

Study of Self-Organizing Maps (SOM) Algorithm

Anshul Rao

Khoury College of Computer Sciences
Northeastern University
Boston, MA 02115
rao.ans@northeastern.edu

Abstract

I studied the self-organizing map algorithm and wrote a simple implementation in Python to demonstrate the basic intuition behind the algorithm. I also used SOM to perform topic modeling on a scraped textual dataset about stocks. Lastly, I discussed two recent papers that have used SOM and throw light on its use cases.

Introduction

A self-organizing map (SOM) is an unsupervised machine learning technique used to produce a low-dimensional (typically two-dimensional) representation of a higher dimensional data set while preserving the topological structure of the data. For example, a data set with p variables measured in n observations could be represented as clusters of observations with similar values for the variables. These clusters then could be visualized as a two-dimensional "map" such that observations in proximal clusters have more similar values than observations in distal clusters. This can make high-dimensional data easier to visualize and analyze.^[1] An SOM is a particular kind of artificial neural network, however unlike other artificial neural networks, it is taught via competitive learning as opposed to

error-correction learning (such as backpropagation with gradient descent). The SOM is frequently referred to as a Kohonen map or Kohonen network because it was developed by the Finnish academic Teuvo Kohonen in the 1980s.

Algorithm

We have two components, one is the input X and the second are the weights, which we will call W .

Input X is a $N \times D$ matrix. It is a matrix of N vectors v_1, v_2, \dots, v_N where each vector is of D dimensions.

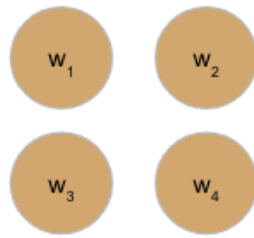
For example,

```
X = [[102, 179, 92],  
      [14, 106, 71],  
      [188, 20, 102],  
      [43, 123, 76]]
```

has $N = 4$ and $D = 3$ where $v_1 = [102, 179, 92]$, $v_2 = [14, 106, 71]$, $v_3 = [188, 20, 102]$ and $v_4 = [43, 123, 76]$.

The weight matrix W is basically a lattice of nodes or a map, each of them has a weight vector associated with it (which is of the same dimension as the input vector).

For example, a 2x2 lattice of nodes will look like below.



The key idea in the algorithm is that SOM performs competitive learning as opposed to error-correction learning by having units compete for the current object. During training, for each input vector v_i each node competes with the other as being able to represent v_i most adequately. The node (or weight vector) that best represents the input vector wins, and then that node and its neighbors are optimized to more closely resemble the v_i .

Step#1: *Initialize each node's weight vector.*

The most popular method for initializing the nodes is random initialization. It simply means to initialize the vectors of the nodes by iterating through each value that needs to be initialized and setting it to a random value. This strategy was proven to be effective for generating an equal distribution, but because of unpredictability, a random distribution will always be generated.

For example, randomly initiated w_1 , w_2 , w_3 and w_4 :

$$\begin{aligned}w_1 &= [3, 43, 12], \\w_2 &= [54, 132, 23], \\w_3 &= [120, 87, 94], \\w_4 &= [57, 20, 32]\end{aligned}$$

Step#2: *Randomly pick an input vector v from the input X .*

With the majority, if not all, neural networks, it is crucial that the samples are not continuously given to the network in the same order during training. Depending on the type of network, doing this can bias either the starting or ending input samples. So, a random pick is made.

For example, for the X mentioned earlier, let the randomly picked input vector be $v_1 = [102, 179, 92]$

Step #3: *For the input vector, find the best matching unit (BMU).*

BMU is the winning node and it is the one which is closest to the input vector. The most common way to find it is to use euclidean distance.

For example, for $v_1 = [102, 179, 92]$, the euclidean distances from each node are as follows:-

$$w_1 = \sqrt{(102 - 3)^2 + (179 - 43)^2 + (92 - 12)^2} = \sqrt{34697}$$

$$w_2 = \sqrt{10346}$$

$$w_3 = \sqrt{7390}$$

$$w_4 = \sqrt{14650}$$

Node w_3 is the BMU since it is closest to v_1 .

Step #4: *Get the neighborhood of BMU.*

Apart from the BMU, we also find the neurons that belong to the neighborhood set of BMU. They are also activated and the weight vectors of all activated neurons are adjusted towards the input vector (and not just the BMU). The closer a node is to the BMU, the more its weights get altered and the farther away the neighbor is from the BMU, the less it learns. This is controlled by a parameter called influence. Influence

decreases with time and it is greater for nodes closer to BMU and lesser for nodes far away.

Along with this, the radius of the neighborhood also decreases with time.



$$R(t + 1) = R(t)e^{-t/\lambda}$$

For example, in the above formula $R(t+1)$ is the radius at $(t+1)^{\text{th}}$ iteration and we can see how it exponentially decays over time.

$$I_k(t) = e^{-(d_k/2R(t))}$$

Similarly, $I_k(t)$ is the influence on k^{th} weight vector at time t where d_k is the distance of the weight vector from BMU.

Step#5: Update the weights of the BMU and its neighbors.

The winning weight receives compensation by resembling the sample vector more. Additionally, the weights of the neighbors are adjusted too so that they also start to resemble the sample vector.

$$w_k(t + 1) = w_k(t) + I_k(t)L(t)[v_l(t) - w_k(t)]$$

$L(t)$ is the learning rate which is used to control the amount of impact an input vector should have on the BMU and its neighbors. It also degrades over time like the radius of the neighborhood.

$$L(t + 1) = L(t)e^{-t/\lambda}$$

For example,

Let $L(0)$ be 0.5 (and $I(0) = 1$ for BMU), then our BMU's weight is updated as:

$$w_3 = [120, 87, 94] + 0.5([102, 179, 92] - [120, 87, 94])$$

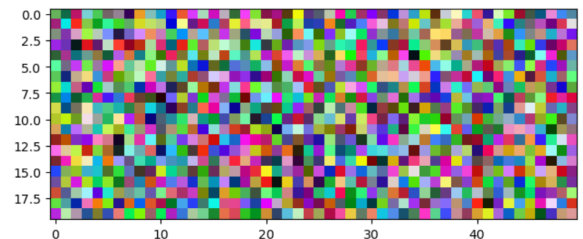
$\Rightarrow w_3 = [111, 133, 93]$ and as can be seen, w_3 is now more closer to v_1 (distance = $\sqrt{2198}$)

Step#6: Repeat Step#2 to Step#5 for N iterations.

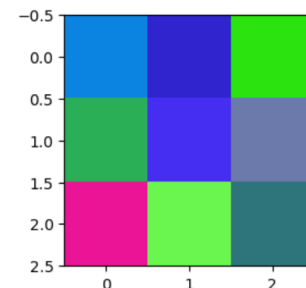
Simple SOM

Implementation in Python

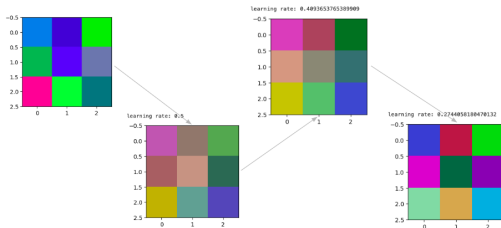
- We have 1000 input vectors (colors). Each vector is of dimension 3 (for each channel R, G and B)



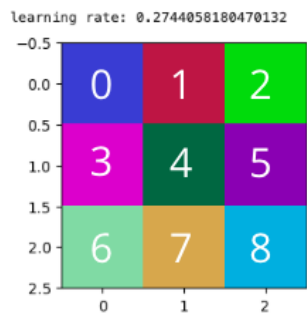
- We have a 2D lattice of 3x3 dimensions that is initialized randomly.




- We train the SOM model using the steps mentioned earlier for three epochs.
- The weight vectors change as below after each epoch:-




- The final lattice (weight vectors) looked like below and it can be seen as nine different clusters our input data of 1000 colors have been divided into.




- I tested three different input colors and they were assigned to correct clusters as below.

Hex: #	CC0000	
Red:	204	
Green:	0	
Blue:	0	

```
[25]: x_test = [204, 0, 0] # shade of red
[26]: find_bmu(x_test) # the node at index 1 in our final lattice above has dark red shade :)
[26]: 1
```

Hex: #	009900	
Red:	0	
Green:	153	
Blue:	0	

```
[27]: x_test = [0, 153, 0] # shade of green
[28]: find_bmu(x_test) # the node at index 4 in our final lattice above has dark green shade :)
[28]: 4
```

Hex: #	66B2FF	
Red:	102	
Green:	178	
Blue:	255	

```
[29]: x_test = [102, 178, 255] # shade of blue
[30]: find_bmu(x_test) # the node at index 8 in our final lattice above has bright blue shade :)
[30]: 8
```

Complete code for the demo can be found [here](#).

Python Libraries

There is no SOM module in scipy or other popular Python packages but there are some other open-source packages which are most popularly used and I tried using two of them MiniSom^[4] and sklearn-som^[5] to generate the weight vectors for same input data as in the demo illustrated in previous section. They can be found [here](#)

Topic Modeling using SOM

I also worked on some textual data that was scraped from Stocktwits (a social media platform designed for sharing ideas between investors, traders, and entrepreneurs) using a crawler I wrote in Python [here](#)

The data used are the posts from popular users (followed by many people on the platform) from 2018 to mid 2022.

The data was first cleaned and tokenized as follows:

- Removing hyperlinks.
- Removing stopwords.
- Removing tickers (all posts have \$<ticker-name> like a hashtag that tells which ticker the post is about.)
- Removing numbers and keeping only alphabetical words.
- Lemmatizing using WordNet.
- Removing posts of users that post programmatically (are like bots).

After cleaning the corpus where each post is like a document was converted into numerical vectors using TF-IDF. TF-IDF (term frequency-inverse document frequency) is a statistical measure that

evaluates how relevant a word is to a document in a collection of documents.

The TF-IDF score for the word t in the document d from the document set D is calculated as follows:

$$tf(t, d) = \log(1 + freq(t, d))$$

$$idf(t, D) = \log(N / count(d \in D : t \in d))$$

$$tf-idf(t, d, D) = tf(t, d) \cdot idf(t, D)$$

The TF-IDF vectors were used as input to the SOM. I chose a 3x3 lattice for the SOM model. I used the MiniSom library (mentioned in the previous section) and trained the model for 40k iterations.

Once the lattice/map was obtained, I extracted the top ten words in each node of the lattice and displayed them using WordCloud.

The lattice can be thought of as a topic model which is used for discovering the abstract "topics" that occur in a collection of documents.

The Jupyter notebook can be found [here](#)



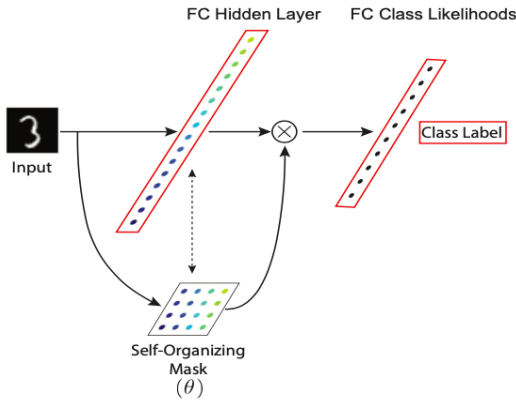
If we look at the topics above, then most of the words like stock, buy, earnings, volume, etc. belong to the financial domain which makes sense since the platform is for discussion between investors, traders, etc. I also analyzed each topic cluster to see which words were standing out (highlighted in yellow boxes). For example, if we look at topic 7, then we have words like vaccine and march (which is around when COVID pandemic exploded around the world in 2020) then it can be said that the seventh topic/cluster/node has some connection with posts written about or related to the pandemic.

Discussion about Latest Applications of SOM

A. Continual Learning with Self-Organizing Maps^[3]

In this paper, to tackle catastrophic forgetting of neural networks a memoryless method that combines standard supervised neural networks with self-organizing maps was proposed. When training on new tasks or categories, the neural networks forgets the information learned in previous tasks, this is called catastrophic forgetting. The role of the self-organizing map was to adaptively cluster the inputs into appropriate task contexts – without explicit labels – and allocate network resources accordingly. The map learns simultaneously with a supervised feedforward neural net in such a way that the SOM routes each

input sample to the most relevant part of the network. Unlike previous methods, SOMs do not require explicit information about the change in task, or an explicit memory of previous samples, while still achieving performance levels close to the current state-of-art on several benchmarks.



Above figure shows the SOMLP model setup. Red borders indicate supervised training. In the Self-Organized Multi-Layer Perceptron (SOMLP) an SOM layer is used in parallel with a fully-connected layer. The SOM receives the same input as the fully-connected layer and is trained without supervision on the network inputs. During the training it learns the input distribution for each task and, most importantly, a 2D map of relationships between the tasks. For every input, an output mask is computed based on the Euclidean distances between the input and SOM weight vectors, and is multiplied with the output of the fully connected layer. This allows the network to allocate resources (i.e.

nodes) to learning input-output associations for each task, and mask irrelevant nodes. In addition, the SOM shares nodes between similar tasks while using separate nodes for dissimilar tasks.

B. A Deterministic Self-Organizing Map Approach^[2]

In this paper, a deterministic SOM was proposed to eliminate the randomness of the standard self-organizing map. SOM is perplexed by its inherent randomness, which produces dissimilar SOM patterns even when being trained on identical training samples with the same parameters every time, and thus causes usability concerns for other domain practitioners and precludes more potential users from exploring SOM based applications in a broader spectrum.

Multiple ways to get rid of randomness were presented in this paper. For example, to eliminate the initialization randomness of the standard SOM method, we set up the nodes with a smooth transition from the top left to the bottom right corner. The gradient initialization computes the initial values of the weight vector of any node (a, b) in the following way:

$$d(1, 1)(a, b) / d(1, 1)(m, n)$$

where $d_{(1,1)(m,n)}$ is the maximum distance possible of the map, that is the distance from the top left node to the bottom right node.

Similarly, a staggered selection method that tries to maintain the good characteristics of randomness and eliminate the actual randomness is devised for selecting input sample vectors. The staggered approach gives all of the training samples equal opportunity to start an iteration at some point during the training. As well as giving the samples this equal opportunity it ensures that the samples are not shown to the learning algorithm in the same order during each training iteration. The staggered method of selection produces equivalent results to its random counterpart every time. This equivalence is guaranteed because the results are not random and therefore have the same performance every time.

Conclusion

SOM is a unique unsupervised algorithm with its own merits and demerits. One advantage of SOMs is that they can automatically cluster data, meaning that similar data points will be grouped together on the map. This makes it easy to see relationships and patterns in the data. Another advantage of SOMs is that they can be used to reduce the dimensionality of data, which can make it more manageable and easier to analyze. However, one disadvantage of SOMs is that they can be difficult to train, particularly for large or complex datasets. This can make them computationally expensive to use. Additionally, SOMs may not always

produce the most accurate results, particularly when working with noisy or non-linear data. Overall, the advantages of SOMs include their ability to visualize complex data, cluster data automatically, and reduce dimensionality. However, their disadvantages include the potential for computational expense and reduced accuracy for certain types of data.

While working on this project, I realized that the Python libraries present are not very actively maintained and they are not very advanced either and provide limited capabilities. This has motivated me to contribute to the open-source community and help enhance the robustness and add new features to the most popular SOM library.

All in all, it was exciting to learn about a new unsupervised algorithm that helps in visualization of data in lower dimensions and also helps in clustering. The idea of using competitive learning instead of error-correction was also very fascinating. I am sure that the insight and learning I obtained as part of this project will aid me in my future work in the long run.

References

1. Wikipedia: Self-organizing map, https://en.wikipedia.org/wiki/Self-organizing_map
2. A Deterministic Self-Organizing Map Approach and its Application on Satellite Data based Cloud Type Classification, <https://arxiv.org/pdf/1808.08315.pdf>

3. Continual Learning with Self-Organizing Maps,
<https://arxiv.org/pdf/1904.09330.pdf>
4. MiniSom,
<https://github.com/JustGlowing/minisom>
5. sklearn-som,
<https://github.com/rileypsmith/sklearn-som>
6. Research of fast SOM clustering for text information,
<http://140.127.22.205/100-2/ai/papers/SOM/a3.pdf>
7. Improved SOM Algorithm-HDSOM Applied in Text Clustering,
<https://ieeexplore.ieee.org/document/567083>

Appendix

- Stocktwits Crawler:
<https://github.com/anshulrao/stocktwits-crawler>
- SOM code for the project:
<https://github.com/anshulrao/som>