Neural Network Lab, May 2017

Name: Justin Cai
Period: 7
Date Submitted: 5/15/17

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.

14 learnable functions:
1: 0.06186099, -0.28749345, -2.19744232
2: 2.44767166, 2.41634341, -3.64917374
3: 2.60235036, -2.80760067, -1.32932935
4: 3.27474454, -0.13019885, -1.49150723
5: -2.80642197, 2.60060178, -1.3282882
6: -0.1101527 , 3.28004905, -1.50953467
7: 2.63978945, 2.66350476, -1.1560097
8: -2.60393719, -2.62850047, 1.13733015
9: 0.13184021, -3.26781112, 1.48975626
10: 2.7980563 , -2.58542988, 1.31865877
11: -3.26613455, 0.14001065, 1.48104858
12: -2.59101583, 2.80539858, 1.32091771
13: -2.43916856, -2.40735248, 3.6352416
14: 0.84829078, 0.50359994, 1.79496372

16 solutions

1. (1, 6, 4)
2. (1, 7, 7)
3. (2, 4, 6)
4. (2, 9, 9)
5. (4, 2, 6)
6. (4, 11, 9)
7. (6, 1, 2)
8. (6, 12, 1)
9. (7, 1, 7)
10. (7, 12, 4)
11. (9, 2, 11)

12. (9, 11, 12)
13. (11, 4, 11)
14. (11, 9, 12)
15. (12, 6, 1)
16. (12, 7, 2)

12 unique solutions

1. (1, 6, 4)
2. (1, 7, 7)
3. (2, 4, 6)
4. (2, 9, 9)
5. (4, 11, 9)
6. (6, 1, 2)
7. (6, 12, 1)
8. (7, 12, 4)
9. (9, 2, 11)
10. (9, 11, 12)
11. (11, 4, 11)
12. (12, 7, 2)

My algorithm first finds the 14 learnable functions, then instantiates a variable that contains the solution (for XOR, [0, 1, 1, 0]). I then create a for loop inside a for loop for every single combination of two of the learnable functions, then another for loop inside that to test if the combination works for XOR using an int function in place of an Input class/eval function. If it does, I check if the solution is unique or not.

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.

Accuracy: 97.66%
In part 2, I figured out that the int function wouldn't work like it did for part 1, so I made an Input class that essentially stores a value/values and returns the value when I need it to. I used the following weights/thresholds:

```
left = p.percy([1, 0], -1.5)
right = p.percy([-1, 0], -1.5)
bottom = p.percy([0, 1], -1.5)
top = p.percy([0, -1], -1.5)
side = p.percy([1, 1], 1.5)
tb = p.percy([1, 1], 1.5)
circ = p.percy([1, 1], 1.5)
```

where I set tb's inputs to top and bottom, side's inputs to left and right, and circle's inputs to side and tb. I set left, right, top, and bottom's inputs to two Input variables that I instantiated earlier. I then made a for loop that looped 10000 times and each time set the two Input variables to random values between -3/2 and 3/2, and called an eval function that used a sigmoid function in which the k value was 3. My c value was .53.

The first NN I tried to do kept the original int function instead of an Input function, and when I realized I needed it to return decimals instead of integers and was met with AttributeError, I decided to create an Input class instead and write an eval function that was similar to the original int function, but it called my sigmoid actuator function, where k was 3, and returned that value each time instead of returning 1 or 0. My c value was .53. My implementation challenges would definitely be at first trying to figure out how I could keep the int function instead of having to write an Input class and modify my Perceptron class, but I determined that an Input class would be best instead of having to work around the challenges of returning either a 1 or 0 each time. To figure out my best c value I essentially just guess-and-checked.

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.

Final weights: [9.47, -8.10, 2.52 , 16.59, -17.39, -15.78, 5.68, -4.09, -2.05]

```
Goal = [0, 1, 1, 0]
Weights = [random, random, random, random, random, random, random, random, random]
while(program is not done aka error > .01):
    directions = [random, random, random, random, random, random, random, random, random]
    x = error
    temp weights = weights + directions * lambda
    if x > error of temp weights:
        for range(100):
            if program is not done:
                temp weights += directions * lambda
                x = error of temp weights
                if x < .01: (aka if error is really really small)
                    program is done
                    print iterations
```

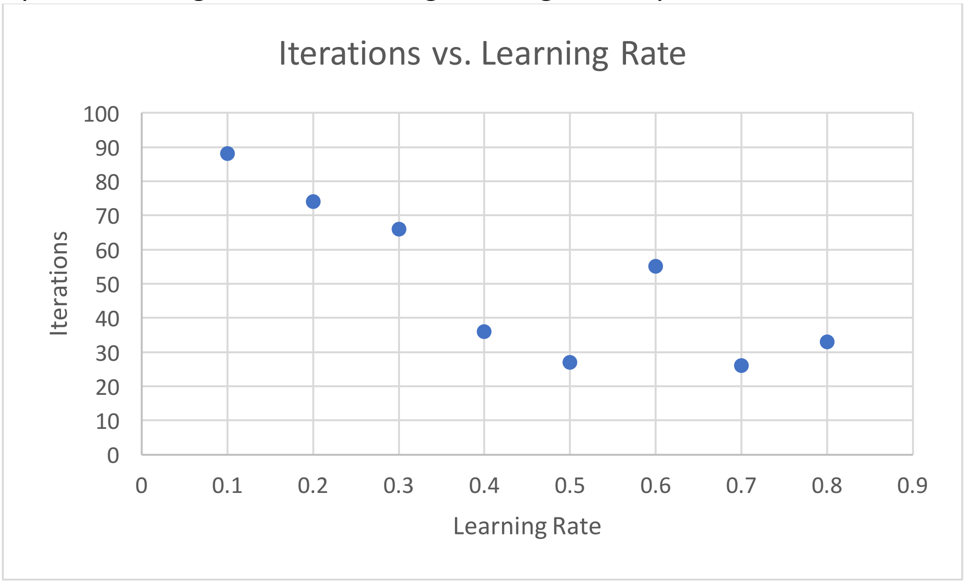
Originally I tried to do this code without having an error method and just manually run the error method every time, but I figured out that that way was very inefficient and wrote helper methods instead. Other than that, my ideas were pretty much like my pseudocode from the start, except I wasn't quite sure what lambda or the learning rate should be. As for failed attempts, for a long time my code wouldn't work and I didn't figure out that it was because the random values in the weights/directions arrays weren't from -1 to 1 but rather from 0 to 1. That wasted a lot of class time where I just sat there staring at my computer wondering why my code was infinite looping and not doing anything. As for optimal lambda, I basically just guess and checked at first, but I noticed there wasn't any one value that produced consistently low iterations, so I just stuck with .5 as my lambda.

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.

It seems a lambda of .5 or .7 minimizes convergence time.

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

I used learning rates of .1 to .8, with intervals of .1, testing how many iterations it would take for each learning rate. For some reason, any learning rate from .9 onwards wouldn't run properly, giving me errors in my sigmoid function. My algorithm clearly depends on lambda, as some lambdas gave me a large amount of iterations whereas others, such as .5 or .7, gave me small iterations. Lambda in my algorithm is the factor by which the direction array is multiplied by before being added to the original weights array.



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.