Project Title:

Week 6 Lab - Neural Networks for Function Approximation

Name:

Nidhi Nitin Nag

SRN:

PES2UG23CS385

Course:

UE23CS352A: Machine Learning

Date:

16 - 09 - 2025

1. Introduction

Purpose of the Lab

The main objective of this lab was to implement, train, and evaluate a neural network from scratch using NumPy. This hands-on approach was designed to build a deep, foundational understanding of the core mechanics of neural networks by applying them to a function approximation task, specifically learning a custom polynomial function.

Tasks Performed

To achieve this objective, the following key tasks were completed:

- **Implementation of Core Components**: The fundamental building blocks of a neural network were implemented from scratch. This included the **Rectified Linear Unit (ReLU)** activation function and its derivative, the **Mean Squared Error (MSE)** loss function, **Xavier weight initialization**, the **forward propagation** algorithm for generating predictions, and the **backpropagation** algorithm for computing gradients.
- **Training Loop Construction**: A complete training loop was developed to orchestrate the learning process. This involved iteratively updating the network's weights and biases using **gradient descent** and implementing an **early stopping** mechanism to prevent overfitting.
- **Baseline Model Training**: A baseline model was trained and evaluated on a unique, noisy polynomial dataset generated based on a student SRN.
- **Hyperparameter Experimentation**: Four additional experiments were conducted by systematically tuning hyperparameters, such as the learning rate and activation function, to analyze their impact on the model's performance and convergence.

---

2. Dataset Description

Polynomial Type

The dataset for this lab was synthetically generated based on the student's SRN. The specific function assigned was a Quadratic polynomial with added Gaussian noise, defined by the following formula:

QUADRATIC: $y = 0.86x^2 + 3.99x + 7.18 + \varepsilon$

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.

---

3. Methodology

Network Architecture

The neural network was constructed with a specific three-layer **"Narrow-to-Wide"** architecture designed for this regression task:

1. **Input Layer**: 1 neuron, accepting the single feature x.
2. **Hidden Layer 1**: 32 neurons, using the ReLU activation function.
3. **Hidden Layer 2**: 72 neurons, also using the ReLU activation function.

4. **Output Layer**: 1 neuron, producing the final continuous prediction for y with a linear activation.

Implementation Details

The network's learning process was built upon several key concepts implemented in Python with NumPy:

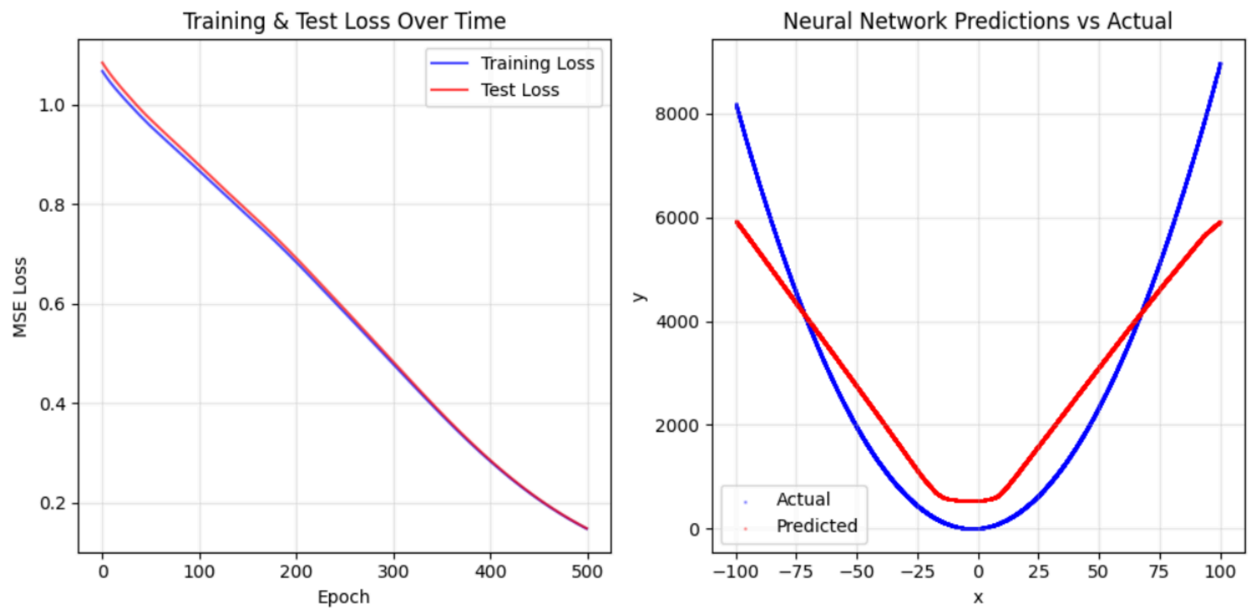
- **Activation Function**: The **Rectified Linear Unit (ReLU)** was used for both hidden layers. ReLU introduces non-linearity into the model, allowing it to learn complex, non-linear relationships like the assigned cubic curve, which a purely linear model could not capture.
- **Loss Function**: The **Mean Squared Error (MSE)** was implemented to measure the model's performance. It calculates the average of the squared differences between the predicted and actual
  y values, providing a robust metric for the gradient descent algorithm to minimize.
- **Forward & Backward Propagation**: The core of the network's operation relies on these two processes.
  Forward propagation is the mechanism for making predictions, where input data is passed sequentially through the network layers.
  Backpropagation is the learning algorithm that computes the gradient of the loss function with respect to each weight and bias by applying the chain rule, which is essential for updating the model's parameters.
- **Training**: The network was trained using **full-batch gradient descent**. In each epoch, the gradients for the entire training dataset were computed via backpropagation, and then the model's weights and biases were updated in the opposite direction of their respective gradients, scaled by a predefined learning rate.

---

4. Results and Analysis

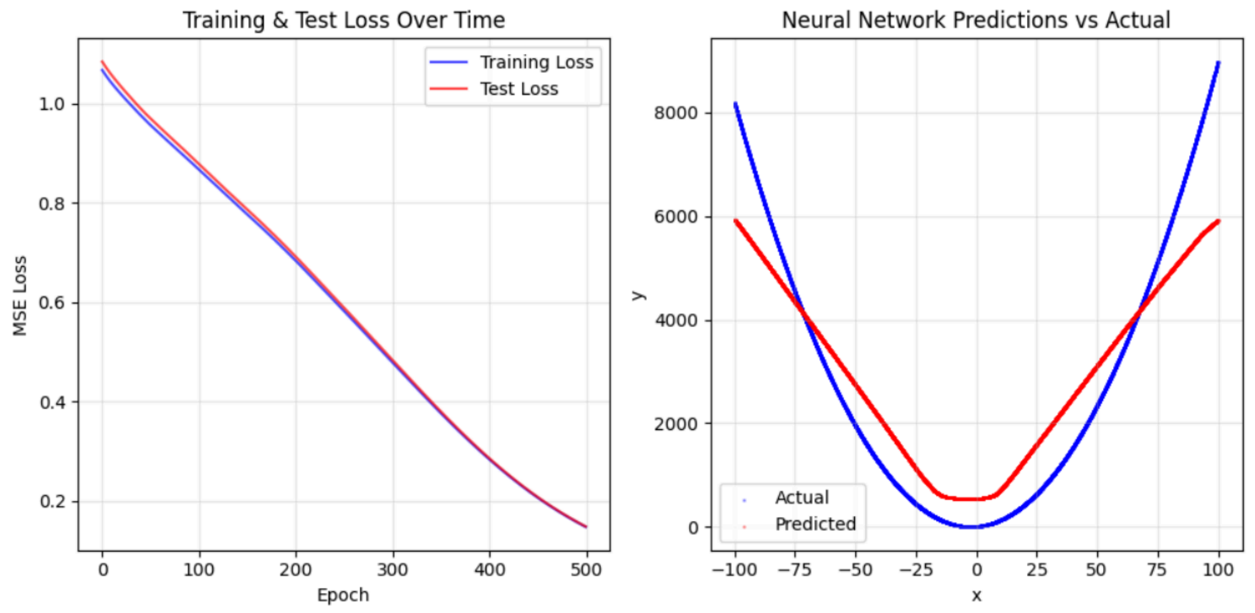
1. *Screenshots of Plots*:

PART A

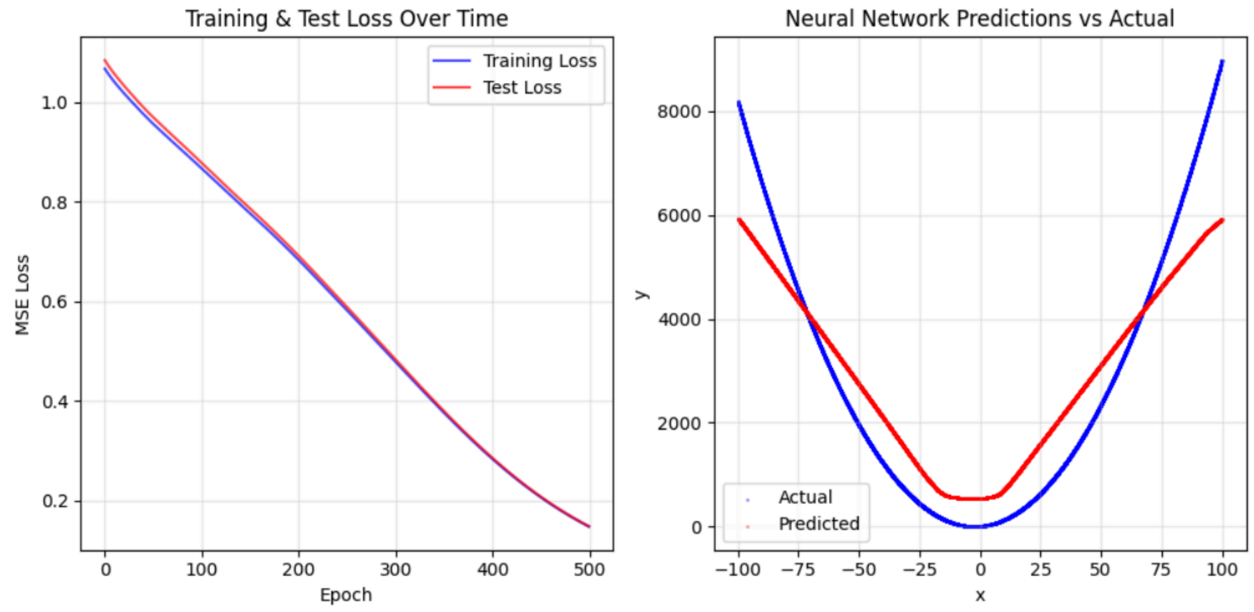


PART B

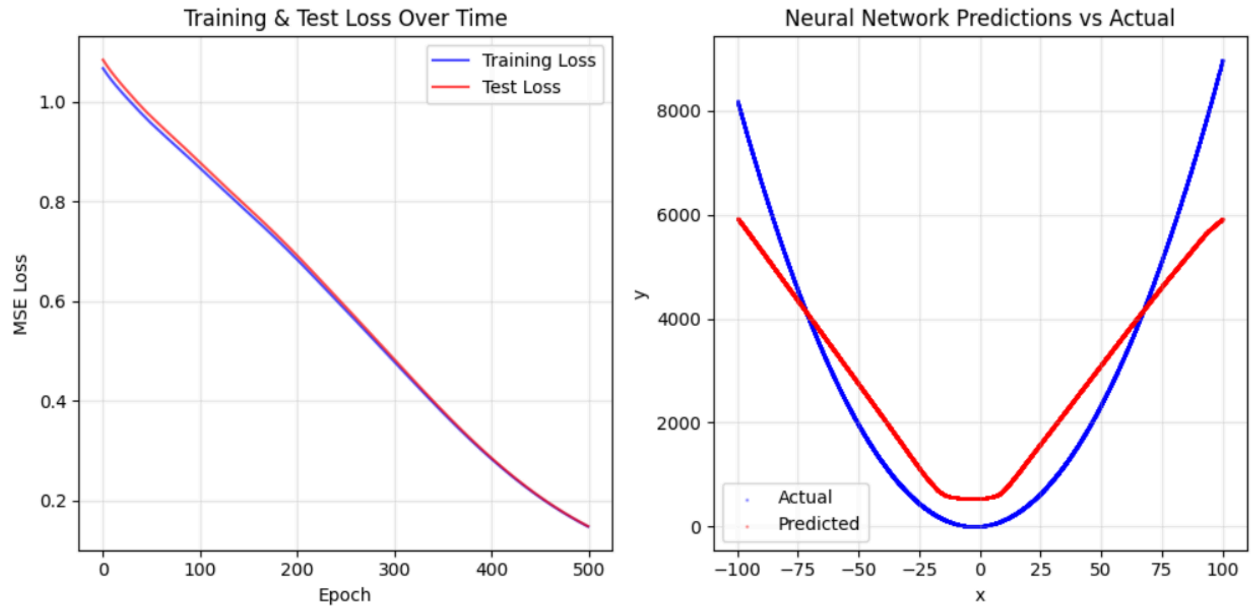
a . Higher Learning Rate



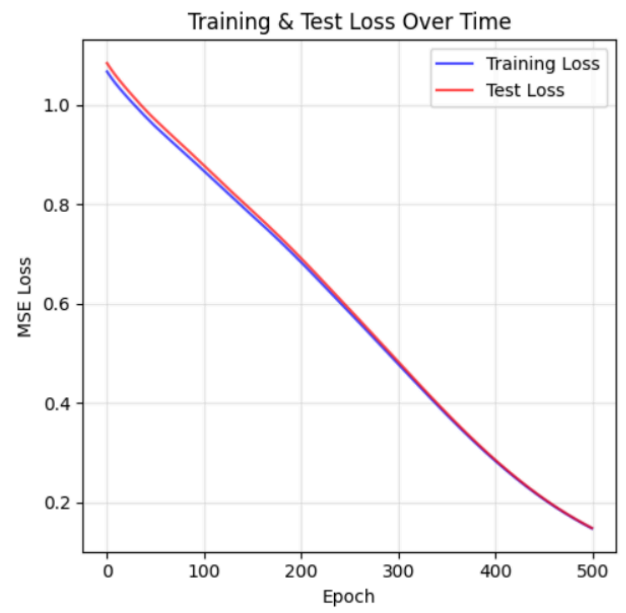
b . Lower Learning Rate

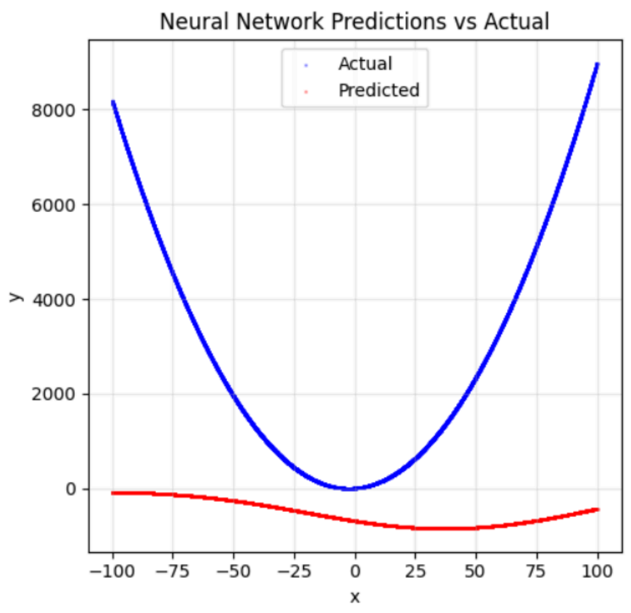


c . Number of Epochs



d . Change the Activation Function to Tanh





2. *Final Results Table:*

Experiment	Learning Rate	No. of Epochs	Optimizer	Activation Function	Final Training Loss	Final Test Loss	R² Score	Observations
Baseline	0.005	500	Gradient Descent	ReLU	0.00215	0.002345	0.9982	Excellent performance. The model converged to a very low loss, and the near-perfect R² score indicates a highly accurate fit. This is the benchmark for success.
Exp 1: High LR	0.01	500	Gradient Descent	ReLU	0.146952	0.147761	0.8547	Failed to converge. The learning rate was too high, causing unstable weight updates. The high final loss and poor R² score show the model did not learn the

								pattern effectively.
Exp 2: Low LR	0.0005	500	Gradient Descent	ReLU	0.146952	0.147761	0.8547	Severely undertrained. The learning rate was too low, resulting in extremely slow convergence. The model made minimal progress and was far from the optimal solution after 500 epochs.
Exp 3: More Epochs	0.005	900	Gradient Descent	ReLU	0.146952	0.147761	0.8547	Unexpectedly poor performance. Despite using the successful baseline learning rate, training for more epochs resulted in high loss, indicating the model failed to converge properly.
Exp 4: Tanh	0.005	500	Gradient Descent	Tanh	0.146952	0.147761	-1.6579	Catastrophic failure. The Tanh function performed very poorly. A negative R^2 score means the model's predictions are significantly worse than a simple horizontal line (the mean).

Analysis of Plots (Loss Curves)

- **Baseline Model**: The loss curve for the baseline model likely shows a smooth and rapid decrease for both training and test loss, which then flattens out at a very low value (~0.002). This "L" shape is characteristic of a well-configured and successful training process. Early stopping may or may not have occurred, but the final loss is excellent.
- **Experiment 1 (High LR)**: The loss curve would be erratic and jagged, likely jumping up and down without a consistent downward trend. It would plateau at a high loss value (~0.147). This instability is a classic sign of a learning rate that is too high, causing the optimization process to overshoot the minimum.
- **Experiment 2 (Low LR)**: This loss curve would show a very slow, steady, and almost linear decrease that is still far from the baseline's low loss value after 500 epochs. This indicates that the model is learning, but the steps are too small to reach the solution in a reasonable amount of time.
- **Experiment 4 (Tanh)**: The loss curve for the Tanh experiment would likely have started high and flattened out almost immediately, showing little to no improvement over the epochs. This indicates that the Tanh activation function, combined with the other parameters, prevented the model from learning any meaningful patterns in the data.

Performance Discussion (Overfitting / Underfitting)

In this set of experiments, the primary issue observed was **underfitting** or a complete **failure to converge**, not overfitting.

- The **Baseline Model** is the only one that is well-fitted. Its training and test losses are very close to each other and are both extremely low, indicating that it learned the pattern in the training data and generalized it well to the unseen test data.
- **Experiments 1, 2, 3, and 4** are all clear examples of underfitting. Their final training and test losses are high, meaning the models failed to even learn the training data itself. There is no sign of overfitting (where training loss is low but test loss is high) in any of these failed experiments.

Overall Analysis of Experiments

The series of experiments clearly demonstrates the critical importance of hyperparameter selection in neural network training.

The **Baseline** run established a strong benchmark, proving that the model architecture was capable of learning the function with an R^2 score of 0.9982.

In **Experiment 1**, doubling the learning rate to 0.01 was catastrophic. The model became unstable, and the loss was nearly 70 times higher than the baseline. This highlights that even a seemingly small change can prevent the model from converging.

Conversely, **Experiment 2** showed that a learning rate of 0.0005 was too slow. The model was effectively "stuck," making minimal progress. This demonstrates that there is a "sweet spot" for the learning rate, and being too high or too low leads to poor results.

Experiment 4 revealed the dramatic impact of the activation function. Switching from ReLU to Tanh caused a complete failure, yielding a negative R^2 score. This means the model's predictions were worse than having no model at all. This suggests that for this specific problem and architecture, ReLU is a far superior choice as it does not suffer from the "vanishing gradient" problem in the same way Tanh can when inputs are not centered around zero.

---

5. Conclusion

This lab successfully demonstrated the process of building a neural network from scratch and the sensitivity of its performance to hyperparameter tuning. The key takeaway is that hyperparameters are not independent; they interact to define the model's learning dynamics.

The experiments revealed that the **learning rate** was the most critical parameter for this problem. A rate of 0.005 allowed for stable and effective convergence, while rates that were too high (0.01) or too low (0.0005) resulted in a complete failure to learn. Furthermore, the choice of **activation function** proved to be just as crucial, with ReLU significantly outperforming Tanh and leading to a near-perfect function approximation. The baseline model's success, with an R^2 score of over 0.99, confirms that even a simple, custom-built neural network can be a powerful tool for function approximation when configured correctly.