Classification Using Multilayer Fully Connected Neural Networks

Monica Chan

Data Background:

I used the UCI wine data set for my project. The neural network's goal is to ultimately classify the cultivar of wines grown in the same region of Italy. There are 3 different cultivars, and so there are 3 classes.

There are 13 features for this data set, but sadly the UCI website does not give much information about these features, any description I provide (if any) are based on my guesses.

1) Alcohol - percentage alcohol in the wine

2) Malic acid - concentration in grams per liter of malic acid

3) Ash - concentration of nonorganic substances

4) Alkalinity of ash - self explanatory

5) Magnesium - magnesium concentration in grams per liter

6) Total phenols - concentration of total phenols in grams per liter

7) Flavanoids - concentration of flavanoids in grams per liter

8) Nonflavanoid phenols - concentration of nonflavanoid phenols in grams per liter

9) Proanthocyanins - concentration of proanthocyanins in grams per liter

10)Color intensity

11)Hue

12)OD280/OD315 of diluted wines - concentration of OD280/OD315 in grams per liter

13)Proline - concentration of proline in grams per liter

I did some research on some of the features that I suspected may have been repetitive (trying to filter out for collinearity), such as Ash vs. Alkalinity of ash, or Phenols vs. Flavanoids

vs. Nonflavanoid phenols, but realized that these features are distinct due to chemical properties. For instance, with Ash vs. Alkalinity of Ash, there can be a total ash concentration between two similar wines, but the chemical makeup of that Ash (composed of alkaline metals and even trace amounts of transition metals) are different, hence the need for the Alkalinity of Ash trait. Flavanoids and Nonflavanoid phenols are also different for a similar reason. For these reasons, I decided to use all the features the UCI dataset provided.

Some background information about my neural network:

Since my dataset's goal is to classify wines into 3 different classes, all my tests for this paper had 3 neurons in the output layer and 13 neurons in the input layer since there are 13 features. Therefore, most of my experimentation was with the number and size of hidden layers and tuning certain parameters.

My implementation was nearly identical to what was discussed in class, but with a few additions. My condition for convergence besides max epochs is checking for a small change in training error value. Additionally, I implemented a parameter called momentum to assist with plateaus I noticed in my dataset.

Similar to the concept of momentum in physics, the momentum parameter is checking to see how much the weights increment is changing between iterations. Therefore, the equation for the weight increment looks like this:

$$\Delta w_l = \frac{1}{N*\eta} * \Delta \odot \sigma'(Z^L) + \rho * \Delta w_{l-1}$$

Where rho is the momentum value, and l indicates the layer. The simplified concept of why this momentum value helps can be more easily understood after analyzing a training error vs. epoch graph:

Initially, I had noticed that my neural network kept converging where the training error was 0.3 which was a bit concerning. I then decided to graph a Training error vs. epoch line chart and let my neural network just converge when max epochs were reached. As seen in the chart above, the data plateaus at around 0.32 which is consistent with what I had noticed.

By implementing a momentum value, the hope is the network can overcome the plateau faster. The higher the momentum value, the faster the training error decreases.

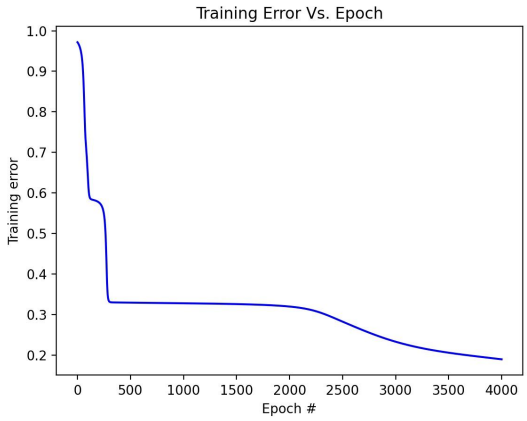


Figure 1: momentum 0

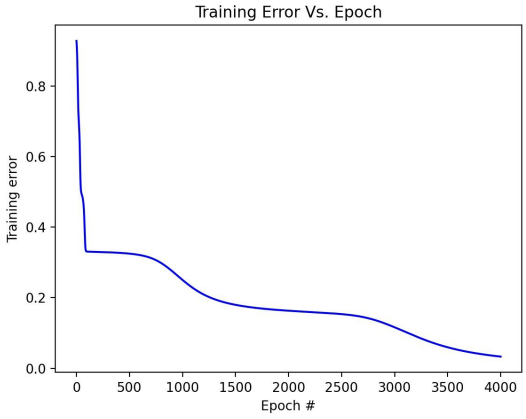


Figure 2: momentum 0.5

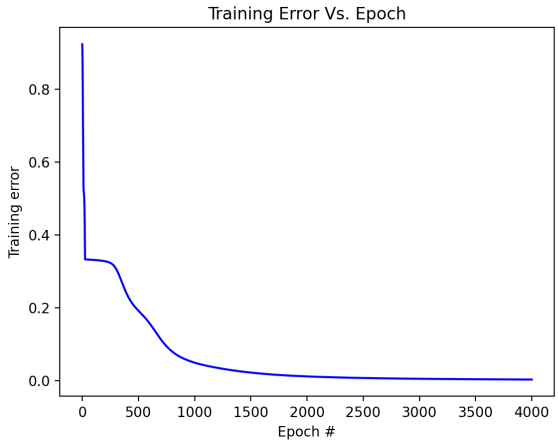


Figure 4: momentum 10

The above 3 figures visualize the impact the momentum value has. Of course, there are drawbacks with using a high momentum value, such as the model tends to "overshoot" the minimum of the cost function, and when using minibatch gradient descent the fluctuation tends to be even greater than it would be without a momentum value. Initially, I had tried some ridiculously large momentum values (35 was one that worked particularly well) but this is frowned upon so I tried to see if I could get a good result using a momentum value between 0 and 1. With a really large momentum value in practice with a larger set, more tendency to overfit or overshoot the minimum is possible. This was one of the limitations of using a small set I think.

I also used minibatch gradient descent for all these models in the interest of runtime, and I used 20 samples in my batches, and my total training set was 130/178 of the samples, about 73% of the data. I also just used sigmoid for all of my tests to make things simpler, more on activation functions later.

Experimenting with network architecture:

I first tried with 1 hidden layer, and the hidden layer has 8 neurons, the average of input and output layer size (average of 13 and 3 is 8). I played around a little with the parameters and found that a learning rate of 0.1, the momentum of 0.9, and the error change threshold of 0.0005 converged reasonably well (5/8 trials converged <5000 iterations). However, the model would often get caught on the 0.3 error snag that I discussed earlier, and if I decreased the error threshold it would just never converge. I did see this 0.3 error snag in many of the other configurations I tried. The training error was around 0.4% and the testing error was around 2%.

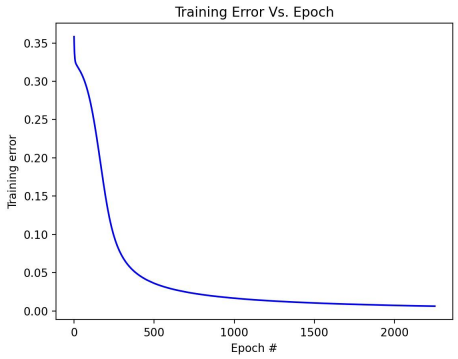


Figure 5: Training error vs. epoch for 1 hidden

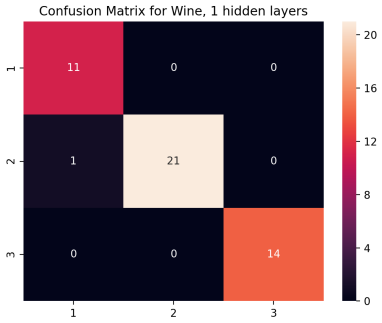
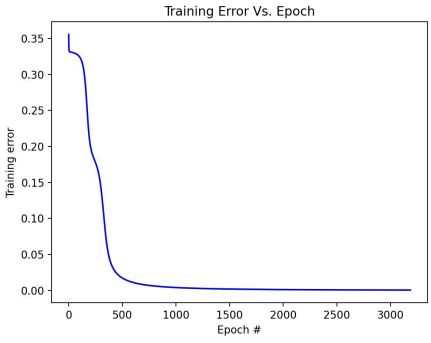
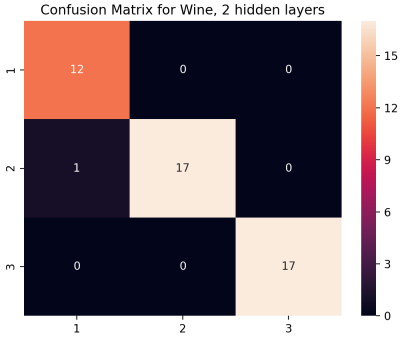


Figure 6: Corresponding confusion matrix

When I tried 2 hidden layers instead of 1, I got fewer trials where my network would converge at 0.3, but I also had far more iterations on some of the trials. With the 2 hidden layer configuration, I found that a learning rate of 0.2, momentum value of 0.9, and error threshold of 0.0001 worked pretty well. While there were fewer trials where the network prematurely converged, sometimes the network would simply not converge until max_epochs were reached. This isn't ideal, but the testing error in these situations was virtually always 0, and I set max_epochs to 10,000 so it was not *too* long to run.





This trial which did converge reached a training error of 0.0007 and had a testing error of 2.1%.

I tried to see if increasing the number of hidden layers to 3 would make any improvement, but performance-wise it seemed about the same.

I wanted to see if using different number of neurons would maybe increase the performance of the models, so I tried another common configuration: ⅔ the size of the input layer plus the size of the output layer, which in my case would be 12.This did not help very much with the 1 hidden layer configuration; I saw similar patterns to the 8 neuron network.

When I tried 12 neurons in the hidden layers for the 2 hidden layer configuration, there was also not much of a performance increase. So then I tried using 12 neurons on the first hidden layer then the original 8 on the second to see what would happen. I also did not see much performance increase with this configuration. For these reasons, if I were to pick a "best" network architecture for this data set I'd probably just use my original 2 layer configuration with a learning rate of 0.2, momentum value of 0.9, and error threshold of 0.0001. I wonder if the data set was larger (178,000 samples rather than 178), whether the behavior could have been any different. I tried tuning my parameters as much as I could but the problem I could not seem to solve was the neural network's lack of convergence; about 60% of the trials would converge properly but 40% would just reach the maximum iterations and terminate. Like I mentioned earlier, this was not a huge problem because the training and testing error was so low by the time the neural net reached 10,000 iterations; but I wonder whether the testing error would have been as low if the data set was larger.

Things I tried that did not work out so well:

The main thing I wish I could have gotten working was my softmax function. Typically for multi-output neural networks, the softmax function is used for the last layer to output the final probabilities. While sigmoid worked fine, I wonder if using softmax may have led to better

performance, since it seems that is what the general world of machine learning tends to use for the output layer. The problem with softmax was that I have trouble trying to understand what exactly is going on in the derivative function. Maybe this is something I will revisit later.