I. Project Overview

The objective is to classify the type of line in a user-provided 3x3 grid by building a neural network from scratch, without relying on high-level libraries. This project addresses a common challenge students face: despite discussions about neural networks, the underlying logic and flow often remain unclear. The goal is to create a straightforward, easy-to-understand code base that demystifies the neural network's mechanics and helps students grasp its core principles.

II. Data Sources

The dataset used in this project was created independently. It consists of 3x3 grids paired with labels indicating the type of line present (e.g., horizontal, vertical, or diagonal). This custom dataset was manually designed to suit the specific objectives of the project, ensuring simplicity and clarity for understanding the neural network's classification process.

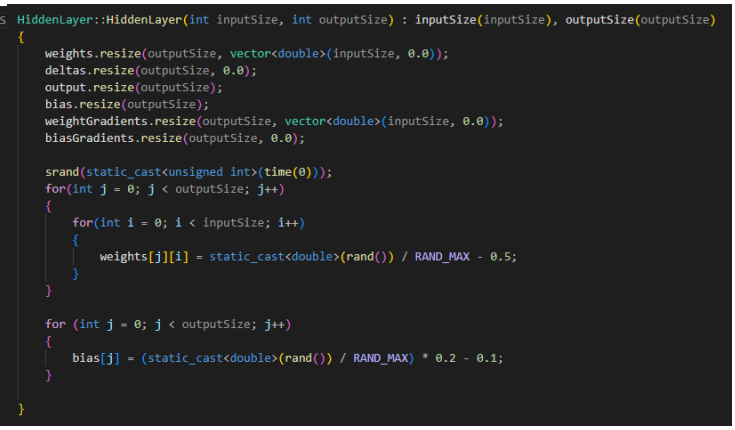
III. Methodology

Neural Network Process:

1. Initialize network with random weights and biases

```
OutputLayer::OutputLayer(int inputSize, int outputSize) : inputSize(inputSize), outputSize(outputS
{
    weights.resize(outputSize, vector<double>(inputSize, 0.0));
    deltas.resize(outputSize, 0.0);
    output.resize(outputSize);
    weightGradients.resize(outputSize, vector<double>(inputSize, 0.0));
    output.resize(outputSize);
    vector<double> deltas(outputSize);
    biasGradients.resize(outputSize, 0.0);
    bias.resize(outputSize);
    logits.resize(outputSize);

    srand(static_cast<unsigned int>(time(0)));
    for(int j = 0; j < outputSize; j++)
    {
        for(int i = 0; i < inputSize; i++)
        {
            weights[j][i] = static_cast<double>(rand()) / RAND_MAX - 0.5;
        }
    }
    for (int j = 0; j < outputSize; j++)
    {
        bias[j] = (static_cast<double>(rand()) / RAND_MAX) * 0.2 - 0.1;
    }
}
```



```
HiddenLayer::HiddenLayer(int inputSize, int outputSize) : inputSize(inputSize), outputSize(outputSize)
{
    weights.resize(outputSize, vector<double>(inputSize, 0.0));
    deltas.resize(outputSize, 0.0);
    output.resize(outputSize);
    bias.resize(outputSize);
    weightGradients.resize(outputSize, vector<double>(inputSize, 0.0));
    biasGradients.resize(outputSize, 0.0);

    srand(static_cast<unsigned int>(time(0)));
    for(int j = 0; j < outputSize; j++)
    {
        for(int i = 0; i < inputSize; i++)
        {
            weights[j][i] = static_cast<double>(rand()) / RAND_MAX - 0.5;
        }
    }

    for (int j = 0; j < outputSize; j++)
    {
        bias[j] = (static_cast<double>(rand()) / RAND_MAX) * 0.2 - 0.1;
    }
}
```

2. For EACH training image :

2a. Compute activations for the entire network (Forward Pass) and activate using sigmoid.

For EACH neuron's activated output per layer:

$$a_j^L = \sigma\left(\left(\sum_{i=0}^{n-1} a_i^{L-1} * w_{j,i}^L\right) + b_j^L\right) \textbf{ where } \sigma(x) = \frac{1}{1+e^{-x}}$$

Why sigmoid?

Because it maps inputs to a range between 0 and 1, making it suitable for classification tasks where outputs represent probabilities.

```
void HiddenLayer::propagateForward(const vector<double>& inputData)
{
    for(int j = 0; j < outputSize; j++)
    {
        output[j] = 0;
        for(int i = 0; i < inputSize; i++)
        {
            output[j] = output[j] + (inputData[i] * weights[j][i]);
        }
        output[j] += bias[j];
        output[j] = sigmoid(output[j]);
    }
}
```

```
void OutputLayer::propagateForward(const vector<double> &inputData)
{
    for(int j = 0; j < outputSize; j++)
    {
        output[j] = 0;
        for(int i = 0; i < inputData.size(); i++)
        {
            output[j] = output[j] + (inputData[i] * weights[j][i]);
        }
        output[j] += bias[j];
        logits[j] = output[j];
        output[j] = sigmoid(output[j]);
    }
}
```

2b. Compute cost to calculate how different the predicted output compared to the true value using Mean Squared Error formula:

$$MSE = \sum_{j=0}^{n-1} \left(\hat{a} - a_j^F\right)^2 \quad Where\ \hat{a} = true\ value, a_j^F = predicted\ value$$

```
double OutputLayer::meanSquaredErrorCostPerImage(const vector<double>& target)
{
    double costPerImage = 0.0;
    for(int i = 0; i < outputSize; i++)
    {
        costPerImage += (pow(target[i] - output[i], 2));
    }
    return costPerImage;
}
```

Why MSE?

Because it penalizes larger errors more heavily by squaring the difference between predicted and actual values, making it effective for guiding the neural network to make precise adjustments during training.

2c. Compute delta**(δ)** for neurons in the output layer using the chain rule in calculus.

$$\frac{\partial C_x}{\partial w_{j,i}^F} = \frac{\partial C_x}{\partial a_j^F} * \frac{\partial a_j^F}{\partial z_j^F} * \frac{\partial z_j^F}{\partial w_{j,i}^F} \quad where\ C_x = cost\ per\ image, z = raw\ output$$

$$which\ is\ \delta_j^F = MSE'\left(\hat{a}, a_j^F\right) * \sigma'\left(a_j^F\right) * a_i^{L-1}$$

δ represents the sensitivity of each neuron with respect to cost. It can be computed using the values we already have in the output layer from the chain rule.

$$\delta_j^F = MSE'\left(\hat{a}, a_j^F\right) * \sigma'\left(a_j^F\right)$$

```
vector<double> OutputLayer::deltaForOutputNeurons(const vector<double>& targetOutput)
{
    for (int i = 0; i < outputSize; i++)
    {
        deltas[i] = meanSquaredErrorDerivative(targetOutput[i], output[i]) * sigmoidDerivative(output[i]);
    }

    return deltas;
}
```

2d. Using the weights and **δ** computed from the output layer, compute **δ** for ALL neurons in previous layer/s using:

$$\boldsymbol{\delta}_{\mathbf{j}}^{\boldsymbol{L}} = \sum_{\boldsymbol{i}}^{\boldsymbol{n}} \boldsymbol{\delta}_{\mathbf{i}}^{\boldsymbol{L+1}} * \boldsymbol{w}_{\boldsymbol{j,i}}^{\boldsymbol{L+1}} * \boldsymbol{\sigma}'\left(\boldsymbol{a}_{\boldsymbol{j}}^{\boldsymbol{L}}\right)$$

```
void HiddenLayer::calculateDelta(vector<double> deltaNextLayer, vector<vector<double>> weightsNextLayer)
{
    for(int j = 0; j < outputSize; j++)
    {
        double deltaPerNeuron = 0.0;

        for(int i = 0; i < weightsNextLayer.size(); i++)
        {
            deltaPerNeuron += deltaNextLayer[i] * weightsNextLayer[i][j] * sigmoidDerivative(output[j]);
        }
        deltas[j] = deltaPerNeuron;
    }
}
```

2e. Compute gradient for each weight and biases for the entire network using each layer's respective **δ**s:

$$\frac{\partial C_x}{\partial w_{j,i}^{L}} = \boldsymbol{\delta}_{\boldsymbol{j}}^{\boldsymbol{L}} * \boldsymbol{a}_{\boldsymbol{i}}^{\boldsymbol{L-1}} \quad \& \quad \frac{\partial C_x}{\partial b_{j}^{L}} = \boldsymbol{\delta}_{\boldsymbol{j}}^{\boldsymbol{L}}$$

```
void HiddenLayer::calculateGradientsWeight(vector<double> activationPrevLayer)
{
    for(int j = 0; j < outputSize; j++)
    {
        for(int i = 0; i < inputSize; i++)
        {
            weightGradients[j][i] = deltas[j] * activationPrevLayer[i];
        }
    }
}

void HiddenLayer::calculateGradientsBias()
{
    for(int j = 0; j < outputSize; j++)
    {
        biasGradients[j] = deltas[j];
    }
}
```

```
void OutputLayer::calculateGradientsWeight(vector<double> activationPrevLay
{
    for(int j = 0; j < outputSize; j++)
    {
        for(int i = 0; i < inputSize; i++)
        {
            weightGradients[j][i] = deltas[j] * activationPrevLayer[i];
        }
    }
}

void OutputLayer::calculateGradientsBias()
{
    for(int j = 0; j < outputSize; j++)
    {
        biasGradients[j] = deltas[j];
    }
}
```

2f. Update weights and biases on the entire network using Gradient Descent(?) using:

$$w_{j,i}^{L} = w_{j,i}^{L} + \eta \frac{\partial C_x}{\partial w_{j,i}^{L}} \;\&\; b_j^L = b_j^L + \eta \frac{\partial C_x}{\partial b_j^L} \; where \; \eta = \text{learning rate}$$

```
void OutputLayer::updateWeights(double learningRate)
{
    for(int j = 0; j < outputSize; j++)
    {
        for(int i = 0; i < inputSize; i++)
        {
            weights[j][i] += (learningRate * weightGradients[j][i]);
        }
    }
}

void OutputLayer::updateBias(double learningRate)
{
    for(int j = 0; j < outputSize; j++)
    {
        bias[j] += learningRate * biasGradients[j];
    }
}
```

```
void HiddenLayer::updateWeights(double learningRate)
{
    for(int j = 0; j < outputSize; j++)
    {
        for(int i = 0; i < inputSize; i++)
        {
            weights[j][i] += (learningRate * weightGradients[j][i]);
        }
    }
}

void HiddenLayer::updateBias(double learningRate)
{
    for(int j = 0; j < outputSize; j++)
    {
        bias[j] += learningRate * biasGradients[j];
    }
}
```

Repeat step 2 until the cost is acceptable.

Prediction and Classification:

```
vector<double> OutputLayer::predict()
{
    vector<double> softmaxValues(outputSize);
    vector<double> prediction(2);
    double denominator = 0.0;

    for(int i = 0; i < outputSize; i++)
    {
        denominator += exp(logits[i]);
    }

    for(int i = 0; i < outputSize; i++)
    {
        softmaxValues[i] = exp(logits[i]) / denominator;
    }

    int maxIndex = 0;
    for(int i = 1; i < outputSize; i++)
    {
        if(softmaxValues[i] > softmaxValues[maxIndex]) maxIndex = i;
    }
    prediction[0] = maxIndex;
    prediction[1] = softmaxValues[maxIndex];
    return prediction;
}
```

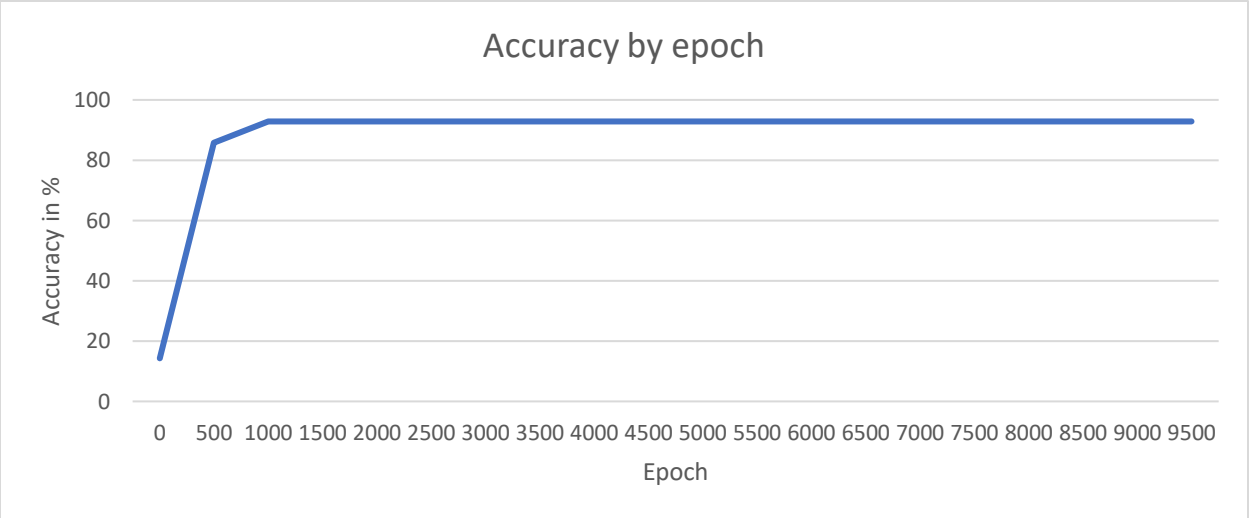
Used in the prediction is the softmax function:

$$\sigma(\vec{z})_i = \frac{(e^{z_i})}{\sum_{j=1}^{K} e^{z_j}} \textbf{ where } \boldsymbol{z = logits\ (unactivated\ value)}$$

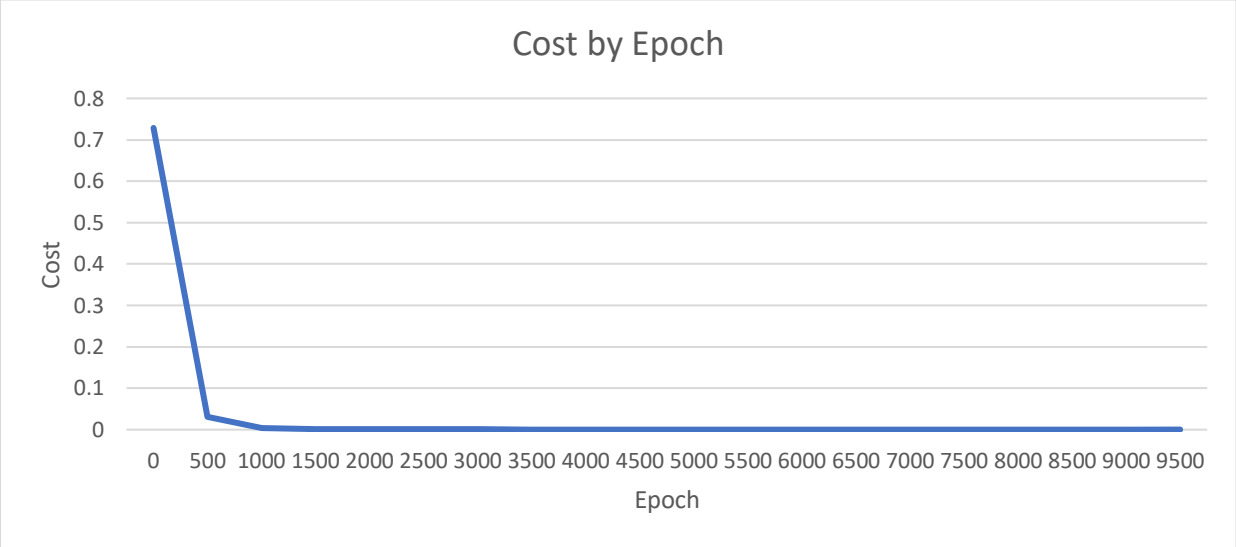
Why Softmax?

Because it is ideal for multi-class classification problems, such as determining the type of line in the grid, where the output needs to represent probabilities for each class. The function normalizes the logits (unactivated values) into a probability distribution, ensuring the sum of probabilities across all classes equals 1. This makes it particularly suitable for interpreting the output as class probabilities, which is not as straightforward with other activation functions.

IV. Model Performance



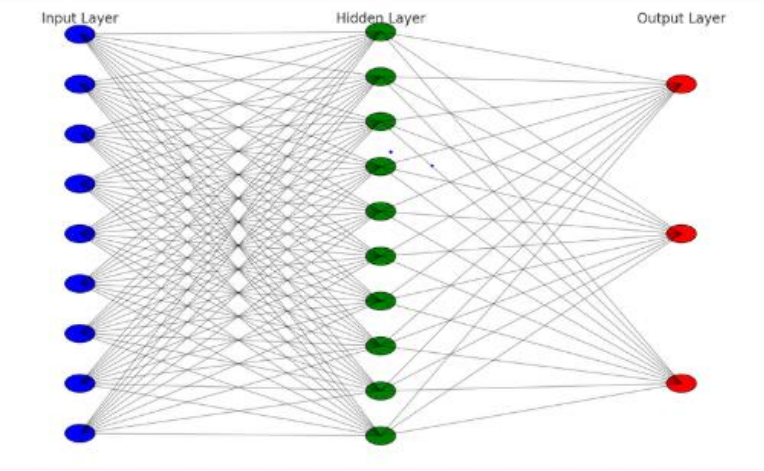
Initial Accuracy: 14.285714285% Final Accuracy: 92.857142857%



Initial Cost: 0.728637 Final Cost: 0.000145964

The model demonstrated strong performance, with accuracy quickly increasing in the early training epochs and stabilizing at 92.86% by the end of training. This indicates that the model effectively learned the classification task and converged well, supported by the cost graph, which shows an exponential decline to a minimal value of 0.000145964. However, the model could have stopped training by the 500th epoch or earlier, as the cost showed minimal reduction beyond that point, suggesting that additional epochs had little impact on improving the model's performance.

Architecture



Confusion Matrix

	Predicted Diagonal	Predicted Vertical	Predicted Diagonal
Actual Diagonal	4	0	0
Actual Vertical	1	5	0
Actual Diagonal	0	0	4

The confusion matrix highlights that the model correctly classified most test samples, with 4 instances of "Diagonal," 5 of "Vertical," and 4 of "Horizontal" accurately predicted, and only 1 misclassification in the "Actual Vertical" category. This high accuracy and low error suggest that the model generalized well to unseen data

V. Overall insight

The model effectively classified noisy, real-world data with a final accuracy of 92.857%. Accuracy increased rapidly during training and stabilized early, indicating efficient learning. Minimal cost improvement beyond the 500th epoch suggests the training process could have ended earlier without performance loss. The noisy testing data (one-pixel perturbations) validated the model's robustness by mimicking real-world conditions, rendering training vs. validation accuracy comparisons unnecessary. These results highlight the model’s reliability for line classification, with potential for further optimization through early stopping.

VI. Tools and Technologies

1. IDE:

- Visual Studio Code
- Visual Studio

2. Libraries Used:

2a. Standard C++ Libraries:

- #include <iostream>
- #include <vector>
- #include <cstdlib>
- #include <algorithm>
- #include <chrono>
- #include <windows.h>

2b. Custom Libraries:

- #include "InputLayer.h"
- #include "hiddenLayer.h"
- #include "outputLayer.h"

3. There are no external frameworks are used in the implementation.

VII. Code Structure

Hyperparameters:

- Epoch = 10,000, Learning Rate = 0.1, Input Layer – 8 neurons, Hidden Layer – 10 nuerons, Output Layer – 3 neurons

Training (1st Image) and User input for testing (2nd image)

```
// Loop again until the cost is acceptable
for (int epoch = 0; epoch < epochs; epoch++)
{
    double cost = 0.0;
    // Step 2: For EACH training image
    for(int i = 0; i < trainingImages.size(); i++)
    {
        // Step 2a: Forward Pass
        inputLayer.setInputData(trainingImages[i]);
        hiddenLayer.propagateForward(inputLayer.getInputData());
        outputLayer.propagateForward(hiddenLayer.getOutput());

        // Step 2b: Calculate cost PER image
        cost += outputLayer.meanSquaredErrorCostPerImage(targetOutputs[i]);

        // Backpropagation
        // Step 2c: Calculate delta for outputLayer
        outputLayer.deltaForOutputNeurons(targetOutputs[i]);

        // Step 2d: Calculate delta for previous layer
        hiddenLayer.calculateDelta(outputLayer.getDeltas(), outputLayer.getWeights());

        // Step 2e: Calculate gradients for each weight and biases for the network
        // Calculate Weight Gradients w^L_(j,i) and bias of Neuron^L_j
        outputLayer.calculateGradientsWeight(hiddenLayer.getOutput());
        outputLayer.calculateGradientsBias();
        // Calculate Calculate Weight Gradients w^(L-1)_(j,i) and bias of Neuron^(L-1)_j
        hiddenLayer.calculateGradientsWeight(inputLayer.getInputData());
        hiddenLayer.calculateGradientsBias();

        // Step 2f: Update the weights and biases using Gradient Descent(?)
        outputLayer.updateWeights(learningRate);
        outputLayer.updateBias(learningRate);
        hiddenLayer.updateWeights(learningRate);
        hiddenLayer.updateBias(learningRate);;
    }

    finalCost = calculateLossPerEpoch(cost, trainingImages.size());
    if(trainCount == 0 && epoch == 0) firstLoss = finalCost;
}
```

```
while (true)
{
    vector<double> userInput = getUserInput(gridState);
    inputLayer.setInputData(userInput);
    hiddenLayer.propagateForward(inputLayer.getInputData());
    outputLayer.propagateForward(hiddenLayer.getOutput());
    vector<double> prediction = outputLayer.predict();

    string res;
    if (prediction[0] == 0) res = "Diagonal";
    else if (prediction[0] == 1) res = "Vertical";
    else res = "Horizontal";

    cout << "Your input is " << prediction[1] * 100<< "% a " << res << "." << endl;

    Sleep(500);
}
return 0;
```

VIII. References

- Baeldung. (2024, March 18). *Random initialization of weights in a neural network*. Baeldung.
- DataCamp. (2024, October). *Introduction to activation functions in neural networks*. DataCamp.
- DataCamp. (n.d.). *Loss function in machine learning: Exploring MSE and other approaches*. DataCamp.
- Machine Learning Mastery. (n.d.). *Using activation functions in neural networks*. Machine Learning Mastery.
- Machine Learning Mastery. (n.d.). *Gradient descent and its application in machine learning*. Machine Learning Mastery.
- GeeksforGeeks. (n.d.). *The role of softmax in neural networks: Detailed explanation and applications*. GeeksforGeeks.