Vivian Liu and Samia Menon

Art of Engineering Department Project

Prof. Ali Hirsa

December 28th, 2019

Findings from "A Visual Display of Optimization & AI"

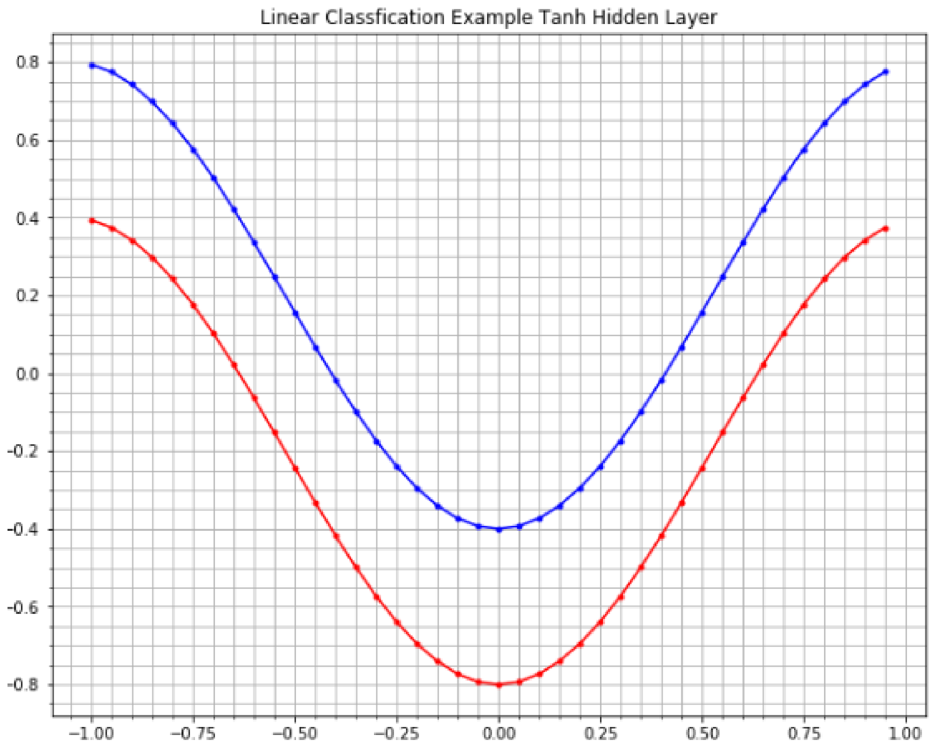
i. Introduction

In this project, our task was to take the following curves:

$$y_1 = -0.6\sin\left(\frac{\pi}{2} + 3x\right) - 0.20$$

$$y_2 = -0.6\sin\left(\frac{\pi}{2} + 3x\right) - 0.20$$

For reference, these are what the curves look like:



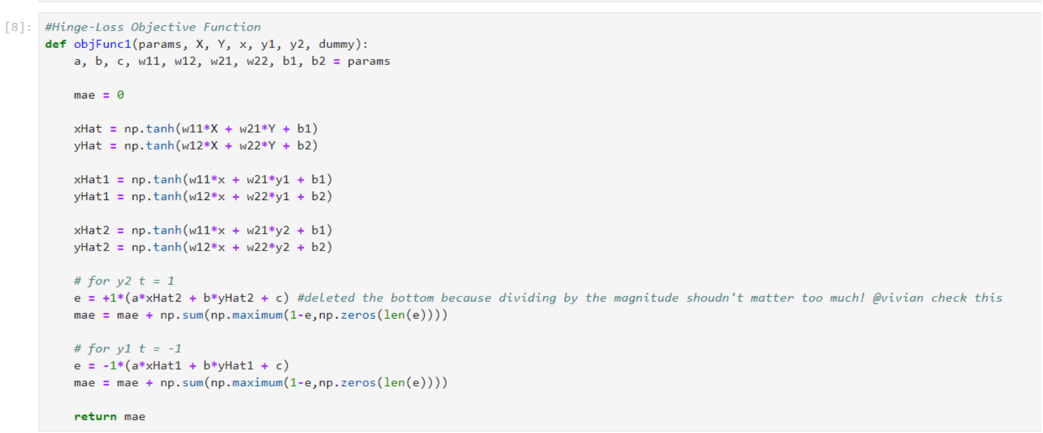
The next step is to transform the space (Ω) onto which they are mapped via the given equations in the project, and then take these transformed curves and find parameters for a "line of best fit" that is drawn directly between them. To do this we define a vector **w** so that:

$$\boldsymbol{w}^T\eta \geq 0 \; when \; t = \; +1$$

$$\boldsymbol{w}^T\eta < \; 0 \; when \; t = \; -1$$

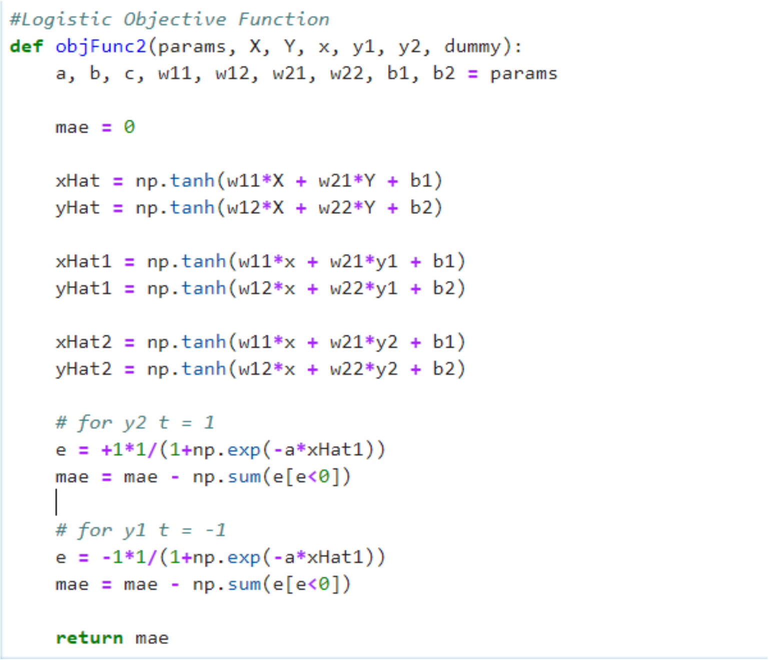
Finally, we minimize the value of the sum of $\mathbf{w}^T\eta$ across all of the misclassified inputs. As there is less of the line over the top transformed curve and under the bottom transformed curve this value will decrease along with the amount of error. There are infinitely many ways to define an objective function that will represent the error we seek to minimize. In this project, we decided to use a *hinge-loss* objective function and a *logistic* objective function.

This is our code for our hinge-loss function:



```
[8]: #Hinge-Loss Objective Function
     def objFunc1(params, X, Y, x, y1, y2, dummy):
         a, b, c, w11, w12, w21, w22, b1, b2 = params

         mae = 0

         xHat = np.tanh(w11*X + w21*Y + b1)
         yHat = np.tanh(w12*X + w22*Y + b2)

         xHat1 = np.tanh(w11*x + w21*y1 + b1)
         yHat1 = np.tanh(w12*x + w22*y1 + b2)

         xHat2 = np.tanh(w11*x + w21*y2 + b1)
         yHat2 = np.tanh(w12*x + w22*y2 + b2)

         # for y2 t = 1
         e = +1*(a*xHat2 + b*yHat2 + c) #deleted the bottom because dividing by the magnitude shoudn't matter too much! @vivian check this
         mae = mae + np.sum(np.maximum(1-e,np.zeros(len(e))))

         # for y1 t = -1
         e = -1*(a*xHat1 + b*yHat1 + c)
         mae = mae + np.sum(np.maximum(1-e,np.zeros(len(e))))

         return mae
```

This is our code for our logistic function:



```
#Logistic Objective Function
def objFunc2(params, X, Y, x, y1, y2, dummy):
    a, b, c, w11, w12, w21, w22, b1, b2 = params

    mae = 0

    xHat  = np.tanh(w11*X + w21*Y + b1)
    yHat  = np.tanh(w12*X + w22*Y + b2)

    xHat1 = np.tanh(w11*x + w21*y1 + b1)
    yHat1 = np.tanh(w12*x + w22*y1 + b2)

    xHat2 = np.tanh(w11*x + w21*y2 + b1)
    yHat2 = np.tanh(w12*x + w22*y2 + b2)

    # for y2 t = 1
    e = +1*1/(1+np.exp(-a*xHat1))
    mae = mae - np.sum(e[e<0])

    # for y1 t = -1
    e = -1*1/(1+np.exp(-a*xHat1))
    mae = mae - np.sum(e[e<0])

    return mae
```

Note: After research on Hinge-Loss in Python, we are more confident about our Hinge-Loss objective function, but are not as confident about how to take the partial derivatives, thus, our gradient Hinge-Loss function is based off of a few assumptions. For our logistic function, we are less confident about where exactly we put the solenoid activation function, but we are more confident in how we took our partial derivatives.

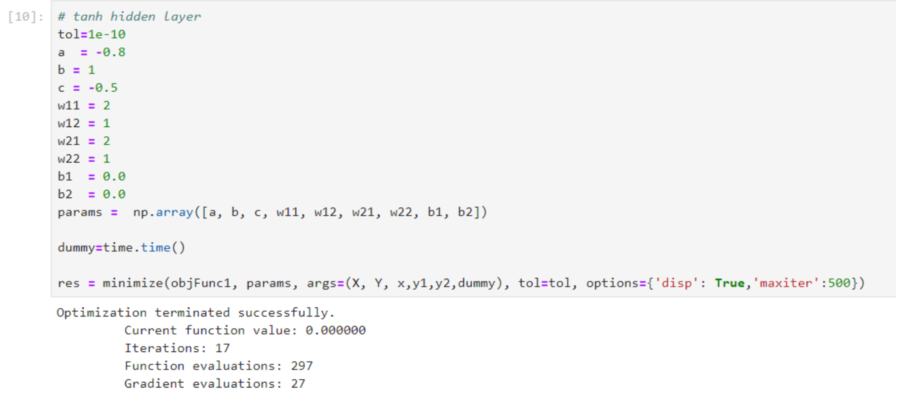
ii) *Findings*

We used two methods of minimizing each function, Simplex and Gradient Descent. For Simplex, we simply used the minimize function available in Python to minimize the objective function, thus, minimizing the error, and then returning the ideal parameter set at that minimum. Using those ideal parameters, we were able to create a Linear Approximation for our transformed equations. For Gradient Descent, we created our own gradient descent function by differentiating the objective functions in respect to each variable, choosing a "rate", and then updating the values for each partial in a loop until it reached the maximum number of iterations. Because the direction of this gradient will always point towards the maxima, we subtract to arrive at the minimum of our objective function, thus also minimizing the error. Then, once more, we return these "ideal" parameters.

We tested each function with three different data sets, labeled Set 1, Set 2, and Set 3.

a) Simplex
   a. Hinge-Loss
      i. Set 1)

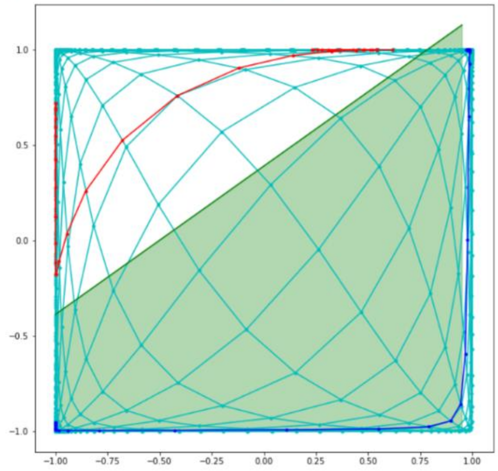
Optimization:



```
[10]: # tanh hidden layer
      tol=1e-10
      a  = -0.8
      b = 1
      c = -0.5
      w11 = 2
      w12 = 1
      w21 = 2
      w22 = 1
      b1  = 0.0
      b2  = 0.0
      params =  np.array([a, b, c, w11, w12, w21, w22, b1, b2])

      dummy=time.time()

      res = minimize(objFunc1, params, args=(X, Y, x,y1,y2,dummy), tol=tol, options={'disp': True,'maxiter':500})

      Optimization terminated successfully.
               Current function value: 0.000000
               Iterations: 17
               Function evaluations: 297
               Gradient evaluations: 27
```

Graph:

```
[13]: <matplotlib.collections.PolyCollection at 0x7f8808f938d0>
```

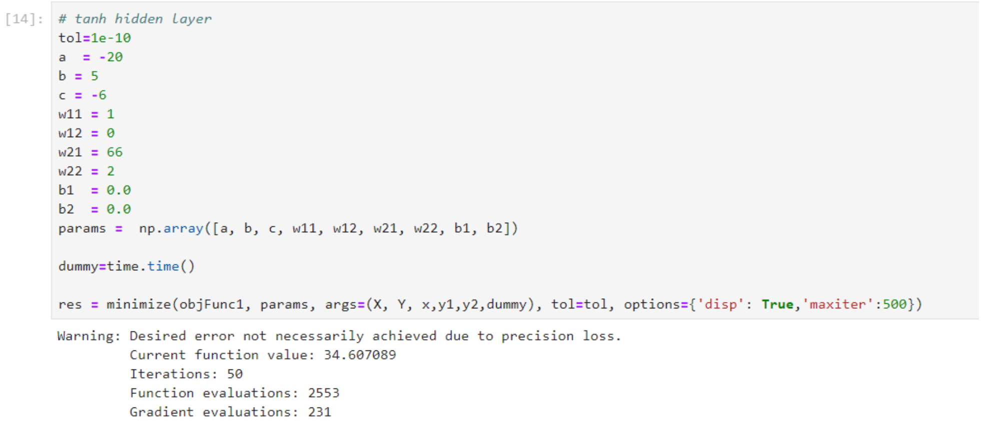


Brief Observations:

Overall, this linear approximation looks quite accurate. The line goes right between the two transformed curves without touching either one, and more or less evenly goes through the middle.

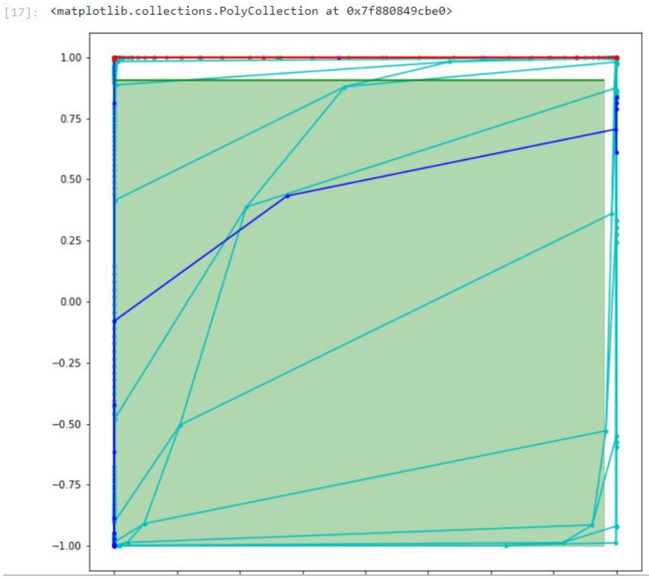
ii. Set 2)

Optimization:



```
[14]: # tanh hidden layer
      tol=1e-10
      a  = -20
      b = 5
      c = -6
      w11 = 1
      w12 = 0
      w21 = 66
      w22 = 2
      b1  = 0.0
      b2  = 0.0
      params =  np.array([a, b, c, w11, w12, w21, w22, b1, b2])

      dummy=time.time()

      res = minimize(objFunc1, params, args=(X, Y, x,y1,y2,dummy), tol=tol, options={'disp': True,'maxiter':500})

      Warning: Desired error not necessarily achieved due to precision loss.
               Current function value: 34.607089
               Iterations: 50
               Function evaluations: 2553
               Gradient evaluations: 231
```

Graph:



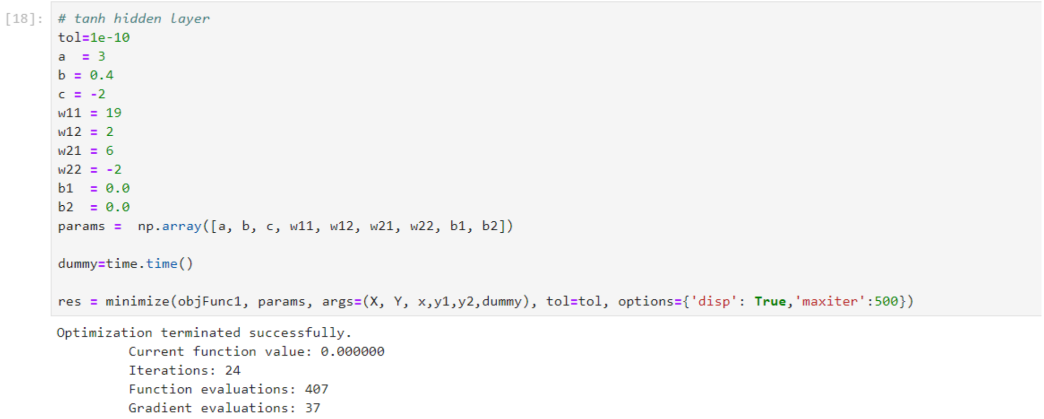
```
[17]: <matplotlib.collections.PolyCollection at 0x7f880849cbe0>
```

Brief Observations:

Although our line does go between the two transformed curves, it is clear to see it is not as accurate as Set 1. This is likely because the Set 2 starting points varied wildly from Set 1 and were farther away from the ideal parameters. In addition, the minima this function found was 35, which is significantly larger than the minima of 0 found earlier, supporting the fact that this approximation is not as accurate.

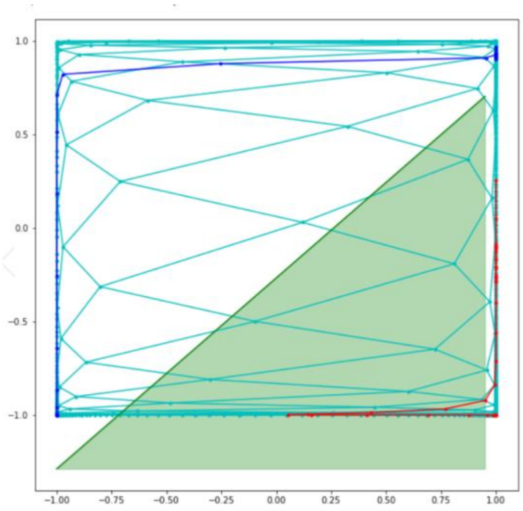
iii. Set 3)

Optimization:



```
[18]: # tanh hidden layer
      tol=1e-10
      a  = 3
      b = 0.4
      c = -2
      w11 = 19
      w12 = 2
      w21 = 6
      w22 = -2
      b1  = 0.0
      b2  = 0.0
      params =  np.array([a, b, c, w11, w12, w21, w22, b1, b2])

      dummy=time.time()

      res = minimize(objFunc1, params, args=(X, Y, x,y1,y2,dummy), tol=tol, options={'disp': True,'maxiter':500})

      Optimization terminated successfully.
               Current function value: 0.000000
               Iterations: 24
               Function evaluations: 407
               Gradient evaluations: 37
```

Graph:



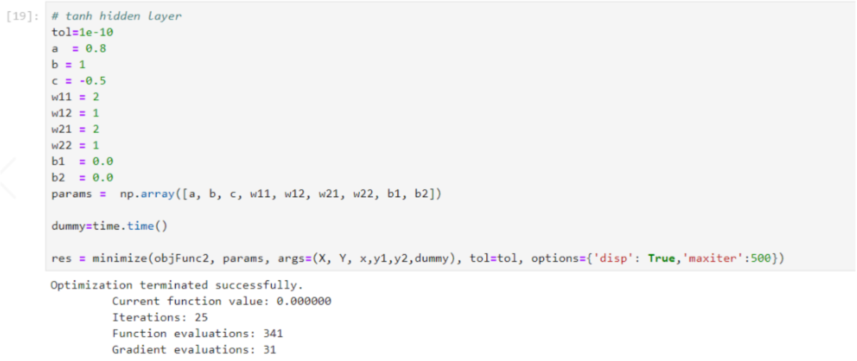
Brief Observations:

This approximation seems to have a similar accuracy to the one we discovered in Set 1. The line clearly goes between the two transformed curves, and the minimum function value found was 0, which ties with Set 1 for the lowest value of the minima, and thus, the lowest error.

b. Logistic

i. Set 1)

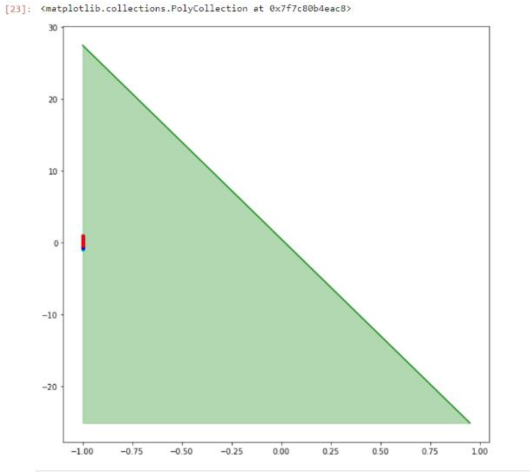
Optimization:



```
[19]: # tanh hidden layer
      tol=1e-10
      a  = 0.8
      b = 1
      c = -0.5
      w11 = 2
      w12 = 1
      w21 = 2
      w22 = 1
      b1  = 0.0
      b2  = 0.0
      params =  np.array([a, b, c, w11, w12, w21, w22, b1, b2])

      dummy=time.time()

      res = minimize(objFunc2, params, args=(X, Y, x,y1,y2,dummy), tol=tol, options={'disp': True,'maxiter':500})
```

```
      Optimization terminated successfully.
               Current function value: 0.000000
               Iterations: 25
               Function evaluations: 341
               Gradient evaluations: 31
```

Graph:



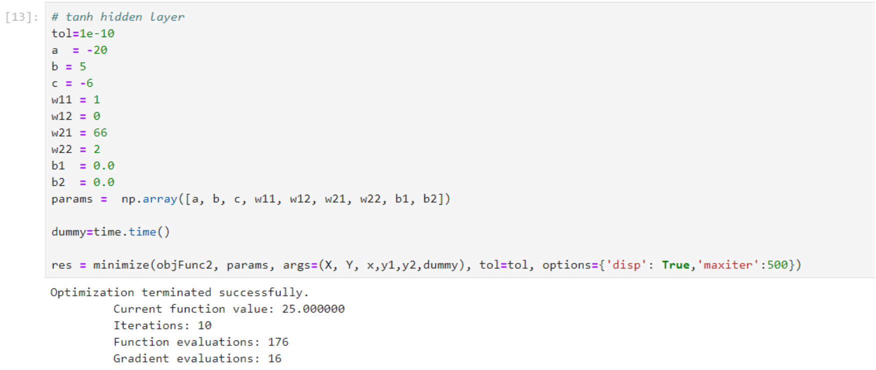
```
[23]: <matplotlib.collections.PolyCollection at 0x7f7c80b4eac8>
```

Brief Observations:

Our optimization process seems to make sense (we landed on a minima of zero, similar to the Set 1 findings in our Hinge-Loss function), but our graph (in particular, the transformed curves) seem like the scale did not work properly. The line itself does seem to make sense, however, relative to the new curves' placement, as a median line. This graph would seem to make more sense if the transformed curves were scaled to cover this field.

ii. Set 2)

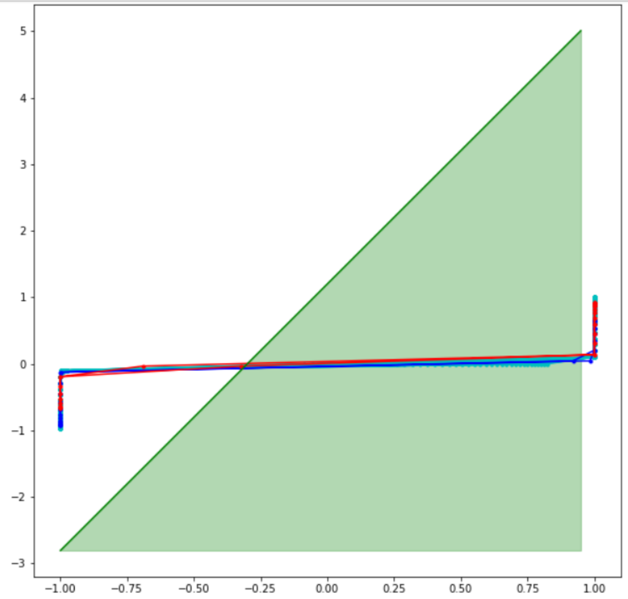
Optimization:



```
[13]: # tanh hidden layer
      tol=1e-10
      a  = -20
      b = 5
      c = -6
      w11 = 1
      w12 = 0
      w21 = 66
      w22 = 2
      b1  = 0.0
      b2  = 0.0
      params =  np.array([a, b, c, w11, w12, w21, w22, b1, b2])

      dummy=time.time()

      res = minimize(objFunc2, params, args=(X, Y, x,y1,y2,dummy), tol=tol, options={'disp': True,'maxiter':500})
      Optimization terminated successfully.
               Current function value: 25.000000
               Iterations: 10
               Function evaluations: 176
               Gradient evaluations: 16
```

:

Graph:

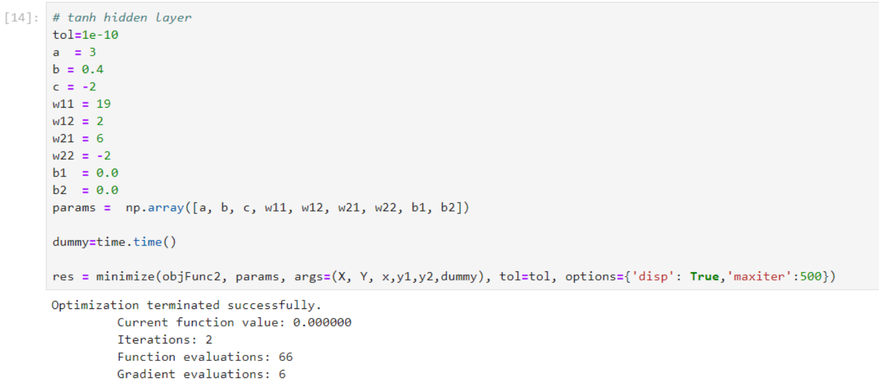


Brief Observations:

Once again, the optimization process seems to make sense – landing on a value of 25 which is less than the value that Hinge Loss found with Step 2. However, the actual transformation of the curves is having some problem. The direction of the line seems accurate, the bending of the plane on which the curves are mapped does not.

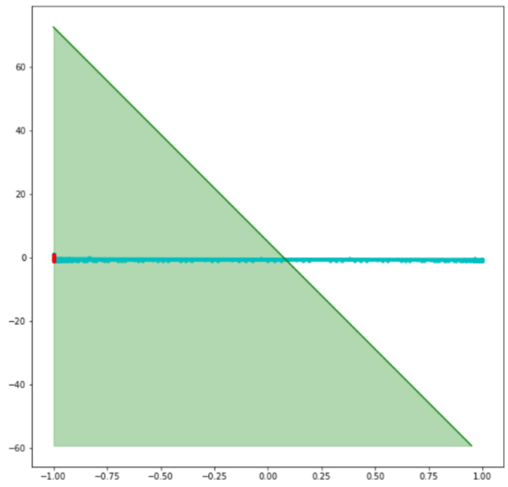
iii. Set 3)

Optimization:



```
[14]: # tanh hidden layer
      tol=1e-10
      a  = 3
      b = 0.4
      c = -2
      w11 = 19
      w12 = 2
      w21 = 6
      w22 = -2
      b1  = 0.0
      b2  = 0.0
      params =  np.array([a, b, c, w11, w12, w21, w22, b1, b2])

      dummy=time.time()

      res = minimize(objFunc2, params, args=(X, Y, x,y1,y2,dummy), tol=tol, options={'disp': True,'maxiter':500})
      Optimization terminated successfully.
               Current function value: 0.000000
               Iterations: 2
               Function evaluations: 66
               Gradient evaluations: 6
```

Graph:



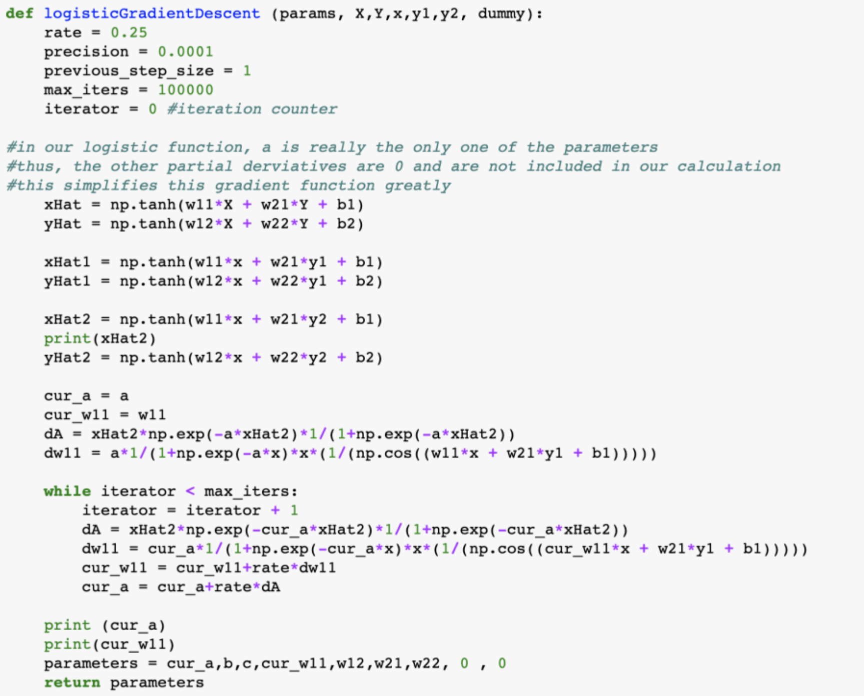
Brief Observations:

This approximation, like Set 1, once again landed on a minima of 0. Thus, it makes sense that the graph is very similar to that of Logistic Set 1. Unfortunately, it suffers a similar visual fate to Set 1 as well, although it seems, if the transformation plane was extended, the line would be a good approximation of the curves.

Gradient Descent

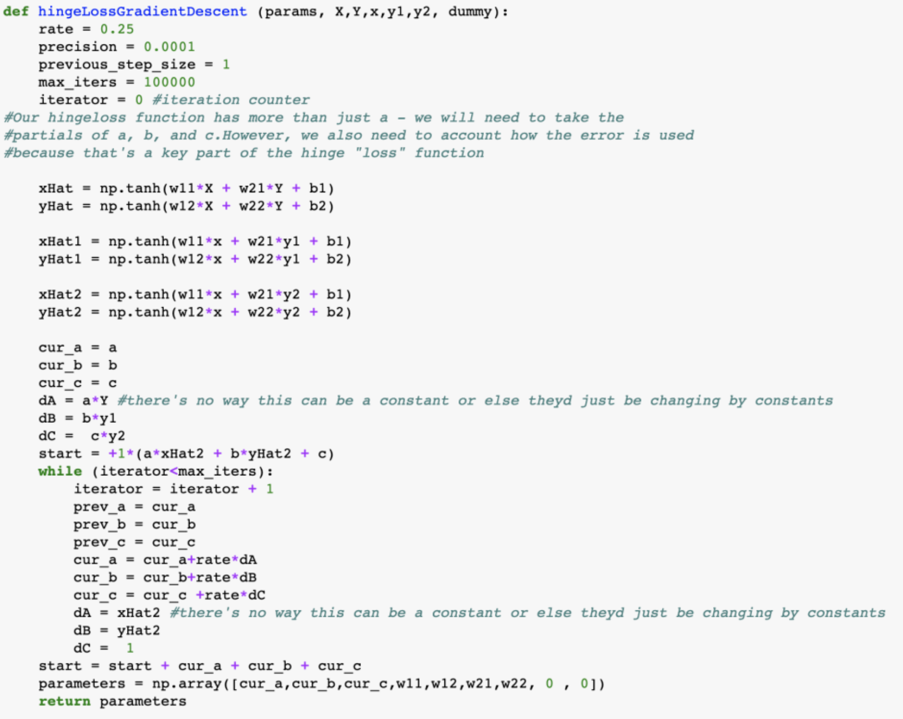
For Gradient Descent, we used two different functions and took their gradients to find the optimal parameter set.

This is our code for the logistic gradient descent:



```
def logisticGradientDescent (params, X,Y,x,y1,y2, dummy):
    rate = 0.25
    precision = 0.0001
    previous_step_size = 1
    max_iters = 100000
    iterator = 0 #iteration counter

#in our logistic function, a is really the only one of the parameters
#thus, the other partial derviatives are 0 and are not included in our calculation
#this simplifies this gradient function greatly
    xHat = np.tanh(w11*X + w21*Y + b1)
    yHat = np.tanh(w12*X + w22*Y + b2)

    xHat1 = np.tanh(w11*x + w21*y1 + b1)
    yHat1 = np.tanh(w12*x + w22*y1 + b2)

    xHat2 = np.tanh(w11*x + w21*y2 + b1)
    print(xHat2)
    yHat2 = np.tanh(w12*x + w22*y2 + b2)

    cur_a = a
    cur_w11 = w11
    dA = xHat2*np.exp(-a*xHat2)*1/(1+np.exp(-a*xHat2))
    dw11 = a*1/(1+np.exp(-a*x)*x*(1/(np.cos((w11*x + w21*y1 + b1)))))

    while iterator < max_iters:
        iterator = iterator + 1
        dA = xHat2*np.exp(-cur_a*xHat2)*1/(1+np.exp(-cur_a*xHat2))
        dw11 = cur_a*1/(1+np.exp(-cur_a*x)*x*(1/(np.cos((cur_w11*x + w21*y1 + b1)))))
        cur_w11 = cur_w11+rate*dw11
        cur_a = cur_a+rate*dA

    print (cur_a)
    print(cur_w11)
    parameters = cur_a,b,c,cur_w11,w12,w21,w22, 0 , 0
    return parameters
```

This is our code for the hinge-loss gradient descent

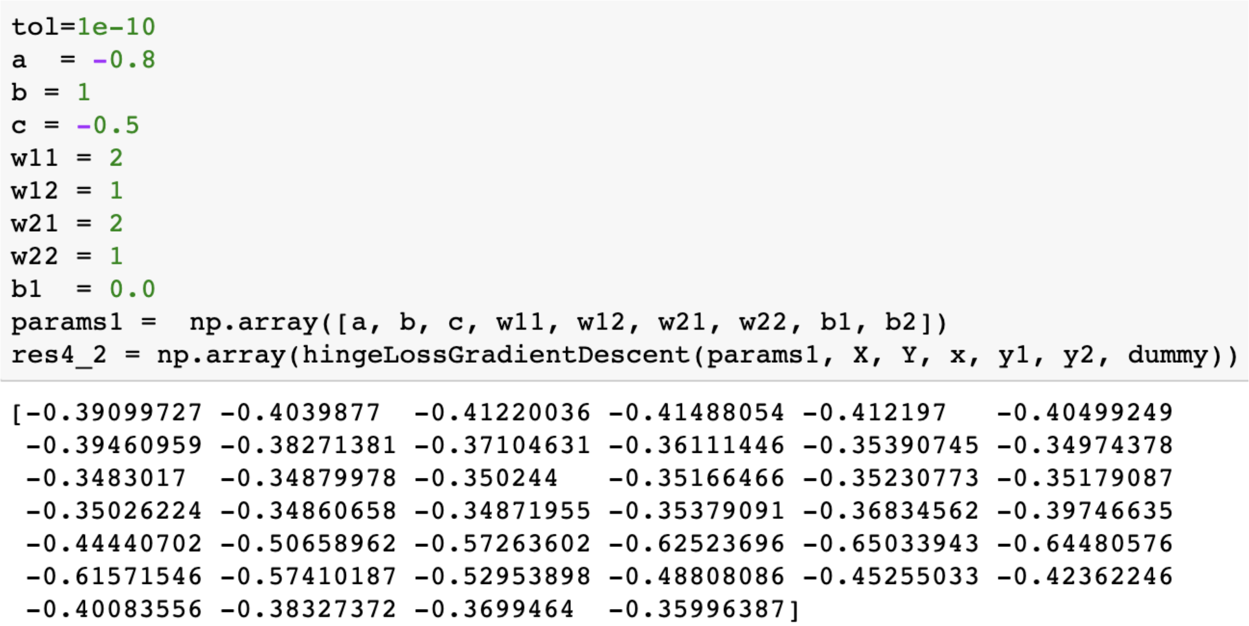


```
def hingeLossGradientDescent (params, X,Y,x,y1,y2, dummy):
    rate = 0.25
    precision = 0.0001
    previous_step_size = 1
    max_iters = 100000
    iterator = 0 #iteration counter
#Our hingeloss function has more than just a - we will need to take the
#partials of a, b, and c.However, we also need to account how the error is used
#because that's a key part of the hinge "loss" function

    xHat = np.tanh(w11*X + w21*Y + b1)
    yHat = np.tanh(w12*X + w22*Y + b2)

    xHat1 = np.tanh(w11*x + w21*y1 + b1)
    yHat1 = np.tanh(w12*x + w22*y1 + b2)

    xHat2 = np.tanh(w11*x + w21*y2 + b1)
    yHat2 = np.tanh(w12*x + w22*y2 + b2)

    cur_a = a
    cur_b = b
    cur_c = c
    dA = a*Y #there's no way this can be a constant or else theyd just be changing by constants
    dB = b*y1
    dC =  c*y2
    start = +1*(a*xHat2 + b*yHat2 + c)
    while (iterator<max_iters):
        iterator = iterator + 1
        prev_a = cur_a
        prev_b = cur_b
        prev_c = cur_c
        cur_a = cur_a+rate*dA
        cur_b = cur_b+rate*dB
        cur_c = cur_c +rate*dC
        dA = xHat2 #there's no way this can be a constant or else theyd just be changing by constants
        dB = yHat2
        dC =  1
    start = start + cur_a + cur_b + cur_c
    parameters = np.array([cur_a,cur_b,cur_c,w11,w12,w21,w22, 0 , 0])
    return parameters
```

1) Hinge-Loss

   a. Set 1)

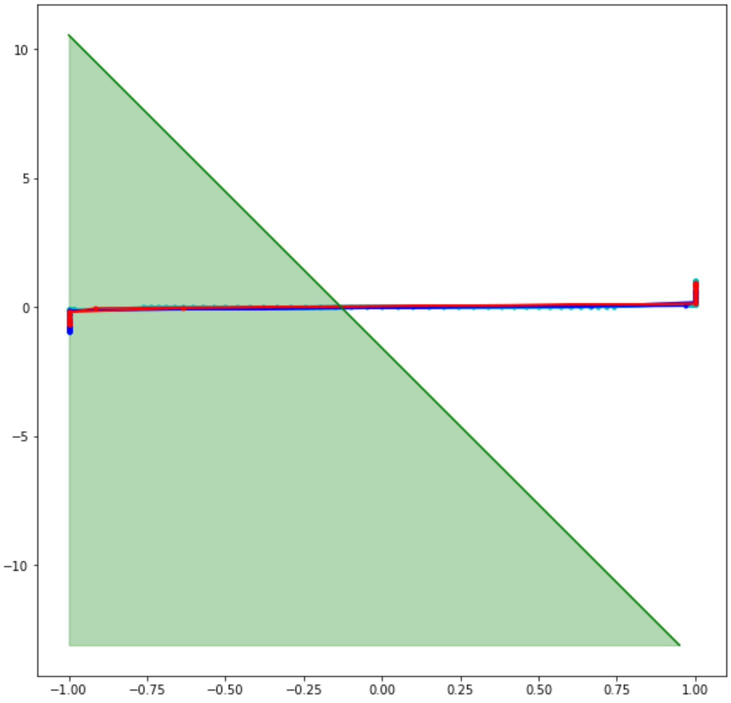
Optimization:



```
tol=1e-10
a  = -0.8
b = 1
c = -0.5
w11 = 2
w12 = 1
w21 = 2
w22 = 1
b1  = 0.0
params1 =  np.array([a, b, c, w11, w12, w21, w22, b1, b2])
res4_2 = np.array(hingeLossGradientDescent(params1, X, Y, x, y1, y2, dummy))
```

```
[-0.39099727 -0.4039877  -0.41220036 -0.41488054 -0.412197   -0.40499249
 -0.39460959 -0.38271381 -0.37104631 -0.36111446 -0.35390745 -0.34974378
 -0.3483017  -0.34879978 -0.350244   -0.35166466 -0.35230773 -0.35179087
 -0.35026224 -0.34860658 -0.34871955 -0.35379091 -0.36834562 -0.39746635
 -0.44440702 -0.50658962 -0.57263602 -0.62523696 -0.65033943 -0.64480576
 -0.61571546 -0.57410187 -0.52953898 -0.48808086 -0.45255033 -0.42362246
 -0.40083556 -0.38327372 -0.3699464  -0.35996387]
```

Graph:

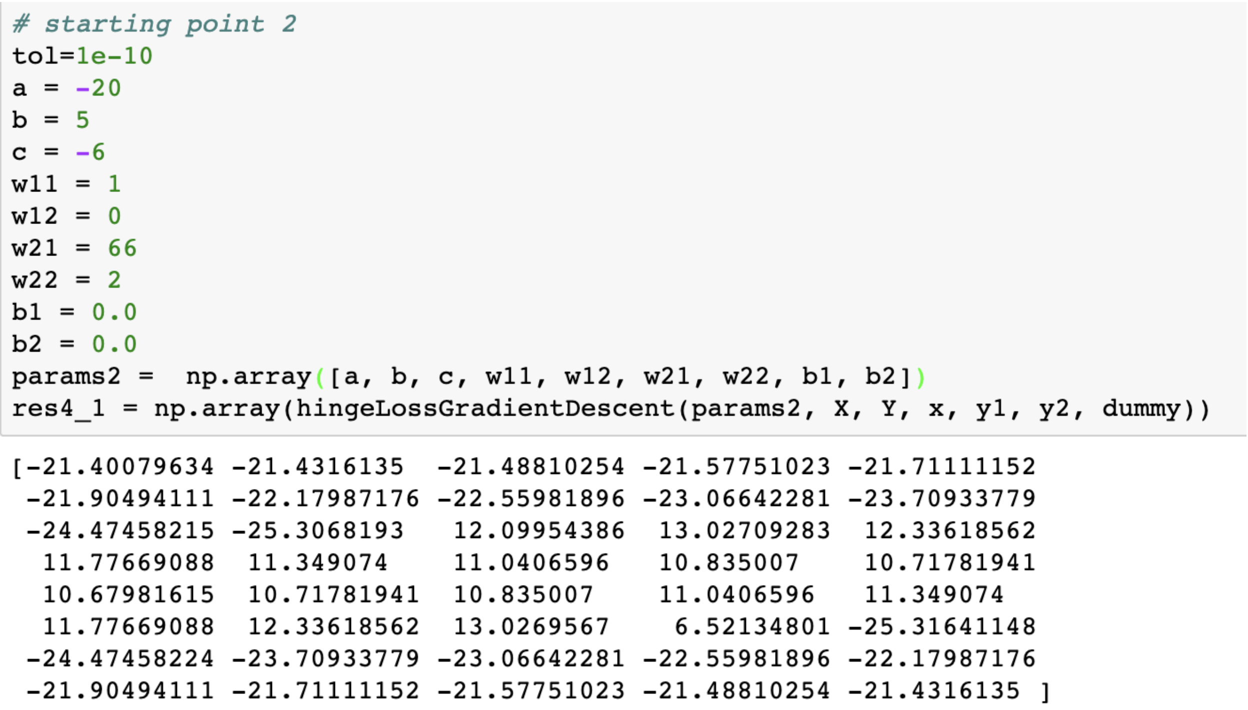


Brief Observations:

The issue we had with our gradient functions generally came in the transformation of the curves. We are generally sure that we did optimize the parameters in some capacity (because our final values of parameters did shift from the initial). However, in all, this graph does not look accurate to the curves, not even following their general slope. Another issue is that we got all of our answers in a vector format, which we tried to account for as well.

b. Set 2)

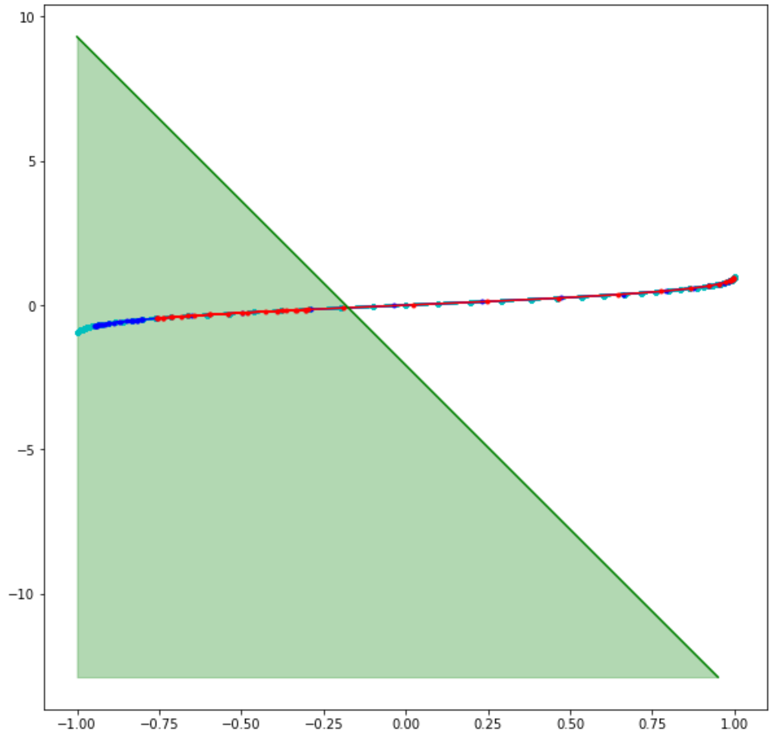
Optimization:



```
# starting point 2
tol=1e-10
a = -20
b = 5
c = -6
w11 = 1
w12 = 0
w21 = 66
w22 = 2
b1 = 0.0
b2 = 0.0
params2 =  np.array([a, b, c, w11, w12, w21, w22, b1, b2])
res4_1 = np.array(hingeLossGradientDescent(params2, X, Y, x, y1, y2, dummy))
```

```
[-21.40079634 -21.4316135  -21.48810254 -21.57751023 -21.71111152
 -21.90494111 -22.17987176 -22.55981896 -23.06642281 -23.70933779
 -24.47458215 -25.3068193   12.09954386  13.02709283  12.33618562
  11.77669088  11.349074    11.0406596   10.835007    10.71781941
  10.67981615  10.71781941  10.835007    11.0406596   11.349074
  11.77669088  12.33618562  13.0269567    6.52134801 -25.31641148
 -24.47458224 -23.70933779 -23.06642281 -22.55981896 -22.17987176
 -21.90494111 -21.71111152 -21.57751023 -21.48810254 -21.4316135 ]
```

Graph:

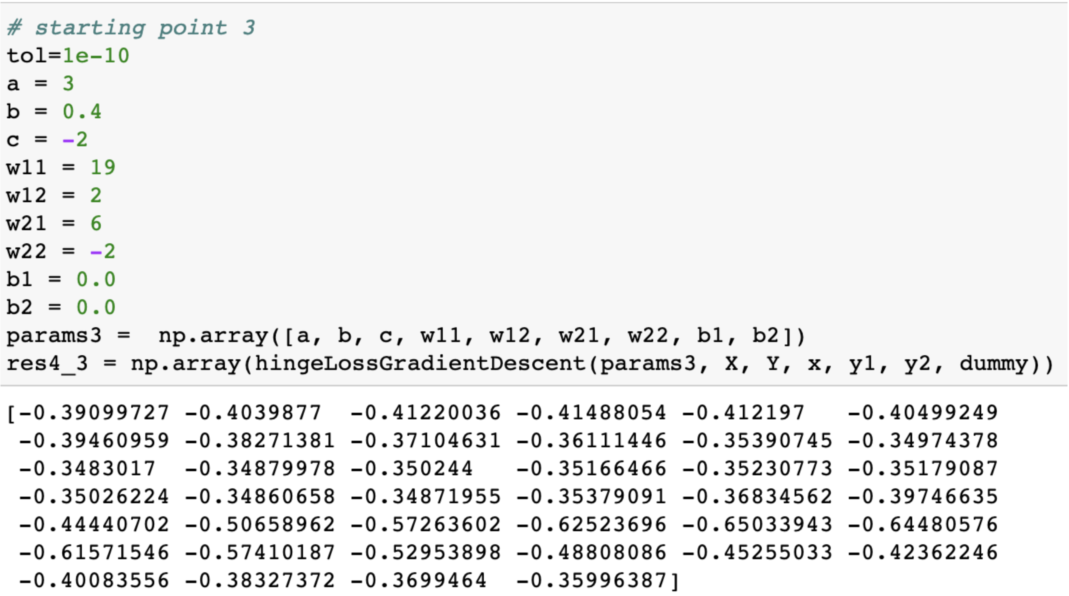


Brief Observations:

This graph seems to have found itself in a similar result as Step 1. The transformation of the curves do not seem to be well approximated by the line whatsoever. However, our simplex approximation with Hinge-Loss worked well – next time we do a similar project, we will further check our partial derivatives.

c. Set 3)

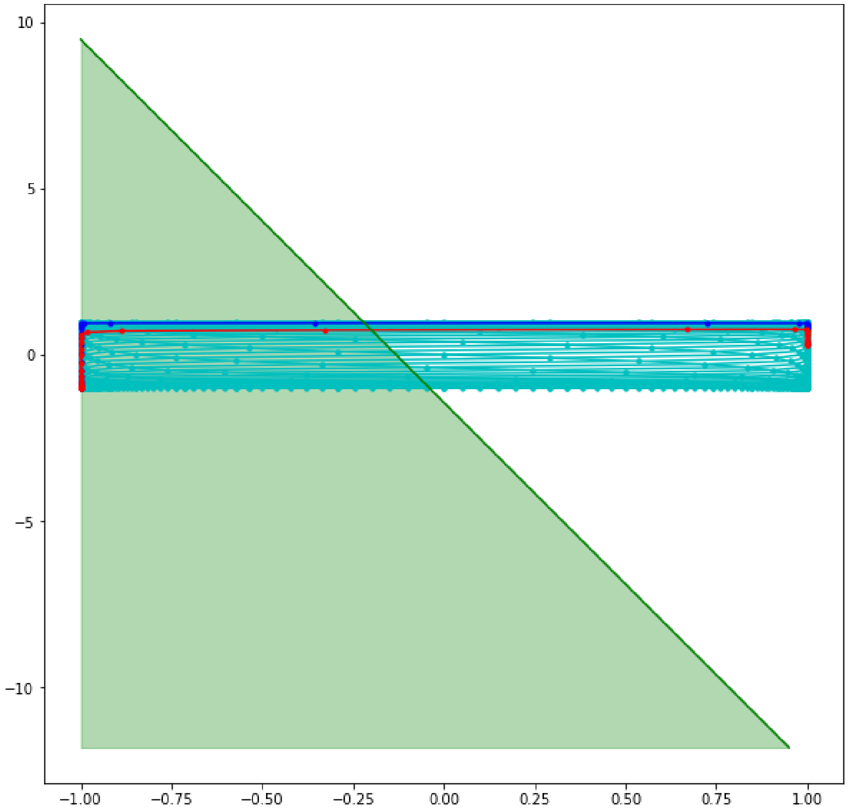
Optimization:



```
# starting point 3
tol=1e-10
a = 3
b = 0.4
c = -2
w11 = 19
w12 = 2
w21 = 6
w22 = -2
b1 = 0.0
b2 = 0.0
params3 =  np.array([a, b, c, w11, w12, w21, w22, b1, b2])
res4_3 = np.array(hingeLossGradientDescent(params3, X, Y, x, y1, y2, dummy))
```

```
[-0.39099727 -0.4039877  -0.41220036 -0.41488054 -0.412197   -0.40499249
 -0.39460959 -0.38271381 -0.37104631 -0.36111446 -0.35390745 -0.34974378
 -0.3483017  -0.34879978 -0.350244   -0.35166466 -0.35230773 -0.35179087
 -0.35026224 -0.34860658 -0.34871955 -0.35379091 -0.36834562 -0.39746635
 -0.44440702 -0.50658962 -0.57263602 -0.62523696 -0.65033943 -0.64480576
 -0.61571546 -0.57410187 -0.52953898 -0.48808086 -0.45255033 -0.42362246
 -0.40083556 -0.38327372 -0.3699464  -0.35996387]
```

Graph:



\

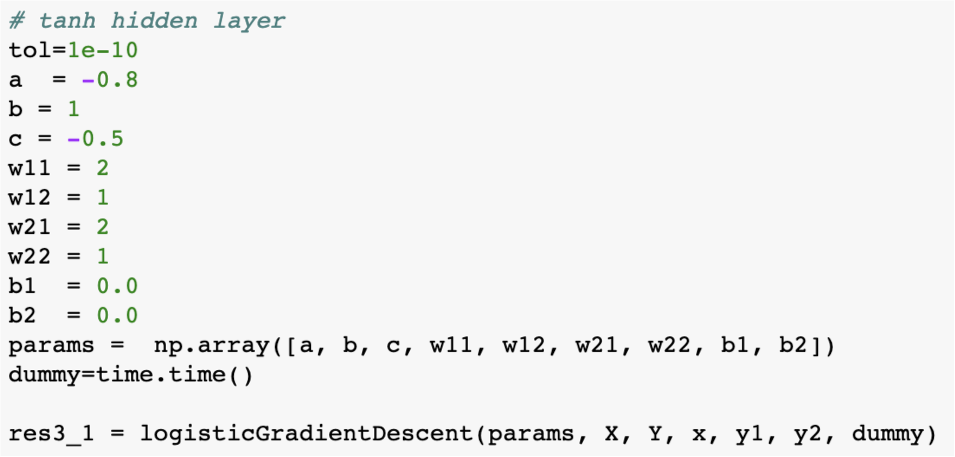
Brief Observations:

This final set seems to be the most accurate to the transformations (although, quite clearly, it could use some work). Through our last trials, this data set is one of the closest to the actual approximation, thus it makes sense that this could be our most accurate model in our gradient descent model as well.

2) Logistic

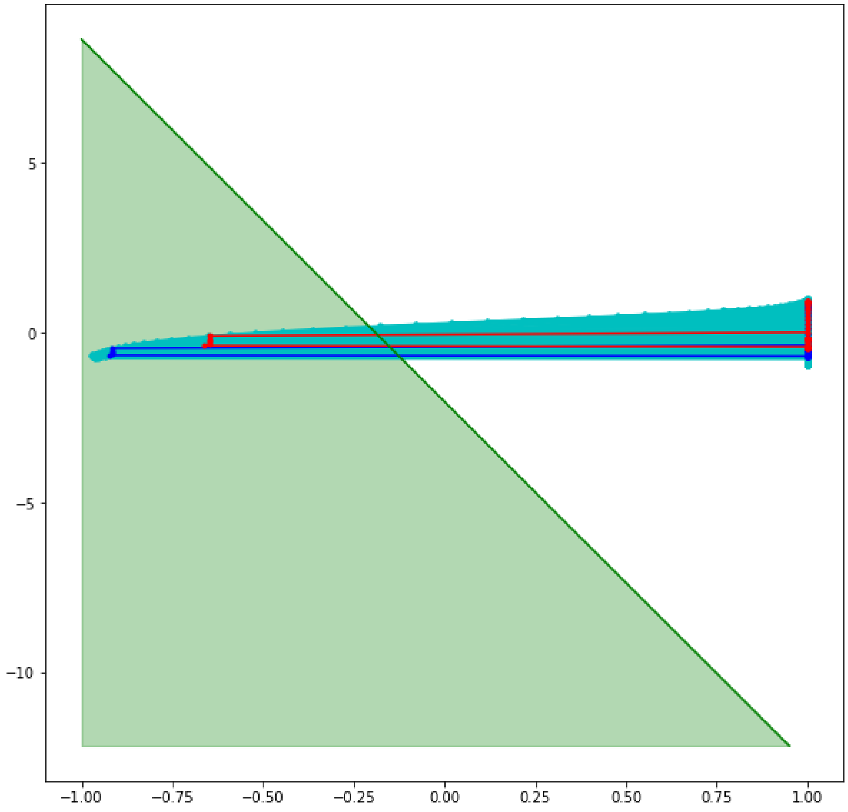
a. Set 1)

Optimization:



```
# tanh hidden layer
tol=1e-10
a  = -0.8
b = 1
c = -0.5
w11 = 2
w12 = 1
w21 = 2
w22 = 1
b1  = 0.0
b2  = 0.0
params =  np.array([a, b, c, w11, w12, w21, w22, b1, b2])
dummy=time.time()

res3_1 = logisticGradientDescent(params, X, Y, x, y1, y2, dummy)
```

Graph:



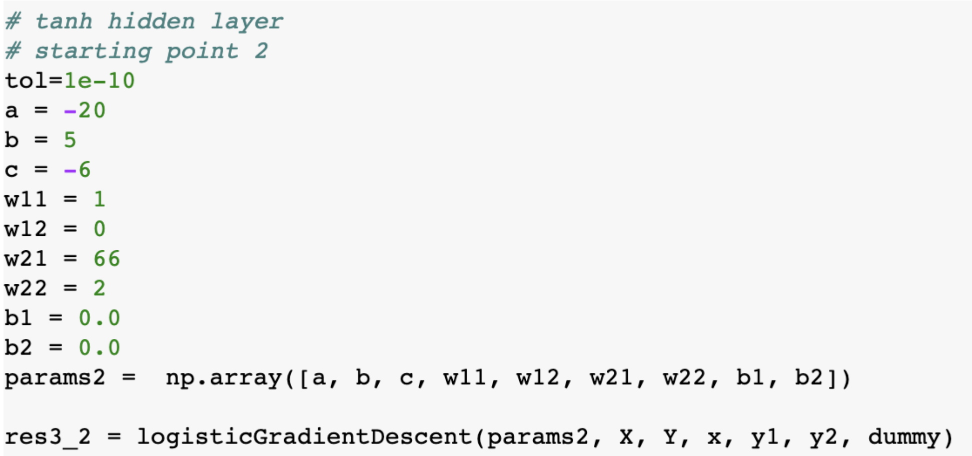
:

Brief Observations:

The Logistic Gradient Descent function seems to be slightly more successful than the Hinge-Loss function; however, we still cannot say it is accurate to the transformed curves.

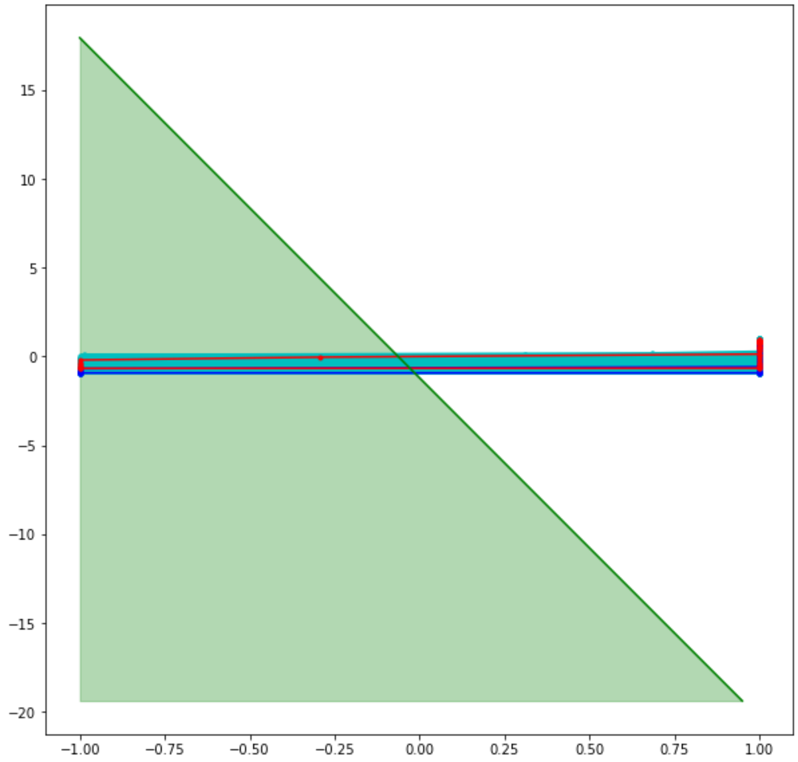
b. Set 2)

Optimization:



```
# tanh hidden layer
# starting point 2
tol=1e-10
a = -20
b = 5
c = -6
w11 = 1
w12 = 0
w21 = 66
w22 = 2
b1 = 0.0
b2 = 0.0
params2 =  np.array([a, b, c, w11, w12, w21, w22, b1, b2])

res3_2 = logisticGradientDescent(params2, X, Y, x, y1, y2, dummy)
```

Graph:

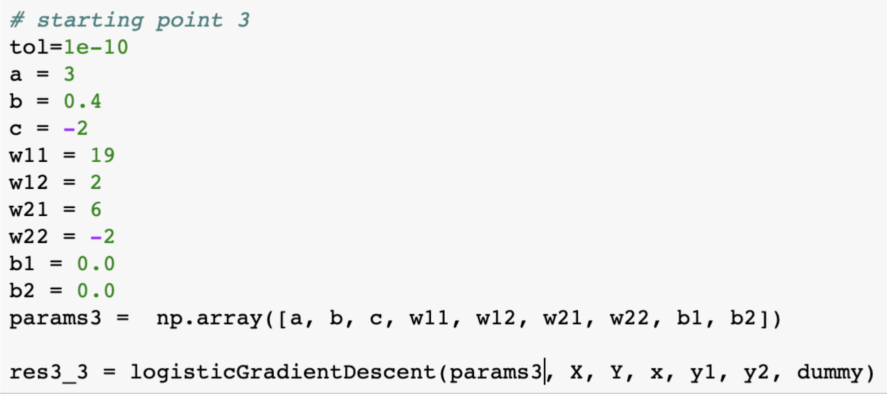


Brief Observations:

Surprisingly, this Set 2 Logistic Gradient-Descent graph seems slightly more accurate than Set . Ultimately, however, the line still crosses both of the transformed curves and cannot be presented as an accurate approximation.

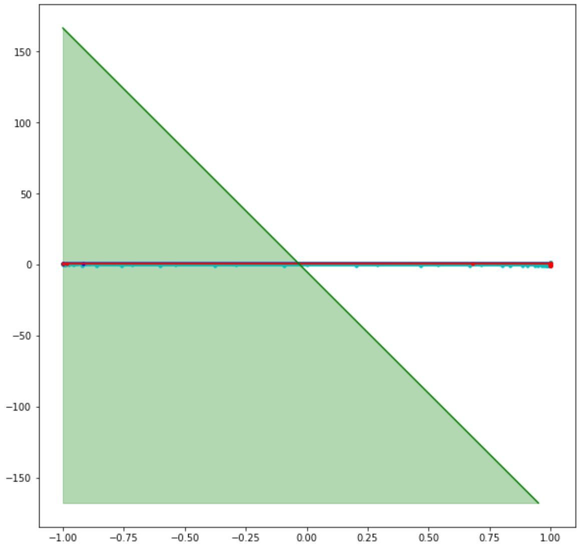
c. Set 3)

Optimization:



```
# starting point 3
tol=1e-10
a = 3
b = 0.4
c = -2
w11 = 19
w12 = 2
w21 = 6
w22 = -2
b1 = 0.0
b2 = 0.0
params3 =  np.array([a, b, c, w11, w12, w21, w22, b1, b2])

res3_3 = logisticGradientDescent(params3, X, Y, x, y1, y2, dummy)
```

Graph:



Brief Observations:

This third approximation only confirms that there must have been an error in our ability to compute the gradient of this logistic function; initially, for our line, we got a complex curve. After doing some computation with the vectors to make them constants, we arrived at this line.

Conclusion:

Through this project – despite our mixed results in the accuracy of our models – we learned a significant amount about data, transformations, and optimization. Our most accurate model of the data was definitely our Hinge-Loss objective function optimized via Simplex (non-gradient) methods. The rest of our models often had errors in the transformation or the slope of the line itself; in retrospect, we may not have taken partial derivatives of enough of the variables or subtracted the gradient values instead of adding them in our gradient-based optimization functions. One of the biggest issues we faced was dimension errors – oftentimes after calculations, the parameters were returned as vectors, creating dimensional and qualitative errors. Another error may have come from not accounting from whether or not the inputs are misclassified by the objective function in our gradient descent algorithms. Nevertheless, we were able to develop skills in Python (especially using numpy) and now understand the general concepts behind creating a linear approximation of data.