Self Organizing Map

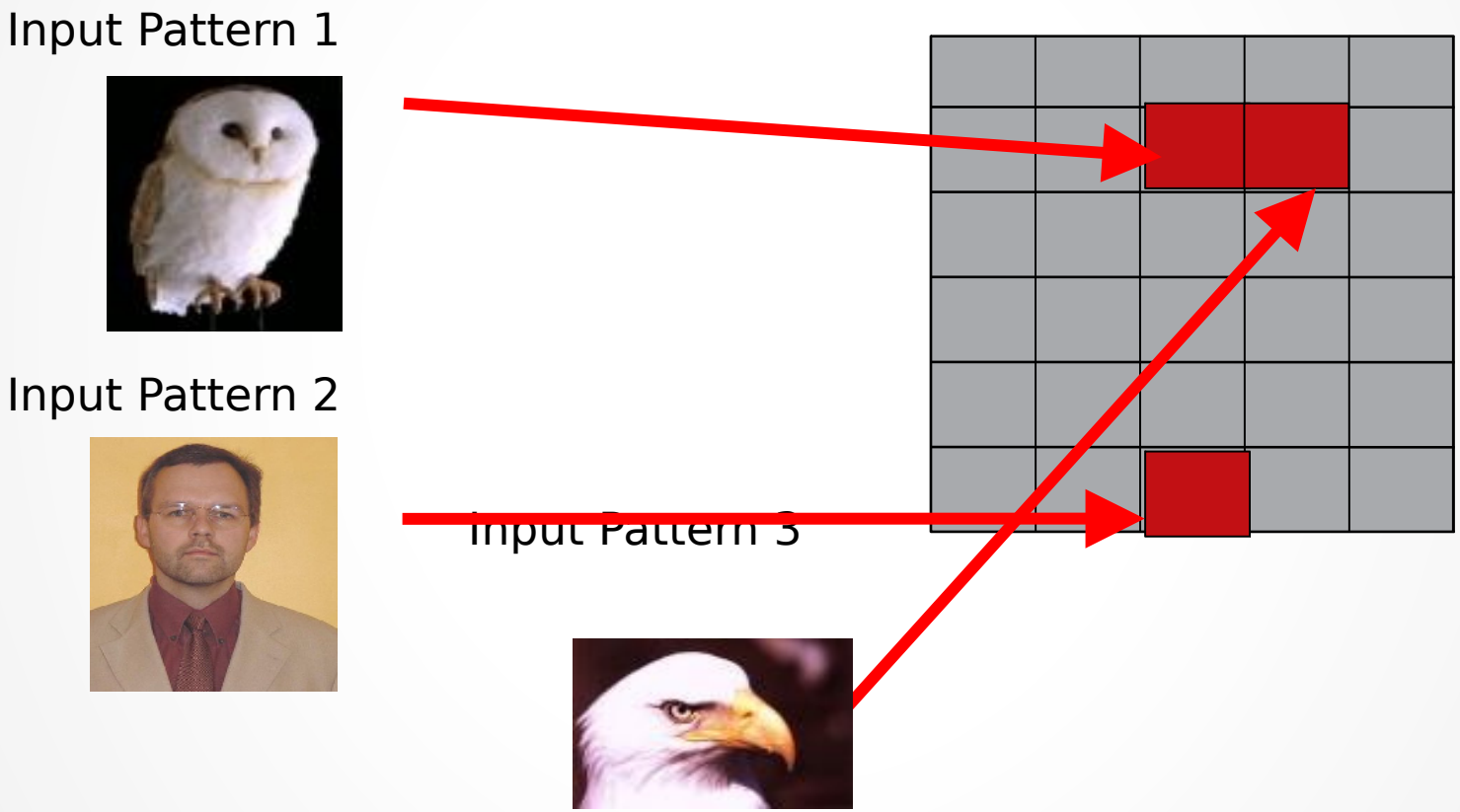
Jony Sugianto
jony@evolvemachinelearners.com
0812-13086659
github.com/jonysugianto

Self-Organizing Maps : Origins

- Ideas first introduced by C. von der Malsburg (1973), developed and refined by T. Kohonen (1982)
- Neural network algorithm using unsupervised competitive learning
- Primarily used for organization and visualization of complex data
- Biological basis: ‘brain maps’

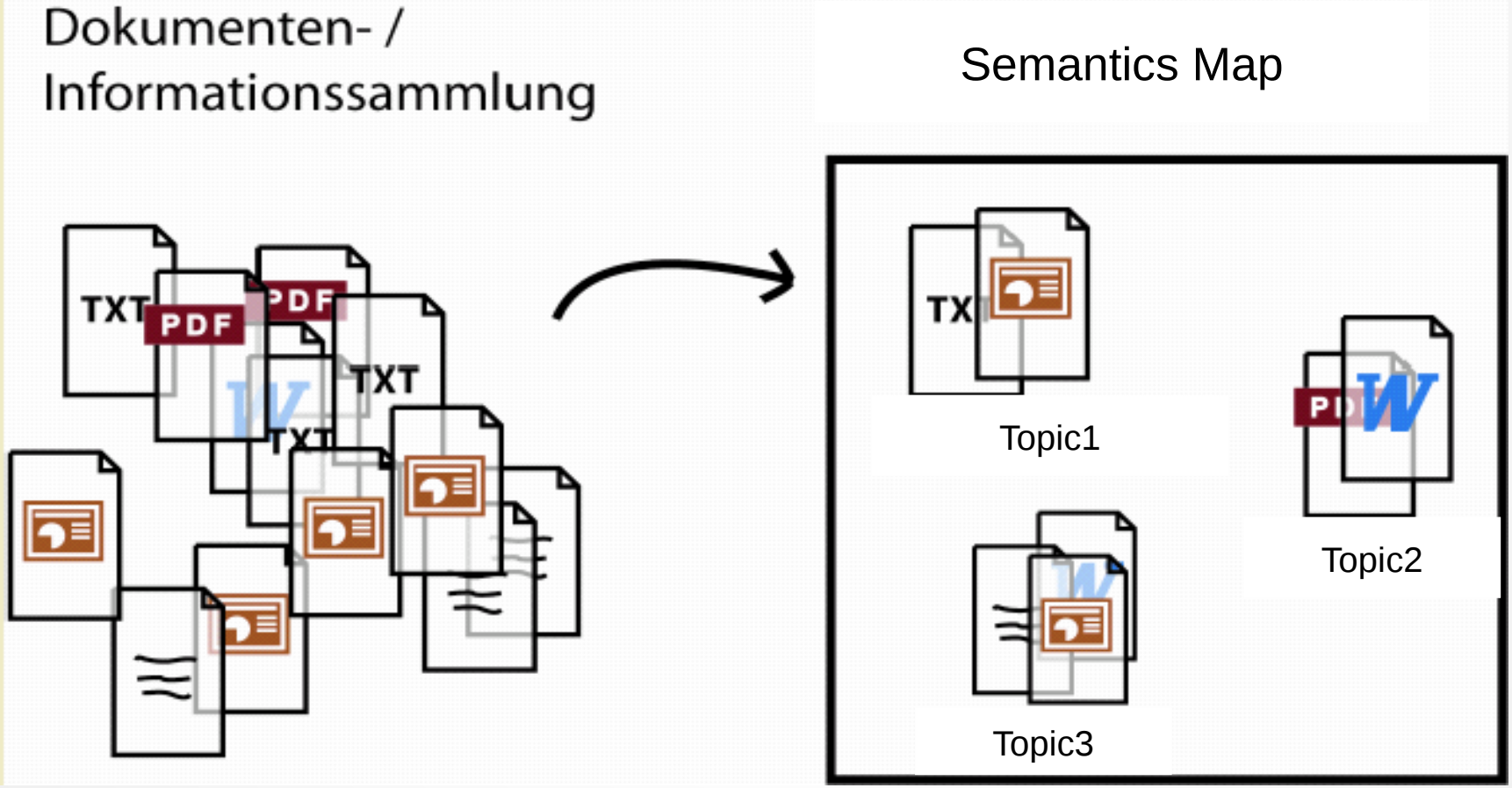
Motivation: The Idea

Computer know automatically information classification and put them together



Motivation: The Idea

Text objects must be automatically produced with semantics relationships



What are self organizing maps?

- Are named “Self-Organizing” because no supervision is required.
- SOMs learn on their own through unsupervised competitive learning.
- They attempt to map their weights to conform to the given input data.

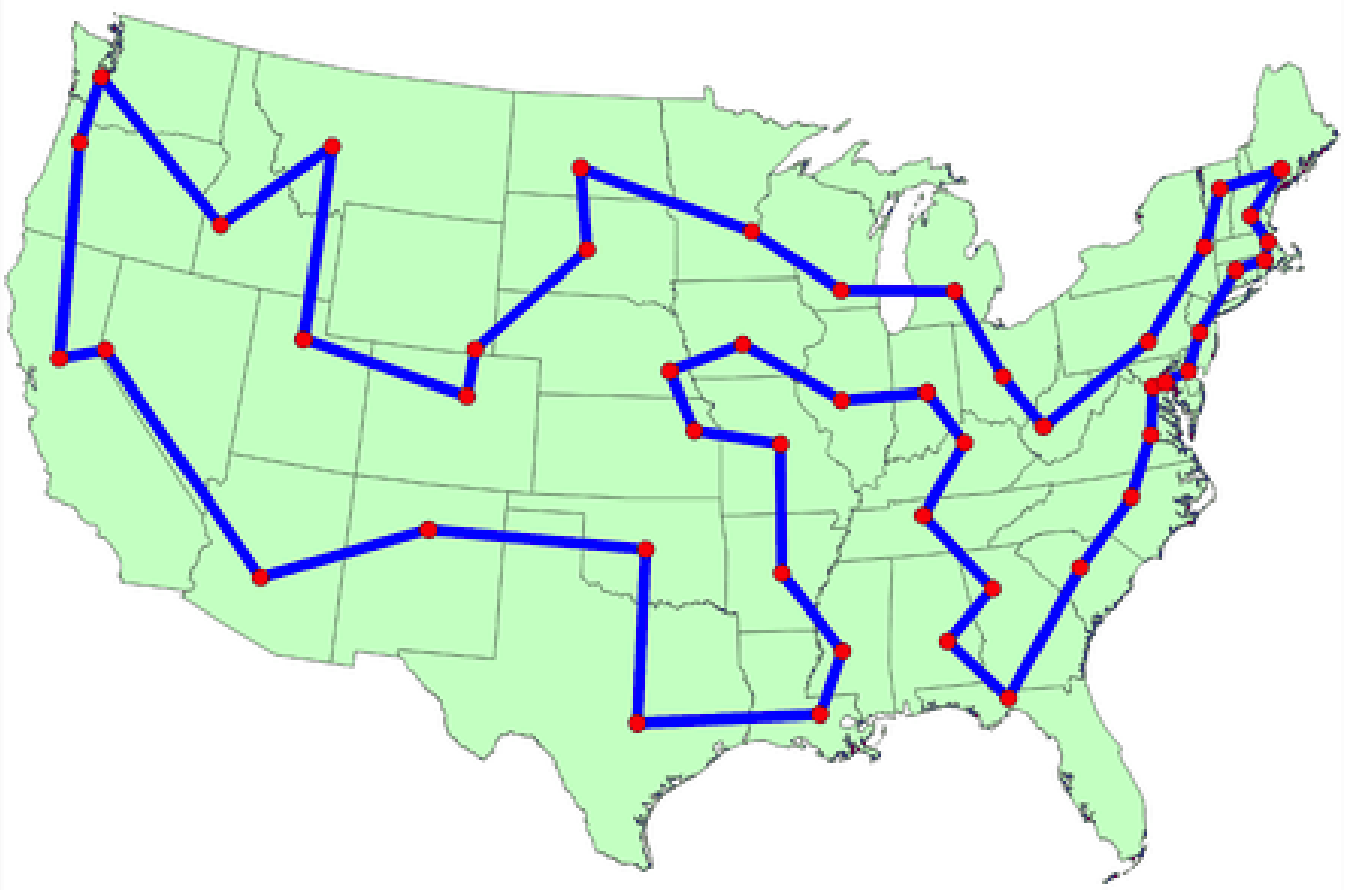
What are self organizing maps?

- *Thus SOMs are neural networks that employ unsupervised learning methods, mapping their weights to conform to the given input data with a goal of representing multidimensional data in an easier and understandable form for the human eye.* (pragmatic value of representing complex data)

What are self organizing maps?

- Training a SOM requires no target vector. A SOM learns to classify the training data without any external supervision.

Traveling Salesman Problem

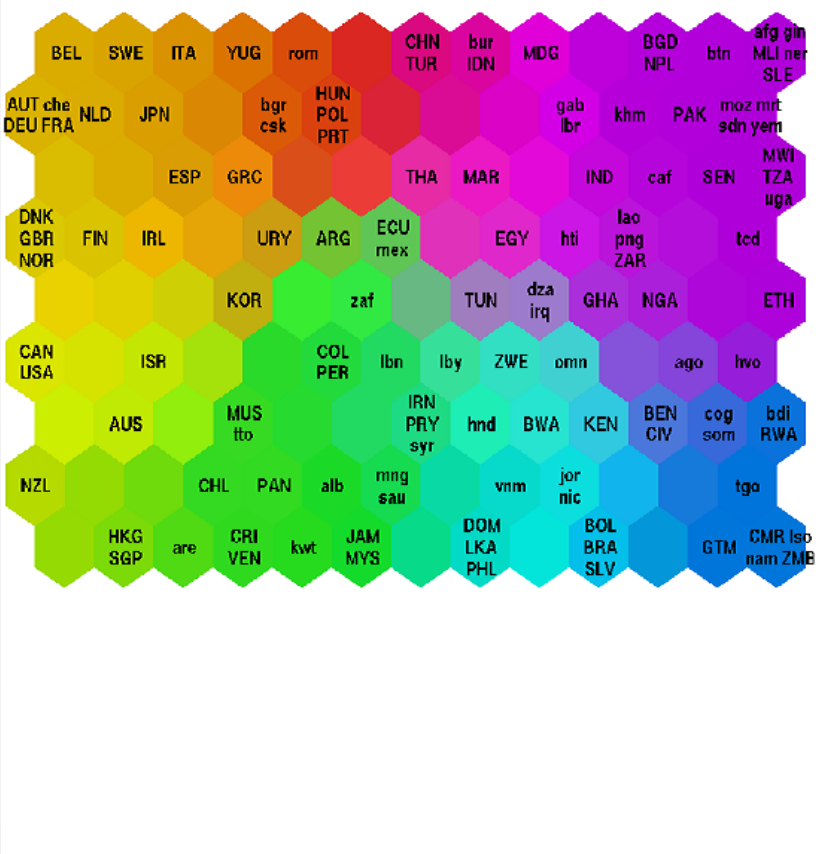


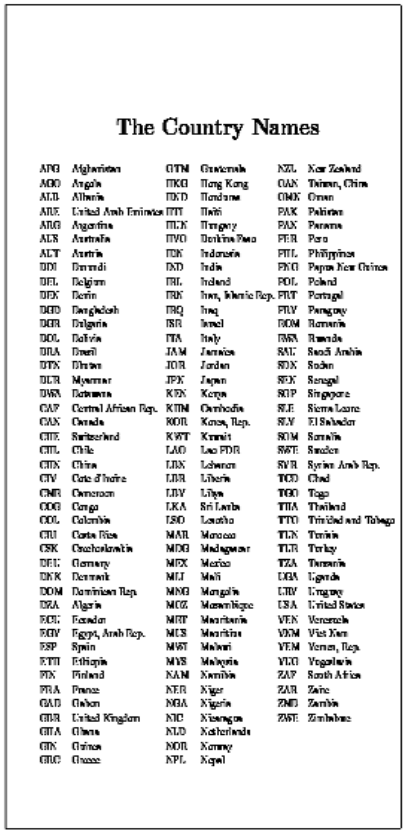
Self-Organizing Maps

SOM – Result Example

Classifying World Poverty

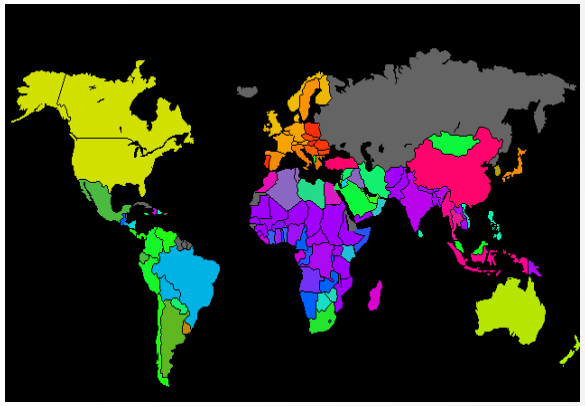
Helsinki University of Technology





The Country Names

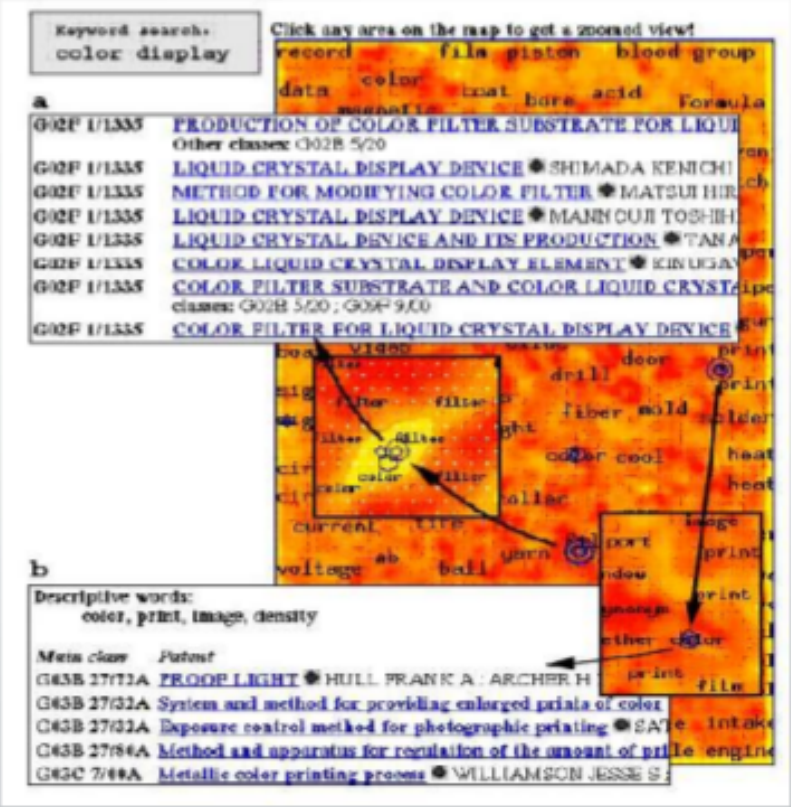
AFG	Afghanistan	GTM	Guatemala	NZL	New Zealand
AGO	Angola	HKG	Hong Kong	OAN	Taiwan, China
ALB	Albania	HND	Honduras	OMN	Oman
ARE	United Arab Emirates	HTI	Haiti	PAK	Pakistan
ARG	Argentina	HUN	Hungary	PAN	Panama
AUS	Australia	IVO	Ivory Coast	PER	Peru
AUT	Austria	IDN	Indonesia	PHL	Philippines
BDI	Burundi	IND	India	PNG	Papua New Guinea
BEL	Belgium	IRL	Ireland	POL	Poland
BEN	Benin	IRN	Iran, Islamic Rep.	PRT	Portugal
BGD	Bangladesh	IRQ	Iraq	PRY	Paraguay
BGR	Bulgaria	ISR	Israel	ROM	Romania
BOL	Bolivia	ITA	Italy	RWA	Rwanda
BRA	Brazil	JAM	Jamaica	SAU	Saudi Arabia
BTN	Bhutan	JOR	Jordan	SDN	Sudan
BUR	Myanmar	JPN	Japan	SEN	Senegal
BWA	Botswana	KEN	Kenya	SGP	Singapore
CAF	Central African Rep.	KHM	Cambodia	SLE	Sierra Leone
CAN	Canada	KOR	Korea, Rep.	SLV	El Salvador
CHE	Switzerland	KWT	Kuwait	SOM	Somalia
CHL	Chile	LAO	Lao PDR	SWE	Sweden
CHN	China	LBN	Lebanon	SYR	Syrian Arab Rep.
CIV	Cote d'Ivoire	LBR	Liberia	TCD	Chad
CMR	Cameroon	LBY	Libya	TGO	Togo
COG	Congo	LKA	Sri Lanka	THA	Thailand
COL	Colombia	LSO	Lesotho	TTO	Trinidad and Tobago
CRI	Costa Rica	MAR	Morocco	TUN	Tunisia
CSK	Czechoslovakia	MDG	Madagascar	TUR	Turkey
DEU	Germany	MEX	Mexico	TZA	Tanzania
DNK	Denmark	MLI	Mali	UGA	Uganda
DOM	Dominican Rep.	MNG	Mongolia	URY	Uruguay
DZA	Algeria	MOZ	Mozambique	USA	United States
ECU	Ecuador	MRT	Mauritania	VEN	Venezuela
EGY	Egypt, Arab Rep.	MUS	Mauritius	VNM	Viet Nam
ESP	Spain	MWI	Malawi	YEM	Yemen, Rep.
ETH	Ethiopia	MYS	Malaysia	YUG	Yugoslavia
FIN	Finland	NAM	Namibia	ZAF	South Africa
FRA	France	NER	Niger	ZAR	Zaire
GAB	Gabon	NGA	Nigeria	ZMB	Zambia
GBR	United Kingdom	NIC	Nicaragua	ZWE	Zimbabwe
GHA	Ghana	NLD	Netherlands		
GIN	Guinea	NOR	Norway		
GRC	Greece	NPL	Nepal		



'Poverty map' based on 39 indicators from World Bank statistics (1992)

WEBSOM

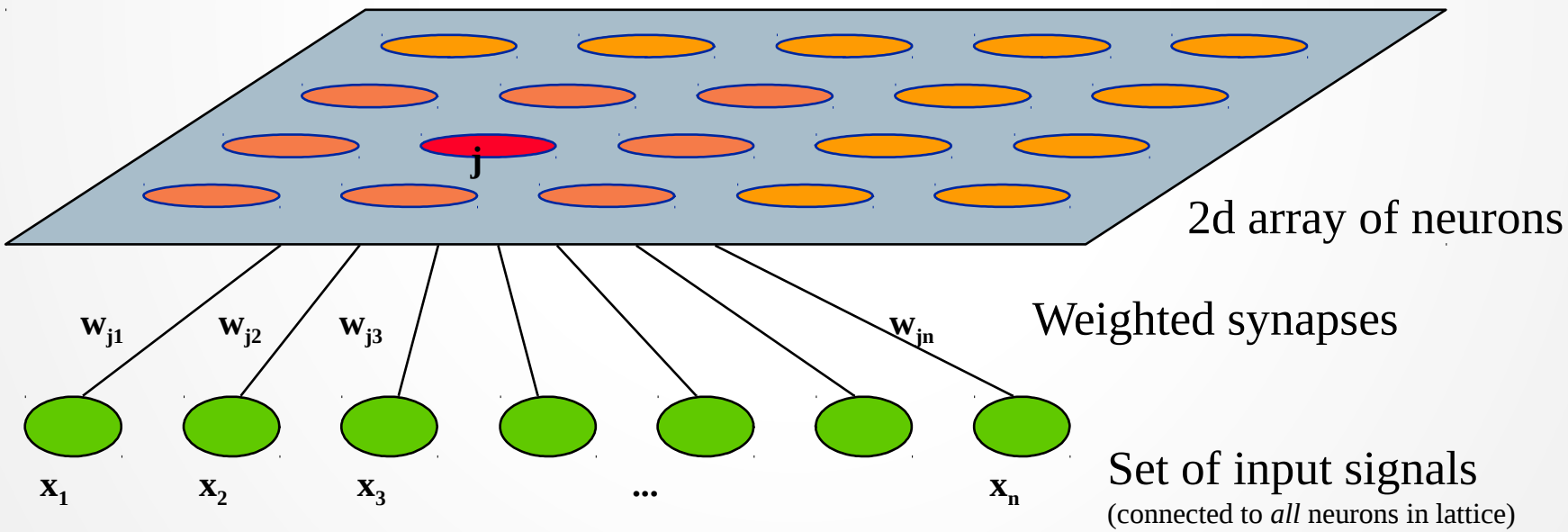
- WEBSOM: Organization of a Massive Document Collection



Self-Organizing Maps

SOM - Architecture

- Lattice of neurons ('nodes') accepts and responds to set of input signals
- Responses compared; 'winning' neuron selected from lattice
- Selected neuron activated together with 'neighbourhood' neurons
- Adaptive process changes weights to more closely resemble inputs



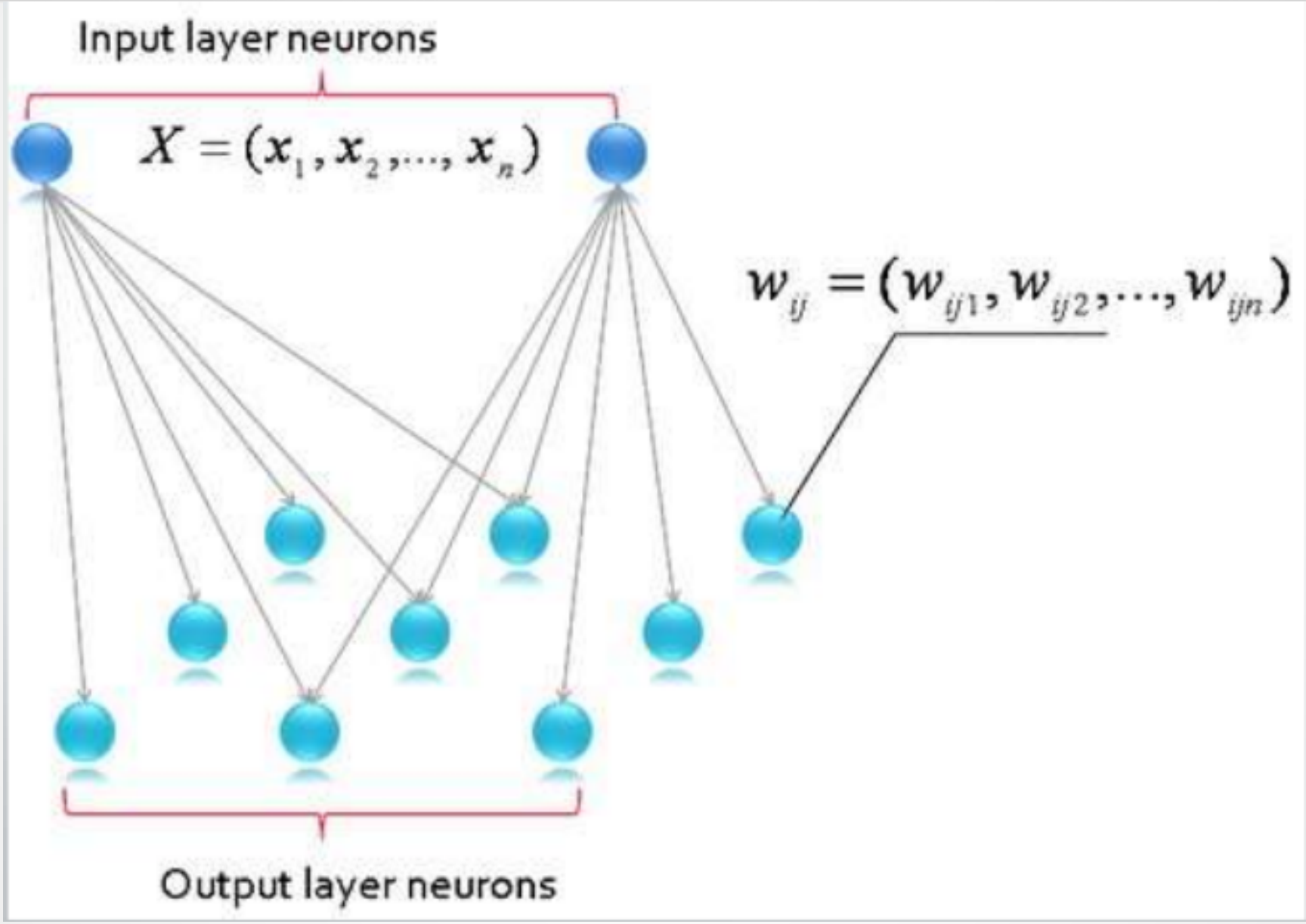
The Architecture

- There are no interconnections among the computational nodes.
- The number of input nodes is determined by the dimensions of the input vector.

Representing Data

- Weight vectors are of the same dimension as the
- input vectors. If the training data consists of
- vectors, V, of n dimensions:
  - $V_1, V_2, V_3 \ldots V_n$
  –
- Then each node will contain a corresponding weight vector W, of n dimensions:
  - $W_1, W_2, W_3 \ldots W_n$

A sample SOM


Input layer neurons
$X = (x_1, x_2, \ldots, x_n)$
$w_{ij} = (w_{ij1}, w_{ij2}, \ldots, w_{ijn})$
Output layer neurons

Terms used in SOMs

- ***vector quantization*** - This is a data compression technique. SOMs provide a way of representing multidimensional data in a much lower dimensional space; typically one or two dimensions

Terms used in SOMs...

- ***Neighbourhood***
- ***Output space***
- ***Input space***

Learning Algorithm

- Initialize weights
  - Random
  - Pregenerated
- Iterate through inputs
- For each input, find the “winning” neuron
  - Euclidean distance(Distance measurement)
- Adjust “winning” neuron and its neighbors
  - Gaussian
  - Mexican Hat

EXPLANATION:
How Kohonen SOMs work The SOM Algorithm

- The Self-Organizing Map algorithm can be broken up into 6 steps
- **1). Each node's weights are initialized.**
- **2). A vector is chosen at random from the set of training data and presented to the network.**

EXPLANATION: The SOM Algorithm...

- **3). Every node in the network is examined to calculate which ones' weights are most like the input vector. The winning node is commonly known as the *Best Matching Unit (BMU)*.**

EXPLANATION: The SOM Algorithm...

- **4). The radius of the neighbourhood of the BMU is calculated. This value starts large. Typically it is set to be the radius of the network, diminishing each time-step.**

EXPLANATION: The SOM Algorithm...

- **5). Any nodes found within the radius of the BMU, calculated in 4), are adjusted to make them more like the input vector. The closer a node is to the BMU, the more its' weights are altered**
- **6). Repeat 2) for N iterations.**

Find the Best Matching Unit (BMU)

- **Computing the similarity between Input-features with the Node-Weights using Distance Measurement**

The Winning Node

- **Node j** becomes the winning node since it has the lowest score.
- This implies that its weight vector values are similar to the input values of the presented instance.
- i.e. The value of **node j** is closest to the input vector.
- As a result, the weight vectors associated with the winning node are adjusted so as to reward the node for winning the instance.

Common Distance measures:

- *Distance measure* will determine how the *similarity* of two elements is calculated and it will influence the shape of the clusters.
  They include:

1. The Euclidean distance (also called 2-norm distance) is given by:

$$d(x, y) = \sqrt[2]{\sum_{i=1}^{p} |x_i - y_i|^2}$$

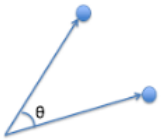
2. The Manhattan distance (also called taxicab norm or 1-norm) is given by:

$$d(x, y) = \sum_{i=1}^{p} |x_i - y_i|$$

Common Distance measures:

3. Cosinus Similarity: The angle between two vectors can be used as a distance measure when clustering high dimensional data

$$sim(A,B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$

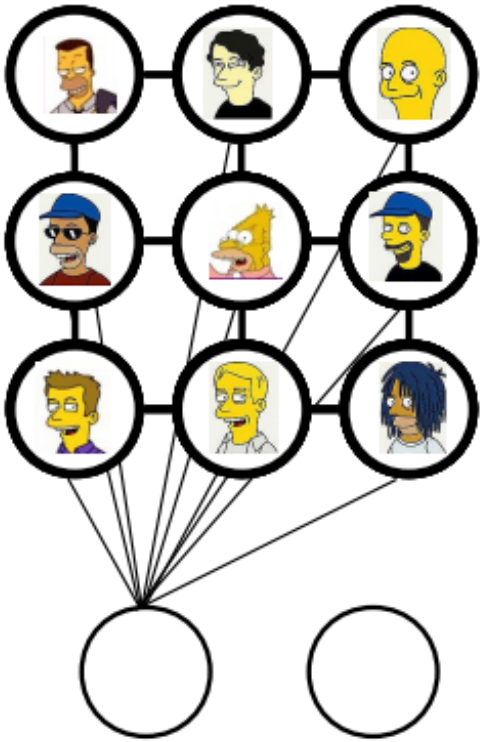


4. Hamming distance (sometimes edit distance) measures the minimum number of substitutions required to change one member into another.

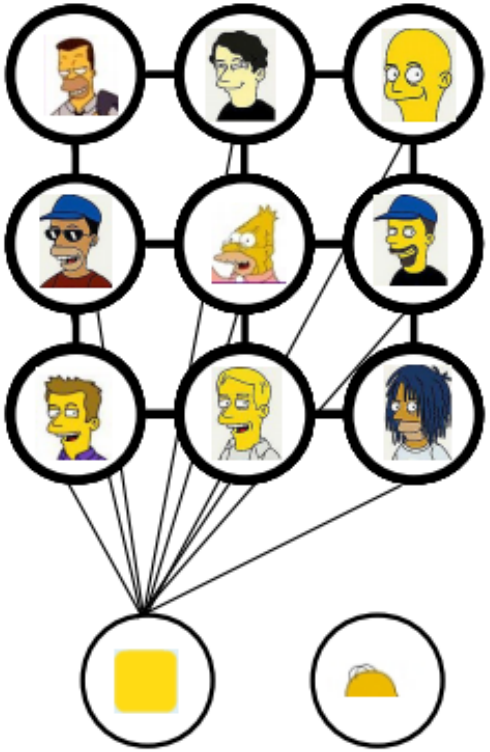
The algorithm

The weights of the neurons are initialized

t = 0



The algorithm



Example

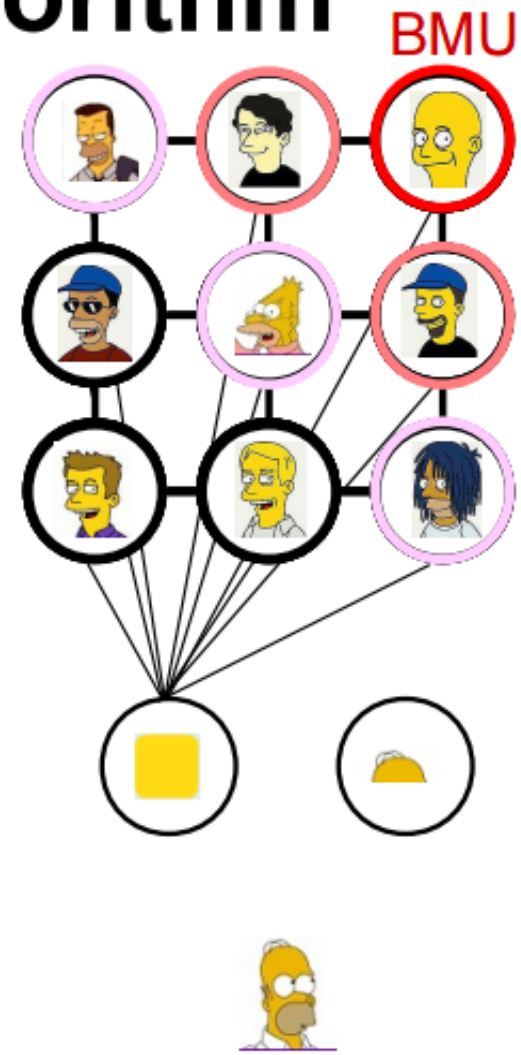
The algorithm

The training utilizes competitive learning.

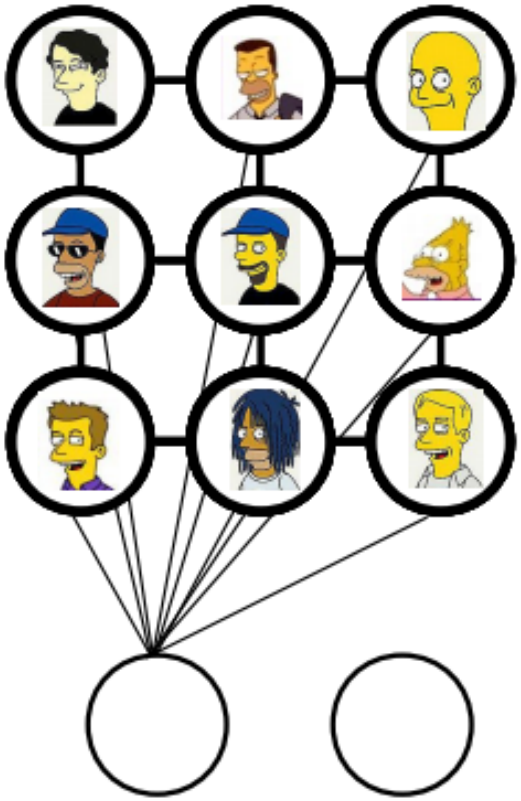
The neuron with weight vector most similar to the input is called the best matching unit (BMU).

The weights of the BMU and neurons close to it in the SOM lattice are adjusted towards the input vector.

The magnitude of the change decreases with time and with distance from the BMU.

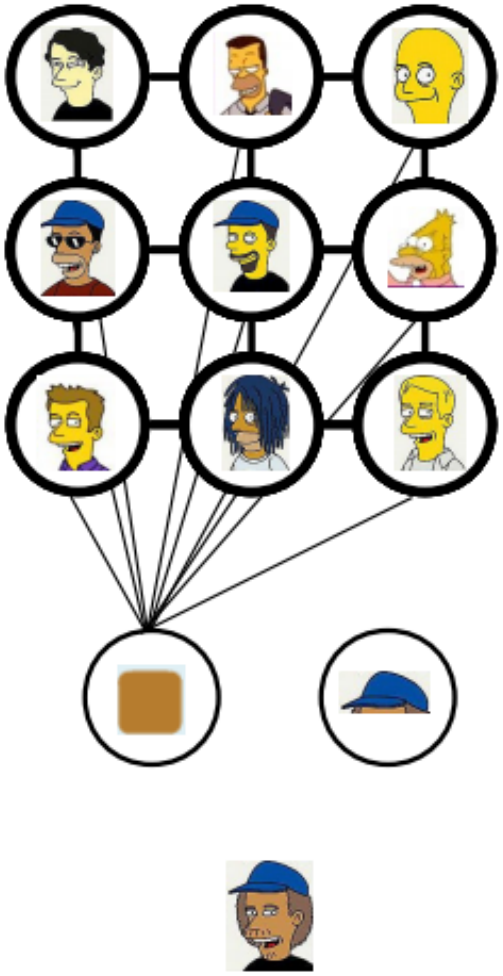


The algorithm

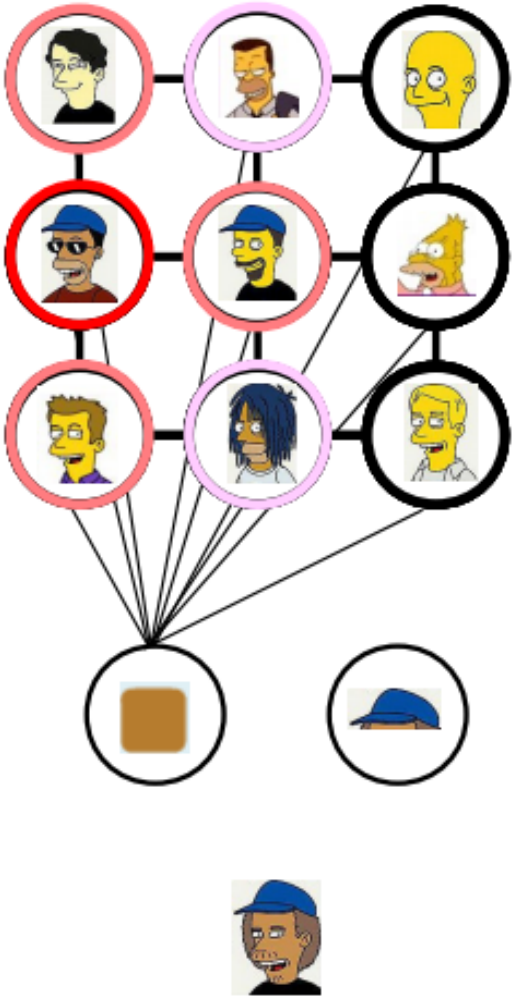


Next example

The algorithm



The algorithm



The Algorithm

- The unit closet to the input vector is call Best Matching Unit (BMU).
- The BMU and other units will adjust its position toward the input vector.
- The update formula is

$$Wv(t + 1) = Wv(t) + \Theta (v, t)\, \alpha(t)(D(t) - Wv(t))$$

The Algorithm

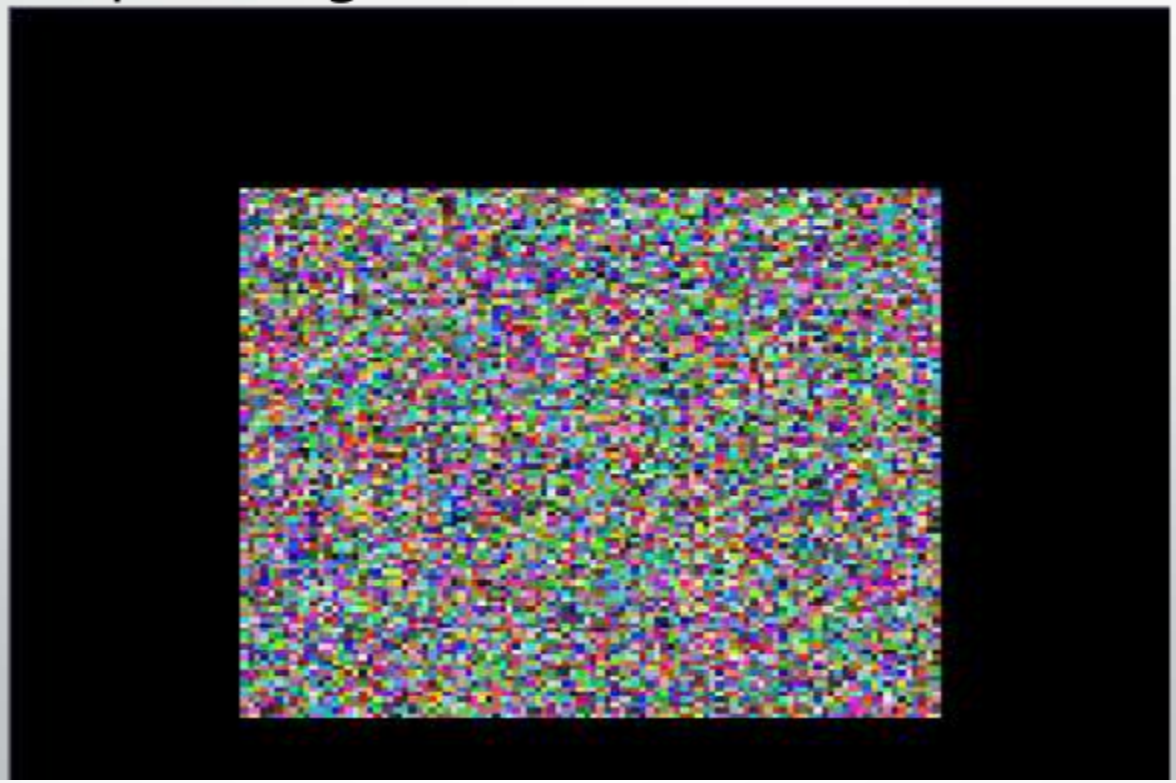
$$Wv(t + 1) = Wv(t) + \Theta(v, t)\,\alpha(t)(D(t) - Wv(t))$$

- **Wv**(t) = weight vector
- α(t) = monotonically decreasing learning coefficient
- **D**(t) = the input vector
- Θ (v, t) = neighborhood function

This process is repeated for each input vector for a (usually large) number of cycles **λ**.

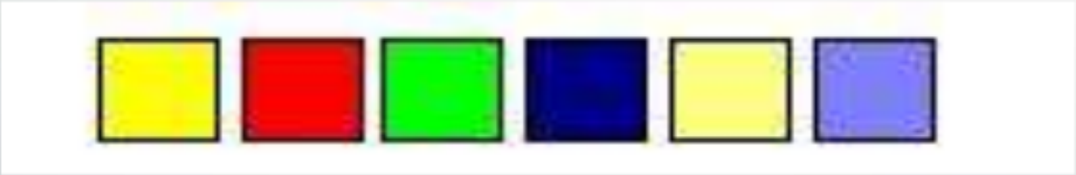
ILLUSTRATION: A Color Classifier

- Problem: Group and represent the primary colors and their corresponding shades on a two dimensional plane.



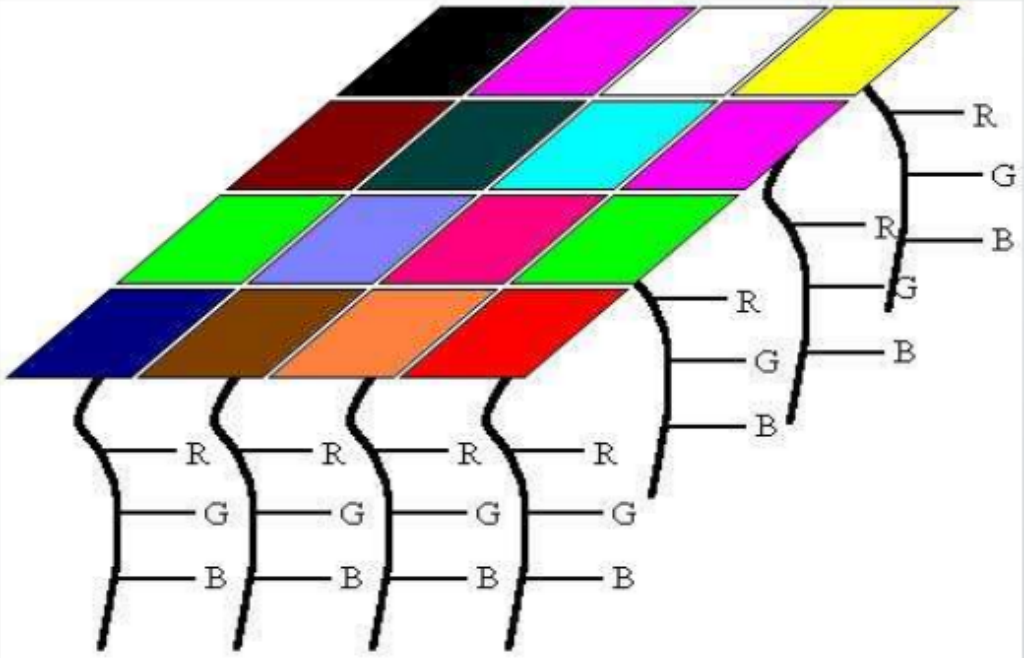
A Color classifier: Sample Data

- The colors are represented in their RGB values to form 3-dimensional vectors.

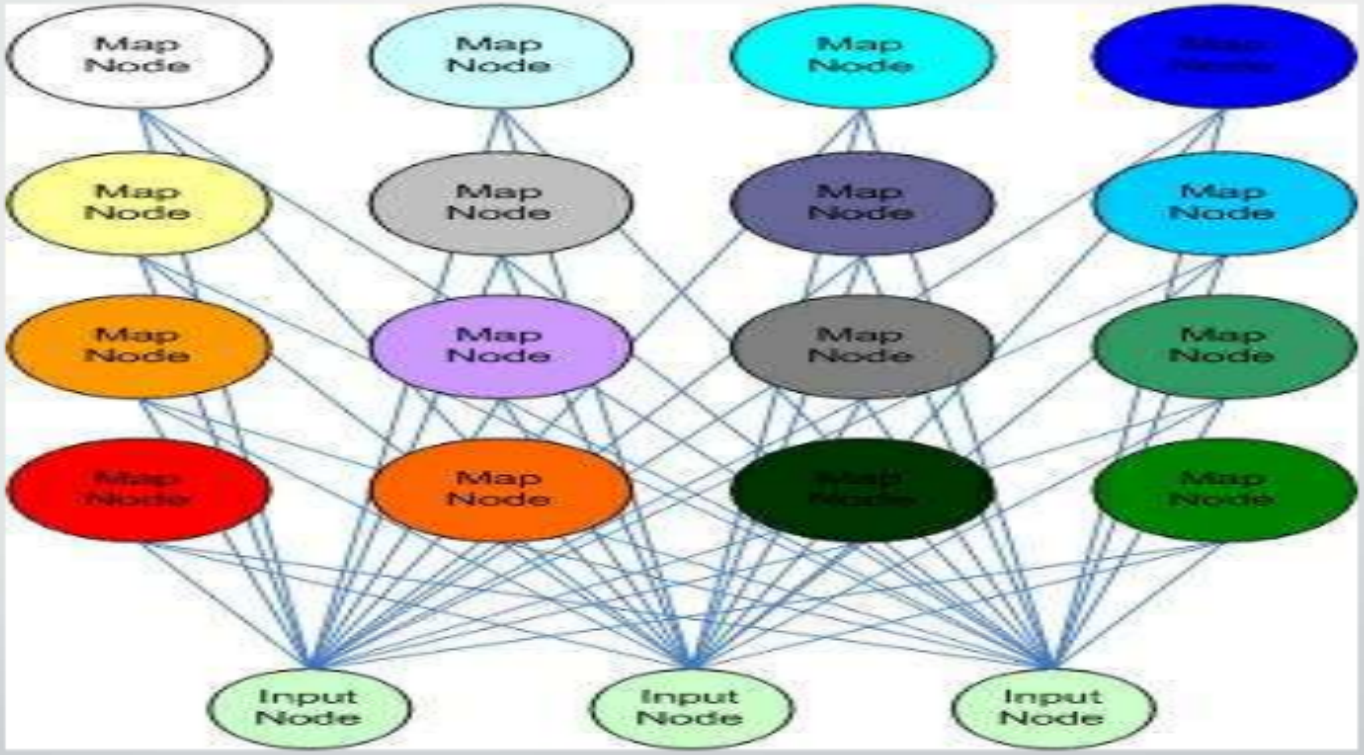


A Color classifier: **Node Weighting**

- Each node is characterized by:
  - Data of the same dimensions as the sample vectors
  - An X, Y position



A Color classifier: The Layout



A Color classifier: **Getting a winner**

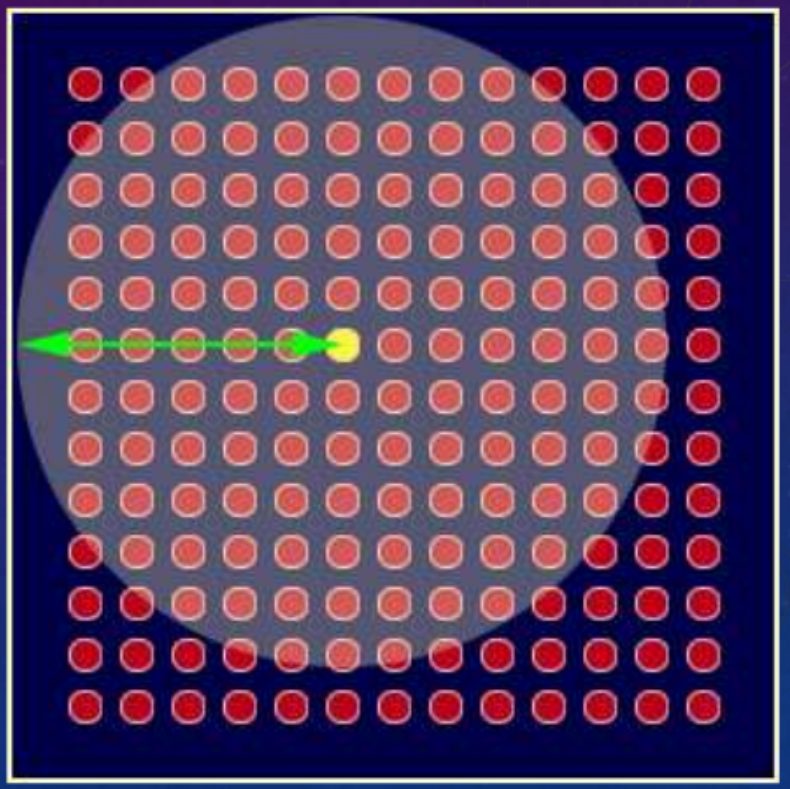
- Go through all the weight vectors and calculate the Euclidean distance from each weight to the chosen sample vector
- Assume the RGB values are represented by the values 0 – 6 depending on their intensity.
  - i.e. Red = (6, 0, 0); Green = (0, 6, 0); Blue = (0, 0, 6)

A Color classifier: **Getting a winner…**

- If we have colour green as the sample input instance, a probable node representing the colour light green (3,6,3) will be closer to green than red.
- Light green = Sqrt((3-0)^2+(6-6)^2+(3-0)^2) = 4.24
  Red = Sqrt((6-0)^2+(0-6)^2+(0-0)^2) = 8.49

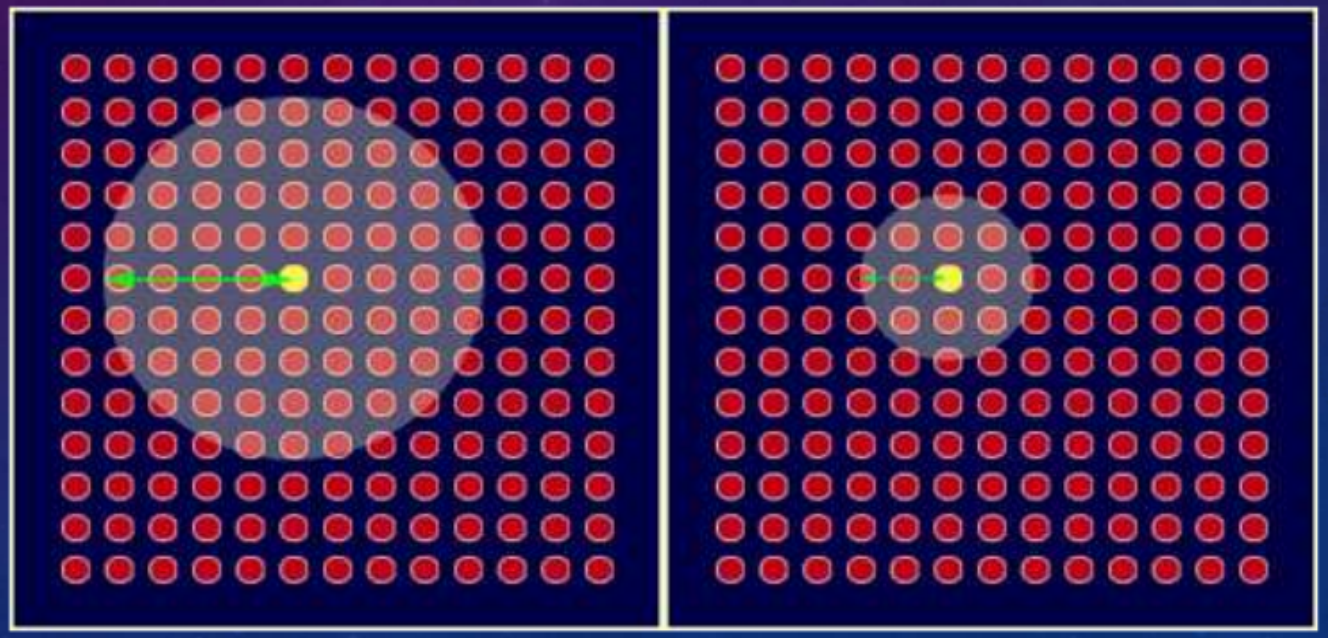
A COLOR CLASSIFIER: DETERMINING THE NEIGHBORHOOD

- Since a node has an X – Y position, it's neighbors can be easily determined based on their radial distance from the BMU coordinates.



A COLOR CLASSIFIER: **DETERMINING THE NEIGHBORHOOD…**

- The area of the neighbourhood shrinks over time with each iteration.



A Color classifier: **Learning**

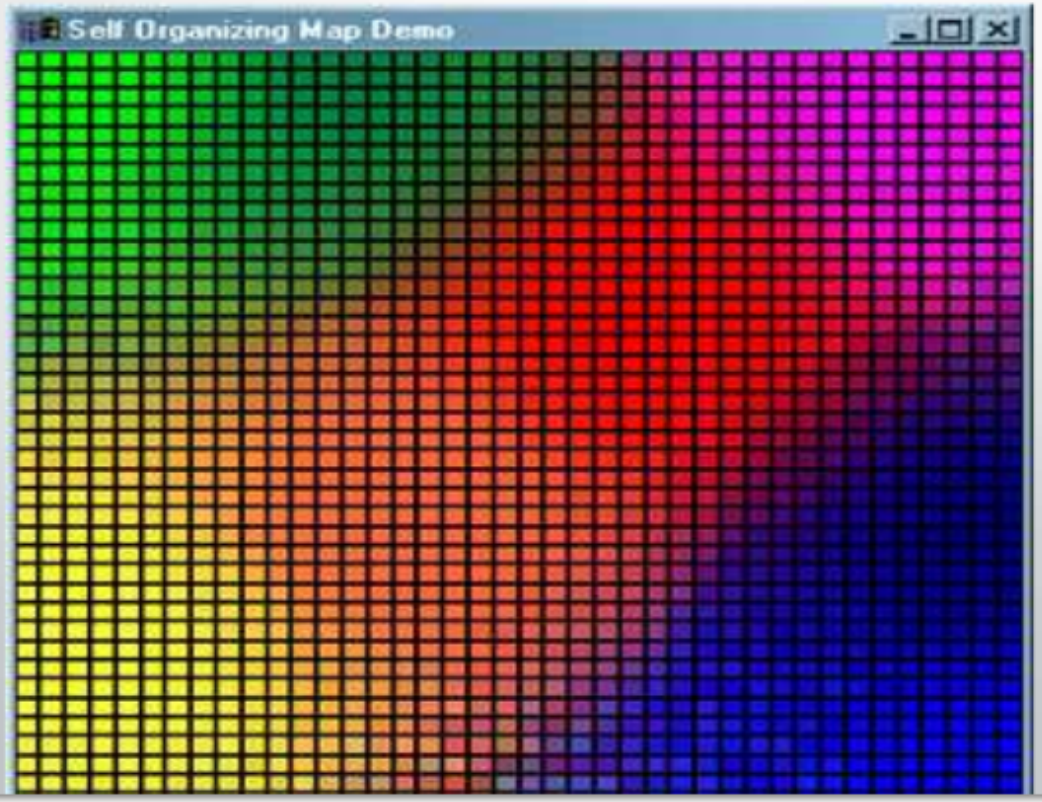
- Every node within the BMU's neighbourhood (including the BMU) has its weight vector adjusted according to a pre-determined equation
- The learning rate is decayed over time.

A Color classifier: **Learning**

- The effect of learning should be proportional to the distance a node is from the BMU.
- A Gaussian function can be used to achieve this, whereby the closest neighbors are adjusted the most to be more like the input vector.

A Color classifier: **Sample output**

- Classification on a 40 x 40 SOM



Example: Clustering Animals

- The animals should be ordered by a neural networks.
- And the animals will be described with their attributes(size, living space).
- e.g. Mouse = (0/0)
- Size: Living space:
  small=0 medium=1 big=2 Land=0 Water=1 Air=2

	Mouse	**Lion**	**Horse**	**Shark**	**Dove**
Size	small	medium	big	big	small
Living space	Land	Land	Land	Water	Air
	(0/0)	(1/0)	(2/0)	(2/1)	(0/2)

Example: Clustering Animals

After the fields of map will be initialized with random values, animals will be ordered in the most similar fields. If the mapping is ambiguous, anyone of fields will be seleced.

(0/0) Mouse (0/0), Lion (1/0)	(0/2) Dove (0/2)	(2/2)
(2/1) Shark (2/1)	(0/0)	(2/0) Horse (2/0)
(1/1)	(1/1)	(0/0)

Example: Clustering Animals

- This training will be done in every field. After the network had been trained, animals will be ordered in the similarest field again.

(1/0.75) Lion	(0.25/1) Dove	(1.5/1.5)
(1.25/0.5)	(1/0.75)	(2/0) Horse
(1.25/1) Shark	(1/1)	(0.5/0) Mouse

Example: Clustering Animals

- This training will be very often repeated. In the best case the animals should be at close quarters ordered by similarest attribute.

(0.75/0.6875)	(0.1875/1.25) Dove	(1.125/1.625)
(1.375/0.5)	(1/0.875)	(1.5/0) Hourse
(1.625/1) Shark	(1/0.75) Lion	(0.75/0) Mouse

Land animals

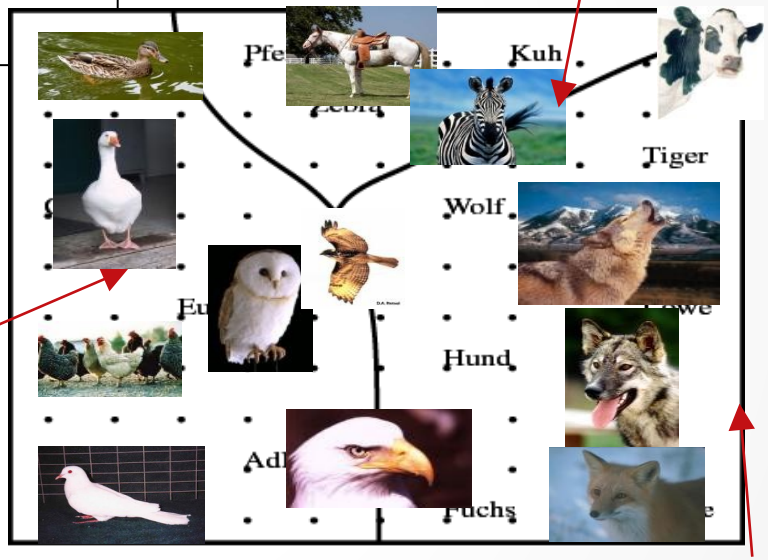
Example: Clustering Animals

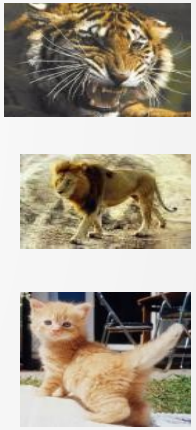
Animal names and their attributes

		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

A grouping according to similarity has emerged

peaceful



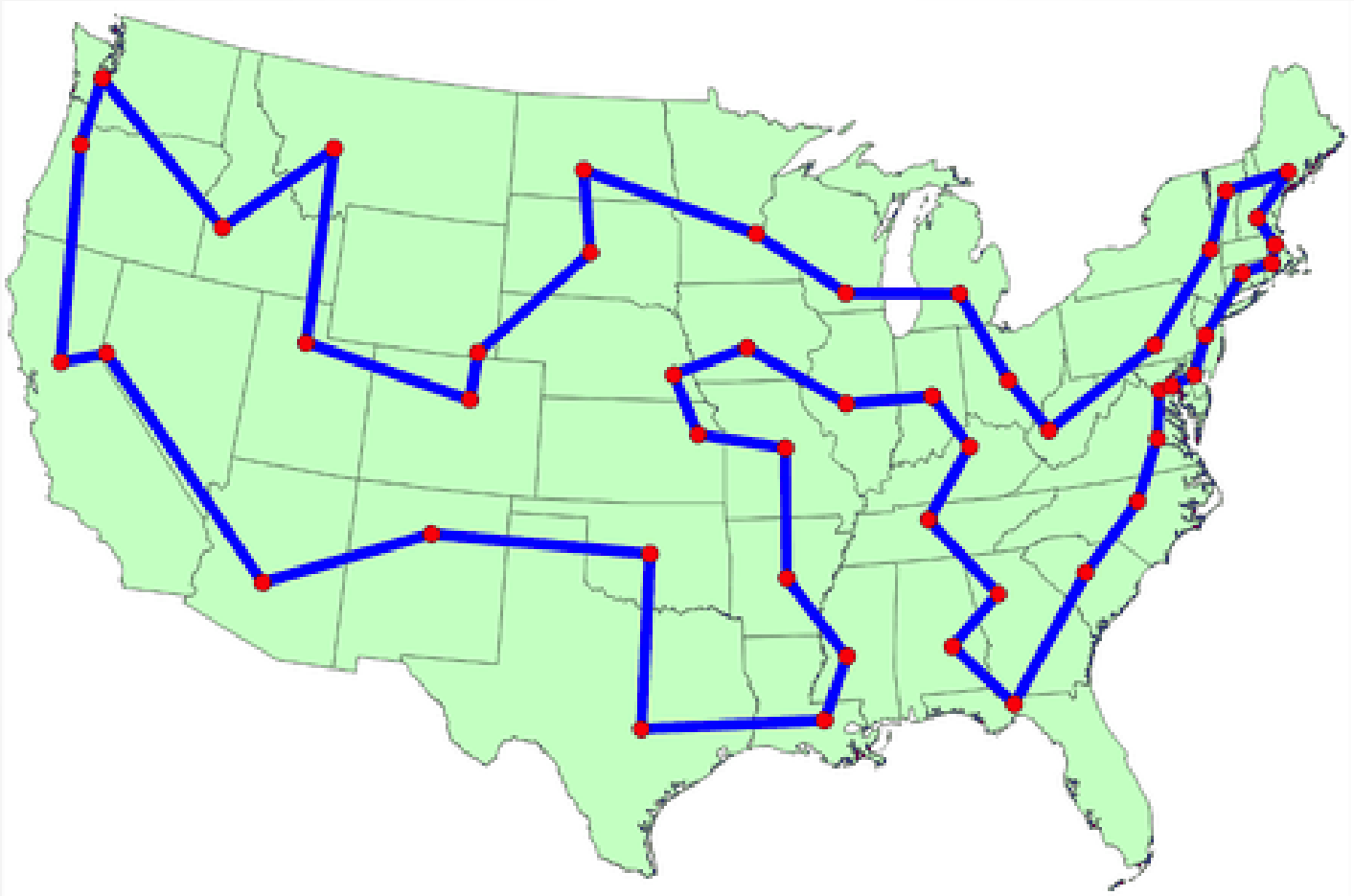


birds

hunters

[Teuvo Kohonen 2001] Self-Organizing Maps; Springer;

How To Solve Traveling Salesman Problem using SOM



Install Mini

- Source activate env
- Pip install minisom

Color SOM

Import library

```
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import patches as patches
```

Color SOM

Create Color Data

```
def generate_data():
    # use random colours
    raw_data = np.random.randint(0, 255, (3, 100))

    data = raw_data / raw_data.max()
    return data
```

Color SOM

Create SOM network

```
def create_som(width_som,height_som, number_features):
    # setup random weights between 0 and 1
    # weight matrix needs to be one m-dimensional vector for each neuron in the SOM
    net = np.random.random((width_som, height_som, number_features))
    return net
```

Color SOM

Find BMU

```
def find_bmu(t, net, m):
    """
        Find the best matching unit for a given vector, t, in the SOM
        Returns: a (bmu, bmu_idx) tuple where bmu is the high-dimensional BMU
                 and bmu_idx is the index of this vector in the SOM
    """
    bmu_idx = np.array([0, 0])
    # set the initial minimum distance to a huge number
    min_dist = np.iinfo(np.int).max
    # calculate the high-dimensional distance between each neuron and the input
    for x in range(net.shape[0]):
        for y in range(net.shape[1]):
            w = net[x, y, :].reshape(m, 1)
            # don't bother with actual Euclidean distance, to avoid expensive sqrt operation
            sq_dist = np.sum((w - t) ** 2)
            if sq_dist < min_dist:
                min_dist = sq_dist
                bmu_idx = np.array([x, y])
    # get vector corresponding to bmu_idx
    bmu = net[bmu_idx[0], bmu_idx[1], :].reshape(m, 1)
    # return the (bmu, bmu_idx) tuple
    return (bmu, bmu_idx)
```

Color SOM

Decay Radius

```
def decay_radius(initial_radius, i, time_constant):
    return initial_radius * np.exp(-i / time_constant)
```

Color SOM

Decay Learning rate

```
def calculate_influence(distance, radius):
    return np.exp(-distance / (2 * (radius ** 2)))
```

Color SOM

Compute Neighborhood influence

```
def calculate_influence(distance, radius):
    return np.exp(-distance / (2 * (radius ** 2)))
```

Color SOM

Training

```
def train(data, n_iterations, net, init_radius, time_constant, init_learning_rate):
    n=data.shape[1]
    m=data.shape[0]

    for i in range(n_iterations):
        if (i % 1000)==0:
            plot_som(net, i)
        # select a training example at random
        t = data[:, np.random.randint(0, n)].reshape(np.array([m, 1]))
        bmu, bmu_idx = find_bmu(t, net, m)
        r = decay_radius(init_radius, i, time_constant)
        l = decay_learning_rate(init_learning_rate, i, n_iterations)

        for x in range(net.shape[0]):
            for y in range(net.shape[1]):
                w = net[x, y, :].reshape(m, 1)
                w_dist = np.sum((np.array([x, y]) - bmu_idx) ** 2)
                if w_dist <= r ** 2:
                    influence = calculate_influence(w_dist, r)
                    new_w = w + (l * influence * (t - w))
                    net[x, y, :] = new_w.reshape(1, 3)
```

Color SOM

Plot color

```
def plot_som(net, n_iterations):
    fig = plt.figure()
    # setup axes
    ax = fig.add_subplot(111, aspect='equal')
    ax.set_xlim((0, net.shape[0]+1))
    ax.set_ylim((0, net.shape[1]+1))
    ax.set_title('Self-Organising Map after %d iterations' % n_iterations)

    # plot the rectangles
    for x in range(1, net.shape[0] + 1):
        for y in range(1, net.shape[1] + 1):
            ax.add_patch(patches.Rectangle((x-0.5, y-0.5), 1, 1,
                    facecolor=net[x-1,y-1,:],
                    edgecolor='none'))
```

Color SOM

Main function

```
def main():
    data=generate_data()
    print(data)
    number_features=3
    width_som=5
    height_som=5
    n_iterations = 10000
    init_learning_rate = 0.01
    # initial neighbourhood radius
    init_radius = max(width_som, height_som) / 2
    # radius decay parameter
    time_constant = n_iterations / np.log(init_radius)
    net=create_som(width_som, height_som, number_features)
    plot_som(net, 0)
    train(data, n_iterations, net, init_radius, time_constant, init_learning_rate)
    plot_som(net, n_iterations)
    plt.show()


if __name__ == '__main__':
    main()
```

Irish SOM

import library

```
from minisom import MiniSom

import numpy as np
import matplotlib.pyplot as plt
```

Irish flower SOM

Load irish flower data

```
def load_irish():
    data = np.genfromtxt('iris.csv', delimiter=',', usecols=(0, 1, 2, 3))
    # data normalization
    data = np.apply_along_axis(lambda x: x/np.linalg.norm(x), 1, data)
    return data
```

Irish flower SOM

Create SOM

```
def create_som(data):
    som = MiniSom(5, 5, 4, sigma=2.0, learning_rate=0.2)
    som.random_weights_init(data)
    return som
```

Irish flower SOM

Plot SOM

```
def plot_irish(som, data):
    # Plotting the response for each pattern in the iris dataset
    plt.bone()
    plt.pcolor(som.distance_map().T)  # plotting the distance map as background
    plt.colorbar()

    target = np.genfromtxt('iris.csv', delimiter=',', usecols=(4), dtype=str)
    t = np.zeros(len(target), dtype=int)
    t[target == 'setosa'] = 0
    t[target == 'versicolor'] = 1
    t[target == 'virginica'] = 2

    # use different colors and markers for each label
    markers = ['o', 's', 'D']
    colors = ['r', 'g', 'b']
    for cnt, xx in enumerate(data):
        w = som.winner(xx)  # getting the winner
        # palce a marker on the winning position for the sample xx
        plt.plot(w[0] + .5, w[1] + .5, markers[t[cnt]], markerfacecolor='None',
                 markeredgecolor=colors[t[cnt]], markersize=12, markeredgewidth=2)
    plt.axis([0, 7, 0, 7])
    plt.show()
```

Irish flower SOM

Main function

```
if __name__ == '__main__':
    data=load_irish()
    som=create_som(data)

    print("Training...")
    som.train_random(data, 100)  # random training
    print("\n...ready!")


    plot_irish(som, data)
```

Mnist SOM

Import library

```
from sklearn import datasets
from minisom import MiniSom
import numpy as np
import matplotlib.pyplot as plt
```

Mnist SOM

Load data

```
def load_data():
    # load the digits dataset from scikit-learn
    # 901 samples,  about 180 samples per class
    # the digits represented 0, 1, 2, 3, 4

    digits = datasets.load_digits(n_class=4)
    data = digits.data  # matrix where each row is a vector that represent a digit.
    num = digits.target  # num[i] is the digit represented by data[i]
    return digits, data,num
```

Mnist SOM

Create SOM

```
def create_som():
    som = MiniSom(20, 20, 64, sigma=1.5, learning_rate=0.5)
    return som
```

Mnist SOM

Plot SOM

```
def plot_mnist(data, num, digits):
    plt.figure(figsize=(7, 7))
    wmap = {}
    im = 0
    for x, t in zip(data, num):  # scatterplot
        w = som.winner(x)
        wmap[w] = im
        plt.text(w[0] + .5, w[1] + .5, str(t),
                 color=plt.cm.Dark2(t / 4.), fontdict={'weight': 'bold', 'size': 11})
        im = im + 1
    plt.axis([0, som.get_weights().shape[0], 0, som.get_weights().shape[1]])
    plt.figure(figsize=(10, 10), facecolor='white')
    cnt = 0
    for j in reversed(range(20)):  # images mosaic
        for i in range(20):
            plt.subplot(20, 20, cnt + 1, frameon=False, xticks=[], yticks=[])
            if (i, j) in wmap:
                plt.imshow(digits.images[wmap[(i, j)]],
                           cmap='Greys', interpolation='nearest')
            else:
                plt.imshow(np.zeros((8, 8)), cmap='Greys')
            cnt = cnt + 1

    plt.tight_layout()
    plt.show()
```

Mnist SOM

Main function

```
if __name__ == '__main__':

    digits, data, num=load_data()
    som=create_som()

    print("Training...")
    som.train_random(data, 1500)  # random training
    print("\n...ready!")

    plot_mnist(data,num, digits)
```