Neural Networks Project Report

Module: "Softwaretechnik"

Wintersemester 2024/2025

Lecturer: Prof. Dr. Algorri

Author: Abdul-Malik Akiev | 11159247
Date: January 24, 2025

Contents

1 Introduction

In this report, we describe the development and results of our Neural Networks project. The project revolves around creating a simple feed-forward neural network in C# (WPF) and testing it on different datasets (*MNIST digits* and *Fashion-MNIST* from Zalando).

The following sections document the system architecture, the code structure, the experiments, and the final results for digits and clothing classification.

2 System Architecture

2.1 Class Diagram

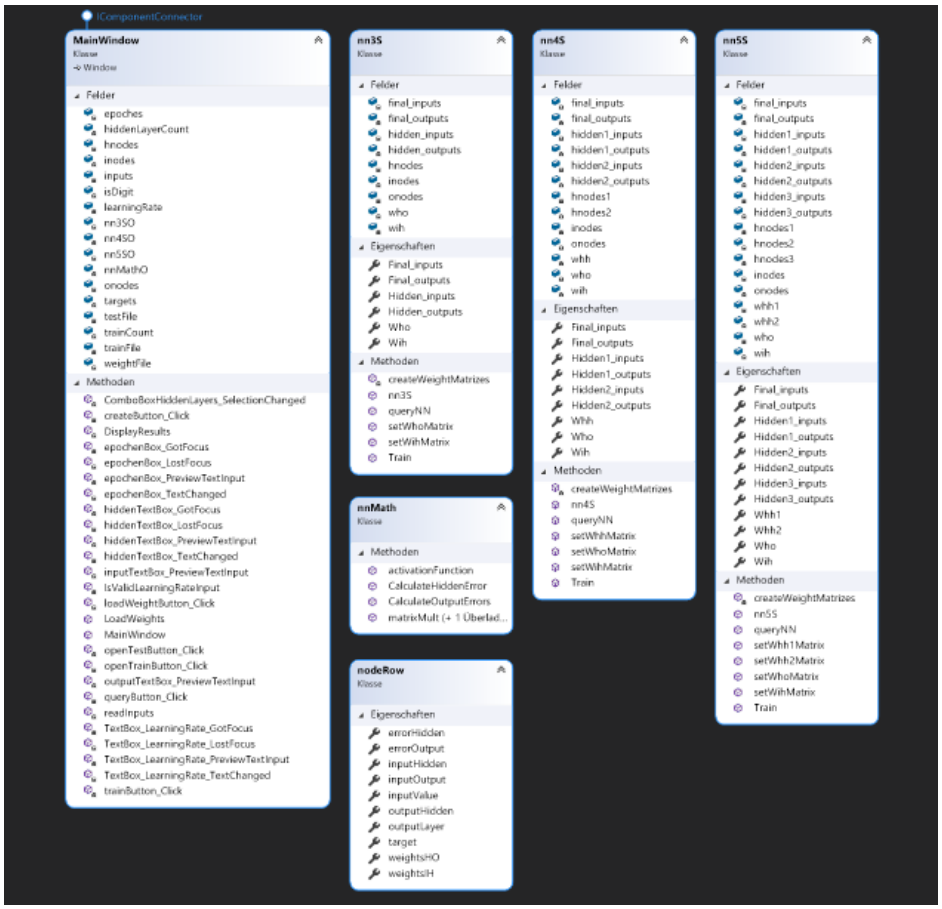

Figure 1: Class diagram of our neural network project.

2.2 Project Structure

The project has a graphical user interface (`MainWindow.xaml`), as well as several classes:

- **MainWindow.xaml/.cs**: Handles all UI logic, event binding (e.g., loading files, training, querying).
- **nn3S, nn4S, nn5S**: Implement the neural network with 1, 2, or 3 hidden layers, respectively.
- **nnMath**: Utility class with common matrix operations and the sigmoid function, etc.

3 Implementation Details

3.1 MainWindow.xaml / MainWindow.xaml.cs

Here is an excerpt from the `MainWindow.xaml` file, showing how the UI elements (DataGrid, Image, Buttons, etc.) are defined:

Listing 1: MainWindow.xaml (excerpt)

```
<Grid>
 <Button x:Name="queryButton" Content="Query␣Network"
    HorizontalAlignment="Left" Margin="833,277,0,0"
    VerticalAlignment="Top" Width="115" Click="
    queryButton_Click" Height="32" IsEnabled="False"/>
        <Button x:Name="trainButton" Content="Train␣Network"
           HorizontalAlignment="Left" Margin="1045,131,0,0"
           VerticalAlignment="Top" Width="121" Height="33"
           Click="trainButton_Click" IsEnabled="False"/>
        <Button x:Name="createButton" Content="Create␣Network
           " HorizontalAlignment="Left" Margin="832,232,0,0"
           VerticalAlignment="Top" Width="116" Click="
           createButton_Click" Height="32"/>

  ...
</Grid>
```

Below is a short excerpt from `MainWindow.xaml.cs` showing how we handle the user events for creating and training the network:

Listing 2: MainWindow.xaml.cs (excerpt)

```
private void createButton_Click(object sender,
    RoutedEventArgs e)
        {
            if (inodes > 1 && hnodes > 1 && onodes > 1)
            {
                if (hiddenLayerCount == 1)
                {
                    nn3S0 = new nn3S(inodes, hnodes, onodes);
                    MessageBox.Show($"3-Schicht-Netzwerk
                        erstellt:Eingänge:{inodes}||Hidden:
                        {hnodes}||Ausgänge:{onodes}||
                        Lernrate:{learningRate}||Epochen:{
                        epoches}");
                }
                else if (hiddenLayerCount == 2)
                {
                    nn4S0 = new nn4S(inodes, hnodes, hnodes,
                        onodes);
                    MessageBox.Show($"4-Schicht-Netzwerk
                        erstellt:Eingänge:{inodes}||Hidden
                        1:{hnodes}||Hidden2:{hnodes}||
                        Ausgänge:{onodes}||Lernrate:{
                        learningRate}||Epochen:{epoches}");
                }
                else if (hiddenLayerCount == 3)
                {
                    nn5S0 = new nn5S(inodes, hnodes, hnodes,
                        hnodes, onodes);
                    MessageBox.Show($"5-Schicht-Netzwerk
                        erstellt:Eingänge:{inodes}||Hidden
                        1:{hnodes}||Hidden2:{hnodes}||
                        Hidden3:{hnodes}||Ausgänge:{onodes
                        }||Lernrate:{learningRate}||Epochen
                        :{epoches}");
                }
                else
                {
                    MessageBox.Show("Anzahl der Hidden-Layer
                        wird nicht unterstützt!");
                    return;
                }

                openTrainButton.IsEnabled = true;
                loadWeightButton.IsEnabled = true;
            }
            else
            {
                MessageBox.Show("Die Anzahl der Neuronen muss
```

```
                    ␣größer␣als␣1␣sein!");
            }
        }
```

3.2 Neural Network Classes

The following listing shows an excerpt from `nn3S.cs` where we handle a 3-layer network (input, single hidden layer, output). Note how random weights are initialized and how the training is done:

Listing 3: nn3S.cs (excerpt)

```
public class nn3S  //Neural Network 3 Layers
    {
        double[,] wih, who;
        int inodes, hnodes, onodes;
        double[] hidden_inputs;
        double[] hidden_outputs;
        double[] final_inputs;
        double[] final_outputs;

        public double[] Hidden_inputs { get { return
            hidden_inputs; } }
        public double[] Hidden_outputs { get { return
            hidden_outputs; } }
        public double[] Final_inputs { get { return
            final_inputs; } }
        public double[] Final_outputs { get { return
            final_outputs; } }
        public double[,] Wih { get { return wih; } }
        public double[,] Who { get { return who; } }

        public nn3S(int inodes, int hnodes, int onodes)
        {
            this.inodes = inodes;
            this.hnodes = hnodes;
            this.onodes = onodes;

            createWeightMatrizes();
        }

        private void createWeightMatrizes()
        {
            wih = new double[hnodes, inodes];
            who = new double[onodes, hnodes];

            // Eine einzige Instanz von Random erzeugen
            Random random = new Random();
```

```
33
34              // Gewichte für Input-Hidden-Schicht
                    initialisieren
35              for (int j = 0; j < hnodes; j++)
36              {
37                  for (int i = 0; i < inodes; i++)
38                  {
39                      wih[j, i] = random.NextDouble() * 2.0 -
                            1.0; // Werte im Bereich [-1.0, 1.0]
40                  }
41              }
42
43              // Gewichte für Hidden-Output-Schicht
                    initialisieren
44              for (int j = 0; j < onodes; j++)
45              {
46                  for (int i = 0; i < hnodes; i++)
47                  {
48                      who[j, i] = random.NextDouble() * 2.0 -
                            1.0; // Werte im Bereich [-1.0, 1.0]
49                  }
50              }
51          }
52
53
54          public void queryNN(double[] inputs)
55          {
56              nnMath nnMath0 = new nnMath();
57
58              hidden_inputs = new double[hnodes];
59              hidden_inputs = nnMath0.matrixMult(wih, inputs);
60
61              hidden_outputs = new double[hnodes];
62              hidden_outputs = nnMath0.activationFunction(
                    hidden_inputs);
63
64              final_inputs = new double[onodes];
65              final_inputs = nnMath0.matrixMult(who,
                    hidden_outputs);
66
67              final_outputs = new double[onodes];
68              final_outputs = nnMath0.activationFunction(
                    final_inputs);
69          }
70
71          public void Train(double[] inputs, double[] targets,
                double learningRate)
72          {
73              nnMath nnMath0 = new nnMath();
```

```
74
75              // Forward Pass
76              queryNN(inputs);
77
78              // Fehlerberechnung
79              double[] outputErrors = nnMath0.
                    CalculateOutputErrors(targets, final_outputs);
80              double[] hiddenErrors = nnMath0.
                    CalculateHiddenError(who, outputErrors);
81
82              // Gradienten für die Ausgabeschicht
83              double[] outputGradients = new double[onodes];
84              for (int i = 0; i < onodes; i++)
85              {
86                  outputGradients[i] = outputErrors[i] *
                        final_outputs[i] * (1 - final_outputs[i]);
                         // f'(x) = f(x) * (1 - f(x))
87              }
88
89              // Gradienten für die versteckte Schicht
90              double[] hiddenGradients = new double[hnodes];
91              for (int i = 0; i < hnodes; i++)
92              {
93                  hiddenGradients[i] = hiddenErrors[i] *
                        hidden_outputs[i] * (1 - hidden_outputs[i
                        ]);
94              }
95
96              // Gewichtsanpassung für who
97              for (int i = 0; i < hnodes; i++)
98              {
99                  for (int j = 0; j < onodes; j++)
100                 {
101                     who[j, i] += learningRate *
                            outputGradients[j] * hidden_outputs[i
                            ];
102                 }
103             }
104
105             // Gewichtsanpassung für wih
106             for (int i = 0; i < inodes; i++)
107             {
108                 for (int j = 0; j < hnodes; j++)
109                 {
110                     wih[j, i] += learningRate *
                            hiddenGradients[j] * inputs[i];
111                 }
112             }
113         }
```

—

4 Mini-Projects and Experiments

4.1 Step 1

Training with the full MNIST Dataset (60,000 images) + saving weight matrices

In this project, we moved from small-scale testing files (with 10 or 100 images) to the complete MNIST training set of 60,000 handwritten digit images. After loading the mnist_60k.txt file, we iterated through each training sample for the specified number of epochs. Each sample goes through a forward pass (to compute the outputs) and a backpropagation step (to adjust the network weights). The learning rate, the number of epochs, and the number of hidden layers/nodes were set via the GUI elements.

At the end of training, the application saves the learned weights to disk in text files to preserve the network's training progress. In our code, this is done by iterating over the weight matrices (e.g., `Wih`, `Whh`, `Who` for multi-layer networks) and writing all their values line by line to a `.txt` file. We implemented a simple naming scheme for these files (for example, weight-trainCount-epochs-hnodes-hiddenLayerCount.txt), so that each saved weight file reflects the essential training parameters.

By storing these parameters, we can reload the trained weights later — without re-running the entire 60,000-image training process — by clicking the *Load Weight Matrices* button in the GUI. This functionality streamlines further experiments, since we can quickly switch between different configurations (1, 2, or 3 hidden layers) or test different training parameters without losing any previously computed training results.

4.2 Step 2

Testing with the 10K test images

After training the network on the 60,000-image MNIST dataset for one and two epoches, we tested it using the official 10,000 MNIST test images (e.g., `mnist_test_10k.txt`). We simply loaded the file via our GUI and queried the network in a loop over all test samples. For each image, we compared the network's predicted digit with the true label and incremented a score counter if the prediction was correct (in the code of course).

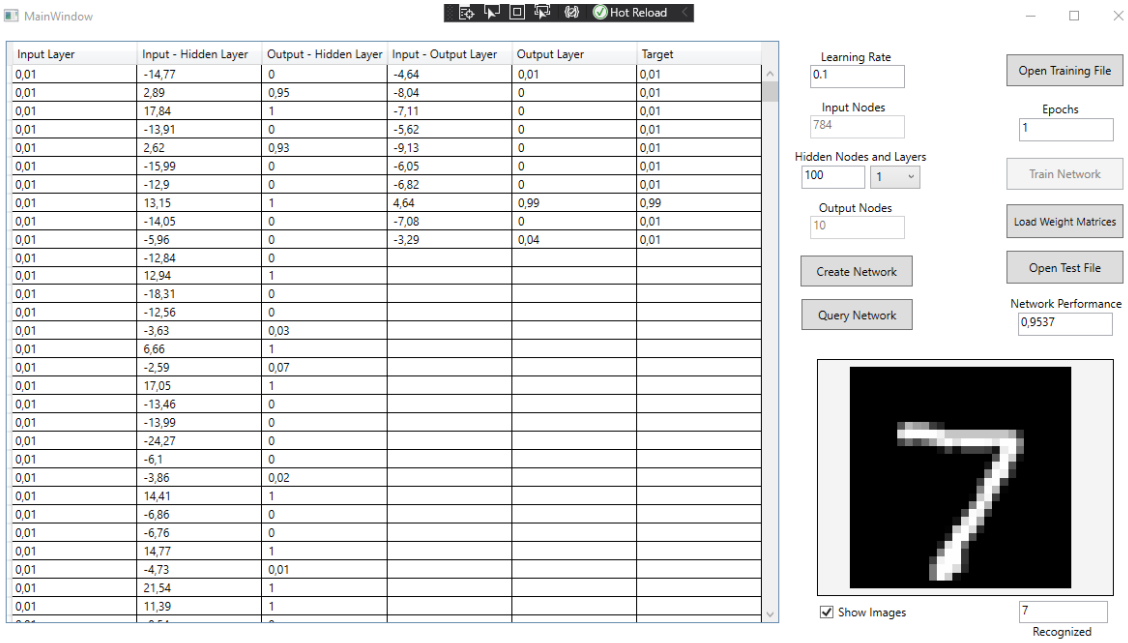

Figure 2: Best reached Performance achieved with 2 epoches

Finally, we computed the accuracy by dividing the total number of correct answers by 10,000. As shown in the screenshot below, this accuracy value is displayed in the *Network Performance* text box in the GUI.

Our best results ranged from *92%* (1 epoch) to *95%* (2 epoches), depending on factors such as the learning rate and number of hidden layers (further details can be found in the corresponding experiments).

4.3 Step 3

Using 1, 2, or 3 hidden layers

In this project, we extended the network to allow for 1, 2, or 3 hidden layers by implementing the classes `nn3S`, `nn4S`, and `nn5S`. The input and output layers remain the same; only the internal structure changes, depending on the number of hidden layers selected. Specifically:

- `nn3S`: 1 hidden layer
- `nn4S`: 2 hidden layers
- `nn5S`: 3 hidden layers

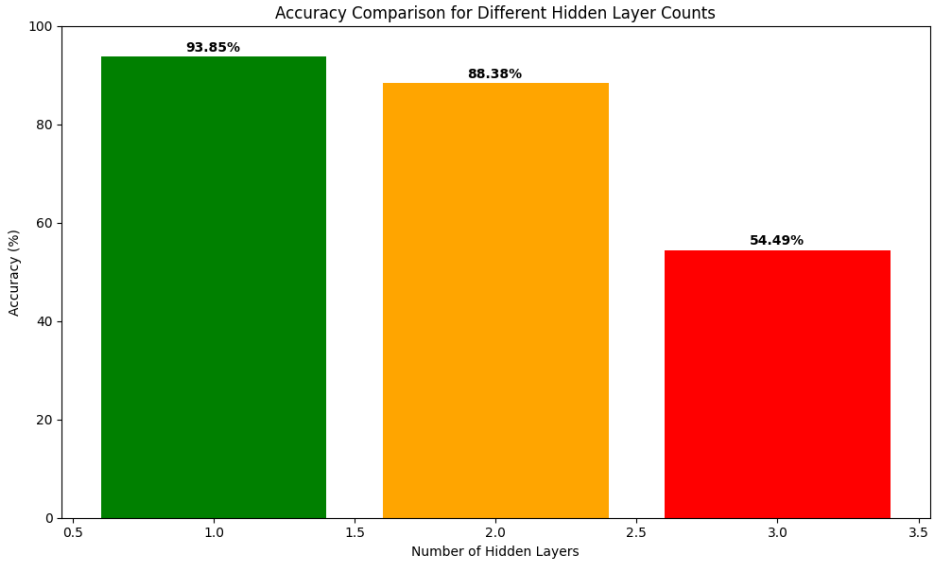

Figure 3: Accuracy comparison for 1, 2, and 3 hidden layers, each with 1 epoch, learning rate = 0.1, and 100 hidden nodes per layer.

We kept the same hyperparameters for all tests:

- **Learning Rate:** 0.1
- **Epochs:** 1
- **Hidden Nodes:** 100 (per layer)

Unexpectedly, our results showed a *decrease* in performance as we increased the number of hidden layers from 1 to 3:

- 1 hidden layer (`nn3S`): **93.85%**
- 2 hidden layers (`nn4S`): **88.38%**
- 3 hidden layers (`nn5S`): **54.49%**

While deeper networks in theory offer greater representational power, our quick test with only one epoch may not have been sufficient. Moreover, deeper architectures often require more careful tuning of hyperparameters (learning rate, epochs, and regularization) to unlock their full potential.

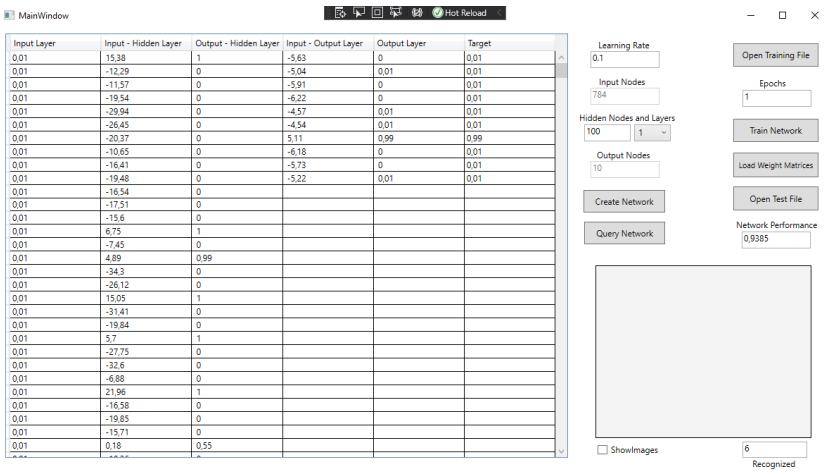

Figure 4: Network with 1 Hidden Layer, 0.1 learningrate, 100 neurons in the hidden layer and 1 epoch

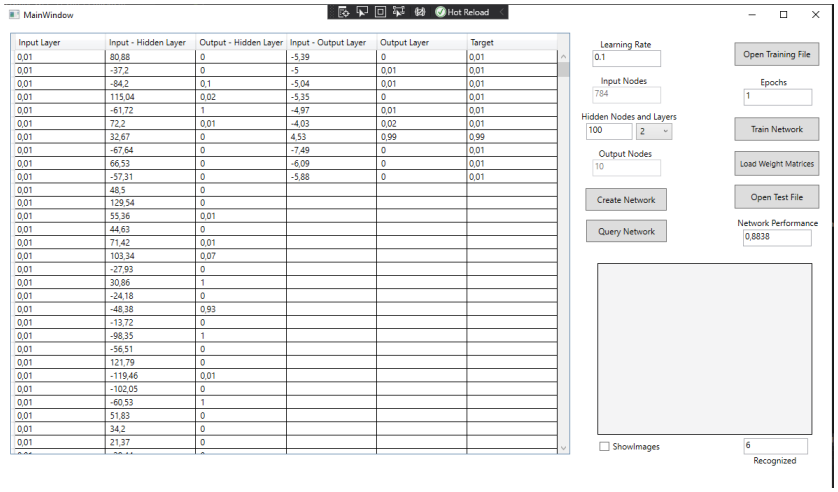

Figure 5: Network with 2 Hidden Layers, 0.1 learningrate, 100 neurons in the hidden layer and 1 epoch

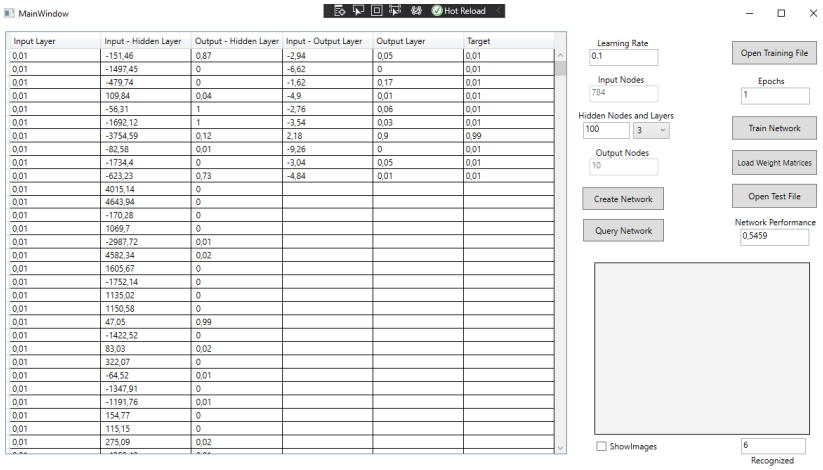

Figure 6: Network with 3 Hidden Layer, 0.1 learningrate, 100 neurons in the hidden layer and 1 epoch

4.4 Step 4

Zalando `fashion_mnist` dataset

Having completed the experiments on the standard MNIST digit dataset, we repeated the process on the Zalando `fashion_mnist_train.txt` (60,000 training images) and `fashion_mnist_test.txt` (10,000 test images). This dataset contains grayscale images of various clothing items (e.g., T-shirt, trouser, pullover, dress, sandal, sneaker, etc.), labeled from 0 to 9. The format and resolution (28×28) remain the same as in MNIST digits, allowing us to reuse the same code structure (loading files, training, querying).

Training and Testing Setup.
We employed the same hyperparameters as in our MNIST digit experiments to allow a rough comparison:

- Learning Rate = 0.1
- Hidden Nodes = 100 (per layer)
- 1 epoch of training (unless otherwise noted)

Results and Observations.
In our tests with 1, 2, or 3 hidden layers, we obtained noticeably lower accuracies compared to digit classification. Below are the approximate performances (correct labels / total) read off from the GUI:

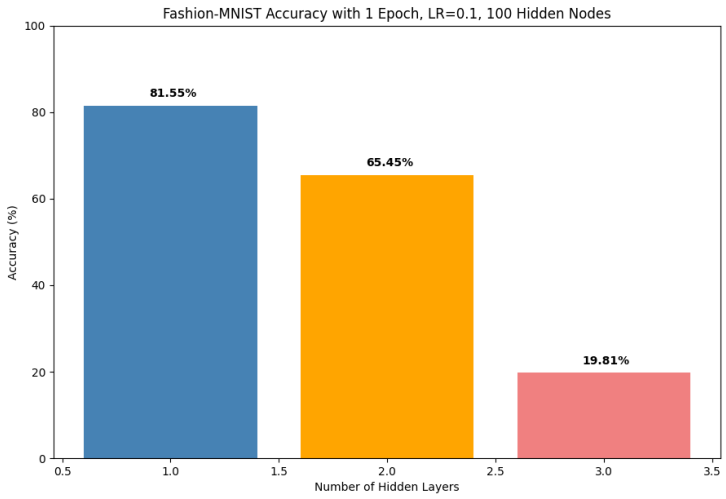

Figure 7: Accuracy comparison for 1, 2, and 3 hidden layers, each with 1 epoch, learning rate = 0.1, and 100 hidden nodes per layer.

- **1 hidden layer:** around 81.55% performance
- **2 hidden layers:** around 65.45% performance
- **3 hidden layers:** around 19.81% performance

A key observation is that while a single hidden layer still yields a decent accuracy (over 80%), adding more hidden layers *without adjusting other hyperparameters* resulted in a drop-off — similar to the MNIST digit case but even more pronounced. It is likely that fashion images have greater intra-class variability compared to handwritten digits, so the network struggles more, especially with minimal training epochs. Longer training (or parameter tuning) might improve the results further.

Example GUI screenshots.
In the images below, we see that certain items are recognized as "Sandal," "Sneaker," or "Ankle Boot", and the *Network Performance* box displays the final accuracy. The classification tends to be reliable with one hidden layer for some easily distinguishable items (e.g. sandals), but with deeper networks we see a dramatic performance drop at the given training settings.

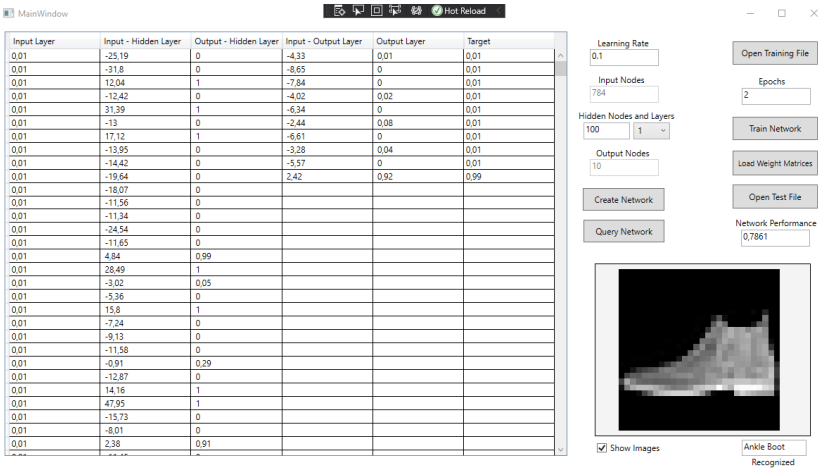

Figure 8: Test-Run with 2 epoches, result: This time the successrate gets slightly worse in comparision to only one epoch

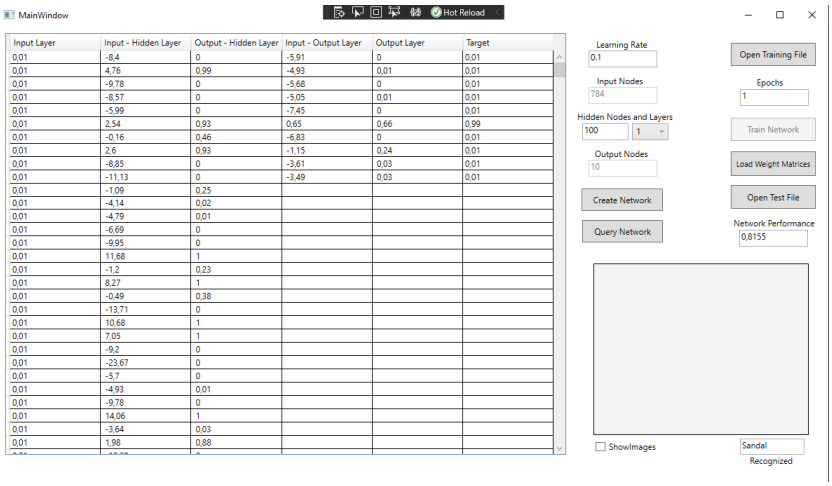

Figure 9: Network with 1 Hidden Layer, 0.1 learningrate, 100 neurons in the hidden layer and 1 epoch

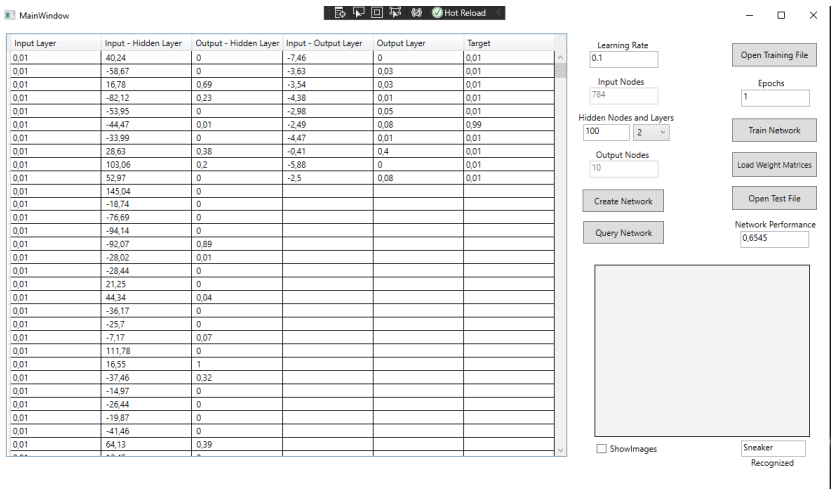

Figure 10: Network with 2 Hidden Layers, 0.1 learningrate, 100 neurons in the hidden layer and 1 epoch

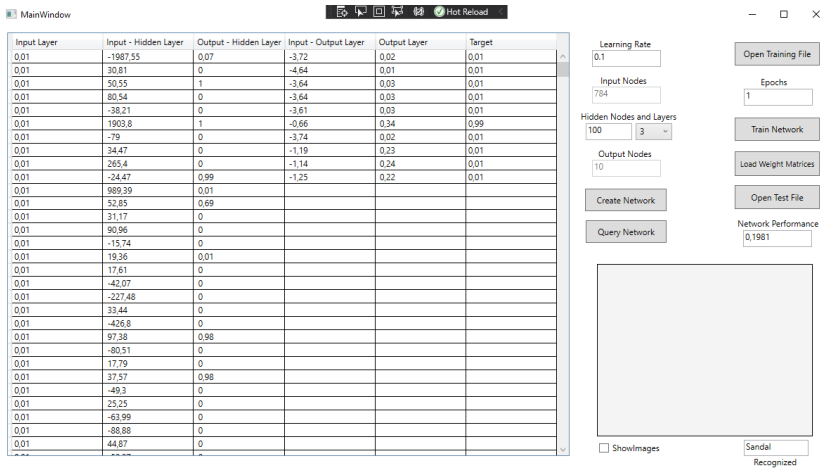

Figure 11: Network with 3 Hidden Layers, 0.1 learningrate, 100 neurons in the hidden layer and 1 epoch

Comparison to digit classification.
Compared to digit classification, fashion items are more varied in shape and texture. Hence, the same single epoch and learning rate of 0.1 did not yield accuracy as high as with digits. Proper hyperparameter tuning or additional epochs would likely narrow the gap. Nonetheless, the results demonstrate that our neural network implementation can handle different data, provided the data is properly normalized and formatted.

—

5 Results and Discussion

5.1 Performance on MNIST Digits

For the MNIST digit dataset, we conducted experiments with different numbers of hidden layers and evaluated the performance after training for a single epoch with a learning rate of 0.1. The results were as follows:

- 1 hidden layer, 1 epoch, learning rate 0.1, final performance: **93.85%**
- 2 hidden layers, 1 epoch, performance: **88.38%**
- 3 hidden layers, 1 epoch, performance: **54.49%**

From these results, we can observe that a single hidden layer provided the best performance, achieving nearly 94% accuracy. Adding a second hidden layer caused a performance drop to about 88%, and with three hidden layers, the performance significantly declined to approximately 54%. This suggests that, at least with the chosen hyperparameters, increasing the number of hidden layers leads to diminishing returns in performance. Additionally, we note that even with the same number of epochs, the network with more hidden layers required more precise tuning and training to avoid overfitting or underfitting.

It is possible that the reduced accuracy with deeper networks is due to a combination of factors such as insufficient training (only one epoch), the risk of overfitting, or an inappropriate learning rate for the added complexity. Further experimentation with a larger number of epochs and different learning rates could provide better insight into the network's ability to generalize with more hidden layers.

5.2 Performance on Zalando Fashion-MNIST

For the Zalando Fashion-MNIST dataset, we repeated the same experiments using the training file `fashion_mnist_train.txt` and the testing file `fashion_mnist_test.txt`. The results were as follows:

- 1 hidden layer, 1 epoch, performance: **81.55%**
- 2 hidden layers, 1 epoch, performance: **65.45%**
- 3 hidden layers, 1 epoch, performance: **19.81%**

The performance on the Zalando Fashion-MNIST dataset is lower than the MNIST digit dataset across all configurations. This difference in accuracy can be attributed to the inherent complexity of the fashion items compared to handwritten digits. Fashion images have higher intra-class variability (e.g., different clothing types and styles) and less distinct boundaries than the digits, making the task more difficult for the network.

Similar to our observations in the MNIST digit dataset, the performance of the network decreased with the addition of more hidden layers. However, the drop in accuracy was more significant for Fashion-MNIST, especially with three hidden layers, where the performance fell drastically to about 20%. This suggests that the network struggled to generalize effectively with more hidden layers, likely due to the limited training time (only one epoch) and potentially insufficient tuning of the learning rate or network architecture for the more complex dataset.

The lower performance with deeper networks could also be attributed to overfitting or an inefficient use of the increased number of layers. More epochs or improved hyperparameter tuning would likely be required to achieve better accuracy on this more complex dataset.

—

6 Conclusion

In this project, we implemented and tested a neural network with multiple configurations, evaluating its performance on both the MNIST digit dataset and the Zalando Fashion-MNIST dataset. The network was structured using up to three hidden layers, with hyperparameters such as the learning rate and number of hidden nodes set consistently across experiments.

- **Implementation details and code structure:** The code was divided into several modules, with the main focus being on the neural network logic (`nn3S`, `nn4S`, `nn5S`) for training and querying. The graphical user interface (GUI) was designed to allow easy interaction with the network, such as loading training data, querying the trained network, and visualizing performance metrics.
- **Main difficulties:** One of the main challenges was ensuring that the application remained responsive during training, especially when processing large datasets like the full MNIST or Fashion-MNIST dataset. To avoid UI freezing, asynchronous methods were employed for data loading and training. Additionally, managing the saving and loading of weights to disk to preserve the trained models was a critical aspect of the implementation.
- **Observations about performance for different datasets and topologies:** We observed that the performance on the MNIST dataset was generally high, with the best results achieved using a single hidden layer. However, adding more hidden layers led to a decline in accuracy, especially with deeper networks. This trend was also observed in the Zalando Fashion-MNIST dataset, where the accuracy was lower across all configurations due to the higher variability in the images. In both cases, using a single hidden layer gave the best performance, and further tuning of the network parameters or additional training epochs would likely improve results.

—

References

[1] Y. LeCun, C. Cortes, and C. J. C. Burges, "MNIST handwritten digit database," 1998. http://yann.lecun.com/exdb/mnist/

[2] Han Xiao, Kashif Rasul, and Roland Vollgraf, "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms," https://github.com/zalandoresearch/fashion-mnist

[3] Arnulf Jentzen, Benno Kuckuck, and Philippe von Wurstemberger, "Mathematical Introduction to Deep Learning: Methods, Implementations, and Theory," 2023. https://arxiv.org/abs/2310.20360