COMP30230 Connectionist Computing:
Multi-Layer Perceptron Project

By: Ryan Koenig

Programming Decisions:

- Coding in Java:

  This proved to be a bit of a challenging decisions I regretted slightly. Java is the language I know best and have had the most time working with, however with the functional nature of a MLP, Java's rigid object-oriented ruleset made it difficult to implement certain functions, and eventually led to somewhat of a mess of code.

- Class Structure:

  With Java, having many classes is very much encouraged. However, I realised very quickly that a lot of the functions of a Perceptron are linked to each other and thus splitting everything into classes would have led to a much more confusing program, for me and any reader. Thus, I created a main 'Perceptron' class that executes all the important functions of a neural network. Then I created the MLP class to handle running those functions as necessary and properly handle the output of those functions, storing them in Lists and writing to files as necessary.

- Arrays, Arrays, Arrays. Loops, Loops, Loops:

  This, I think is the weakest part of my code, and what makes it the most confusing. As a large part of how MLPs are programmed involves arrays, and Java doesn't have functional methods like 'map' or 'foldLeft' I ended up using a large amount of for loops, and likely more arrays then necessary.
  This was the main reason I regret choosing Java. Many of the loops I had would be simpler if they were able to use functional methods. This is also the spot where I believe I have the greatest chance to improve in terms of my skill and optimising the code of the MLP.

- Sigmoidal Function:

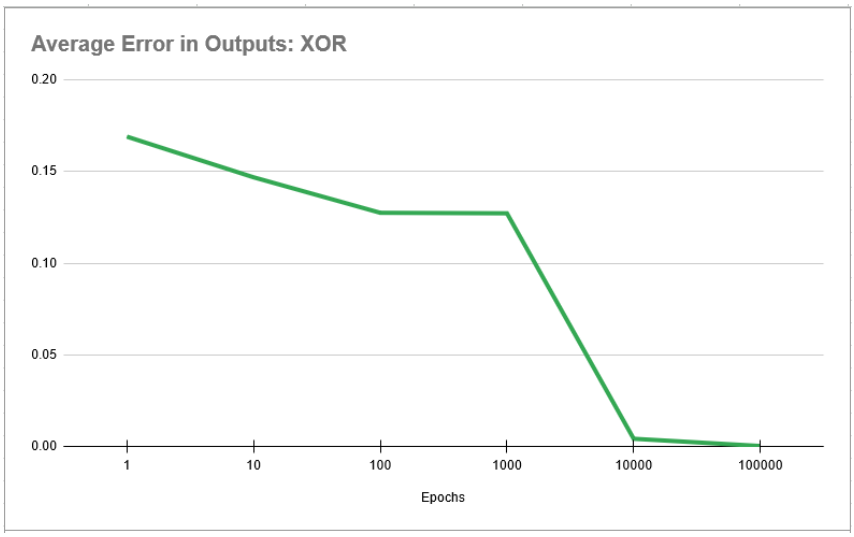
  To address the use of the sigmoidal activation function for neurons: It appears to me to be the activation function that has the highest functionality return for effort. It was easy to create a static method for it and to find an output. From memory, a sigmoidal function is also the one encouraged in the slides provided for this class, as well as the most common function I saw when looking at online resources about neural networks.

- Backwards Propagation Δw Function

  This was the hardest part of creating the code and took me the most time. I do not have a good understanding of partial derivatives or the functions that were given in the slides for the class. After looking at online resources I found "Matt Mazur: A Step By Step Backpropagation Example" which was extremely helpful and explained the functions I was confused on in a way such that I was able to implement them into code. You will likely see a lot of correlation between his work and my code. For example: how he segments out calculating node delta and my private nodeDelta function, or the ways he breaks the partial derivatives down being used as the basis for my functions. Matt Mazur, and his work did a lot of the mathematical work for the partial derivatives that I was unable to do.

1. XOR Test & Functionality:

XOR functionality is one of the first tests I ran on my MLP in order to make sure it was functional. It also happened to be quickest and most simple of the trainings. As my MLP uses a sigmoidal activation function, to return 1 or 0 is trivial in terms of using my code.



It took my program less than 100,000 epochs to create a functional XOR gate. This number could likely be lower, but this point specifically is interesting to me. As the graph to the right shows, my MLP gets stuck between 100 and 1,000 epochs, staying at around the same average error value. Then, somewhere between 1,000 and 100,000 epochs it manages to drop to an error of effectively 0. This is likely due to my MLP getting caught in some local minima between 100 and 1000, and only being able to break free in later epochs.

As a break from more formal writing, this was super cool to me. Even after multiple attempts with randomized weighting there was always this plateau and cliff in the error.

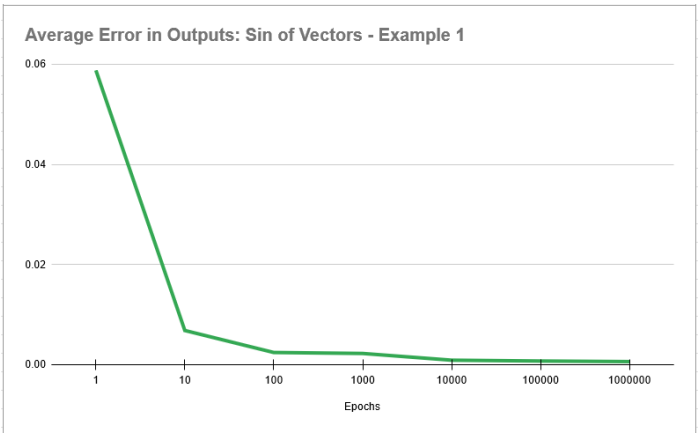
Regardless, you can see that after < 10,000 epochs the MLP escapes the local minima and becomes an effective XOR gate with minimal error, and was able to (almost) exactly get every output correct. In practice, even after 100,000 epochs of training the MLP was only able to get the outputs to either ~0.01 or ~0.99 before it truly flattens out and is unable to get significantly closer.

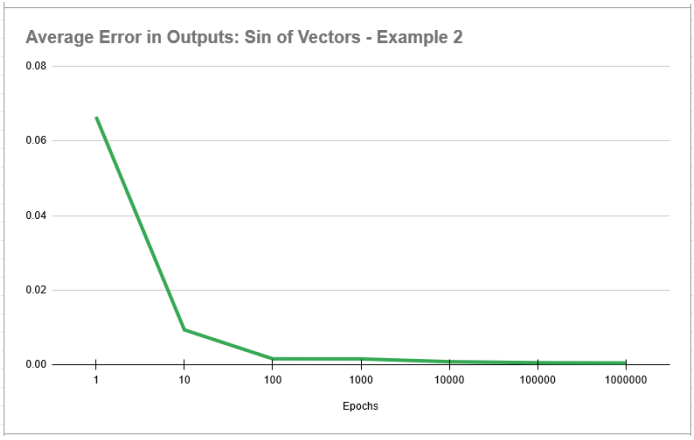
2. Sine of Vectors Test and Functionality

This specific test proved theoretically difficult to me. I spent a lot of time at the beginning trying to figure out how to create a neural network that could output between -1 and 1. Most of what I saw online as I was attempting to understand the functions of an MLP almost always returned a value of between 0 and 1.

In the end, my MLP does strictly output between 0 and 1, but once it was programmed and that was set in stone I realized I could simply adjust the outputs to correlate to -1 and 1. So, for the following information keep in mind all training inputs and outputs from the MLP were between 0 and 1, then multiplied by 2 and subtracted by 1 so they would fall between -1 and 1 or vice versa if necessary. This undoubtedly removes some level of precision from the MLP and its results. However, it seems unrealistic, and certainly out of scope, to create an MLP that can adjust to any range of numbers for any use case when a simpler MLP can output the same and let the user adjust the data as needed.

The training for this test was somewhat inconsistent, as evident by the average error of the test sample. The two graphs shown to the right show two different runs of this test, with varying results from the randomized vectors and weights. The inconsistency shows in the error of the test sample for the first MLP is ~0.000893 whereas the second MLP has an average test error of 0.00145. Granted both values are small, and the MLP is more than capable at determining the sine of a vector with reasonable accuracy, but its still a significant variance.





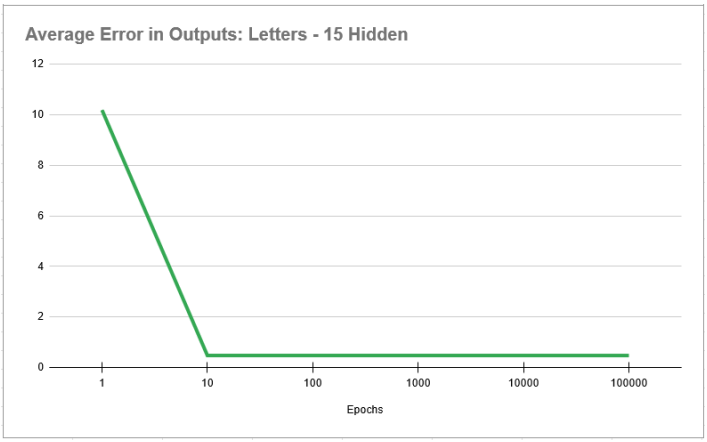
The use of a test sample that wasn't included in the original learning sample also provides interesting data. Keeping in mind that since we are calculating the input using doubles, there is a huge variance in the input and output of the function. And the total error somewhat shows that a training sample of 400 is not enough to truly create a MLP that has no bias for new, non-training, inputs. As an example, the error at the end of the excessive 1,000,000 epochs I did on the second MLP had a final average error of 4.96E-4, whereas using the testing set it got an error of 1.45E-3. Which is almost 3x greater than the error on the test set, showing the second MLP overfits the data it was trained on. This is likely also partially do to the fact I had so many

epochs. Between 1,000 and 10,000 is likely the best amount to train the MLP to the function, but not overfit to the specific cases of the training data.

3. Letter Recognition

This test was the most educational to me. Partially because it was the test my MLP performed the worst. I would even admit that my MLP fails this test. Admittedly I'm not sure why, but regardless let's cover what I tried in an attempt to fix it and what I learned in the process.

Upon a first run of this program, within just 10 epochs of shrunken dataset, 1/10th the size of the given dataset, the error settled on ~0.48[1] as shown in the graph. Now, I expected a larger error with this test as there are far more outputs and the nature of the question makes it a much tougher task for my MLP, but to me an error of 0.48 is not low enough, thus began my journey to attempt to lower it.



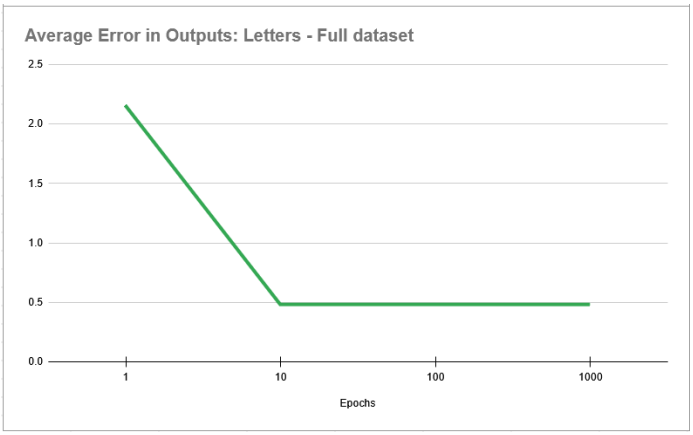
[1] There are changes in this graph, it is just imperceptible and insignificant.

After a small amount of thought, I figured an easy first check would be to increaase the number of hidden nodes in the MLP. It seemed unreasonable that the input would have 16 nodes and the output would have 26 and hidden only have the 15 I decided to give (already increased from the recommended 10). This, however, did little to fix the problem. In fact, the error stayed so similar I am not even going to show a chart for it. It did increase the amount of time I had to run the program for tenfold, which I found curious.

I had thought the number of epochs or the data size would be the most impactful choices in how long (real-life time) the MLP would have to train for, yet even at small number of epochs and a moderate to small data size, a small increase to 20 or 25 hidden nodes would increase the time complexity by so much. After some thought this does make sense. Epochs and dataset sizes multiply together with hidden nodes, however it's likely my method of programming with too many for loops ended up causing the number of hidden nodes to multiply exponentially.

My first thought was that the dataset was simply too small. Yet, I thought I might be able to make it work with the reduced dataset by tweaking other numbers, so it wasn't until much later in my attempts to make the MLP learn the data did I resort to.

Eventually, after enough testing showing no results I decided to use the entirety of the dataset. This, as expected, broke my poor laptop. I suspect memory issues, or that there is a serious memory leak in my code. As such when the code is sumbitted it will be reverted to the original much smaller dataset as to prevent potentially harming your computer as well. Regardless, the code started outputting all 0s as the final calculations. After a restart of my laptop to free up more memory, The results of the test are shown on the right. Exactly the same as my initial test. I did have to lower the number of epochs in this test, as the larger dataset made the time cost of this slightly higher, but didn't increase the amount of time as much as increasing the number of hidden nodes at all.



It is possible that there is another variable limiting the functionality of this MLP, learning rate, or something I'm not remembering. It's also possible that a combination of changes in these variables could lead to an improvement, but as I'm seeing no improvement independently, or any indication that is the case, I am at a point where I believe my MLP is incapable of properly learning this dataset.

4. Conclusion

I did enjoy this project, and will likely look towards similar projects in future classes or in my free time. There is no doubt that these are programs that can and do have a huge range of applications, and as we already see they can have a huge impact on the world, for better or worse.

The XOR test is a great test to help you figure out whether your code works even the slightest bit in the first place. It was the test I used as I was developing and debugging my code, and also proved to be a reliable dataset to get extremely consistent answers once the code is written. It also seems to have an interesting local minima that my MLP would briefly get caught in while learning.

The vector test showed exactly how to make an MLP work with various problems. Where you may need to adjust how you input and output data to an MLP to get the results you want. I also happened to stumble upon an example of overfitting and how that can damage the end product if you aren't careful with your datasets and number of epochs.

The letter recognition test proved the most educational, and the toughest. Struggling against a large dataset that the MLP is unable to properly learn led to a lot of analyzing code, functions, and variables and how they affect and feed into the MLP. How poor programming can lead certain variables to have huge effects on time complexity. How sometimes there is no dataset or epoch large enough to train the MLP, and given more time, how the limitations of my functions, like strictly positive weights, a sigmoidal function of 0 to 1, or roughly coded biases could potentially be the culprit to why the MLP isn't able to learn the dataset.

I enjoyed this project, and I have a much better grasp on how multi-layer-perceptrons work in reality. If I am able to find the time, I will likely revisit perceptrons, either this one or a new one, in an attempt to create an AI that can pass test 3

Link to sheet holding error data written down cleanly and graphs.