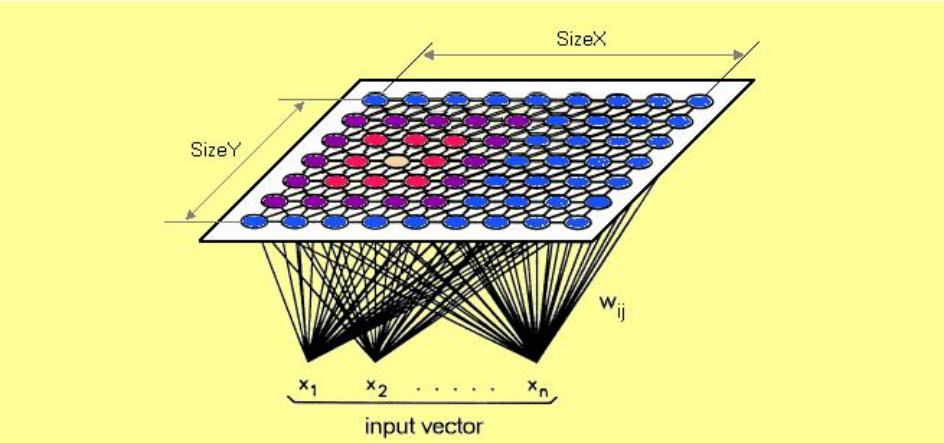
Self Organizing Map

- Manav Kaushik

Introduction

A self-organizing map (SOM) is a type of artificial neural network (ANN) that is trained using unsupervised and competitive learning to produce low dimensional, discretized representation of presented high dimensional data, while simultaneously preserving similarity relations between the presented data items. Such low dimensional representation is called a feature map.SOMs map multidimensional data onto lower-dimensional subspaces where geometric relationships between points indicate their similarity and are therefore used in dimensionality reduction, data visualization, and clustering. Self-organizing maps differ from other artificial neural networks as they apply competitive learning(winner takes all) as opposed to error-correction learning for updating its weight. SOM is a grid of neurons that each contain a weight vector of the same dimensionality as the input vector.



The goal of learning in the self-organizing map is to cause different parts of the network to respond similarly to certain input patterns. This is partly motivated by how visual, auditory, or other sensory information is handled in separate parts of the human brain. In this method, a training vector sample is selected from the input space and the map of weight vectors is searched to find which weight best represents that sample, also called the winning neuron. The weight that is chosen is rewarded by being able to become more like that randomly selected sample vector. In this way, the map changes its shape and grows.

Steps involved in the algorithm:

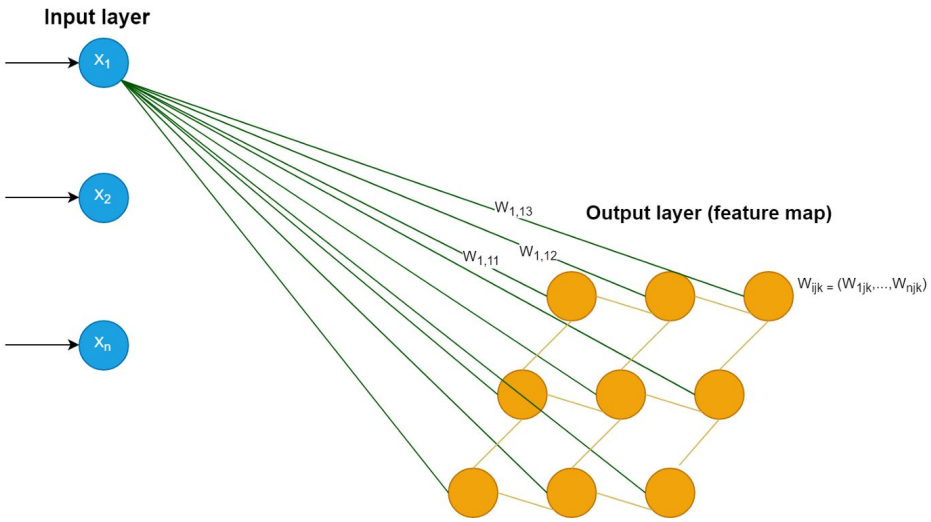
1. Each data point weight is initialized.
2. A vector is chosen at random from the set of training data.
3. Every node is examined to determine which ones are closer to the input vector by calculating the distance between them. The winning node is commonly known as the Best Matching Unit (BMU).
4. Then the neighborhood of the BMU is calculated. The amount of neighbors decreases over time.
5. The winning weight is rewarded by converting it so that it is like the sample vector. The neighbors also become more like the sample vector. The closer a node is to the BMU, the more its weights get altered.
6. Repeat step 2 for N iterations.

Competitive Learning

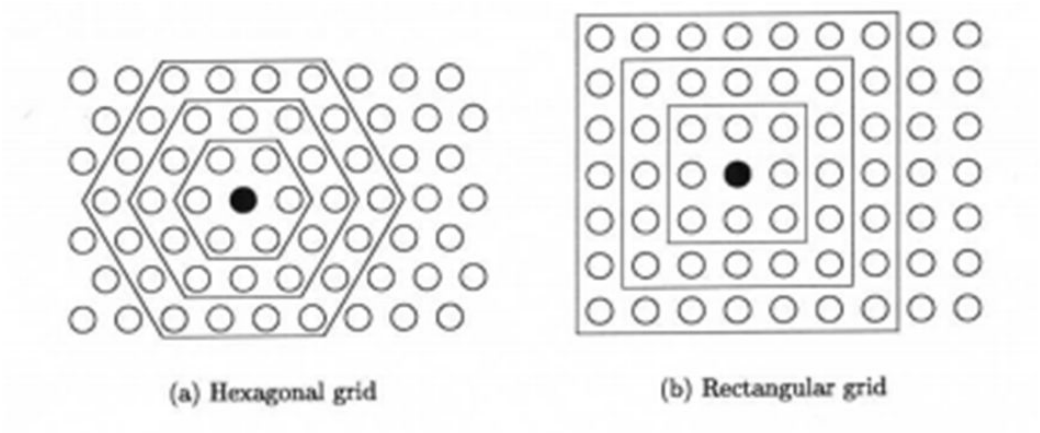
Competitive Learning is based on three processes:

1. Competition
2. Cooperation
3. Adaptation

1. **Competition:** Each neuron in a SOM is assigned a weight vector with the same dimensionality as the input space. The distance between each neuron (neuron from the output layer) and the input data is calculated, and the neuron with the lowest distance is the winner of the competition. The Euclidean metric is commonly used to measure distance.
   In the example below, in each neuron of the output layer, we will have a vector with dimension n.



2. **Cooperation:** We update the vector of the winner neuron along with its neighbors in the final process (adaptation). The neighbors are chosen using the kernel functions. This function depends on two factors: time (incremented with each new input data) and distance between the winner neuron and the neighbor neuron. The image below shows us how the winner neuron's neighbors are chosen depending on distance and time factors.

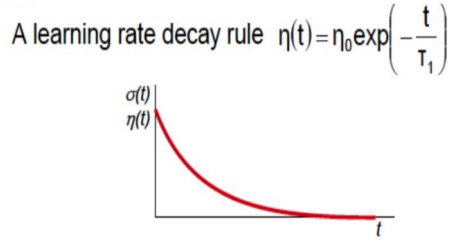


Images representing different topology of the winning neighbour in SOM

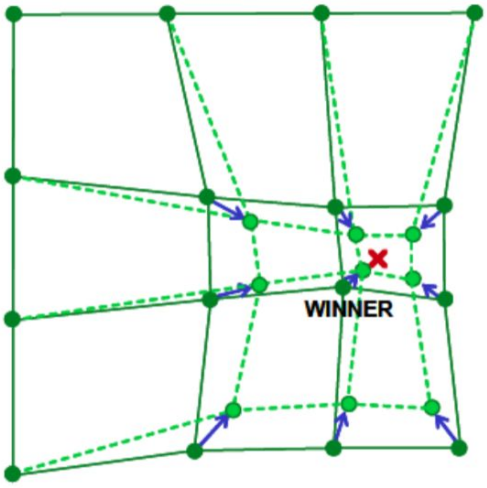
3. **Adaptation:** After choosing the winner neuron and its neighbors, the weight update is calculated. Those chosen neurons will be updated but not by the same magnitude- more the distance between the neuron and the input data, less is the update. The weights are updated using the formula:

$$\Delta w_{ij} = \begin{cases} \alpha\,(x_i - w_{ij}), & \text{if neuron } j \text{ wins the competition} \\ 0, & \text{if neuron } j \text{ loses the competition} \end{cases}$$

where x_i is the input and w_{ij} is the associated weight.

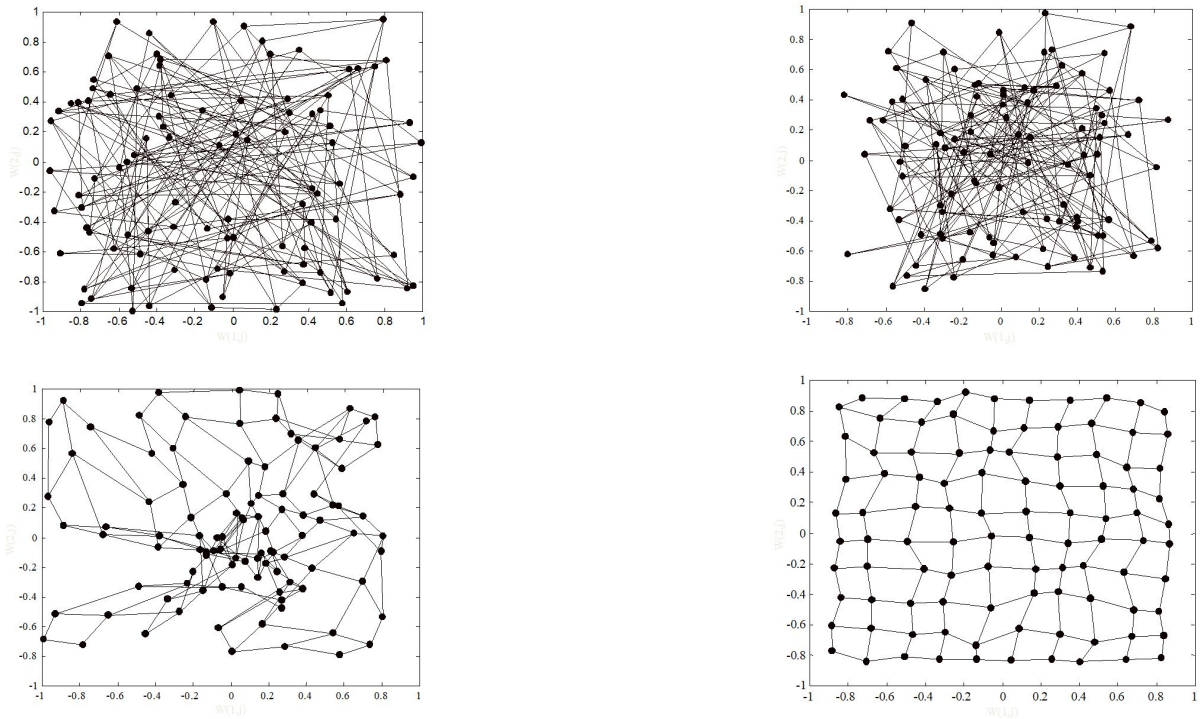


This learning rate indicates how much we want to adjust our weights. After some positive infinite time, this learning rate will converge to zero so we will have no update even for the winner neuron.



Visualisation of the adaptation process in competitive learning

Visualization of SOM for a dataset in different iterations: The output of the algorithm, in this case, is 2-dimensional.



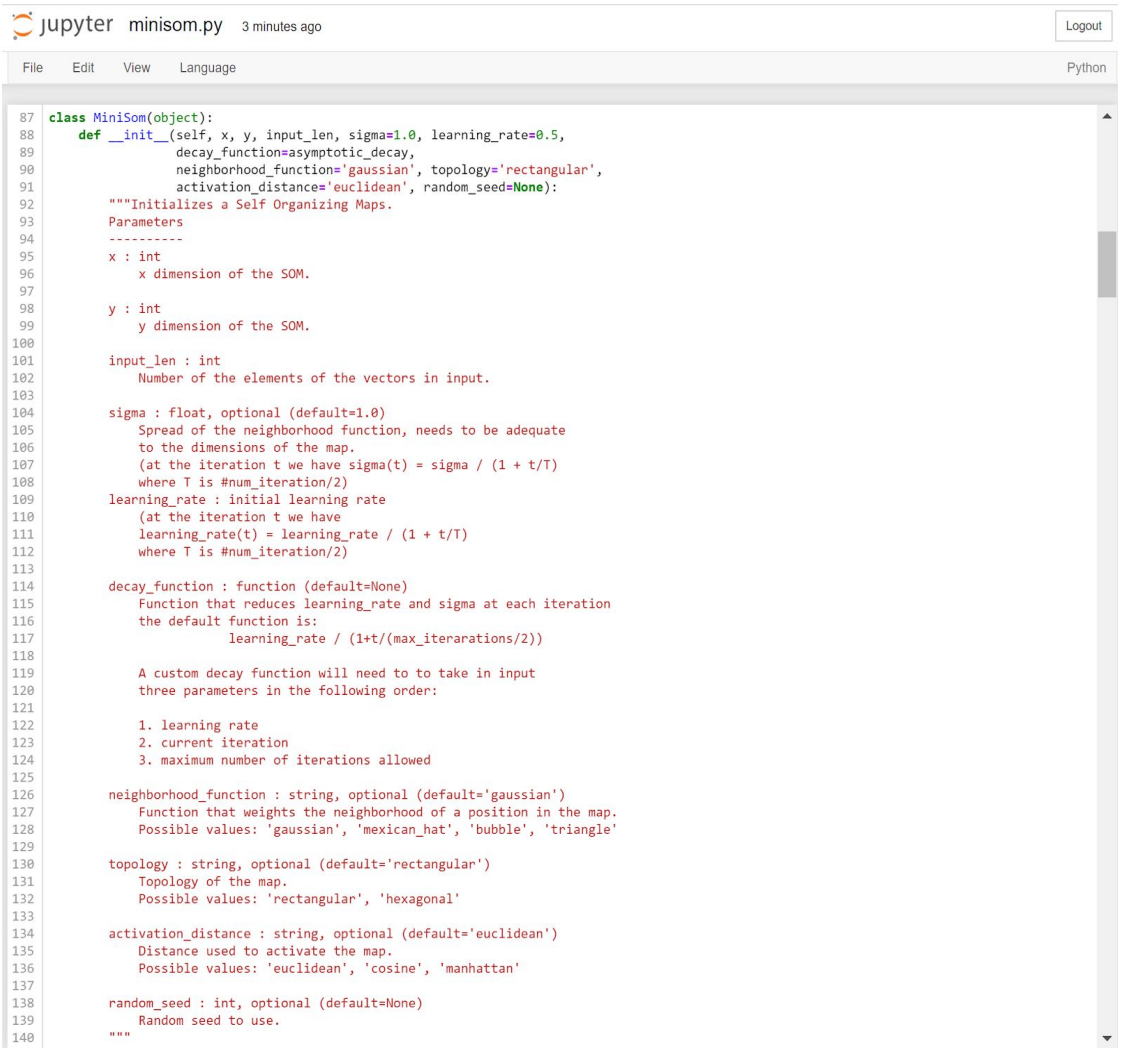
Visualisation of the SOM algorithm (1) initially (2) after 100 iterations (3) after 1000 iterations (4) after 10000 iterations

Implementation of SOM in python

We have implemented SOM on a high dimensional dataset UCI Wine dataset having 13 dimensions. The dataset has a target variable of the kind of wine having 3 classes. There are a total of 178 data points in the dataset. Our objective is to classify these points into the 3 classes and also to visualize the dataset in 2-dimensions using SOM hexagonal grids.

The attributes of the dataset are 1) Alcohol (Target variable) 2) Malic acid 3) Ash 4) Alkalinity of ash 5) Magnesium 6) Total phenols 7) Flavonoids 8) Non Flavonoid phenols 9) Proanthocyanins 10)Color intensity 11)Hue 12)OD280/OD315 of diluted wines 13)Proline. All the attributes are real and continuous in nature.

We began by implementing a function for creating a SOM architecture which takes the following parameters:




```python
class MiniSom(object):
    def __init__(self, x, y, input_len, sigma=1.0, learning_rate=0.5,
                 decay_function=asymptotic_decay,
                 neighborhood_function='gaussian', topology='rectangular',
                 activation_distance='euclidean', random_seed=None):
        """Initializes a Self Organizing Maps.
        Parameters
        ----------
        x : int
            x dimension of the SOM.

        y : int
            y dimension of the SOM.

        input_len : int
            Number of the elements of the vectors in input.

        sigma : float, optional (default=1.0)
            Spread of the neighborhood function, needs to be adequate
            to the dimensions of the map.
            (at the iteration t we have sigma(t) = sigma / (1 + t/T)
            where T is #num_iteration/2)
        learning_rate : initial learning rate
            (at the iteration t we have
            learning_rate(t) = learning_rate / (1 + t/T)
            where T is #num_iteration/2)

        decay_function : function (default=None)
            Function that reduces learning_rate and sigma at each iteration
            the default function is:
                       learning_rate / (1+t/(max_iterarations/2))

            A custom decay function will need to to take in input
            three parameters in the following order:

            1. learning rate
            2. current iteration
            3. maximum number of iterations allowed

        neighborhood_function : string, optional (default='gaussian')
            Function that weights the neighborhood of a position in the map.
            Possible values: 'gaussian', 'mexican_hat', 'bubble', 'triangle'

        topology : string, optional (default='rectangular')
            Topology of the map.
            Possible values: 'rectangular', 'hexagonal'

        activation_distance : string, optional (default='euclidean')
            Distance used to activate the map.
            Possible values: 'euclidean', 'cosine', 'manhattan'

        random_seed : int, optional (default=None)
            Random seed to use.
        """
```

- **x**: x-dimension of the SOM in 2-dimensions. We have chosen it as 14 for the wine dataset for better visualization.
- **y**: y-dimension of the SOM in 2-dimensions. We have chosen it as 14 for the wine dataset for better visualization.
- **input_len**: No. of dimensions in the input. For our dataset, we have it as 13.
- **Sigma:** spread defined the spread of the neighborhood function.
- **learning _rate:** This parameter takes the initial learning rate which decays with time. For our case, we have taken it as 0.7 after optimizing.
- **activation_distance:** This parameter defines the distance metric used to calculate the distance between the input vector and the weight vector. For our case, we have taken it as euclidean.

- **topology:** This parameter defines the topology of the SOM. In the question, we were asked to make hexagonal topology for the SOM.
- **neighbour_function:** It is the function that weights the neighborhood of a neuron in the SOM.

Some important functions in class Minisom:

- **Train_batch():** This is used for training the model with our 13-dimensional input data along with the specified number of iterations.

```
403
404     def train_batch(self, data, num_iteration, verbose=False):
405         """Trains the SOM using all the vectors in data sequentially.
406
407         Parameters
408         ----------
409         data : np.array or list
410             Data matrix.
411
412         num_iteration : int
413             Maximum number of iterations (one iteration per sample).
414
415         verbose : bool (default=False)
416             If True the status of the training
417             will be printed at each iteration.
418         """
419         self.train(data, num_iteration, random_order=False, verbose=verbose)
420
```

- **Get_euclidean_coordinates():** This function returns the position of the neurons on a euclidean plane that reflects the chosen topology in two mesh grids xx and yy. Neuron with map coordinates (1, 4) has coordinates (xx[1, 4], yy[1, 4]) in the euclidean plane.

```
204
205     def get_euclidean_coordinates(self):
206         return self._xx.T, self._yy.T
207
```

- **distance_map():** This function is used for calculating the euclidean distance map of the weights from the inputs.

```
    def distance_map(self):
        """Returns the distance map of the weights.
        Each cell is the normalised sum of the distances between
        a neuron and its neighbours. Note that this method uses
        the euclidean distance."""
        um = zeros((self._weights.shape[0],
                    self._weights.shape[1],
                    8))  # 2 spots more for hexagonal topology

        ii = [[0, -1, -1, -1, 0, 1, 1, 1]]*2
        jj = [[-1, -1, 0, 1, 1, 1, 0, -1]]*2

        if self.topology == 'hexagonal':
            ii = [[1, 1, 1, 0, -1, 0], [0, 1, 0, -1, -1, -1]]
            jj = [[1, 0, -1, -1, 0, 1], [1, 0, -1, -1, 0, 1]]

        for x in range(self._weights.shape[0]):
            for y in range(self._weights.shape[1]):
                w_2 = self._weights[x, y]
                e = y % 2 == 0   # only used on hexagonal topology
                for k, (i, j) in enumerate(zip(ii[e], jj[e])):
                    if (x+i >= 0 and x+i < self._weights.shape[0] and
                            y+j >= 0 and y+j < self._weights.shape[1]):
                        w_1 = self._weights[x+i, y+j]
                        um[x, y, k] = fast_norm(w_2-w_1)

        um = um.sum(axis=2)
        return um/um.max()
```

- **get_weights():** This function returns the weights of the SOM architecture.

```
    def get_weights(self):
        return self._weights
```

- **winner():** This function returns the winner neuron on taking an input x.

```
    def winner(self, x):
        self._activate(x)
        return unravel_index(self._activation_map.argmin(),
                             self._activation_map.shape)
```

- **convert_map_to_euclidean():** This function converts SOM map coordinates into euclidean coordinates that reflects the chosen topology (here, hexagonal)

```
214     def convert_map_to_euclidean(self, xy):
215         
216         return self._xx.T[xy], self._yy.T[xy]
```

Implementation steps:

1. **Data preparation:** We first import the required libraries.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
```

We also import the libraries required for plotting the hexagonal grids in 2-dimensional.

```
from matplotlib.patches import RegularPolygon, Ellipse
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib import cm, colorbar
from matplotlib.lines import Line2D
```

We then import our required dataset.

```
In [2]: df = pd.read_csv('wine.csv')
        df
```

Out[2]:

	Wine	Alcohol	Malic.acid	Ash	Acl	Mg	Phenols	Flavanoids	Nonflavanoid.phenols	Proanth	Color.int	Hue	OD	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
173	3	13.71	5.65	2.45	20.5	95	1.68	0.61	0.52	1.06	7.70	0.64	1.74	740
174	3	13.40	3.91	2.48	23.0	102	1.80	0.75	0.43	1.41	7.30	0.70	1.56	750
175	3	13.27	4.28	2.26	20.0	120	1.59	0.69	0.43	1.35	10.20	0.59	1.56	835
176	3	13.17	2.59	2.37	20.0	120	1.65	0.68	0.53	1.46	9.30	0.60	1.62	840
177	3	14.13	4.10	2.74	24.5	96	2.05	0.76	0.56	1.35	9.20	0.61	1.60	560

We then separate the predictor and the target variables.

```
y=df['Wine'].values
df.drop(['Wine'], 1, inplace=True)
X=np.array(df)
print(X.shape)
```

```
(178, 13)
```

2. **Data pre-processing:** We standardize the input variables by making the mean as 0 and standard deviation as 1.

```
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler().fit(X)
newX=scaler.transform(X)
```

3. **Applying Minisom class developed above:** We applied the MiniSom class with x-dimensions as 14, y-dimensions as 14, number of input features as 13, initial learning rate as 0.7, distance metric as euclidean, topology as hexagonal, and gaussian neighbourhood function.

```
som = MiniSom(14, 14, 13, sigma=1.5, learning_rate=.7, activation_distance='euclidean',
              topology='hexagonal', neighborhood_function='gaussian', random_seed=10)

som.train_batch(newX, 1000, verbose=True)
```

```
[  923 / 1000 ]  92% - 0:00:00 left

[ 1000 / 1000 ] 100% - 0:00:00 left
quantization error: 1.3885785313546974
```

4. **Identifying the winner and weight update:** We traverse through each data point and assign a winner neuron using som.winner(x). We then update the weights using som.convert_map_to_euclidean(w) according to the hexagonal topology.

```
for cnt, x in enumerate(newX):
    #print(cnt,x)
    w = som.winner(x)  # getting the winner
    # palce a marker on the winning position for the sample xx
    wx, wy = som.convert_map_to_euclidean(w)
    wy = wy*2/np.sqrt(3)*3/4
    plt.plot(wx, wy, markers[y[cnt]], markerfacecolor='None',
             markeredgecolor=colors[y[cnt]], markersize=12, markeredgewidth=2)
```

5. **Plotting the 2-D feature map having hexagonal grids:** We have represented the 3 classes using different markers and different colours.

```
markers = ['o', '+', 'x']
colors = ['C0', 'C1', 'C2']

xrange = np.arange(weights.shape[0])
yrange = np.arange(weights.shape[1])
plt.xticks(xrange-.5, xrange)
plt.yticks(yrange*2/np.sqrt(3)*3/4, yrange)

divider = make_axes_locatable(plt.gca())
ax_cb = divider.new_horizontal(size="5%", pad=0.05)
cb1 = colorbar.ColorbarBase(ax_cb, cmap=cm.Blues,
                            orientation='vertical', alpha=.4)
cb1.ax.get_yaxis().labelpad = 16
cb1.ax.set_ylabel('distance from neurons in the neighbourhood',
                  rotation=270, fontsize=16)
plt.gcf().add_axes(ax_cb)

legend_elements = [Line2D([0], [0], marker='o', color='C0', label='Class 1',
                   markerfacecolor='w', markersize=14, linestyle='None', markeredgewidth=2),
                   Line2D([0], [0], marker='+', color='C1', label='Class 2',
                   markerfacecolor='w', markersize=14, linestyle='None', markeredgewidth=2),
                   Line2D([0], [0], marker='x', color='C2', label='Class 3',
                   markerfacecolor='w', markersize=14, linestyle='None', markeredgewidth=2)]
ax.legend(handles=legend_elements, bbox_to_anchor=(0.1, 1.08), loc='upper left',
          borderaxespad=0., ncol=3, fontsize=14)

plt.savefig('resulting_images/som_iris_hex.png')
plt.show()
```

Result: We got this final 2-dimensional feature map with hexagonal grids using self organizing maps of the UCI wine high-dimensional dataset having 13 dimensions. The darker areas represent larger distance whereas lighter areas represent smaller distance between the data points. All the three wine classes are clearly separated even in 2-D.