Neural Network Implementation from Scratch for Regression and Classification Tasks

Mini Project Report

Authors:
Hillol Pratim Kalita, 210710007016
Rishika Hazarika, 210710007037

Supervised By:
Mr. Diganta Baishya

Associate Professor
Department of Computer Science & Engineering
Jorhat Engineering College

Acknowledgement

We are immensely grateful to **Mr. Diganta Baishya**, Associate Professor of the Computer Science and Engineering Department, Jorhat Engineering College, for his invaluable guidance, support, and encouragement throughout the duration of our mini project. His expertise and insights have been instrumental in shaping our work and helping us overcome various challenges.

We would also like to express our heartfelt gratitude to **Dr. Rupam Baruah**, Head of the Department and Principal of Jorhat Engineering College, for providing us with the opportunity and the resources necessary to undertake this project. His leadership and commitment to academic excellence have greatly inspired us.

Their combined support and mentorship have been pivotal in the successful completion of this project, and for that, we are deeply thankful.

Self-Declaration

We, the undersigned, declare that the mini project entitled **Neural Network implementation for Regression and Classification** is a record of work carried out by us during the academic year 2024 under the supervision of **Mr. Diganta Baishya**, Associate Professor of the Computer Science and Engineering Department, Jorhat Engineering College. We further declare that this work has not been submitted elsewhere for any other degree or diploma.

Hillol Pratim Kalita
210710007016

Rishika Hazarika
210710007037

1. Abstract

This report aims to detail the design and development of a neural network from scratch, implemented to perform regression and classification tasks. The 2-layer Neural Network essentially demonstrates the basis of finding patterns within given range of data and eventually formulating what the next value would be. The report encompasses the entirety of our project.
Data loading, forward propagation, backpropagation, and gradient updates are managed by Custom Trainer classes, which puts together the network to work. We can observe better validation results in terms of accuracy and generalization ability to novel samples due to appropriate choices made concerning activation functions and evaluation metrics. This report provides significant insights into the design and implementation of neural networks to carry out a variety of machine learning tasks.

2. Introduction

2.1. Background

Neural networks have become the anchors of the machine learning domain, enabling developers to create advanced, sophisticated models for a wide range of predictive and decision-based problems. Even if the most popular, mainstream frameworks like TensorFlow and PyTorch allow users with easy implementation of neural networks through pre-defined modules, it is imperative to understand the fundamentals and principles of neural networks in order to design and optimize models effectively.

2.2. Objectives

The primary objectives of this project are as follows:

1. Explore neural network architecture,creation and optimization for different machine learning problems with NumPy.
2. Generate synthetic datasets to ease the evaluation of the model
3. Implementing different design choices, such as network architecture and activation functions.
4. Making deductions and drawing conclusions based on the performance of the neural network.

3. Methodology

3.1. Primary Implementation: Single-layer Perceptron using synthetic data

1. **Activation and Loss Functions:**
   - We utilized a linear activation function given by:

$$a(z) = z \tag{1}$$

   - Mean Squared Error (MSE) was used as the loss function, given by:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2}$$

2. **Initialization of Parameters:**
   - We randomly initialized weights for height (W_{height}) and weight (W_{weight}), and the bias (b).

3. **Hyperparameters:**
   - The learning rate (α) and number of epochs for training were set.

4. **Synthetic Data Generation:**
   - We generated random synthetic data for height and weight.
   - Synthetic body fat percentage data was created using a linear function of height and weight coupled with noise:

$$\text{body\_fat\_percentage} = 0.25 \times \text{height} + 0.2 \times \text{weight} + \epsilon \tag{3}$$

     where ϵ represents Gaussian noise.

5. **Training the Model:**
   - **Forward Propagation:** Predictions were computed using the linear activation function:

$$\hat{y} = W_{\text{height}} \times \text{height} + W_{\text{weight}} \times \text{weight} + b \tag{4}$$

   - **Loss Calculation:** The loss was calculated using MSE.
   - **Backward Propagation:** Gradients for weights and bias were computed as:

$$\frac{\partial \text{MSE}}{\partial W_{\text{height}}} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i) \times \text{height}_i \tag{5}$$

$$\frac{\partial \text{MSE}}{\partial W_{\text{weight}}} = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) \times \text{weight}_i \tag{6}$$

$$\frac{\partial \text{MSE}}{\partial b} = \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i) \tag{7}$$

- **Parameter Updates:** We updated weights and bias using gradient descent:

$$W_{\text{height}} \leftarrow W_{\text{height}} - \alpha \frac{\partial \text{MSE}}{\partial W_{\text{height}}} \tag{8}$$

$$W_{\text{weight}} \leftarrow W_{\text{weight}} - \alpha \frac{\partial \text{MSE}}{\partial W_{\text{weight}}} \tag{9}$$

$$b \leftarrow b - \alpha \frac{\partial \text{MSE}}{\partial b} \tag{10}$$

- The neural network was trained for 600 epochs. The learning rate was specified at 0.00001. The training process was monitored by printing the loss periodically.

6. **Making Predictions:**
   - The trained neural network model was used to make predictions on the input data.
7. **Model Evaluation:**
   - Mean Squared Error (MSE) and Mean Absolute Error (MAE) were calculated to evaluate the performance of the model.
   - A threshold for accuracy of prediction was defined and the accuracy percentage was computed accordingly.

```
Mean Squared Error: 22.898567012316864
Mean Absolute Error: 3.804099405054029
Accuracy(close upto 5 units)): 71.33333333333334
```

Figure 1: Model accuracy and loss

This methodology outlines the fundamental steps for implementing a single-layer perceptron from scratch, highlighting the key processes involved in neural network training and evaluation.

3.2. Neural Network Implementation

Following the implementation of the single-layer perceptron, we expanded the project to a more modular structure to handle various machine learning tasks effectively. This section provides an overview of each module implemented from scratch for the purpose and describes their roles and interactions within the overall neural network system.

3.2.1 Project Modules Overview

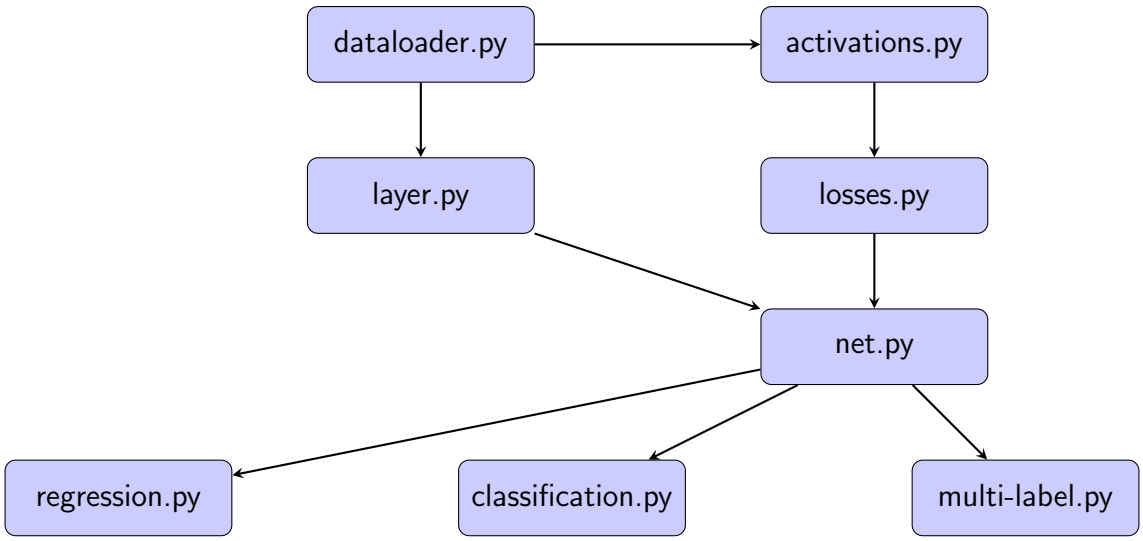

Figure 2: Module Interaction Flowchart

The project is organized into several key modules, each handling a specific aspect of the neural network:

- **activations.py**
  - **Purpose:** Contains various activation functions. Activation functions are essential in deep learning as to introduce non-linearity in the neural network.
  - **Functions Provided:** ReLU, sigmoid, tanh, softmax, linear, leaky_relu, silu.
  - **Key Equations:**

$$\text{ReLU:} \quad f(z) = \max(0, z) \tag{11}$$

$$\text{Sigmoid:} \quad f(z) = \frac{1}{1 + e^{-z}} \tag{12}$$

$$\text{Softmax:} \quad f(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{13}$$

- **dataloader.py**
  - **Purpose:** Manages the generation and loading od data, including batching and shuffling of datasets.
  - **Function:** Returns the `dataloader` class, designed to load the data efficiently while training the model.
  - **Key Methods:** `init`, `iter`, `len`, `next`.
- **layer.py**

- **Purpose:** Defines the `layer` class, which represents individual layers in the neural network.
- **Functionality:** Manages weight and bias initialization and applies activation functions.
- **Key Components:** Initialization of layers, forward pass, activation function application.

- **losses.py**
  - **Purpose:** Defines and returns the loss functions used to compute the error and optimize the model during training
  - **Functions Provided:** Mean Squared Error (MSE), Cross Entropy Loss.
  - **Key Equations:**

$$\text{MSE:} \quad L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{14}$$

$$\text{Cross Entropy Loss:} \quad L = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) \tag{15}$$

- **net.py**
  - **Purpose:** Core module that combines the layers along with the activation and loss functions to create and build the neural network.
  - **Functionality:** Handles forward propagation, backpropagation and updates weights as required.
  - **Key Methods:** `add`, `forward`, `backward`, `update_gradients`.

- **classification.py**
  - **Purpose:** Implements the neural network for single-label classification tasks.
  - **Functionality:** Configures the network to train it on synthetic data and evaluate performance.

- **regression.py**
  - **Purpose:** Implements the neural network for regression tasks.
  - **Functionality:** Sets the network configuration with appropriate activation and loss functions for predicting continuous values.

Every module has its own importance in constructing and training the neural network.. For example, the `net.py` module builds a neural network by integrating layers created by `layer.py`, applies activation functions from `activations.py`, measures the losses using `losses.py`, and feeds and processes data using `dataloader.py`. The modules (`classification.py` and `regression.py`) demonstrate application of the neural network on classification and regression tasks respectively.

3.2.2 Implementation Details

The implementation of this modular neural network follows several key steps:

1. **Data Preparation**

   - **Data Generation**:
     - Synthetic data is created using the `make_regression` function from the scikit-learn library.
     - The generated dataset consists of **10000 samples** with **10 features**, coupled with noise to simulate real-world scenarios.
   - **Data Splitting**:
     - The generated dataset is split into training and validation sets using the `train_test_split` function.
     - The split is performed such that 80% of the data is allocated for training (`X_train`, `y_train`), and the remaining 20% is reserved for validation (`X_val`, `y_val`).
   - **Data Loading**:
     - Data loaders are created using the `DataLoader` class from PyTorch.
     - These loaders enable efficient loading of data in batches during model training and evaluation.

The DataLoader(`dataloader.py`)

We defined a `DataLoader` class in Python, to facilitate batch loading of data during the training of neural networks. This class is highly useful when dealing with large datasets which must be processed in smaller, manageable chunks. The major components and their functionalities of the `dataloader` class are described below.

Initialization

The `DataLoader` class is initialized with the following parameters:

- `X`: The input data to the neural network.
- `y`: The labels corresponding to the input data `X`.
- `batch_size`: Defines the number of data samples per batch.
- `drop_last`: Boolean value that specifies whether to drop the last batch or not.
- `shuffle`: Boolean value that indicates whether to shuffle the data or not at the beginning of each training epoch.

Methods

- `__iter__()`: Initializes the iterator.
- `__len__()`: Returns the number of batches. If `drop_last` is true, it returns $\left\lfloor \frac{\text{len}(X)}{\text{batch\_size}} \right\rfloor$. Otherwise, it returns $\left\lceil \frac{\text{len}(X)}{\text{batch\_size}} \right\rceil$.
- `__next__()`: Returns the next batch of data. If the end of the data is reached, it raises a `StopIteration` exception.

The `__len__` method calculates the number of batches based on the batch size and the length of the dataset:

$$\text{num\_batches} = \begin{cases} \left\lfloor \frac{\text{len}(X)}{\text{batch\_size}} \right\rfloor, \textit{if drop\_last is true} \\ \\ \left\lceil \frac{\text{len}(X)}{\text{batch\_size}} \right\rceil, \textit{otherwise} \end{cases} \tag{16}$$

The `__next__` method retrieves the next batch of data for each iteration:

$$\begin{aligned} \text{start} &= \text{batch\_size} \times i \\ \text{end} &= \text{batch\_size} \times (i+1) \end{aligned} \tag{17}$$

If `end` exceeds the length of `X`, it either drops the batch or returns the remaining data based on the `drop_last` parameter.

Main Function

The main function demonstrates how to use the `DataLoader` class:

(a) Randomly generates synthetic data for `X` and `y` using `numpy`.

(b) Initializes a `DataLoader` instance with a batch size of 32.

(c) Iterates over the data in batches, printing the shape of each batch.

2. Model Building

The `layer.py` module defines the network layers. The `activations.py` module provides the activation functions. Loss functions are defined in the `losses.py` module.

The Activation Functions(`activations.py`)

We implemented various activation functions commonly used in neural networks, encapsulated within Python classes. Each class includes methods to compute the activation function, its string representation, and its gradient. This implementation aids in the creation of modular and reusable components for neural network development.

- **ReLU (Rectified Linear Unit):** This activation function is defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{18}$$

Its gradient is:

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{19}$$

- **Sigmoid:** This activation function is given by:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{20}$$

Its gradient is:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{21}$$

- **Linear:** The linear activation function is:

$$\text{Linear}(x) = x \tag{22}$$

Its gradient is:

$$\text{Linear}'(x) = 1 \tag{23}$$

- **Tanh (Hyperbolic Tangent):** The tanh function is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{24}$$

Its gradient is:

$$\tanh'(x) = 1 - (\tanh(x))^2 \tag{25}$$

- **Leaky ReLU:** This activation function introduces a small slope for negative values:

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{26}$$

Its gradient is:

$$\text{Leaky ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases} \tag{27}$$

- **SiLU (Sigmoid-Weighted Linear Unit):** The SiLU function combines the sigmoid and linear functions:

$$\text{SiLU}(x) = x \cdot \sigma(x) \tag{28}$$

Its gradient is:

$$\text{SiLU}'(x) = \sigma(x) + x \cdot \sigma'(x) \tag{29}$$

- **Softmax:** The softmax function is used for multi-class classification:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{30}$$

Its gradient is computed for the entire vector, typically in a vectorized form.

The main function creates an instance of the `LeakyReLU` class with $\alpha = 0$, then applies it to a sample input array `x`. It prints the results of the activation and its gradient.

The Layers(`layer.py`)

We implemented a neural network layer with various activation functions imported from the `activations.py` module, encapsulated within Python classes. This code provides a flexible and modular approach to building neural networks, allowing for easy integration of different activation functions.

Neuron Class

The `Neuron` class represents a single neuron within a neural network layer. Each neuron is initialized with the input dimension and an activation function. This class is responsible for storing the gradients of the neuron's weights and biases and computing these gradients during the forward pass.

- `__init__`: The constructor method initializes the neuron with the following parameters:
  - `dim_in`: The number of input features to the neuron.
  - `activation`: The activation function to be used by the neuron.

  Additionally, the constructor initializes the gradients `dzw`, `dzx`, and `daz` to zero.

$$\texttt{dzw} = 0, \quad \texttt{dzx} = 0, \quad \texttt{daz} = 0 \tag{31}$$

- `get_grads`: This method returns the current values of the gradients as a list:

$$\text{grads} = [\texttt{dzw}, \texttt{dzx}, \texttt{daz}] \tag{32}$$

- `calculate_grad`: This method computes the gradients of the input, weights, and activation with respect to the loss. It takes the following parameters:
  - `x`: The input to the neuron.
  - `z`: The pre-activation output of the neuron.
  - `w`: The weights of the neuron.
  - `index`: The index of the neuron in the layer.

The gradients are computed as follows:

$$\texttt{dzw} = x \tag{33}$$

$$\texttt{dzx} = w[\texttt{index}] \tag{34}$$

$$\texttt{daz} = \text{activation.grad}(z[:, \texttt{index}]) \tag{35}$$

The method returns the gradients as a list:

$$\text{grads} = [\texttt{dzw}, \texttt{dzx}, \texttt{daz}] \tag{36}$$

Layer Class

The `Layer` class defines a single layer in our neural network, consisting of multiple neurons. The class is initialized with the following parameters:

- `dim_in`: The number of input features to the layer.
- `dim_out`: The number of neurons (output units) in the layer.
- `activation`: The activation function to be used in the layer, selected from a predefined dictionary of activation functions.

The `Layer` class initializes the weights and biases of the neurons as follows:

$$W \sim \mathcal{N}(0, 1) \quad \text{and} \quad b \sim \mathcal{N}(0, 1) \tag{37}$$

where W is a matrix of shape `(dim_out, dim_in)` and b is a vector of shape `(dim_out)`.

The `Layer` class also maintains a list of `Neuron` objects, each representing a neuron in the layer.

The `get_grads` method collects and returns the gradients of the weights and biases:

$$\text{grads} = [\text{stack}(\text{dzw}, \text{axis} = 1), \text{stack}(\text{dzx}, \text{axis} = -1), \text{stack}(\text{daz}, \text{axis} = -1)] \tag{38}$$

The `__call__` method performs the forward pass through the layer. It takes an input x of shape `(batch_size, dim_in)` and computes the pre-activation values z:

$$z = xW^T + b \tag{39}$$

The activation function is then applied to z to obtain the output a:

$$a = \text{activation}(z) \tag{40}$$

During the forward pass, the gradients of the activation, weight, and input with respect to the loss are computed and stored for each neuron.

The Loss Functions(`losses.py`)

Loss functions are commonly used to quantify prediction errors, guide model optimization during training and evaluate model performance, tailored to specific machine learning tasks like regression or classification. In this context, each class implements the loss function as well as its gradient with respect to the predicted outputs.

- **Softmax:** The `Softmax` class defines the softmax activation function, which is commonly used in multi-class classification tasks. The softmax function is defined as:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{41}$$

  The `__call__` method computes the softmax of the input vector x, and the `grad` method returns a vector of ones, representing the gradient of the softmax function.

- **Mean Squared Error (MSE) Loss:** The `MSELoss` class implements the mean squared error loss function, which measures the average squared difference between the predicted and true values. The loss function is defined as:

$$L_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_{\text{true},i} - y_{\text{pred},i})^2 \tag{42}$$

  The `__call__` method computes the mean squared error given the true and predicted values, while the `grad` method returns the gradient of the loss function with respect to the predicted values.

- **Binary Cross Entropy (BCE) Loss:** The `BCELoss` class is intended to implement binary cross-entropy loss.

- **Cross Entropy Loss:** The `CrossEntropyLoss` class computes the cross-entropy loss, which is commonly used in classification tasks with multiple classes. The loss function is defined as:

$$L_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{i,j}\log(\hat{y}_{i,j}) \tag{43}$$

  where N is the number of samples, C is the number of classes, $y_{i,j}$ is the true label of sample i for class j, and $\hat{y}_{i,j}$ is the predicted probability of sample i belonging to class j. The `__call__` method computes the cross-entropy loss given the true labels and predicted probabilities, while the `grad` method returns the gradient of the loss function with respect to the predicted probabilities.

The Net module(net.py)

The `Model` class represents a neural network model with various methods for adding layers, performing forward and backward passes, and updating gradients. Below is a detailed description of its components and methods.

Initialization

The `__init__` method initializes the attributes of the `Model` class. It sets up the loss function, learning rate, number of epochs, verbosity level, and other necessary parameters.

- `loss_fxn`: The loss function used for training.
- `logger`: An optional logger for tracking training progress.
- `lr`: The learning rate for gradient descent (default is 0.001).
- `type`: The type of model (e.g., 'regression').
- `epochs`: The number of epochs for training (default is 1000).
- `verbose`: A flag for verbosity (default is False).
- `layers`: A list to store the layers of the model.
- `dW`, `dB`: Lists to store gradients of weights and biases.
- `history`: A dictionary to keep track of training and validation loss and accuracy.

String Representation

The `__str__` method provides a string representation of the model by iterating over its layers and calling the `__str__` method of each layer.

Adding Layers

The `add` method is used to add layers to the model. It appends the specified layer to the list of layers in the model.

Forward Pass

The `__call__` method performs the forward pass through the model. It takes input x and passes it through each layer sequentially, returning the final output.

$$\text{output} = f_n(f_{n-1}(\dots f_1(x)\dots)) \tag{44}$$

where f_i represents the i-th layer of the model.

Backward Pass

The `backward` method implements the backward pass through the model for computing gradients. It first calculates the gradient of the loss function (dL_y). Then, it iterates through the layers in reverse order, computing gradients for each layer's parameters

(dzw, dzx, daz). These gradients are then used to update `dW` and `dB`, which are lists containing gradients of weights and biases respectively.

$$\frac{\partial L}{\partial \hat{y}} = dL_y \tag{45}$$

Each layer's gradients (dzw, dzx, daz) are obtained and used to compute the weight and bias gradients (dW and dB).

Updating Gradients

The `update_gradients` method updates the parameters of each layer based on the computed gradients. It iterates through the reversed lists of `dW` and `dB`, updating weights and biases using gradient descent (`self.lr` is the learning rate). Finally, it clears the lists of gradients for the next iteration.

The weight and bias updates are performed as follows:

$$W_i \leftarrow W_i - \text{lr} \cdot dW_i \tag{46}$$

$$b_i \leftarrow b_i - \text{lr} \cdot dB_i \tag{47}$$

where lr is the learning rate, dW_i is the gradient of the weights, and dB_i is the gradient of the biases.

3. Running-code

We run the code-script for implementing both `Regression` and `Classification` the code provides a complete implementation for training the neural network. It includes,

- **Data generation**:
  Synthetic Data generation of **10000** samples with **10 features** using `makeregression` function.
- **Model architecture setup**:
  `model` class defined that encapsulates the structure, supports **adding layers, forward pass computation, backpropagation** and **updating gradients.**
- **Training**:
  `trainer` class defined, calculates `accuracy, R2 score, training loss, val loss.`

  The resulting Neural Network architecture of both the implementations are given below,

`Regression`

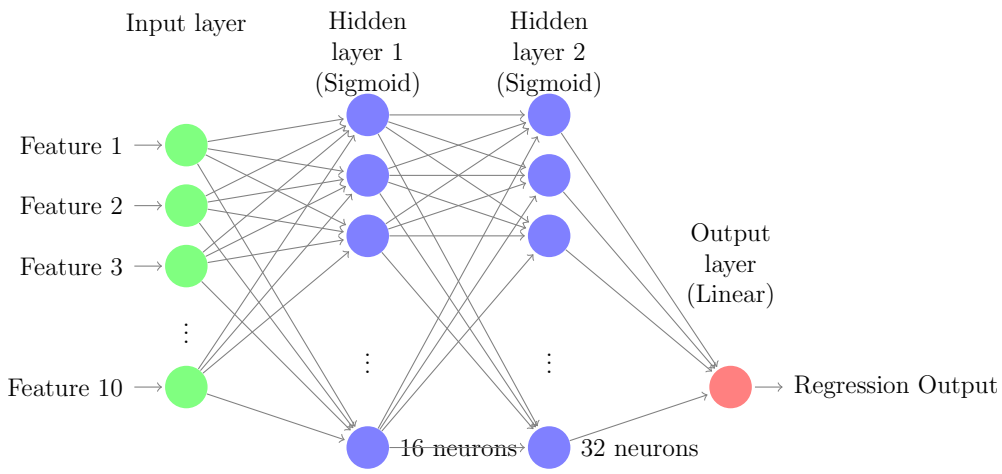

Figure 3: Architecture of the implemented Neural Network: Regression

`classification`

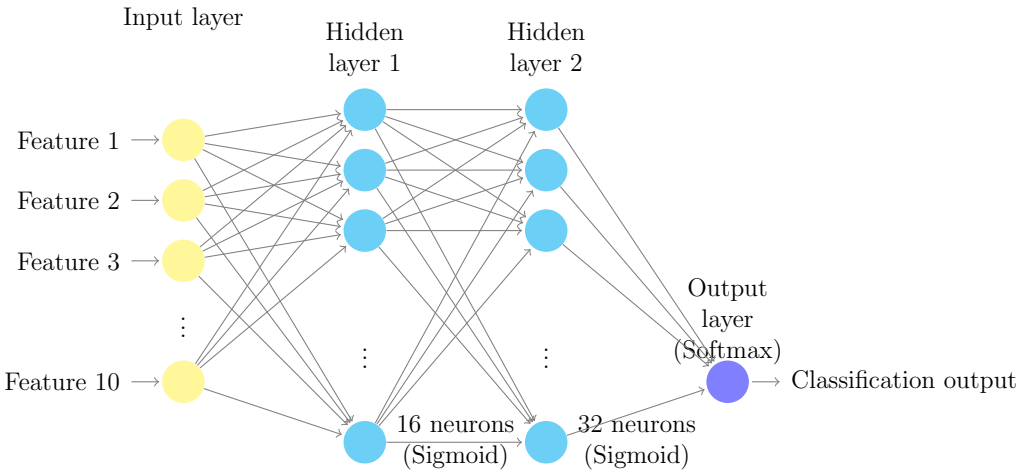

Figure 4: Architecture of the implemented Neural Network: Classification

4. **Evaluation** The final evalutaion was effectively done based on `accuracy scores`, `training` and `validation loss` for both the implementations.

Table 1: Training and Validation Metrics: **Regression**

Epoch	Train Loss	Train Accuracy	Val Loss	Val Accuracy
0	35253.0299	0.0036	30818.6609	0.0777
30	2098.4440	0.9434	1805.7763	0.9461
60	1424.4698	0.9605	1262.7255	0.9611
90	1224.5769	0.9654	1116.0044	0.9652
120	1132.5262	0.9676	1051.9749	0.9669
150	1081.6524	0.9688	1017.4747	0.9678
180	1049.1773	0.9696	995.5270	0.9683
210	1026.2615	0.9702	980.4657	0.9687
240	1009.2592	0.9706	969.5165	0.9690
270	996.2006	0.9709	961.6663	0.9692

Table 2: Training and Validation Metrics: **Classification**

Epoch	Train Loss	Train Accuracy	Val Loss	Val Accuracy
0	1.2292	0.2792	1.1146	0.3212
10	0.8626	0.5885	0.8676	0.5875
20	0.7994	0.6264	0.8126	0.6145
30	0.7429	0.6701	0.7542	0.6615
40	0.6777	0.7121	0.6842	0.7180
50	0.6182	0.7424	0.6238	0.7413
60	0.5749	0.7621	0.5828	0.7569
70	0.5444	0.7774	0.5548	0.7740
80	0.5210	0.7864	0.5334	0.7820
90	0.5019	0.7966	0.5153	0.7908

5. **Results**
   For both implementations, the neural network fares fairly well given the noise added in the data and the non-linear dependencies.

   - Regression:
     `Training accuracy: 97%`
     `Validation accuracy: 96.92%`
   - Classification:
     `Training accuracy: 79.67%`
     `Validation accuracy: 79%`

   Below are the visualized results from training and validation.

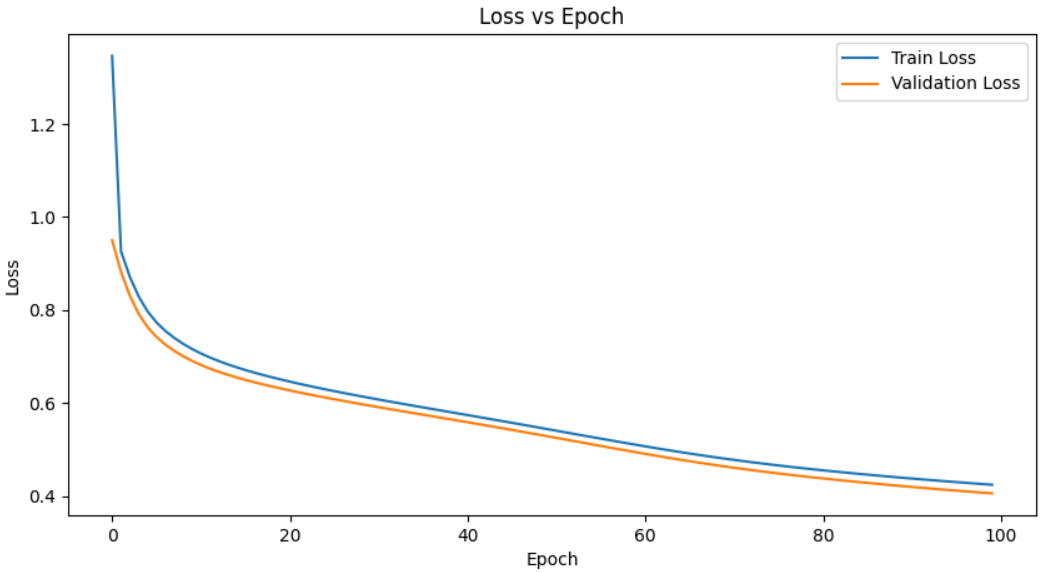

Figure 5: Training and Validation loss: Classification

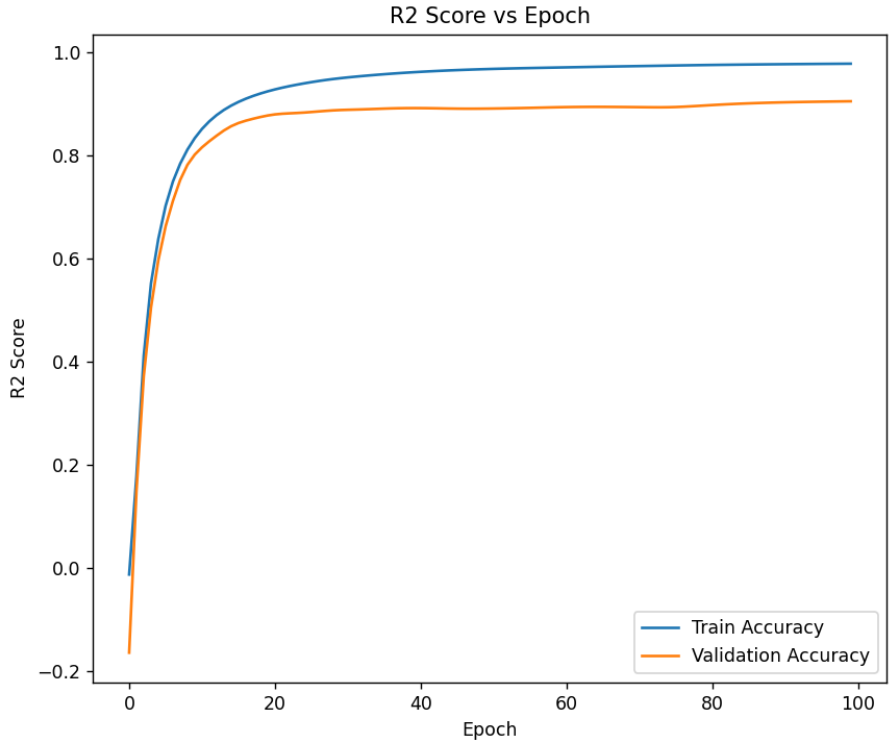

Figure 6: Training and Validation loss: Regression

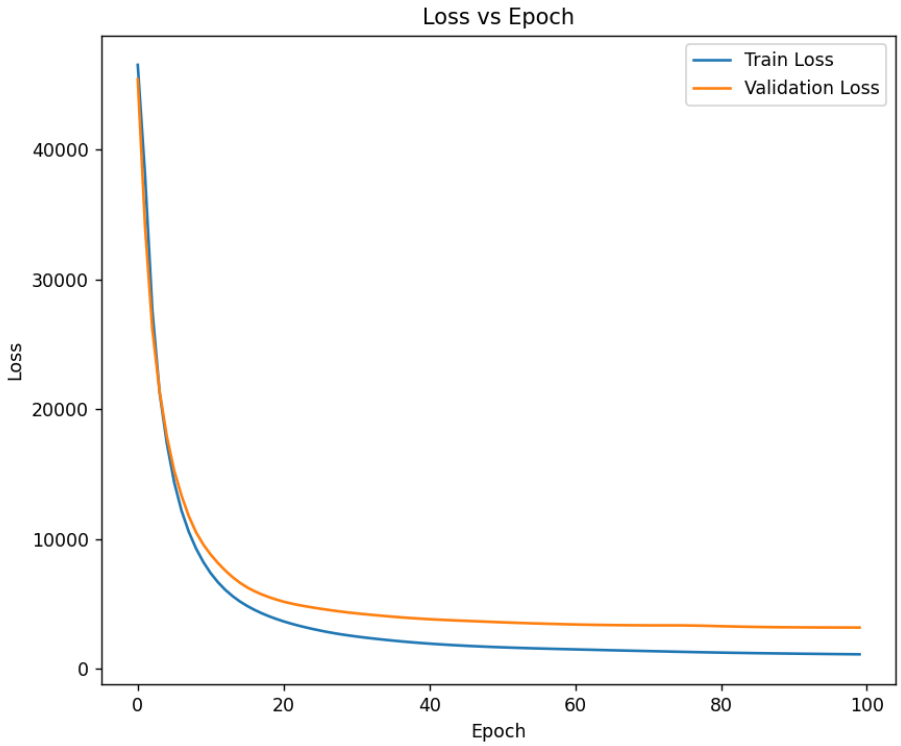

Figure 7: Training and Validation loss: Regression

4. Conclusion

We have implemented a 2-layer Neural Network from scratch using only the `numpy` module to demonstrate, observe and evaluate how, at it's core, a Neural Network evolves over range of provided data, thereby learning the underlying patterns and eventually formulating a mathematical model for the existing variations to perform any user-defined task.
For implementation purpose, we leveraged on synthetic data to better capture the non-linear dependencies and added noise existing in it.
There is optimal scope for further improvements and implementation of more complex, deeper networks which would mitigate all sorts of dependencies of real-world data. However, those algorithms require higher hardware-acceleration to be trained locally over large data.