F.C.T. Universidade do Algarve

Departamento de Engenharia Eletrónica e Informática (DEEI)

Digit handwriting binary classification. Follow-up report to: "Gradient descent applied to non-layered neural networks"

Diogo Fonseca

(no. 79858, grp. 3)

Professor(s): José V. Oliveira and Pedro M. S. V. Martins

A report submitted in partial fulfilment of the requirements of
UAlg for the degree of Engenharia Informática in *Artificial Intelligence*

(Problem 4: Zero or one)

December 9, 2024

Abstract

This is an undergraduate project with it's main goal to make a neural network capable of performing a binary classification of handwritten digits: the number zero and the number one. Other than simply classifying with a high accuracy, it's other goal is to be able to do so with as few connections as possible (and consequentially with as few neurons as possible). In the model, the MSE was used as a loss function and the sigmoid was used as an activation function. The handwritten digits fed as input are 20x20 grayscale images. Upon several different network designs, it was found that a single neuron with a few connections is good enough for a high accuracy.

Contents

Chapter 1

Introduction

1.1 Background

This project aims to find the minimum-weight neural network such that it can accurately classify a handwritten digit into two categories, namely identify a zero or a one. The model uses the MSE as it's loss function, the sigmoid as it's activation function and gradient descent for backpropagation (with no further backpropagation optimizations, like the momentum term).

1.2 Problem statement

Find the neural network with the minimum amount of connections that accurately discriminates a handwritten digit between the numbers 0 and 1. Furthermore each image is grayscale with a resolution of 20x20. Each pixel is a non-normalized unknown double value corresponding to a grayscale value. It could be said that "accurate" for our purposes means an accuracy, $acc > 99\%$

A follow-up restriction is that the neural network must use MSE as it's loss function and the sigmoid as it's activation function. (NOTE: the training data is given, consisting of 800 samples).

1.3 Solution approach

The dataset was normalized and shuffled, after which it was fed into a neural network with a single neuron. After this process the network was trained with a learning rate of 0.7, and a training-to-testing ratio of 0.66 (meaning 66% of the samples were used for training and 33% were used for testing). The network was trained until early stop to prevent overfitting.

Once training is complete, some very simple "naive" methods were used in order to decrease the number of connections of the neuron: first the border of the image was cropped, then a dithering-like filter was applied such that half of the image got removed.

Chapter 2

Methodology

2.1 Code structure

The code implementation is made up of 9 classes.

- **Matrix** - this is a class for matrix calculations, it is not generic as it allows only 64bit floating point variables, this was a decision made mainly to avoid constant boxing of the class Double, and also to hyper-specialize the class. This class could be easily swapped by any other framework supporting matrix operations (especially because it is completely synchronous).
- **IPropagable** - interface that represents an object which can propagate forward or backwards. It is specially constructed for machine learning purposes, and in this context it represents a "node" in a neural network.
- **InputNode** - propagable node that mainly acts as to propagate the input.
- **Neuron** - propagable node that represents a neuron.
- **NeuralNetwork** - holds a graph of neurons (not necessarily layered) and has many utilities for easily training neural networks.
- **BinaryClassifierStats** - holds the statistics of a binary classifier, (like precision or accuracy) and the functions to get such values.
- **DataPreprocessor** - class responsible for preprocessing the dataset.
- **NeuralNetworkSerializer** - serializes and de-serializes a neural network, being capable of saving or loading it from a file (NOTE: as to avoid large file sizes, java serialization was not used, instead doing it from scratch).
- **RandomNumberGenerator** - project-wide random number generator singleton, used to implicitly share the same seed with every class.

The general use case is to read from a file and preprocess the data with DataPreprocessor, create a layered network with the builder from NeuralNetwork feeding it the preprocessed data, configure the NeuralNetwork flags and finally to train the network. It should be noted that even though there is a builder to create a layered network, the NeuralNetwork supports any network that doesn't contain cycles. Furthermore it will still internally behave as if it isn't layered (no optimizations for layered).

2.2 Noteworthy implementation specifics and improvements

2.2.1 "Neuron" class

The main changes to the Neuron class was to allow for a type of propagation that was decided to be called "ghostpropagation". To understand what ghostpropagation is and why it was added, some specifics of the previous implementation should be understood: When propagation happens the network automatically caches a large chunk of data for backpropagation. Not only is this computationally expensive from the get-go, but there is another computationally expensive follow-up problem:

Assume we want to perform two forward propagations, one for the training set, and another for the testing set. This requires we switch all the inputs twice for each iteration, and to calculate tons of caches for the testing set which won't be backpropagated. We then reach the point of needing what was called "ghostpropagation": Simply a forward propagation (**with separate inputs** to the normal propagation) that **does not cache** anything for backpropagation. Not only does this fix the issue of constantly switching inputs, but also the issue of constantly calculating useless caches.

There are more computational benefits to a forward propagation that doesn't cache anything with different inputs, of which one is being able to evaluate a single input at any point (perhaps by user request). Previously this required the inputs to be switched to the evaluation inputs, then propagated with (useless) caches, and finally the inputs were once again set to what they were and propagated once again, to ensure the neural network isn't in an invalid state for several of it's operations. Such a brute-force approach is not needed with the addition of ghostpropagation, needing only to ghostpropagate with the given input, with no further cost.

2.2.2 "NeuralNetwork" class

Many changes were done to the NeuralNetwork class to ensure reasonable efficiency, such as more caching of common operations that often need to be repeated several times (and some memoization of some matrix operations).

Furthermore support for binary classifier network's statistics were added, such as the metrics precision, accuracy, kappa... These are only available for networks that are binary classifiers.

Other than this there is one more static method for loading a (static) method for loading a neural network scheme from a file and run k-fold cross validation on it.

2.2.3 "DataPreprocessor" class

New class for data preprocessing operations, such as normalization, shuffling, splitting, and some filters. Works as a static class with many data preprocessing utility.

2.2.4 "NeuralNetworkSerializer" class

The need for this class came to fruition from two needs:

- Separation of concerns and further encapsulation for NeuralNetwork.
- Lightweight neural network serialization

Given that the serialization and externalization implementations provided by java were proven extremely inefficient memory-wise even when using best practices, there came the need to

implement the serialization process from scratch. Given that the neural network only contains the input layer and the input layer, this was not an easy task. The current implementation ended up printing the weight information using a "one-pass" DFS onto a file. Each line contains the (unique) string name of the starting node and ending node, followed by the connection weight. It also prints the bias-type weights, printing only the starting node and the weight.

To de-serialize, the NeuralNetworkSerializer then interprets this file, building a reference map, and constructing a neural network with the connections and weights transcribed in the file.

NOTE: The file is not encoded nor compressed, so it is perfectly human readable.

2.2.5 "RandomNumberGenerator" class

Singleton for syncing a random number generator project wide. Came from the need of having a project-wide seed for random number generation. Passing the random number generation for each class and function breached some separation of concerns and harmed readability.

2.3 Project UML

The UML class diagram found in appendix B.1 was automatically generated using the assistance of the software Visual Paradigm and slightly adjusted for visual clarity.

2.4 Early stopping

In order to prevent overfitting, early stopping was used, stopping the training process as soon as the testing error went up from the previous epoch.

2.5 Choosing neural network architecture

Initially it was thought that a model with >2 neurons would be necessary for discriminating between a 0 and a 1. But upon experimenting, it was understood that a single neuron was necessary as not only the task was simple, but because the neural network is a **binary classifier**.

2.6 Choosing learning rate

The chosen learning rate was $\eta = \mathbf{0.7}$. After much experimentation, it was observed that a lower learning rate did not seem to affect results, acting only as a computational bottleneck, making training a long process.

Given this initial observation, it may seem that a value higher than 0.7 might be better, which might be true, but 0.7 was a compromise between having the lowest learning rate while maintaining training time within reason.

Having a low learning rate is a requirement to ensure that it converges and also to prevent overfitting, in other words maximize how good the neural network is before early stop. A decaying learning rate could have been a better option, having the best of both worlds.

2.7 Choosing training-to-testing rate

Upon experimenting with this value, it was observed that a low value (<0.5) for this parameter makes so that there is too little training data, and the network has trouble generalizing, which also means early stop happens very fast.

In contrast, when the value is very high (>0.9) we don't have a clear understanding if the network is generalizing well or if it's overfitting, often managing to have a testing error below the training error for an (extremely) extended amount of epochs.

The rate **0.66** was being chosen, selecting $\frac{2}{3}$ to be the training data and $\frac{1}{3}$ to be the testing data. This comes with a compromise, where we trade the network's precision and accuracy for a better understanding of how well it's generalizing. Given the small amount of total samples, it was thought to be critical to really understand how well the network is generalizing to a much greater dataset. More on this in section 2.8.

Some training with the rate at 0.8 was also done, outperforming the rate at 0.66 on average as expected. Not enough empirical data was taken to study nor confirm this behavior, but such is not needed as all the observations seem to go according to the general thought process described.

2.8 K-fold cross validation

For each network devised, k-fold cross validation was ran to understand if the given model would generalize well or would suffer from overfitting issues. Because K-fold is very computational expensive, and given that for higher values of k, the computation time rounds up to a few hours, a compromise between statistical accuracy and computation time was achieved, with $\boldsymbol{k = 5}$ and **1000** training iterations (or until early stop).

For each fold, the statistics of the resulting network are outputted, and then an average is computed. Such statistics include accuracy, precision and the Cohen's Kappa (for the testing set).

2.9 Training the network

It must be decided when to stop training the network. It was observed that any network stagnated after $\sim$ 1000 to $\sim$ 5000 epochs depending on the seed.

Upon this observation it was decided that the network would stop training if over 5000 epochs had passed, or if early stopping conditions were detected, whichever came first.

The implementation for such a network can be found in appendix A.1 NOTE for replication: on the provided implementation this training takes a bit of time, taking around five minutes in the hardware used for benchmarking.

2.10 Cropping filter optimization

A cropping filter was implemented as preprocessing, cropping all sides by an amount of pixels p. It was thought that the edges of the image mostly hold no significant information, and that at least 2 pixels **from each side** would be removed, meaning the image would go from 20x20 to 16x16, and the number of inputs from 400 to 256. More notably this number is a power of 2, making it easily divisible by 2, which makes a cleaner cut for the following optimization.

A note on the terminology used, these filters are hardly filters, acting more as "blindfolds", proving not every pixel is necessary to accurately discriminate the written number.

The implementation for such a network can be found in appendix A.2.

2.11 Dithering-like filter optimization

A dithering-like filter was also implemented as preprocessing, essentially removing every other pixel. The idea behind it is that it would still preserve most of the information of the information, or at least enough to distinguish between a 0 and a 1.

With this filter applied on top of the cropping filter explained in section 2.10, we reach 128 inputs instead of the initial 400.

It acts as an *extremely* crude and very lossy downscaling of the image. Any other more sophisticated form of downscaling would provide much better results and provide for perhaps an even greater downscaling. (Even something as simple as taking the average of the grayscale values of the surrounding pixels would probably outperform this technique).

Further dithering was not used (down to 64 inputs) because the network is no longer able to accurately discriminate between 0 and 1.

The implementation for such a network can be found in appendix A.3.

Chapter 3

Results

3.1 Base network

A **logarithmic** plot of the mse over the epochs and the k-fold cross validation when $k = 5$ (average for statistics gotten from each testing set) for the base network.

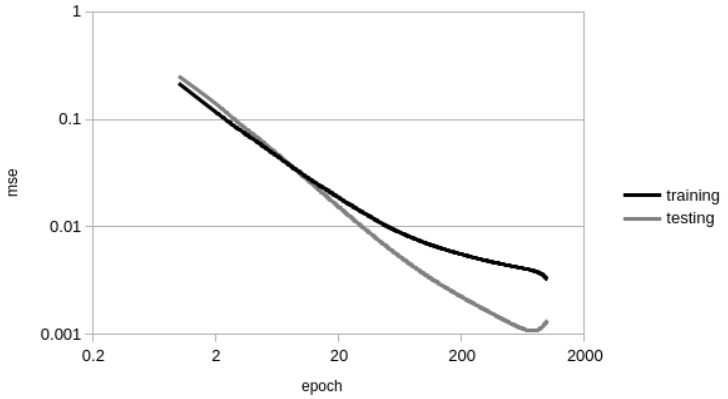

Figure 3.1: Logarithmic plot of the base network's MSE over the epochs.

MSE:	0.0046
Precision:	0.9947
Accuracy:	0.9938
Kappa:	0.9874

Table 3.1: k-fold cross validation with $k = 5$ (average values) for the base network with at most 1000 epochs.

3.2 Cropping filter network

A **logarithmic** plot of the mse over the epochs and the k-fold cross validation when $k = 5$ (average for statistics gotten from each testing set) for the cropping filter network.

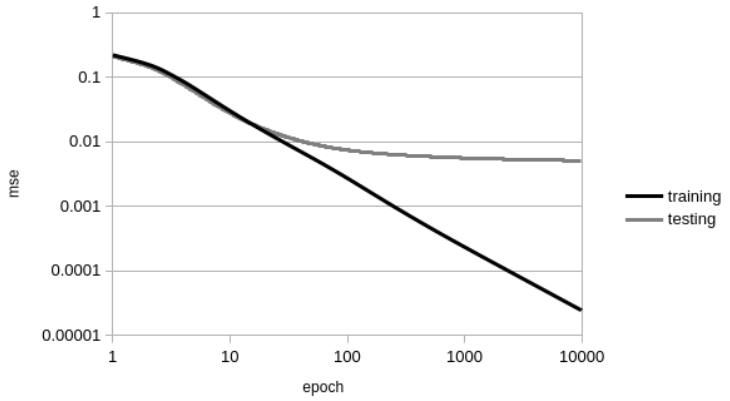

Figure 3.2: Logarithmic plot of the cropping filter network's MSE over the epochs.

MSE:	0.0027
Precision:	0.9976
Accuracy:	0.9963
Kappa:	0.9924

Table 3.2: k-fold cross validation with $k = 5$ (average values) for the cropping filter network with at most 1000 epochs.

3.3 Dithering-like filter network

A **logarithmic** plot of the mse over the epochs and the k-fold cross validation when $k = 5$ (average for statistics gotten from each testing set) for the dithering-like filter network.

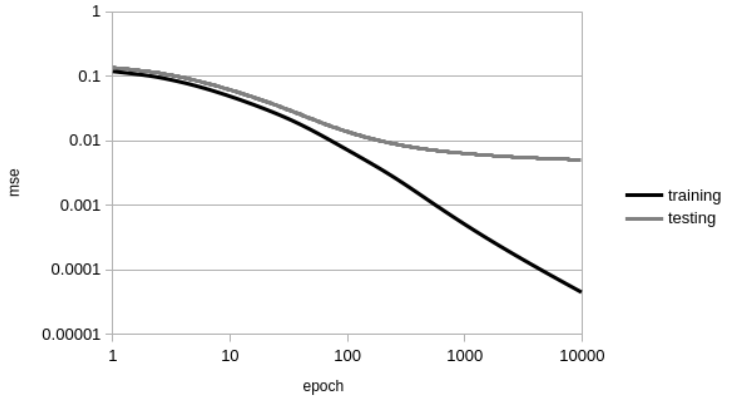

Figure 3.3: Logarithmic plot of the dithering-like filter network's MSE over the epochs.

MSE:	0.0033
Precision:	0.9951
Accuracy:	0.9950
Kappa:	0.9900

Table 3.3: k-fold cross validation with $k = 5$ (average values) for the dithering-like filter network with at most 1000 epochs.

Chapter 4

Discussion and Analysis

4.1 MSE analysis

Analyzing the MSE plotted in the graphs in chapter 3, we can observe several behaviors and relations forming.

Firstly, we can see that the testing error either reaches a plateau or starts increasing. Such is the expected behavior: as the training progresses, the neural network becomes progressively better at identifying the dataset it was trained on, at which point it starts overfitting. From such a point onward, the testing error will plateau, eventually reaching an inflection point and start increasing.

On both the network with the dithering-like filter and the network with the cropped edges, there was no apparent increase of the error. To the base network, however, we can clearly see the point at which the testing error starts increasing. Note that the graph goes up to epoch 1000, just as overfitting becomes apparent, contrary to the other graphs that go up to epoch 10000. (Note that in actual training, early stopping is applied, and the network starts training in the inflection point).

There is an oddity with the base network's MSE as the testing error starts higher than the training one, but after a few iterations, the testing error manages to get lower than the training error up until the end when overfitting starts making a big impact. This can be due to an imbalance in training to testing difficulty, meaning that the training data was much harder than the testing data.

Furthermore, we can also see that the base network has the lowest testing error of the three, reaching an MSE of around ~ 0.001. such is to be expected as it is the one which is fed the highest amount of information.

4.2 K-fold cross validation

Upon performing 5-fold cross validation upon the networks, several patterns can be observed.

The MSE of the cropping networking seems to be the lowest, such was also observed experimentally as the testing MSE of the cropping function always seemed to lower faster than the other two, even though such can't be easily discerned from the MSE graphs of a single training.

Overall the cropped network seems to statistically outperform the other two. Once again this was also expected as experimentally this network always seemed to not only converge faster, but also get better results. It's high precision and accuracy mean that it's guesses are overall correct and also has a very small amount of false positives (in this case erroneously

thinking a 0 is a 1). Cohen's kappa also shows us that the network is in almost perfect agreement with the true data, giving us a high reliability score.

Something that came as a surprise however, is the fact that the base neural network got outperformed in every metric. A possible explanation is that given the simple complexity of the task, the networks with less inputs were able to more efficiently learn the patterns that discriminate a 0 from a 1. While the base neural network was busy learning redundant data such as the edges, which are of extremely low importance in this classification task, and hold no significant information or pattern to the classification. The other two networks were able to efficiently train themselves based on the critical and information dense parts of the image, not only that but less weights means they can converge faster towards their goal, as they don't have to "share" their impact on one another.

4.3 Other observations

The only other noteworthy observation is that the time to both train and execute each network is not equal. Even though empirical tests were not done, experimentally it could be observed that the cropped network was much faster than the base network, and the network with the dither-like filter was much faster than the cropped one. Such an outcome seems very evident, as halving the inputs each time would surely have a big performance impact.

Chapter 5

Conclusions

It can be concluded that the network with the dithering-like filter excels when our goal is to minimize connections/weights. For both reliability and accuracy when our training is limited, the network with the cropping filter seems most appropriate. However for overall accuracy with a lot of training backing it up, the base network is best, being able to get the lowest MSE out of the three over a large amount of epochs.

5.1 Further optimizations

Many further optimizations in computation, training, accuracy, weight minimization and even code readability could be done. Here lie some examples of what could be done to further increase all of these.

Firstly, the code should get slight refactors, as there are some functions too big with too much responsibilities. Furthermore the NeuralNetwork class has grown too big and should be separated into other classes. With the latest additions it is clear there should be other subclasses of the NeuralNetwork class, such as "BinaryClassifier", for now binary classifier neural networks can find their methods on the NeuralNetwork class itself.

Secondly, for further optimizing both training, connection minimization and accuracy, dataset augmentation could be done, perhaps with Gaussian noise. To further optimize these, some more sophisticated image processing / convolution could be applied (such as downscaling techniques).

Further adding to the optimization list, adding a momentum term to backpropagation could greatly enhance the speed of convergence. A decaying learning rate could also be of benefit.

Other than the optimizations mentioned, the code could be greatly enhanced with parallelization, not only in it's propagations, but also in it's matrix calculations which are currently fully synchronous. Other than that there are several key points where performance could be greatly enhanced, namely by detecting layered networks and providing optimizations for them over the less efficient all-purpose implementation for non-layered networks. As mentioned in the previous report, the topological order of the network should also be precomputed, as to further speed up propagation and backpropagation.

Chapter 6

Reflection

This project helped give a bigger insight into machine learning, and a basic sense on how one would go as to train and optimize a neural network, and what techniques to use and pitfalls to avoid. It should also be emphasized just how remarkably little is necessary to identify a zero from a one. When starting the project it was thought that the minimum for such a task would be much more.

A big criticism is that the tests done were very rudimentary and are *not sufficient* for scaling the project. The ambition of the project paired with the very limited time arose to cutting corners on the testing department, not granting a completely sane usage of the implementation. Furthermore, as explained in chapter 5, the code should be refined for future usage.

References

José V. Oliveira (2024), 'Engenharia informática ualg slides'. (accessed December 2024).
URL: *https://tutoria.ualg.pt/2024/course/view.php?id=2354*

Appendix A

Code implementations

A.1 Base network training implementation

```
String separator = ",";
DataPreprocessor.normalize("dataset/dataset.csv", "dataset/
    normalized_dataset.csv", separator);
RandomNumberGenerator.setSeed(718143649137017447 6L);

// tuning parameters
int iterations = 5000;
double learningRate = 0.70;
double trainingToTestingRatio = 0.66;

// Read from file
String saveFile = "src/main/mooshak_network.ser";
String loadFile = saveFile;
String inSetFile = "dataset/normalized_dataset.csv";
String targetOutFile = "dataset/labels.csv";
double[][] outputs = DataPreprocessor.readMatrix(targetOutFile, separator)
    ;
double[][] inputs = DataPreprocessor.readMatrix(inSetFile, separator);
DataPreprocessor.shuffleRowsPreserve(inputs, outputs);

// split by training to test ratio
Matrix[] allSets = DataPreprocessor.getSplitSetsFromDataset(inputs,
    outputs, trainingToTestingRatio);
Matrix trainingSet = allSets[0];
Matrix trainingTargetOutput = allSets[1];
Matrix testingSet = allSets[2];
Matrix testingTargetOutput = allSets[3];

System.out.println("training size: " + trainingSet.columns());
System.out.println("testing size: " + testingSet.columns());

// hidden layers (none)
ArrayList<Integer> layerSizes = new ArrayList<>();

// build network
NeuralNetwork network = NeuralNetwork.layeredBuilder(400, 1, trainingSet,
    trainingTargetOutput, layerSizes);

// read if exists
File readFile = new File(loadFile);
if (readFile.exists())
    network = NeuralNetwork.loadFromFile(loadFile);
```

```
network.setTrainingData(trainingSet, trainingTargetOutput);
network.setTestingSet(testingSet, testingTargetOutput);

// set flags
network.setEarlyStopping(true);
network.setPrintingTestingError(true);

network.setExportingLoss(false);
network.setPrettyPrint(true);
network.setPrintOutputs(false);
network.setShouldPrintWhileTraining(false);
network.setShouldPrintWeights(false);

// train
network.train(iterations, learningRate);

// save network
network.saveToFile(saveFile);
```

A.2 Cropping filter network training implementation

```
String separator = ",";
DataPreprocessor.normalize("dataset/dataset.csv", "dataset/
    normalized_dataset.csv", separator);
DataPreprocessor.cropEdges("dataset/normalized_dataset.csv", "dataset/
    cropped_dataset.csv", separator, 20, 2);
RandomNumberGenerator.setSeed(247955930715666747 4L);

// tuning parameters
int iterations = 5000;
double learningRate = 0.70;
double trainingToTestingRatio = 0.66;

// Read from file
String saveNetworkToFile = "src/main/mooshak_network_v2.ser";
String loadNetworkFromFile = saveNetworkToFile;
String inSetFile = "dataset/cropped_dataset.csv";
String targetOutFile = "dataset/labels.csv";
double[][] outputs = DataPreprocessor.readMatrix(targetOutFile, separator)
    ;
double[][] inputs = DataPreprocessor.readMatrix(inSetFile, separator);
DataPreprocessor.shuffleRowsPreserve(inputs, outputs);

// split by training to test ratio
Matrix[] allSets = DataPreprocessor.getSplitSetsFromDataset(inputs,
    outputs, trainingToTestingRatio);
Matrix trainingSet = allSets[0];
Matrix trainingTargetOutput = allSets[1];
Matrix testingSet = allSets[2];
Matrix testingTargetOutput = allSets[3];

System.out.println("training size: " + trainingSet.columns());
System.out.println("testing size: " + testingSet.columns());

// hidden layers (none)
ArrayList<Integer> layerSizes = new ArrayList<>();

// build network
```

```
34 NeuralNetwork network = NeuralNetwork.layeredBuilder(256, 1, trainingSet,
       trainingTargetOutput, layerSizes);
35
36 // read if exists
37 File readFile = new File(loadNetworkFromFile);
38 if (readFile.exists())
39     network = NeuralNetwork.loadFromFile(loadNetworkFromFile);
40 network.setTrainingData(trainingSet, trainingTargetOutput);
41 network.setTestingSet(testingSet, testingTargetOutput);
42
43 // set flags
44 network.setEarlyStopping(true);
45 network.setPrintingTestingError(true);
46
47 network.setExportingLoss(false);
48 network.setPrettyPrint(true);
49 network.setPrintOutputs(false);
50 network.setShouldPrintWhileTraining(false);
51 network.setShouldPrintWeights(false);
52
53 // train
54 network.train(iterations, learningRate);
55
56 // save network
57 network.saveToFile(saveNetworkToFile);
```

A.3 Dithering-like filter network training implementation

```
1 String separator = ",";
2 DataPreprocessor.normalize("dataset/dataset.csv", "dataset/
      normalized_dataset.csv", separator);
3 DataPreprocessor.cropEdges("dataset/normalized_dataset.csv", "dataset/
      cropped_dataset.csv", separator, 20, 2);
4 DataPreprocessor.ditheringNoise("dataset/cropped_dataset.csv", "dataset/
      dithered_dataset.csv", separator);
5 RandomNumberGenerator.setSeed(2479559307156667474L); // learning rate =
      0.1, trainingRatio = 0.66
6
7 // tuning parameters
8 int iterations = 5000;
9 double learningRate = 0.70;
10 double trainingToTestingRatio = 0.66;
11
12 // Read from file
13 String saveNetworkToFile = "src/main/mooshak_network_v3.ser";
14 String loadNetworkFromFile = saveNetworkToFile;
15 String inSetFile = "dataset/dithered_dataset.csv";
16 String targetOutFile = "dataset/labels.csv";
17 double[][] outputs = DataPreprocessor.readMatrix(targetOutFile, separator)
      ;
18 double[][] inputs = DataPreprocessor.readMatrix(inSetFile, separator);
19 DataPreprocessor.shuffleRowsPreserve(inputs, outputs);
20
21 // split by training to test ratio
22 Matrix[] allSets = DataPreprocessor.getSplitSetsFromDataset(inputs,
      outputs, trainingToTestingRatio);
23 Matrix trainingSet = allSets[0];
24 Matrix trainingTargetOutput = allSets[1];
25 Matrix testingSet = allSets[2];
```

```
26 Matrix testingTargetOutput = allSets[3];
27
28 System.out.println("training size: " + trainingSet.columns());
29 System.out.println("testing size: " + testingSet.columns());
30
31 // hidden layers (none)
32 ArrayList<Integer> layerSizes = new ArrayList<>();
33
34 // build network
35 NeuralNetwork network = NeuralNetwork.layeredBuilder(128, 1, trainingSet,
       trainingTargetOutput, layerSizes);
36
37 // read if exists
38 File readFile = new File(loadNetworkFromFile);
39 if (readFile.exists())
40     network = NeuralNetwork.loadFromFile(loadNetworkFromFile);
41 network.setTrainingData(trainingSet, trainingTargetOutput);
42 network.setTestingSet(testingSet, testingTargetOutput);
43
44 // set flags
45 network.setEarlyStopping(true);
46 network.setPrintingTestingError(true);
47
48 network.setExportingLoss(false);
49 network.setPrettyPrint(true);
50 network.setPrintOutputs(false);
51 network.setShouldPrintWhileTraining(false);
52 network.setShouldPrintWeights(false);
53
54 // train
55 network.train(iterations, learningRate);
56
57 // save network
58 network.saveToFile(saveNetworkToFile);
```

Appendix B

Graphics

B.1 UML

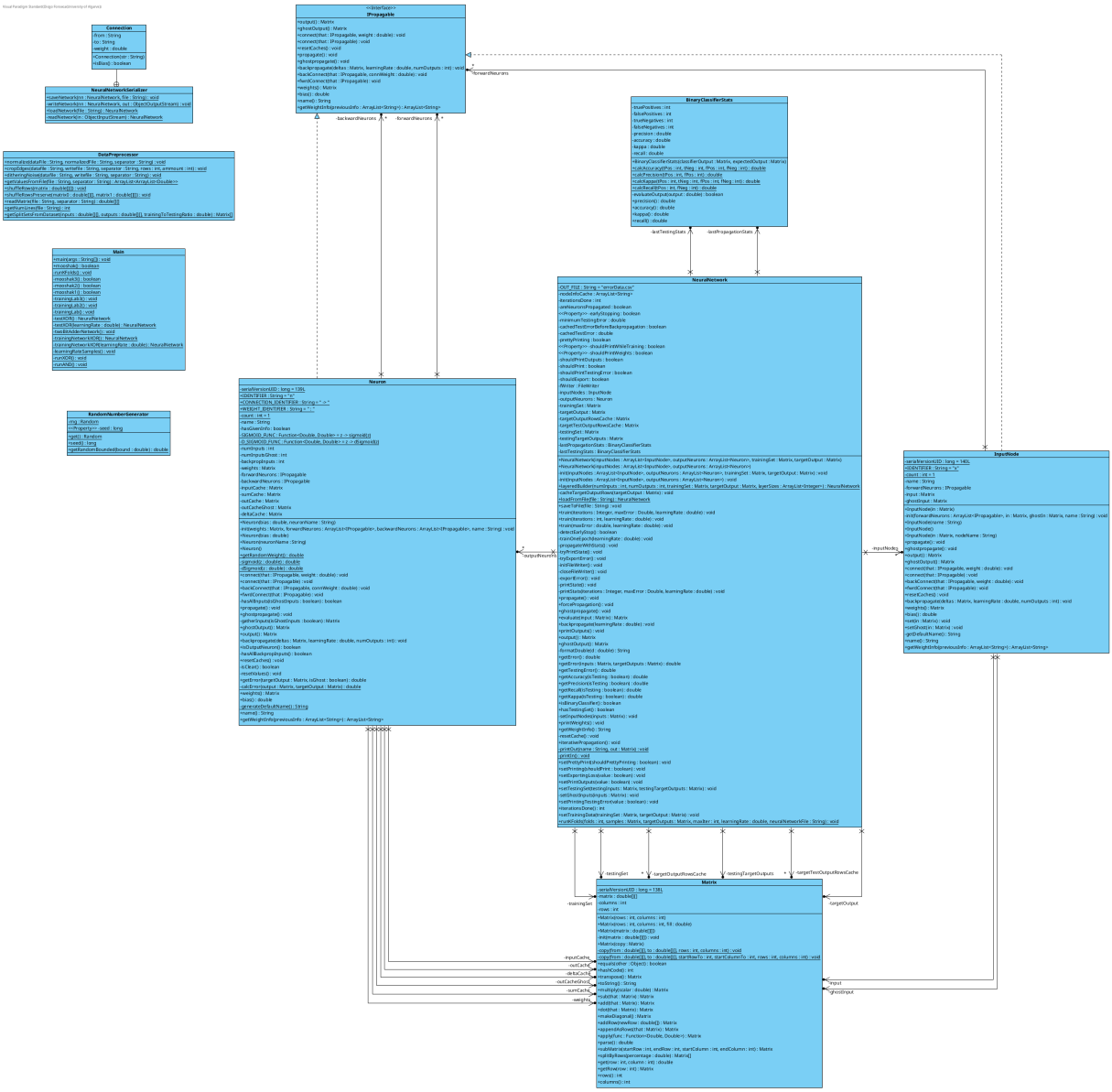

Figure B.1: UML class diagram automatically generated. NOTE: if viewing digitally, **zoom in**