SOM Visualisation with VectorFields

Authors: Mario Stoff - 11777706, Bernhard Weissenbach - 11777709

Submission Repository (https://github.com/BernyWeiss/VectorFields_Visualisation)

This Notebook implements the visualisation of Vectorfields for SOMs.

To use this notebook, just run all cells up to the definition of parameters; define the set of parameters according to the description; execute the Run Visualisaton cell to calculate and visualise the VectorField

This notebook uses numpy, holoviews, bokeh and IPython libraries. Make sure that these are available in the environment.

It also uses the SOMToolBox_Parse.py file to load SOMs trained with the the Java SOMToolbox (http://www.ifs.tuwien.ac.at/dm/somtoolbox/index.html).

In [1]:
```
import numpy as np

import holoviews as hv
from holoviews import opts
hv.extension('bokeh')

from typing import Literal

from IPython.display import HTML
```

Vectorfield Calculation

The next code-cell implements the calculation of flows for the VectorField visualisations. we used the FlowBorderlineVisualizer Class of the Java SOMToolbox (http://www.ifs.tuwien.ac.at/dm/somtoolbox/index.html) as a baseline. We first translated the Java Code directly to the corresponding Python syntax. Because we had performance issues with big SOMs, we then tried to make use of vector and matrix calculations which improved performance a lot.

We used code comments to describe the most important parts of the calculation. However, we want to mention the main steps here

- Validate attributes (mainly for Groups)
- Split the weights according provided grouping
- Initialize arrays for calculation results
- Calculate flow for each unit

In [2]:
```
ALLOWED_GROUP_INDICES = [None,0,1,2]

def Kernel(distance_in_output_space: np.ndarray, sigma):
    return np.exp((-1.0 * np.power(distance_in_output_space,2) / (2*sigma**2))

def VectorFields(ySize, xSize, _weights, _idata, kernel_size=None, type='', gr

    # find dimensionality of data and reshape weights according to it.
    n_dims_in_data = _idata.shape[1]
    reshaped_weights = _weights.reshape(ySize, xSize, n_dims_in_data)

    group_masks = list()

    if groups is not None:
        if groups.__len__() != n_dims_in_data:
            raise ValueError("groups doesn't have same number of dimensions as
        if any(group_idx not in ALLOWED_GROUP_INDICES for group_idx in groups)
            raise ValueError("Invalid group assigned to attribute, only use 0,

        # assigning weight dimensions to the respective groups
        group_index_values = np.unique(groups[[g is not None for g in groups]]
        n_groups = len(group_index_values)

        for g in group_index_values:
            group_masks.append(np.nonzero(groups==g)[0])

    else:
        # no grouping specified -> all columns are used
        n_groups = 1
        group_masks.append(np.array(range(n_dims_in_data)))

    grouped_weights = list()

    for mask in group_masks:
        # select relevant weight columns according to groups
        grouped_weights.append(reshaped_weights[:,:,mask])

    x_indices = np.array(range(ySize))
    y_indices = np.array(range(xSize))


    ax = np.zeros((n_groups, ySize, xSize))
    ay = np.zeros((n_groups, ySize, xSize))
    angles = np.zeros((n_groups, ySize, xSize))
    magnitudes = np.zeros((n_groups, ySize, xSize))

    for g in range(n_groups):
        for x in range(ySize):
            for y in range(xSize):

                # calculate the distance in output space
                doutx = x_indices-x
                douty = y_indices-y

                #create something like a mgrid but for distances of x and y to
```

```
                distance_grid = np.zeros((2,ySize,xSize))

                distance_grid[0] = np.repeat(doutx, len(douty)).reshape((len(d
                distance_grid[1] = np.repeat(douty, len(doutx)).reshape((len(d

                alpha = np.arctan2(distance_grid[1], distance_grid[0])

                # euclidian distance of current x any y to very other unit (ou
                dout = np.sqrt(distance_grid[0]**2 + distance_grid[1]**2)

                # calculation of the neighborhood based on distance in output
                h = Kernel(dout, sigma)

                # distance in feature space (L2 Norm)
                distances_feature_space = np.linalg.norm(grouped_weights[g][x,

                # scaling x and y component with kernel
                omegax = np.cos(alpha) * h
                omegay = np.sin(alpha) * h

                # set distance to itself to 0
                omegax[x,y] = 0
                omegay[x,y] = 0

                # defining index masks for positive and negative omega
                pos_omegax_idx = omegax>0.0
                pos_omegay_idx = omegay>0.0

                neg_omegax_idx = np.ones(omegax.shape,np.bool_)
                neg_omegax_idx[pos_omegax_idx]=0

                neg_omegay_idx = np.ones(omegay.shape,np.bool_)
                neg_omegay_idx[pos_omegay_idx]=0

                # multiplying distance values with kernel
                roxplus = omegax[pos_omegax_idx] * distances_feature_space[pos
                roxminus = -omegax[neg_omegax_idx] * distances_feature_space[n

                royplus = omegay[pos_omegay_idx] * distances_feature_space[pos
                royminus = -omegay[neg_omegay_idx] * distances_feature_space[n

                # define x and y component for group and unit
                ax[g][x][y] = -roxminus.sum() * omegax[pos_omegax_idx].sum() +
                ay[g][x][y] = -royminus.sum() * omegay[pos_omegay_idx].sum() +


                #convert x and y distance into magnitude and angle
                magnitudes[g][x][y] = np.sqrt(ax[g][x][y]**2 + ay[g][x][y]**2)
                if magnitudes[g][x][y] == 0: magnitudes[g][x][y] = 0.0001
                angles[g][x][y] = (np.pi/2.) - np.arctan2(ax[g][x][y]/magnitud


    gx, gy = np.mgrid[0:ySize, 0:xSize]

    return gx, gy, angles, magnitudes
```

```
#gx, gy, angles, magnitudes = VectorFields(weights['ydim'], weights['xdim'], w
```

Visualisation

The next cell defines the method to visualize vectorfields. It implements Flow, Borderline and combination of both.

The method calls the calculation method and visualises the result using holoviews and the bokeh library. This choice was made to be easily compatible with the PySOMVis (https://github.com/smnishko/PySOMVis/tree/main/PySOMVis) library.

We use holoviews' VectorField to plot the required vectors. Here we specify the positions of the vectors, as well as magnitude and angle. The options we use include using the magnitude for the length of the arrows, going through the given color cycle, and setting the pivot point to 'tail'. This last one makes it so that the arrows originate at the given position, the standard value is 'mid' where the center of the arrow is positioned at the given x/y.
For borderline visualizations, we add 180 degrees to the calcualated angle and remove the arrow tips via the options. We also leave the arrow pivot at 'mid' which centers the borderline over the unit's position.

In [3]:
```
# These are the allowed types of visualization, throws an error if none of the
Visualizations = Literal['flow', 'borderline', 'both']

def visualizeVectorFields(vis: Visualizations, groups=None, sigma=1.5):

    # The visualization goes through this color cycle depending on how many gr
    color_cycle = hv.Cycle(['blue', 'red', 'green', 'cyan', 'pink', 'yellow'])

    # Calculation of the vector values
    gx, gy, angles, magnitudes = VectorFields(weights['ydim'], weights['xdim']

    # Matching the type of chosen visualization, plot the necessary vectorfiel
    match vis:
        case 'flow':
            overlay = None
            # len(angles) is the number of groups
            for i in range(0,len(angles)):
                if overlay == None:
                    overlay = hv.VectorField((gx, gy, angles[i], magnitudes[i]
                else:
                    overlay = overlay * hv.VectorField((gx, gy, angles[i], mag
            return overlay

        case 'borderline':
            overlay = None
            # len(angles) is the number of groups
            for i in range(0,len(angles)):
                if overlay == None:
                    overlay = hv.VectorField((gx, gy, angles[i]+(np.pi/2.), ma
                else:
                    overlay = overlay * hv.VectorField((gx, gy, angles[i]+(np.
            return overlay

        case 'both':
            overlay = None
            # len(angles) is the number of groups
            for i in range(0,len(angles)):
                if overlay == None:
                    overlay = hv.VectorField((gx, gy, angles[i], magnitudes[i]
                    overlay = overlay * hv.VectorField((gx, gy, angles[i]+(np.
                else:
                    overlay = overlay * hv.VectorField((gx, gy, angles[i], mag
                    overlay = overlay * hv.VectorField((gx, gy, angles[i]+(np.
            return overlay

        case _:
            raise ValueError("vis must be either 'flow', 'borderline', or 'bot
    return
```

Loading a pretrained SOM

The next code-cell defines a method to load one of a few pretrained SOMs. The choice is made via the dataset parameter. This supports only SOMs trained with the Java SOMToolbox.

Loading is done with the SOMToolBox_Parse file which was given in the PySOMVis (https://github.com/smnishko/PySOMVis/tree/main/PySOMVis) Library. Choices are described in the next cell.

In [4]:
```
from SOMToolBox_Parse import SOMToolBox_Parse
def loadData(dataset):

    match dataset:
        case "10clusters_small":
            dataset_name= "10 Clusters Small"
            idata = SOMToolBox_Parse("datasets\\10clusters_small\\10clusters.v
            weights = SOMToolBox_Parse("datasets\\10clusters_small\\10clusters
        case "10clusters_big":
            dataset_name= "10 Clusters Big"
            idata = SOMToolBox_Parse("datasets\\10clusters_big\\10clusters.vec
            weights = SOMToolBox_Parse("datasets\\10clusters_big\\10clusters_b
        case "chainlink_small":
            dataset_name= "Chainlink Small"
            idata = SOMToolBox_Parse("datasets\\chainlink_small\\chainlink.vec
            weights = SOMToolBox_Parse("datasets\\chainlink_small\\chainlink_s
        case "chainlink_big":
            dataset_name= "Chainlink Big"
            idata = SOMToolBox_Parse("datasets\\chainlink_big\\chainlink.vec")
            weights = SOMToolBox_Parse("datasets\\chainlink_big\\chainlink_big
        case _:
            raise ValueError("dataset must be either '10clusters_small', '10cl
    return (dataset_name, idata, weights)
```

Defining Parameters

The next cell is used to define parameters for the calculation.

dataset: describes which dataset to load
available datasets: '10clusters_small', '10clusters_big', 'chainlink_small', 'chainlink_big'

vis_type: type of visualization
available types: 'flow', 'borderline', 'both'

sigma: specifies the kernel size for calculation
high sigma = more global influence, low sigma = more local influence

groups: assign attributes to different groups (None, 0, 1, 2) or None (all attributes collectively)

Examples:
All attributes: groups=None
Chainlink: 3 attributes, eg. groups=np.array([0,0,1])
10-clusters: 10 attributes, eg. groups=np.array([0,0,0,1,1,1,2,2,2,None])

In [5]:
```
dataset = "10clusters_small"
vis_type = "flow"
sigma = 1.5

groups = None
#groups = np.array([0,0,2])
```

Run Visualisation

The next cell contains a call for the visualisation method.

It loads the correct dataset, calculates vector values, and plots the resulting visualization

In [6]:
```
dataset_name, idata, weights = loadData(dataset)

visualization = visualizeVectorFields(vis=vis_type, groups=groups, sigma=sigma

title = str(weights['xdim']) + "x" + str(weights['ydim']) + " SOM: " + dataset
aspect_ratio = weights['xdim']/weights['ydim']
```

Out[6]:

Examples and Evaluation

The following cells show the results of our implementation, as well as comparisons to what the SOMViewer of the java SOMToolbox produces with the same SOMs. Our solutions are shown on the left, and the SOMToolbox plots on the right.

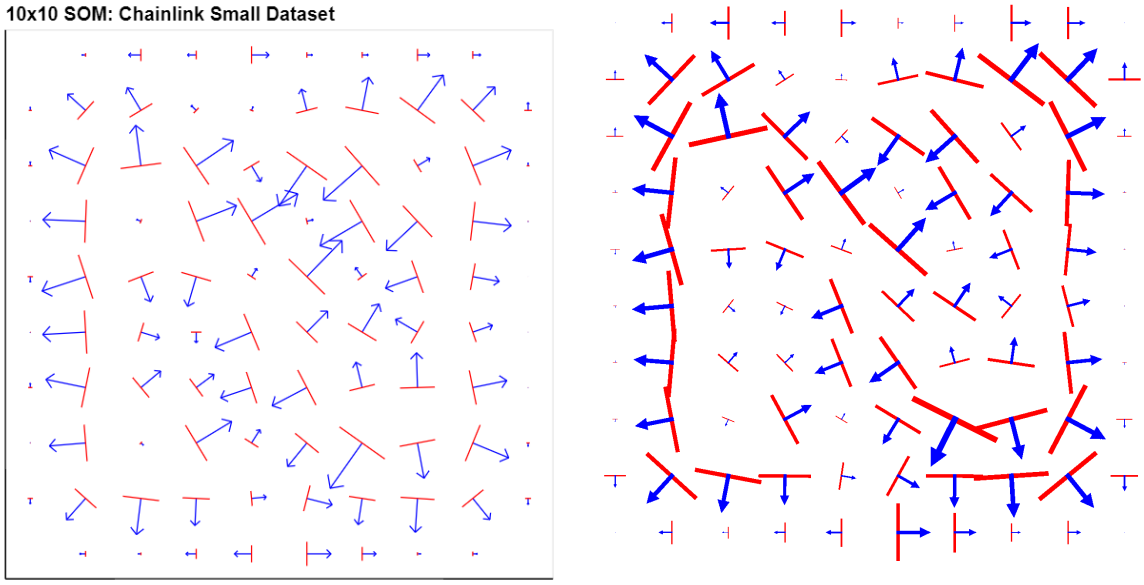
With the first three comparisons we want to show that our implementation works correctly with different settings. The first one shows both the flow vectors and borderline visualization on the small chainlink SOM. The second illustrates the grouping function by assigning each of the dataset's dimensions to their own group. This results in three vectors per unit. The third small visualization shows that it also works on the 10 Clusters SOM.
With just this small size of SOM we can already clearly show cluster boundaries, but obviously the small resolution cannot show all the intricacies of these datasets which is why we also visualize the vector fields on a bigger SOM afterwards.

10x10 SOM Chainlink Dataset

Training: lr=0.7 sigma=6 iterations=10000
Visualization: sigma = 2

In [7]:
```
HTML(f"""
        <div class="row" style="display:flex">
          <div class="column" style="flex:50%">
            <img src="pics/chainlink_small.png" style="width:100%">
          </div>
          <div class="column" style="flex:50%">
            <img src="pics/chainlink_small_java.png" style="width:95%; transfo
          </div>
        </div>
    """)
```

Out[7]:



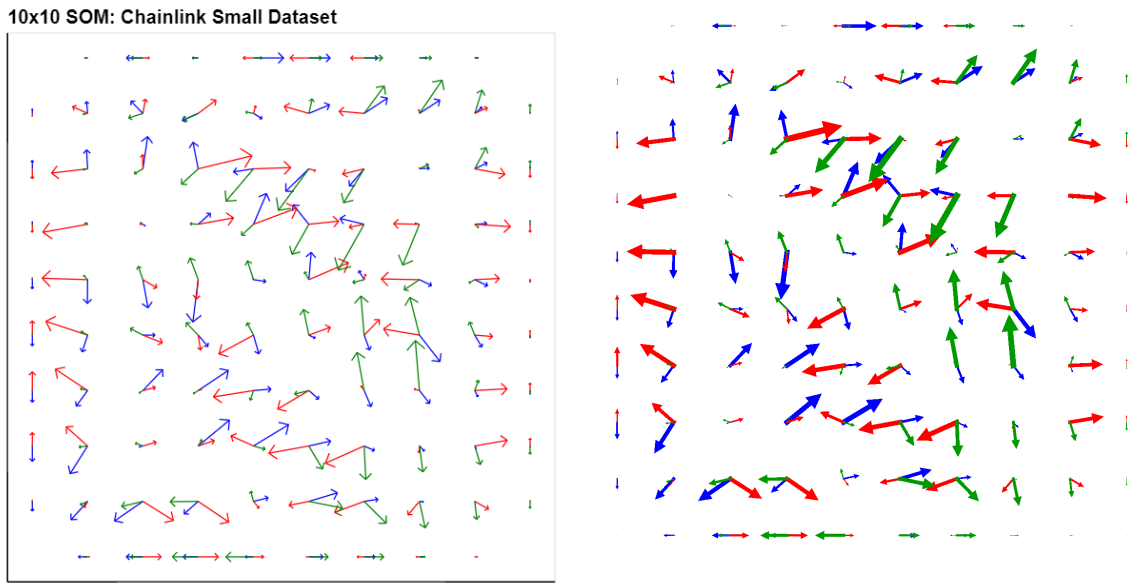
10x10 SOM Chainlink Dataset 3 Groups

Training: lr = 0.7; sigma = 6; iterations = 10000

Visualization: sigma = 1.5

In [8]:
```
HTML(f"""
        <div class="row" style="display:flex">
          <div class="column" style="flex:50%">
            <img src="pics/chainlink_small_3g.png" style="width:100%">
          </div>
          <div class="column" style="flex:50%">
            <img src="pics/chainlink_small_3g_java.png" style="width:95%; tran
          </div>
        </div>
    """)
```

Out[8]:



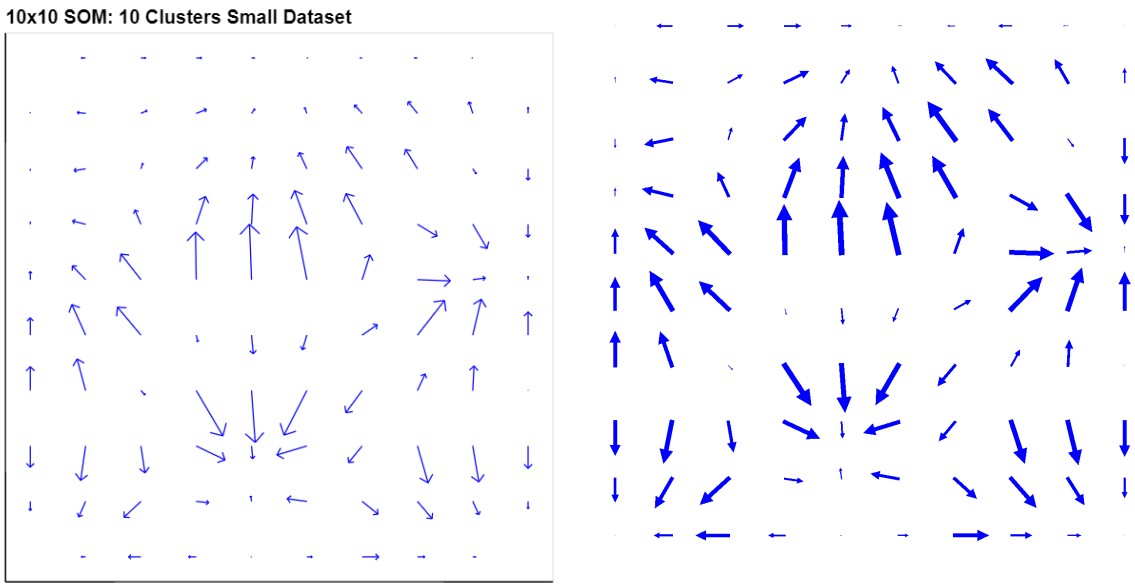
10x10 SOM 10 Clusters Dataset

Training: lr = 0.7; sigma = 6; iterations=10000

Visualization: sigma = 1.7

In [9]:
```
HTML(f"""
        <div class="row" style="display:flex">
          <div class="column" style="flex:50%">
            <img src="pics/10clusters_small.png" style="width:100%">
          </div>
          <div class="column" style="flex:50%">
            <img src="pics/10clusters_small_java.png" style="width:95%; transf
          </div>
        </div>
    """)
```

Out[9]:



Bigger SOMs

With the bigger SOMs we show some different parameters to see their effects.

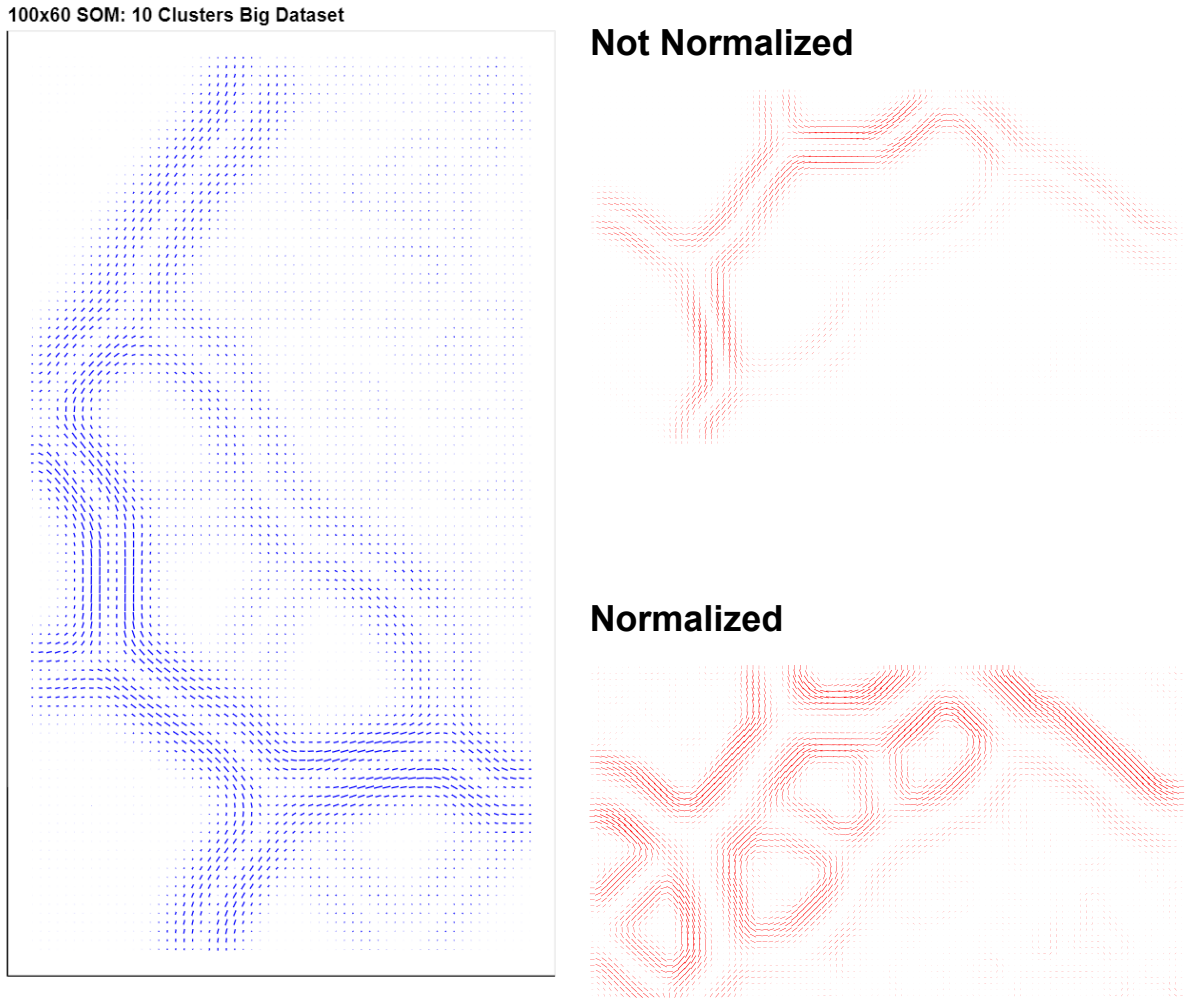
60x100 SOM 10 Clusters Dataset

Training: lr = 0.7; sigma = 10; iterations=10000
Visualization: sigma = 1.7; grouping = None

The next visualization shows the comparison of the big 10 Clusters SOM. The two plots on the right are rotated by 90 degrees for spacing purposes.
The SOMToolbox offers a checkbox for edge normalization. This is a functionality that we have not implemented in our solution. The difference this make can be well illustrated in this example. Where the normalized visualization shows well the "weaker" cluster boundaries (eg. between the three clusters in the middle), these same boundaries are only faintly visible in our implementation. The same goes for the not normalized version of the Java SOMToolbox. We do think, however, that these boundaries are still visible enough in our solution, but adding normalization would be a good potential improvement.

In [10]:
```
HTML(f"""
        <div class="row" style="display:flex">
          <div class="column" style="flex:50%">
            <img src="pics/10clusters_big.png" style="width:100%">
          </div>
          <div class="column" style="flex:50%">
            <h3>Not Normalized</h3>
            <img src="pics/10clusters_big_border_java.png" style="width:100%;
            <h3 style="width:100%; transform: translateY(50px)">Normalized</h3
            <img src="pics/10clusters_big_border_normalized_java.png" style="w
          </div>
        </div>
    """)
```

Out[10]:



Effects of kernel size (sigma)

The next two plots show the influence of the kernel size when calculating the vectors for visualization. For this, we use the large chainlink SOM with the first two dimensions of the data being shown in blue and the third dimension in red. The first plot uses a small kernel size (sigma=1.5) and the second plot uses a relatively large kernel (sigma=20). We can see that a large kernel results in smoother and overlapping areas of influence, whereas a small kernel leads to sharper edges.
It can also be seen pretty clearly in these plots, that the different data dimensions have varying

influence in different area of the mapped data space.

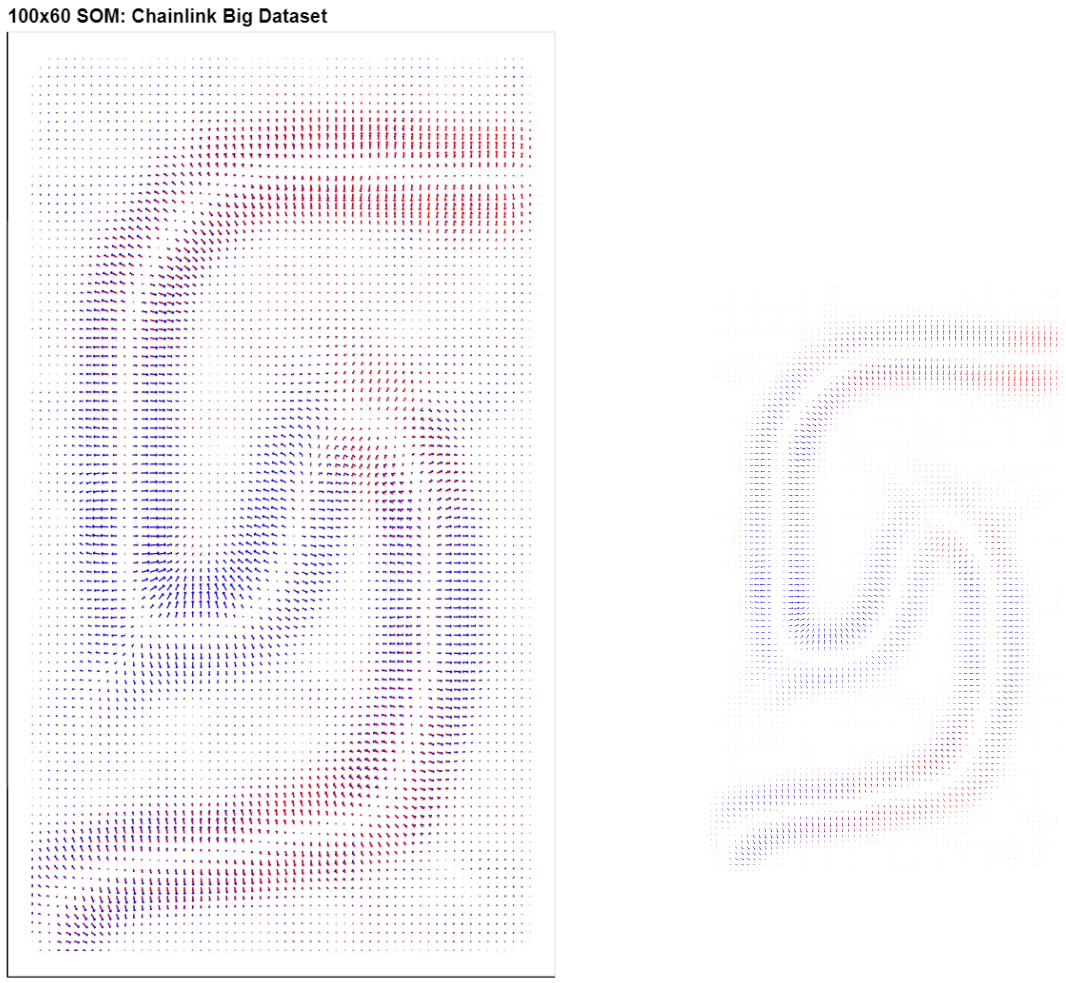
60x100 SOM Chainlink Dataset

Training: lr=0.7 sigma=20 iterations=10000

Visualization: sigma = 1.5

In [11]:
```
HTML(f"""
        <div class="row" style="display:flex">
          <div class="column" style="flex:50%">
            <img src="pics/chainlink_big_1_5.png" style="width:100%">
          </div>
          <div class="column" style="flex:50%">
            <img src="pics/chainlink_big_1_5_java.png" style="width:100%; tran
          </div>
        </div>
    """)
```

Out[11]:



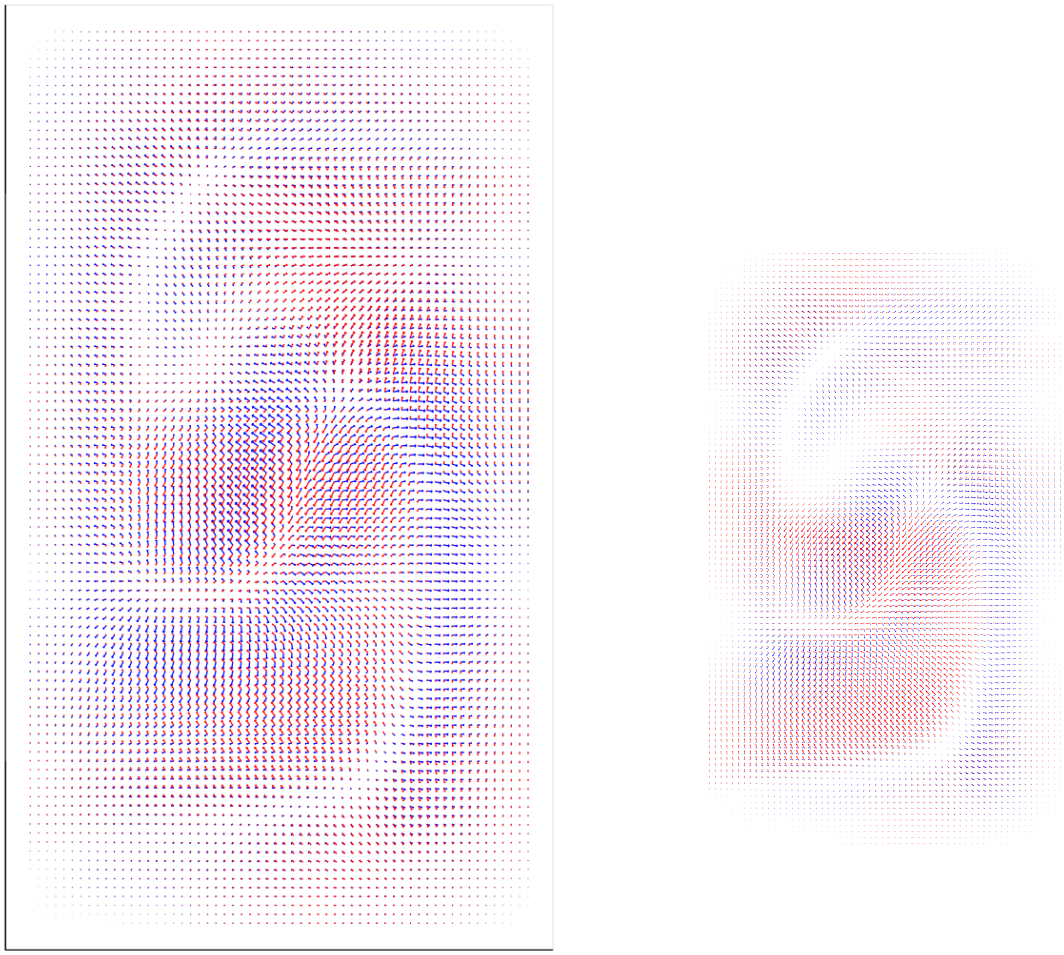
60x100 SOM Chainlink Dataset

Training: lr=0.7 sigma=20 iterations=10000

Visualization: sigma = 20

In [13]:
```
HTML(f"""
        <div class="row" style="display:flex">
          <div class="column" style="flex:50%">
            <img src="pics/chainlink_big_20.png" style="width:100%">
          </div>
          <div class="column" style="flex:50%">
            <img src="pics/chainlink_big_20_java.png" style="width:100%; trans
          </div>
        </div>
    """)
```

Out[13]:



In [ ]: