EE 456: Artificial Neural Networks

November 9, 2022

Sameer Khan

**Overview:**

The objective of the project is to implement the back propagation algorithm in a multilayer perceptron network. This would allow the neural network to learn and ultimately distinguish between the two classifications of moons present in the sample data which are non-linearly separable.

**Implementation:**

To implement this back propagation algorithm, first I decided to shuffle the data set to remove all biases from clumps of feature data. I did this by taking advantage of a permutation function in MATLAB. After the data is shuffled, I split the data into the training set and validation set. I chose to make the training set 4 times the validation set as per good practice and have had no issues with this. Next, I setup all of the variables that will be needed to accomplish the algorithm. The threshold and learning rate were given in the instructions and the weights were initialized to random small values. There are also variables created to hold intermediate values like the neural network size, different error measurements, and other miscellaneous values for readability.

Moving into the main loop of the algorithm, the stopping condition is one of the most tunable aspects of the code. I decided to count iterations of the algorithm by epochs and therefore used a max epoch value to control when the algorithm stops. After playing around with very small values while testing the functionality of the back propagation, I found that values near 1000 allowed for the error to settle into steady state. Thus, I chose 2000 as a max epoch to account for any outliers to this trend and allow for time to guarantee a minimized error. Every epoch, I loop through all inputs of the training set. Each loop of a single input can be broken down into a few key steps: feed forward, back propagation from output to hidden nodes, back propagation from hidden to input nodes, weight updating, and calculating error.

Feed forward: The feed forward step is all covered by a single function, feedforward.m, in my implementation. This encapsulation allowed for more readable and organized code and took care of the reimplantation of a process that is used often in the code. This pattern is followed with a coupe other step as well. The feed forward step is that same as other perceptron feed forward cases we have done. The input is fed into the system, multiplied by the weights and acted upon by a activation function to get output values for each hidden node. Then this process is repeated from the hidden node outputs to the output node and a value for the output of the system is procured. The function outputs all needed values that future steps may need including output node output, output node input, hidden node outputs, and hidden node inputs.

Back Propagation from Output to Hidden: This section is also encapsulated in a function called backPropagationOutput.m. This function handles the weight change calculations for the wights between the hidden nodes and output node. This is calculated by finding the error between the calculated output from the feed forward function and the target output and multiplying it by the derivative of the activation function of output node input y\_in. After calculating this error δk, we can use the value to update all the weights by using the equation where is the learning rate and Z\_j is the output from hidden node j. These weight changes are then output to the main loop to be added to the current weights along with some other useful values.

Back Propagation from Hidden to Input: This step is similar to the previous step and is encapsulated in the backPropagationHidden.m function. For each hidden node, we multiply the delta error from the previous back propagation step with the weight associated with those two nodes form the previous step. This value is the input delta error for this step of back propagation. We multiply this input delta error to the hidden node by the derivative of the transfer function to get our delta error for that hidden node . Then we use the equation where x is the input to the hidden node. This will give us the final set of weight changes so we can update all of our weights

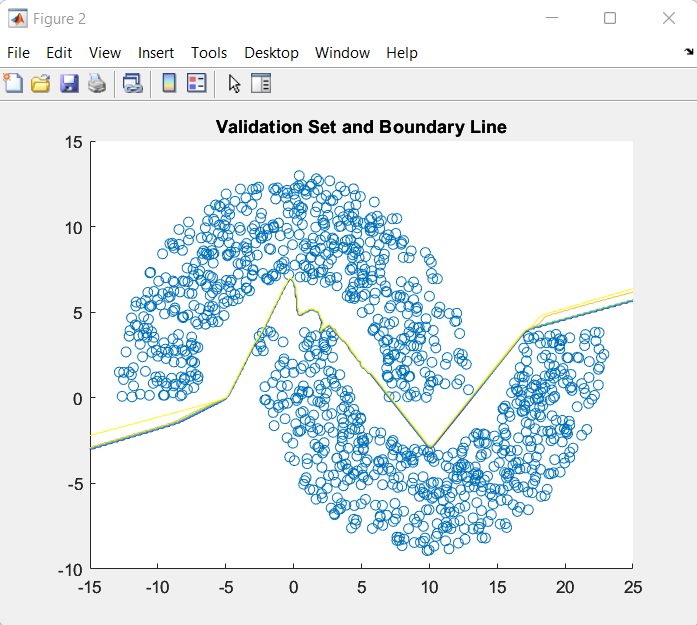
Weight Update: The next step is to take these weight changes from the back propagation steps and add them to our current weight values to get a new update weight values for the next iteration of the loop.

Error Calculation: To calculate the error at any step, the function calculateError.m takes in the given output and target output and delivers the instantaneous error energy between these values. These are used to create the error graphs and to give the user a sense of the effectiveness of the code.

After 4 epochs of these phases are completed on the training data, I run one feed forward iteration of all validation data inputs and calculate the error for these to get a measure of how effective the training has been on non-training values. Also, at the end of every 500 epochs, I decrease the learning rate by a factor of 10 to reduce learning and clue in on a steady state error.

Finally, after the stopping condition is met, I graph the error of the training data and validation data vs epochs to give a sense of the decreasing error over time. I also plot the validation test data and create a contour plot of the mesh grid of the neural network to get a boundary line approximation representative of the calculated weights. An error rate is output to the command window calculated by the number of incorrectly classed inputs vs the total number of inputs.

**Results:**

Data Set 1 Results:

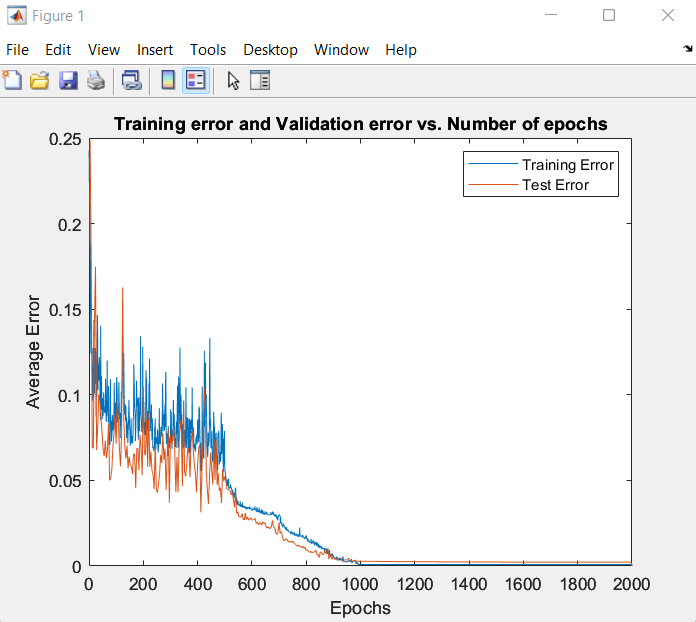
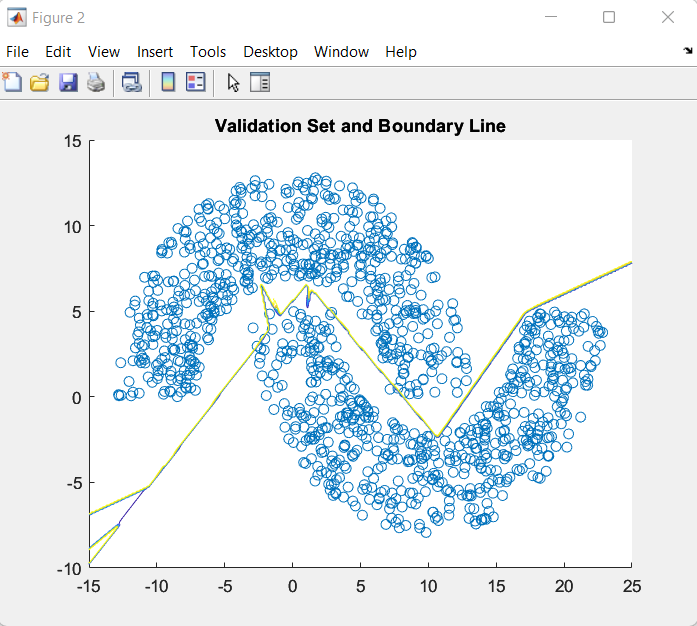


Figure 1. Training and validation error over epochs for data set 1 Figure 2. Boundary line calculated based on weights after training is

done for data set 1

Overall Error Rate: 8.333333e-04

Data Set 2 Results:

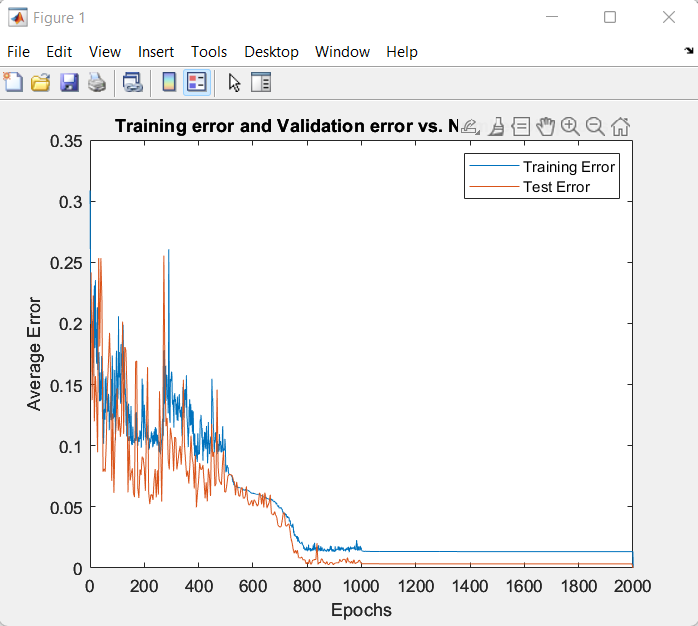


Figure 3. Training and validation error over epochs for data set 2 Figure 4. Boundary line calculated based on weights after training is

done for data set 2

Overall Error Rate: 1.666667e-03

Evaluation:

In general, the results of the algorithm are very positive. In Figures 1 and 3, a clear decrease in the error of the network is always seen and tend to round out towards a small number close to 0. On occasion the error may settle to a larger number, but this is probably due to the random shuffle of inputs at the beginning having some sort of clump of similar inputs or bias that was not removed. You do seem some oscillation in these figures, but this could be due to the algorithm overcorrecting itself on its way down due to a large learning rate. A smoothly decreasing learning rate my remove these giant oscillations but overall would not affect the final error value found.

When looking at Figures 2 and 4, we see the decision boundary created by using the contour plot function. This function maps out contours on a 3D surface created by the mesh grid function and the neural network weights. The multicolored bands come from the contour function creating contours at multiple levels of the 3D surface and I was unable to figure out how to only keep the most accurate one. Although, usually, the multiple contour lines all overlay each other so it does not cause much of an issue. The boundary lines seem to be very accurate shown by the Overall Error Rate for each data set but also do not appear to be smooth as I expected a perfect boundary line to be. This could probably be attributed to the number of weights or hidden nodes in the network. I hypothesize that the more weights and nodes we have, the more ability the network’s boundary line has to bend and turn and the limit of only 20 hidden nodes creates a boundary line with a proportional number of turns indicated in the graph.

In general, the number of errors in the system never exceeded 70 mistakes even in the worst cases and averaged close to the numbers seen above. This would allow for a network that was always over 90% correct which I believe to be very good for the data set given and the number of nodes we have. Also, an observation I made while creating this project was that the back propagation algorithm seemed to run faster computationally , make a more accurate decision boundaries, and take less epochs than any other method we have coded so far making this a favorite in my opinion.