Backprop Lab

Drake Foltz

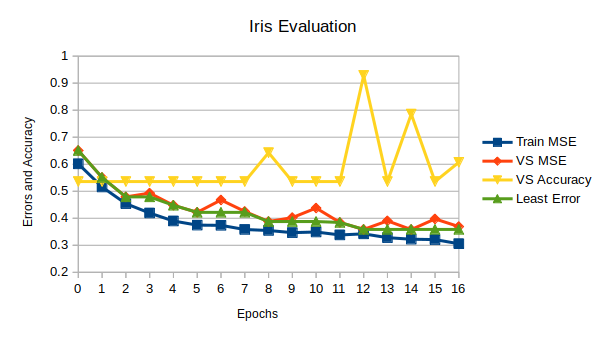
# Task 1: Implement Backprop

I started this lab with what I thought was a clear understanding of the algorithm and proceeded from there. However, as with all programming projects, I learned that I really didn’t understand it. Part of the problem was that I decided on doing a cross between object-oriented programming and the numpy implementation. My objects were “layers” which consisted of the weights preceding the set of nodes and then the activation and back-propagation methods, as well as the deltas and other things associated with the weights. The actual MLPClassifier would hold a collection of these layer objects, run a datapoint through them all, then calculate the error for all the layers, and then it would finally flush the deltas.

I started my project by testing it on the homework problem for backprop; it was a small example and easy to follow. I had an especially hard time implementing the actual back-propagation method, because the equations given in the slides were written in terms of individual nodes, and it feels like a long time since I have had much to do with matrix multiplication. But with some help I managed to get it working, and also working on the debug sample, the results of which will be included.

# Task 2: Iris and One-Hot

I must confess that I did not scrutinize the requirements, and some of these caught me by surprise. However, I had already tried to account for multidimensionality. That stood me in good stead with this lab. It was really on this part of the lab that I found that my algorithm was not quite correct for more complex examples. The same goes for my method of splitting the data, I found that it was incorrect, and I had to fall back on the allowed tools for that.

I made the decision to split and do the one-hot encoding outside of my MLPClassifier. This means that it would ordinarily be unaware of the one-hot encoding and just would follow the algorithm anyway, with the appropriate number of output nodes. If it was passed in as an argument that it was performing on one-hot encoded data, then it could take some measures, such as flattening real-valued output to ones and zeroes.

## Task 2a: Graph and Discussion

Please note that VS Accuracy is a percentile based on a “correct result” count out of a total count, whereas all the MSE and Least Error entries are based on the direct difference between the target and the output. That said, there appears to be an easy trend with the errors, always decreasing the error. I had code in place that compares the difference between the target and the output, and checks if that difference is less than some tolerance. If the difference is more or equal to that tolerance, then it will save that difference and improve the weights. If the difference is less than the tolerance, then we start using up our window after the minima.

With this graph, it makes sense that the Train MSE would go down, because that is exactly what we are training the model on. The weird spikes in VS accuracy must be variations in the splitting of the data. VS MSE has an interesting relationship with Least Error, showing the explorations of the model as it fits to the data.

# Task 3: Vowel Discussion

This dataset has a lot more classes to identify as well as many features to use. I saw nothing in the requirements to change which features were selected, but the model selected them. I would say that Train/Test is the most significant, because that size difference can change how the model fits. Gender would modify pitch of the spoken value, and speaker would be too specific.

## Task 3a: Learning Rate

In “Learning Rate” we see a pretty clear correlation between the Learning Rate and the number of epochs needed to reach convergence. That is, the smaller that the Learning Rate becomes, exponentially more time is needed to come to a satisfactory accuracy.

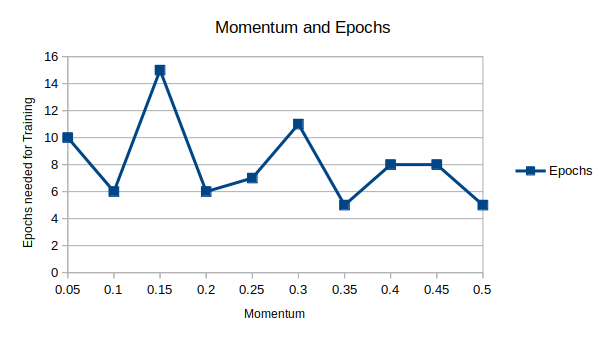
It isn’t quite so clear with the next graph, where errors are compared with the learning rate. There appears to be some optimal point around 0.05. But I would say that, given how closely related the different errors are, anywhere from 0.1 to 0.01 would be a good rate.

An important thing to note is that the set tolerance for minimum viable change in VS error would limit how small of a learning rate that I could test. I did not change it for these.

## Task 3b: Hidden Nodes

Firstly I had some difficulty getting the program to double correctly, which is why I have no entry for only one node. However, given the trend around those lesser number of nodes, I would expect something comparable as an extrapolation. Each of these error values seem to do just fine until 64 nodes, when there is a sudden spike. There are no datapoints in between, and afterwards it descends linearly.

## Task 3c: Momentum

The instructions for graphing these results are not clear, so I will just include both possibilities. In Momentum and Epochs I compare how momentum affects how many epochs are needed for satisfactory training. As you can see, there doesn’t appear to be much correlation, which I think is confusing. Even accounting for the random initial weights, I would expect to find a more optimum point somewhere along the way.

In the next graph I compare Error and Momentum, and there is a clear point that is rather erroneous. Either something in those terms was not kind to the data or my algorithm had some error, perhaps dealing with negatives. Otherwise it seems that the error was rather consistent. I am also curious as to why this run did not go beyond a momentum of 0.5. Perhaps the difference in accuracy was not enough? Or perhaps I mislabeled something?

# Task 4: Scikit-learn

This last part confused me quite badly. I looked up and managed to find out how to use the sci-kit implementation of the MLPClassifier, and I also found a dataset on all the legal end positions of a game of tic-tac-toe. I then ran my implementation and that of sci-kit with my best approximations of similar hyper-parameters. I was expecting to see that the sci-kit version would get a better score than my own. However, for every run, no matter the random hyper-parameters that I chose, both types came out with the **exact** same accuracies, every time, across all hyper-parameters. And to make matters even more confusing, each different set of hyper-parameters had the same accuracy as the previous set! My only conclusion is that the dataset may be too small to have enough variance for misclassification, or that my classifiers are not being reset with every run.

# Conclusion

To close, I have learned to start earlier, and will hopefully do so with the coming labs, as well as doing a better reading of the requirements. While I know that the best hyper-parameters are generally chosen per dataset, usually after some intrinsic analysis. Some of my results I have no idea how they came to be so confusing. Only my first 3 graphs have clear correlations and interpretations. And then I am completely confused as to why the sci-kit learn version and my version would both get the same result like they did.

As well, I am aware that this is my last late day.