Implementing Neural Networks using OOP Concepts

1. Introduction

Just like the human brain, neural networks are built up of interconnected assembly of simple processing units called nodes which act like neurons in the brain. It is the style and architecture of brain that is incorporated in neural networks [1]. The nodes act as biological neurons and synapses are depicted as single weight so that each input can be multiplied by a weight. The weighted signals are added together to provide node activation. If the node activation is more than the threshold it gives an output 1 or else it gives output 0.

With the availability of huge training datasets and computing power, artificial neural network is widely used today in several applications from classification of skin cancer, image compression and handwriting recognition. Although current neural network models seem to be common enough to apply them to many applications they are still limited to specific problems and impose requirement on the data that is available and interpretability of the solution. Therefore, in some cases, the output or the solution of a neural network model can be restrictive or might be difficult to understand.

Our project's aim is to overcome this problem. It follows the approach of allagmatic method that programs and executes the model on its own with few limitations while supporting human interpretability. We apply this approach in object-oriented programming to create a metamodel that implements a working dense neural network with variable layer length which makes the models as robust as possible. The input data would guide the selection of a suitable neural network from the metamodel so that possible models would be generated from certain code blocks that are meaningful to humans. Object-oriented programming is selected for the project due to two reasons. First, classes allow the abstract description of structure and operation and objects allow implementing the metastable regime through initialization. Second, they usually provide dynamic and generic types.

2. Motivation

Neural Network is an effective learning algorithm for supervised learning and non-linear statistical modeling. However, in some cases, these models might be restrictive on the workable solutions or their solutions might be difficult to interpret. This sort of problem arises because of the "black box" nature of the neural network, which means when one deals with complex neural networks, feedback neural networks while the network can approximate any sort of function, but it won't provide any insights whatsoever on the structure of function being approximated or about the learnt function from the parameters and therefore are not humanly interpretable. The non-linearity in the activation function and ample number of decisive parameters are one of the reasons for the above problem.

Here we are using concepts of object-oriented programming in determining the relations between the various functionalities of a neural network, creating classes, objects, class diagrams to best fits the relations on OOP's and thus implementing and the obtained model on various datasets.

The goal is to implement a working dense neural network with variable layer length using the object-oriented programming paradigm while supporting human interpretability and making it more robust at the core.

3. Objective

Implement a working dense neural network with variable layer length using the object-oriented programming paradigm to make the models as robust as possible.

4. Methodology

The entire project has been divided into four parts, shown in fig 2. The middle part containing Helper functions and Class schema are used to make the neural network. Here usage of object-oriented programing helps designing a better and optimal classes for neural network integration. The other modules help in represent the output more interpretable. These uses various common and standard statistical techniques for analyzing and representation of the provided input. In case meta model module the input would clean train data. For Representation module, it would be final network after training.

Note: - Some Specifications and Assumptions used in this project need to be acknowledged.

Specifications:

- Our model is for creation of dense neural network. Works with specific hyperparameters.
- The intermediate layers use only ReLU activation.
- The final layer will use sigmoid in case of Classification and linear in case of Regression

Assumptions:

- The data provided to the model is clean and ready for training.

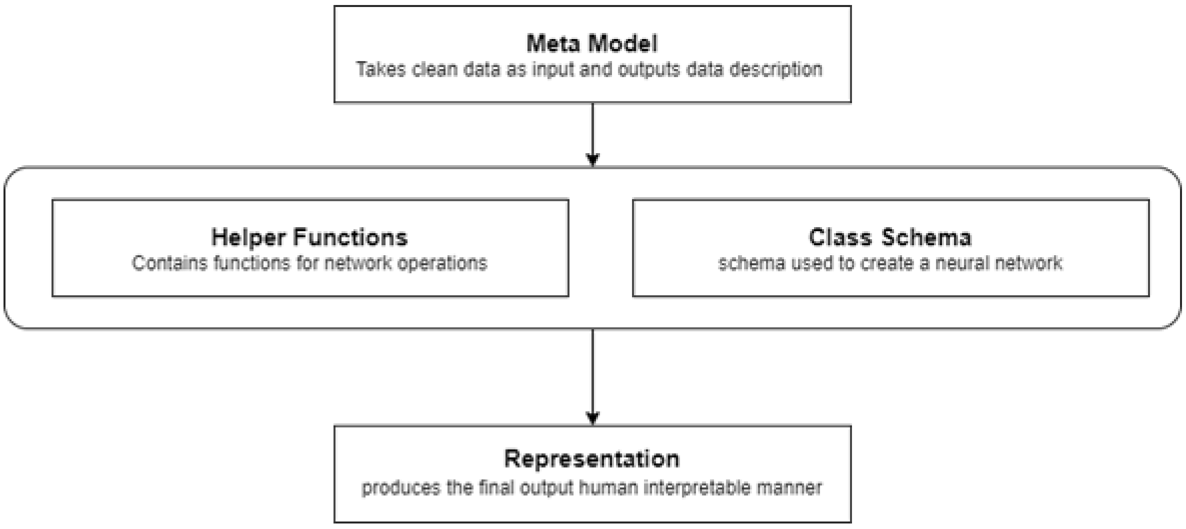
Meta Model
Takes clean data as input and outputs data description
Helper Functions
Contains functions for network operations
Class Schema
schema used to create a neural network
Representation
produces the final output human interpretable manner

fig 1: The General Methodology of the Project

5. Modularity

5.1. Meta Model Module

a. *Description*

- *Fetching the preprocessed data..*
- Performing statistical and graphical analysis to visualize the preprocessed data.

b. *Data Flow Diagram*

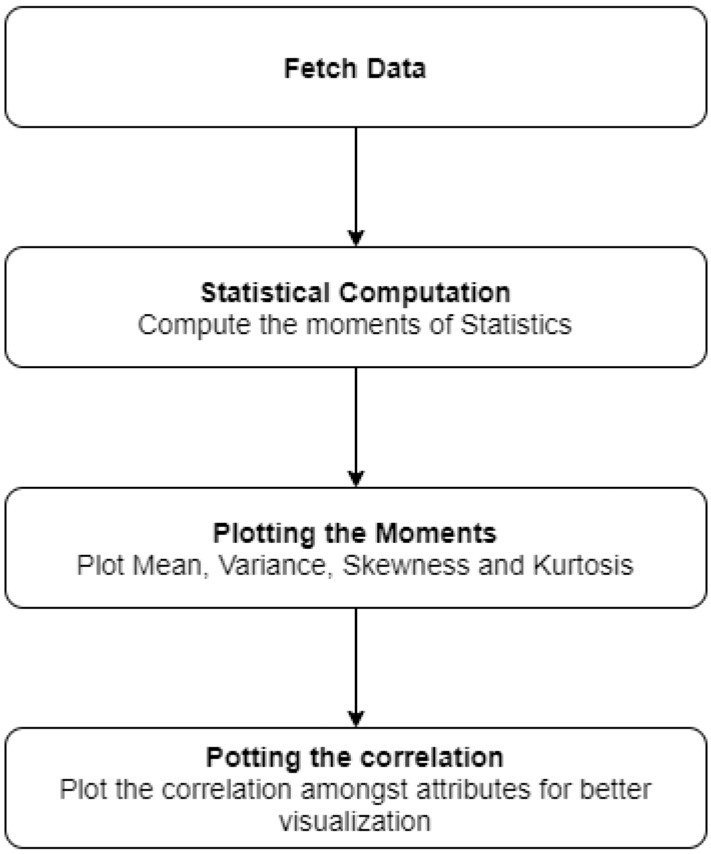
Fetch Data
Statistical Computation
Compute the moments of Statistics
Plotting the Moments
Plot Mean, Variance, Skewness and Kurtosis
Potting the correlation
Plot the correlation amongst attributes for better visualization

Fig 2: Meta Model Module Data Flow Diagram

c. *Algorithm*

Step 1: Start

Step 2: Fetch the input data.

Step 3: Compute the 4 moments of statistics namely Mean, Variance, Skewness and Kurtosis.

Step 4: Plot a graph for each moment for better representation.

Step 5: Plot a stacked bar graph depicting the relationship amongst the various attributes in data.

Step 6: Stop.

5.2. Helper Function Module

a. *Description*

- In this module we create two classes for helper functions (Cost and Activation)

- It applies several functions to make the input data simpler for the Neural Network.

b. *Data Flow Diagram*

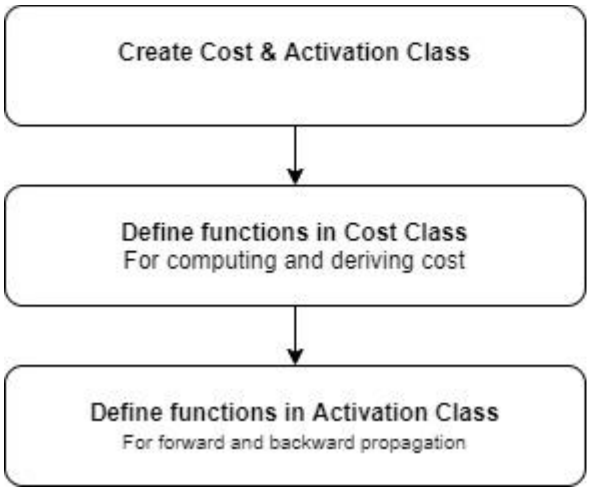
Create Cost & Activation Class
Define functions in Cost Class
For computing and deriving cost
Define functions in Activation Class
For forward and backward propagation

Fig 3: Helper Function Module Data Flow Diagram

c. *Algorithm*

Step 1: Start

Step 2: Create Cost and Activation class

Step 3: Define function to compute and derive the cost in Cost Class

Step 4: Define functions to apply ReLU, sigmoid on the input data for forward propagation in activation class

Step 5: Define functions ReLU backward and sigmoid backward for backward propagation in activation class

Step 6: End

5.3. Class Schema Module

a. *Description*
   - In this module we will create multiple classes for implementing Neural Network.
   - It takes help of various helper functions defined in previous module

b. *Data Flow Diagram*

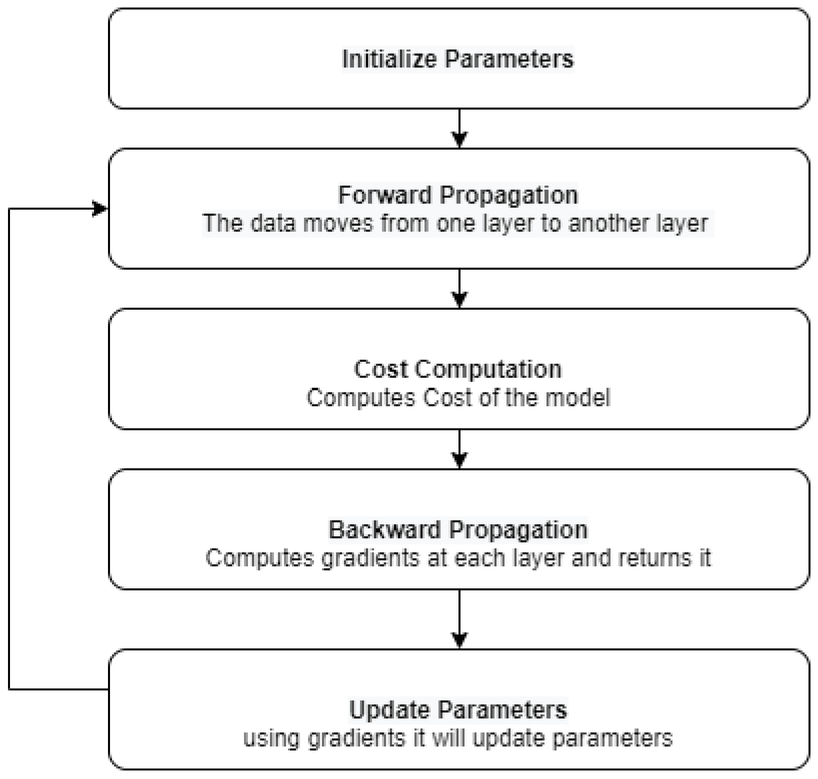
Initialize Parameters
Forward Propagation
The data moves from one layer to another layer
Cost Computation
Computes Cost of the model
Backward Propagation
Computes gradients at each layer and returns it
Update Parameters
using gradients it will update parameters

Fig 4: Class Schema Module Data Flow Diagram

c. *Algorithm*

Step 1: Start

Step 2: Create basic Model class inherited by Classification and Regression.

Step 3: Classification and Regression classes have specialized forward and backward functions

Step 4: Training module with help of specialized functions crate a neural network and initialize the parameters.

Step 5: compute cost and update parameters using backpropagation.

Step 6: Repeat 5 till the number of epochs. Plot training graphs

Step 7: End

5.4. Representation Module

a. *Description*

- In this module, for each hidden layer neuron weights of all the input layer neuron are taken as input.

- For every change in cpoch a graph will be plotted for each and every weight.

b. Data Flow Diagram

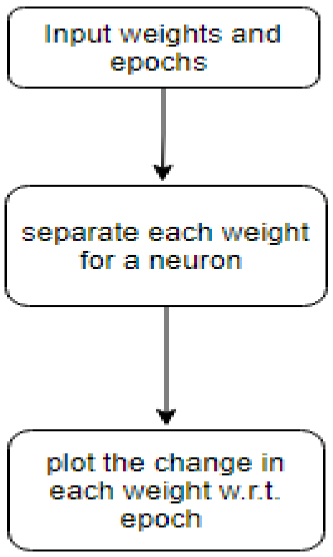
Input weights and epochs
separate each weight for a neuron
plot the change in each weight w.r.t. epoch

Fig 5: Representation Module Data Flow Diagram

c. Algorithm

```
Step 1: Start
Step 2: i = no. of neuron in input layer
        j = no. of neuron in hidden layer
        k = no. of epochs
Step 3: v[i] = [v1, v2, v3,…....., v(i)]
  e[k] = [e1, e2, e3,……..., e(k)]
Step 4: for each value of j
            for each value of i
                Plot v[i][j] v/s e[k]
Step 5: Stop.
```

References

[1] Braspenning, P., 1995. *Artificial neural networks*. Springer.

[2] Patrik Christen and Olivier Del Fabbro, 2020. *Automatic Programming of Cellular Automata and Artificial Neural Networks Guided by Philosophy*. Springer, pages 131–146.

[3] Christen, Patrik & Fabbro, Olivier. (2020). *Adaptation in a System Metamodel for Evolutionary Computation*. [online] Available at: https://www.researchgate.net/publication/344038715_Adaptation_in_a_System_Metamodel_for_Evolutionary_Computation