CSC 535 Data Mining

Evolving Neural Network Shape with the Ant Colony Optimization Algorithm for Predicting Eye State with EEG Data

Submitted to: Dr. Jamil Saquer

Authors:

*Ryan Bagby*

*Lennard Masseau*

*Michael Knapp*

**EVOLVING NEURAL NETWORK SHAPE WITH THE ANT COLONY OPTIMIZATION ALGORITHM FOR PREDICTING EYE STATE WITH EEG DATA**

**Introduction**

Neural networks have gained prominent attention for their extraordinary skill for problems of classification as well as regression. Though useful, these networks still require careful tuning of parameters and preprocessing of data in order to attain the high levels of performance. For this reason there is a wide-ranging literature dedicated to developing techniques to algorithmically fine tune and optimize the network. The Ant Colony Optimization algorithm (ACO) from the related field of Evolutionary Computation is one such technique which draws insight from the foraging and resource management strategies of real ants. For evolving the shape of a neural network, the algorithm balances the trade-off of exploring new shapes and exploiting variations of known shapes with the current best performance by probabilistically constructing a candidate solution through artificial pheromone signals which are updated and decayed according to local and global best solutions. The resulting neural network constructed using the shape found by ACO consistently displays a high degree of accuracy on a classification task consisting of predicting the state of an individual's eye (open or closed) from EEG data of 14 regions of the brain.

**Background**

The data set used was the EEG Eye State data from the UCI Machine Learning Repository and consisted of 14,980 data points each of which represented 14 attributes corresponding to 14 different regions of the brain and a label of 0 or 1 for the patient's eyes being closed or open[[1]](#footnote-1). The data was randomly split into 2 disjoint sets, 90% of which was used for training and the remaining 10% used for testing.

Ant Colony Optimization (ACO) is modeled on real ants where dropping pheromones where the ant finds food attracts other ants in the colony to look for food in that location. If another ant finds food in that location, he then drops off more pheromones to alert other aunts he also found food in that location. The more food found at a location the more pheromones are dropped and in turn the more ants gravitate to that location to find food. It’s a natural solution to searching based very large sample spaces. Using this idea, an algorithm was designed to mimic this foraging and resource management procedure to search for efficient hidden layer shapes for a Multilayer Perceptron (MLP) classifier. This will increase the likely hood of finding unique mutation or bias’s in the hidden layer. We hypothesize that way of setting up a NN will fine new more creative ways for the machine to learn solutions to the problem. This could lead to quicker solutions or solutions that haven’t been thought of before.

**Implementation**

We used the Python libraries Numpy, Pandas, and Scikit Learn for their ease of use for data processing. This is the first time we have used these tools to implement our solution. I think we all agree that this makes things a lot easier to not only work with the data but to also to create the neural network. One thing that really help us was the Book Clever Algorithms. With the examples of online test, we decided to use nature to help randomize the hidden layer of our neural network with natures solution to the traveling salesperson problem. Most of the algorithms described in the in this book were originally inspired by biological and natural systems, such as the adaptive capabilities of genetic evolution and the vast complexity of the immune system. They derive this inspiration from the behaviors of vast species of creatures like birds, bees, ants and bacteria.

Another method that we found was useful in manipulating the data to a format that the neural network that we built could understand and process the data correctly was the TRAIN\_TEST\_SPLIT. If anyone has used Python before they know that this process can be have the battle of finding a programming solution to there problem. This method did exactly as you would think I would from the previous description. It split the training data and the test data into format that could be processed using the Scikit Learn library to build the neural network. This essentially made a few lines of code not only format the data easily but also using this method is conjunction with the Scikit Learn methods on building the neural network far easier than it would have been without them. We would defiantly suggest spending time researching these three libraries thoroughly if you want to implement a NN.

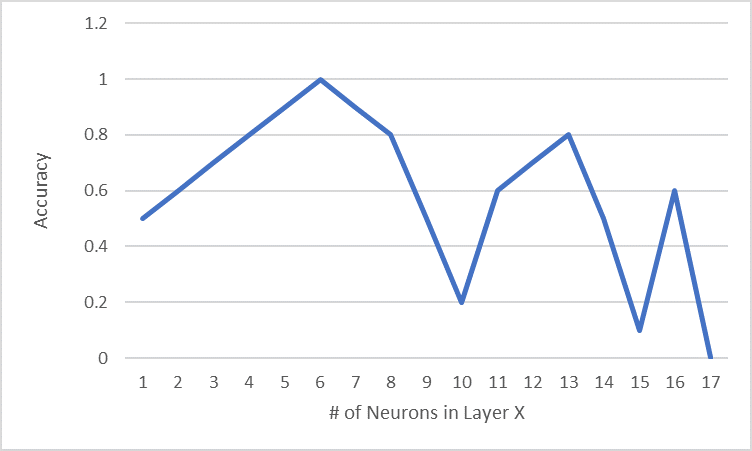
**Experimental Setup and Results**

With Scikit Learn we also had the option of scaling the data. This was important step in our process to ensure that our implementation was able to get better accuracy rating than we initially started with. We changed the scale of the data to stay within the range of [-1,1] this changed our initial accuracy level from ranges in the 50-60% to well above 80% accuracy. These results were with the (greed) variable set to a static range of .09 you could call this are min threshold value. Ryan had a great idea, we could improve the possibility of the overall accuracy by trying to decrease the accuracy to start with. He hypothesized if we changed the greed variable to the accuracy rate we could find a new path that would lead us to an increase the overall accuracy rate. The variable in our code Greed is set to a static threshold rate of 0.9 so we change the variable from the static value to the accuracy rate. At first this would lead to lower accuracy rates, but it gave the chance of finding a mutation that was not found previously giving us a higher chance of locating a new path that was more efficient. We show this concept in the graphs shown in figure # 1 and figure #2.

Fist graph is showing a representation of the data. Not the actual data but a visual representation of how we how to improve changing the scale being processed to find the most efficient path. Figure #1 shows the scale at [-1,1]

 Figure #1

The second figure #2 shows when we change the greed variable in our code to not be fixed at .9 but to be set to the accuracy variable. This increases the scale of paths being searched which might cause less accuracy at first but could find a higher peak in the. When we did this we found that our data did achieve higher accuracy rates from 80% to above 96%.

 Figure #2

**Conclusion**

The ant colony optimization preforms sufficiently for searching and selecting hidden layers for a muti-layered perceptron. The MLP constantly showed greater than 90 percent accuracy. In the future you could expand it so search not only for the hidden shapes but other parameters and also experiment with ways to make it more computationally efficient. The group members would like to give special credit to Ryan Bagby. He knowledge was detrimental in the implementation of the NN as was his creative ability to conceptualize this project’s unique application of using the ant (TSP) randomization of the hidden layers.

*Give any concluding remarks here. If you learned anything talk about that here as well. If you discovered anything interesting (extra credit) then talk about it here too.*

**Code**

**import** numpy **as** np

**import** pandas **as** pd

**from** math **import** sqrt

**from** random **import** randint**,** random

**import** itertools **as** itti

**from** sklearn**.**neural\_network **import** MLPClassifier

**from** sklearn**.**model\_selection **import** train\_test\_split

**from** sklearn**.**preprocessing **import** scale

#import matplotlib.pyplot as plt

#from matplotlib.animation import FuncAnimation

**def** accuracy**(**hidden\_shape**,** X**,** y**):**

classifier **=** MLPClassifier**(**hidden\_layer\_sizes**=**hidden\_shape**,** solver**=**'adam'**,** activation**=**'relu'**,**

max\_iter**=**50**,** batch\_size**=**125**)**

X\_train**,** X\_test**,** y\_train**,** y\_test **=** train\_test\_split**(**X**,** y**,** test\_size**=**0.5**,** train\_size**=**0.03**)**

classifier **=** classifier**.**fit**(**X\_train**,** y\_train**)**

score **=** classifier**.**score**(**X\_test**,** y\_test**)**

#print('{} has accuracy of {}%'.format(hidden\_shape, round(score,4)\*100))

**return** score

**def** initialize\_pheromone**(**num\_bits**,** init\_phero**):**

pheromones **=** **[[[**init\_phero **for** j **in** range**(**2**)]** **for** k **in** range**(**2**)]** **for** l **in** range**(**num\_bits**)]**

**return** pheromones

**def** get\_choices**(**X**,** y**,** last\_index**,** start\_layer**,** start\_shape**,** next\_shape**,** pheromone**,** heur**,** history**,** min\_max**,** num\_bits**):**

choices **=** **[]**

**for** bit **in** range**(**2**):**

shape **=** **[**start\_shape**,** next\_shape**]**

prob **=** **{**'shape'**:** shape**}**

prob**[**'history'**]** **=** pheromone**[**last\_index**][**start\_layer**[**last\_index**]][**bit**]** **\*\*** history

prob**[**'accuracy'**]** **=** accuracy**(**shape**,** X**,** y**)**

prob**[**'cost'**]** **=** **(**1.0 **-** prob**[**'accuracy'**])** **\*** 100 **if** prob**[**'accuracy'**]** **<** 1.0 **else** 1.0

prob**[**'heuristic'**]** **=** **(**1.0 **/** prob**[**'cost'**])** **\*\*** heur

prob**[**'prob'**]** **=** prob**[**'history'**]** **\*** prob**[**'heuristic'**]**

prob**[**'binary'**]** **=** bit

choices**.**append**(**prob**)**

**return** choices

**def** prob\_select**(**choices**):**

**print(**'PROBABILISTIC CHOICE \n'**)**

probs **=** **[**c**[**'prob'**]** **for** c **in** choices**]**

s **=** sum**(**probs**)**

**if** s **==** 0**:**

c **=** choices**[**randint**(**0**,** 1**)]**

**return** c**[**'binary'**]**

v **=** random**()**

**for** c **in** choices**:**

v **-=** c**[**'prob'**]/**s

**if** v **<** 0.0**:**

**return** c**[**'binary'**]**

**return** choices**[-**1**][**'binary'**]**

**def** greedy\_select**(**choices**):**

**print(**'GREEDY CHOICE \n'**)**

probs **=** **[**c**[**'prob'**]** **for** c **in** choices**]**

i **=** np**.**argmax**(**probs**)**

**return** choices**[**i**][**'binary'**]**

**def** calc\_shape**(**bin\_list**):**

shape **=** sum**([**bin\_list**[**i**]** **<<** i **for** i **in** range**(**len**(**bin\_list**))])**

**return** shape

**def** stepwise\_construct**(**X**,** y**,** start\_layer**,** pheromone**,** heur**,** greed**,** min\_max**,** num\_bits**):**

next\_shape **=** 1

bin\_list **=** **[**next\_shape**]**

start\_shape **=** calc\_shape**(**start\_layer**)**

**for** j **in** range**(**1**,**num\_bits**):**

i **=** num\_bits **-** j

cand **=** get\_choices**(**X**,** y**,** i**,** start\_layer**,** start\_shape**,** next\_shape**,** pheromone**,** heur**,** 1.0**,** min\_max**,** num\_bits**)**

greedy **=** random**()** **<=** greed

next\_bit **=** greedy\_select**(**cand**)** **if** greedy **else** prob\_select**(**cand**)**

next\_shape **+=** next\_bit **<<** i **+** 1

bin\_list**.**append**(**next\_bit**)**

bin\_list**.**reverse**()**

candidate **=** **{**'binary'**:** bin\_list**}**

#print('Start layer: {} & binary list: {}'.format(start\_layer, bin\_list))

candidate**[**'shape'**]** **=** **[**start\_shape**,** next\_shape**]**

#print('Stepwise candidate shape is {} \n'.format(candidate['shape']))

candidate**[**'accuracy'**]** **=** accuracy**(**candidate**[**'shape'**],** X**,** y**)**

**return** candidate**[**'shape'**],** candidate**[**'binary'**],** candidate**[**'accuracy'**]**

**def** global\_update\_phero**(**start\_layer**,** pheromone**,** best**,** decay**,** num\_bits**):**

**print(**'Global pheromone update.'**)**

h1 **=** start\_layer

h2 **=** best**[**'binary'**]**

**for** bit **in** range**(**num\_bits**):**

value **=** **((**1.0 **-** decay**)\***pheromone**[**bit**][**h1**[**bit**]][**h2**[**bit**]])** **+** **(**decay **\*** best**[**'accuracy'**])**

pheromone**[**bit**][**h1**[**bit**]][**h2**[**bit**]]** **=** value

**def** local\_update\_phero**(**start\_layer**,** pheromone**,** candidate**,** local\_phero**,** init\_phero**,** num\_bits**):**

**print(**'Local pheromone update.'**)**

h1 **=** start\_layer

h2 **=** candidate**[**'binary'**]**

**for** bit **in** range**(**num\_bits**):**

value **=** **((**1.0 **-** local\_phero**)\***pheromone**[**bit**][**h1**[**bit**]][**h2**[**bit**]])** **+** **(**local\_phero**\***init\_phero**)**

pheromone**[**bit**][**h1**[**bit**]][**h2**[**bit**]]** **=** value

**def** search**(**X**,** y**,** layer\_shapes**,** max\_iters**,** num\_ants**,** decay**,** heur**,** local\_phero**,** greed**,** min\_max**,** num\_bits**):**

b\_hidden1 **=** layer\_shapes**[**randint**(**0**,** len**(**layer\_shapes**)-**1**)]**

b\_hidden2 **=** layer\_shapes**[**randint**(**0**,** len**(**layer\_shapes**)-**1**)]**

best **=** **{**'binary'**:** b\_hidden2**}**

b\_hidden1 **=** calc\_shape**(**b\_hidden1**)**

b\_hidden2 **=** calc\_shape**(**b\_hidden2**)**

best**[**'shape'**]** **=** **[**b\_hidden1**,** b\_hidden2**]**

best**[**'accuracy'**]** **=** accuracy**(**best**[**'shape'**],** X**,** y**)**

**print(**'Starting shape is {} with accuracy {}% \n'**.**format**(**best**[**'shape'**],**round**(**best**[**'accuracy'**],**4**)\***100**))**

init\_phero **=** 1.0**/**len**(**layer\_shapes**)**

pheromone **=** initialize\_pheromone**(**len**(**layer\_shapes**),** init\_phero**)**

**for** i **in** range**(**max\_iters**):**

**for** a **in** range**(**num\_ants**):**

**print(**'Iteration {} of {} \n'**.**format**(**i**+**1**,** max\_iters**))**

**print(**'Ant {} of {} \n'**.**format**(**a**+**1**,** num\_ants**))**

start\_layer **=** layer\_shapes**[**randint**(**0**,** len**(**layer\_shapes**)-**1**)]**

start\_shape **=** calc\_shape**(**start\_layer**)**

**print(**'Starting layer shape is {} \n'**.**format**(**start\_shape**))**

candidate **=** **{}**

candidate**[**'shape'**],** candidate**[**'binary'**],** candidate**[**'accuracy'**]** **=** stepwise\_construct**(**X**,** y**,** start\_layer**,** pheromone**,**

heur**,** greed**,** min\_max**,** num\_bits**)**

**print(**'Chose new candidate with shape {} and accuracy of {}% \n'**.**format**(**candidate**[**'shape'**],** round**(**candidate**[**'accuracy'**],**4**)\***100**))**

**print(**'Current best is shape {} with accuracy {}%'**.**format**(**best**[**'shape'**],** round**(**best**[**'accuracy'**],**4**)\***100**))**

best **=** candidate **if** candidate**[**'accuracy'**]** **>** best**[**'accuracy'**]** **else** best

**print(**'Now the best is shape {} with accuracy {}% \n'**.**format**(**best**[**'shape'**],** round**(**best**[**'accuracy'**],**4**)\***100**))**

local\_update\_phero**(**start\_layer**,** pheromone**,** candidate**,** local\_phero**,** init\_phero**,** num\_bits**)**

**print(**'======================================================= \n'**)**

global\_update\_phero**(**start\_layer**,** pheromone**,** best**,** decay**,** num\_bits**)**

**print(**'======================================================= \n'**)**

**print(**'Finished Search. \n'**)**

**return** best

# --- Problem constants

# heuristic; significance of historical choices; typically between 2 and 5

HEUR **=** 2.5

# pheromone influence factor

LOCAL\_PHERO **=** 0.1

# likelihood of choosing greedily (instead of probabilistically)

GREED **=** 0.9

min\_max **=** **(**0**,** 300**)**

build\_net **=** **True**

setup\_range **=** range**(**min\_max**[**0**]+**1**,** min\_max**[**1**])**

multiplier **=** 1

num\_bits **=** 6

layer\_shapes **=** **[**list**(**a**)** **for** a **in** it**.**product**([**0**,** 1**],** repeat**=**num\_bits**)]**

zero\_vector **=** **[**0 **for** i **in** range**(**num\_bits**)]**

layer\_shapes**.**remove**(**zero\_vector**)**

max\_iters **=** 5

num\_ants **=** 10

decay **=** 0.1

file **=** "EEG\_Eye\_Detection.csv"

data **=** pd**.**read\_csv**(**file**)**

X **=** data**[:].**iloc**[:,** 0**:**14**]**

y **=** data**[:].**iloc**[:,** 14**]**

X **=** scale**(**X**)**

# --- Run ant colony search algorithm

best **=** search**(**X**,** y**,** layer\_shapes**,** max\_iters**,** num\_ants**,** decay**,** HEUR**,** LOCAL\_PHERO**,** GREED**,** min\_max**,** num\_bits**)**

**print(**'Shape of best network is {} \n'**.**format**(**best**[**'shape'**]))**

**if** build\_net**:**

X\_train**,** X\_test**,** y\_train**,** y\_test **=** train\_test\_split**(**X**,** y**,** test\_size**=**0.1**,** train\_size**=None)**

**print(**'Starting to train network returned by ants. \n'**)**

classifier **=** MLPClassifier**(**hidden\_layer\_sizes**=**best**[**'shape'**],** solver**=**'adam'**,** activation**=**'relu'**,** max\_iter**=**200**,** batch\_size**=**200**)**

classifier **=** classifier**.**fit**(**X\_train**,** y\_train**)**

scores **=** classifier**.**score**(**X\_test**,** y\_test**)**

# Best MLP so far: hidden\_layer\_sizes=(262,18), solver='adam', activation='relu', max\_iter=200, batch\_size=200

# Accuracy: 89.39% with test\_size=0.1, train\_size=None

# 2nd Best: shape=(272,235) and accuracy=89.19%

# 3rd Best: shape=(193,76) and accuracy=89.12%

# 4th Best: shape=(160,94) and accuracy=88.92%

**print(**'After further training, network of shape {} has accuracy of {}%.'**.**format**(**best**[**'shape'**],** round**(**scores**,**4**)\***100**))**

1. https://archive.ics.uci.edu/ml/machine-learning-databases/00264/ [↑](#footnote-ref-1)