Neural Network Lab, May 2017

Name: Shreyas Angara
Period: 7
Date Submitted: 5/8/17

Instructions: Complete the parts below. You may leave the instructions in tact or delete them but please leave the 4 parts clearly separate. Submit final version on dropbox, as indicated on BB.

Part 1:
Using the 14 linearly separable Boolean functions on 2 variables, how many 2-2-1 networks can you construct to calculate XOR? List all the solutions you find. How many unique solutions exists if you consider symmetry (e.g. ABC is just a mirror image of BAC)? Describe how your algorithm works.

(Note to list the solutions you should provide a numbered list of the 14 functions and their weights, as you learned them in your previous lab. Then solutions should take the form (x,y,z) where (x,y,z) is a triple of numbers identifying the functions.)

Functions	Solutions
0. [0, 0, 0]	**1.** 1 6 12
1. [2, 1, -2]	**2.** 1 12 6 – not unique
2. [1, -2, 0]	**3.** 2 6 1
3. [1, 0, 0]	**4.** 2 12 7
4. [-1, 1, 0]	**5.** 4 1 6
5. [0, 2, 0]	**6.** 4 7 12
6. [1, 1, 0]	**7.** 6 2 4
7. [-1, -1, 1]	**8.** 6 4 2 – not unique
8. [0, -2, 1]	**9.** 7 1 7
9. [1, -1, 1]	**10.** 7 7 1 – not unique
10. [-1, 0, 1]	**11.** 9 2 9
11. [-1, 2, 1]	**12.** 9 4 11
12. [-2, -1, 3]	**13.** 11 9 2
13. [0, 0, 1]]	**14.** 11 11 4
	15. 12 9 11
	16. 12 11 9 – not unique

After considering symmetry, there are 12 unique solutions which produce the XOR function.

My algorithm first produces a list of the 14 learnable functions and then converts these functions to perceptrons. This is done by using the first two values of the vector as the weights of the perceptron and the negative of the third value as the threshold. Using this list of perceptrons, I used three nested for loops to create every combination of those 14 perceptrons. For each combination, I made two perceptron's inputs an Input object and for the other perceptron, I set the inputs as the other two perceptrons. After running my evaluate function on each set of inputs, I added them to a list to create an output (ex. [0,0,1,1]). I then checked if my output equaled the XOR output ([0,1,1,0]) and increased the correct count if it did.

Part 2:

Construct a NN using any topology you like to recognize coordinates within the unit circle. Implement your best model in code and compute its accuracy over 10,000 randomly selected points within the square -3/2 <= x,y <= 3/2. Describe your best model and report its accuracy.

My best model topology is essentially a square consisting of four perceptrons. The four perceptrons form the inequalities: x>-1, x<1, y>-1, y<1. Two of these perceptrons feed into an and gate and the other two feed into another and gate. Lastly, these two and gates feed into a final gate. The topology uses the sigmoid function as the actuator function instead of a step function. This model yielded an accuracy of approximately 96.5%.

Describe the first NN you came up with and how your design evolved into its final form. If your best model uses sigmoid actuator functions, what is the optimal "k" and "c" values you found (exponential coefficient, and rounding threshold). If your best model uses step actuators, how do you design the nodes and find their weights? Discuss your implementation challenges and how you determined your best model.

After creating my square topology, I initially attempted to use the step function we had used for earlier perceptron labs. This was extremely inaccurate as the step function can only yield 0 or 1. I then replaced the step function with the sigmoid actuator function. I was able to attain the maximum accuracy when using a k=4.65 and c=0.5. Attain the optimal k was difficult as it was essentially trial and error. I was finding that moving away from around 5 yielded lower accuracies in either direction so after experimenting around 5, I attained the optimal value of 4.65.

Part 3:

Implement an ad-hoc learning algorithm to discover the weights in a 2-2-1 network topology for XOR, using sigmoid actuator functions. Show the final weights learned by your algorithm (rounded and formatted nicely). Write pseudocode that clearly describes how your search algorithm works. Also discuss how your ideas evolved and any failed attempts along the way.

Sample Final Weights

A: [-12.067, 3.578, 1.536]

B: [-11.458, 40.713, -2.868]

C: [27.732, -25.523, -3.599]

Pseudocode:

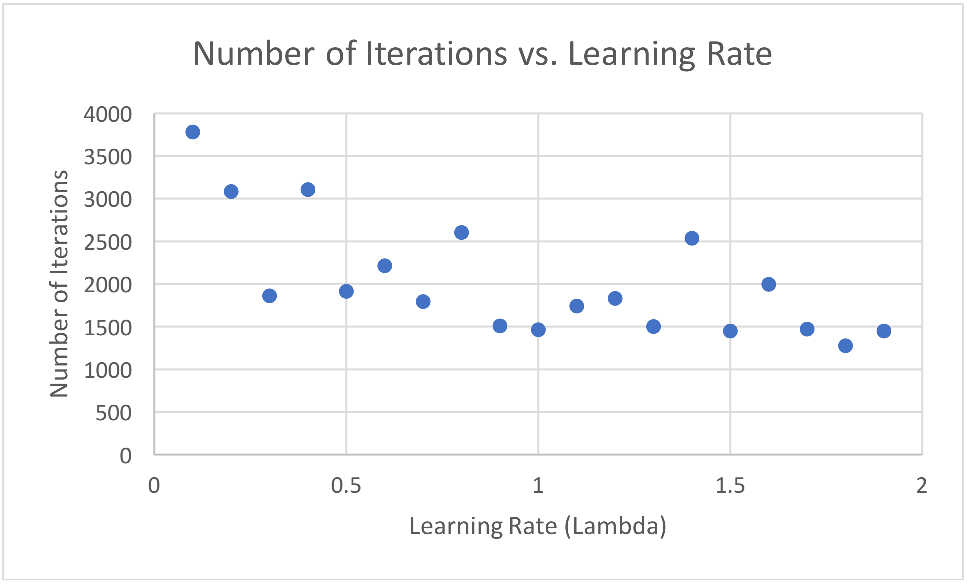
```
    W = 9 random weights from -1 to 1
    Output = [0,1,1,0] (XOR)
    Target = any decimal < 0.01
    Lambda = any integer (I used 30 for best results)
```

```
Direction = 9 random weights from -1 to 1 to determine direction
While error of current W > target:
    Delta = 9 random weights times the direction and lambda
    Temp = current W + delta
    If error of temp<error of current W:
        Temp becomes the current W
    Else:
        Direction = new 9 random weights
    If the while loop has occurred more than 500 times:
        Reset the W
Return the weights found
```

Along the way, the main problem I was finding at first is I never set a direction vector so my search would go random directions. After adding the direction vector, I would find that my search would sometime never converge on one weight vector. Finally, I added a reset check inside my while loop which allowed to exit if I never converged on one weight vector. This allowed me to produce efficient results.

Analysis: Explore the convergence rate of your algorithm as a function of the learning rate. Count the number of iterations required for your algorithm to converge, averaged over several samples (say 10-100 per trial), as you vary the learning rate, lambda, and produce a graph. The appropriate range of lambda will depend on your implementation, but generally a lambda of "1" means you add "1" to the weights at each step. Find the approximate lambda which seems to minimize convergence time.

Describe how your algorithm depends on lambda and show a graph of "lambda vs. iterations".



The graph shown above, although with major exceptions, generally shows an exponential decay relationship between lambda and the number of iterations needed to converge. This makes sense because in my algorithm, lambda essentially determines how large my delta vector is so if lambda is higher, the new weight vector I test will be further away from the old vector than if I used a smaller lambda. This would allow less iterations to converge on the correct weights. If lambda gets too large, however, the new weights skip too far ahead and cannot converge on the correct weight vector. The lambda that gave me the best results was 30.

Part 4: (Bonus/Extension)

Modify your learning algorithm (or create a new one) to find optimal weights for part 2. You may also want to include “k” and “c” in your search space. Describe your results.