Kohonen Algorithm
Neuro Computation
Part B

Anthony Assayah – 209971258

Nerya Bigon – 208240754

Najeeb Abdalla - 316436328

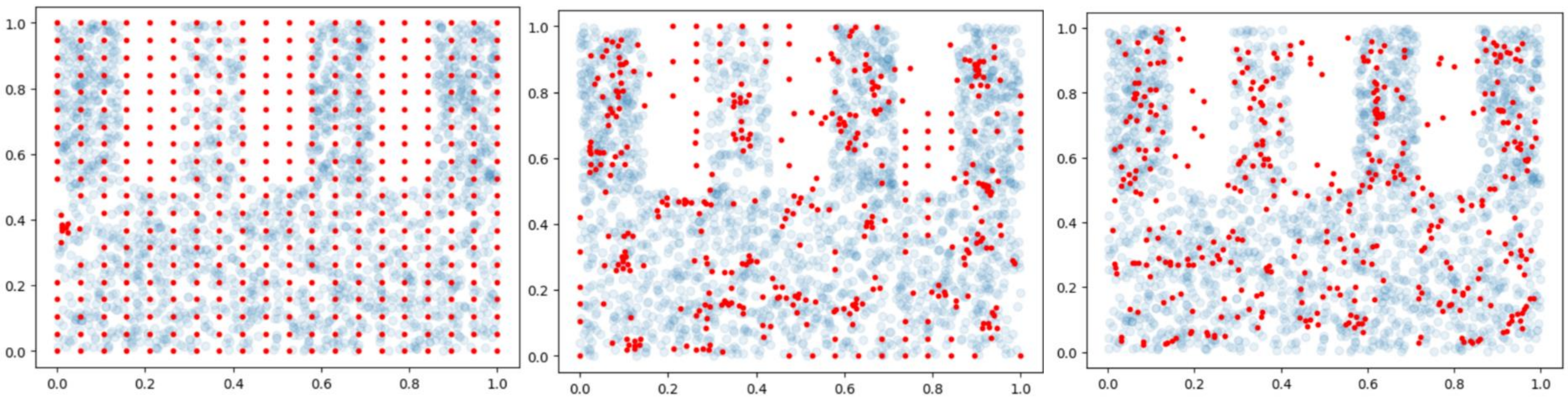
Part B.1: Discussion and illustration:

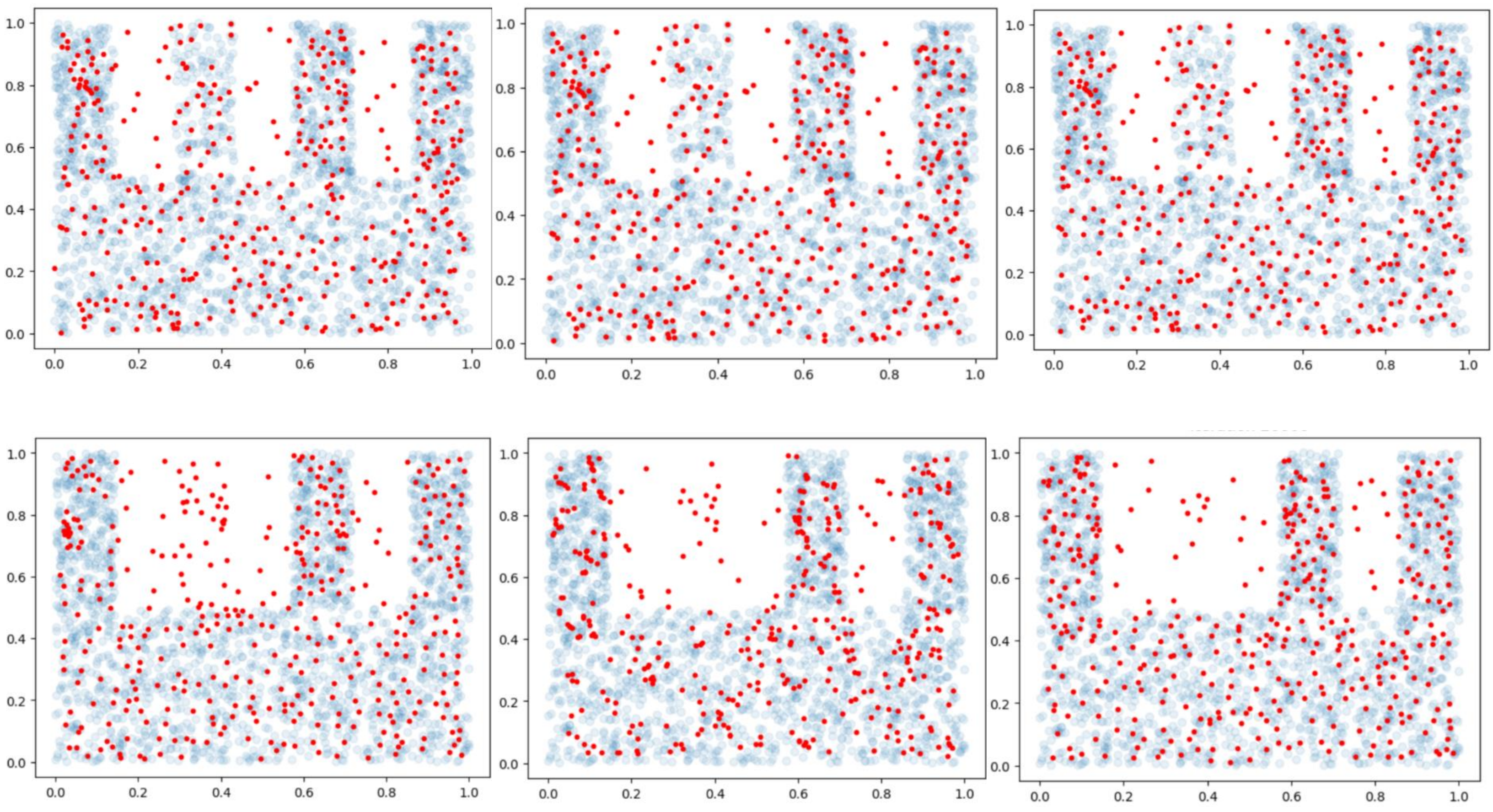
In the second part of the assignment, we focus on visualizing the self-organizing map (SOM) mesh superimposed on a diagram representing a hand shape. The data points, denoted by <x, y>, are confined to the region within the hand, which is a subset of the coordinate space defined as {<x, y> | 0 ≤ x ≤ 1, 0 ≤ y ≤ 1}. The SOM consists of a grid of 400 neurons arranged in a 20x20 mesh. We explore how the SOM mesh adapts and changes over iterations to capture the underlying patterns within the "monkey" hand-shaped data. Additionally, we introduce a modification by "cutting off a finger," restricting the data points to the hand with three fingers and continuing from the stopping point of the previous section. By visualizing the evolving mesh, we gain insights into the self-organization and representation capabilities of the SOM algorithm in capturing complex spatial relationships within the hand-shaped dataset.

By observing the results, we can analyze the impact of the number of neurons on the quality of the representation and the ability to capture the underlying structure of the data. In this report you will find all the **illustrations of evolution of the maps** with explanation and **screenshot of the code**.

GitHub link: https://github.com/AnthonyAssayah/Kohonen_NeuroComputation.git (you can see the code there too)

We run the Kohonen algorithm on 400 neurons using a dataset of 2200 input data points.



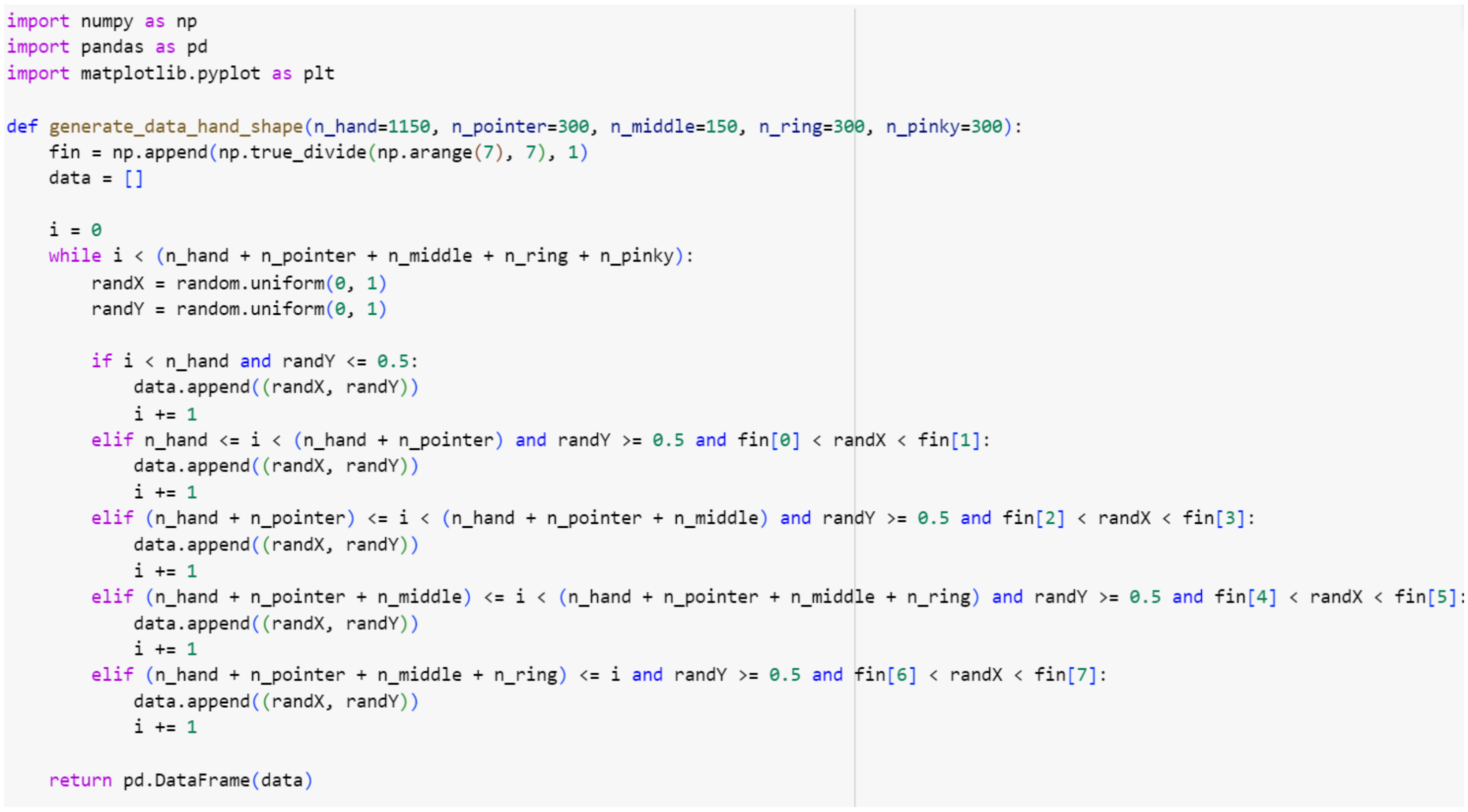


<u>Observation and Conclusion:</u>

Upon observing the evolution of the self-organizing map (SOM) mesh from the initial hand shape with four fingers to the modified hand shape with three fingers, we can draw several conclusions. Initially, the SOM mesh aligns itself to the distribution of data points within the hand, effectively capturing the overall shape and spatial arrangement of the fingers. As the iterations progress, the SOM mesh undergoes a process of adaptation, refining its representation to better fit the data. Notably, during the transition from four fingers to three fingers, the SOM mesh adjusts and reallocates its neurons to accommodate the new hand shape. This demonstrates the capability of the SOM algorithm to flexibly adapt and self-organize based on the underlying patterns in the data. Overall, the visualized evolution of the SOM mesh provides insights into the algorithm's ability to capture and represent complex spatial relationships, showcasing its usefulness in analyzing and understanding multidimensional data structures.

Algorithm and explanation:

1) Generate data function:

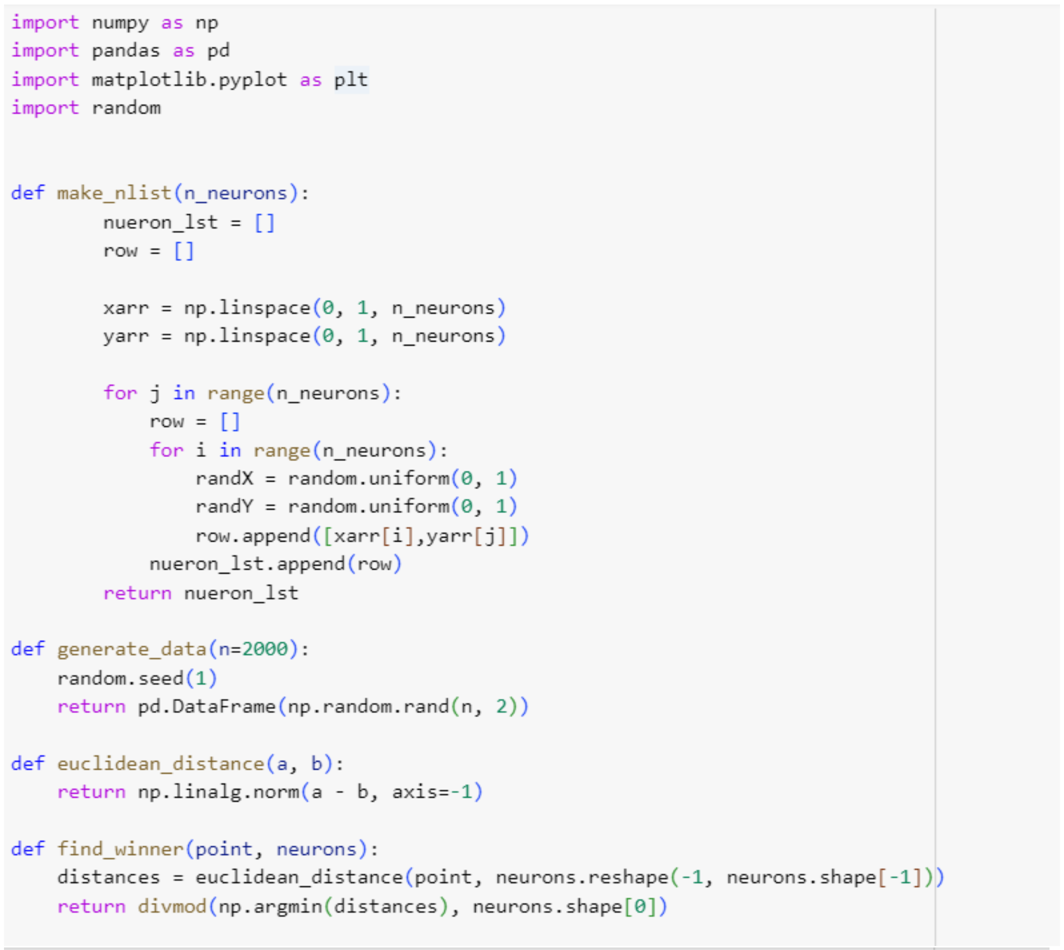


```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def generate_data_hand_shape(n_hand=1150, n_pointer=300, n_middle=150, n_ring=300, n_pinky=300):
    fin = np.append(np.true_divide(np.arange(7), 7), 1)
    data = []

    i = 0
    while i < (n_hand + n_pointer + n_middle + n_ring + n_pinky):
        randX = random.uniform(0, 1)
        randY = random.uniform(0, 1)

        if i < n_hand and randY <= 0.5:
            data.append((randX, randY))
            i += 1
        elif n_hand <= i < (n_hand + n_pointer) and randY >= 0.5 and fin[0] < randX < fin[1]:
            data.append((randX, randY))
            i += 1
        elif (n_hand + n_pointer) <= i < (n_hand + n_pointer + n_middle) and randY >= 0.5 and fin[2] < randX < fin[3]:
            data.append((randX, randY))
            i += 1
        elif (n_hand + n_pointer + n_middle) <= i < (n_hand + n_pointer + n_middle + n_ring) and randY >= 0.5 and fin[4] < randX < fin[5]:
            data.append((randX, randY))
            i += 1
        elif (n_hand + n_pointer + n_middle + n_ring) <= i and randY >= 0.5 and fin[6] < randX < fin[7]:
            data.append((randX, randY))
            i += 1

    return pd.DataFrame(data)
```

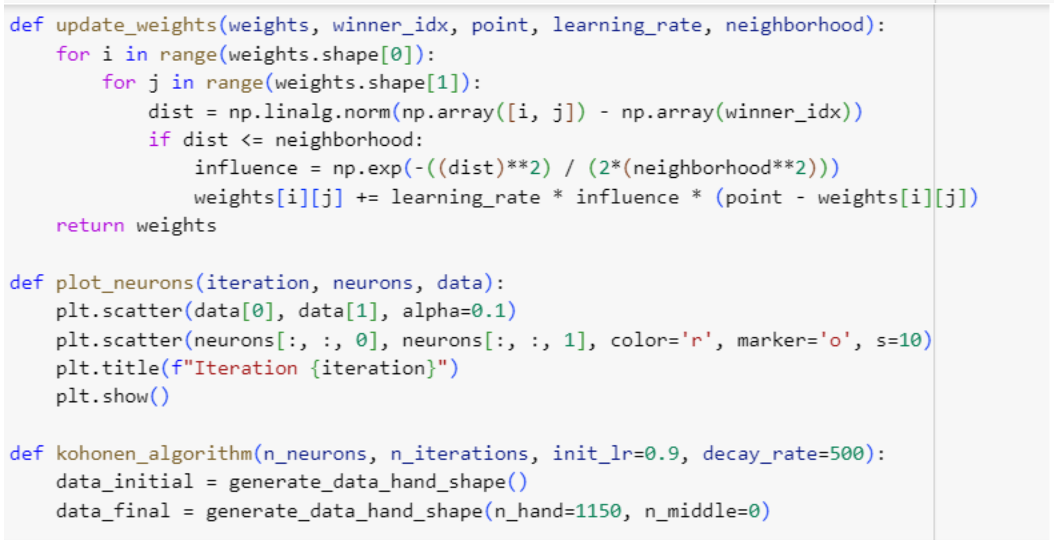
The **generate_data_hand_shape** function generates a dataset representing a hand shape with specified finger counts. The function takes several parameters: **n_hand** represents the number of points within the hand shape, n_pointer represents the number of points in the pointer finger, n_middle represents the number of points in the middle finger**, n_ring** represents the number of points in the ring finger, and **n_pinky** represents the number of points in the pinky finger.

The function uses random sampling to generate points within the defined regions of the hand shape. It ensures that the points are distributed according to the specified finger counts. The resulting points are returned as a Pandas DataFrame, providing a dataset that can be used for training or analysis in various machine learning tasks.

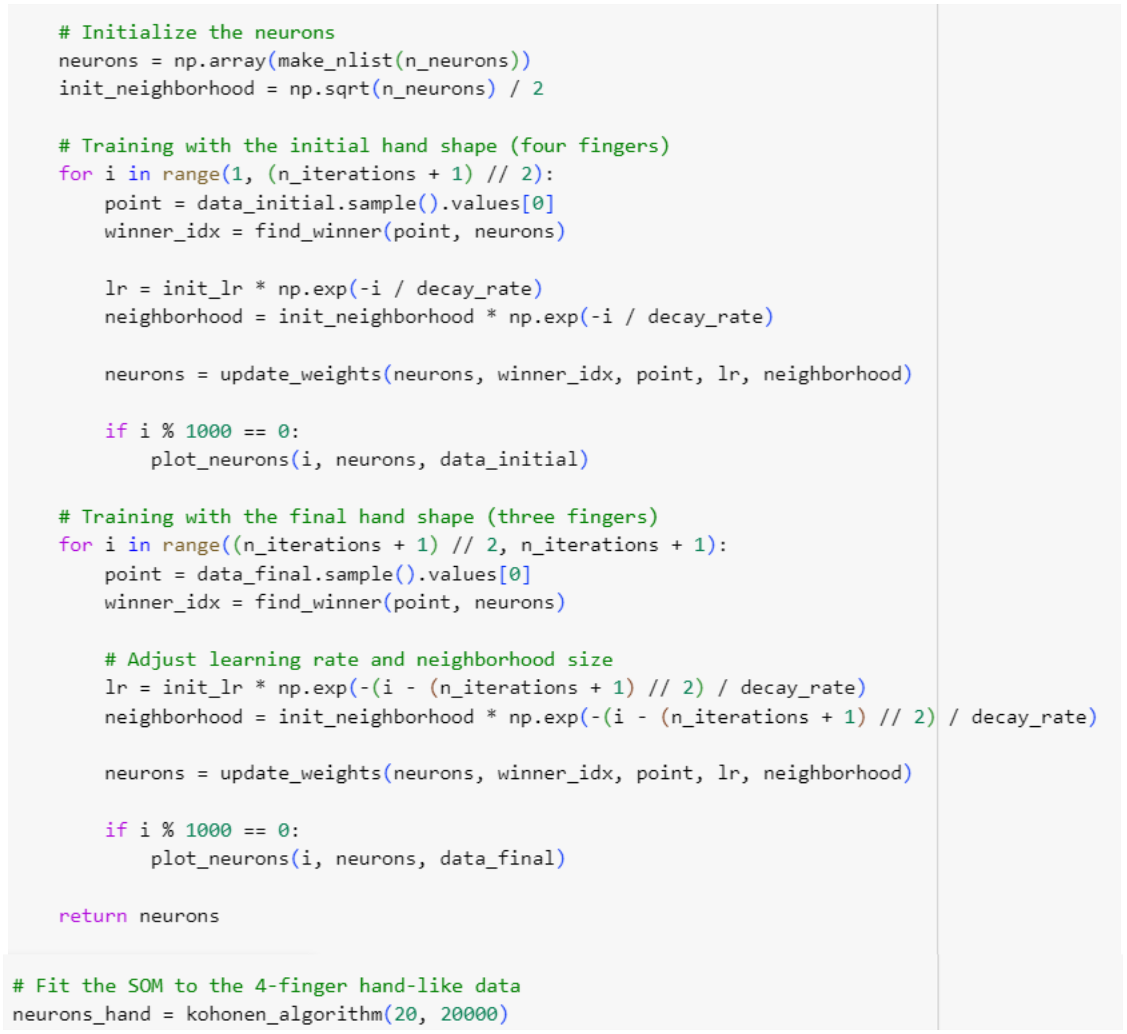
2) Kohonen algorithm:



```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random


def make_nlist(n_neurons):
        nueron_lst = []
        row = []

        xarr = np.linspace(0, 1, n_neurons)
        yarr = np.linspace(0, 1, n_neurons)

        for j in range(n_neurons):
            row = []
            for i in range(n_neurons):
                randX = random.uniform(0, 1)
                randY = random.uniform(0, 1)
                row.append([xarr[i],yarr[j]])
            nueron_lst.append(row)
        return nueron_lst

def generate_data(n=2000):
    random.seed(1)
    return pd.DataFrame(np.random.rand(n, 2))

def euclidean_distance(a, b):
    return np.linalg.norm(a - b, axis=-1)

def find_winner(point, neurons):
    distances = euclidean_distance(point, neurons.reshape(-1, neurons.shape[-1]))
    return divmod(np.argmin(distances), neurons.shape[0])
```



```
def update_weights(weights, winner_idx, point, learning_rate, neighborhood):
    for i in range(weights.shape[0]):
        for j in range(weights.shape[1]):
            dist = np.linalg.norm(np.array([i, j]) - np.array(winner_idx))
            if dist <= neighborhood:
                influence = np.exp(-((dist)**2) / (2*(neighborhood**2)))
                weights[i][j] += learning_rate * influence * (point - weights[i][j])
    return weights

def plot_neurons(iteration, neurons, data):
    plt.scatter(data[0], data[1], alpha=0.1)
    plt.scatter(neurons[:, :, 0], neurons[:, :, 1], color='r', marker='o', s=10)
    plt.title(f"Iteration {iteration}")
    plt.show()

def kohonen_algorithm(n_neurons, n_iterations, init_lr=0.9, decay_rate=500):
    data_initial = generate_data_hand_shape()
    data_final = generate_data_hand_shape(n_hand=1150, n_middle=0)
```



```
    # Initialize the neurons
    neurons = np.array(make_nlist(n_neurons))
    init_neighborhood = np.sqrt(n_neurons) / 2

    # Training with the initial hand shape (four fingers)
    for i in range(1, (n_iterations + 1) // 2):
        point = data_initial.sample().values[0]
        winner_idx = find_winner(point, neurons)

        lr = init_lr * np.exp(-i / decay_rate)
        neighborhood = init_neighborhood * np.exp(-i / decay_rate)

        neurons = update_weights(neurons, winner_idx, point, lr, neighborhood)

        if i % 1000 == 0:
            plot_neurons(i, neurons, data_initial)

    # Training with the final hand shape (three fingers)
    for i in range((n_iterations + 1) // 2, n_iterations + 1):
        point = data_final.sample().values[0]
        winner_idx = find_winner(point, neurons)

        # Adjust learning rate and neighborhood size
        lr = init_lr * np.exp(-(i - (n_iterations + 1) // 2) / decay_rate)
        neighborhood = init_neighborhood * np.exp(-(i - (n_iterations + 1) // 2) / decay_rate)

        neurons = update_weights(neurons, winner_idx, point, lr, neighborhood)

        if i % 1000 == 0:
            plot_neurons(i, neurons, data_final)

    return neurons

# Fit the SOM to the 4-finger hand-like data
neurons_hand = kohonen_algorithm(20, 20000)
```

The code implements the Kohonen Self-Organizing Map (SOM) algorithm. The kohonen_algorithm function trains a SOM to capture patterns in a hand-shaped dataset. It initializes a grid of neurons, represented as a numpy array, and iteratively adjusts the weights of the neurons to match the input data.

The training process consists of two stages. In the first stage, the SOM is trained on a hand shape with four fingers. The learning rate and neighborhood size decrease over time, allowing the neurons to adapt to the input data. The progress is visualized with scatter plots at regular intervals.

In the second stage, the SOM is trained on a modified hand shape with three fingers. The learning rate and neighborhood size are adjusted separately for this stage. The training continues, and the final state of the neurons is visualized using scatter plots.

The resulting trained neurons, neurons_hand, represent the captured patterns and can be further analyzed or used for other purposes.