Homework 2

Anthon Odengard
19950114-3953

November 2023

1 Self organizing map

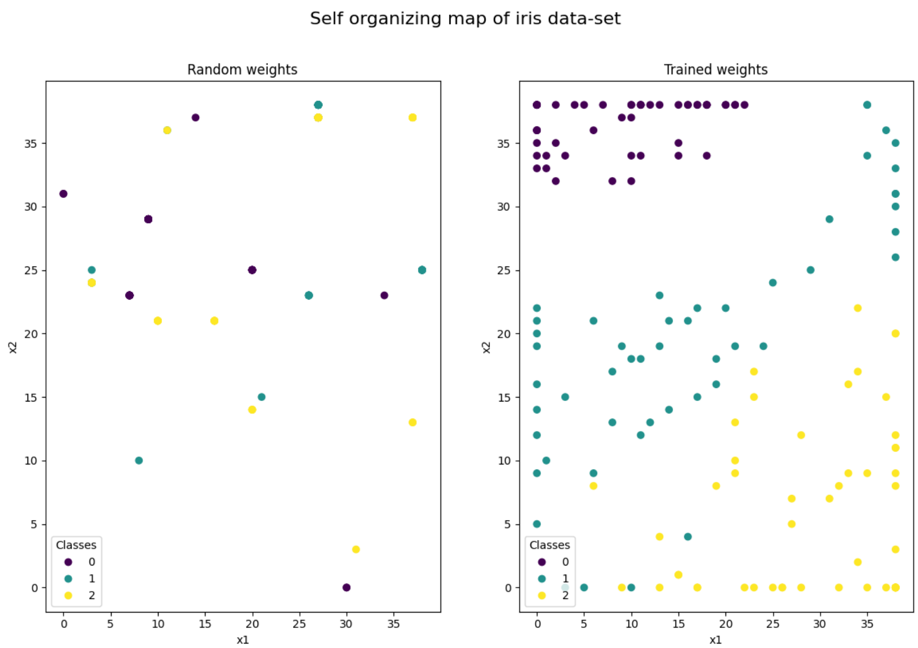

Figure 1: Winning neurons of self organizing map before and after training of weights

The result yielded three easily distinguishable clusters, with a few outliers. The weight were trained by applying the learning rule that was given in the problem description. The fact that the outliers mainly is centered around the not so dense regions on the cluster boundaries indicates that the results are reasonable. This is a result of the learning process were the weight vectors are trained to reflect the distribution of the input patterns, which has low density near the boundaries and therefore is harder to distinguish/learn.

```python
import numpy
import numpy as np
import sklearn.linear_model

class Reservoir:

    def __init__(self, input_weights, reservoir_connection_weight, output_shape):
        self.input_weights = input_weights
        self.reservoir_connection_weights = reservoir_connection_weight
        self.output_weights = 0
        self.reservoir_states = np.zeros((reservoir_connection_weight.shape[0], 1))
        self.training_output = np.zeros(output_shape)

    def update_reservoir_state(self, inputs):
        self.reservoir_states = np.zeros((self.reservoir_connection_weights.shape[0], inputs.shape[0]+1))
        i = 0
        for input in inputs:
            reservoir_contribution = np.matmul(self.reservoir_connection_weights, self.reservoir_states[:, i])
            input_contribution = np.matmul(self.input_weights, input)
            local_field = reservoir_contribution + input_contribution
            reservoir_new_state = np.tanh(local_field)
            self.reservoir_states[:, i+1] = reservoir_new_state
            i = i + 1
        return self.reservoir_states

    def predict_output(self):

        output = np.matmul(self.output_weights, self.reservoir_states[:, -1])
        reservoir_contribution = np.matmul(self.reservoir_connection_weights, self.reservoir_states[:, -1])
        input_contribution = np.matmul(self.input_weights, output)
        local_field = reservoir_contribution + input_contribution
        reservoir_new_state = np.tanh(local_field)
        self.reservoir_states = np.c_[self.reservoir_states, reservoir_new_state]
        return output

    def train_output_weights(self, training_set):
        self.reservoir_states = np.zeros((self.reservoir_states.shape[0], 1))
        training_set_transpose = np.transpose(training_set)
        self.update_reservoir_state(training_set_transpose[:-1, :])
        targets = training_set
        ridge1 = np.matmul(targets, self.reservoir_states.T)
        ridge2 = np.linalg.inv((np.matmul(self.reservoir_states, self.reservoir_states.T) + 0.01 * np.identity(500)))
        ridge = np.matmul(ridge1,ridge2)
        print(ridge)
        self.output_weights = ridge

    def iterate_reservoir_state(self, inputs, number_of_predictions):
        self.reservoir_states = np.zeros((self.reservoir_states.shape[0], 1))

        inputs_transpose = np.transpose(inputs)
        outputs = np.zeros((self.output_weights.shape[0], number_of_predictions))
        self.update_reservoir_state(inputs_transpose[:-1, :])
        t = 0
        while t < number_of_predictions:
            input = self.predict_output()
            outputs[:, t] = input
            t = t + 1
        return outputs
```

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

import Reservoir


training_set = np.genfromtxt('training-set.csv', delimiter=',')
validation_set = np.genfromtxt('test-set.csv', delimiter=',')
print(training_set.shape)
input_size = 3
number_of_reservoir_neurons = 500
input_weight_variance = 0.002
reservoir_weight_variance = 2/500



input_weights = np.random.normal(0, np.sqrt(input_weight_variance), (number_of_reservoir_neurons, input_size))
reservoir_connection_weights = np.random.normal(0, np.sqrt(reservoir_weight_variance), size=(number_of_reservoir_neurons,

                                                                               number_of_reservoir_neurons))

reservoir = Reservoir.Reservoir(input_weights, reservoir_connection_weights, 3)

reservoir.train_output_weights(training_set)


outputs = reservoir.iterate_reservoir_state(validation_set, 500)
predictions = pd.DataFrame(outputs)
predictions.to_csv('predicitons.csv', index=False, header=False)
y_coordinates = pd.DataFrame(outputs[1])
print(y_coordinates.transpose())
y_coordinates.to_csv('y_coordinate.csv', index=False, header=False)
print()
```

```python
import numpy as np
import sys
import self_organizing_map
np.set_printoptions(threshold=sys.maxsize)
import matplotlib.pyplot as plt
iris_label = np.genfromtxt('iris-labels.csv', delimiter=',')
iris_data = np.genfromtxt('iris-data.csv', delimiter=',')
iris_max = np.max(iris_data.flatten())
training_data = iris_data/iris_max

eta = 0.1
eta_decay = 0.01
sigma = 10
sigma_decay = 0.05
epochs = 10

weight_matrix = np.random.uniform(low=0, high=1, size=(40, 40, 4))

map = self_organizing_map.self_organizing_map(weight_matrix)
heat_map_untrained = map.generate_heat_map(training_data)
map.train_network(training_data, epochs, eta, sigma, sigma_decay,eta_decay)
heat_map_trained = map.generate_heat_map(training_data)

figure, axis = plt.subplots(1, 2)
scatter_1 = axis[0].scatter(heat_map_untrained[:, 0], heat_map_untrained[:, 1], c=iris_label)
scatter_2 = axis[1].scatter(heat_map_trained[:, 0],heat_map_trained[:, 1], c=iris_label)

axis[0].set_xlabel('x1')
axis[0].set_ylabel('x2')
axis[0].legend(*scatter_1.legend_elements(), loc="lower left", title="Classes")
axis[1].set_xlabel('x1')
axis[1].set_ylabel('x2')
axis[1].legend(*scatter_1.legend_elements(), loc="lower left", title="Classes")
axis[0].set_title('Random weights')
axis[1].set_title('Trained weights')
figure.suptitle('Self organizing map of iris data-set', fontsize=16)
plt.show()
```

```python
import numpy as np


class self_organizing_map:

    def __init__(self, weight_matrix):
        self.weight_matrix = weight_matrix
        self.output = 0

    def evaluate_network(self, input):
        output = np.matmul(self.weight_matrix, input)
        return output

    def winning_neuron(self, input):
        distance_matrix = self.weight_matrix - input
        distance_matrix = np.linalg.norm(distance_matrix, axis=2)

        return distance_matrix

    def get_minimum_distance_index(self, distance_matrix):
        minimum_index = np.unravel_index(np.argmin(distance_matrix), distance_matrix.shape)
        return minimum_index

    def get_maximum_value_index(self, output):
        maximum_index = np.unravel_index(np.argmax(output), output.shape)
        return maximum_index

    def calculate_euclidean_distance(self, output):
        index_matrix = np.moveaxis(np.mgrid[:self.weight_matrix.shape[0], :self.weight_matrix.shape[0]], 0, -1)
        distance_matrix = index_matrix - output
        distance_matrix = np.linalg.norm(distance_matrix, axis=2)
        return distance_matrix

    def neighbourhood_function(self, sigma, euclidean_distance_matrix):
        euclidean_distance_squared = np.square(euclidean_distance_matrix)
        sigma_factor = -1/(2*np.power(sigma, 2))
        exponent = sigma_factor * euclidean_distance_squared
        neigbourhood_matrix = np.exp(exponent)
        return neigbourhood_matrix

    def update_weights(self, input, eta, neighbourhood_matrix):
        dw = np.zeros(shape=self.weight_matrix.shape)
        size_weight_matrix = len(self.weight_matrix) - 1

        for i in range(0, size_weight_matrix):
            for j in range(0, size_weight_matrix):
                dw[i, j, :] = eta * neighbourhood_matrix[i, j] * (input - self.weight_matrix[i, j, :])

        self.weight_matrix = self.weight_matrix + dw

    def train_network(self, training_set, epochs, eta, sigma, sigma_decay, eta_decay):

        for epoch in range(0, 20):
            print(epoch)
            sigma_decayed = sigma * np.exp(-sigma_decay * epoch)
            eta_decayed = eta * np.exp(-eta_decay * epoch)

            for input in training_set:
                output = self.winning_neuron(input)
                min_index = self.get_minimum_distance_index(output)
                euclidian_matrix = self.calculate_euclidean_distance(min_index)
                neighbourhood_matrix = self.neighbourhood_function(sigma_decayed, euclidian_matrix)
                self.update_weights(input, eta_decayed, neighbourhood_matrix)

    def generate_heat_map(self, inputs):

        heat_map = np.zeros((len(inputs), 2))
        i = 0
        for input in inputs:
            output = self.winning_neuron(input)
            min_index = self.get_minimum_distance_index(output)
            heat_map[i] = min_index
            i = i+1
        return heat_map
```