SOS Assignment 03 - Activity Histograms, MST, Cluster Connections

Chosen group of vizualizations: h (Activity Histograms, MST, Cluster Connections)

Group members:

- Fleischmann Markus (11813846)
- Hadl Jan (01609664)
- Corpaci Luiza (12037284)

Repository URL: https://github.com/aWdas/self-organising-systems-2021

Implementation

This section contains the code for the three visualizations that we implemented.

Imports

In [70]:
```
import networkx as nx
import numpy as np
import panel as pn
import holoviews as hv
from holoviews import opts, dim
from somtoolbox import SOMToolbox
from scipy.spatial import distance_matrix
from SOMToolBox_Parse import SOMToolBox_Parse
from minisom import MiniSom
import pickle
from sklearn.cluster import AgglomerativeClustering
import random

hv.extension('bokeh')
np.warnings.filterwarnings('ignore', category=np.VisibleDeprecationWarning)
```

Minimum Spanning Tree (MST)

In [71]:
```
# Minimum Spanning Tree
def MST(_m, _n, _weights, _idata, root_x, root_y, weighting="input_space", mode="units"):
    if mode == "units":
        root_i = root_x * _m + root_y
        vectors = _weights
    else:
        root_i = root_x
        vectors = _idata

    pairwise_distances = distance_matrix(vectors, vectors)

    graph = nx.convert_matrix.from_numpy_matrix(pairwise_distances)
    mst = nx.minimum_spanning_tree(graph)

    # determine the distance from the root node either by hops or
    # weighted by the edge distances in input space
    if weighting == "input_space":
        dist_from_root = dict(nx.single_source_dijkstra_path_length(mst, root_i))
        max_dist = max(dist_from_root.values())
    elif weighting == "hop_count":
        dist_from_root = dict(nx.single_source_shortest_path_length(mst, root_i))
        max_dist = max(dist_from_root.values())

    if mode == 'input':
        closest_unit = {}
        for i, vector in enumerate(_idata):
            closest_unit[i] = np.argmin(np.sqrt(np.sum(np.power(_weights - vector, 2), axis=1)))

    paths = []
    for edge in mst.edges():
```

```
        # if we have an MST of inputs, plot the paths between the closest units
        unit0 = closest_unit[edge[0]] if mode == 'input' else edge[0]
        unit1 = closest_unit[edge[1]] if mode == 'input' else edge[1]
        x0 = (unit0 // _m + 0.5)
        y0 = (unit0 % _m + 0.5)

        x1 = (unit1 // _m + 0.5)
        y1 = (unit1 % _m + 0.5)

        if weighting != "none":
            # the thicker the line, the closer to the root input/unit
            value = 1 - max([dist_from_root[edge[0]], dist_from_root[edge[1]]]) / max_dist
        else:
            # all lines have the same weight
            value = 1

        paths.append((x0,y0,x1,y1,value))

    return paths
```

In [104…

```
# The Hit Histogram visualization from the template, provided as a backdrop for the MST visualization
def HitHist(_m, _n, _weights, _idata):
    hist = np.zeros(_m * _n)
    for vector in _idata:
        position = np.argmin(np.sqrt(np.sum(np.power(_weights - vector, 2), axis=1)))
        hist[position] += 1
    return np.rot90(hist.reshape(_n, _m))

# The SDH visualization from the template, provided as a backdrop for the MST visualization
def SDH(_m, _n, _weights, _idata, factor, approach):
    import heapq

    sdh_m = np.zeros( _m * _n)

    cs=0
    for i in range(factor): cs += factor-i

    for vector in _idata:
        dist = np.sqrt(np.sum(np.power(_weights - vector, 2), axis=1))
        c = heapq.nsmallest(factor, range(len(dist)), key=dist.__getitem__)
        if (approach==0): # normalized
            for j in range(factor):  sdh_m[c[j]] += (factor-j)/cs
        if (approach==1):# based on distance
            for j in range(factor): sdh_m[c[j]] += 1.0/dist[c[j]]
        if (approach==2):
            dmin, dmax = min(dist[c]), max(dist[c])
            for j in range(factor): sdh_m[c[j]] += 1.0 - (dist[c[j]]-dmin)/(dmax-dmin)

    return np.rot90(sdh_m.reshape(_n, _m))

# Interactive wrapper around the raw MST visualization that adds controls to adjust its parameters
def MST_interactive(m, n, weights, idata, sdh_factor=10):
    def load_mst_conditional(_m, _n, _weights, _idata, target_mode, color='blue'):
        mode_names = {'Input MST': 'input', 'Unit MST': 'units'}
        weighting_names = {'Input Space': 'input_space', 'Hop Count': 'hop_count', 'None': 'none'}

        def __load_mst_conditional__(root_x, root_y, weighting, mode):
            if mode == target_mode or mode == 'Both':
                return hv.Segments(MST(_m, _n, _weights, _idata, root_x,
                                       root_y, weighting_names[weighting], mode_names[target_mode]),
                                   ['x', 'y', 'x1', 'y1'], 'value').opts(
                    line_width=dim('value')*2, color=color)
            else:
                return hv.Segments([], ['x', 'y', 'x1', 'y1'], 'value')

        return __load_mst_conditional__

    sdh = hv.Image(SDH(m, n, weights, idata, sdh_factor, 0), bounds=(0,0,n,m))

    mode = pn.widgets.Select(name='Mode',
                             options=['Input MST', 'Unit MST', 'Both'], value='Input MST')
    weighting = pn.widgets.Select(name='Line Weighting',
                                  options=['Input Space', 'Hop Count', 'None'], value='None')

    # MST of output units
    root_unit_x = pn.widgets.IntInput(name='Unit MST RootX', value=0, step=1, start=0, end=n-1)
    root_unit_y = pn.widgets.IntInput(name='Unit MST RootY', value=0, step=1, start=0, end=m-1)
    root_unit = pn.Row(root_unit_x, root_unit_y, width=200)

    load_mst_unit = load_mst_conditional(m, n, weights, idata, 'Unit MST', color='black')
    dmap_unit = hv.DynamicMap(pn.bind(load_mst_unit, root_x=root_unit_x,
```

```
                                    root_y=root_unit_y, weighting=weighting, mode=mode))

    # MST of input vectors
    root_input = pn.widgets.IntInput(name='Input MST Root', value=0, step=1, start=0, end=idata.shape[0]-1)
    root_input_row = pn.Row(root_input, width=200)

    load_mst_input = load_mst_conditional(m, n, weights, idata, 'Input MST')
    dmap_input = hv.DynamicMap(pn.bind(load_mst_input, root_x=root_input,
                                       root_y=0, weighting=weighting, mode=mode))

    app = pn.Row(pn.WidgetBox('### Options', mode, weighting, root_unit, root_input_row),
                 (sdh.opts(cmap='blues') * dmap_unit * dmap_input).opts(width=250//m*n, height=250))

    return app
```

Activity Histograms

In [73]:

```
## Calculate Activity Histogram
def ActivityHist(_m, _n, _weights, vector):
    # Calculate - for every unit - the distance of the weight to a given vector
    distances = np.sqrt(np.sum(np.power(_weights - vector, 2), axis=1)) # Euclidean Distance/L_2
    distance_matrix = distances.reshape(_n,_m)
    return np.rot90(distance_matrix)

## Interactive wrapper
def ActivityHist_interactive(_m,_n,_weights,_idata):
    def load_ahist(_m,_n,_weights,_idata):
        def __load_ahist__(input):
            return hv.Image(ActivityHist(_m,_n,_weights,_idata[input-1]),bounds=(0,0,_n,_m))
        return __load_ahist__

    input = pn.widgets.IntInput(name='Datapoint', value=1, step=1, start=1, end=len(_idata))
    input_row = pn.Row(input, width=200)

    load_ahist_input = load_ahist(_m, _n, _weights, _idata)
    dmap = hv.DynamicMap(pn.bind(load_ahist_input, input=input))

    app = pn.Row(pn.WidgetBox('### Options', input_row),
                 dmap.relabel('Activity Histogram').opts(
                     cmap='blues',width=300//_m*_n, height=300)).servable()

    return app
```

Cluster Connections

In [74]:

```
# Cluster connections
def ClusterConnection(_m, _n, _weights, threshold1, threshold2, threshold3):
    weight_matrix = _weights.reshape(_n, _m, _weights.shape[1])
    paths = []

    for ix, iy in np.ndindex(weight_matrix.shape[:2]):
        # neighbor above
        if iy + 1 < _m:
            distance_above = np.sqrt(np.sum(np.power(weight_matrix[ix,iy] - weight_matrix[ix,iy+1], 2)))
            if distance_above <= threshold1:
                paths.append((ix+0.5, iy+0.7, ix+0.5, iy+1.3, 1))
            elif distance_above <= threshold2:
                paths.append((ix+0.5, iy+0.7, ix+0.5, iy+1.3, 0.3))
            elif distance_above <= threshold3:
                paths.append((ix+0.5, iy+0.7, ix+0.5, iy+1.3, 0.1))

        # neighbor to the right
        if ix + 1 < _n:
            distance_right = np.sqrt(np.sum(np.power(weight_matrix[ix,iy] - weight_matrix[ix+1,iy], 2)))
            if distance_right <= threshold1:
                paths.append((ix+0.7, iy+0.5, ix+1.3, iy+0.5, 1))
            elif distance_right <= threshold2:
                paths.append((ix+0.7, iy+0.5, ix+1.3, iy+0.5, 0.3))
            elif distance_right <= threshold3:
                paths.append((ix+0.7, iy+0.5, ix+1.3, iy+0.5, 0.1))

    return paths


# A utility function that collects all the distances between neighboring units,
# to determine good thresholds for the ClusterConnection visualization
def NeighborDistances(_m, _n, _weights):
    weight_matrix = _weights.reshape(_n, _m, _weights.shape[1])
```

```
    distances = []

    for ix, iy in np.ndindex(weight_matrix.shape[:2]):
        # neighbor above
        if iy + 1 < _m:
            distances.append((np.sqrt(np.sum(np.power(weight_matrix[ix,iy] - weight_matrix[ix,iy+1], 2))),
                              random.random()))

        # neighbor to the right
        if ix + 1 < _n:
            distances.append((np.sqrt(np.sum(np.power(weight_matrix[ix,iy] - weight_matrix[ix+1,iy], 2))),
                              random.random()))

    return distances
```

In [75]:

```
def ClusterConnection_interactive(_m, _n, _weights, thresholds_max, thresholds_steps=0.01, line_width=2):
    threshold1 = pn.widgets.FloatSlider(name='Threshold 1', start=0, end=thresholds_max,
                                        step=thresholds_steps, value=thresholds_max/4, width=200)
    threshold2 = pn.widgets.FloatSlider(name='Threshold 2', start=0, end=thresholds_max,
                                        step=thresholds_steps, value=thresholds_max/2, width=200)
    threshold3 = pn.widgets.FloatSlider(name='Threshold 3', start=0, end=thresholds_max,
                                        step=thresholds_steps, value=thresholds_max, width=200)

    # ensure that threshold 1 cannot exceed 2, and 2 cannot exceed 3
    @pn.depends(threshold2, watch=True)
    def _update_threshold1_upper(threshold2_value):
        if threshold1.value > threshold2_value:
            threshold1.value = threshold2_value

    @pn.depends(threshold3.param.value, watch=True)
    def _update_threshold2_upper(threshold3_value):
        if threshold2.value > threshold3_value:
            threshold2.value = threshold3_value

    @pn.depends(threshold1.param.value, watch=True)
    def _update_threshold2_lower(threshold1_value):
        if threshold2.value < threshold1_value:
            threshold2.value = threshold1_value

    @pn.depends(threshold2.param.value, watch=True)
    def _update_threshold3_lower(threshold2_value):
        if threshold3.value < threshold2_value:
            threshold3.value = threshold2_value

    def load_cluster_connections(threshold1, threshold2, threshold3):
        return hv.Segments(ClusterConnection(_m, _n, _weights, threshold1, threshold2, threshold3),
                  ['x', 'y', 'x1', 'y1'], 'value').opts(
            alpha=dim('value'), color="black", show_grid=True,
            xticks=list(range(_n+1)), yticks=list(range(_m+1))
        )

    cc = hv.DynamicMap(pn.bind(load_cluster_connections, threshold1=threshold1,
                               threshold2=threshold2, threshold3=threshold3))
    return pn.Row(pn.WidgetBox('### Thresholds', threshold1, threshold2, threshold3),
                  cc.opts(width=300//_m*_n, height=300, line_width=line_width))
```

SOM Training

In this section, we will train a 40x20 (small) SOM on the "Chainlink" dataset, and a 100x60 (large) SOM on the "10-Clusters" dataset retrieved from the url given in the exercise description (http://www.ifs.tuwien.ac.at/dm/somtoolbox/datasets.html).

We used Minisom for the training and used the recommended parameters from the dataset page as a starting point, adjusting them until the results showed the structures that we would expect.

40x20 SOM for the "Chainlink" dataset

In [76]:

```
# Load the input data

idata_chainlink = SOMToolBox_Parse("datasets/chainlink/chainlink.vec").read_weight_file()
```

In [77]:

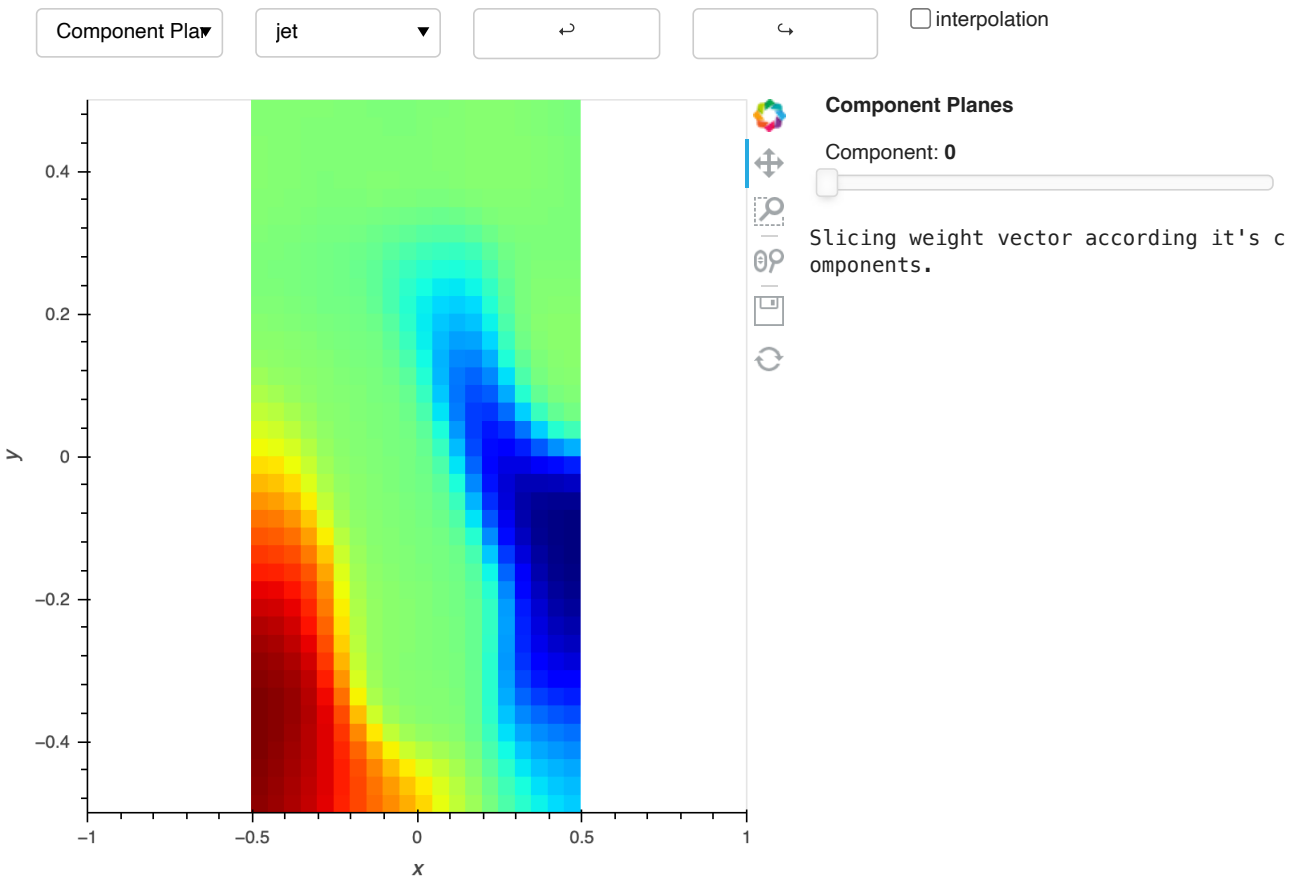
```
# Train the SOM

som_chainlink = MiniSom(40, 20, 3, sigma=6.0, learning_rate=0.4)
som_chainlink.train(idata_chainlink['arr'], 20000)
```

In [78]:
```
# Save the SOM to load it later
# Uncomment this code if you want to save the newly trained SOM

# with open('datasets/chainlink/som_chainlink.p', 'wb') as outfile:
#     pickle.dump(som_chainlink, outfile)
```

In [79]:
```
# Load the stored SOM (skip the previous steps if you want to re-use a stored SOM)

with open('datasets/chainlink/som_chainlink.p', 'rb') as infile:
    som_chainlink = pickle.load(infile)
```

In [80]:
```
# Visualize the SOM with the SOM Toolbox

sm = SOMToolbox(weights=som_chainlink._weights.reshape(-1,3), m=40, n=20, dimension=3,
                input_data=idata_chainlink['arr'])
sm._mainview
```

Out[80]:



100x60 SOM for the "10-Clusters" dataset

In [81]:
```
# Load the input data

idata_10clusters = SOMToolBox_Parse("datasets/10clusters/10clusters.vec").read_weight_file()
```

In [82]:
```
# Train the SOM

som_10clusters = MiniSom(100, 60, 10, sigma=6.0, learning_rate=0.5)
som_10clusters.train(idata_10clusters['arr'], 20000)
```

In [83]:
```
# Save the SOM to load it later
# Uncomment this code if you want to save the newly trained SOM
# rather than using an already trained one

# with open('datasets/10clusters/som_10clusters.p', 'wb') as outfile:
#     pickle.dump(som_10clusters, outfile)
```

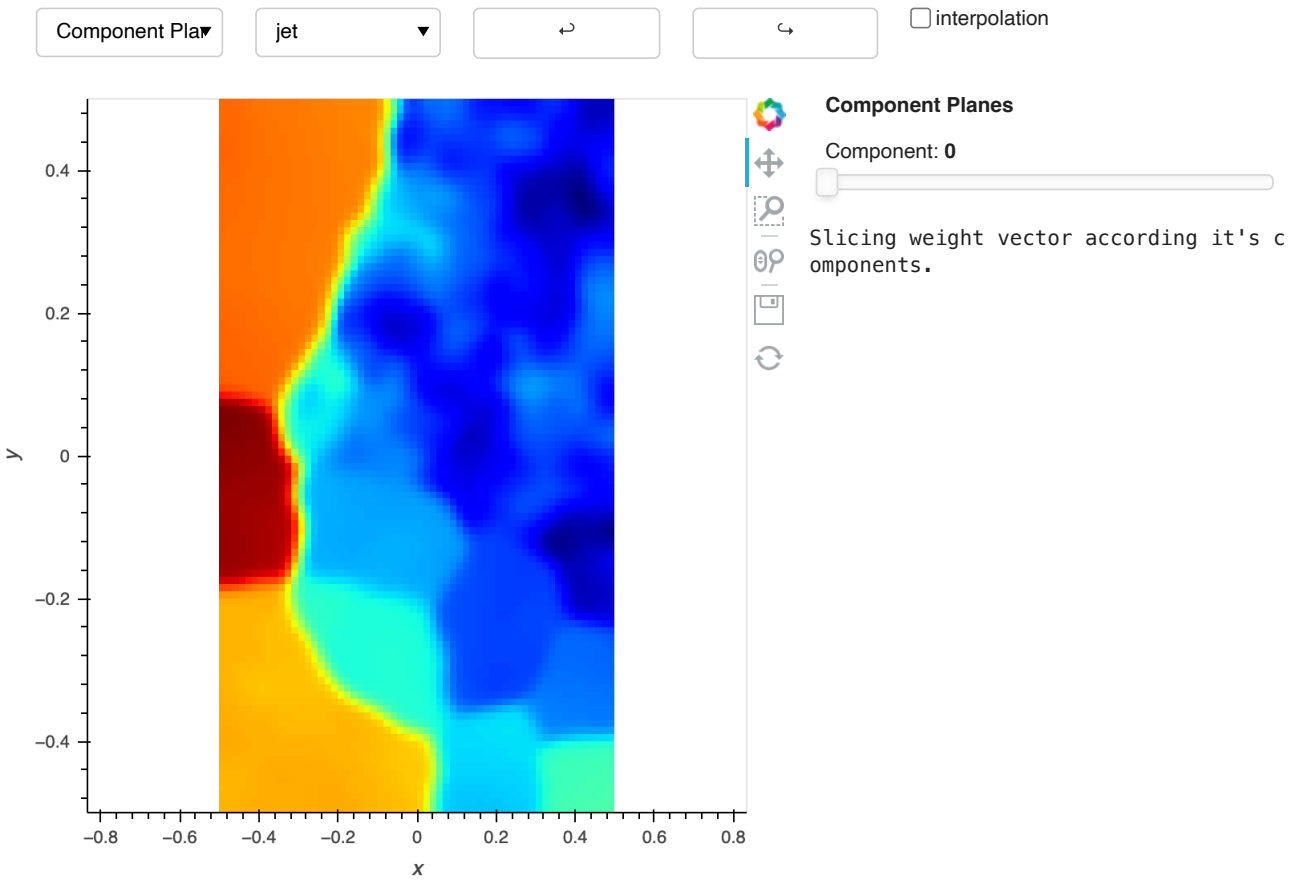
In [84]:
```
# Load the stored SOM (skip the previous steps if you want to re-use a stored SOM)
```

```
with open('datasets/10clusters/som_10clusters.p', 'rb') as infile:
    som_10clusters = pickle.load(infile)
```

In [85]:

```
# Visualize the SOM with the SOM Toolbox

sm = SOMToolbox(weights=som_10clusters._weights.reshape(-1,10), m=100, n=60, dimension=10,
                input_data=idata_10clusters['arr'])
sm._mainview
```

Out[85]:



Evaluation Report

Usage

Minimum Spanning Tree (MST)

For the MST, two functions are provided: `MST` and `MST_interactive`. The former needs to have all options of the MST visualization provided and returns a list of tuples that can be passed to a `hv.Segments` element to render them as edges in the visualization. The latter returns a complete visualization that includes controls to change most of the MST's options.

The visualization adheres to the axis conventions of Holoviews, i.e. the output unit (0,0) is shown in the bottom left corner. The MST can be computed on either the weights of the output units, or on the input vectors, in which case an edge between input vectors i, j will be drawn between their closest output units c_i, c_j on the SOM. Optionally, a root input vector / output unit can be defined, and the line width of the MST edges can be weighted by the length of the path from the root in the input space, or simply by hop count.

In detail, the `MST` function takes the following arguments:

- `_m` : the vertical number of neurons in the SOM grid, i.e., the size of the SOM in the y-dimension (in keeping with the nomenclature of the `SOMToolbox` class provided with the template)
- `_n` : the horizontal number of neurons in the SOM grid, i.e., the size of the SOM in the x-dimension
- `_weights` : a `np.ndarray` that contains the weights of the output units, of shape `[n_units, n_input_space_dims]`
- `_idata` : a `np.ndarray` that contains the weights of the input vectors, of shape `[n_vectors, n_input_space_dims]`
- `root_x` : if `mode=="units"`, then this is the horizontal position of the MST root unit (i.e., from `0` to `_n-1`); if `mode=="input"`, then this is simply the index of the MST root vector in the `_idata` array
- `root_y` : if `mode=="units"`, then this is the vertical position of the MST root unit (i.e., from `0` to `_m-1`); if `mode=="input"`, then this is argument is ignored

- `weighting` : determines how the line weigths of the MST edges are modified. There are three different options:
  - `none` : All lines have the same width
  - `input_space` : A root input vector / output unit r is defined through the arguments `root_x` and `root_y` . Each edge in the MST is weighted by the distance of the two input vectors / output units it connects in the input space. Then for each input vector / output unit u, the length of its path l_u in the MST to r is determined. Let $l_{max} = \max_u l_u$. The line width of an edge (u,v) is then determined by $1 - \max(l_u \,/\, l_{max}, l_v \,/\, l_{max})$. Therefore, the further an edge is away from the root r, the smaller the line width.
  - `hop_count` : This scaling is done in the same way as in the previous option, but here, the weight of each edge is simply set to 1.
- `mode` : set to `"input"` if the MST should be computed on the input vectors, and to `"units"` if the MST should be computed on the output units

The function `MST_interactive` only takes the first four arguments of `MST` , since the remaining arguments can be altered by the user through a GUI that is rendered beside the actual visualization. Additionally, it takes as a fifth argument `sdh_factor` the smoothing factor of the SDH that is used as a backdrop.

Activity Histogram

For the Activity Histogram there are also two functions, `ActivityHist` and `ActivityHist_interactive` . The first takes the size and weights of the SOM as well as an input vector and computes returns an n x m matrix containing the distances. The latter returns an interactive visualisation.

In detail, the `ActivityHist` function takes the following arguments:

- `_m, _n, _weights` : same as for `MST`
- `vector` : the input vector to which the distances are computed

`ActivityHist_interactive` takes the same first three arguments `_m, _n, _weights` as the non-interactive version, and `_idata` as a fourth argument, which is again as in `MST` . It lets the user choose which input vector to render the Activity histogram for.

Cluster Connections

As for the previous visualizations, we implemented two functions. The first is `ClusterConnection` , which takes as input the shape of the data, the weights between the nodes and the 3 thresholds and the second is `ClusterConnection_interactive` , which returns a complete visualization with the option to control the values of the 3 thresholds.

In addition to the 2 functions, we also implemented a utility function, `NeighborDistances` , used to determine good values for the thresholds.

The arguments for the `ClusterConnection` function are:

- `_m, _n, _weights` : same as for the previous functions
- `thresholds` : thresholds used to map the distances to gray-scale values
  - `threshold1` : neighboring nodes with weight distances below `threshold1` are connected with a black line
  - `threshold2` : nodes with distances between `threshold1` and `threshold2` are connected with a dark grey line
  - `threshold3` : nodes between `threshold2` and `threshold3` are connected with a lighter grey line. The nodes with weight vector distances above `threshold3` aren't shown as connected.

The function `ClusterConnection_interactive` only takes the first three arguments of function `ClusterConnection` , as the thresholds can be modified by the user. Besides the common arguments, it also takes the following ones:

- `thresholds_max` : the maximum possible value on the sliders for the threshold values
- `thresholds_steps` : the step value of the sliders
- `line_width` : the line width of the connecting lines; set this to a higher or lower value depending on the size of your SOM (for larger SOMs, you will likely want to lower the line width to still see the individual connections)

The function `NeighborDistances` takes as arguments `_m, _n` , and `_weights` , as described previously, and returns the distances between the output units. It can be used to determine plausible values for the thresholds for a given SOM.

Implementation Details

Minimum Spanning Tree (MST)

To compute the MST, first a matrix of all the pairwise distances between the relevant vectors (output units or input vectors) is computed, using the `distance_matrix` function from the `scipy.spatial` module. This distance matrix is then converted into a `networkx` graph, on which we can then use the `minimum_spanning_tree` function provided by `networkx` to compute the MST.

The paths that the `MST` function returns are 5-tuples `(x0,y0,x1,y1,value)`, the format expected by the `hv.Segments` element. For each edge in the MST, the indices of the edge endpoints denote indices in the `_weights` or `_idata` array, depending on the `mode`. In the former case, `x0,y0` (the position of one end of the line segment) and `x1,y1` (the position of the other end of the line segment) can directly be determined by some trivial index calculation. In the latter case, the MST edge endpoints are input vectors, so first the index in `_weights` of the closest output units for these have to be found. For this, the `closest_unit` dictionary is built.

The `value` item is determined depending on the `weighting` parameter. If `weighting=="none"`, it is simply set to `1` for every edge. If `weighting` is either `"input_space"` or `"hop_count"`, the `single_source_dijkstra_path_length` and `single_source_shortest_path_length` functions of the `networkx` package are used to determine the shortest path from the root to every node in the MST graph, weighted by input space distance or unweighted respectively. This information is then used to scale the `value` for each edge as explained in the "Usage" section above.

In `MST_interactive`, the `hv.DynamicMap` element is used in combination with the `pn.bind` function to implement the interactive visualization. `pn.bind`, as the name suggests, binds arguments of a function to values of GUI controls. We bind a function `__load_mst_conditional__` that is returned from another function `load_mst_conditional`. `load_mst_conditional` is provided with all the `MST` parameters that are non-configurable in the GUI, and the returned `__load_mst_conditional__` then provides all the remaining arguments for binding. Additionally, `__load_mst_conditional__` only returns the MST segments if the mode selected by the user coincides with the specified `target_mode` (with this mechanism we can show/hide the output units & input vectors MST).

The GUI controls are then arranged with a `pn.WidgetBox` and placed side-by-side with the visualization via a `pn.Row`.

Activity Histograms

Computing activity histograms is not very difficult. One simply needs to compute the distance between the input vector and the weight vectors on a per unit basis. As a distance metric we use the euclidean distance (L_2), as this seems to give the clearest results and is also what is used for computing the hit-histogram. This can be done concisely and quickly using numpy (`numpy.sqrt(), numpy.pow, numpy.sum`). The resulting array then only needs to be reshaped (using `numpy.reshape`) into an appropriately sized distance matrix.

For the interactive visualisation we again use a `holoview.DynamicMap` with the `pn.bind` function to allow for selecting one specific input (`input`) vector out of the input data (`idata`). Note that we always select the vector `idata[input-1]` to avoid showing a datapoint "0" in the GUI.

Cluster Connections (CC)

In order to compute the CC, we iterate over the output units in row-major order, and compute for each its distance to the output unit above (i.e., `y+1`) and to the right (i.e., `x+1`). We do not have to calculate the distances to the left and bottom neighbors, since these distances are computed during the loop iteration of the respective neighbor. Depending on the thresholds that these distances clear, we add paths with the right value for the gray-scale mapping to the output.

The result is a list of paths in the format of 5-tuples `(x0,y0,x1,y1,value)`. The coordinates (`x0,y0,x1,y1`) have the same meaning as in the case of the MST computation, i.e., they mark the starting and ending point of a line segment in the visualization.

The `value` parameter represents the gray-scale mapping (0.1 - for light-grey, 0.3 - for dark-grey, 1 - for black). The three gray-scale levels are achieved by adjusting the `alpha` level of the `"black"` base color.

In the case of the `ClusterConnection_interactive` function, we continue using the `pn.bind` function for the implementation of the interactive visualization. We bind the `load_cluster_connections` function (which sets all the non-configurable parameters for the call of `ClusterConnection`) with the values of the thresholds `threshold1`, `threshold2`,

`threshold3` chosen by the user. We also employ `@pn.depends` change listeners to ensure that the values of the sliders stay consistent, i.e., that `threshold1` cannot have a higher value than `threshold2`, and so on.

Visualization Showcase

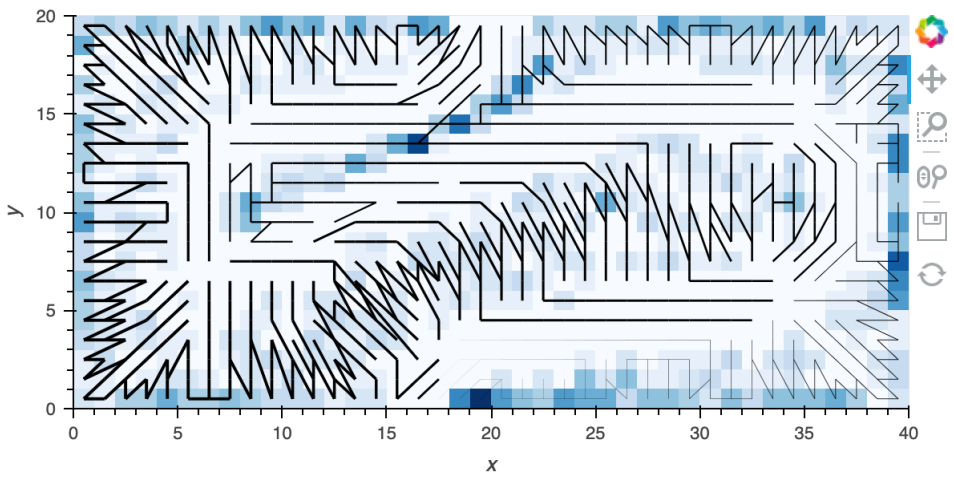
Minimum Spanning Tree (MST)

"Chainlink" SOM, MST on the output units, line width by input space distance with unit (0,0) as root, HitHistogram backdrop

We can clearly discern the two rings from the line widths of the MST, since they drop markedly from the units in the left chainlink (of which the root (0,0) is a part) to the units of the right chainlink. A Hit Histogram is shown as a backdrop, and it is obvious that the structure of the MST and of the Hit Histogram coincide.

In [86]:
```
hithist = hv.Image(HitHist(20, 40, som_chainlink._weights.reshape(-1,3), idata_chainlink['arr']),
                   bounds=(0,0,40,20)).opts(cmap='blues')
mst = hv.Segments(MST(20, 40, som_chainlink._weights.reshape(-1,3), idata_chainlink['arr'],
                      0, 0, "input_space", "units"), ['x', 'y', 'x1', 'y1'], 'value').opts(
    line_width=dim('value')*2, color="black")
(hithist * mst).opts(width=600, height=300)
```

Out[86]:



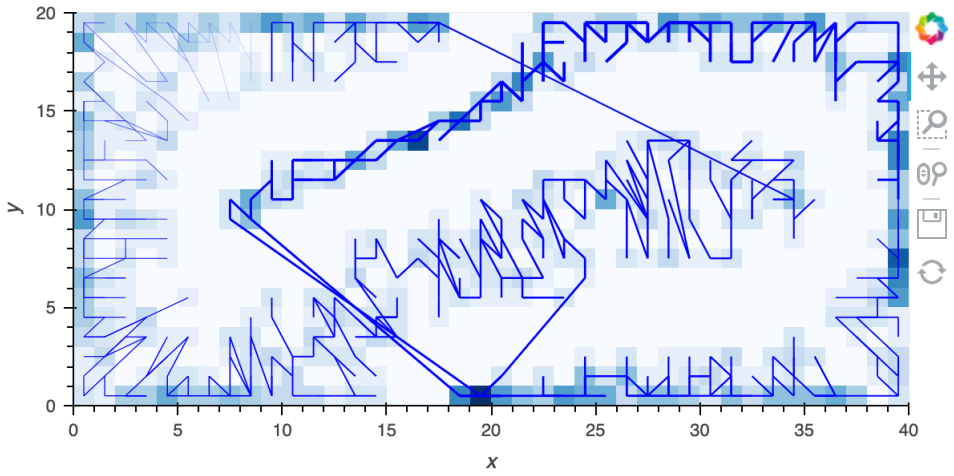
"Chainlink" SOM, MST on the input units, line width by hop count with input vector 400 as root, HitHistogram backdrop

We can see from the line widths that the root input vector 400 was mapped to an output unit somewhere in the upper center-right region of the SOM. The line widths are much slimmer in the left chainlink, since the inputs are further apart from the root input vector. Interestingly though, parts of the left chainlink that were mapped to the center region of the SOM have quite high line widths still, which would indicate that the root input unit comes from the section of the right chainlink where it is closest to the left one. The long lines in the graph indicate the topology violations (the points where the two chainlinks are broken).

Again using a Hit Histogram as a backdrop, we can see that the MST edges only connect units where input vectors where actually mapped to, as is intended.

In [87]:
```
hithist = hv.Image(HitHist(20, 40, som_chainlink._weights.reshape(-1,3), idata_chainlink['arr']),
                   bounds=(0,0,40,20)).opts(cmap='blues')
mst = hv.Segments(MST(20, 40, som_chainlink._weights.reshape(-1,3), idata_chainlink['arr'], 400, 0,
                      "hop_count", "input"), ['x', 'y', 'x1', 'y1'], 'value').opts(
    line_width=dim('value')*2, color="blue")
(hithist * mst).opts(width=600, height=300)
```

Out[87]:



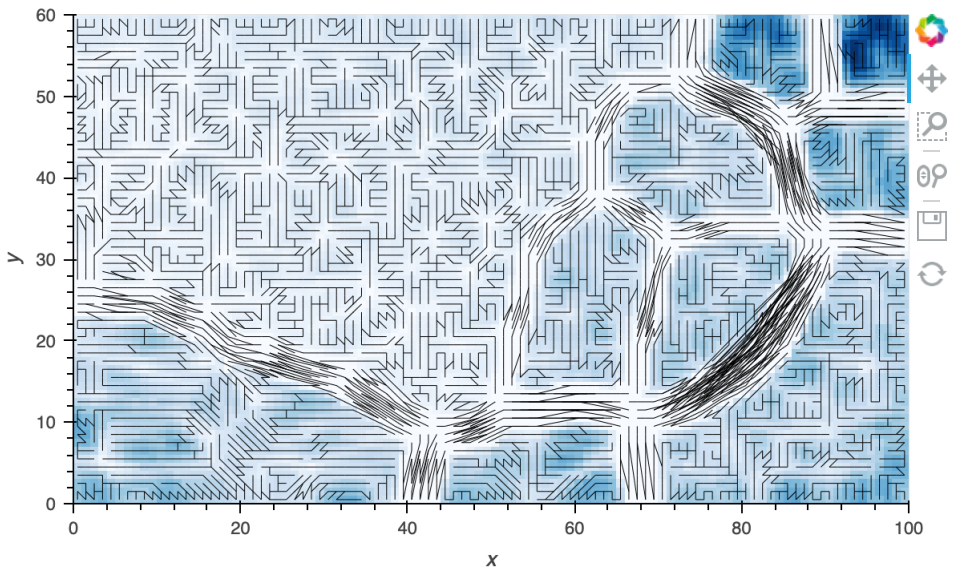
"10 Clusters" SOM, MST on the output units, unweighted line width, SDH backdrop (factor 25)

This visualization takes a bit of time to compute, since determining the MST for this many units is quite slow (at least when run inside a Jupyter notebook). To make the visualization clearer, a SDH has been used as a backdrop here. One can clearly see, from the SDH and from the MST edges, the boundaries between the 10 clusters.

In [88]:

```
sdh = hv.Image(SDH(60, 100, som_10clusters._weights.reshape(-1,10), idata_10clusters['arr'], 25, 0),
                   bounds=(0,0,100,60)).opts(cmap='blues')
mst = hv.Segments(MST(60, 100, som_10clusters._weights.reshape(-1,10), idata_10clusters['arr'],
                      0, 0, "none", "units"), ['x', 'y', 'x1', 'y1'], 'value').opts(
    line_width=dim('value')*0.5, color="black")
(sdh * mst).opts(width=600, height=360)
```

Out[88]:



"Chainlink" SOM, interactive

In [105…

```
MST_interactive(20, 40, som_chainlink._weights.reshape(-1,3), idata_chainlink['arr'], 10)
```

Out[105]:

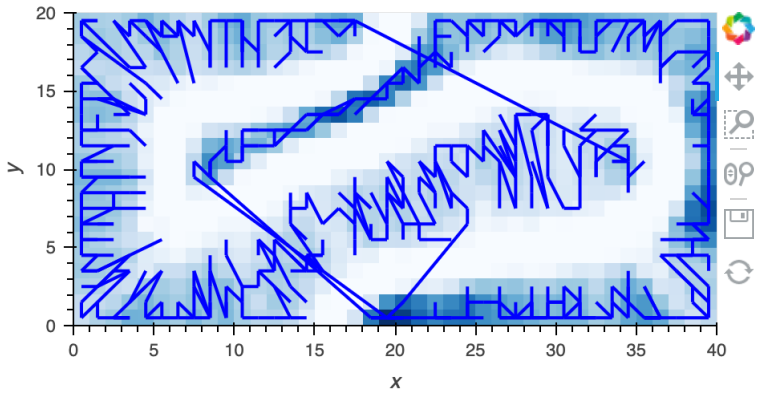
Options

Mode: Input MST

Line Weighting: None

Unit MST RootX: 0 Unit MST RootY: 0

Input MST Root: 0



Activity Histogram

"Chainlink" SOM, interactive

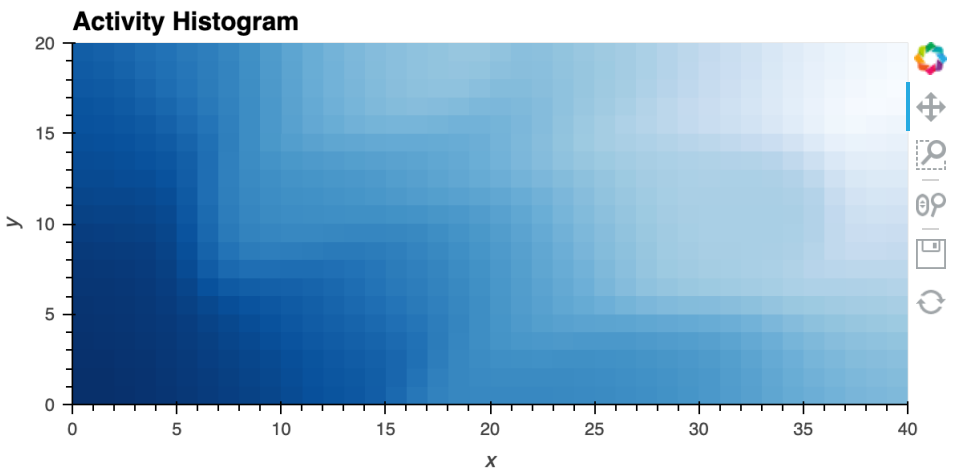
While going through the different input vectors, we can see how different regions get activated. Usually these regions are continuous, though sometimes topology violations do appear (e.g., in datapoint 16, see below). Comparing the activity to, e.g., the D-Matrix we can also see that the activity usually does not cross the 'peaks' in the D-Matrix.

In [90]:
```
ActivityHist_interactive(20, 40, som_chainlink._weights.reshape(-1,3), idata_chainlink['arr'])
```

Out[90]:

Options

Datapoint: 1

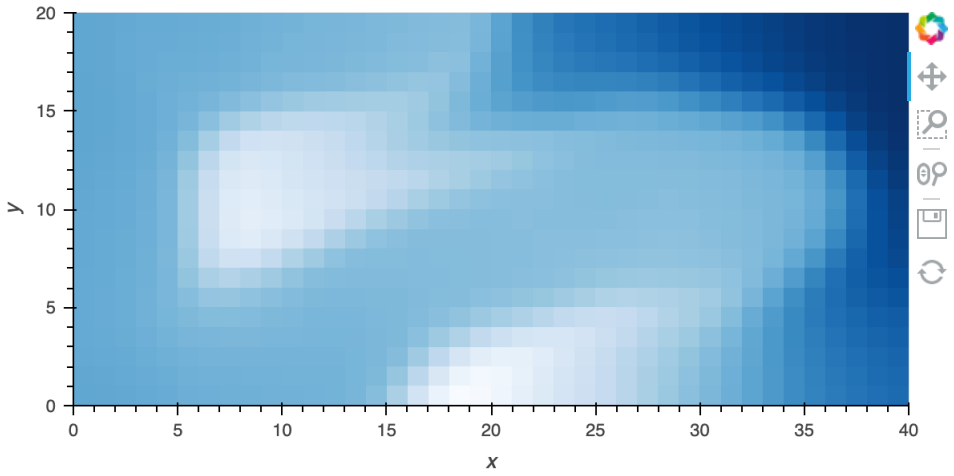


"Chailink" SOM, topology violation

Here we have a nice example of a topology violation, where two distant regions are both activated by the same input. In fact these regions are active at the same time for many vectors in the dataset, indicating the break in the ring structure that is inherent to all visualizations of the "Chainlink" dataset.

In [91]:
```
hv.Image(ActivityHist(20, 40, som_chainlink._weights.reshape(-1,3), idata_chainlink['arr'][15]),
         bounds=(0,0,40,20)).opts(cmap='blues', width=600, height=300)
```

Out[91]:

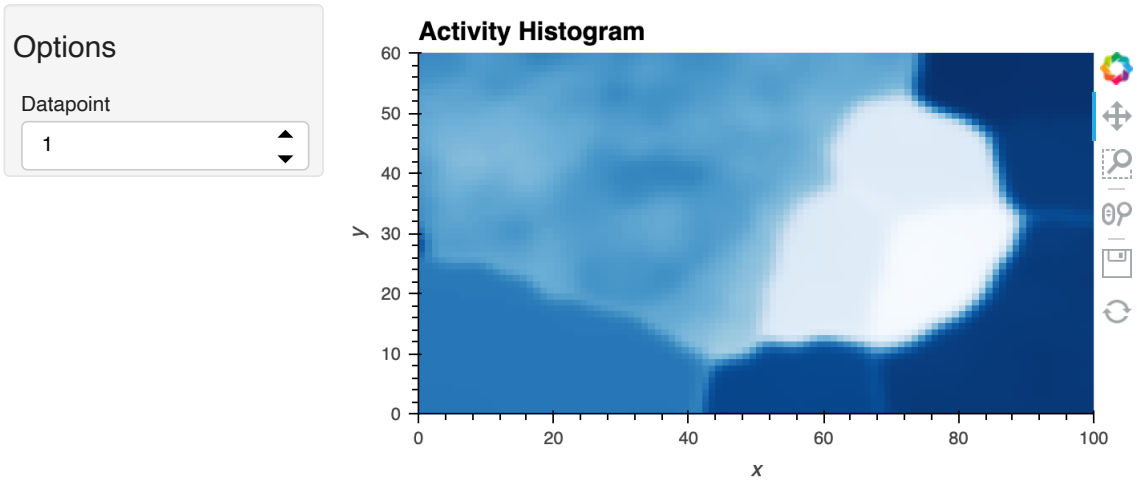


"10 Clusters" SOM, Interactive

Looking at the activity histograms for the "10 Clusters" Dataset, the cluster the current datapoint belongs to and some of the other clusters become very well visible. Due to the colour scaling it is almost impossible to discern any differences within a cluster.

In [92]:
```
ActivityHist_interactive(60, 100, som_10clusters._weights.reshape(-1,10), idata_10clusters['arr'])
```

Out[92]:

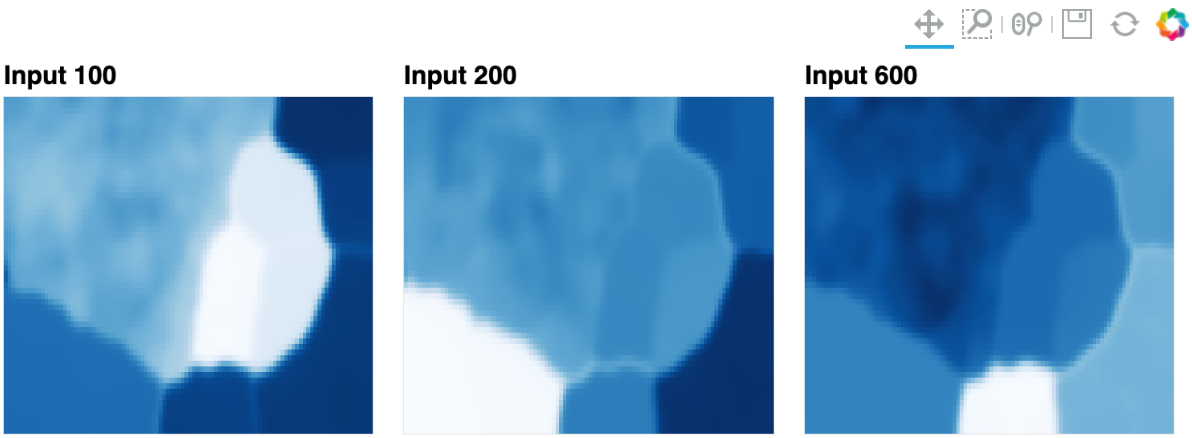


"10 Clusters" SOM, different clusters

It is clear to see which cluster an input belongs to.

In [93]:
```
c1 = hv.Image(ActivityHist(60, 100, som_10clusters._weights.reshape(-1,10), idata_10clusters['arr'][100]),
    bounds=(0,0,100,60)).opts(cmap='blues', xaxis=None,yaxis=None, width=250, height=250)
c2 = hv.Image(ActivityHist(60, 100, som_10clusters._weights.reshape(-1,10), idata_10clusters['arr'][200]),
    bounds=(0,0,100,60)).opts(cmap='blues', xaxis=None,yaxis=None, width=250, height=250)
c3 = hv.Image(ActivityHist(60, 100, som_10clusters._weights.reshape(-1,10), idata_10clusters['arr'][600]),
    bounds=(0,0,100,60)).opts(cmap='blues', xaxis=None,yaxis=None, width=250, height=250)
hv.Layout([c1.relabel('Input 100'),c2.relabel('Input 200'),c3.relabel('Input 600')])
```

Out[93]:



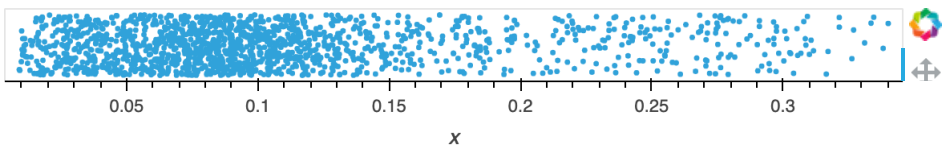
Cluster Connections

"Chainlink" SOM

By using a little utility function we can see that the majority of neighboring units have a distance between 0 and 0.15 in the input space. The maximum distance is roughly 0.35. Using this information, we can find threshold values that yield good results, but also counter-examples that provide no information whatsoever.

In [94]:
```
hv.Points(NeighborDistances(20, 40, som_chainlink._weights.reshape(-1,3))).opts(height=100, width=600, yaxis=N
```

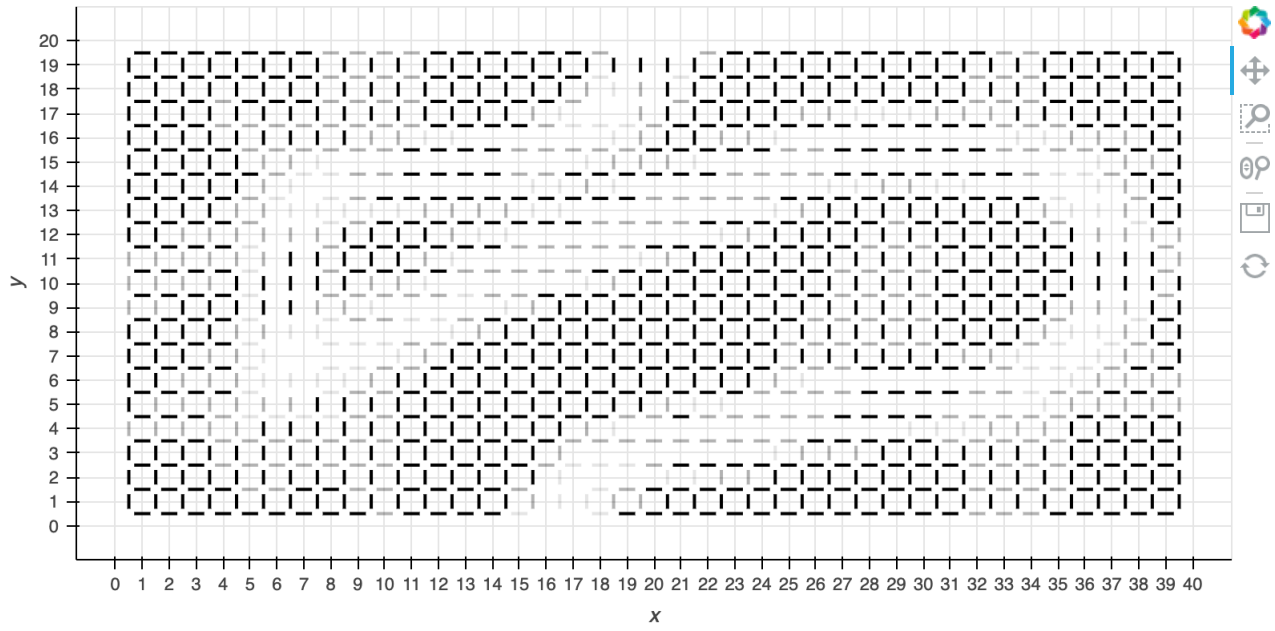
Out[94]:



Setting the thresholds to 0.10, 0.15, and 0.2 yields exactly the structure we would expect from the chainlink dataset, with the two rings being clearly separated from one another, and units belonging to each of the rings in close proximity to each other. The interpolating units are sometimes close together along one axis (i.e., to their vertical or horizontal neighbors), but always further away from the units of the two rings (notice how the connections from interpolating units to neighboring ring units almost never clear even the highest threshold).

In [95]:
```
hv.Segments(ClusterConnection(20, 40, som_chainlink._weights.reshape(-1,3), 0.1, 0.15, 0.2),
                  ['x', 'y', 'x1', 'y1'], 'value').opts(
    line_width=2, alpha=dim('value'), color="black", width=800, height=400,
    show_grid=True, xticks=list(range(41)), yticks=list(range(21)))
```

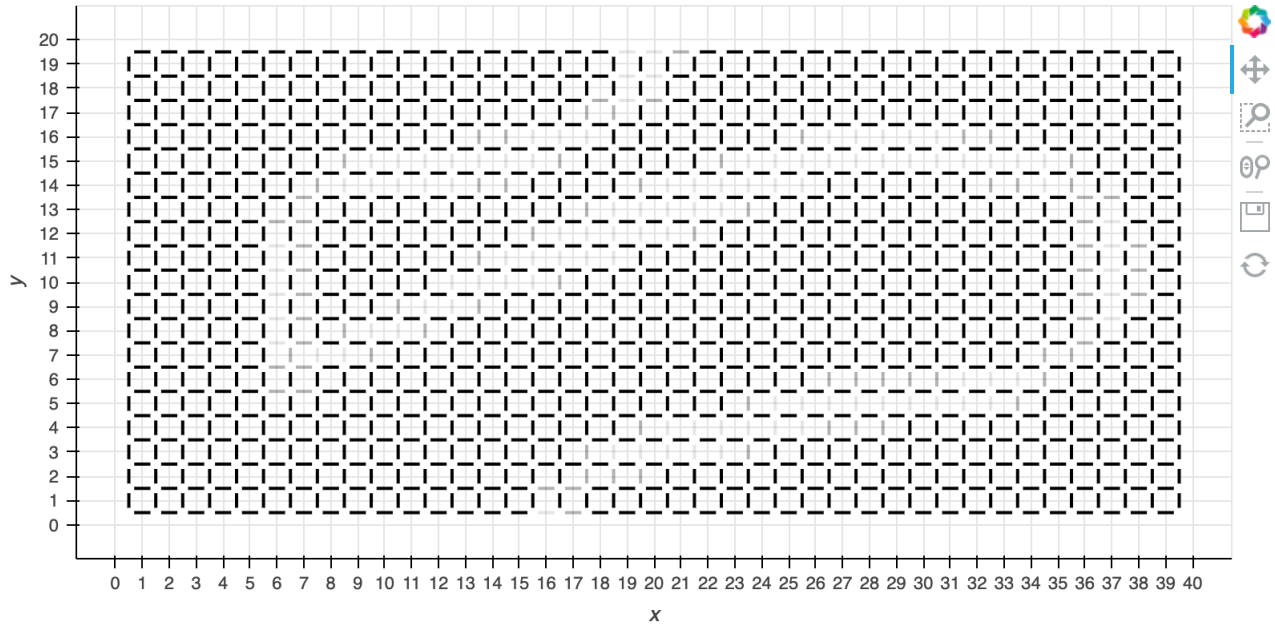
Out[95]:



To produce a counter-example from which no information can be gained, we set the thresholds to 0.2, 0.25, and 1.0. Here, almost all connections clear the lowest threshold, making it very difficult to detect any structure in the visualization. This highlights the importance of analysing the distance distribution for the given dataset and setting the thresholds accordingly.

In [96]:
```
hv.Segments(ClusterConnection(20, 40, som_chainlink._weights.reshape(-1,3), 0.2, 0.25, 1.0),
                  ['x', 'y', 'x1', 'y1'], 'value').opts(
    line_width=2, alpha=dim('value'), color="black", width=800, height=400,
    show_grid=True, xticks=list(range(41)), yticks=list(range(21)))
```

Out[96]:

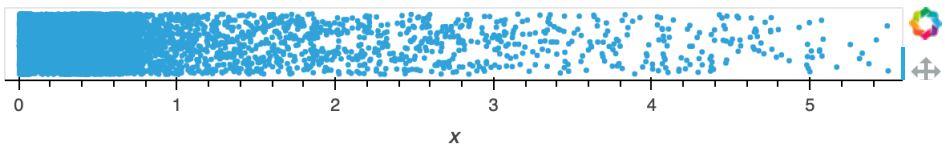


"10 Clusters" SOM

Running our helper visualization again, we see that for the "10 Clusters" dataset, the distances between neighboring units vary in a much broader range, from 0 up to around 6. Most are concentrated in the [0,1] range, with the number of occurences decreasing steadily for higher values.

In [97]:
```
hv.Points(NeighborDistances(60, 100, som_10clusters._weights.reshape(-1,10))).opts(
    height=100, width=600, yaxis=None)
```

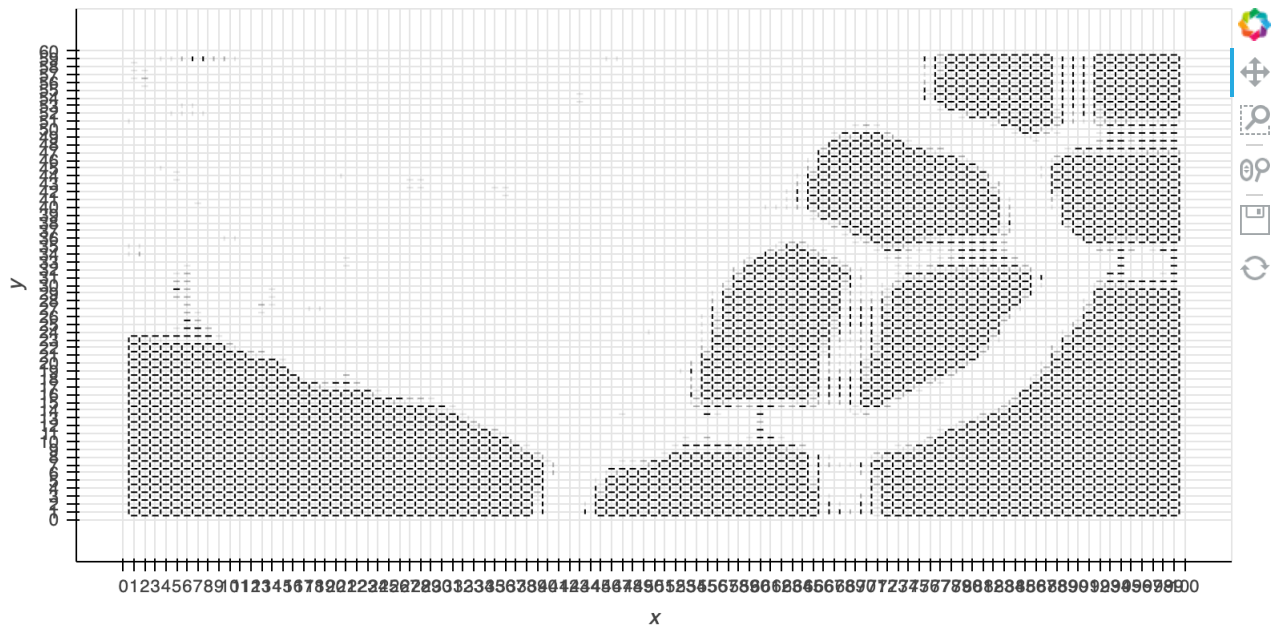
Out[97]:



When we use the same thresholds that were optimal for the "Chainlink" dataset, we again see the importance of proper threshold scaling: 9 clusters are clearly distinguishable, but the largest cluster has a sufficiently low density that almost none of its connections clear even the highest threshold. It is therefore simply not visible in the graphic below.

In [98]:
```
hv.Segments(ClusterConnection(60, 100, som_10clusters._weights.reshape(-1,10), 0.1, 0.15, 0.2),
                  ['x', 'y', 'x1', 'y1'], 'value').opts(
    line_width=1, alpha=dim('value'), color="black", width=800, height=400,
    show_grid=True, xticks=list(range(101)), yticks=list(range(61)))
```

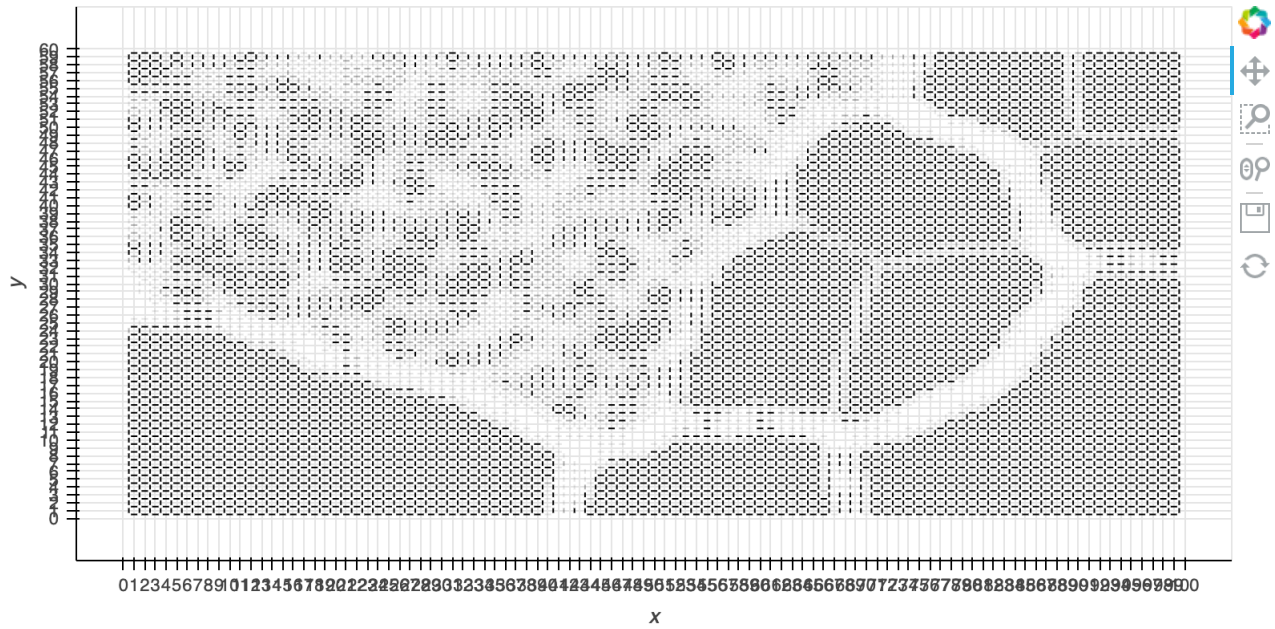
Out[98]:



Tweaking the thresholds according to the info from our helper visualization above, we retrieve much more information from the visualization: the 10th cluster is now visible, but its lower density is apparent from the many higher-distance connections in its region of the SOM. The two groups of three clusters each that are closer together in the input space are also more easily discernible now (center and top-right).

In [99]:
```
hv.Segments(ClusterConnection(60, 100, som_10clusters._weights.reshape(-1,10), 0.4, 0.55, 1.5),
                  ['x', 'y', 'x1', 'y1'], 'value').opts(
    line_width=1, alpha=dim('value'), color="black", width=800, height=400,
    show_grid=True, xticks=list(range(101)), yticks=list(range(61)))
```

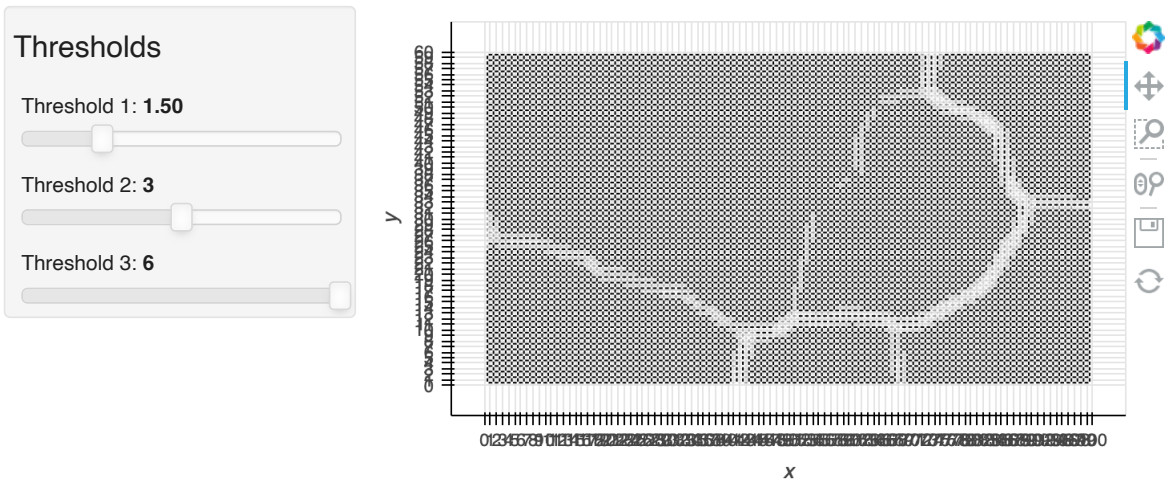
Out[99]:



Lastly, we have an interactive version of the Cluster Connections visualization where the thresholds can be set by the user. The maximum values and steps of the thresholds can be configured to fit the current SOM, and have been set to 6 and 0.05 here. Also, the line width can be configured to suit SOMs of different sizes.

In [100…
```
ClusterConnection_interactive(60, 100, som_10clusters._weights.reshape(-1,10), 6, 0.05, 1)
```

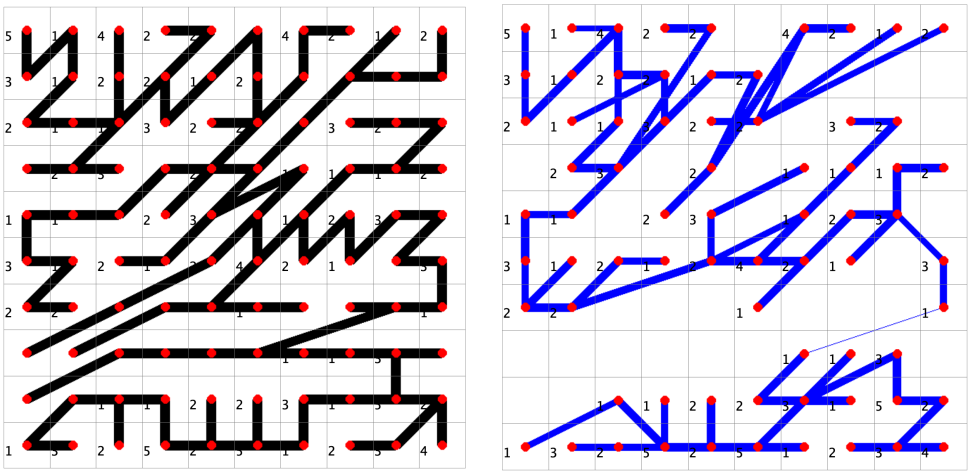
Out[100]:



Comparison with the Java SOM Toolbox

Minimum Spanning Tree (MST)

We used the Iris dataset provided with the SOMToolbox for comparison. We let it render the MST of the output units and of the input vectors, the latter with the "Weight lines" option active. Both are shown below, the output unit SOM with black lines and the input vector SOM with blue lines. The visualizations were rotated so that unit (0,0) is in the lower left corner, like in our implementation.

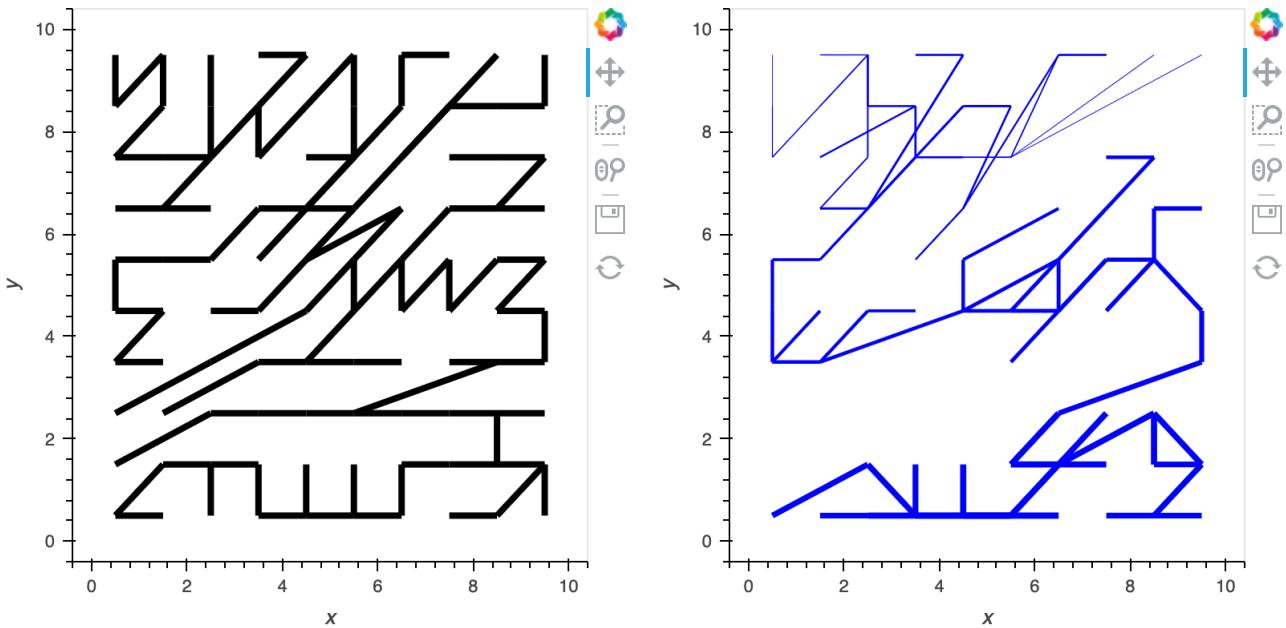


Below, you can see the equivalent visualizations with our implementation. The weighting for the input vector MST was set to "input space distance", with input vector 0 chosen as the root. We can see that the exact same trees are rendered, but the line weighting is performed differently: in our implementation, the line width decreases the further an edge is from the MST root (input vector 0 in our case, which is mapped to a unit in the cluster at the bottom judging from the line widths), as was requested in the exercise description, while the Java SOM Toolbox weights the lines (as far as we could discern from the source code) by their own distance in the input space.

In [101…

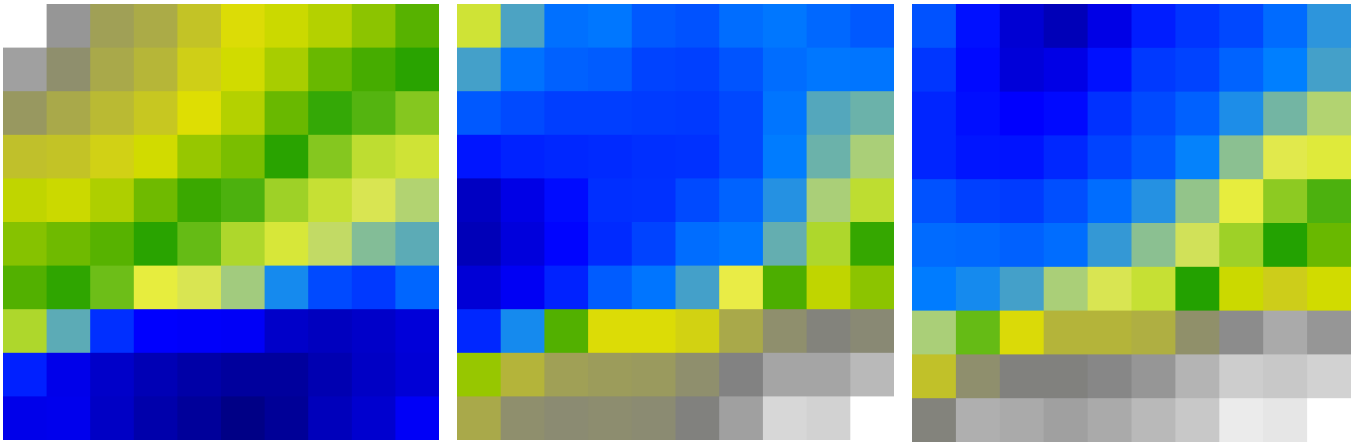
```
idata_iris = SOMToolBox_Parse("datasets/iris/iris.vec").read_weight_file()
weights_iris = SOMToolBox_Parse("datasets/iris/iris.wgt.gz").read_weight_file()

mst_units = hv.Segments(MST(10, 10, weights_iris['arr'], idata_iris['arr'], 0, 0, "none", "units"),
                  ['x', 'y', 'x1', 'y1'], 'value').opts(
    line_width=dim('value')*4, color="black", width=400, height=400)

mst_inputs = hv.Segments(MST(10, 10, weights_iris['arr'], idata_iris['arr'], 0, 0, "input_space", "input"),
                  ['x', 'y', 'x1', 'y1'], 'value').opts(
    line_width=dim('value')*4, color="blue", width=400, height=400)

pn.Row(mst_units, mst_inputs)
```

Out[101]:



Activity Histograms

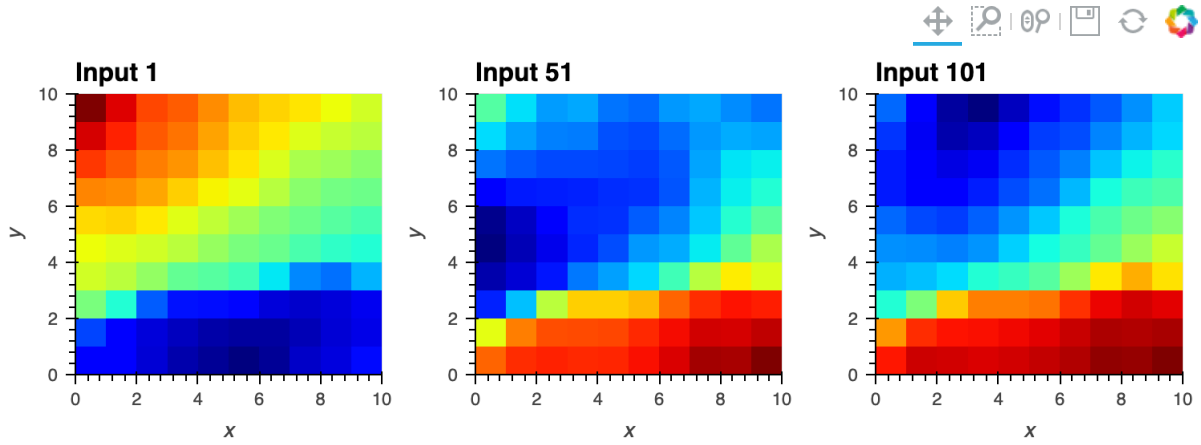
Once more we use the Iris Dataset for comparison. We render the Activity Histograms for input vectors 1, 51, and 101. Below you can see the Java SOMToolbox output and the results of our implementation for these same inputs.



In [102…

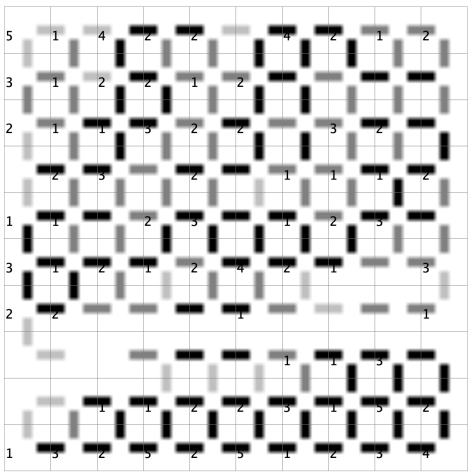
```
c1 = hv.Image(ActivityHist(10,10,weights_iris['arr'], idata_iris['arr'][0]),
              bounds=(0,0,10,10)).opts(width=250,height=250,cmap='jet')
c2 = hv.Image(ActivityHist(10,10,weights_iris['arr'], idata_iris['arr'][50]),
              bounds=(0,0,10,10)).opts(width=250,height=250, cmap='jet')
c3 = hv.Image(ActivityHist(10,10,weights_iris['arr'], idata_iris['arr'][100]),
              bounds=(0,0,10,10)).opts(width=250,height=250, cmap='jet')
hv.Layout([c1.relabel('Input 1'),c2.relabel('Input 51'),c3.relabel('Input 101')])
```

Out[102]:



Cluster Connections

Again, we used the Iris dataset for comparison. We used threshold values of 0.33, 0.5, and 1.0, at which the separation of the most distinct cluster (Setosa) from the two intersecting clusters (Versicolor, Virginica) is immediately apparent. Also noticeable is that the Setosa cluster is denser than the others, since the distance between its units undercuts the lowest threshold in almost all cases, while the units that cover the other two clusters are often further apart.



Below, you can see our implementation with the same parameters, yielding the same output (apart from the hit count that is used as a backdrop in the Java SOMToolbox).

In [103…

```
hv.Segments(ClusterConnection(10, 10, weights_iris['arr'], 0.33, 0.5, 1.0),
                  ['x', 'y', 'x1', 'y1'], 'value').opts(
    line_width=6, alpha=dim('value'), color="black", width=400, height=400,
    show_grid=True, xticks=list(range(11)), yticks=list(range(11)))
```

Out[103]: