LAB-4

NAME: NIDHI N| SRN:PES2UG23CS384| SEC:F

Purpose of the Lab:

The main point of this lab was to build a neural network from scratch. Instead of using powerful libraries that do everything for you, we wanted to understand how a network really works by putting all the core pieces together ourselves. We then used our custom-built network to learn and predict the values of a polynomial function.

Tasks Performed

To do this, we followed a few main steps. We started by generating our own unique dataset. Then, we wrote the code for the network's key parts: the ReLU activation function, the MSE loss function, and the forward and backward passes. After building the network, we trained it with a method called gradient descent and checked its performance. Finally, we ran four more experiments, changing things like the learning rate, to see how it affected the results.

We started with a dataset that we created ourselves using code. The data follows the pattern of a Cubic function with an Inverse term, which was assigned based on the student SRN 'PES2UG23CS384'.

We made 100,000 data points in total. Each point had one input number (x) and one output number (y). To make the data more realistic, like something you'd find in the real world, we added a bit of random 'noise' to the y values.

Before giving the data to our network, we prepared it. We split it into a training set (80%) and a

testing set (20%). We also standardized all the numbers, which resets the data to be centered around zero. This simple step helps the network learn much more effectively.

3. Methodology

The network we built had a straightforward structure, with an input layer, two hidden layers, and an output layer. Specifically, it had:

- An Input Layer that takes in one number.
- Two Hidden Layers in the middle, each with 96 neurons.
- An Output Layer that produces one number as the final prediction.

We had to code all the parts that make this structure work:

- Activation Function: The hidden layers used the ReLU function. Its job is to add complexity, which lets the network learn curved patterns instead of just straight lines.
- Initialization: We used Xavier initialization to set the network's starting weights. This technique provides a smart starting point that helps the training process begin smoothly. Biases were all set to zero.
- Training Process: The model learned using gradient descent. In each training cycle, it would first make a prediction in a forward pass. Then, it would use the MSE loss function to

calculate how wrong that prediction was. Next, backpropagation would work backward to figure out exactly how much each weight and bias contributed to the error. Finally, we would update those weights and biases just a tiny bit, using a learning rate of 0.003, so the network would be slightly more accurate on the next cycle. To avoid training for too long, we used early stopping, which automatically stopped the process if the model's performance on the test data didn't improve for 10 cycles.

Experiment	Learning Rate	No. of epochs	optimizer (if used)	Activation function	Final Training Loss	Final Test Loss	R^2 Score
Baseline	0.001	500	Gradient Descent	ReLU	0.284968	0.283181	0.7144