Project #3 – Self-Organizing Map (SOM)

Introduction

This project will introduce you to a simple type of machine learning algorithm neural network grid called a Self-Organizing (Feature) Map (SOM, or SOFM). We will train this SOM machine so that it recognizes the class/category of each input vector that we feed it. We train the SOM by feeding it "training vectors" for which we already know the correct classifications, and the SOM self-learning mechanism adjusts the SOM neural nodes accordingly. Thus, the SOM will be a Classifier machine for us – in goes a "wilder", out comes its class.

The project will be written in HTML+JS+P5.

SOM Architecture

A SOM is a Flat 2D array of neural nodes each of which is connected to its neighbors. The simplest connection is NSEW (to the North, South, East, and West neighbors, which is what we will use. The array size, N, and shape depend on the problem and the computational costs, and can be rather subjective.

Each node has a vector of "weights", and a class value. We will be using 3D vectors. Note that a vector is equivalent to a point in normal Euclidean space, so you can think of them either way. We use 3D vectors for the nodes because our input data are also 3D vectors, "training" vectors used to train our machine.

One of the key things that a SOM classifier does is that it maps higher-dimensional (vector) data onto a lower-dimensional space (here, 3D to 2D).

SOM Learning

Initially, before training, the SOM node weight vectors and class value are set at random. During training, some of the SOM node weight vectors are adjusted for each training vector feed into the SOM.

The SOM trains by "competition learning". Each input vector is handed to every SOM neural node in the array, which then calculates the distance between that input vector and the node's internal set of weights (its "weight vector"). Then a winning closest node is chosen (AKA the BMU == Best Matching Unit). Finally, the winner and its buddies/neighbors adjust their weights to move toward the already-adjusted winning vector. To speed things up a bit, each buddy's buddies (step-buddies), except for the winner of course, are then adjusted toward the already-adjusted buddy in the same way.

Note that there are 8 step-buddies for a NSEW neighborhood and 4 of those step-buddies have two already-adjusted buddies that they will "move toward".

Also, along with the training vector, its correct classification value is also input, and the winner and buddies change to use that input class.

This continues until all the training vectors have been input, one at a time. One pass through all the training vectors is called an "epoch". Usually, several epochs are run. For each epoch, we keep track of the largest distance in any of the buddies (whose changes are always bigger than their winners). Once the epoch's max buddy distance is below some preset value, we decalre the SOM fully trained and ready for action.

Distance and Weight Adjustment

Use Euclidean distance (3D) to determine the winner. (A trick to speed up computation is to ignore taking the square root, because you still get the same winner.)

For the weight adjustment, for target, T, toward whom you are moving vector, V, use the directed J-th coordinate difference (T.J - V.J), times the moving vector's J-th coordinate V.J, times a "learning rate" of 20%. The learning rate being well below 100% "smoothes things out". Remember that vector T should have already had its weights adjusted before moving vector V.

Classifying

Once the SOM is trained, we take a wilder (a vector from "the Wild" whose class we do not know), and feed it into the SOM.

The input wilder is handed to each of the SOM's nodes. The BMU winner is calculated, and the SOM outputs the class value of the winner (ie, the class value stored inside the winning node). Again, the winner is the SOM node with the closest weight vector to the input wilder vector.

Training Set

Call the features/coordinates P, Q, and R, so a vector looks like "(P,Q,R)". Each training vector and each wilder (a Wild-Space vector) will have constraints on the range of values for each coordinate.

The wildspace constrains the P and Q values to the range -5..5. And R is constrained to -60..60. These are float numbers, not integers -- though the training set only has integer P and Q values. Therefore, the random SOM node weights should be in these ranges for its (P,Q,R) weight vectors.

The three class values are 1, 2 and 3.

The training data is shown below, at the end. Each training vector includes an ID number, the actual (P,Q,R) vector, and its class value.

Rain-Training Display

We'd like to see the training and wilder classification in color, so we'll use a 21 by 21 grid of cells, each "containing" a SOM node, so the grid size is N=441.

Each cell color consists of red, green and blue "channels" -- integer values, each in the range 0..255. The colors should initially be set to some gray, say RGB = (100,100,100) decimal, or maybe in hexadecimal RGB = (80,80,80).

Each cell should display a color representing its class value (which initially is random). We will associate color channels to classes this way 1=red, 2=green and 3=blue. Then each time the cell's node is adjusted, change the cell's color by increasing its class's color channel up by K% and the other two channel's down by K %. This way, for every training vector that is input during learning, you should see the winning node's grid cell and adjusted neighbors change their colors more toward the input vector class's color.

Try K=10% to start, and check to see if it makes each training vector's affect moderately visible -- sort of like a rain-drop splash onto the grid. This way the viewer will see how the training is going.

Also, during training, elsewhere in the display, you should display the epoch number currently running and the ID of the training vector currently running.

Once the SOM is trained, when running wilders through the SOM, the display should highlight the grid cell of the winning node that gives the output class. Here, the epoch and training counters should be blanked (or set to 0, whichever seems easier).

Output Delay

After each training vector is input you should **delay** input the next training vector for **between one fifth and one half a second**, in order for the audience to see and understand the results.

Testing

Later on, some testing data will be provided. It will be in the same format as the training data. You will need some other display counts to track during testing: 1) a count of the test vectors so far, 2) a count of the correct class answers, and 3) a count of the incorrect class answers. (You may also want a button or something like it to input a test vector.)

Running Time

You should prepare a 1-page (at most) paper describing your analysis of the **rough running time** (not Big-O) of each algorithm as you have implemented it, not counting any GUI operations. Your basic operation is (likely) the arithmetic operations needed to find the winner and update the nodes weights. If you feel that other operations should also be included -- perhaps because they end up taking a significant (above 5% of the total)

time -- then they should be included as well. Once you have an expression for running time in terms of the number of operations used (including the algorithm's setup), then show (briefly) how this running time is **converted to a Big-O** running time.

Team

The team size is the same as before, but you can change team members (and team name) from the previous project if you wish.

Academic Rules

Correctly and properly attribute all third party material and references, if any, lest points be taken off.

Project Reporting Data, Readme File, Submission & Readme File, Grading

Same as for Project #1.

Training Data Set

We've left out the commas to avoid visual clutter. Note that while the P & Q coordinate values are integers, for wilders, they can also be floats (within the -5..5 range). Testing data will be supplied later.

```
(ID Vector Class)      (31 (3 -3 -2.7) 2)     (62 (0 -1 0.2) 2)      (93 (-3 1 13.3) 3)
(1 (5 5 -53.5) 1)      (32 (3 -4 10.5) 3)     (63 (0 -2 -5.4) 1)     (94 (-3 0 -4.0) 1)
(2 (5 4 -18.8) 2)      (33 (3 -5 -10.5) 1)    (64 (0 -3 5.4) 2)      (95 (-3 -1 -0.1) 2)
(3 (5 3 -6.9) 2)       (34 (2 5 -28.0) 3)     (65 (0 -4 12.8) 2)     (96 (-3 -2 -13.8) 1)
(4 (5 2 -15.6) 1)      (35 (2 4 -17.6) 2)     (66 (0 -5 25.0) 2)     (97 (-3 -3 25.1) 3)
(5 (5 1 19.3) 3)       (36 (2 3 -7.8) 2)      (67 (-1 5 -19.5) 2)    (98 (-3 -4 18.8) 2)
(6 (5 0 0.0) 2)        (37 (2 2 -2.4) 2)      (68 (-1 4 -23.2) 1)    (99 (-3 -5 35.5) 2)
(7 (5 -1 -11.3) 1)     (38 (2 1 7.8) 3)       (69 (-1 3 -3.3) 2)     (100 (-4 5 -2.0) 1)
(8 (5 -2 -0.4) 3)      (39 (2 0 -11.0) 1)     (70 (-1 2 -0.6) 2)     (101 (-4 4 1.4) 1)
(9 (5 -3 -14.1) 1)     (40 (2 -1 -15.6) 1)    (71 (-1 1 0.1) 2)      (102 (-4 3 6.6) 2)
(10 (5 -4 0.8) 3)      (41 (2 -2 -0.8) 2)     (72 (-1 0 9.0) 3)      (103 (-4 2 -9.2) 1)
(11 (5 -5 -12.5) 2)    (42 (2 -3 -1.4) 1)     (73 (-1 -1 -1.7) 1)    (104 (-4 1 2.2) 2)
(12 (4 5 -22.0) 3)     (43 (2 -4 15.8) 3)     (74 (-1 -2 19.2) 3)    (105 (-4 0 -8.0) 1)
(13 (4 4 -14.2) 3)     (44 (2 -5 13.0) 2)     (75 (-1 -3 6.9) 2)     (106 (-4 -1 14.4) 3)
(14 (4 3 -7.8) 2)      (45 (1 5 -29.5) 2)     (76 (-1 -4 15.6) 2)    (107 (-4 -2 7.6) 3)
(15 (4 2 -4.6) 1)      (46 (1 4 -10.6) 3)     (77 (-1 -5 29.5) 2)    (108 (-4 -3 -4.1) 1)
(16 (4 1 0.6) 2)       (47 (1 3 -8.9) 1)      (78 (-2 5 -9.0) 3)     (109 (-4 -4 19.2) 2)
(17 (4 0 0.0) 2)       (48 (1 2 -2.2) 2)      (79 (-2 4 -4.8) 2)     (110 (-4 -5 24.0) 1)
(18 (4 -1 6.8) 3)      (49 (1 1 -0.3) 2)      (80 (-2 3 -16.6) 1)    (111 (-5 5 23.5) 3)
(19 (4 -2 11.2) 3)     (50 (1 0 10.0) 3)      (81 (-2 2 0.8) 2)      (112 (-5 4 13.2) 2)
(20 (4 -3 -6.6) 2)     (51 (1 -1 -0.1) 2)     (82 (-2 1 -13.4) 1)    (113 (-5 3 1.0) 1)
(21 (4 -4 -6.4) 2)     (52 (1 -2 0.6) 2)      (83 (-2 0 0.0) 2)      (114 (-5 2 7.4) 2)
(22 (4 -5 -3.0) 2)     (53 (1 -3 3.3) 2)      (84 (-2 -1 0.2) 2)     (115 (-5 1 19.3) 3)
(23 (3 5 -17.5) 3)     (54 (1 -4 -3.7) 1)     (85 (-2 -2 -2.5) 1)    (116 (-5 0 -18.0) 1)
(24 (3 4 -29.8) 1)     (55 (1 -5 34.5) 3)     (86 (-2 -3 18.8) 3)    (117 (-5 -1 10.7) 3)
(25 (3 3 -8.1) 2)      (56 (0 5 -30.0) 1)     (87 (-2 -4 17.6) 2)    (118 (-5 -2 0.6) 2)
(26 (3 2 -2.2) 2)      (57 (0 4 -6.8) 3)      (88 (-2 -5 25.0) 1)    (119 (-5 -3 6.9) 2)
(27 (3 1 0.1) 2)       (58 (0 3 13.6) 3)      (89 (-3 5 -5.4) 2)     (120 (-5 -4 18.8) 2)
(28 (3 0 0.0) 2)       (59 (0 2 12.4) 3)      (90 (-3 4 0.4) 2)      (121 (-5 -5 35.5) 1)
(29 (3 -1 -1.3) 2)     (60 (0 1 -0.2) 2)      (91 (-3 3 2.7) 2)
(30 (3 -2 5.3) 3)      (61 (0 0 0.0) 2)       (92 (-3 2 6.6) 3)
```