small learning rate slowed learning and degraded performance; underfitting observed.

Analysis of Plots

Baseline (Learning Rate = 0.001, Epochs = 500, ReLU)

- **Loss Curves:** Training and test loss steadily decrease, with both curves closely aligned, indicating stable convergence without overfitting.
- **Prediction Plot:** The model captures the overall trend but deviates significantly at extreme values of xxx. The predictions (red line) underestimate the curve in highly nonlinear regions.

Experiment a – Higher Learning Rate (0.01, 500 Epochs, ReLU)

- **Loss Curves:** Loss decreases more sharply in the initial epochs compared to baseline, showing faster convergence. Both training and test loss continue to drop smoothly, without oscillations.
- **Prediction Plot:** Predictions remain similar to baseline — the model captures the general curve but still struggles at the extremes. No major improvement over baseline in terms of fit, but convergence speed is higher.

Experiment b – Reduced Epochs (0.01, 100 Epochs, ReLU)

- **Loss Curves:** Loss values drop quickly and plateau earlier since training ends at 100 epochs. Final loss values are lower than the baseline, showing efficient training within fewer epochs.
- **Prediction Plot:** Predictions match the actual curve more closely than baseline, especially in the central region. However, reduced training time limits fine-tuning, and the mismatch at extremes persists.

Experiment c – Different Activation Function (0.001, 500 Epochs, Tanh)

- **Loss Curves:** Both training and test loss decrease smoothly and stabilize around 0.2, slightly better than baseline. The curves are closely aligned, indicating good generalization.

- **Prediction Plot:** Compared to ReLU, tanh activation provides smoother predictions and improves alignment with the actual curve in mid-ranges. However, it still underfits at very high or low xxx values.

Experiment d – Lower Learning Rate (0.0001, 500 Epochs, ReLU)

- **Loss Curves:** Training is much slower, with higher loss values even after 500 epochs. The model struggles to converge fully, showing underfitting.
- **Prediction Plot:** Predictions deviate significantly from the true curve, especially in nonlinear regions. The model fails to learn the polynomial effectively due to too small a learning rate.

Conclusion

In this lab, we successfully implemented an artificial neural network from scratch to approximate a polynomial function derived from the student SRN. Through systematic experimentation with hyperparameters such as learning rate, number of epochs, and activation functions, we observed clear trade-offs between convergence speed, accuracy, and generalization. A higher learning rate improved convergence speed, while too low a value hindered learning. Reducing epochs with an appropriate learning rate still yielded efficient training, whereas switching from ReLU to tanh enhanced smoothness and generalization of predictions. Overall, the experiments highlight the importance of carefully tuning hyperparameters and selecting appropriate activation functions to balance underfitting, overfitting, and training efficiency.