EECE 6036: Intelligent Systems

Satish Kumar Loganathan

Problem 1: Implementation a 10x10 self-organized feature map to cluster 16 animals according to a set of 13 attributes.

The input of the network is a set of 16 animals described in terms of 13 attributes that characterize the appearance and life style of each animal. Table 1.1 shows the 13-bit **attribute codes** for each animal.

Animal		Dove	Hen	Duck	Goose	Owl	Hawk	Eagle	Fox	Dog	Wolf	Cat	Tiger	Lion	Horse	Zebra	Cow
is	small	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0
	medium	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
	big	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
has	2 legs	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
	4 legs	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	hair	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
	hooves	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
	mane	0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0
	feathers	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
likes to	hunt	0	0	0	0	1	1	1	1	0	1	1	1	1	0	0	0
	run	0	0	0	0	0	0	0	0	1	1	0	1	1	1	1	0
	fly	1	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
	swim	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0

Table 1.1

In addition, each input is associated with a 16-bit **symbol code** that varies only by a single bit with respect to the other symbol codes.

$$x_s^k = [0 \quad 0 \quad . \quad . \quad 0 \quad c \quad 0 \quad . \quad . \quad 0]^T$$
$$k = 1, 2, \ldots, 16$$

The inputs fed to the system are vectors of the form [symbol code attribute code]. For example, a hen is represented as [0 c 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0] ^{T}.

<u>System description:</u>

The self-organizing feature map has been implemented as a 2-dimensional grid of 10x10 neurons whose positions are represented in terms of the coordinates on the grid **(i, j)**. Each neuron is designed to have **sigmoid activation**.

The inputs are fed individually to each neuron. For each input vector, the winning neuron is identified as the neuron whose weight vector lies closest to the input vector.

$$i^* = \arg\min \left\| w_i - x^q \right\|$$

Euclidean distance metric was used to determine the proximity of a neuron from the winning neuron. Based on this information, the weights of the network is updated as follows utilizing a **neighborhood function** that relies on a time-varying **σ** value.

$$\Delta w_{ij} = \eta \Lambda\left(i, i^*, t\right)\left(x_j^q - w_{ij}\right)$$

$$\Lambda\left(i, i^*, t\right) = \exp\left[\frac{-\left\|r_i - r_{i^*}\right\|^2}{2\sigma^2(t)}\right]$$

$$\sigma(t) = \sigma_o \exp\left(-\frac{t}{\tau_N}\right)$$

Choice of parameters:

The other parameters of the network, namely the **learning rate** and **number of epochs** were determined through trial and error approach. Similar to **σ**, the system utilizes an adaptive learning rate that varies over time (every epoch in this implementation).

$$\eta(t) = \eta_0 e^{-t/\tau}$$

The initial learning rate and initial sigma value were set to 1. The exponential function ensures that this value decreases over time. The number of epochs was set to 2000 as in the original experiment.

Results:

After the training is complete, the self-organizing feature map was evaluated against the input with the attribute code set to zero i.e. input vectors of the form [symbol code 0] ^{T}. For each of the 100 neurons, the animal which had the highest response was recorded and Figure 1.1 is a representation of the same on a 2-dimensional grid.

–	–	–	lion	–	–	–	–	tiger	–
–	–	–	–	–	–	–	–	–	–
cow	–	–	–	–	–	zebra	–	–	fox
–	–	–	–	–	–	–	–	–	–
–	–	–	horse	–	–	–	–	dog	–
–	–	–	–	–	–	wolf	–	–	cat
owl	–	–	–	–	–	–	–	–	–
–	–	–	–	hawk	–	–	eagle	–	goose
–	–	–	–	–	–	–	–	–	–
–	hen	–	–	–	–	dove	–	–	duck

Figure 1.1

Analysis of Results:

It can be observed from **Figure 1.1** that the SOFM automatically learnt to cluster the animals on the basis of their attributes. All the birds are clustered around the lower segment of the grid, mammals (cow, horse and zebra) are positioned around the center of the map, and the felines are clustered along the upper-right segment of the map.

This is a logical grouping of the input, given that the animals were evaluated in terms of their size, physical characters and behavioral traits. Animals that are similar, are positioned closer to each other on the grid as the activity of the network begins to model the pattern of the input data.

The clusters are similar to the responses obtained in Haykin's original experiment and vary only in terms of their position on the grid. This is attributed to the choice of initial weights (random assignment as per the current implementation), owing to which the position of the winning neuron for each animal differs from time to time.

Problem 2: Evaluating the map of animals and attributes obtained in the previous problem with new animals described only in terms of their attributes.

<u>Results:</u>

–	–	–	blue whale	–	–	killer whale	–	–	–
–	–	–	–	–	–	–	–	–	–
–	–	–	–	–	–	pig	–	–	goat
–	–	–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–	–	badger
bat	ostrich	–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–	–	–

Figure 2.1

New Animals Presented	Existing Animals with Highest Degree of Similarity
Blue Whale	Duck
Bat	Owl
Killer Whale	Tiger
Ostrich	Hen
Pig	Cow
Goat	Fox
Badger	Cat

Table 2.1

<u>Analysis of Results:</u>

Figure 2.1 shows the spatial ordering produced by the SOFM in response to the new animals. The bat and ostrich were positioned to the lower segment of the grid, where all the birds were seen in the previous result, Fig 1.1. All four-legged animals – pig, goat and badger were also positioned similar to the previous results as well.

However, the new input comprised of mammals that dwell in the sea – blue whale and killer whale, a class which the SOFM is not familiar with. Yet, the SOFM generated outputs for these creatures. To understand the behavior of the system with the new inputs, a proximity analysis was performed, in which the new animals were mapped against their most similar counter-parts (based on attribute values) of the previously seen animals, captured in Table 2.1.

We can observe that the spatial arrangements of the new input patterns generated by the SOFM were majorly based on their similarity with the known patterns. This can be misleading, like in the case of sea mammals being categorized along with felines, and exposes a limitation of the system is classifying novel input patterns that need to be categorized as a new class/cluster.

Appendix – Source code:

HaykinSOFM.py

```
import numpy as np
import random as rand
import Constants as cts
from scipy.spatial.distance import euclidean
import SOFM

weights = {}

# Generating the data for representing 16 animals using attribute code and
symbol code
attribute_code = np.array(
                    np.matrix(  '1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 0;'
                                '0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0;'
                                '0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1;'
                                '1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0;'
                                '0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1;'
                                '0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1;'
                                '0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1;'
                                '0 0 0 1 0 0 0 0 0 1 0 0 1 1 1 0;'
                                '1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0;'
                                '0 0 0 0 1 1 1 1 0 1 1 1 1 0 0 0;'
                                '0 0 0 0 0 0 0 0 1 1 0 1 1 1 1 0;'
                                '1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0;'
                                '0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0').transpose())

symbol_code = []
for idx1 in range(len(cts.ANIMALS)):
    temp = []
    for idx2 in range(len(cts.ANIMALS)):
        if idx2 == idx1:
            temp.append(cts.CONSTANT)
        else:
            temp.append(0)
    symbol_code.append(temp)
```

```
symbol_code = np.array(symbol_code)

# Initializing small random weights for the network
for idx1 in range(cts.MAP_SIZE):
    for idx2 in range(cts.MAP_SIZE):
        temp = []
        for idx3 in range(len(cts.ANIMALS) + cts.NUM_ATTRIBUTES):
            temp.append(rand.random() / 100)
        weights[idx1, idx2] = temp


# Train the SOFM with the existing data
updated_weight_matrix = SOFM.train_sofm(weights, symbol_code, attribute_code)
print "Weights obtained:", updated_weight_matrix

test_attribute_code = np.zeros((len(cts.ANIMALS), 13))

# Test the SOFM model with a zero attribute vector
SOFM.test_sofm(updated_weight_matrix, symbol_code, test_attribute_code, 'test')

# Testing the SOFM model with new input and zero symbol_code
new_symbol_codes = np.zeros((len(cts.NEW_ANIMALS), 16))

new_attribute_codes = np.array(np.matrix(  '0 1 0 0 1 1 1 0 0 0 0 0 0;'
                                           '0 0 1 0 1 1 1 0 0 0 0 0 0;'
                                           '1 0 0 0 1 1 0 0 0 1 0 0 0;'
                                           '0 0 1 1 0 0 0 0 1 0 1 0 0;'
                                           '1 0 0 1 0 1 0 0 0 1 0 1 0;'
                                           '0 0 1 0 0 0 0 0 0 0 0 0 1;'
                                           '0 0 1 0 0 0 0 0 0 1 0 0 1'))

SOFM.test_sofm(updated_weight_matrix, new_symbol_codes, new_attribute_codes,
'new')

max_similarity = {}
for idx1 in range(len(cts.NEW_ANIMALS)):
    min_idx = 0
    min_dist = euclidean(new_attribute_codes[idx1], attribute_code[0])
    for idx2 in range(1, len(cts.ANIMALS)):
        dist = euclidean(new_attribute_codes[idx1], attribute_code[idx2])
        if dist < min_dist:
            min_dist = dist
            min_idx = idx2
    max_similarity[cts.NEW_ANIMALS[idx1]] = cts.ANIMALS[min_idx]
print "Similarity of New Input to existing Input:\n", max_similarity
```

SOFM.py

```
from __future__ import division
import numpy as np
import Constants as cts
from scipy.spatial.distance import euclidean
import math

sigma_0 = 1
learning_rate_0 = 1
num_epochs = 2000


# Time varying learning rate
def get_learning_rate(time):
```

```
    return learning_rate_0 * math.exp((-1 * time) / 100)


# Time varying sigma value
def get_sigma_val(time):
    return sigma_0 * math.exp((-1 * time)/1000)


# Neighborhood function of the SOFM
def get_neighborhood_value(neuron_idx, winning_neuron_idx, time):
    neighborhood = math.exp((-1 * euclidean(neuron_idx, winning_neuron_idx) **
2)/get_sigma_val(time))
    return neighborhood


# Function to compute the change in weight at every point in time
def compute_weight_change(neuron_idx, winning_neuron_idx, weight_vector,
input_vector, time):
    dist_vector = input_vector - weight_vector
    learning_rate = get_learning_rate(time)
    neighborhood = get_neighborhood_value(neuron_idx, winning_neuron_idx, time)
    weight_updates_vector = learning_rate * neighborhood * dist_vector
    return weight_updates_vector


# Function to train the SOFM
def train_sofm(weights, symbol_code, attribute_code):
    print 'Training SOFM for classification'
    for epoch in range(num_epochs):
        print 'Epoch - ', epoch+1
        for data_idx in range(len(cts.ANIMALS)):
            input_vector = np.concatenate((symbol_code[data_idx],
attribute_code[data_idx]))
            winning_neuron_idx = (0, 0)
            min_dist = euclidean(input_vector, weights[0, 0])
            # Identify the winning neuron
            for neuron_idx1 in range(cts.MAP_SIZE):
                for neuron_idx2 in range(cts.MAP_SIZE):
                    if (neuron_idx1, neuron_idx2) == (0, 0):
                        continue
                    dist = euclidean(weights[neuron_idx1, neuron_idx2],
input_vector)
                    if dist < min_dist:
                        min_dist = dist
                        winning_neuron_idx = (neuron_idx1, neuron_idx2)
            # Update weights
            for neuron_idx1 in range(cts.MAP_SIZE):
                for neuron_idx2 in range(cts.MAP_SIZE):
                    weight_vector = weights[neuron_idx1, neuron_idx2]
                    weights[neuron_idx1, neuron_idx2] = weight_vector +
compute_weight_change(
                        (neuron_idx1, neuron_idx2), winning_neuron_idx,
weight_vector, input_vector, epoch+1)
    return weights


def test_sofm(weights, symbol_code, attribute_code, flag):
    print 'Generating output of the network for %s data...' % flag
    test_outcome = {}
    if flag == 'test':
        output_dict = cts.ANIMALS
    else:
        output_dict = cts.NEW_ANIMALS
```

```
    for data_idx in range(len(output_dict)):
        input_vector = np.concatenate((symbol_code[data_idx],
attribute_code[data_idx]))
        winning_neuron_idx = (0, 0)
        max_output = np.dot(weights[0, 0].transpose(), input_vector)
        # Identify the winning neuron
        for neuron_idx1 in range(cts.MAP_SIZE):
            for neuron_idx2 in range(cts.MAP_SIZE):
                if (neuron_idx1, neuron_idx2) == (0, 0):
                    continue
                output = np.dot(weights[neuron_idx1, neuron_idx2].transpose(),
input_vector)
                if output > max_output:
                    max_output = output
                    winning_neuron_idx = (neuron_idx1, neuron_idx2)
        test_outcome[winning_neuron_idx] = output_dict[data_idx]

    print 'Output of the network:'
    for neuron_idx1 in range(cts.MAP_SIZE):
        output_line = ''
        for neuron_idx2 in range(cts.MAP_SIZE):
            neuron_idx = (neuron_idx1, neuron_idx2)
            if neuron_idx in test_outcome.keys():
                output_line += '\t' + '{:15s}'.format(test_outcome[neuron_idx])
            else:
                output_line += '\t' + '{:15s}'.format(' - ')
        print output_line
        print ''
```

Constants.py

```
# Constant parameters
NUM_ATTRIBUTES = 13
MAP_SIZE = 10
CONSTANT = 1

ANIMALS = {0: 'dove', 1: 'hen', 2: 'duck', 3: 'goose',
           4: 'owl', 5: 'hawk', 6: 'eagle', 7: 'fox',
           8: 'dog', 9: 'wolf', 10: 'cat', 11:  'tiger',
           12: 'lion', 13: 'horse', 14: 'zebra', 15: 'cow'}

NEW_ANIMALS = {0: 'goat', 1: 'pig', 2: 'badger',
               3: 'ostrich', 4: 'bat', 5: 'blue whale',
               6: 'killer whale'}
```