Backprop Lab Report

1

For the section of the lab, we were tasked with implementing a backpropagation based MLP. As I was starting this lab, I was determined to use numpy arrays. I spent an entire day trying to figure out the exact series of matrix operations that would leave me with an elegant and efficient solution. When I returned to the project after a night's sleep, I realized I was horribly underprepared for the complex linear algebra task that this lab was turning into. I ultimately ended up scrapping all of my numpy progress to switch to python lists which I am much more comfortable with. Once I made the switch and started using lists, the coding process was much more intuitive and I was able to complete this section with only a few hiccups.

The first hiccup I ran into was in implementing momentum. While the calculation for momentum is very simple, it is apparently difficult for me to remember to update the 'previous' matrix to get my momentum to actually do something. As a result of my forgetfulness, I spent around two hours trying to identify where my code was diverging from the debug example. Once I was able to identify my mistake, it was a literal one line fix then my code worked much better.

The bigger issue with my implementation is the execution speed. As a functional programmer, I love to use list comprehensions to generate beautiful (to me) code and avoid using mutation wherever possible. For most tasks, the immutability pattern is fairly performant and leads to a lot fewer bugs in my experience. However, when working with potentially large matrices full of 64 bit floats, immutability leads to a lot of unnecessary copies and allocations. As I would later discover, even a model with only 32 hidden nodes and 14 output classes ends up taking about three minutes to train up. This made testing some of the larger hidden node sizes quite a long and painful process. As I was discovering the lack of speed, I made some optimizations in areas I viewed as slow, but the overall performance was still not where I would have liked. The solution to this is to use either a) a more performant language like C++ or b) use powerful libraries written in faster languages like numpy. For the perceptron, performance was not an issue, but I should have known that it would be for the significantly more powerful MLP.

After I finished working through all the bugs in the debug set, I took a stab at the evaluation set. As a result of all weights being initialized to zero, all hidden weight vectors are identical, so I have taken the liberty of only including one of them. Each of these values are truncated to two points past the decimal.

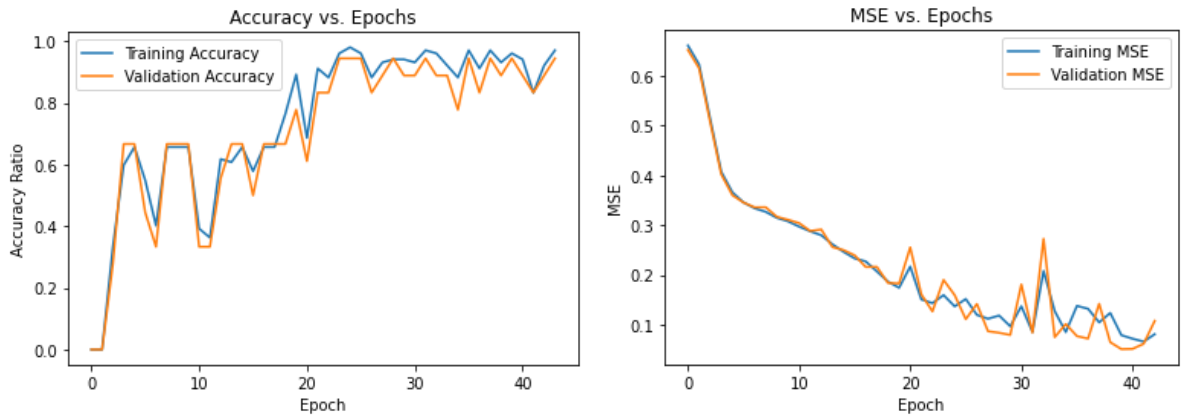
Evaluation hidden weights: [1.30, 0.80, 0.92, 0.22, -1.45]
Evaluation output weights: [-1.88, -1.88, -1.88, -1.88, 4.57]

2

This segment was focused on applying our newly designed model to the iris classification problem. As I began running data through my model, I saw all sorts of new issues with my model. Whenever I tried to fit the problem, I was greeted with an

error message informing me that some shapes were of an incompatible length. While I don't know about any one specific bug that was causing these issues, it led me down the print debug rabbithole for long enough to be frustrating. After a while of working through what data structures should exist and what size they should be, I was able to get my model running error-free for problems of any size that I could find.

During the perceptron lab, I was persistently plagued with graphing issues, so I was worried I would be trapped in pyplot purgatory permanently. To my surprise, I had no issues creating any of the charts for this lab. As I started generating graphs, I was glad to see they looked how I would expect. After a few runs on the iris dataset, my average accuracy was hovering around 90%. Some runs would yield 100% accuracy on the test set, but the majority would just bump around the low 90s, high 80s. Below is my accuracy vs. epoch and MSE vs epoch graphs



In the above figures, you can see that the validation set scores are quite tightly correlated with the training set scores. More specifically, the validation accuracy and MSE were just a tad worse than their training counterparts. From this we can infer that no crazy overfit was happening and that our validation sample was representative of the training set.

3

This section is where my poorly optimized code really started to show itself. While trying to optimize 3.2, my code took literally 19 minutes to run from start to finish. The code is particularly slow in this section mainly because there are a) many output nodes and b) there are many input nodes. For each input node, we need a weight in each hidden node. That means that there are an exponential number of weights on the number of input nodes. This is compounded by the high number of output nodes. Here we have 11 output nodes which leads to a lot of output weights. The large number of weights would not be an issue except that my code creates multiple lists containing each of them for each row dealt with. Even with the momentum term pushing the weights forward faster, I was still stuck with ~3 minutes per run of my vowel MLP. This level of performance would be completely unacceptable for any form of production code.

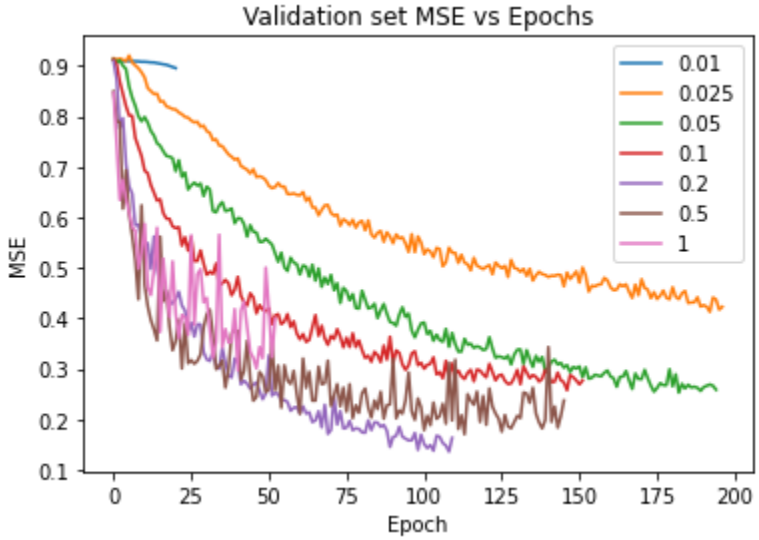
Learning rate is a difficult hyperparameter to set correctly. When it is set too low, your training takes far too long to be practical, but when it is too high, it leads to the best solutions being skipped over at every weight update. This happens due to the sporadic nature of gradient descent in higher dimensions. I'll do my best to explain more about why the learning rate needs to be set so carefully. Essentially, after each training instance is seen, the algorithm has some idea of which way is 'improvement' in each dimension. This lets the algorithm descend the gradient by simply following the current slope. When the learning rate is set too high, the algorithm is overeager at chasing the slope which can lead it to overshooting the actual minima. However, when the learning rate is too low the descent is excruciatingly slow which can lead to things like early stopping pulling the plug before improvement is found.

The vowel dataset is more difficult than iris for a few reasons. Most importantly, the curse of dimensionality means we need larger and larger amounts of data to populate our search space to properly train a model. While the vowel dataset has more training instances than the iris set, it also has significantly more inputs and outputs which more than offset the increase in samples. A one-hot model training on iris may have 3 target vectors, but a one-hot model trained on the vowel dataset will have 11. This effectively means that for each output there are only about 90 samples total. Once you remove instances for the test set and the validation set, the model simply does not have as much data to go off. Secondary to the dimensionality, this dataset does not correlate in a simply linear way. A perceptron is simply unable to solve this dataset because it is not linearly separable. The complex interactions seen in the vowel dataset mean a model has to find more complex interactions to be able to effectively solve the problem. Learning these interactions is difficult and is not always possible without a very powerful model.

After running some quick analysis code on each problem, I believe each output in each dataset is represented evenly. For vowels, this means the baseline is 90/900, 1/11 or 9.1%. For Iris, this means the baseline accuracy is 50/150, ⅓, or 33.3%. As previously mentioned, on Iris I was able to consistently score in the low to mid 90s reasonably often. While working on vowels, I was more frequently sitting around 80% test set accuracy across the best of my models.

While working on the vowel problem I decided to exclude the 'Train or Test' and 'Speaker Number' features. I excluded 'Train or Test' because I was going to split the data into my own proportion of train or test. However, even if I was not going to do my own split, I would not feed this into the model for training because I believe it is a useless feature. Presumably whoever gathered this data did not pick test data which was significantly different from training data. If we assume the training data is representative of the test data, then group membership gives us no information about output features. I excluded 'Speaker Number' because I wanted to keep the model more simple. If I wanted to include a one-hot encoding of speaker number, it would more than double the number of inputs to my already decently sized model. While I do agree that there may be some good data in this attribute, I do not believe the dataset is large enough to properly extract this information without overfitting.

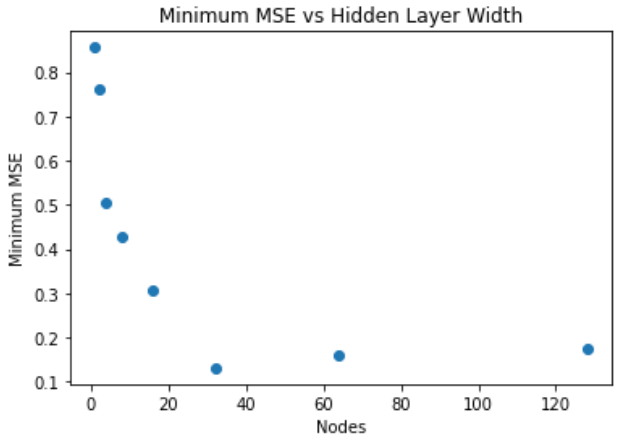
The next phase of this lab revolves around testing out different hyperparameters to see how they affect the performance of the models. The first one I tried out was learning rate.



As labeled, this graph shows the relationship between the validation set's MSE over time for a few different learning rates. The most interesting thing about this graph to me is the very clear effects of my moderately aggressive early stopping. As soon as a model is not making clear progress, my 'smart epoch' system decides it is time to stop training. For some of the smaller learning rates such as .01 and .025, you can see them making some form of progress while they are killed. The epoch system I am using compares the proportion of validation set instances gotten right to determine when to stop. This means it is possible for a model to still be making progress in terms of MSE at the time of it being stopped. Unless it gets at least one additional problem right every X epochs, the training will stop. Initially, I had the number X hard coded at 20, but after playing around with it, I decided to add it as a hyperparameter.

One possible modification to my algorithm would be to base the stopping off MSE rather than validation set accuracy. While this would allow some models to keep progressing past their current level of accuracy, I believe it would also induce overfit by rewarding higher and higher weights to reduce the MSE. I believe this would happen due to the nature of the sigmoid activation function. When using a sigmoid, the only way to get error down to zero is by getting the net value to either positive or negative infinity. This is not behavior we want because we want simple models that do not overfit our problems.

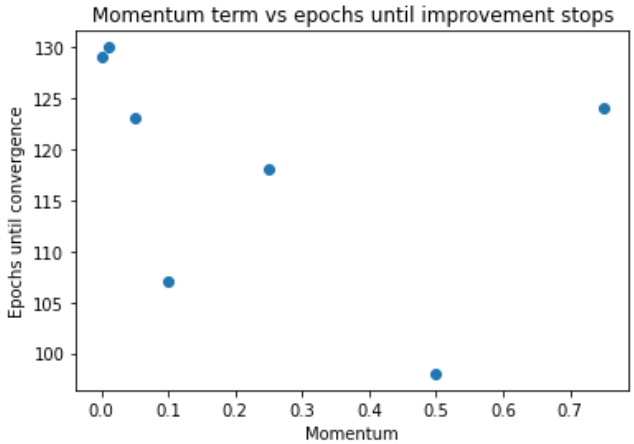
The next hyperparameter I tried playing with was the number of hidden nodes in the model. The goal of this experiment was to find a number of nodes that minimized error without wasting time and creating a needlessly complex model. Below is the graph of the best MSE at each layer against hidden layer width and the test accuracy of each of the associated models.



Nodes	1	2	4	8	16	32	64	128
Accuracy	0%	8%	50%	59%	64%	82%	86%	78%

In the chart we can see that for the first few increases in hidden layer size there are real increases in model performance. The region from 0 to 32 demonstrates that increasing node count is needed to solve some problems up to a point. However, as we look at the region from 32-128, we can see that no real improvement is made compared to the simpler models. Additionally, when we look at the table, the best performance comes from hidden sizes 32 and 64 and not from 128. To me, this says there is not enough data to train a hidden layer of size 128 effectively. It seems there is a balance to be found between not enough nodes and too many nodes to be trained.

The final hyperparameter examined in section 3 is the momentum. Momentum is generally used to quicken the training of a model without having a negative impact on its performance. I found that certain momentum values were very effective at speeding up training without having a significant impact on model accuracy. Below is the chart relating the momentum term and epochs until convergence.



Momentum	0	0.01	0.05	0.1	0.25	0.5	0.75
accuracy	79%	80%	79%	70%	80%	82%	87%

In my case, the higher scoring models happened to have more momentum, but I believe these differences to be attributable to random variance in model performance. There is not a significant difference among accuracy within the models. However ,when looking at training times, you will notice that the momentum models required significantly fewer epochs to converge on their higher accuracy. As a result of this, I believe momentum is an effective way to speed up training without negatively impacting model performance.

4

The Scikit-learn version of MLP seems like a very powerful tool. I played around with a few different hyperparameters manually and was able to get highly accurate results with a very small code footprint. My biggest surprise was the low default learning rate combined with the lower max_iterations parameter. When put together, these two have the potential to destroy accuracy. Once I figured out what was causing the training to terminate early, I was able to easily adjust these parameters until their model was performing as well or better than my own version. Speaking of performance, these models ran many orders of magnitude faster than my model. It is truly impressive how much power is freely accessible and ready to be used.

For the iris dataset I decided to go for a random search approach to identifying hyperparameters. The hyperparameters I included in the search were learning_rate_init (which was actually just learning rate because I initialized learning_rate to be 'constant'), early_stopping, and activation. I was curious if one choice would appear repeatedly in the best_params_ object after the search, but was disappointed to see many different options being chosen each time. My best test accuracy was 97% with the following hyperparameters.

{'learning_rate_init': 0.1, 'early_stopping': False, 'activation': 'logistic'}

Extra Credit:

Seeing as I had chosen random search for 4.2, I used grid search to identify good hyperparameters for 5. The search space I trained the searcher on was 7x2x3, or 42 different models. After running the classifier a few times, two things stuck out to me. Firstly, just how slowly the grid search runs even with just a few different choices to go through. The search I was performing tok around 2 minutes to complete which was much longer than the ~3 seconds of the previous models. Lastly, I was impressed by the quality of the result. After it went through its 42 training sessions, it was able to get 95% accuracy on a test set. This accuracy was achieved with the following parameters.

{'activation': 'logistic', 'early_stopping': False, 'learning_rate_init': 0.025}