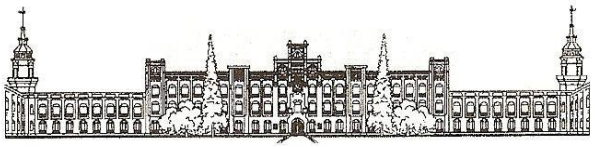

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені Ігоря Сікорського»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Лабораторна робота №3

з дисципліни «Комп'ютерні системи»

«МОДЕЛЮВАННЯ ОБЧИСЛЮВАЛЬНОЇ СИСТЕМИ КЛАСУ ОКМД»

Виконав студент IV курсу

групи: КВ-11

ПІБ: Терентьєв Іван Дмитрович

Перевірив: _____

Київ 2024

Мета лабораторної роботи

Вивчення особливостей функціонування обчислювальних систем класу ОКМД. Моделювання роботи обчислювальної системи класу ОКМД.

Завдання для лабораторної роботи

1 Ознайомитися з описом лабораторної роботи.

2. Отримати варіант завдання у викладача.

3. Написати програму, що моделює функціонування паралельної обчислювальної системи класу ОКМД обраного типу.

Варіант 22(6)

Модель нейронної мережі (без навчання).

Код програми

Neural.cs

```
namespace comp_sys_lab3
{
    public class Neuron
    {
        public List<double> Weights { get; }
        public double Bias { get; }
        public double Output { get; private set; }

        public Neuron(int inputCount)
        {
            var random = new Random();
            Weights = [];

            // Randomly generates weights and bias
            for (int i = 0; i < inputCount; i++)
                Weights.Add((random.NextDouble() * 2) - 1);

            Bias = (random.NextDouble() * 2) - 1 + 0.1;
        }

        public override string ToString() => $"Weights: {string.Join(", ",
Weights)}\nBias: {Bias}";

        public double Activate(List<double> inputs)
        {
            if (inputs.Count != Weights.Count)
                throw new ArgumentException("Inputs count have to be equal to
weights count");

            double sum = 0.0;

            for (int i = 0; i < inputs.Count; i++)
                sum += inputs[i] * Weights[i];

            sum += Bias;
            Output = LeakyReLU(sum);
            return Output;
        }

        private static double LeakyReLU(double x) => x > 0 ? x : 0.01 * x;

    }

    public class Layer
    {
        public List<Neuron> Neurons { get; }
```

```
        public Layer(int neuronCount, int inputsPerNeuron)
        {
            Neurons = [];

            for (int i = 0; i < neuronCount; i++)
                Neurons.Add(new Neuron(inputsPerNeuron));
        }

        public List<double> Activate(List<double> inputs) => (from neuron in
Neurons
                                                               select
neuron.Activate(inputs)).ToList();
    }

    public class NeuralNetwork
    {
        public List<Layer> Layers { get; }
        public int iter = 0;


        public NeuralNetwork(int[] layersStructure)
        {
            Layers = [];

            for (int i = 0; i < layersStructure.Length - 1; i++)
                Layers.Add(new Layer(layersStructure[i + 1],
layersStructure[i]));
        }

        public List<Neuron> GetAllNeurons()
        {
            List<Neuron> allNeurons = [];

            foreach (var layer in Layers)
                allNeurons.AddRange(layer.Neurons);

            return allNeurons;
        }

        public List<List<double>> FeedForwardWithLayerOutputs(List<double>
inputs)
        {
            List<List<double>> outputsPerLayer = [];
            List<double> outputs = inputs;
            iter++;

            foreach (var layer in Layers)
            {
                outputs = layer.Activate(outputs);
                outputsPerLayer.Add(new List<double>(outputs));
            }
```

```
            return outputsPerLayer;
        }

    }
}
```

MainWindow.xaml.cs

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows.Shapes;

namespace comp_sys_lab3
{
    public partial class MainWindow : Window
    {
        private NeuralNetwork network;
        private readonly Random random;
        private readonly List<int> layerStructure;

        public MainWindow()
        {
            InitializeComponent();
            random = new Random();
            layerStructure = [3, 2, 1];
            network = new NeuralNetwork([.. layerStructure]);
            Loaded += OnLoaded;
        }

        private void OnLoaded(object sender, RoutedEventArgs e) =>
DrawNetwork(network.GetAllNeurons());

        private void InitializeNetwork() => network = new NeuralNetwork([..
layerStructure]);

        private void DrawNetwork(List<Neuron> neurons, List<double>? inputs =
null)
        {
            NetworkCanvas.Children.Clear();

            double canvasWidth = NetworkCanvas.ActualWidth;
            double canvasHeight = NetworkCanvas.ActualHeight;
            double layerSpacing = canvasWidth / (layerStructure.Count + 1);
            double nodeRadius = 15;

            int neuronIndex = 0;
            int inputIndex = 0;
            List<Point>? previousLayerPositions = null;
```

```
            for (int i = 0; i < layerStructure.Count; i++)
            {
                int nodeCount = layerStructure[i];
                double nodeSpacing = canvasHeight / (nodeCount + 1);

                List<Point> currentLayerPositions = [];

                for (int j = 0; j < nodeCount; j++)
                {
                    double x = layerSpacing * (i + 1);
                    double y = nodeSpacing * (j + 1);

                    Ellipse node = new()
                    {
                        Width = nodeRadius * 2,
                        Height = nodeRadius * 2,
                        Fill = Brushes.Gray,
                        Stroke = Brushes.Black,
                        StrokeThickness = 2
                    };

                    if (neuronIndex < neurons.Count && previousLayerPositions !=
null)
                    {
                        node.ToolTip = neurons[neuronIndex].ToString();
                        neuronIndex++;
                    }
                    else
                    {
                        if (previousLayerPositions == null)
                        {
                            if (inputs != null)
                            {
                                node.ToolTip = $"Input{inputIndex}:
{inputs[inputIndex]}";
                                inputIndex++;
                            }
                            else
                            {
                                node.ToolTip = "Input empty";
                            }
                        }
                    }

                    Canvas.SetLeft(node, x - nodeRadius);
                    Canvas.SetTop(node, y - nodeRadius);
                    NetworkCanvas.Children.Add(node);

                    currentLayerPositions.Add(new Point(x, y));

                }
```

```
            if (previousLayerPositions != null)
            {
                foreach (var previousPos in previousLayerPositions)
                {
                    foreach (var currentPos in currentLayerPositions)
                    {
                        Line connection = new()
                        {
                            X1 = previousPos.X,
                            Y1 = previousPos.Y,
                            X2 = currentPos.X,
                            Y2 = currentPos.Y,
                            Stroke = Brushes.LightGray,
                            StrokeThickness = 1
                        };
                        NetworkCanvas.Children.Add(connection);
                    }
                }
            }

            previousLayerPositions = currentLayerPositions;
        }
    }

    private void AddLayers(object sender, RoutedEventArgs e)
    {
        for (int i = 0; i < layerStructure.Count; i++)
            layerStructure[i] = Math.Min((layerStructure.Count - i)*3,
layerStructure[i]*2);
        InitializeNetwork();
        DrawNetwork(network.GetAllNeurons(), null);
    }

    private void RemoveLayers(object sender, RoutedEventArgs e)
    {
        for (int i = 0; i < layerStructure.Count; i++)
            layerStructure[i] = Math.Max(layerStructure.Count - i,
layerStructure[i] / 2);
        InitializeNetwork();
        DrawNetwork(network.GetAllNeurons());
    }

    private void SendInputs(object sender, RoutedEventArgs e)
    {
        var inputs = new List<double>();
        for (int i = 0; i < layerStructure[0]; i++)
            inputs.Add(0.1 + (random.NextDouble() * 0.8));

        var outputsPerLayer = network.FeedForwardWithLayerOutputs(inputs);
```

```
            MyTextBox.AppendText($"\nData sent: {network.iter}");
            MyTextBox.AppendText($"\nLayer 0: " + string.Join(", ", inputs));
            for (int layerIndex = 0; layerIndex < outputsPerLayer.Count;
layerIndex++)
                MyTextBox.AppendText($"\nLayer {layerIndex + 1}: " +
string.Join(", ", outputsPerLayer[layerIndex]));
            MyTextBox.ScrollToEnd();

            List<double> allValues = [.. inputs, ..
outputsPerLayer.SelectMany(layer => layer)];
            var myNeurons = FindEllipsesInCanvas(NetworkCanvas);
            int neuronIndex = 0;

            foreach (var input in inputs)
                if (neuronIndex < myNeurons.Count)
                {
                    myNeurons[neuronIndex].Fill = OutputToColor(input, allValues,
0);
                    neuronIndex++;
                }

            for (int layerIndex = 0; layerIndex < outputsPerLayer.Count;
layerIndex++)
                foreach (var output in outputsPerLayer[layerIndex])
                    if (neuronIndex < myNeurons.Count)
                    {
                        myNeurons[neuronIndex].Fill = OutputToColor(output,
allValues, layerIndex + 1);
                        neuronIndex++;
                    }
        }
        private static List<Ellipse> FindEllipsesInCanvas(Canvas canvas)
        {
            List<Ellipse> ellipses = [];

            foreach (var child in canvas.Children)
                if (child is Ellipse ellipse)
                    ellipses.Add(ellipse);

            return ellipses;
        }

        private static SolidColorBrush OutputToColor(double value, List<double>
allValues, int layer)
        {
            double min = allValues.Min();
            double max = allValues.Max();

            max = (max - min == 0) ? min + 1 : max;

            double normalizedValue = (value - min) / (max - min);
```

```
            double layerFactor = 1 + (layer * 0.3);

            normalizedValue = Math.Min(1, Math.Max(0, normalizedValue *
layerFactor));

            byte red = (byte)(normalizedValue * 255);
            byte blue = (byte)((1 - normalizedValue) * 255);

            return new SolidColorBrush(Color.FromRgb(red, 0, blue));
        }


    }
}
```

MainWindow.xaml

```
<Window x:Class="comp_sys_lab3.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Neural Network Visualizer" Height="600" Width="800"
WindowStyle="ToolWindow" ResizeMode="CanMinimize">
    <Grid>
        <Canvas x:Name="NetworkCanvas" Background="#FF3C3C3C"
VerticalAlignment="Stretch" HorizontalAlignment="Stretch" Margin="200,10,10,10"/>
        <StackPanel Orientation="Vertical" Width="180" HorizontalAlignment="Left"
Margin="10,10,10,10">
            <Button Content="Add Layers" Click="AddLayers" Margin="5"/>
            <Button Content="Remove Layers" Click="RemoveLayers" Margin="5"/>
            <Button Content="Send Inputs" Click="SendInputs" Margin="5"/>
            <RichTextBox x:Name="MyTextBox" Height="445"
VerticalScrollBarVisibility="Visible">
                <FlowDocument>
                    <Paragraph>
                        <Run Text=""/>
                    </Paragraph>
                </FlowDocument>
            </RichTextBox>
        </StackPanel>
    </Grid>
</Window>
```

Скріншоти роботи програми

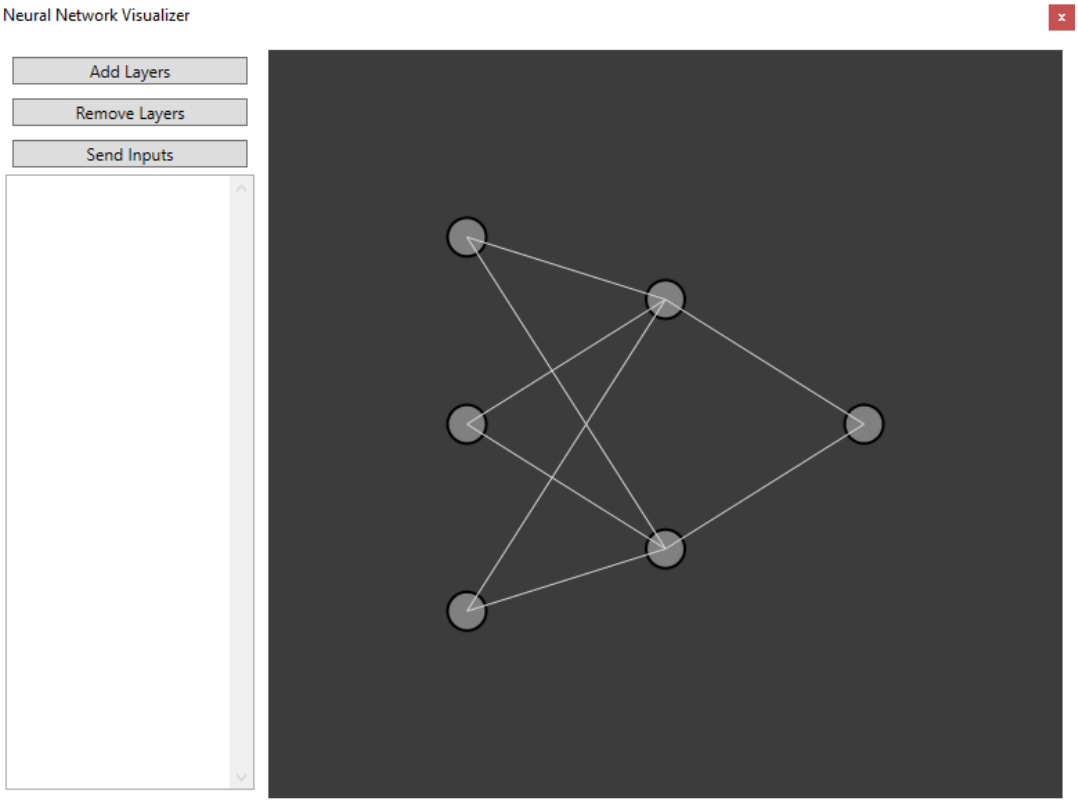

Рис. 1 – Початкове вікно програми

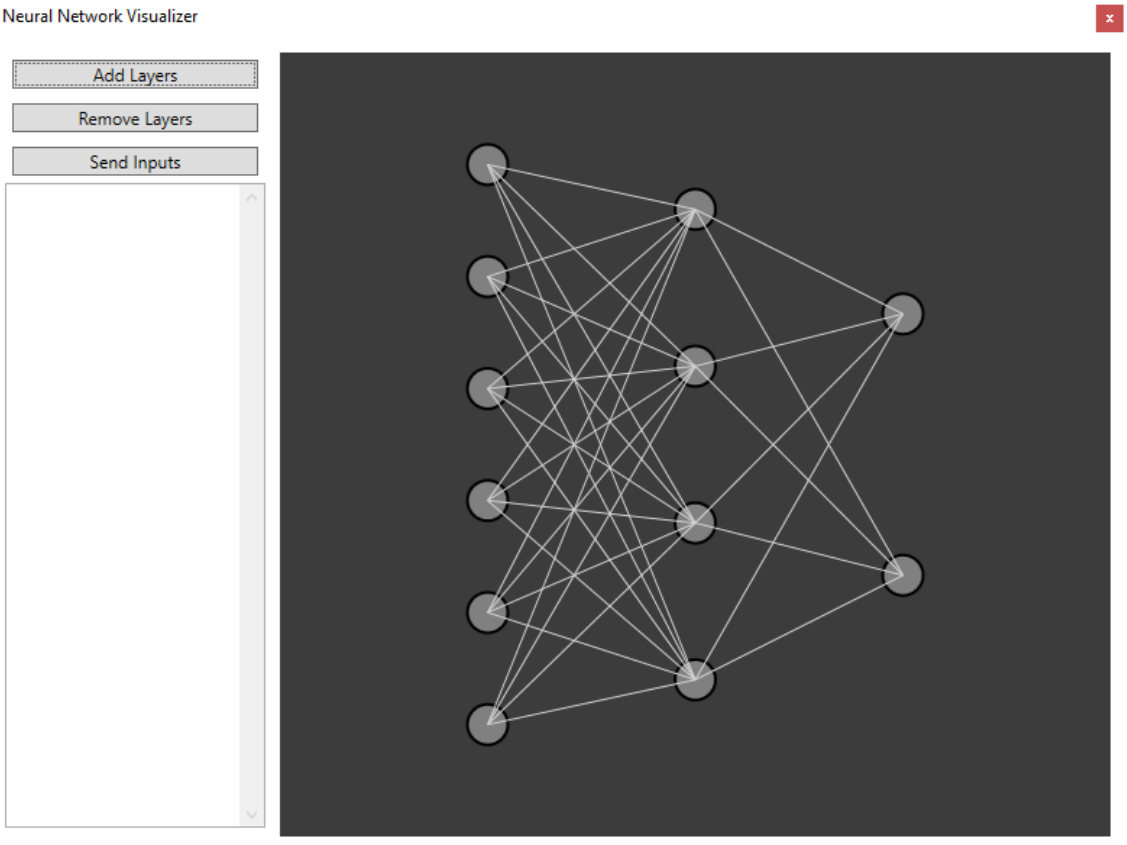

Рис. 2 – Додавання нод

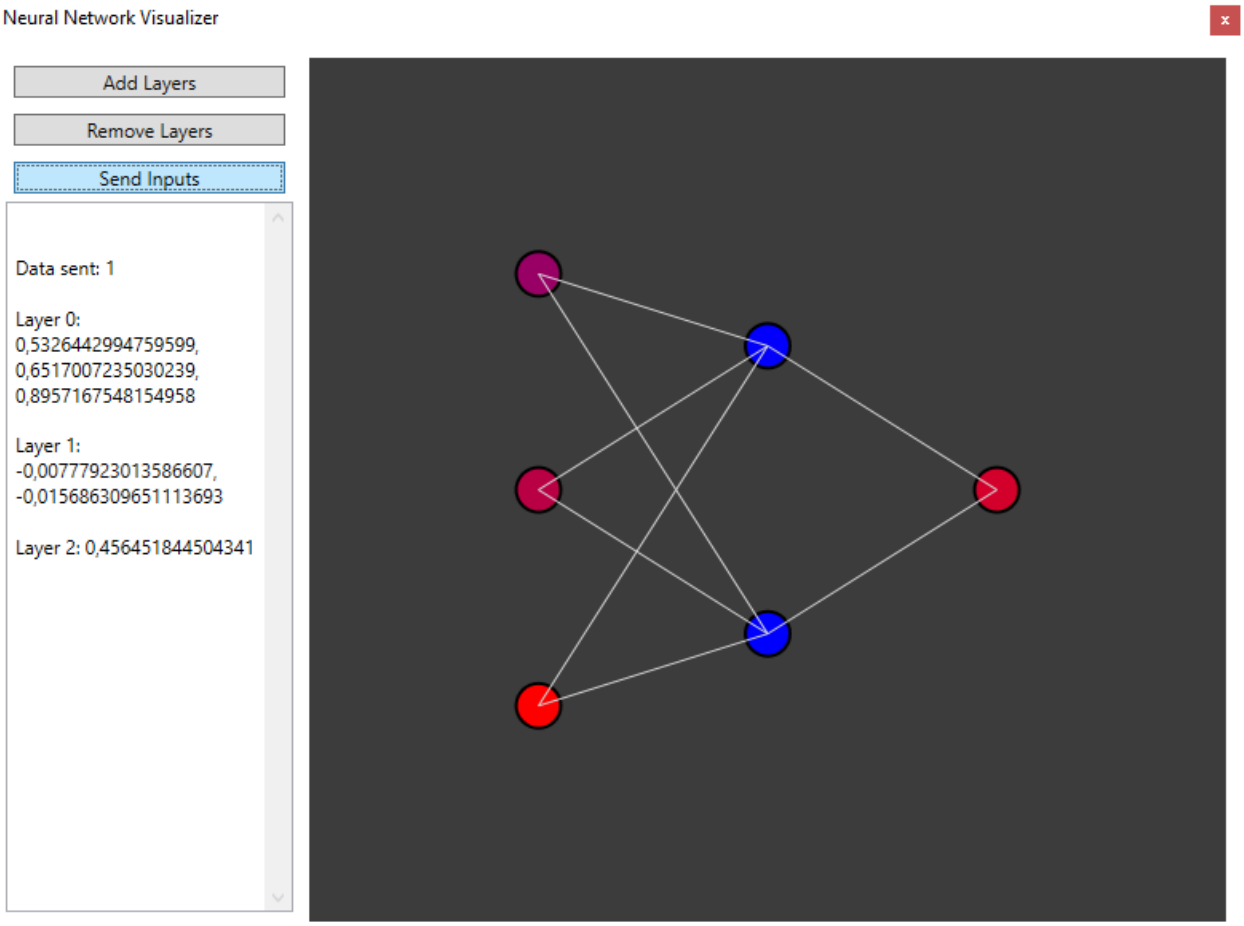

Рис. 3 – Відправка даних

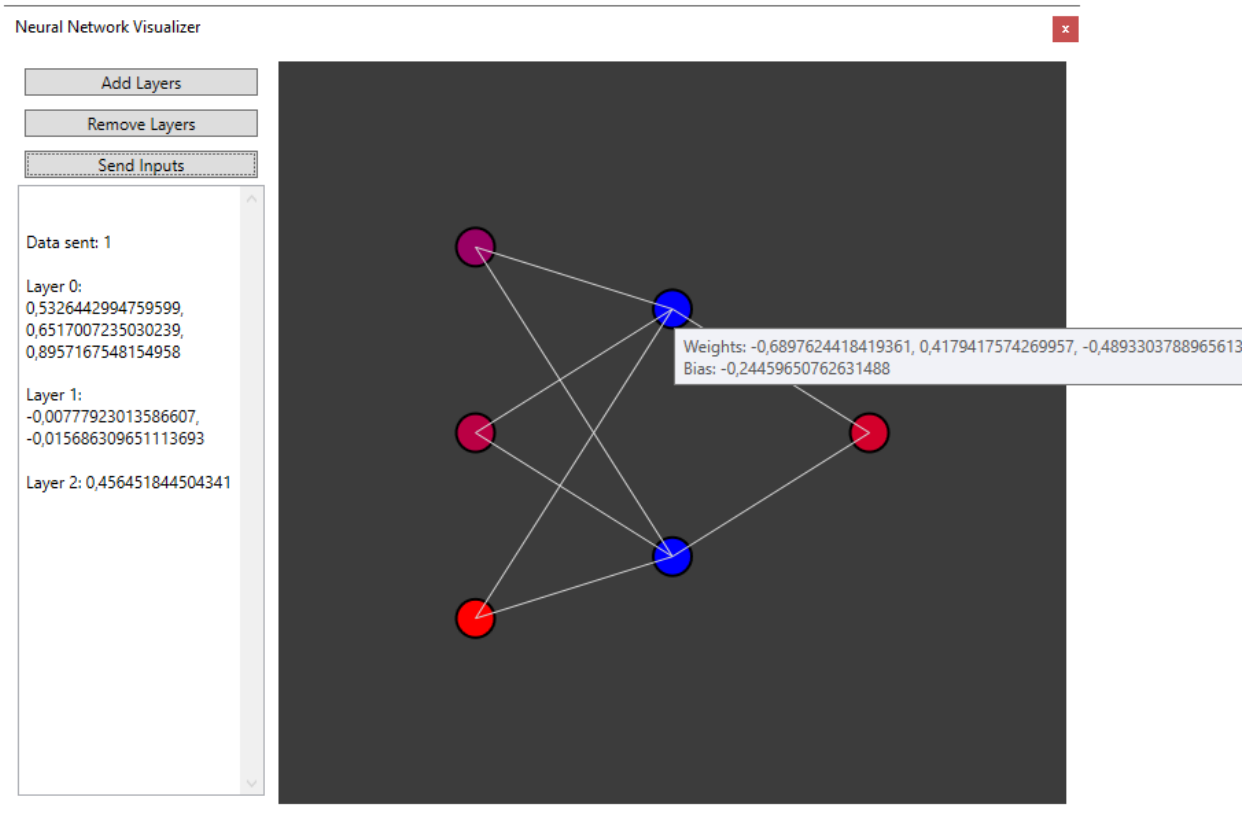

Рис. 4 – Tooltip, вивід ваги та відхилу

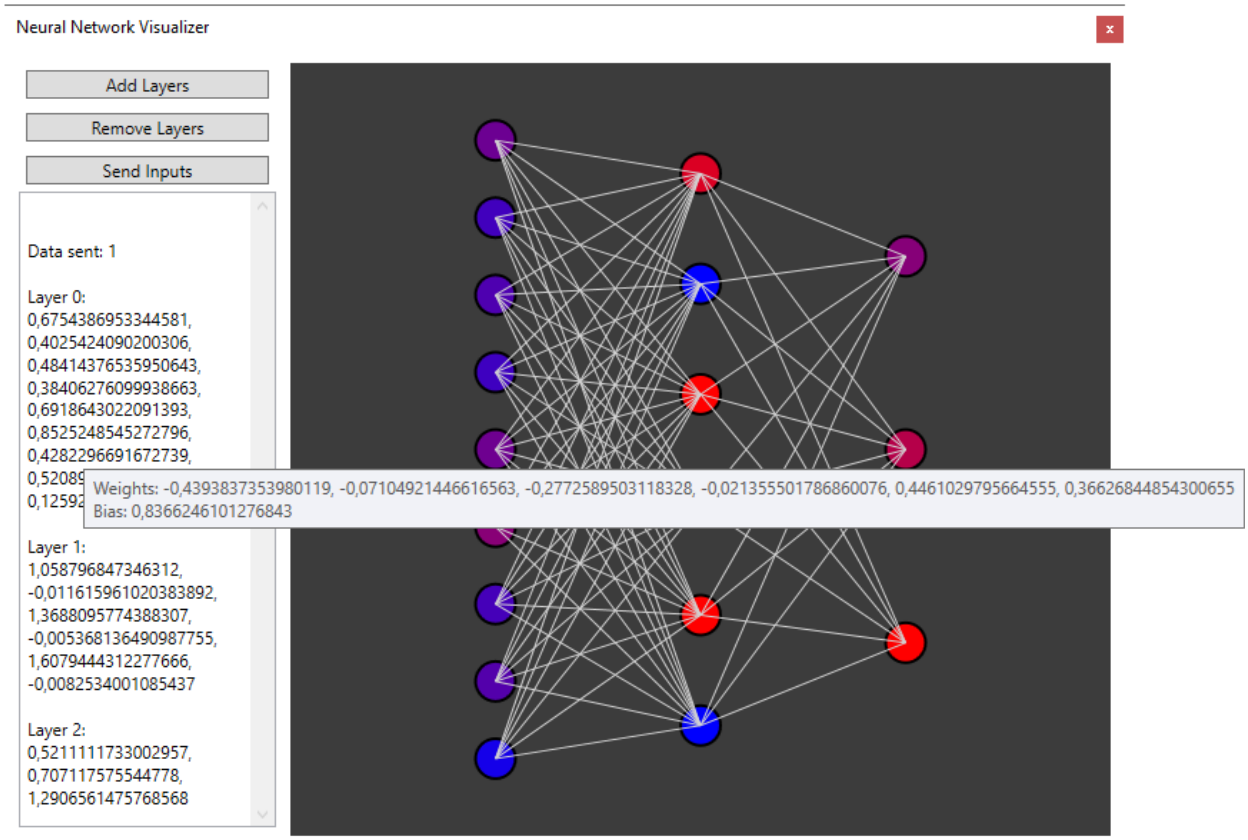

Рис. 5 – Приклад роботи з великою кількістю нод

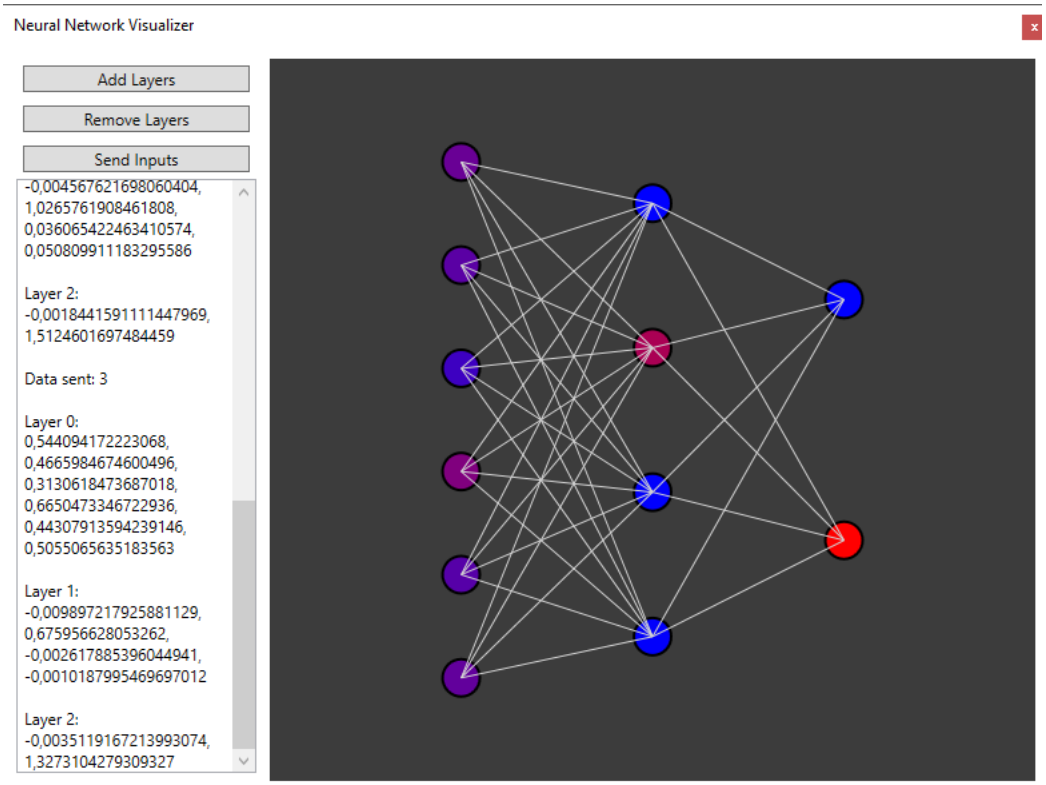

Рис. 6 – Відправка даних декілька разів

Висновки

У цій роботі було реалізовано модель нейронної мережі з використанням функції активації LeakyReLU, яка допомагає уникнути проблеми затухання градієнтів, особливо при роботі з від'ємними значеннями. Це забезпечує стабільний потік інформації через шари мережі, зберігаючи активність нейронів навіть за низьких вхідних значень.

Розроблену програму можна використовувати для візуального аналізу обробки даних у нейронній мережі. Користувач має можливість налаштовувати кількість нейронів у шарах, відправляти нові вхідні дані для обробки та спостерігати за активацією нейронів, що відображається у вигляді зміни кольорів. Підказки (Tooltip) дозволяють переглядати детальну інформацію про ваги та зміщення для кожного нейрона, що робить процес більш наочним та зручним для розуміння. Ваги, зміщення та вхідні значення генеруються випадково, що сприяє демонстрації різноманітних результатів обробки.

У результаті виконання цієї лабораторної роботи ми ознайомилися з основами побудови багатошарових нейронних мереж, особливостями роботи різних функцій активації та методами візуалізації обчислювальних процесів у нейронних структурах.