Study of SOM (Self-Organizing Maps) Algorithm

By: Anshul Rao

What is SOM ?

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.

Algorithm

Input **X** : N x D matrix of **N** vectors v_1, v_2, ... v_N of **D** dimensions.

For example, X =

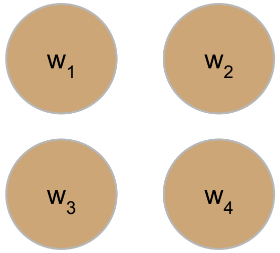
102	179	92
14	106	71
188	20	102
43	123	76

has N = 4 and D = 3 where v_1 = [102, 179, 92] ...

Algorithm

Weights **W**: We have a lattice of nodes, each of them has a weight vector associated with it (which is of same dimension as the input vector).

For example, a 2 x 2 lattice of nodes will look like:-



Algorithm

Key Idea: “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 wins, that node and its neighbours are optimized to more closely resemble the v_i

As the algorithm progresses (and eventually terminates), each of the nodes resemble a certain class/category of input vectors.

Algorithm

Step #1:

Initialize each node’s weight vector.

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

$w_1 = [3, 43, 12]$

$w_2 = [54, 132, 23]$

$w_3 = [120, 87, 94]$

$w_4 = [57, 20, 32]$

Algorithm

Step #2:

For each input vector (chosen at random), **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 do find it is to use euclidean distance.

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

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

$w_2 = \sqrt{10346}$

$w_3 = \sqrt{7390}$

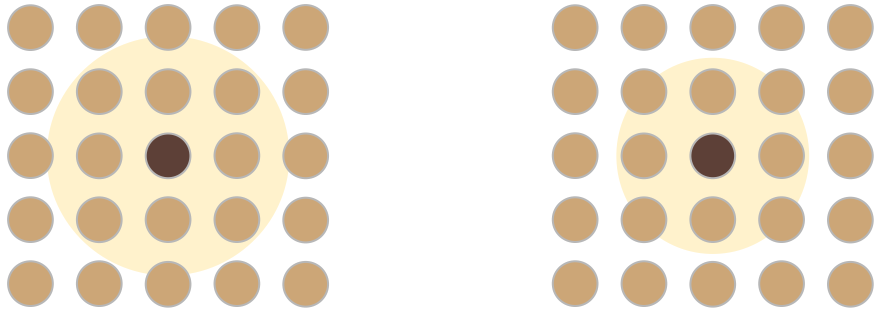
$w_4 = \sqrt{14650}$

Node w_3 is the BMU!

Algorithm

Step #3:

Get the **neighbourhood of BMU** (the neighbourhood decreases over time). 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.



NOTE: The radius of the neighbourhood decreases with time.

Algorithm

Step #4:

Adjust the weights of all nodes in the neighbourhood of BMU (including the BMU).

The equation is to update weight of node w_j using input vector v_i at (t+1)th iteration is:

$$\mathbf{w_j(t+1) = w(t) + I_j(t)L(t)[v_i(t) - w_j(t)]}$$

L is the learning rate (to control the amount of impact) and I is the influence (adjusts the learning rate of a neighborhood node) which decreases with time.

For example, let L(0) be 0.5 (and I(0) = 1 for BMU), then our BMU w_3 = [120, 87, 94] + 0.5([102, 179, 92] - [120, 87, 94])

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

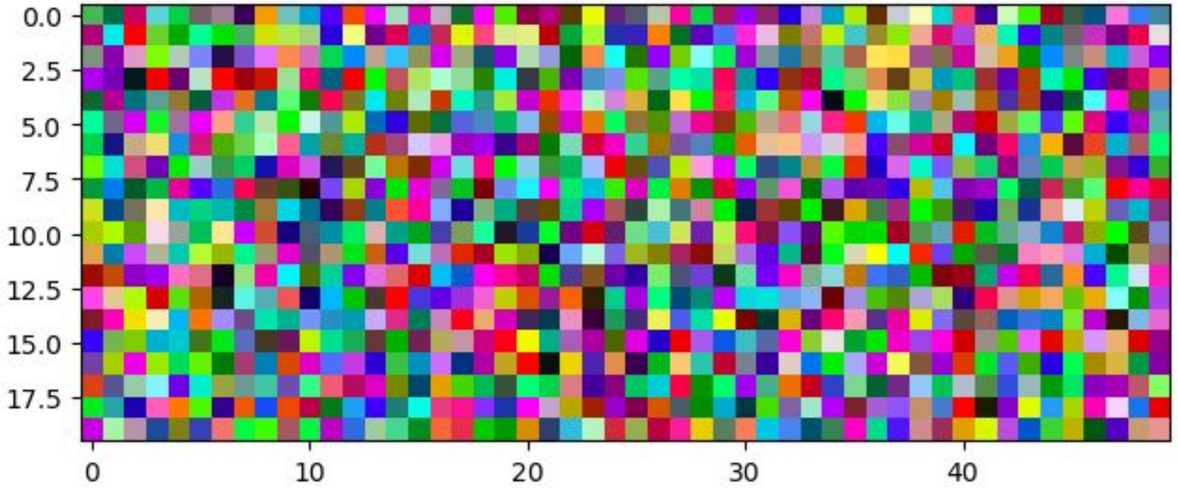
NOTE: Repeat Step#2 to Step#4 for N iterations.

DEMO

X = 1000 x 3 matrix with each input vector being a color.

```
[9]: io.imshow(np.uint8(X.reshape(20, len(X)//20, 3)))
```

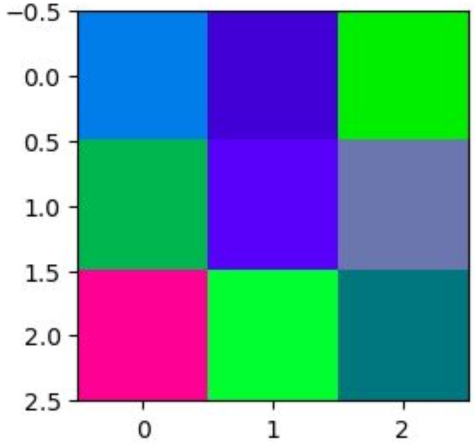
```
[9]: <matplotlib.image.AxesImage at 0x2b56202e6a30>
```



DEMO

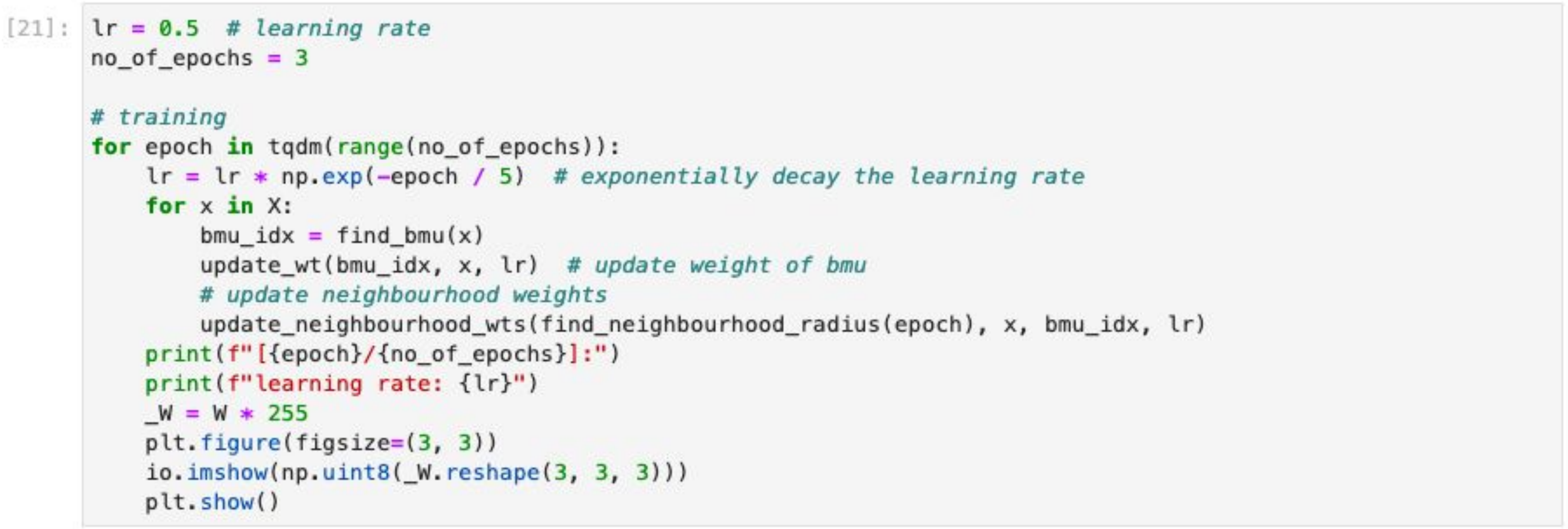
W = 3 x 3 lattice with 9 nodes. They can be shown as colors too.

```
[12]: # initialized lattice wihth random weights
      plt.figure(figsize=(3, 3))
      io.imshow(np.uint8(W.reshape(3, 3, 3)))

[12]: <matplotlib.image.AxesImage at 0x2b56203fe940>
```



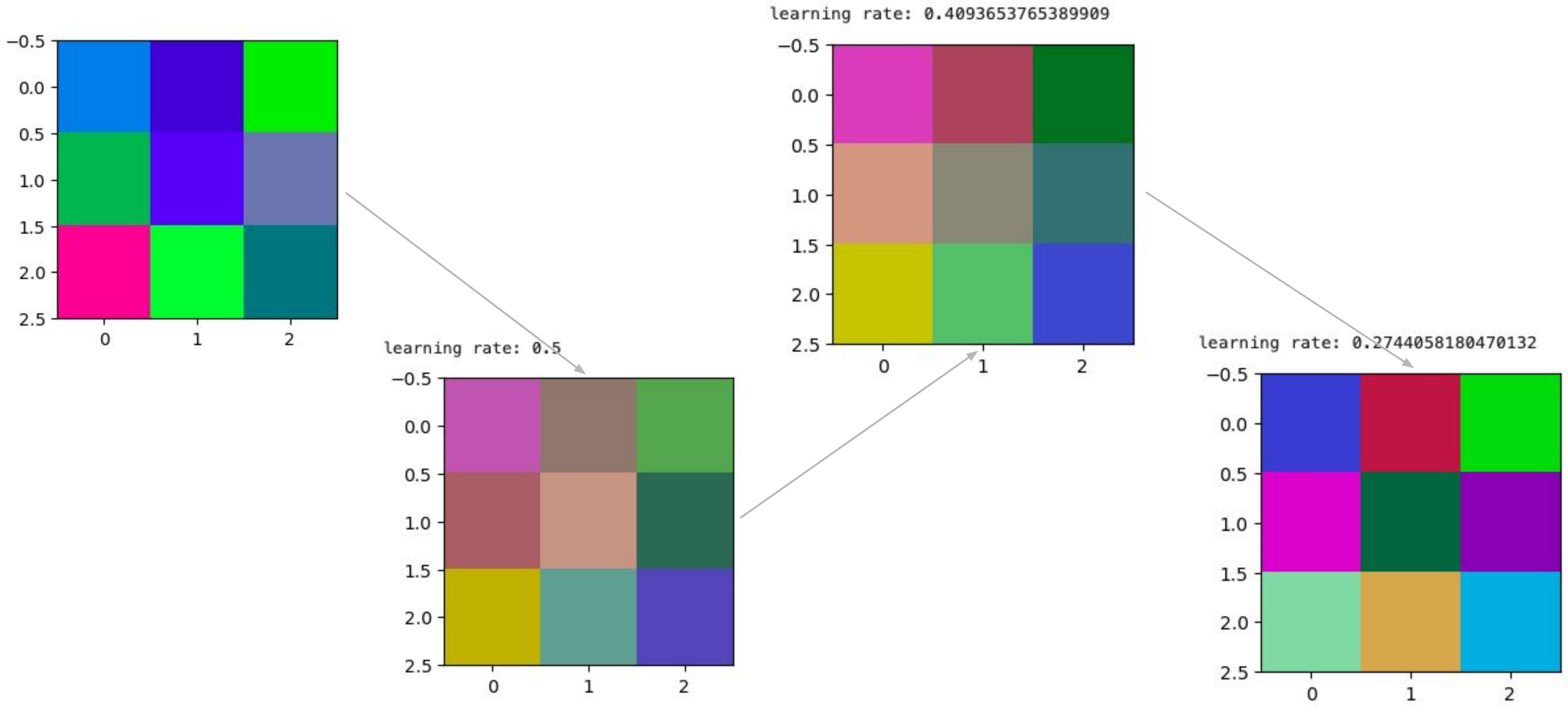
```
array([[0.04313725, 0.51764706, 0.88235294],
       [0.18431373, 0.14117647, 0.80784314],
       [0.16862745, 0.89019608, 0.05490196],
       [0.16470588, 0.68627451, 0.3372549 ],
       [0.27058824, 0.17647059, 0.94509804],
       [0.41960784, 0.47843137, 0.66666667],
       [0.9254902 , 0.07843137, 0.58823529],
       [0.41568627, 0.96470588, 0.30980392],
       [0.18039216, 0.45490196, 0.48235294]])
```

DEMO

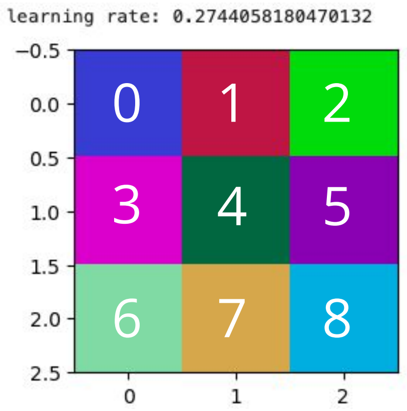


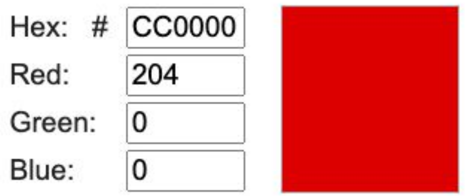
```
[21]: lr = 0.5  # learning rate
      no_of_epochs = 3

      # training
      for epoch in tqdm(range(no_of_epochs)):
          lr = lr * np.exp(-epoch / 5)  # exponentially decay the learning rate
          for x in X:
              bmu_idx = find_bmu(x)
              update_wt(bmu_idx, x, lr)  # update weight of bmu
              # update neighbourhood weights
              update_neighbourhood_wts(find_neighbourhood_radius(epoch), x, bmu_idx, lr)
          print(f"[{epoch}/{no_of_epochs}]:")
          print(f"learning rate: {lr}")
          _W = W * 255
          plt.figure(figsize=(3, 3))
          io.imshow(np.uint8(_W.reshape(3, 3, 3)))
          plt.show()
```

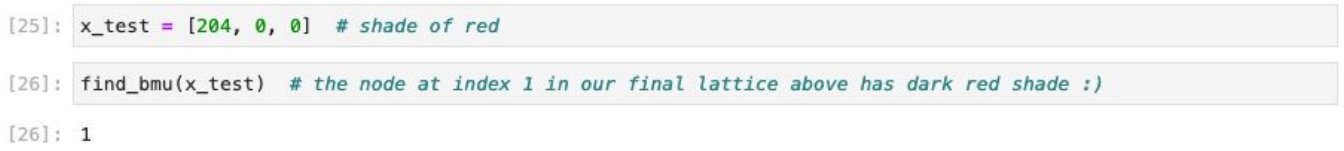
DEMO


learning rate: 0.5
learning rate: 0.4093653765389909
learning rate: 0.2744058180470132

DEMO

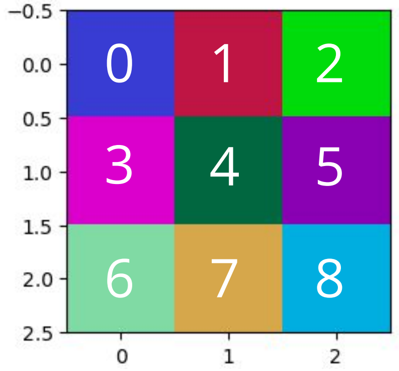


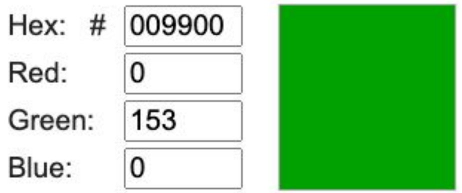

Hex: # CC0000
Red: 204
Green: 0
Blue: 0


```
[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
```

DEMO

```
learning rate: 0.2744058180470132
```



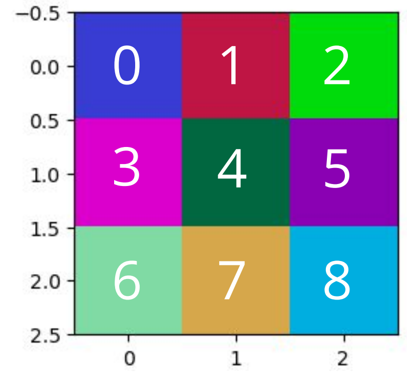


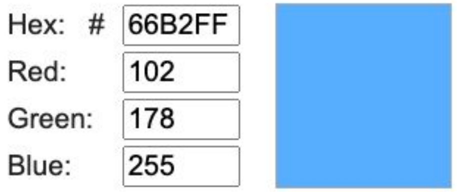
Hex: # 009900
Red: 0
Green: 153
Blue: 0

```
[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
```

DEMO

learning rate: 0.2744058180470132

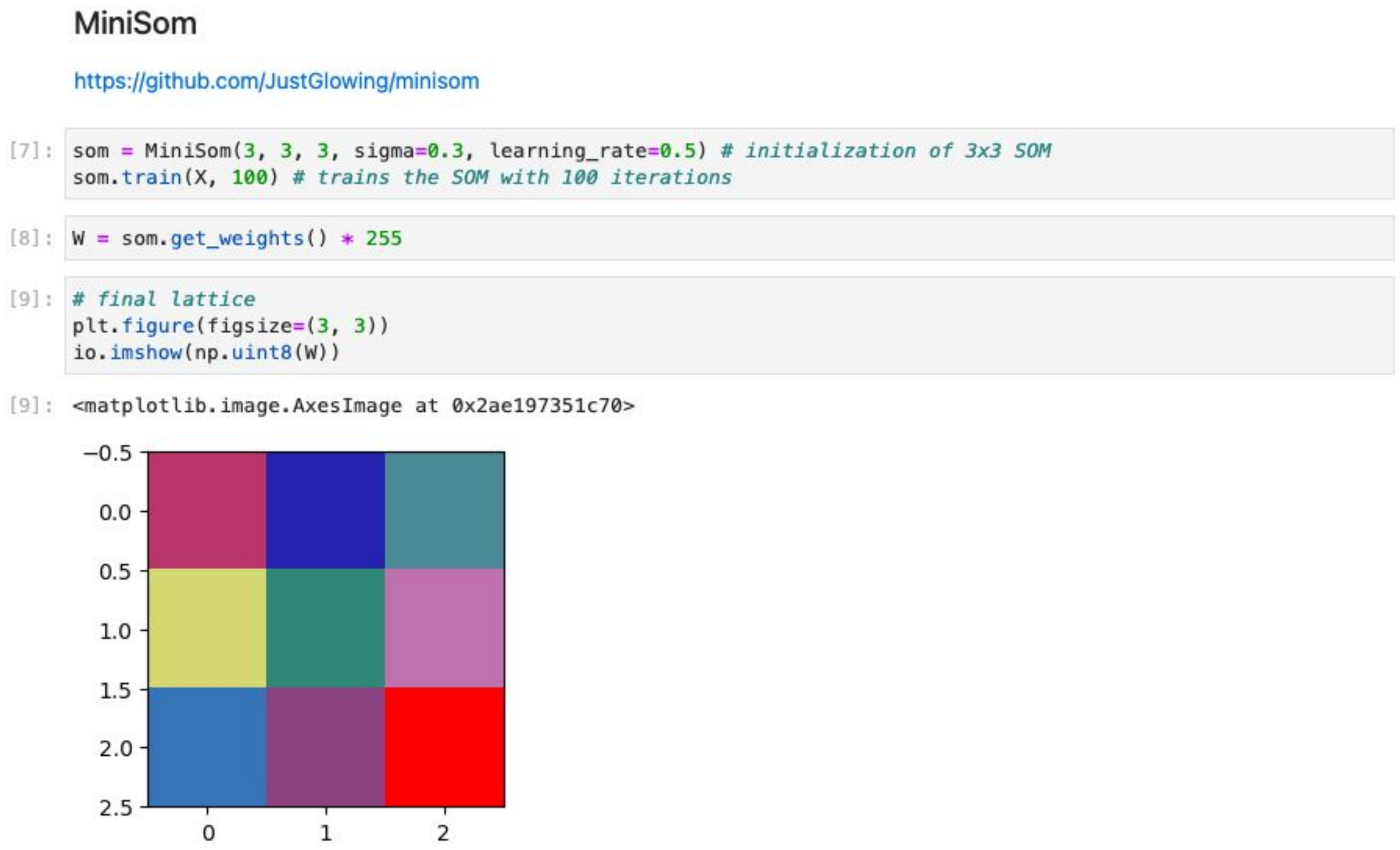




Hex: #	66B2FF
Red:	102
Green:	178
Blue:	255

```
[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
```

Libraries



MiniSom

https://github.com/JustGlowing/minisom

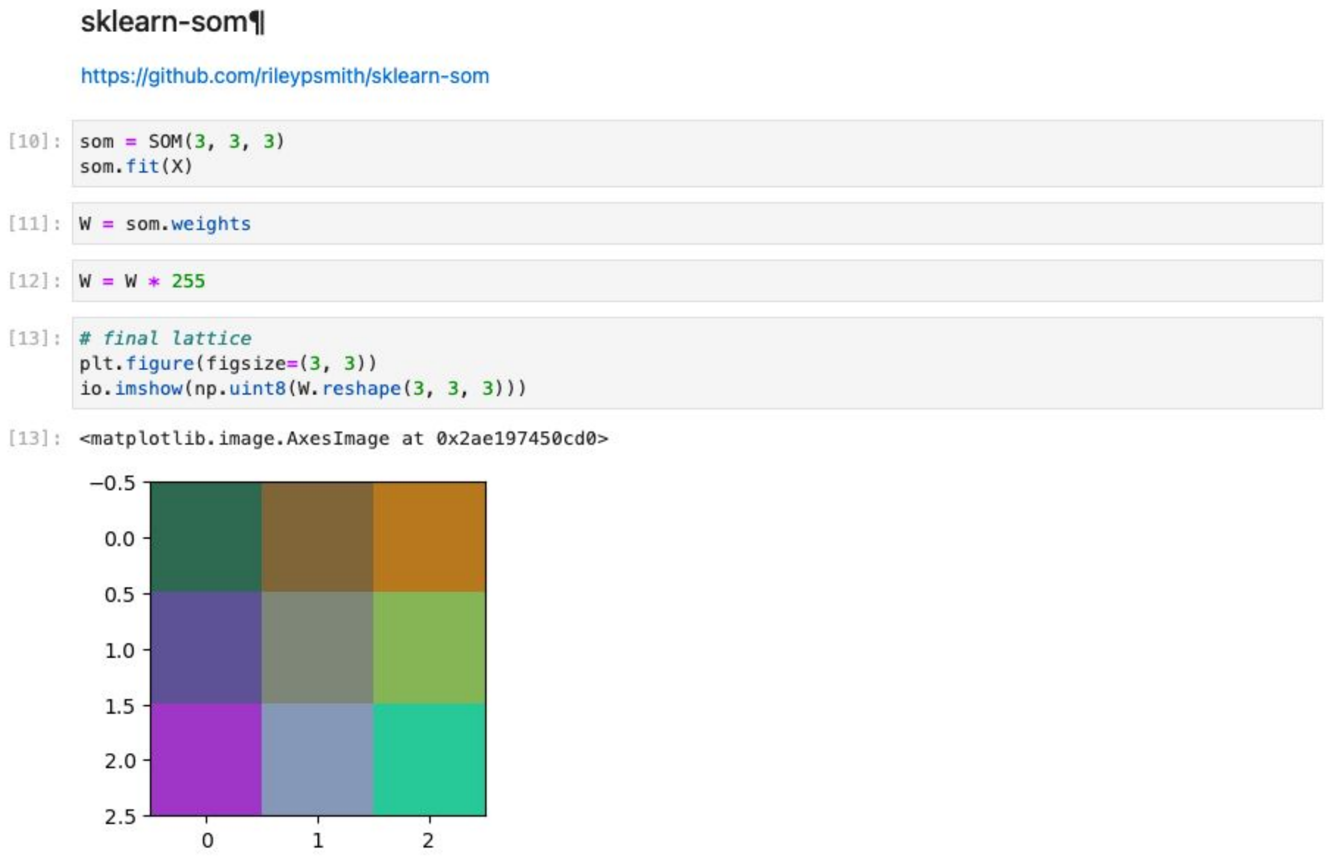
```
[7]: som = MiniSom(3, 3, 3, sigma=0.3, learning_rate=0.5) # initialization of 3x3 SOM
     som.train(X, 100) # trains the SOM with 100 iterations
```

```
[8]: W = som.get_weights() * 255
```

```
[9]: # final lattice
     plt.figure(figsize=(3, 3))
     io.imshow(np.uint8(W))
```

```
[9]: <matplotlib.image.AxesImage at 0x2ae197351c70>
```

Libraries



sklearn-som¶

https://github.com/rileypsmith/sklearn-som

```
[10]: som = SOM(3, 3, 3)
      som.fit(X)
```

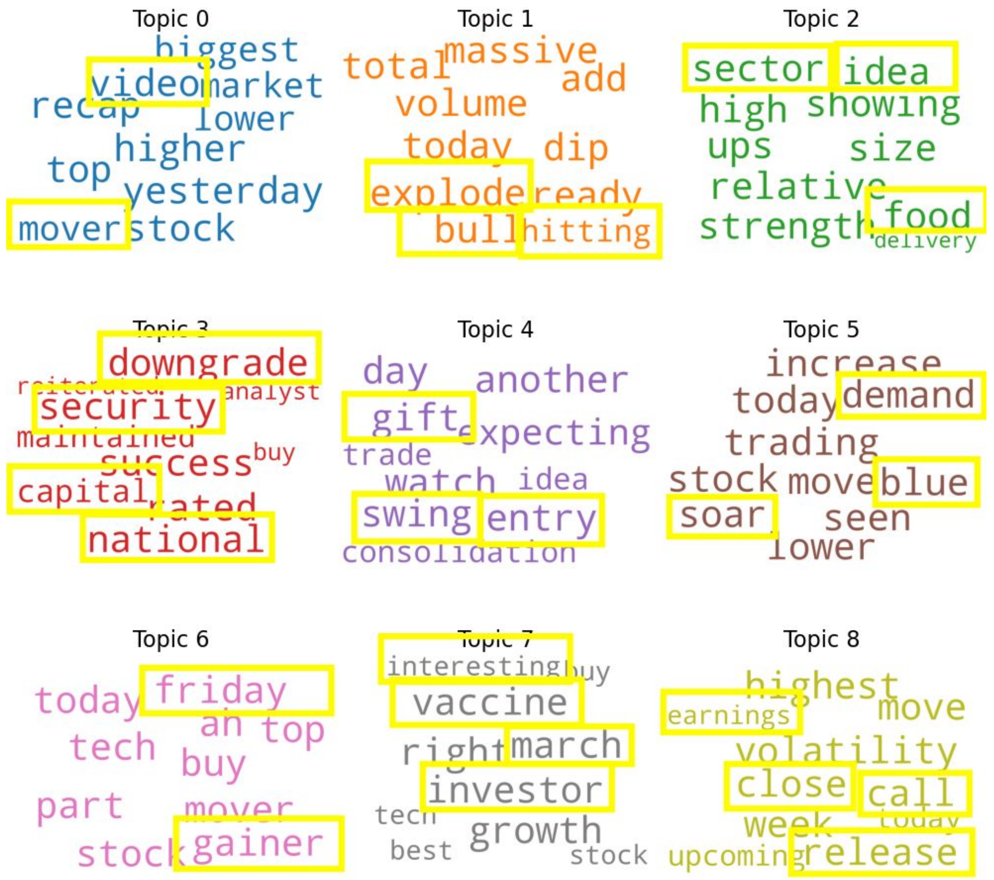
```
[11]: W = som.weights
```

```
[12]: W = W * 255
```

```
[13]: # final lattice
      plt.figure(figsize=(3, 3))
      io.imshow(np.uint8(W.reshape(3, 3, 3)))
```

```
[13]: <matplotlib.image.AxesImage at 0x2ae197450cd0>
```

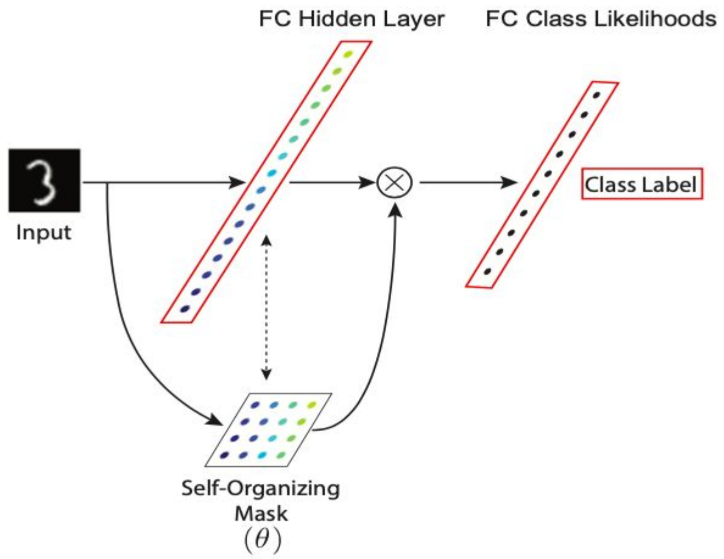
Topic Modeling



Recents Applications

Continual Learning with Self-Organizing Maps

- Deal with catastrophic forgetting in neural networks by using SOM to adaptively cluster the inputs into appropriate task contexts.
- SOM employs unsupervised competitive learning in order to map similar input vectors to physically nearby nodes in the map layer.



Recents Applications

A Deterministic Self-Organizing Map Approach

- Inherent randomness of SOM produces dissimilar SOM patterns even when being trained on identical training samples with the same parameters every time.
- Initialization Method: Instead of random initialization of weights, set up the nodes with a smooth transition from the top left to the bottom right corner. For example, w_i at (a, b) is initialized as

  $$\frac{d_{(1,1)(a,b)}}{d_{(1,1)(m,n)}}$$

  where $d_{(1,1)(m,n)}$ is the maximum distance possible of the map, i.e., the distance from the top left node to the bottom right node.
- Sample Selection: A staggered selection method is used that gives all of the training samples equal opportunity to start an iteration at some point during the training and ensures that the samples are not shown to the learning algorithm in the same order during each training iteration.

https://arxiv.org/pdf/1808.08315.pdf

Advantages/Disadvantages

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.

References

- Wikipedia: Self-organizing map, https://en.wikipedia.org/wiki/Self-organizing_map
- A Deterministic Self-Organizing Map Approach and its Application on Satellite Data based Cloud Type Classification, https://arxiv.org/pdf/1808.08315.pdf
- Continual Learning with Self-Organizing Maps, https://arxiv.org/pdf/1904.09330.pdf
- somoclu: https://somoclu.readthedocs.io/en/stable/
- sklearn-som: https://pypi.org/project/sklearn-som/
- Research of fast SOM clustering for text information, http://140.127.22.205/100-2/ai/papers/SOM/a3.pdf
- Improved SOM Algorithm-HDSOM Applied in Text Clustering, https://ieeexplore.ieee.org/document/567083