coding

January 30, 2022

1 Selforganizing Systems Exercise 3 (Coding)

Group f1: Yana Sakhnovych - 11777748 , Alexander Sifel - 01427034, Christian Obereder - 11704936

Github Repository: https://github.com/GitianOberhuber/Selforganizing_Systems_Ex3_TU_Vienna_2022

1.1 Table of contents:

1.2 1 Preamble

The exercise description frequently mentions how the coding exercise is about visualizing SOMs / adding a new visualization to the existing ones in the template. Unless we misunderstood some important apsect of mnemonic SOMs, this does not seem to fully apply for our task. Instead of taking a trained SOM and visualizing it in some special way, our task appeares to rather be adapting the training algorithm of a SOM in such a way that the provided standard visualization display a mnemonic SOM :

Mnemonic SOM: Reading a black/white silhouette image to determine the shape of the SOM, computing a look-up table for shortest distance between two units for the training process, and mapping it into a regular SOM data structure to support the standard visualizations

Therefore, we adapted the standard SOM training algorithm provided by MiniSom by adding the option to train a mnemonic SOM. As a result of this, parts of our code are in the MiniSom python file minisom.py, although we will show excerpts of that code in this report for the sake of presentation. Apart from MiniSom as codebase, we used [1] as a reference for how to implement mnemonic SOM. We also use images that are very similar to the ones used in [1] (Figure-Person, Austria) for comparrison.

We start by explaining and showing our implementation in Section 2.In 2.5 there is a cell that can be used to easily choose from available datasets, sillhouett images and parameters and run our implementation. In Section 3, we generate multiple images to verify the correctness of our implementation and observe changes in training-parameters on the resulting mnemonic SOM.

[1] Mayer, Rudolf, Dieter Merkl, and Andreas Rauber. Mnemonic SOMs: Recognizable shapes for self-organizing maps. 2005.

1.3 2 Implementation

Imports

[1]:
```
# Read data from Java SOMToolbox
from SOMToolBox_Parse import SOMToolBox_Parse
from minisom import MiniSom
from somtoolbox import SOMToolbox
import pickle
import collections
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
import os.path
from os import path
```

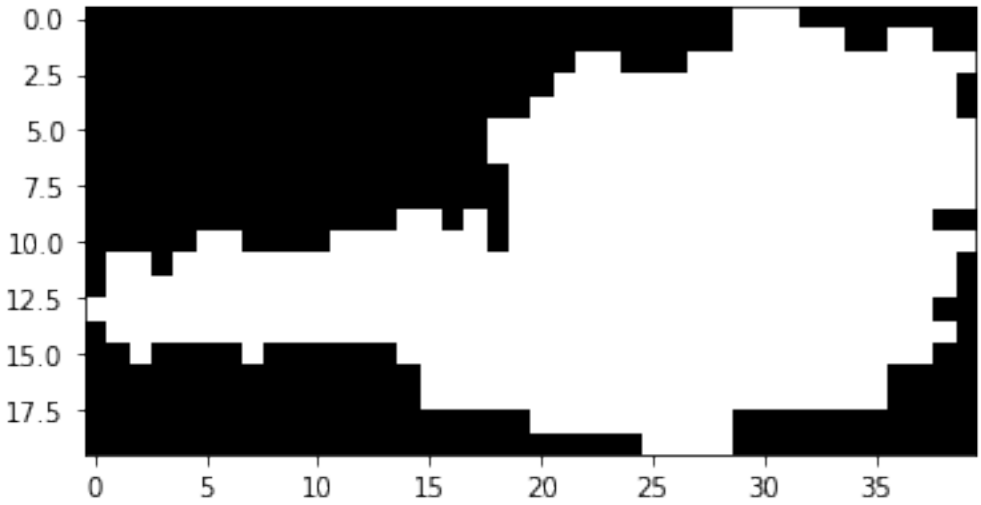
1.3.1 2.1 Reading of black-white sillhouette from an image and generating lookup-tables for shortest distance between units

For determening the shape of the mnemonic SOM, we use black-white images, with white representing positions inside the sillhouette that "have a node" and black representing positions outside that "do not have a node". In our implementation, technically speaking all possible positions contain nodes, that is, they contain a vector of weights. However, as will be shown further below, weights for positions outside of the sillhouette where no node should be will always be full of zeros and these nodes are never selected as winners nor their weights updated during training. The image is converted to a numpy array with ones representing white and zero representing black. This representation was chose in order to be able to easily multiply this shape-matrix with the randomly initialized weight matrix of the SOM during the SOM training, resulting in a weight matrix where the weights for all positions that should not have a node is a vector of zeros.

As an example, here is an image with the sillhouette of austria:

[2]:
```
#takes a black-white image and returns a np-array with 1 for white pixels and 0␣
 ↪for black pixels
def read_bw_silhouette(filename):
    img = Image.open(filename).convert('L')
    return np.array(img) / 255

sample_img = read_bw_silhouette('austria.png')
plt.imshow(sample_img,cmap='gray', vmin = 0, vmax = 1)
```

[2]: <matplotlib.image.AxesImage at 0x7fe32695dd30>



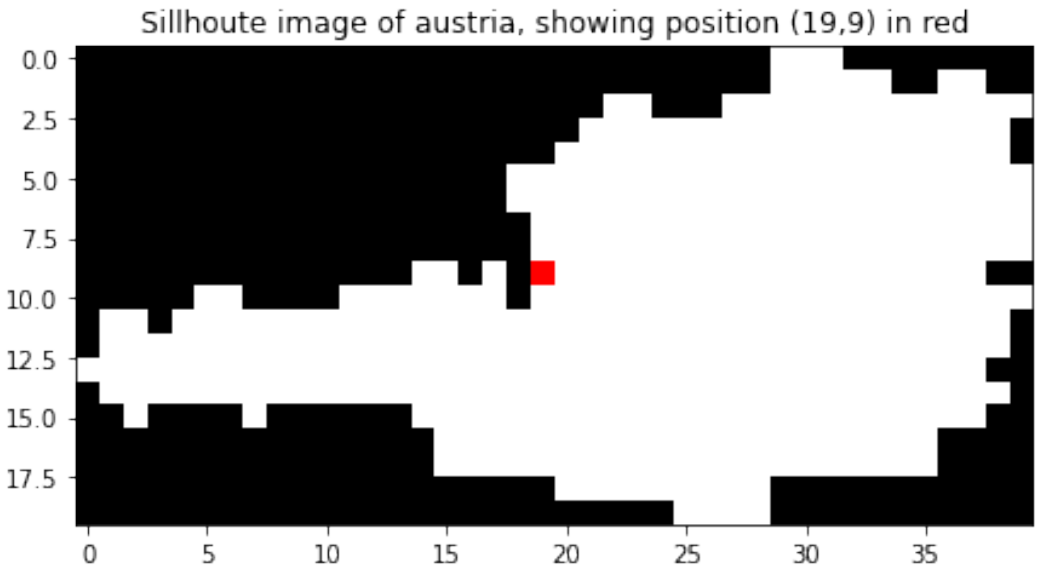
For mnemonic SOMs, it is difficult to use any of the standard neighborhood and distance functions, as neighborhood is no longer dependend on only the coordinates of two positions, but rather on whether or not there is a path between those positions and how long that path is[1]. Therefore, for every possible position (j,k) in the provided image-matrix, we create an adjacency-matrix where the value of some position in that matrix(l,k) is an adapted manhattan-distance between (j,k) and (l,k). This adapted manhattan-distance only considers positions with a value of one to be traversable positions, meaning that zeros, which represent out-of-sillhouette positions, cannot be traversed. If the value of (j,k) itself is zero, a matrix of zeros is returned. This effectively means that for every (j,k), we calculate a matrix with the shortest path to all other positions in the matrix.

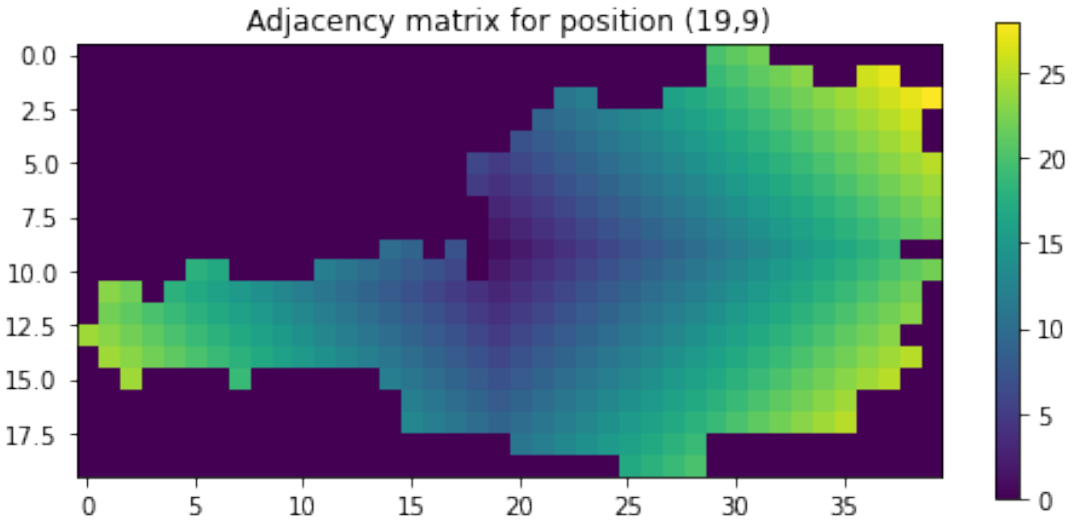
[54]:
```
# Given a matrix, computes manhatten distance to each other point in the␣
 ↪matrix, avoiding "walls"
# grid: A 2d matrix
# start: the position in the matrix from where to calculate manhatten distance␣
 ↪to all other positions
# wall: a value that that can not be traversed and where the distance will␣
 ↪always be 0.
def get_paths_for_point(grid, start, wall):
    max_x, max_y = grid.shape[0],grid.shape[1]
    result = np.zeros((grid.shape[0],grid.shape[1]))
    #if start-point is a 0, distances to all other nodes will be 0
    if grid[start] == 0:
        return result
    queue = collections.deque([[start]])
    seen = set([start])
    i = 1
    result[start] = i
```

```
    while queue:
        path = queue.popleft()
        x, y = path[-1]
        i = result[x, y]
        for x2, y2 in ((x+1,y), (x-1,y), (x,y+1), (x,y-1)):
            if 0 <= x2 < max_x and 0 <= y2 < max_y and grid[x2][y2] != \
            wall and (x2, y2) not in seen:
                queue.append(path + [(x2, y2)])
                seen.add((x2, y2))
                result[x2,y2] = i + 1
    return result

#convert image array to a dictionary, mapping each point (j,k) to its adjacency␣
 ↪matrix
def image_array_to_adjacency_mapping(img_np):
    adjacency_dict = {}

    for i in range(img_np.shape[0]):
        for j in range(img_np.shape[1]):
            cur_result_matrix = get_paths_for_point(img_np, (i,j), 0.0)
            cur_result_matrix[np.nonzero(cur_result_matrix)] = \
            cur_result_matrix[np.nonzero(cur_result_matrix)]
            adjacency_dict[i, j] = cur_result_matrix
    return adjacency_dict
```

To visualize what is described above, an example is shown for an image with the sillhouette of austria in 40x20 and the adjacency values of the position (19,9) (shown in red) :

[55]:
```
fig, (ax1, ax2) = plt.subplots(2, figsize=(8,8))

red_dot_image = np.array(Image.open('austria.png'))
red_dot_image[9,19] = [255,0,0]
ax1.imshow(red_dot_image)
ax1.set_title("Sillhoute image of austria, showing position (19,9) in red")

adjacency_dict = image_array_to_adjacency_mapping(sample_img)
im = ax2.imshow(adjacency_dict[9,19])
cbar = plt.colorbar(im)
ax2.set_title("Adjacency matrix for position (19,9)")
plt.show()
```





A detail worth mentioning is that in the adjacency-matrix for a position (j,k), the value at (j,k) equals 1, because 0 is reserved for the out-of-sillhouette positions. Of course for a neighborhood function $T_{p1,p2}(t) = exp(-\frac{S^2_{p1,p2}}{2 \cdot \sigma(t)^2})$, where S is the adapted manhattan-distance between position p1 and position p2, we would actually want $S^2_{(j,k),(j,k)}$ to be equal to 0, so that $T_{p1,p2}(t)$ would be 1 for any iteration t. We show in 2.2 how we achieve this.

1.3.2 2.2 Adapting of the MiniSom implementation to support mnemonic SOMs

As mentioned in Section 1, we had to make changes to the SOM training algorithm provided by the MiniSom library. We will show excerpts of the code there, particularly of the parts that we had to adapt, as pasting the whole code here would be confusing. The code in its entirety can be viewed in minisom.py .We did our best to mark changes made by us as such.

- Added the optional parameters `lookuptable` and `mnemonic_shape` to the constructor of the MiniSom class. These are supposed to be filled with the map of adjacency-matrices and the numpy array of the read image respectively.
- For the available options for the `neighborhood_function`, the option `'lookuptable'` was added, which uses the map provided with the lookuptable parameter to compute the neighborhood for a position. For the available options for the `activation_distance` function, the option `'manhattan_mnemonic'` was added, which uses the sillhouette provided with the mnemonic_shape parameter to compute the winner in a round of training without taking empty positions into account.
- For the intialization of the MiniSom class, the following changes were made, using the parameters described above. Basically, the new parameters are read, mnemonic shape is read once normally and once with 0 mapped to infinity (which will be used in the activation function). Then, the random weights for the training process are initialized. After intializiation, the weight matrix is multiplied the mnemonic_shape matrix (which is expanded to be of the same dimension of the weight matrix), resulting in a weight matrix where the weight-vectors for positions where there should not be a node consists of only zeros. Technically speaking, in our implementation all positions have a node, it is just for all positions where we do not want a node, the weights of that node are set to all zeros (and these nodes are never considered as candidate for determening a winner and never have their weights updated, as will be shown further below).

```
# Added code: Reading constructor-parameters
self._lookuptable = lookuptable
if (mnemonic_shape is not None):
    self._mnemonic_shape = mnemonic_shape.copy()
    self._mnemonic_shape_inf = mnemonic_shape.copy()
    self._mnemonic_shape_inf[where(self._mnemonic_shape_inf == 0)] = inf

self._weights = self._random_generator.rand(x, y, input_len)*2-1
self._weights /= linalg.norm(self._weights, axis=-1, keepdims=True)
# Added code: Setting out-of-silhouette positions to all zeros if a mnemonic shape is
if (mnemonic_shape is not None):
    self._weights *= self._mnemonic_shape.repeat(self._weights.shape[2]).reshape(*self
```

- Below, the functions for determening a winner used by MiniSom are shown. In this block, only the function _manhattan_distance_mnemonic was written by us. It can be seen that MiniSom uses the function stored in `activasion_distance` (for `example _manhatten_distance`) to produce a matrix with each units activation. It then takes the position with the smallest value in the matrix, which is the winner. We adapted the existing `_manhattan_distance` function by multiplying it with `_mnemonic_shape_inf`, which, as mentioned above, is the matrix of the sillhouette but with zeros replace by infinity. The resulting activation function is one where the distance of any position outside of the sillhoutte / without a node is always infinity. This means that such a position will never be determined as the winner.

```
def _activate(self, x):
    """Updates matrix activation_map, in this matrix
       the element i,j is the response of the neuron i,j to x."""
    self._activation_map = self._activation_distance(x, self._weights)
```

```
def activate(self, x):
    """Returns the activation map to x."""
    self._activate(x)
    return self._activation_map

def winner(self, x):
    """Computes the coordinates of the winning neuron for the sample x."""
    self._activate(x)
    return unravel_index(self._activation_map.argmin(),
                         self._activation_map.shape)

def _manhattan_distance(self, x, w):
    return linalg.norm(subtract(x, w), ord=1, axis=-1)

# Added code: Adapted manhattan distance that returns
# infinity for out-of-sillhouette positions
def _manhattan_distance_mnemonic(self, x, w):
    return linalg.norm(subtract(x, w), ord=1, axis=-1) * self._mnemonic_shape_inf
```

- Lastly, we show the update function, which MiniSom uses to update weights per iteration to explain our changes to the `neighborhood_function`. Here, the `neighborhood_function` is calculated and then weights are updated using the numpy's einsum method. Normally, the `neighborhood_function` (such as `_gaussian` which can be seen below) calculates the neighborhood-matrix based on coordinates. However, using our `_lookuptable_distance` method, we define the neighborhood-matrix as described in 2.1, meaning we use a adjacency_matrix/lookup-table where the values for any empty / out-of-sillhouette positions are 0 and and the sillhouette is generally taken into account. In 2.1 we also mentioned that for the distance for the winner to itself the value would be 0. This must be zero for the neighborhood function to work properly, but out-of-sillhouette positions also have 0 for values. We solve this problem by doing the following for all non-zero positions: subtract 1 and calculate the actual value for the neighborhood matrix using the formula $T_{p1,p2}(t) = exp(-\frac{S^2_{p1,p2}}{2 \cdot \sigma(t)^2})$

```
def update(self, x, win, t, max_iteration):
    """Updates the weights of the neurons.

    Parameters
    ----------
    x : np.array
        Current pattern to learn.
    win : tuple
        Position of the winning neuron for x (array or tuple).
    t : int
        Iteration index
    max_iteration : int
        Maximum number of training itarations.
    """

    eta = self._decay_function(self._learning_rate, t, max_iteration)
```

```
        # sigma and learning rate decrease with the same rule
        sig = self._decay_function(self._sigma, t, max_iteration)
        # improves the performances
        g = self.neighborhood(win, sig)*eta
        # w_new = eta * neighborhood_function * (x-w)
        self._weights += einsum('ij, ijk->ijk', g, x-self._weights)

    def _gaussian(self, c, sigma):
        """Returns a Gaussian centered in c."""
        d = 2*sigma*sigma
        ax = exp(-power(self._xx-self._xx.T[c], 2)/d)
        ay = exp(-power(self._yy-self._yy.T[c], 2)/d)
        return (ax * ay).T  # the external product gives a matrix

    #Added code: Function returning a neighborhood matrix a winner c
    #based on adjacency-matrix (lookuptable)
    def _lookuptable_distance(self, c, sigma):
        d = 2 * sigma * sigma
        cur_adjacency_mat = self._lookuptable[(c[0], c[1])].copy()
        cur_adjacency_mat[np.nonzero(cur_adjacency_mat)] = exp(-power(cur_adjacency_mat[np.nonz
        return cur_adjacency_mat
```

1.3.3 2.3 Summary

A quick summary of the implementation details above: We read a sillhouette/black-white image, with 1/white representing the space inside of the sillhouette and 0/black representing the space outside. We then use this sillhouette to: * Adapt the random weight initialization in such a way that weights for out-of-sillhouette positions are always zero. * Adapt the activation function which is used to determine the winner in such a way that an out-of-sillhouette position can never be the winner * Adaopt the neighborhood function in such a way that out of sillhouette nodes are never updated as part of the training and that the sillhouette is taken into account when calculating distances (shortest paths) between positions.

1.3.4 2.4 Testing implementation with an austria-sillhouette and iris dataset

As an initial test of our implementation, we train a mnemonic SOM with the adapted MiniSom code, using the Iris dataset and an image of Austria as the sillhouette. We then roughly compare the result to the Austria-Iris mnemonic SOM shown in [1]. It must be noted however that different visualization tools were used for these SOMs.

[5]:
```
#training and visualization put into a function to make the notebook less␣
↪bloated

def train_and_display_som(data,  max_x, max_y, learning_rate = 1, sigma = 1,␣
↪iterations = 1000,
                          neighborhood_function = 'gaussian',␣
↪activation_distance= 'euclidean', lookuptable = None,
                          mnemonic_shape = None):
```

```
    som = MiniSom(max_x, max_y, data['vec_dim'], sigma=sigma,␣
↪learning_rate=learning_rate,
                  neighborhood_function = neighborhood_function,␣
↪activation_distance=activation_distance,
                  lookuptable = lookuptable, mnemonic_shape = mnemonic_shape)
    som.train(data['arr'], iterations)
    sm = SOMToolbox(weights=som._weights.reshape(-1, data['vec_dim']),
                    n=max_y, m=max_x, dimension=data['vec_dim'],␣
↪input_data=data['arr'])
    return sm
```

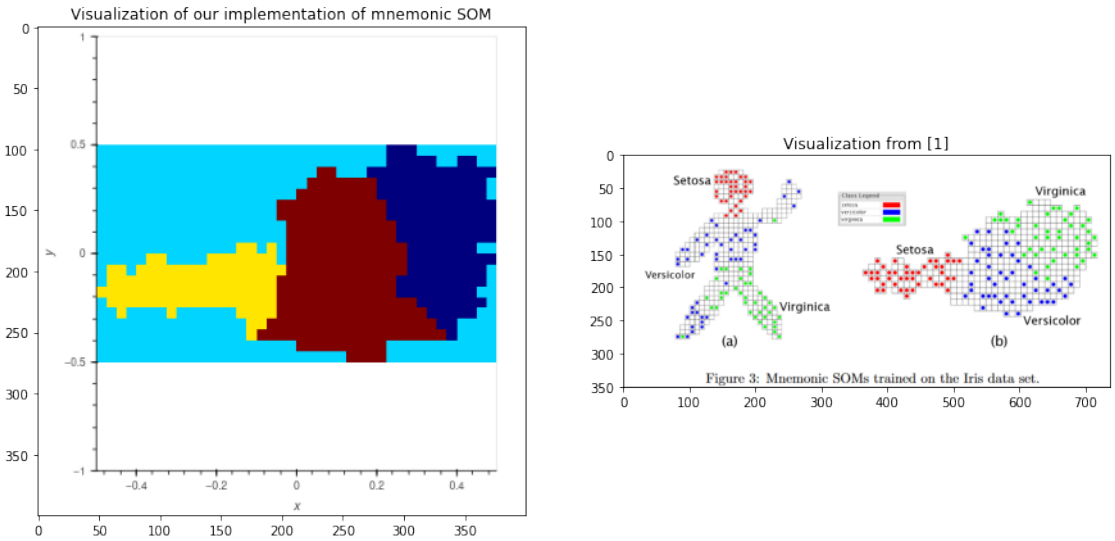
[56]:
```
#train mnemonic SOM on iris data, result can be compared to the 2005 paper
#(https://publik.tuwien.ac.at/files/pub-inf_2979.pdf), Figure 3

#load iris data
idata = SOMToolBox_Parse("datasets/iris/iris.vec").read_weight_file()
weights = SOMToolBox_Parse("datasets/iris/iris.wgt.gz").read_weight_file()
classes = SOMToolBox_Parse("datasets/iris/iris.cls").read_weight_file()

#load sillhouette image and transform to numpy array:
img_arr = read_bw_silhouette('austria.png')
print(img_arr.shape)

#create adjacancy dictionary
adjacency_dict = image_array_to_adjacency_mapping(img_arr)

sm = train_and_display_som(idata,
                           20, 40, sigma=5, learning_rate=1,
                           neighborhood_function = 'lookup', iterations = 5000,
              activation_distance='manhattan_mnemonic' ,
                           lookuptable = adjacency_dict,
                           mnemonic_shape = img_arr)
sm._mainview
```

```
(20, 40)
```

Showing the resulting SOM visualization from the above cell, as to our knowledge the generated visualization get lost when the notebook is closed an reopened. SOMToolbox parameters for generating our visualization: Method = Clustering, Color = jet, Approach = KMeans, Linkage type = single, Clusters = 4. Training parameters: Sigma = 5, learning_rate = 1, iterations = 5000.

[7]:
```
img_self = Image.open("experiment_results/
↪austra_iris_clustering_jet_kmeans_4components.png")
img_paper = Image.open("experiment_results/
↪iris_austria_mnemonic_soms_recoglizable_shapes_for_self-organizing_maps.png")

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15,15))
```

```
ax1.imshow(img_self)
ax1.set_title("Visualization of our implementation of mnemonic SOM")

ax2.imshow(img_paper)
ax2.set_title("Visualization from [1]")
plt.show()
```



1.3.5 2.5 Cell for running the code yourself

The below cell is intended as a convenient way to run our implementation and change datasets, parameters and sillhouette images.

[57]:
```
#load data
#iris
data = SOMToolBox_Parse("datasets/iris/iris.vec").read_weight_file()
weights = SOMToolBox_Parse("datasets/iris/iris.wgt.gz").read_weight_file()
classes = SOMToolBox_Parse("datasets/iris/iris.cls").read_weight_file()
#10clusters data
#data = SOMToolBox_Parse("datasets\\10clusters\\10clusters.vec").
 ↪read_weight_file()
#classes = SOMToolBox_Parse("datasets\\10clusters\\10clusters.cls").
 ↪read_weight_file()
#weights = SOMToolBox_Parse("datasets\\10clusters\\10clusters.wgt").
 ↪read_weight_file()
#chainlink data
#data = SOMToolBox_Parse("datasets\\chainlink\\chainlink.vec").
 ↪read_weight_file()
```

```
#classes = SOMToolBox_Parse("datasets\\chainlink\\chainlink.cls").
 ↪read_weight_file()
#weights = SOMToolBox_Parse("datasets\\chainlink\\chainlink.wgt").
 ↪read_weight_file()

#load sillhouette image
img_arr = read_bw_silhouette('austria.png')
#img_arr = read_bw_silhouette('hand.png')
#img_arr = read_bw_silhouette('figure__large.png') #change x and y below from␣
 ↪20, 40 to 60, 100 if this is chosen

adjacency_dict = image_array_to_adjacency_mapping(img_arr)

sm = train_and_display_som(data, 20, 40, sigma=5, learning_rate=1, iterations =␣
 ↪5000,
                           activation_distance='manhattan_mnemonic' ,␣
 ↪neighborhood_function = 'lookup',
                           lookuptable = adjacency_dict, mnemonic_shape =␣
 ↪img_arr)
sm._mainview
```

1.3.6 3 Evaluation

In this section we try to validate our results and observe the differences in datasets, sillhouette shapes and training parameters. We produce and show multiple images to support this. As far as we are aware, the GUI provided by SOMToolbox will not show in a closed and later re-opened notebook or in a pdf-exported notebook. We therefore saved the visualization results as images and display the images in this notebook.

As for visualization parameters, unless stated otherwise, all of our SOM visualization were created with the SOMToolbox and the following settings: * Clustering * jet * Approach: kmean * Linkage type: single * Clusters: 1 + n, where n is the number of expected clusters for a given dataset (that is, 3 for iris, 2 for chainlink and 10 for 10clusters)

[9]:
```
#load 10clusters data
clusters_data = SOMToolBox_Parse("datasets/10clusters/10clusters.vec").
 ↪read_weight_file()
clusters_classes = SOMToolBox_Parse("datasets/10clusters/10clusters.cls").
 ↪read_weight_file()
clusters_weight = SOMToolBox_Parse("datasets/10clusters/10clusters.wgt").
 ↪read_weight_file()

#load chainlink data
chain_data = SOMToolBox_Parse("datasets/chainlink/chainlink.vec").
 ↪read_weight_file()
chain_classes = SOMToolBox_Parse("datasets/chainlink/chainlink.cls").
 ↪read_weight_file()
```

```
chain_weight = SOMToolBox_Parse("datasets/chainlink/chainlink.wgt").
↪read_weight_file()
```

Here we train a small (40x20) SOM in the shape of austria and a large SOM (100x60) in the shape of a figure-person using the 10clusters dataset and show images of the results. The training parameters can be seen in the below training-cells:

[70]:
```
#load sillhouette image and transform to numpy array:
img_arr = read_bw_silhouette('austria.png')
print(img_arr.shape)

#create adjacancy dictionary
adjacency_dict = image_array_to_adjacency_mapping(img_arr)

sm = train_and_display_som(clusters_data, 20, 40, sigma=5, learning_rate=1,␣
↪neighborhood_function = 'lookup',
              activation_distance='manhattan_mnemonic' , lookuptable =␣
↪adjacency_dict, mnemonic_shape = img_arr)
sm._mainview
```

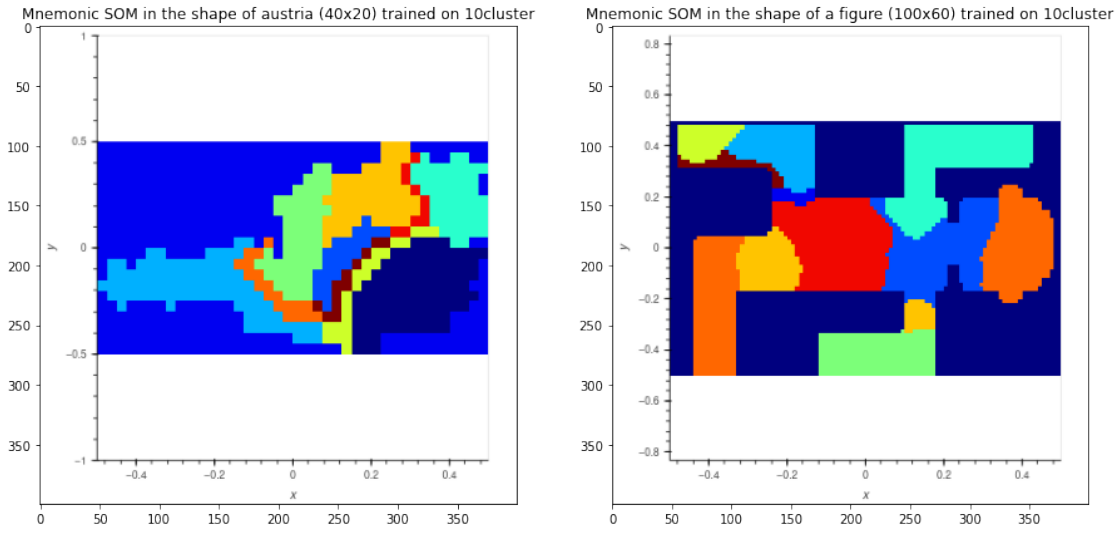
```
(20, 40)
```

[58]:
```
#load sillhouette image and transform to numpy array:
img_arr_fig = read_bw_silhouette('figure_large.png')
print(img_arr_fig.shape)

#create adjacancy dictionary or load if already exists (pickled):
adjacency_dict = {}
if (path.exists("adjacency_figure_large.pickle")):
    with open('adjacency_figure_large.pickle', 'rb') as handle:
        adjacency_dict = pickle.load(handle)
else:
    adjacency_dict = image_array_to_adjacency_mapping(img_arr_fig)
    with open("adjacency_figure_large.pickle", "wb") as output_file:
        pickle.dump(adjacency_dict, output_file)


sm = train_and_display_som(clusters_data, 60, 100, sigma=8,
                           learning_rate=1, neighborhood_function = 'lookup',
                           activation_distance='manhattan_mnemonic' ,
                           mnemonic_shape = img_arr_fig,lookuptable =␣
↪adjacency_dict,
                          iterations = 10000)
sm._mainview
```

```
(60, 100)
```

[12]:
```
som_small = Image.open("experiment_results/
 ↪austria_cluster_clustering_jet_kmeans_11components.png")
som_large = Image.open("experiment_results/
 ↪figure_cluster_clustering_jet_kmeans_11components.png")
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15,15))

print("Training parameters for 40x20 image: sigma=5, learning_rate=1,␣
 ↪iterations = 1000 ")
print("Training parameters for 100x60 image: sigma=8, learning_rate=1,␣
 ↪iterations = 10000")

ax1.imshow(som_small)
ax1.set_title("Mnemonic SOM in the shape of austria (40x20) trained on␣
 ↪10cluster")

ax2.imshow(som_large)
ax2.set_title("Mnemonic SOM in the shape of a figure (100x60) trained on␣
 ↪10cluster")
plt.show()
```

```
Training parameters for 40x20 image: sigma=5, learning_rate=1, iterations = 1000
Training parameters for 100x60 image: sigma=8, learning_rate=1, iterations =
10000
```



[59]:
```
img_arr = read_bw_silhouette('austria.png')
adjacency_dict = image_array_to_adjacency_mapping(img_arr)

sm = train_and_display_som(chain_data, 20, 40, sigma=3.5, learning_rate=1,␣
 ↪neighborhood_function = 'lookup',
```

```
            activation_distance='manhattan_mnemonic' , lookuptable =␣
 ↪adjacency_dict, mnemonic_shape = img_arr)
sm._mainview
```

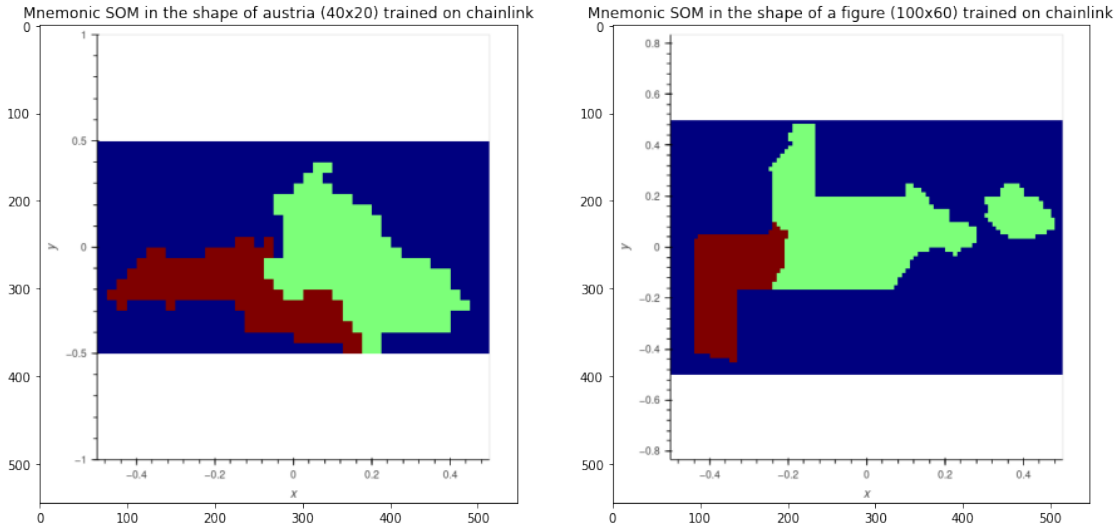
[60]:
```
#load sillhouette image and transform to numpy array:
img_arr_fig = read_bw_silhouette('figure_large.png')
print(img_arr_fig.shape)

#create adjacancy dictionary or load if already exists (pickled):
adjacency_dict = {}
if (path.exists("adjacency_figure_large.pickle")):
    with open('adjacency_figure_large.pickle', 'rb') as handle:
        adjacency_dict = pickle.load(handle)
else:
    adjacency_dict = image_array_to_adjacency_mapping(img_arr_fig)
    with open("adjacency_figure_large.pickle", "wb") as output_file:
        pickle.dump(adjacency_dict, output_file)


sm = train_and_display_som(chain_data, 60, 100, sigma=5, learning_rate=1,␣
 ↪neighborhood_function = 'lookup',
            activation_distance='manhattan_mnemonic' , mnemonic_shape =␣
 ↪img_arr_fig,lookuptable = adjacency_dict,
                        iterations = 10000)
sm._mainview
```

```
(60, 100)
```

[42]:
```
som_small = Image.open("experiment_results/
 ↪austria_chain_clustering_jet_kmeans_11components.png")
som_large = Image.open("experiment_results/
 ↪figure_chain_clustering_jet_kmeans_11components.png")
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15,15))

print("Training parameters for 40x20 image: sigma=3.5, learning_rate=1,␣
 ↪iterations = 1000 ")
print("Training parameters for 100x60 image: sigma=5, learning_rate=1,␣
 ↪iterations = 1000")

ax1.imshow(som_small)
ax1.set_title("Mnemonic SOM in the shape of austria (40x20) trained on␣
 ↪chainlink")

ax2.imshow(som_large)
ax2.set_title("Mnemonic SOM in the shape of a figure (100x60) trained on␣
 ↪chainlink")
```

```
plt.show()
```

```
Training parameters for 40x20 image: sigma=3.5, learning_rate=1, iterations =
1000
Training parameters for 100x60 image: sigma=5, learning_rate=1, iterations =
1000
```



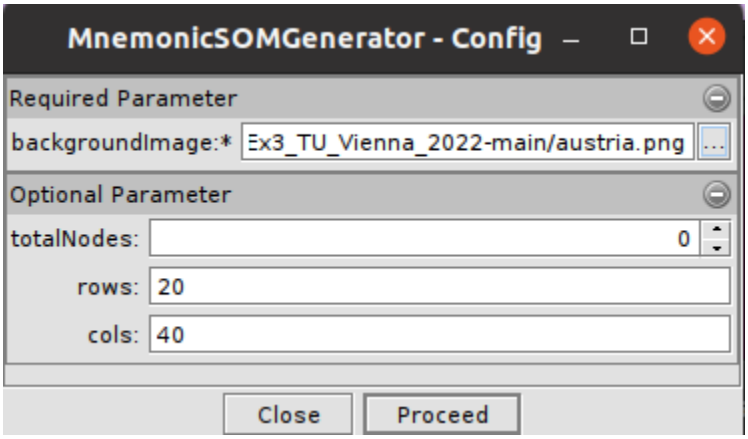
1.3.7 3.1 Comparison with SOM Toolbox

We wanted to compare our results with the ones created by the Mnemonic SOM functionality of the SOM Toolbox as stated in the task description. Unfortunately, we were not able to get it to work. To run the Mnemonic Som training a unit file is required which can be generated using MnemonicSOMGenerator. It contains information about the layout of the grid. When trying to generate the unit file a NullPointerException kept occuring, regardless of release version of the SOMToolbox and the image we uploaded.

You can see our inputs and the Exception below. Instead of the comparison with the SOMToolbox, we provided the comparision with the results of the paper [1] on the Iris dataset which indicates that our implementation is correct

[46]:
```
from IPython.display import Image as Img
Img(filename='GenerateMnemonicSOM.png')
```

[46]:



[47]:
```
Img(filename='GenerateMnemonicSOMException.png')
```

[47]:

```
Starting at.tuwien.ifs.somtoolbox.util.mnemonic.MnemonicSOMGenerator /home/yana/Documents/Uni/Self-Org/Selforganizing_Systems_Ex3_TU_Vienna_2022-main/austria.png -n 0 -r 40 -c 20
Jan 30, 2022 12:26:26 PM at.tuwien.ifs.somtoolbox.util.mnemonic.MnemonicSOMGenerator main
WARNING: Specified "rows", "columns" and "totalNodes". Ignoring "totalNodes".
preferred size: java.awt.Dimension[width=1780,height=900]
set background map: 1760/880, max zoom: 44.0
nodeDiameter: 11.0 spacing x: 77.0 y: 11.0
java.lang.reflect.InvocationTargetException
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.base/java.lang.reflect.Method.invoke(Method.java:566)
        at at.tuwien.ifs.commons.gui.jsap.GenericGUI$5.run(GenericGUI.java:276)
Caused by: java.lang.NullPointerException
        at java.desktop/sun.awt.SunToolkit.getImageFromHash(SunToolkit.java:692)
        at java.desktop/sun.awt.SunToolkit.getImage(SunToolkit.java:728)
        at at.tuwien.ifs.somtoolbox.util.mnemonic.MnemonicSOMGenerator.initFrame(MnemonicSOMGenerator.java:135)
        at at.tuwien.ifs.somtoolbox.util.mnemonic.MnemonicSOMGenerator.<init>(MnemonicSOMGenerator.java:130)
        at at.tuwien.ifs.somtoolbox.util.mnemonic.MnemonicSOMGenerator.main(MnemonicSOMGenerator.java:170)
        ... 5 more
```

1.3.8 3.2 Effects of varying Sigma and Learning Rate

Here, we observe the effect of the training parameters sigma and learning rate. Specifically, we experiment with sigma values 0.5, 1, 5 and 15. For learning rate, we try 0.001, 0.5, 1 and 10

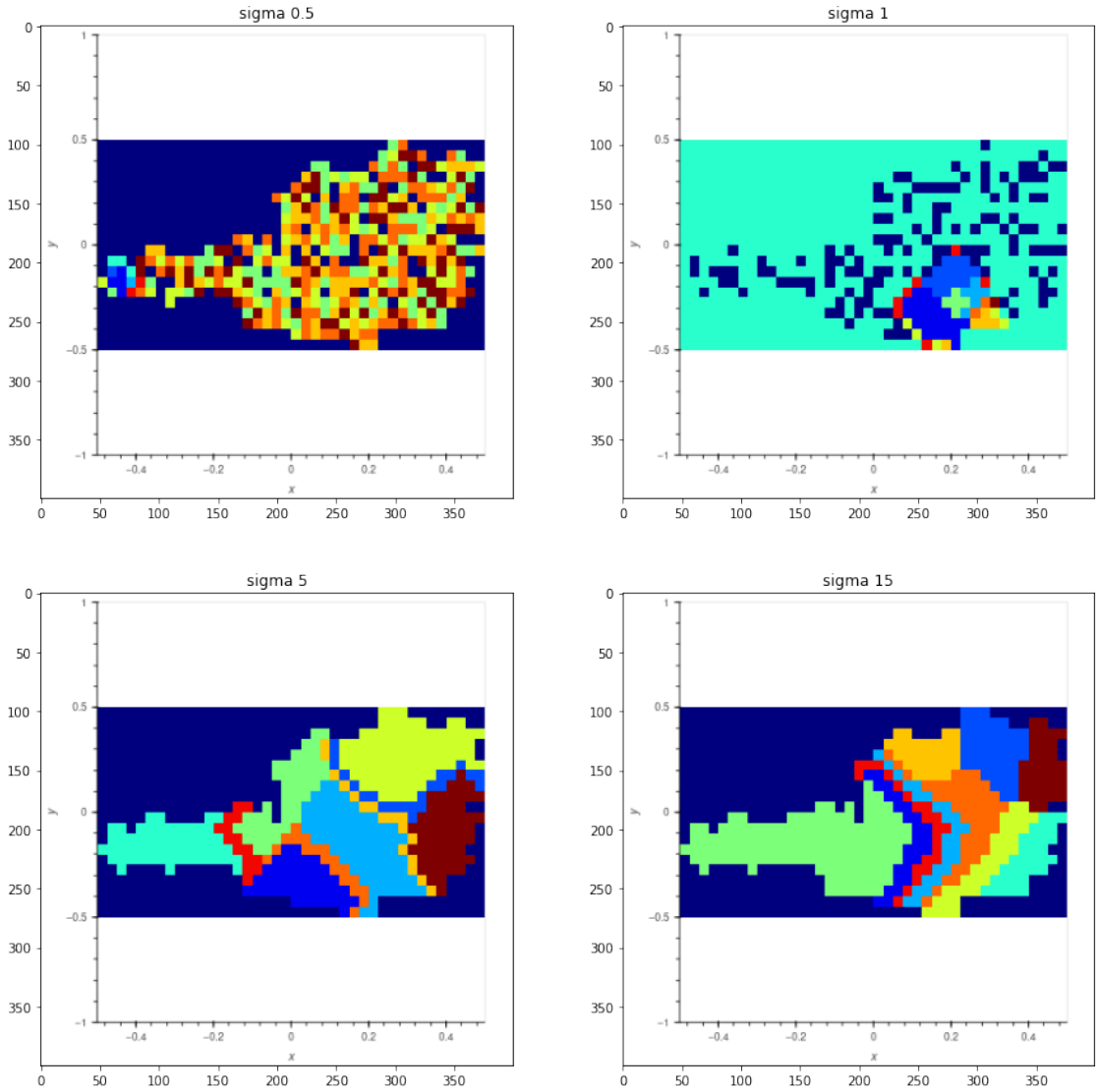
[64]:
```
#load sillhouette image and transform to numpy array:
img_arr = read_bw_silhouette('austria.png')
print(img_arr.shape)

#create adjacancy dictionary or load if already exists (pickled):
adjacency_dict = {}
if (path.exists("adjacency_austria.pickle")):
    with open('adjacency_austria.pickle', 'rb') as handle:
        adjacency_dict = pickle.load(handle)
else:
    adjacency_dict = image_array_to_adjacency_mapping(img_arr)
    with open("adjacency_austria.pickle", "wb") as output_file:
        pickle.dump(adjacency_dict, output_file)

sms = []
```

```
for sigma in [0.5, 1, 5, 15]:
    sm = train_and_display_som(clusters_data, 20, 40, sigma=sigma,␣
 ↪learning_rate=1, neighborhood_function = 'lookup',
                               activation_distance='manhattan_mnemonic' ,␣
 ↪lookuptable = adjacency_dict, mnemonic_shape = img_arr)
    sms.append(sm)
```

```
(20, 40)
```

[65]:
```
sms[3]._mainview
```

[50]:
```
sigma_0p5 = Image.open("experiment_results/austria_clusters_sigma_0p5.png")
sigma_1 = Image.open("experiment_results/austria_clusters_sigma_1.png")
sigma_5 = Image.open("experiment_results/austria_clusters_sigma_5.png")
sigma_15 = Image.open("experiment_results/austria_clusters_sigma_15.png")


fig, ax = plt.subplots(2, 2, figsize=(15,15))

ax[0,0].imshow(sigma_0p5)
ax[0,0].set_title("sigma 0.5")

ax[0,1].imshow(sigma_1)
ax[0,1].set_title("sigma 1")

ax[1,0].imshow(sigma_5)
ax[1,0].set_title("sigma 5")

ax[1,1].imshow(sigma_15)
ax[1,1].set_title("sigma 15")

plt.show()
```



A sigma of 0.5 means very small adaptations of the winner-neighbourhoods. In the first round a direct neighbour with a manhattan distancy of 1 results in a neighbourhood-value of only 0.13. After 1000 iterations that value is nearly 0. This means that the nodes barely influence each other so that no distinct clusters are formed.

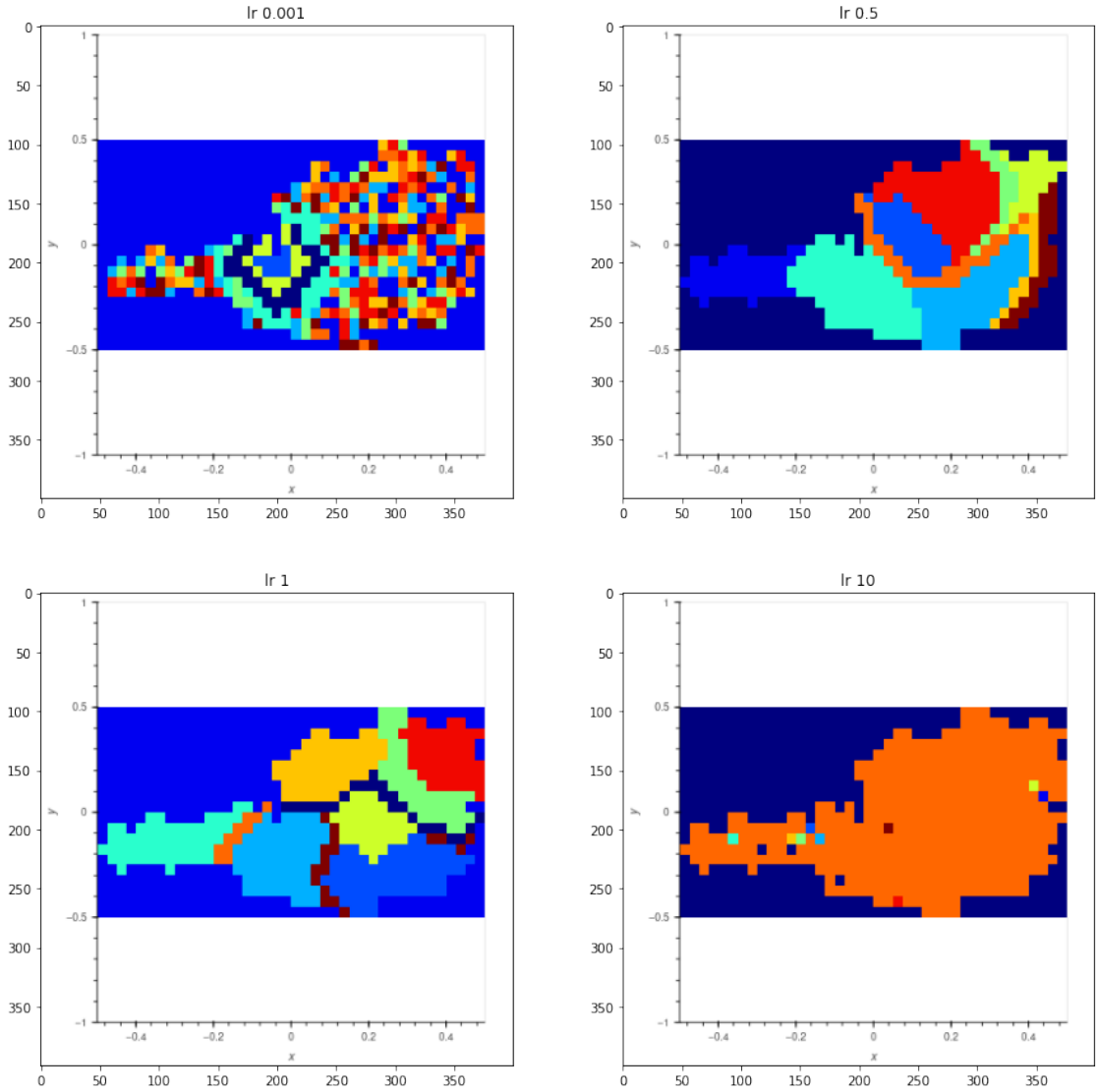
With a sigma of 1 the value of the neighbourhood function for a direct neighbour is 0.6. After 1000 iterations it reaches 0.01. The result is better than the one with the lower learning rate - we have some clusters around Carinthia.

A sigma of 5.0 means high neighbourhood values for close neighbours beginning at 0.98 for the closest neighbours in the first iteration.This results in distinct clusters.

[68]:
```
lr_sms = []
for lr in [0.001, 0.5, 1, 10]:
    sm = train_and_display_som(clusters_data, 20, 40, sigma=5,␣
 ↪learning_rate=lr, neighborhood_function = 'lookup',
                                activation_distance='manhattan_mnemonic' ,␣
 ↪lookuptable = adjacency_dict, mnemonic_shape = img_arr)
    lr_sms.append(sm)
```

[69]:
```
lr_sms[0]._mainview
```

[53]:
```
lr_0p001 = Image.open("experiment_results/austria_clusters_lr_0p001.png")
lr_0p5 = Image.open("experiment_results/austria_clusters_lr_0p5.png")
lr_1 = Image.open("experiment_results/austria_clusters_lr_1.png")
lr_10 = Image.open("experiment_results/austria_clusters_lr_10.png")


fig, ax = plt.subplots(2, 2, figsize=(15,15))

ax[0,0].imshow(lr_0p001)
ax[0,0].set_title("lr 0.001")

ax[0,1].imshow(lr_0p5)
ax[0,1].set_title("lr 0.5")

ax[1,0].imshow(lr_1)
ax[1,0].set_title("lr 1")

ax[1,1].imshow(lr_10)
ax[1,1].set_title("lr 10")

plt.show()
```



- For a learning rate of 0.001 and 1000 iterations, it can be seen that there are barely any clusters. This is presumably because the learning rate is so low that in each training iteration, weights are barely updated, thus the weights of the nodes remain largely as they were initialized: random. However, even with this low of a learning rate, the beginning of a cluster can be seen where Salzburg would be.
- Learning rates of 0.5 and 1.0 in 1000 iterations result in distinct clusters, which is the desired result.
- A learning rate of 10 is very high an results in a very strong adaption of weights. This leads to the formation for a single giant cluster to which most nodes belong.