Using MLP and Backpropagation for Digit Recognition in Java

Robin Gather, s1019522

January 2020

Final Project for Data Mining

1 Abstract

In Machine Learning, the Multi-Layer Perceptron (MLP) can be described as the fundamental structure for Neural Networks (NN). It is the simplest version of this category of AI solutions, yet it can solve many complex classification or regression problems. For my Data Mining project I've implemented an MLP in a modular and customizable Java program. The problem I try to solve is classifying the MNIST handwritten digits datasets. The Machine Learning (ML) algorithm that I've chosen to implement is backpropagation. I've made it my challenge to not rely on any libraries and implement the whole algorithm from scratch in Java. Along with this, I've created a GUI so that any person can test out the accuracy of the algorithm by drawing a digit. The final result is a program that creates an MLP that can classify test digits with accuracy of 95% as well as classify many user-drawn digits.

2 Problem

I've chosen to try to implement an MLP with backpropagation to dig into how the backpropagation algorithm functions. I had to understand the complex step-by-step process whereby the algorithm determines the changes to the weights. Additionally, I could make a neat application out of it. I have diverted from my initial project proposal slightly by changing the dataset that I use for the implementation. Initially I was going to classify the UCI's 'Congressional Voting Records Data Set'. Even though this isn't a more original dataset, it is more complex. To be more original I've added a GUI to support user input by allowing the user to draw a digit themselves, which will be run through the MLP. The output is then displayed as a graphical representation seen in figure 2.

2.1 Application

The dataset that I've chosen to demonstrate the algorithm's performance on is the MNIST database of handwritten digits [1]. This is a database of 60,000 training samples and 10,000 testing samples of handwritten digit images of resolution 28x28. The problem is classifying the images correctly as the digit it's supposed to represent (0 through 9). I've normalized the data by taking the alpha values of the pixels and dividing them by 255 to get values between 0 and 1 as the input data. The input layer is set to a length of 784 (28*28) nodes. In every training round the nodes are set to their respective normalized pixel blackness value. The output layer is set to 10 nodes, one for every digit. The inputs will then propagate through the network, which will output a value between 0 and 1 for every digit. This is it's confidence score. The digit with the highest confidence is its final answer. The learning algorithm will then apply the error function and backpropagation to adjust the weights of the network. The idea is that after hundreds of epochs, the network will approach a minimum in the error function and thus gradually improve in its performance of recognising these handwritten digits.

2.2 Previous Work

The last assignment of the Data Mining course dealt with Artificial Neural Networks in Python. Other than that, I have some minor experience in implementing a NN in Java. But that implementation was far more messy and rudimentary. Also, instead of using backpropagation, I used a genetic selection algorithm.

3 Methods

3.1 Pre-Existing Code

Before going into the actual implementation of the algorithm, I need to mention what external libraries, code, and data I have used from third parties and why I have used them.

As mentioned earlier in this report, I have used the MNIST handwritten digits database, both the training and testing datasets as the data which my algorithm learns from.

To read the MNIST '.idx3-ubyte' and '.idx1-ubyte' files, I've made use of Taşdelen's mnist-data-reader.[2] From the GitHub page, I've used the MnistDataReader and MnistMatrix classes. These files are licensed under the MIT License and therefore I'm allowed to use these files. I've used this code to be able to import the MNIST data and turn it into a workable array. I didn't spend the time to create a reader myself, since the focus of this project should be on the algorithm instead of a ubyte to java.util.array data conversion.

Other than that I have used no additional Java libraries or any third party code. The rest of the code is written using only the stock Java libraries of Java

1.12.

3.2 Customizability

There are two components to my implementation: the MLP and the learning agent. The learning agent can function on any neural network specified. I've made it so that you can also export a well-trained NN and import it when you run the program again. The learning algorithm can then be tuned independently of the neural network.

The MLP structure is customizable in the sense that you can specify the following attributes.

- weight bounds (default = [-1.0, 1.0])
- hidden layer amount (default = 24)
- hidden layer lengths (default = 2)
- bias range (default = 5)

The learning agent has these changeable variables:

- learning rate (default = 1)
- size of batches (default = 100)

The batches refer to the amount of training samples that should be evaluated until the average weight nudges provided by the backpropagation algorithm are applied to the weight matrix.

3.3 Maths

As mentioned in the abstract, an MLP is in essence a large mathematical equation. Inputs are given and processed through the MLP via a series of summations and activations. For every node in a layer, the nodes of the previous layer are multiplied with their respective weights and summed. A bias value is also added to this sum.

$$x_h^{l+1} = \sum_{k=0}^{n} x_k^l w_k^l + b_h^l \tag{1}$$

Both the weights and the biases are learnable variables. The value of the previous layer's nodes depends on the layer before that until the input layer is reached. After this step, the value of the node needs to be activated before being passed to the next layer. For the activation function I've used the Sigmoid function to get a value between 0 and 1.

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

Now the value can be fed forward until the output layer is reached. The feedfoward process is now complete. But since the weights are set randomly at the

beginning, the algorithm will now generate random outputs based on the input data. To allow it to learn from it's errors, we need to implement a learning algorithm. The one that I've chosen to use is back-propagation. This learning method goes backwards through the MLP, calculating the derivatives of the weights relative to the error function. First, the error is calculated.

$$E = \frac{1}{2n} \sum_{x} \|(y(x) - \hat{y}(x)))\|^2 \tag{3}$$

Here $y(x)$ is the output of the MLP $\hat{y}(x)$ is the correct output. This value can now be used in backpropagation through a series of partial derivatives, that boils down to the following equation:

$$w_{ij}{}' = -\eta \frac{\partial E}{\partial w_{ij}} \tag{4}$$

To actually calculate this value, you can break it down into a series of partial derivatives and take their product. Explaining the full step by step process is beyond the scope of this report. What backpropagation does on a more abstract level is optimising the error function by approaching a minimum. In short, it performs gradient descent on the MLP's error function.

3.3.1 Complexity

Space Complexity What needs to be stored in the working memory are the weights and a few values that determine the structure of the MLP. However, in order not to have to recalculate the values of the nodes when performing backpropagation, it is beneficial to also store the values obtained during forward propagation in working memory as well. So the amount of space that is taken up is equal to $n * 2 + |w|$. With every node added, $|n_{(l-1)}| + |n_{(l+1)}|$ weights are added too. So the total space complexity of an MLP is 2 times the amount of nodes plus the amount of weights.

$$O(2 * n + \sum_{l=0}^{l-1} (n_l * n_{l+1})) \tag{5}$$

Time Complexity On every forward propagation iteration, the summation and activation functions are called for every node. For every node this has a complexity of $O(n_{l-1})$. For every layer the complexity is $O(n_l * n_{l-1})$. For the whole system it is equal to a sum of that complexity over all the layers. Additionally, backpropagation algorithm has to calculate the error relative to the weight for every weight. To do this it loops through all weights again. The resulting time complexity is then equal to:

$$O(2 * \sum_{l=0}^{l-1} (n_l * n_{l+1})) \tag{6}$$

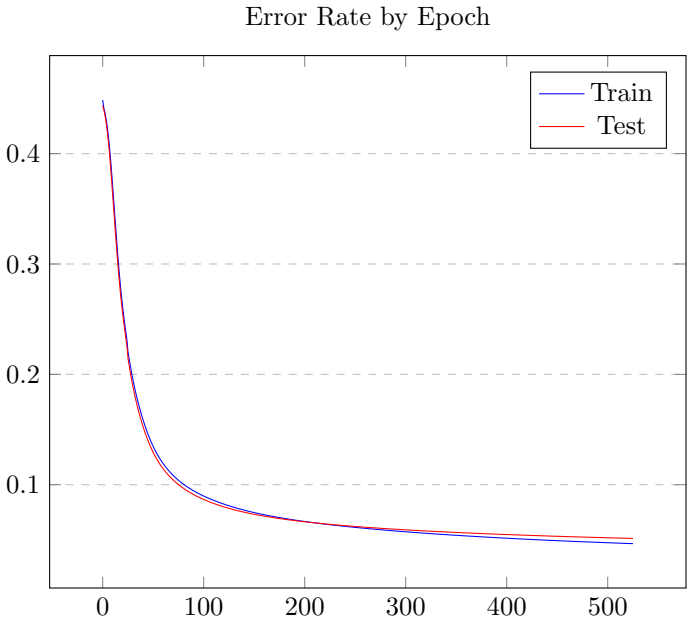

Figure 1: The performance of the algorithm on both training and test sets over the epochs

3.4 Interface

Additionally to the implementation of the algorithm, I've created an interface in which you can draw a digit yourself and see how the program responds to it. You also have the option to train, export, or reset the network. You can also have the program display a random test set digit or clear the input field. The interface is shown in figure 2.

4 Results

4.1 Performance

To show the resulting performance of my implementation, I've made the program log its training and testing error rate after every epoch (running through the 60,000 training samples). The resulting graph is depicted below.

The error rate goes down to about 0.06 for both sets at 300 epochs. However, at this point the training error rate will still decrease gradually over many epochs, whilst the test error rate stays more or less the same. So around 300 epochs a relatively good performance is reached. Going further might still improve it, but the risk of overfitting increases as well.

4.2 GUI

The program can also handle user-given input. It will run the given input through the MLP and display the output to the right. The blacker the digit (starting at alpha = 50) the more confident the MLP is.

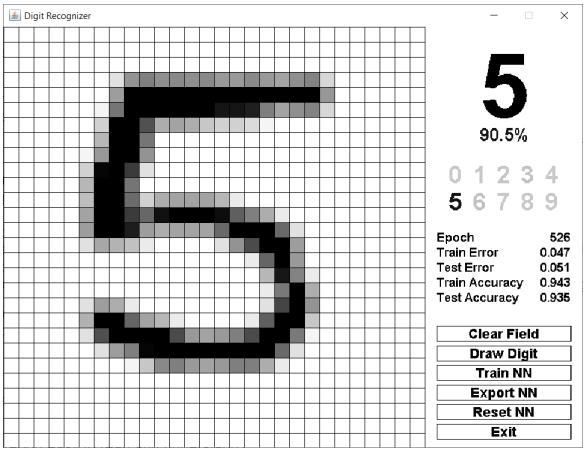

Figure 2: The algorithm's response to a user-drawn digit

5 Conclusion

Going into this project, I set out to create a solution that would have an accuracy of 98% or maybe even 99%. I wanted a program that could recognise almost every digit a user draws. But my implementation of the algorithm can only reach a maximum accuracy of 95%. This is still a relatively fine performance and it often classifies user-drawn digits correctly. But sometimes a digit that is easily classified by the human eye is mistaken by the AI. There is a gap between my expectation and my outcome, because my expectation was based on knowing that the state-of-the-art performance for handwritten digit recognition was 99.81%. Now I realise that to achieve such a high performance, I would have to make use of more modern neural network structures, like CNN or scale up the amount of hidden units to a point that training becomes too computationally expensive.

However, the implementation is still a success, and the end product is something to be proud of. I've definitely learned a lot about the functioning of neural networks and the backpropagation learning algorithm.

6 Discussion

6.1 Possibility for Improvement

To obtain the results of figure 1, I've specified the MLP to have 48 hidden units over 2 hidden layers. The best accuracy I've been able to attain with a training duration of less than a day was 95%. There have been papers written on the subject of handwritten digit classification, using techniques such as Convolutional Neural Networks, that have reached way better results.[3] There are also multiple different applications of neural networks on the MNIST page itself[1]. They range from 200 to 1000 hidden units. It was infeasible for me to test such high amounts of hidden units, since the time it takes to run through an adequate amount of epochs was too long using my single laptop. But perhaps more accuracy could be achieved with my simple MLP if I use more hidden units and a stronger machine.

7 Insight into Code

I've provided the entire code for the program on GitHub. The code itself isn't runnable, since the necessary MNIST data isn't provided in the repository. However, to test the program, I've added a runnable jar file. The Code can be found here: `https://github.com/RGather/DigitsMLP.git`

References

[1] MNIST Database of handwritten digits. Available at: `http://yann.lecun.com/exdb/mnist/`

[2] MNIST Data Reader by Türkdoğan Taşdelen: `https://github.com/turkdogan/mnist-data-reader`

[3] F. Siddique, S. Sakib, A. B. Siddique. "Recognition of Handwritten Digit using Convolutional Neural Network in Python with Tensorflow and Comparison of Performance for Various Hidden Layers" Available at: `https://arxiv.org/ftp/arxiv/papers/1909/1909.08490.pdf`