NEURAL NETWORKS

Assignment 2

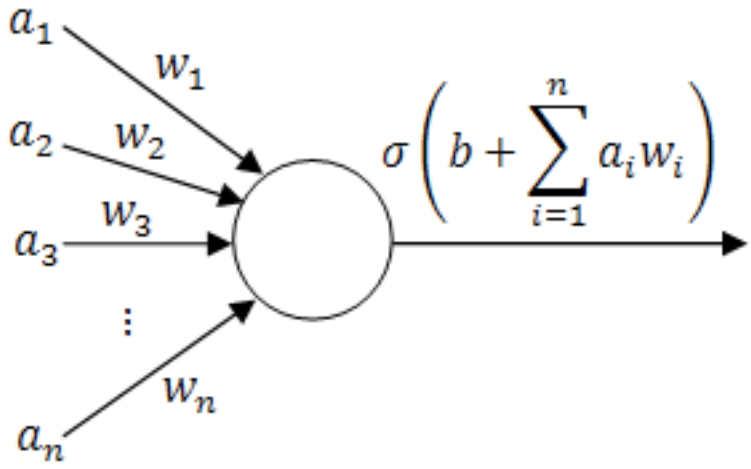
Jihoon Oh

TABLE OF CONTENTS

HOW IT WORKS

PERCEPTRON

Before we dive into how a neural network works, we must first understand how a perceptron works. Much like a single neuron in the brain, a single perceptron is just one of the many components that make up a neural network.



To put it simply, a perceptron takes in various inputs, filters them out, and spits out a single value. However, as you can see in the figure above, the mathematics of outputting a single value can be slightly tedious.

Here is how it works mathematically:

1. **Compute the dot product of the inputs and the weights.** In other words, take all the inputs and their respective weights, multiply them, and sum over all the products. One thing to note here is that the weights are initialized "randomly".
2. **Take the sum and add it with some bias term.** The bias term acts as a measure of how to get the perceptron to fire. For the purpose of this project, the bias is treated as extra weight because it needs to be updated when implementing the learning algorithm.
3. **Pass the results from number 1 and 2 through a sigmoid function.** Unlike a step function, the sigmoid function is ideal for this filtering process because it is smooth and the range values go from (-1,1). Furthermore, instead of having discrete values, such as a 0 or 1, it can be any rational value that falls under the range.

INITIALIZE WEIGHTS

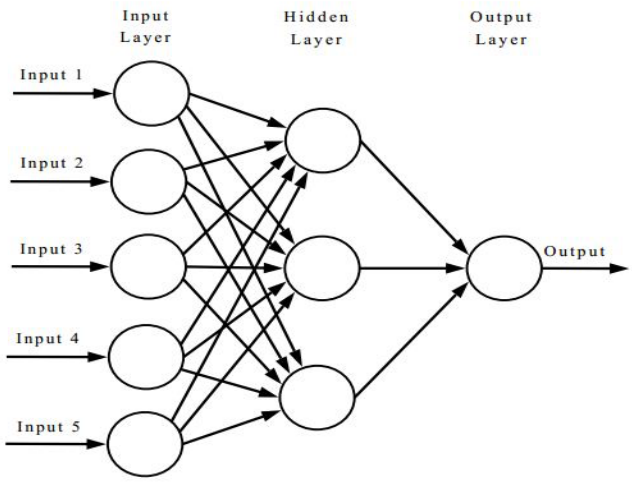
While there are a lot of ways to initialize weights, the most naïve approach would be to set them all to 0. This is particularly a dangerous approach because all the neurons will have the same gradient when we implement the learning algorithm.

After doing some research, I decided to use the Xavier initialization method (link here). The Xavier initialization sets the mean of the weight distribution to 0 and the variance to 1/n, where n is the number of input nodes.

The bias vectors were set up the same way, using the same initialization method.

NEURAL NETWORKS

As flexible as the perceptron is, there is one crucial disadvantage: they can't learn XOR and parity functions in general. To resolve this issue, one must use a collection of perceptron to create a neural network.



As you can see in the figure above, we create a neural network by attaching an extra perceptron to our original perceptron. It goes from the input data, passing through the weights, and outputs a value for the hidden layer. Once we have all the hidden values, we pass it through the weight again and obtain a single value for the output. However, if it just follows this process, the weights stay constant and don't learn anything. In order to dynamically change the weights, we implement a learning method called the "backpropagation algorithm".

Without going into too much detail, the backpropagation algorithm uses gradient descent to readjust the weights. For example, given some input data, the neural network initially passes through and spits out a single value output. We obtain an error value by comparing this value to the true value, and readjust the weights until this error value passes a given threshold. For testing however, we don't train the weights, so we just examine the output error values.

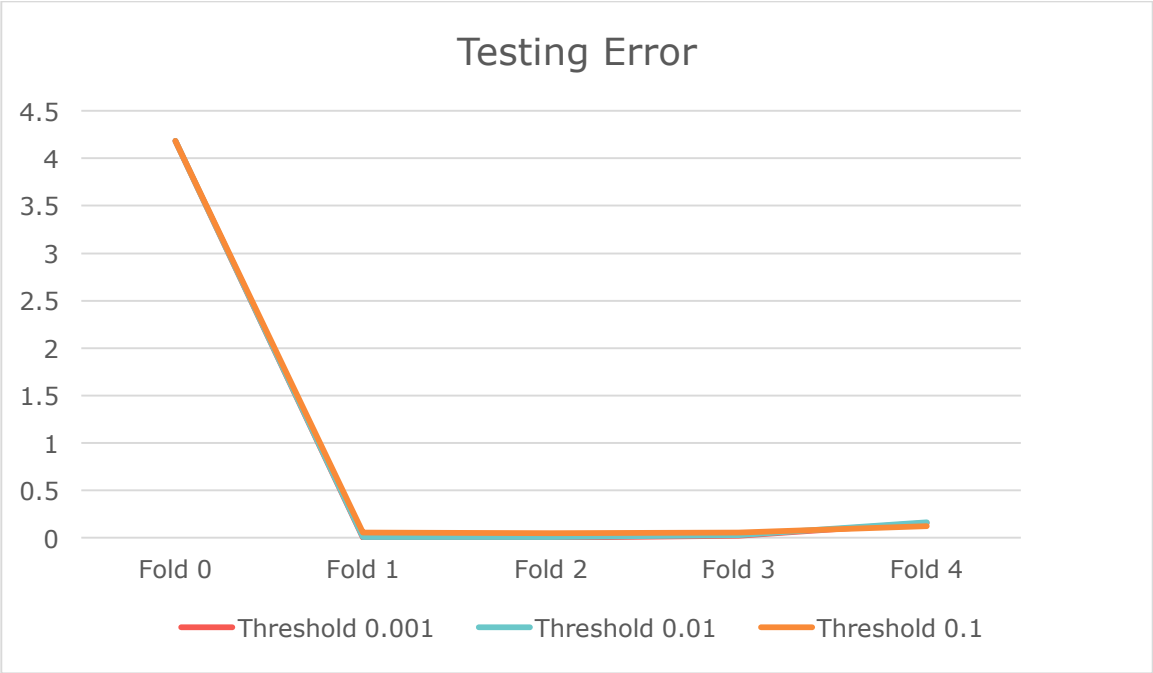
ANALYSIS

5-FOLD CROSS VALIDATION

There are essentially two subjects to analyze: 5-fold cross validation and how the threshold changes the number of epochs. I've organized the process in the following way:

	1ST	2ND	3RD	4TH	5TH
Fold 0	Test	Train	Train	Train	Train
Fold 1	Train	Test	Train	Train	Train
Fold 2	Train	Train	Test	Train	Train
Fold 3	Train	Train	Train	Test	Train
Fold 4	Train	Train	Train	Train	Test

As you can see, as we increase the number of folds, the later the testing phase. This way, we can see the intuition behind the effect that the training phase has on the testing phase. Intuitively, we would expect that as you do the testing in the later phase (i.e., as we do more training), the more accurate the outcome.

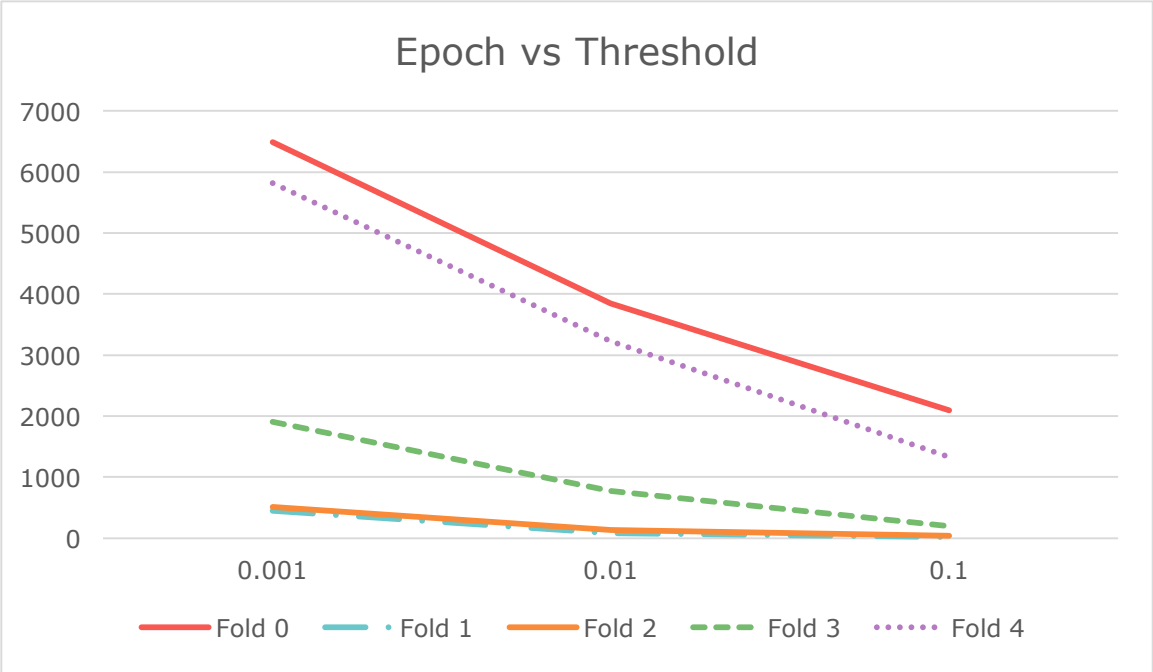


The figure above shows that the error output follows a general trend with respect to the number of folds, regardless of which threshold we use. Furthermore, the testing error for fold 0 is exponentially higher than the other folds, which is to be expected because the neural network does not perform any weight training in fold 0.

What's more interesting, however, is how the error tends to rise after fold 2, forming a local minimum at that point. This shows that there has to be a balance between how much we train. If we train too little, the error is exponentially off, but if we train too much, the error is still off by a couple figures.

EPOCHS VS THRESHOLD

Essentially, the threshold tells the program when it should stop the training. Ideally, this threshold should be at 0 because we don't want any error, but this is takes too long to actually perform in reality.



The graph shouldn't be too surprising of a result. As we increase the threshold, we're allowing the training to stop sooner, so the number of epochs should decrease. As you can see, the number of epochs does decrease as the threshold increases, regardless of what fold we are using.

THRESHOLD VS TOTAL ERROR

This is probably the most imperative analysis portion for this assignment. The question I asked myself while doing the assignment was: how much can the threshold actually affect the total error

Here is a table that maps thresholds against the error:

	FOLD 0	FOLD 1	FOLD 2	FOLD 3	FOLD 4
0.001	4.1845	0.0034	0.0031	0.0208	0.1557
0.01	4.1845	0.0128	0.0072	0.02844	0.1659
0.1	4.1845	0.0560	0.0478	0.0597	0.1259

The table shows that as you decrease the threshold, the less the error output. This makes sense because if you decrease the threshold, you allow more training to happen in the backpropagation algorithm.

To represent this discrepancy graphically, I've also created the following bar graph: