Istanbul Health and Technology University

Software Engineering Department

SWE211 – Object Oriented Programming

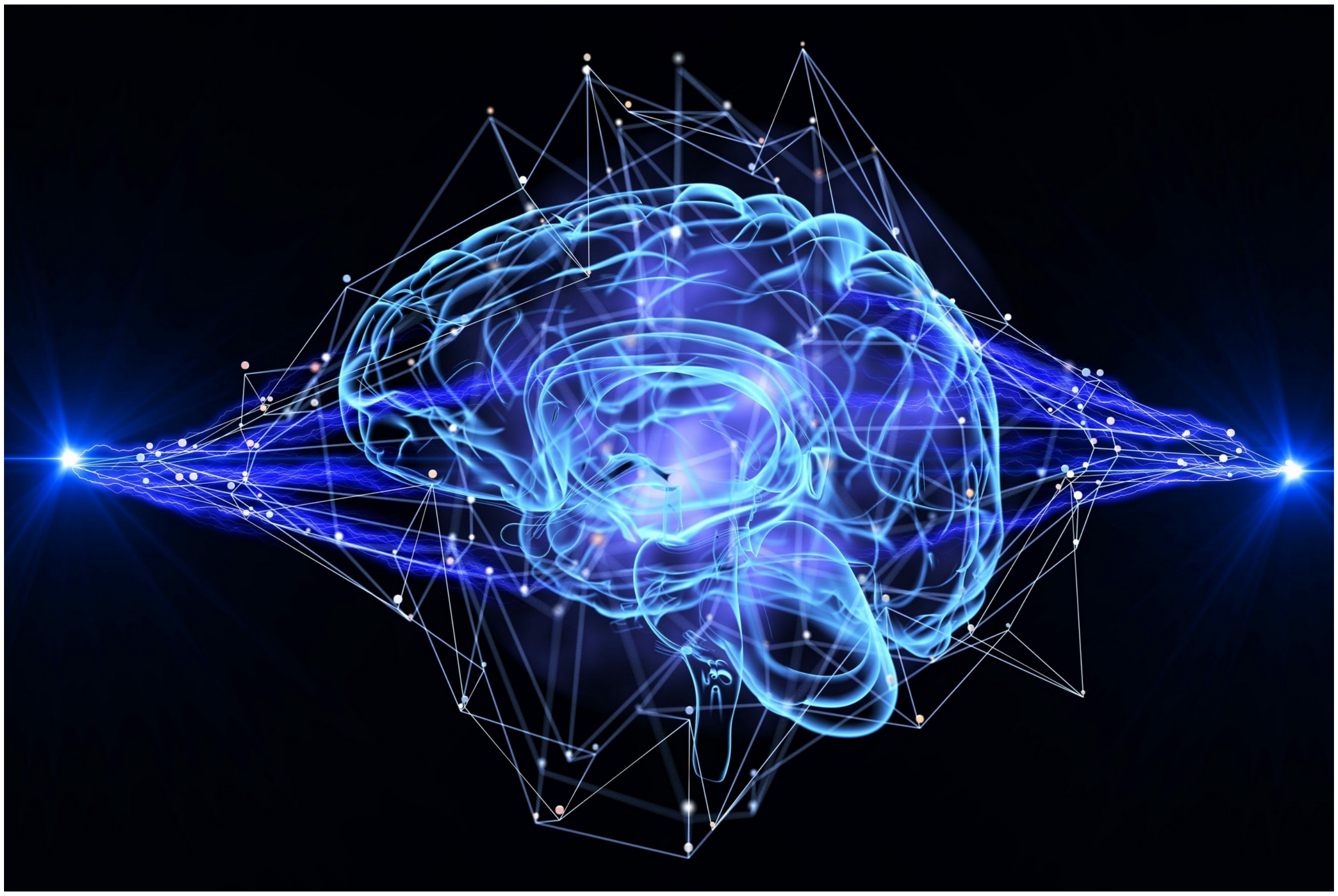
Neural Network Research

Mehmet Akif VARDAR

Introduction

As a mid-term project, I wanted to learn and experience a new area of the software engineering field so I decided to start studying neural networks and eventually build a neural network that can predict the price of an object using the sizes of it.

First things first, let's get started by defining the concept of neural networks.



Introduction to Neural Networks

A neural network is an artificial model inspired by the infrastructure of the human brain. It's a network of connected nodes or so called "neurons" in different layers of the network.

These layers are typically divided into three categories:

1- Input Layer

Input layer receives the data which will be processed by the neural network. It can be anything from images to texts, numbers and frequency signals.

2 – Hidden Layers

These layers are the layers which actually performs mathematical operations to weighted input data and connect output layer with the input layer.

There can be multiple hidden layers in a neural network depending on the problem to be solved.

3 – Output Layer

This is the layer that gives the result generated by the neural network. It generates an output based on the information received from hidden layers.

Neural networks learn by adjusting the weights of the connections between neurons during a process known as “training. This training involves presenting the network with a set of inputs, comparing its output to the desired output and updating the weights accordingly, typically through optimization algorithms like gradient descent.

Let’s continue with the desired problem to be solved using a neural network.

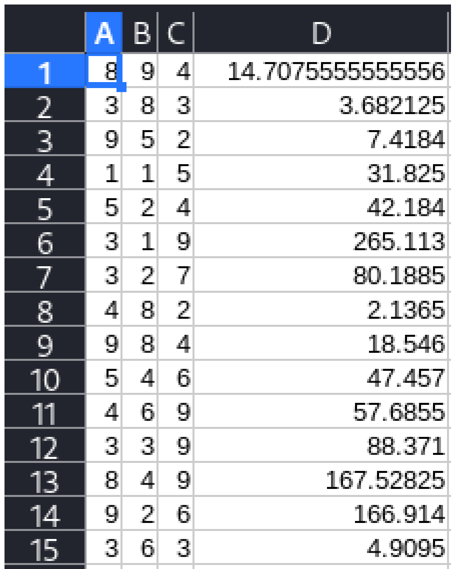
Problem

I wanted to create a neural network which can predict a cube’s price according to its x,y and z sizes. We used the formula below to calculate the true prices of the cube,

$$P = \frac{x_1 + A}{x_2} \times x_3^2$$

Where A is Akif constant, defined as 0.273.

I wrote a Java code to generate a 3000 line dataset with x,y,z sizes and the price calculated using the formula. You can see a part of the dataset from the screenshot below:



	A	B	C	D
1	8	9	4	14.7075555555556
2	3	8	3	3.682125
3	9	5	2	7.4184
4	1	1	5	31.825
5	5	2	4	42.184
6	3	1	9	265.113
7	3	2	7	80.1885
8	4	8	2	2.1365
9	9	8	4	18.546
10	5	4	6	47.457
11	4	6	9	57.6855
12	3	3	9	88.371
13	8	4	9	167.52825
14	9	2	6	166.914
15	3	6	3	4.9095

This dataset will be fed into the neural network to train it. After its training, it is expected to predict the prices of a cube given by its size without knowing the actual formula.

After generating our dataset, now it’s time to decide on the structure of the neural network we want to create.

Planning the Neural Network

Planning step when developing a neural network is very important. That’s why I had to think very hard on how should my neural network be.

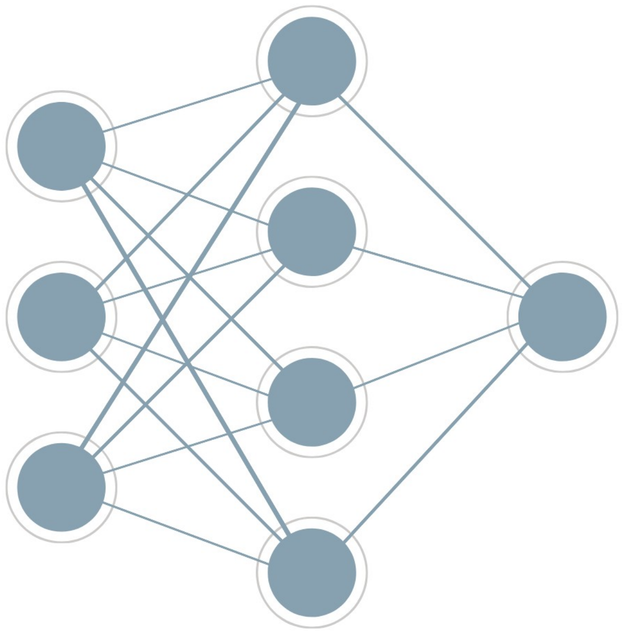
I needed 3 input neurons because 3 input values (x,y and z) will be fed into the neural network.

After deciding to input layer, I decided that 4 hidden layer neurons should’ve helped me achieve my goal. So I decided to create 4 hidden layer neurons contained in 1 hidden layer.

As the output layer, I needed just one prediction, so called “price”. That’s why I decided that 1 output neuron should be enough.

In conclusion, my neural network would include,

- 3 Input Neurons
- 4 Hidden Layer Neurons
- 1 Output Neuron



You can see the structure of our neural network from the image given above.

Crucial Steps in the Neural Network

There are some crucial steps in neural network which affects the flow of the network. Let’s take a look at them:

1 – Initialization

First things first, we need to initialize the weights of each neurons connection to next layer of the neural network. This step sets the starting values for the parameters that the network will adjust training.

We have 3 neurons in input layer and each neuron has 4 connections to the hidden layer. So, we need to initialize 4 different weight values to each neuron in the input layer.

Same applies to hidden layer neurons as well. They all have only one connection to the output layer, thus we need to initialize one different weight value to each of the neuron in the hidden layer.

Initial weight values are given to the network as a starting value as mentioned earlier. During the training process, the network will adjust those values to eventually start guessing right outcomes.

In my neural network, I decided to give values between -1 and 1 as the initial weight values.

2 – Forward Propagation

After initializing the initial values of the neural network, we fed the input data to the network, which is called forward propagation. The input we fed into the network will flow forward through all layers and each layer will apply mathematical operations to the input. Let’s take a look at those mathematical operations:

2.1 – Calculating the Weighted Sum:

- Each input value gets multiplied by the weight of the connection using to connect the neuron to another neuron in hidden layer. This means that all 3 inputs will be multiplied by 12 different weight values and they will be added 3 by 3 to received by 4 hidden layer neurons.
- Each hidden layer neuron will perform an activation function to the received summed and weighted input.

Activation functions serve a crucial role in neural networks for introducing non-linearities into the network’s computations. Their primary purposes is to determine wheter a neuron should be activated or not based on the input it receives.

It processes the weighted sum of inputs and decides whether the neuron should be ‘fired’ to transmit information to the next layer of the network.

Some known activation functions which are actively used in training neural networks are,

- ReLU (Rectified Linear Unit) – We will be using this activation function.
- Leaky ReLU
- ELU (Exponential Linear Unit)
- Sigmoid Function

and so on.

After hidden layer neurons runs activation functions on weighted and summed input data, they then transmit the data to the output layer by multiplying with the weight value which connects them to the output neuron.

Output neuron receives the final input and run an activation function to finalize the prediction. After calculating the final output, forward propagation finishes.

2 – Calculating Loss

After the forward propagation, we compare the prediction with the actual value. The loss function introduced in this step measures the difference between the predicted and actual outputs.

Calculating the difference between the actual output and prediction, helps neural network to adjust its weights in its next prediction attend to eventually decrease the output of the loss function as much as it can.

3 – Back Propagation

This step involves calculating the gradient of the loss function with respect to network’s weights. The gradients are used to update the weights of the network in a way that minimizes the loss.

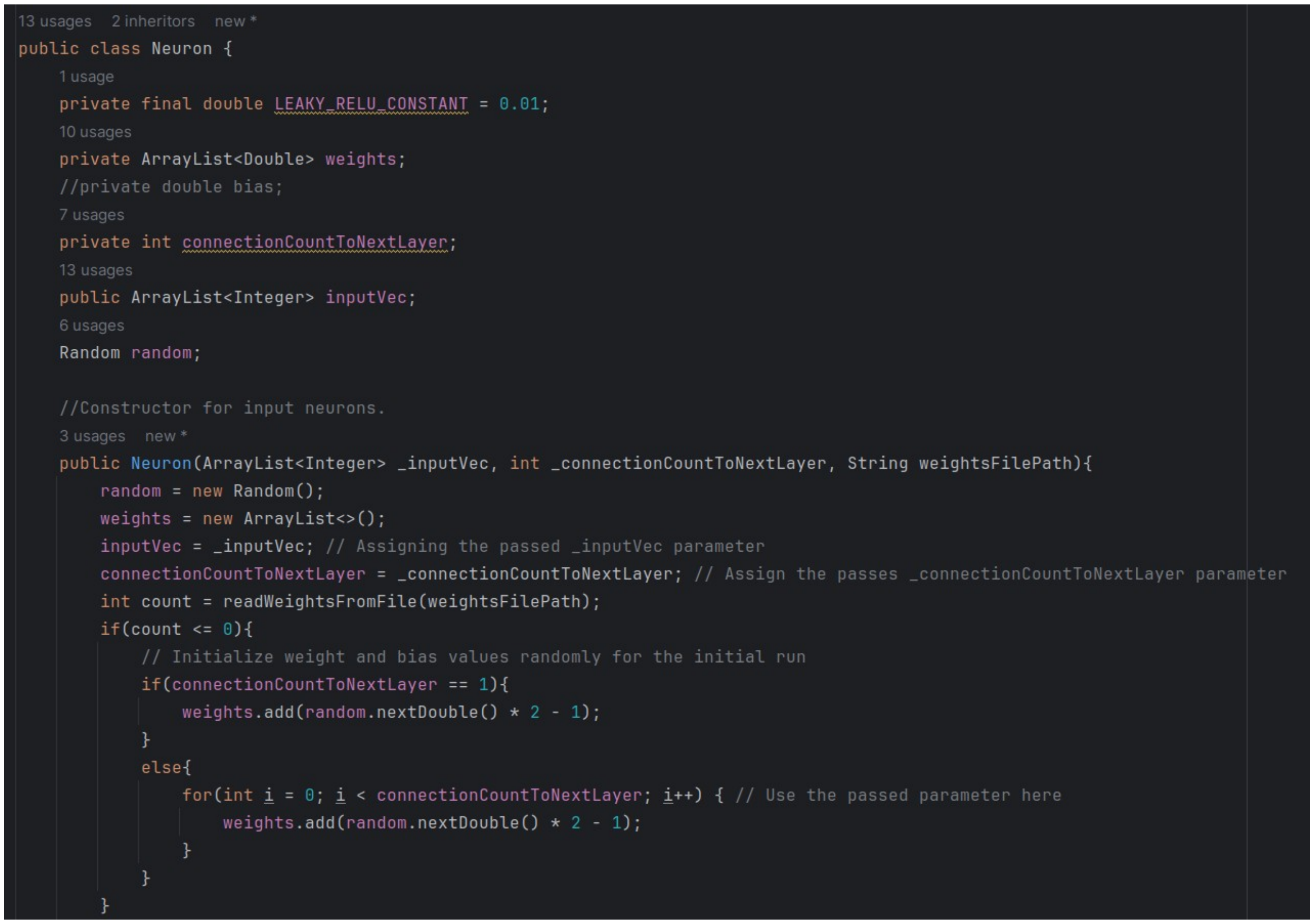
So we basically move from the output to the input layer backwards and adjust all our weights by using an algorithm called gradient descent to eventually find the global minimum of the derivative of the cost function with respect to each weight used in the neural network and update those weights to predict much closer in the next attempt.

We also save the weight values after we updated them to avoid losing our learning progress.

I started my development by creating classes. Here’s my planning:

- Neuron class
  Layer class
- Neural Network class
- Main
- HiddenLayerNeuron & OutputNeuron classes (inherited from Neuron class)
- InputLayer & HiddenLayer & OutputLayer classes (inherited from Layer class)

1- Neuron Class



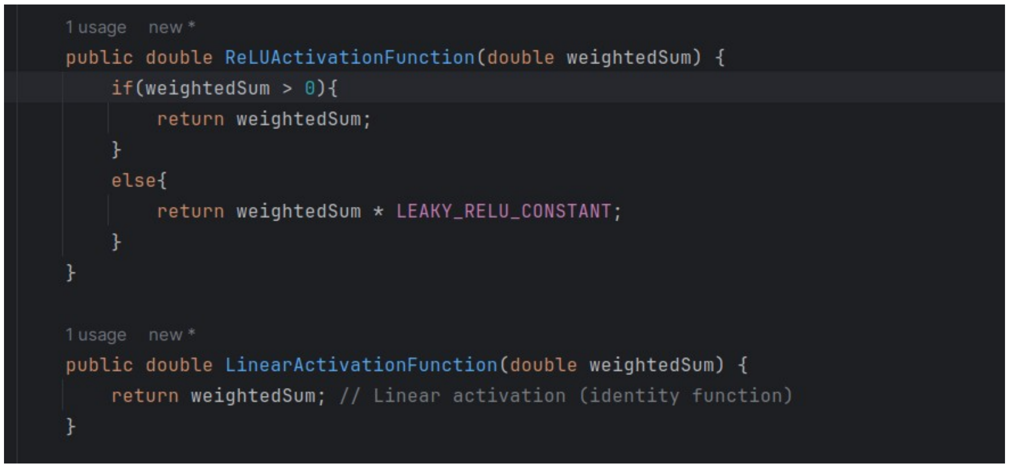
```java
13 usages   2 inheritors   new *
public class Neuron {
    1 usage
    private final double LEAKY_RELU_CONSTANT = 0.01;
    10 usages
    private ArrayList<Double> weights;
    //private double bias;
    7 usages
    private int connectionCountToNextLayer;
    13 usages
    public ArrayList<Integer> inputVec;
    6 usages
    Random random;

    //Constructor for input neurons.
    3 usages   new *
    public Neuron(ArrayList<Integer> _inputVec, int _connectionCountToNextLayer, String weightsFilePath){
        random = new Random();
        weights = new ArrayList<>();
        inputVec = _inputVec; // Assigning the passed _inputVec parameter
        connectionCountToNextLayer = _connectionCountToNextLayer; // Assign the passes _connectionCountToNextLayer parameter
        int count = readWeightsFromFile(weightsFilePath);
        if(count <= 0){
            // Initialize weight and bias values randomly for the initial run
            if(connectionCountToNextLayer == 1){
                weights.add(random.nextDouble() * 2 - 1);
            }
            else{
                for(int i = 0; i < connectionCountToNextLayer; i++) { // Use the passed parameter here
                    weights.add(random.nextDouble() * 2 - 1);
                }
            }
        }
```

We start by defining base properties of the Neuron class and define the constructor to receive all those values from outside.

Some important properties here includes “weights”, “connectionCountToNextLayer”.

We also defined all get set functions to receive and initialize weights when needed.

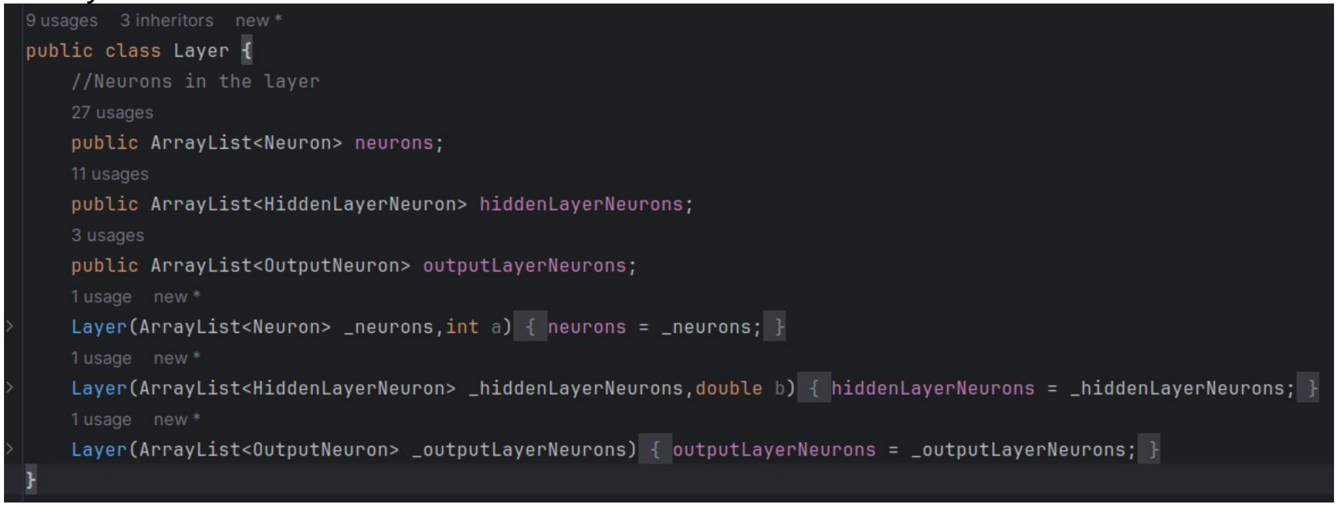


```
1 usage   new *
public double ReLUActivationFunction(double weightedSum) {
    if(weightedSum > 0){
        return weightedSum;
    }
    else{
        return weightedSum * LEAKY_RELU_CONSTANT;
    }
}

1 usage   new *
public double LinearActivationFunction(double weightedSum) {
    return weightedSum; // Linear activation (identity function)
}
```

You can see that we defined our activation functions in Neuron class as mentioned earlier.

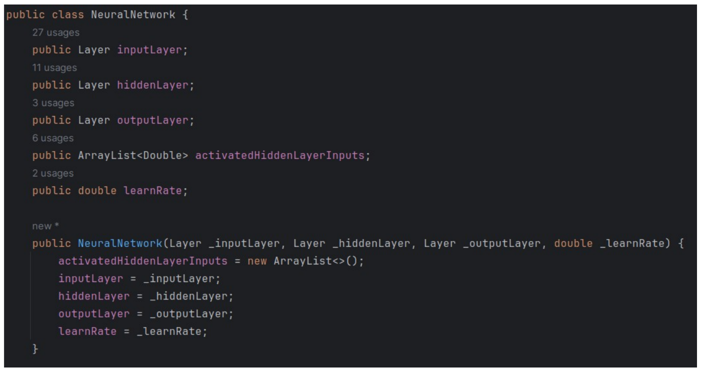
2 - Layer Class



```
9 usages   3 inheritors   new *
public class Layer {
    //Neurons in the layer
    27 usages
    public ArrayList<Neuron> neurons;
    11 usages
    public ArrayList<HiddenLayerNeuron> hiddenLayerNeurons;
    3 usages
    public ArrayList<OutputNeuron> outputLayerNeurons;
    1 usage   new *
    Layer(ArrayList<Neuron> _neurons,int a) { neurons = _neurons; }
    1 usage   new *
    Layer(ArrayList<HiddenLayerNeuron> _hiddenLayerNeurons,double b) { hiddenLayerNeurons = _hiddenLayerNeurons; }
    1 usage   new *
    Layer(ArrayList<OutputNeuron> _outputLayerNeurons) { outputLayerNeurons = _outputLayerNeurons; }
}
```

As you can see, we have defined our Layer function with several constructors so we can instantiate different types of layers.

3 – Neural Network Class

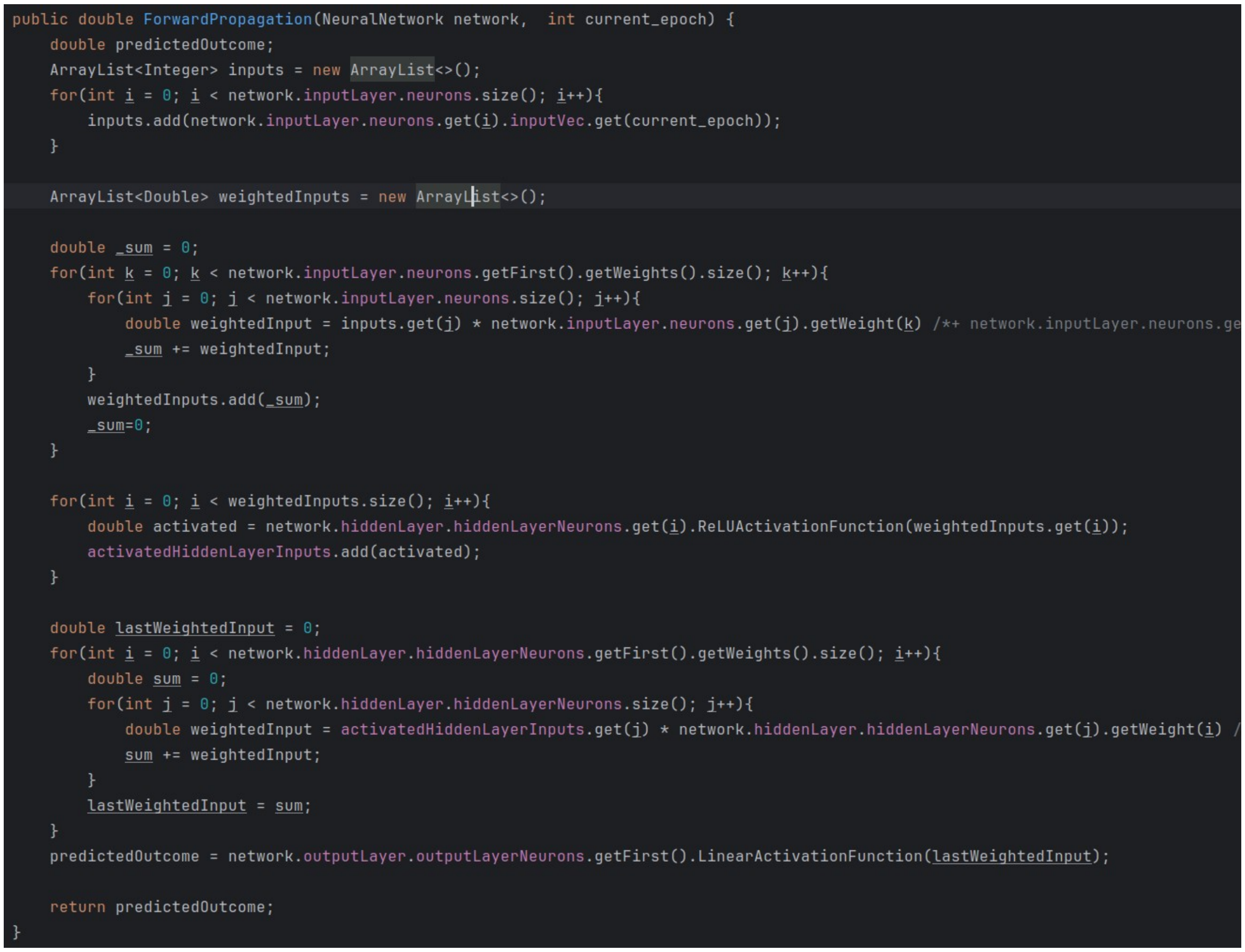


```
public class NeuralNetwork {
    27 usages
    public Layer inputLayer;
    11 usages
    public Layer hiddenLayer;
    3 usages
    public Layer outputLayer;
    6 usages
    public ArrayList<Double> activatedHiddenLayerInputs;
    2 usages
    public double learnRate;

    new *
    public NeuralNetwork(Layer _inputLayer, Layer _hiddenLayer, Layer _outputLayer, double _learnRate) {
        activatedHiddenLayerInputs = new ArrayList<>();
        inputLayer = _inputLayer;
        hiddenLayer = _hiddenLayer;
        outputLayer = _outputLayer;
        learnRate = _learnRate;
    }
```

As you can see we define our main NeuralNetwork class here and initialize all the layers which will be included in our neural network.

In this class, we also defined some functions to handle forward and back propagation processes. Let’s also examine those functions.



```java
public double ForwardPropagation(NeuralNetwork network,  int current_epoch) {
    double predictedOutcome;
    ArrayList<Integer> inputs = new ArrayList<>();
    for(int i = 0; i < network.inputLayer.neurons.size(); i++){
        inputs.add(network.inputLayer.neurons.get(i).inputVec.get(current_epoch));
    }

    ArrayList<Double> weightedInputs = new ArrayList<>();

    double _sum = 0;
    for(int k = 0; k < network.inputLayer.neurons.getFirst().getWeights().size(); k++){
        for(int j = 0; j < network.inputLayer.neurons.size(); j++){
            double weightedInput = inputs.get(j) * network.inputLayer.neurons.get(j).getWeight(k) /** network.inputLayer.neurons.ge
            _sum += weightedInput;
        }
        weightedInputs.add(_sum);
        _sum=0;
    }

    for(int i = 0; i < weightedInputs.size(); i++){
        double activated = network.hiddenLayer.hiddenLayerNeurons.get(i).ReLUActivationFunction(weightedInputs.get(i));
        activatedHiddenLayerInputs.add(activated);
    }

    double lastWeightedInput = 0;
    for(int i = 0; i < network.hiddenLayer.hiddenLayerNeurons.getFirst().getWeights().size(); i++){
        double sum = 0;
        for(int j = 0; j < network.hiddenLayer.hiddenLayerNeurons.size(); j++){
            double weightedInput = activatedHiddenLayerInputs.get(j) * network.hiddenLayer.hiddenLayerNeurons.get(j).getWeight(i) /
            sum += weightedInput;
        }
        lastWeightedInput = sum;
    }
    predictedOutcome = network.outputLayer.outputLayerNeurons.getFirst().LinearActivationFunction(lastWeightedInput);

    return predictedOutcome;
}
```

Here’s the function I used to implement forward propagation algorithm.

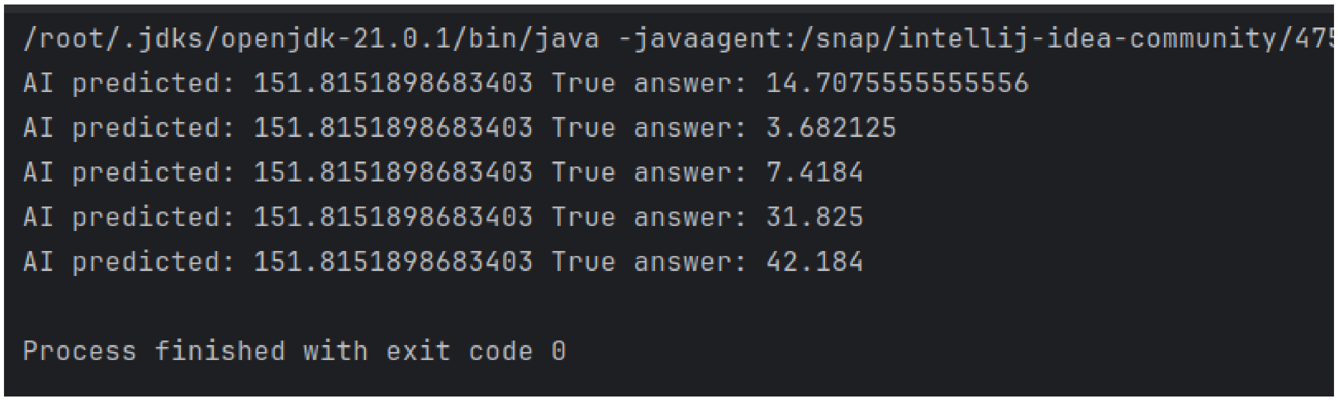
First of all, inputs from dataset are fed into input neurons. Then, all weight values of each neuron are used to calculate first processed data which will be passed to hidden layer.

After passing the data to hidden layer, an activation function (in our case we used ReLU function) used to non-linearize the data and “fire” specific functions based on a treshold.

After calculating activated final data, we use weights of each hidden layer neurons to calculate the data which will be passed to output neuron.

In the last nested for loop, we do just that. And finally, we use an activation function called “LinearActivationFunction” to calculate the final predicted outcome and return it.

Results of our Neural Network



```
/root/.jdks/openjdk-21.0.1/bin/java -javaagent:/snap/intellij-idea-community/47
AI predicted: 151.8151898683403 True answer: 14.7075555555556
AI predicted: 151.8151898683403 True answer: 3.682125
AI predicted: 151.8151898683403 True answer: 7.4184
AI predicted: 151.8151898683403 True answer: 31.825
AI predicted: 151.8151898683403 True answer: 42.184

Process finished with exit code 0
```

After running a few tests and trying various learningRate values and different datasets, it seems like the network isn't generating sensitive data. There could be some possible reasons behind this problem, let's investigate them.

1. **Programming Language Used**

Using Java wasn't really a good idea to develop a neural network. It's because Java is a lot more low level than some other languages which are more popular in Machine Learning field, such as Python.

2. **Deep Mathematics Behind**

Neural Network algorithms requires a solid understanding of Calculus. Because of my lack of experience, I probably made some mistakes during the implementations of algorithms like forward propagation, back propagation or gradient descent algorithm.

Conclusion

In wrapping up, my attempt at creating a neural network using Java didn't pan out due to my limited experience and a shortage of Calculus knowledge. It became clear that Java wasn't the most suitable tool for this specific task. Yet, despite the roadblocks, this journey was far from a failure.

Throughout this process, I delved into the intricate workings of neural networks, gaining insights into their complex inner workings. While I couldn't achieve the intended goal, every obstacle I faced served as a stepping stone for learning. It highlighted the importance of having a solid understanding of math, especially Calculus, in this field.

The setbacks I encountered transformed into invaluable lessons. They not only broadened my understanding of neural networks but also honed my problem-solving skills. This experience was a cornerstone in my learning curve, equipping me with essential knowledge and practical skills that will undoubtedly shape my future endeavors.

In essence, while the intended outcome remained elusive, the wealth of knowledge and the skills acquired throughout this pursuit are immeasurable. This journey underscored the significance of perseverance, adaptability, and continuous learning when navigating the intricate world of emerging technologies.

References

https://www.youtube.com/watch?v=zAPHIAGBjwE – Mahesh Huddar
https://www.youtube.com/watch?v=n2L1J5JYgUk – Mahesh Huddar
https://www.youtube.com/watch?v=tUoUdOdTkRw&t=692s – Mahesh Huddar
https://www.youtube.com/watch?v=khUVIZ3MON8 –Mikael Laine
https://www.youtube.com/watch?v=ayOOMlgb320&t=335s – Simplilearn
https://www.youtube.com/watch?v=hfMk-kjRv4c&t=1916s – Sebastian Lague
https://www.youtube.com/watch?v=w8yWXqWQYmU – Samson Zhang
https://www.youtube.com/watch?v=aircAruvnKk&t=667s – 3Blue1Brown
https://www.youtube.com/watch?v=ZJNklhq1zvg&t=14s – Prototype Project
https://www.youtube.com/watch?v=1DIu7D98dGo&t=26s – Yacine Mahdid
https://www.youtube.com/watch?v=cpYRGNT41Go&t=505s – Applied Coding
https://www.youtube.com/watch?v=XJ7HLz9VYz0&t=82s – The Coding Train
https://www.youtube.com/watch?v=sDv4f4s2SB8 – StatQuest with Josh Starmer
https://www.youtube.com/watch?v=otEGgxlh_EY&t=104s – Mahesh Huddar
https://www.youtube.com/watch?v=ktGm0WCoQOg – Mahesh Huddar
https://chat.openai.com – ChatGPT