Instituto Politécnico Nacional
Escuela Superior de Cómputo
Evolutionary Computing
8 Self-Organizing Maps
Martínez Coronel Brayan Yosafat
Rosas Trigueros Jorge Luis
29/10/2021
12/11/2021

Theorical framework

Self-Organizing maps were described by Teuvo Kohonen around the 80's in its paper: The Self-Organizing Map, although it is considered a neural network, the Kohonen network doesn't use active functions or the Backpropagation algorithm (which is quite innovate when it was published). Kohonen describes a competitive learning and the use of unsupervised learning (which I find hard to read) in its paper [1], [2]:

Basically, it has two layers, the input and the output (also known as feature map), there is no activation function, so it directly passes input to outputs, the output layer is made as a rectangle of some width and some height (both predefined as a hyperparameter), so it has coordinates, each coordinate has a neuron, and each neuron has a weight vector (of the same shape as the input) [1].

As we mentioned, this network uses competitive learning and not the Stochastic Gradient Descent, but it also takes two ideas, adaptation, and cooperation, given a dataset, we iterate over every vector in it, all neurons are evaluated with its weight vector, and the most like the input is called the BMU, after that, all neurons are modified to be more like the BMU. BMU means Best Machine Unit, this is repeated a certain number of iterations (this is also an hyperparameter) and then we can use the SOM (Self-organizing map) to predict (get the BMU) of some input [2].

Something really interesting is the fact that doctor Kohonen tells that the motivation behind this network is physics and scouts, which, at first, doesn’t make sense, but the attraction between neurons reminds us like punctual charges in physics, and searching in a two-dimensional terrain just like scouts, is something that keeps me hyped about this network [3].

Material and equipment

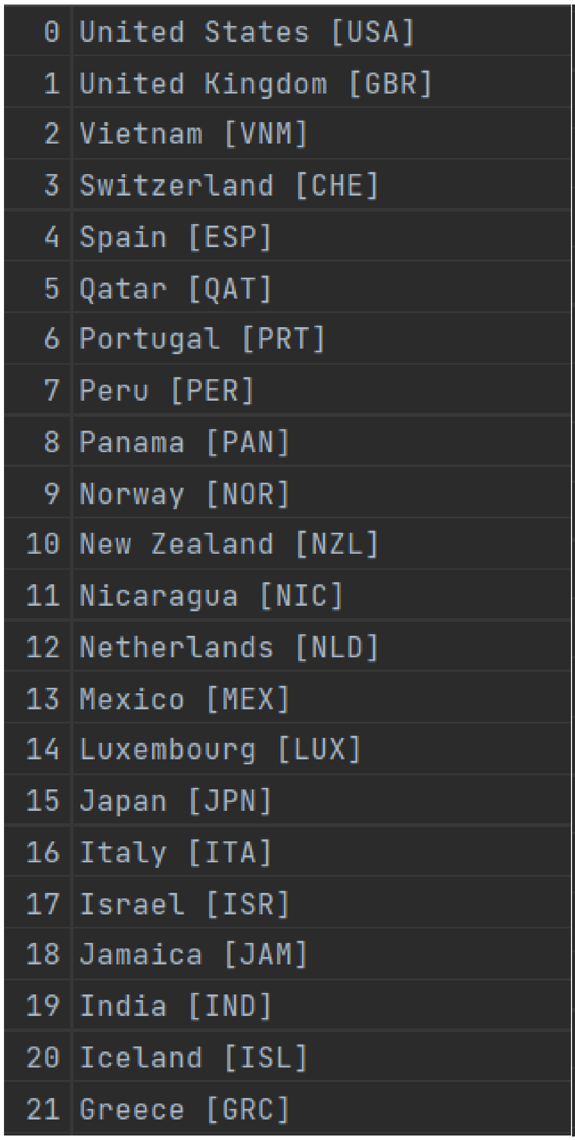
In this practice, DataSpell with Jupiter Notebooks were used to the development, all code were executed in a ThinkPad with a Core i7.

Practice development

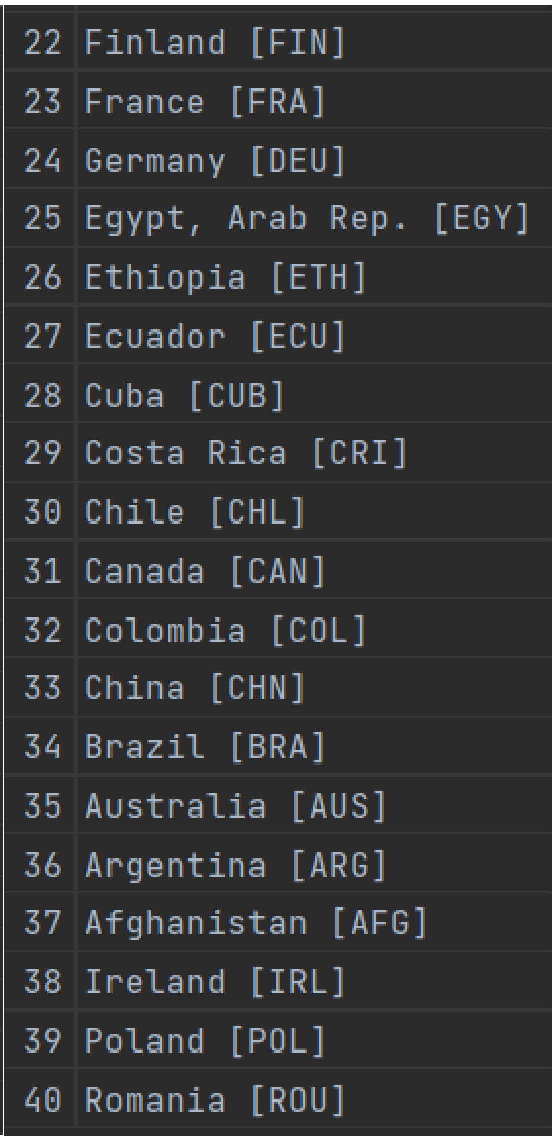
The objective of this practice is to use SOM for 40 countries using 10 values taken from the World Bank, those topics were the next ones:

- Access to electricity (% of population)
- "Adolescent fertility rate (births per 1,000 women ages 15-19)"
- Arable land (% of land area)
- Armed forces personnel (% of total labor force)
- "Energy imports, net (% of energy use)"
- Individuals using the Internet (% of population)
- Current health expenditure (% of GDP)
- CO2 emissions from transport (% of total fuel combustion)
- "Communications, computer, etc. (% of service exports, BoP)"
- "Communications, computer, etc. (% of service imports, BoP)"

And the list of the (41 to be precise) countries is:



```
 0 United States [USA]
 1 United Kingdom [GBR]
 2 Vietnam [VNM]
 3 Switzerland [CHE]
 4 Spain [ESP]
 5 Qatar [QAT]
 6 Portugal [PRT]
 7 Peru [PER]
 8 Panama [PAN]
 9 Norway [NOR]
10 New Zealand [NZL]
11 Nicaragua [NIC]
12 Netherlands [NLD]
13 Mexico [MEX]
14 Luxembourg [LUX]
15 Japan [JPN]
16 Italy [ITA]
17 Israel [ISR]
18 Jamaica [JAM]
19 India [IND]
20 Iceland [ISL]
21 Greece [GRC]
```



```
22 Finland [FIN]
23 France [FRA]
24 Germany [DEU]
25 Egypt, Arab Rep. [EGY]
26 Ethiopia [ETH]
27 Ecuador [ECU]
28 Cuba [CUB]
29 Costa Rica [CRI]
30 Chile [CHL]
31 Canada [CAN]
32 Colombia [COL]
33 China [CHN]
34 Brazil [BRA]
35 Australia [AUS]
36 Argentina [ARG]
37 Afghanistan [AFG]
38 Ireland [IRL]
39 Poland [POL]
40 Romania [ROU]
```

Now, the SOM and node classes are predefined within the code given by the professor, but with some changes for the sake of readability we end up using this one (which, of course we credit the authors):

Code for SOM and Node class definitions

```
## Original code from: Self-Organizing Maps by Paras Chopra and Jorge
Trigueros
from random import *
from math import *

class Node:
    def __init__(self, feature_size=10, prediction_size=10, x=0, y=0):
        self.feature_size = feature_size
        self.prediction_size = prediction_size
        self.feature_vector = [0.0] * feature_size              # Feature
Vector
        self.prediction_vector = [0.0] * prediction_size        #
Prediction Vector
        self.x = x                                              # X
location
        self.y = y                                              # Y
location
        self.feature_vector = [random() for _ in range(feature_size)]
        self.prediction_vector = [random() for _ in
range(prediction_size)]


class SOM:
    # Let distance=False if you want to auto-calculate the distance
    def __init__(self, height=10, width=10, feature_size=10,
prediction_size=10, distance=False, learning_rate=0.005):
        self.height = height
        self.width = width
        self.distance = distance if distance else (height+width)/2
        self.total = height * width
        self.learning_rate = learning_rate
        self.nodes = [0] * self.total
        self.feature_size = feature_size
        self.prediction_size = prediction_size

        for i in range(self.height):
            for j in range(self.width):
                self.nodes[i*self.width + j] = Node(feature_size,
prediction_size, i, j)


    # Train_vector format: [ [feature_vector[0], prediction_vector[0]],
    #                        [feature_vector[1], prediction_vector[1]],
so on..
    def train(self, train_vector, iterations=1000):
        time_constant = iterations / log(self.distance)

        for i in range(1,iterations+1):
            if i % 10 == 0: print(i, end=', ')

            distance_decaying = self.distance * exp(-1.0 *
i/time_constant)
            learning_rate_decaying = self.learning_rate * exp(-
1.0*i/time_constant)

            for j in range(len(train_vector)):
                input_feature = train_vector[j][0]
                input_prediction = train_vector[j][1]
                best = self.best_match(input_feature)

                stack = []
                for k in range(self.total):
                    distance = SOM.distance(self.nodes[best],
self.nodes[k])

                    if distance < distance_decaying:
                        temporal_feature = [0.0] * self.feature_size
                        temporal_prediction=[0.0]*self.prediction_size
                        influence = exp((-1.0*(distance**2)) /
(2*distance_decaying*i))

                        for l in range(self.feature_size):      #
Learning
                            temporal_feature[l] =
self.nodes[k].feature_vector[l] + influence * learning_rate_decaying \
                                                  * (input_feature[l] -
self.nodes[k].feature_vector[l])

                        for l in range(self.prediction_size):   #
Learning
                            temporal_prediction[l] =
self.nodes[k].prediction_vector[l] + influence * learning_rate_decaying \
                                                     *
(input_prediction[l] - self.nodes[k].prediction_vector[l])

                        # Push the unit onto stack to update in next
interval
                        stack[0:0] = [[[k], temporal_feature,
temporal_prediction]]

                for l in range(len(stack)):
                    self.nodes[stack[l][0][0]].feature_vector[:] =
stack[l][1][:]
                    self.nodes[stack[l][0][0]].prediction_vector[:] =
stack[l][2][:]


    # Returns prediction vector
    def predict(self, input_vector, return_coors=False):
        best = self.best_match(input_vector)

        if return_coors:
          return self.nodes[best].prediction_vector, self.nodes[best].x,
self.nodes[best].y

        return self.nodes[best].prediction_vector


    # Returns best matching unit's index
    def best_match(self, target_feature):
        minimum = sqrt(self.feature_size)               # Minimum
distance
        minimum_index = 1                               # Minimum
distance unit

        for i in range(self.total):
            temp = self.feature_distance(self.nodes[i].feature_vector,
target_feature)
            if temp < minimum:
                minimum = temp
                minimum_index = i

        return minimum_index
    def feature_distance(self, feature1, feature2):
        temp=0.0

        for j in range(self.feature_size):
            temp += (feature1[j]-feature2[j])**2

        temp = sqrt(temp)
        return temp

    @staticmethod
    def distance(n1: Node, n2: Node):
        return sqrt((n1.x - n2.x)**2 + (n1.y - n2.y)**2)
```

Code 1. SOM and node class definitions

Both are straightforward to understand, the first just consist about an element from the Self-Organizing Map and the other class definition is just how it works between nodes. Before we dive deeper into code, we should talk about how data had to be preprocessed to work with it.

Data preparation

Null values were represented by two dots in Bank World data, we download a csv from three different years, but just used 2013 for the training of SOM, some columns and rows were taken off from the csv and saved in another csv to preserve the original, there is an entire notebook for data preparation:

```
import pandas as pd
import numpy as np

frame: pd.DataFrame = pd.read_csv('Datos 2013 2014 2015.csv')
print(frame)
```

Code 2. Reading data gathered

```
frame = frame.drop(range(30, 35))
frame = frame.drop([‘Time Code’, ‘Series Code’], axis=1)
print(frame)
```

Code 3. Dropping rows

```
frame = frame.replace('..', np.NaN)
frame[frame.columns[2:]] = frame[frame.columns[2:]].astype(float)
print(frame.dtypes)
```

Code 4. Replacing two dots with NaN and casting as float

```
frame = frame.replace(np.NaN, 0)
frame[:10].to_csv('2013.csv', index=False)
frame[10:20].to_csv('2014.csv', index=False)
frame[20:].to_csv('2015.csv', index=False)
```

Code 5. Saving to csv

So, as we see, nan values were replaced with zeros, of course there are better methods, like getting the average of the row and using it, but this was just to make the data useful for the SOM.

Preparing the dataset

```
import pandas as pd

HEIGHT = 30
WIDTH = 30
FEATURE_SIZE = 10
PREDICTION_SIZE = 1

# for year in ['2013', '2014', '2015']:
data = pd.read_csv('2013' + '.csv')
countries = data.columns[2:]
data = data[countries]
data = (data - data.mean()) / data.std()
training_set = []
kohonen_network = SOM(HEIGHT, WIDTH, FEATURE_SIZE, PREDICTION_SIZE,
distance=False, learning_rate=0.05)

for index in range(len(countries)):
    training_set.append([data[countries[index]].to_numpy(), [index]])
```

Code 6. Normalization of data

Every column from index 2 and forward represents a country, so, the SOM need a list of pairs training vector – expected output, so that is what we make before training the model, also, the model is created at this moment.

Training and testing

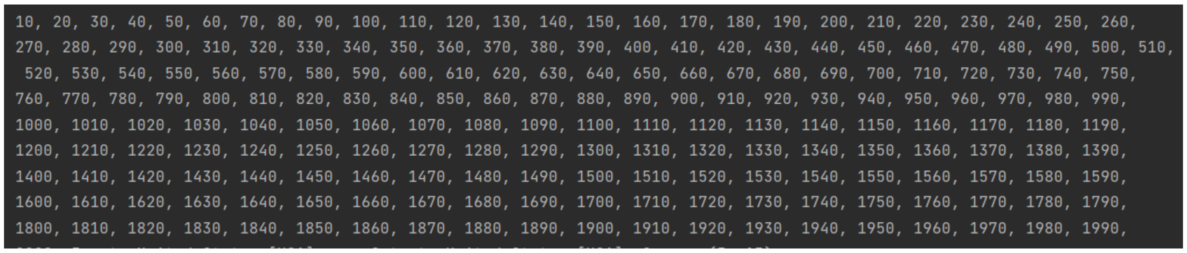
```
ITERATIONS = 2000
kohonen_network.train(training_set, ITERATIONS)
```

Code 7. Training code

```
count = 0
results = pd.DataFrame()
```

```
for index in range(len(countries)):
    prediction =
kohonen_network.predict(data[countries[index]].to_numpy(), True)
    print(f'Input: {countries[index]}      Output:
{countries[round(prediction[0][0])]}  Coors: {prediction[1:]}')
    if countries[index] == countries[round(prediction[0][0])]: count =
count + 1

print(f'Número de aciertos: {count} de {len(countries)}')
```

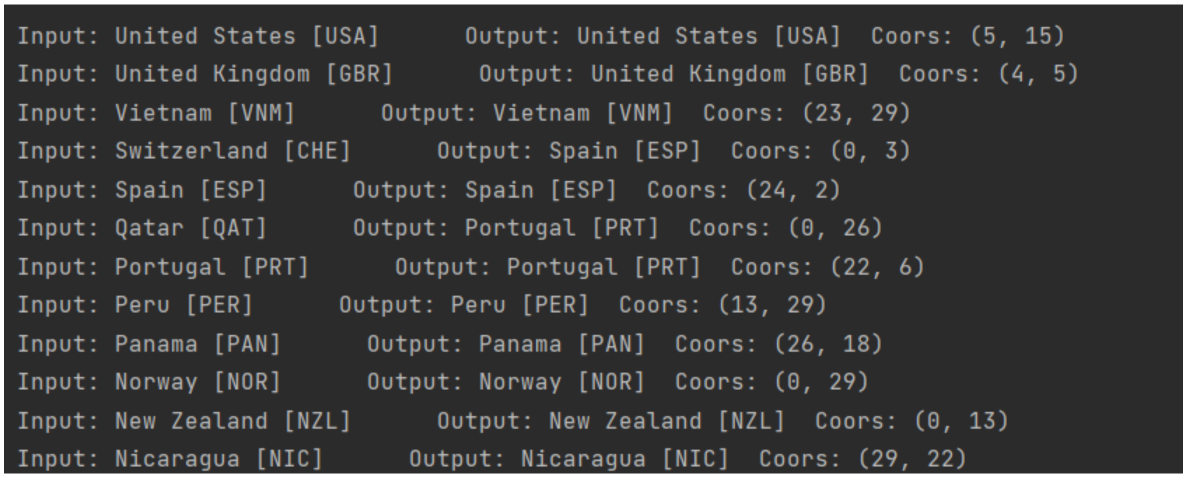
Code 8. Printing all 41 countries and BMU



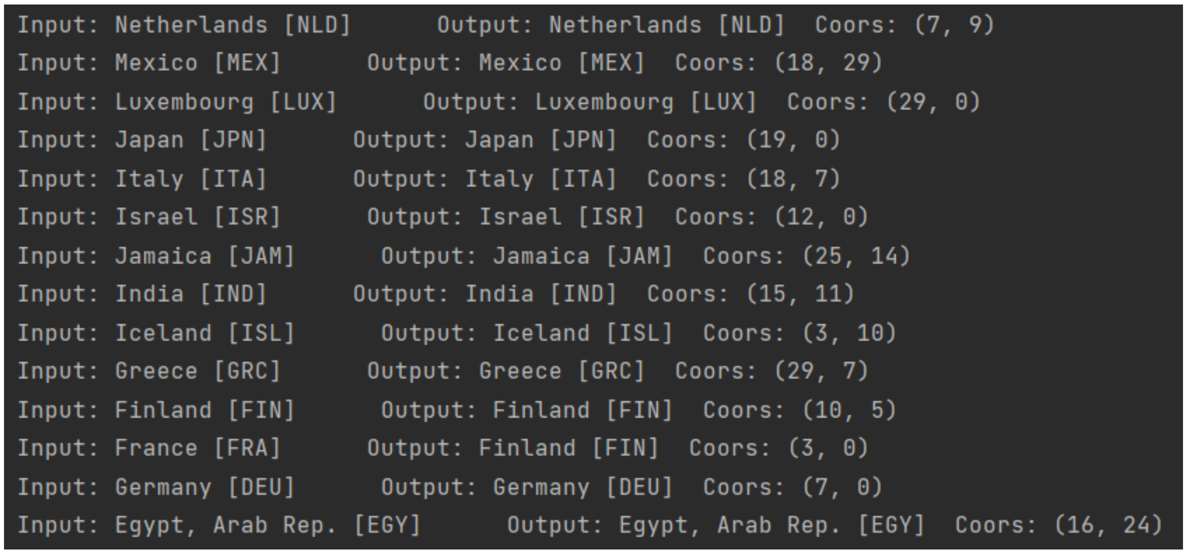
```
10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250, 260,
270, 280, 290, 300, 310, 320, 330, 340, 350, 360, 370, 380, 390, 400, 410, 420, 430, 440, 450, 460, 470, 480, 490, 500, 510,
 520, 530, 540, 550, 560, 570, 580, 590, 600, 610, 620, 630, 640, 650, 660, 670, 680, 690, 700, 710, 720, 730, 740, 750,
760, 770, 780, 790, 800, 810, 820, 830, 840, 850, 860, 870, 880, 890, 900, 910, 920, 930, 940, 950, 960, 970, 980, 990,
1000, 1010, 1020, 1030, 1040, 1050, 1060, 1070, 1080, 1090, 1100, 1110, 1120, 1130, 1140, 1150, 1160, 1170, 1180, 1190,
1200, 1210, 1220, 1230, 1240, 1250, 1260, 1270, 1280, 1290, 1300, 1310, 1320, 1330, 1340, 1350, 1360, 1370, 1380, 1390,
1400, 1410, 1420, 1430, 1440, 1450, 1460, 1470, 1480, 1490, 1500, 1510, 1520, 1530, 1540, 1550, 1560, 1570, 1580, 1590,
1600, 1610, 1620, 1630, 1640, 1650, 1660, 1670, 1680, 1690, 1700, 1710, 1720, 1730, 1740, 1750, 1760, 1770, 1780, 1790,
1800, 1810, 1820, 1830, 1840, 1850, 1860, 1870, 1880, 1890, 1900, 1910, 1920, 1930, 1940, 1950, 1960, 1970, 1980, 1990,
```

Fig. 1 Training prints

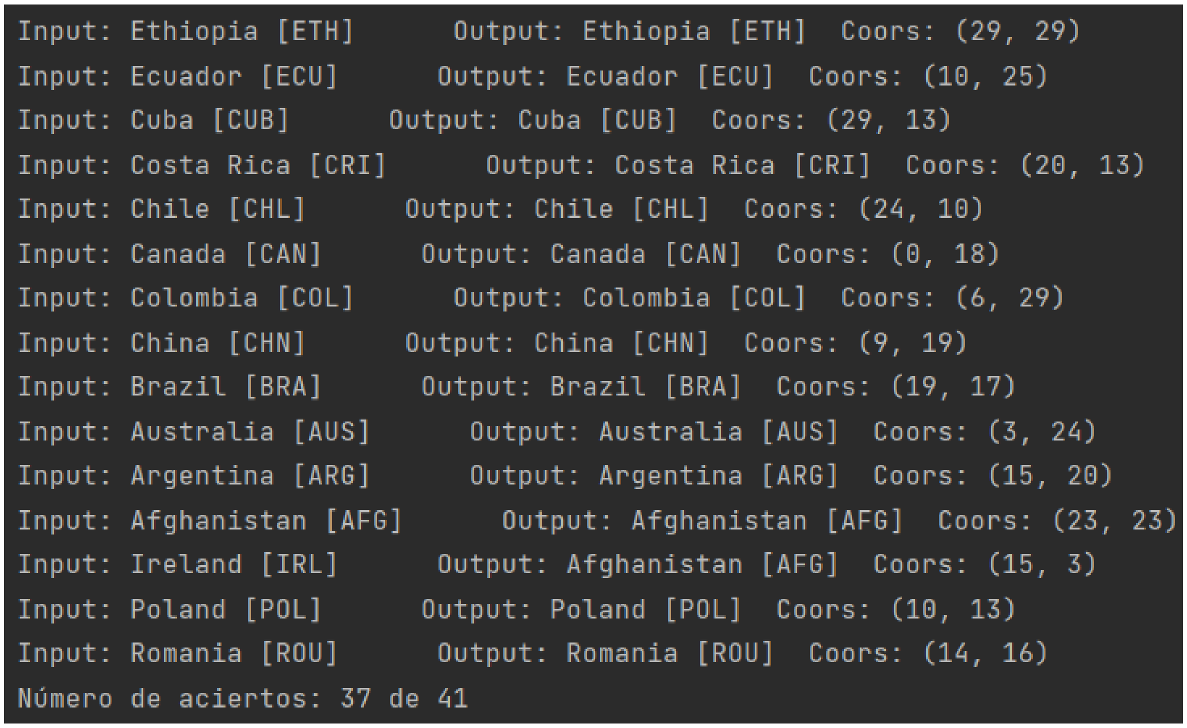
Both codes are small, but generate a relatively big amount of output, the most interesting part about this code is, of course, the results from the BMU of every country:



```
Input: United States [USA]      Output: United States [USA]  Coors: (5, 15)
Input: United Kingdom [GBR]      Output: United Kingdom [GBR]  Coors: (4, 5)
Input: Vietnam [VNM]      Output: Vietnam [VNM]  Coors: (23, 29)
Input: Switzerland [CHE]      Output: Spain [ESP]  Coors: (0, 3)
Input: Spain [ESP]      Output: Spain [ESP]  Coors: (24, 2)
Input: Qatar [QAT]      Output: Portugal [PRT]  Coors: (0, 26)
Input: Portugal [PRT]      Output: Portugal [PRT]  Coors: (22, 6)
Input: Peru [PER]      Output: Peru [PER]  Coors: (13, 29)
Input: Panama [PAN]      Output: Panama [PAN]  Coors: (26, 18)
Input: Norway [NOR]      Output: Norway [NOR]  Coors: (0, 29)
Input: New Zealand [NZL]      Output: New Zealand [NZL]  Coors: (0, 13)
Input: Nicaragua [NIC]      Output: Nicaragua [NIC]  Coors: (29, 22)
```



```
Input: Netherlands [NLD]      Output: Netherlands [NLD]  Coors: (7, 9)
Input: Mexico [MEX]      Output: Mexico [MEX]  Coors: (18, 29)
Input: Luxembourg [LUX]      Output: Luxembourg [LUX]  Coors: (29, 0)
Input: Japan [JPN]      Output: Japan [JPN]  Coors: (19, 0)
Input: Italy [ITA]      Output: Italy [ITA]  Coors: (18, 7)
Input: Israel [ISR]      Output: Israel [ISR]  Coors: (12, 0)
Input: Jamaica [JAM]      Output: Jamaica [JAM]  Coors: (25, 14)
Input: India [IND]      Output: India [IND]  Coors: (15, 11)
Input: Iceland [ISL]      Output: Iceland [ISL]  Coors: (3, 10)
Input: Greece [GRC]      Output: Greece [GRC]  Coors: (29, 7)
Input: Finland [FIN]      Output: Finland [FIN]  Coors: (10, 5)
Input: France [FRA]      Output: Finland [FIN]  Coors: (3, 0)
Input: Germany [DEU]      Output: Germany [DEU]  Coors: (7, 0)
Input: Egypt, Arab Rep. [EGY]      Output: Egypt, Arab Rep. [EGY]  Coors: (16, 24)
```



```
Input: Ethiopia [ETH]      Output: Ethiopia [ETH]  Coors: (29, 29)
Input: Ecuador [ECU]      Output: Ecuador [ECU]  Coors: (10, 25)
Input: Cuba [CUB]      Output: Cuba [CUB]  Coors: (29, 13)
Input: Costa Rica [CRI]      Output: Costa Rica [CRI]  Coors: (20, 13)
Input: Chile [CHL]      Output: Chile [CHL]  Coors: (24, 10)
Input: Canada [CAN]      Output: Canada [CAN]  Coors: (0, 18)
Input: Colombia [COL]      Output: Colombia [COL]  Coors: (6, 29)
Input: China [CHN]      Output: China [CHN]  Coors: (9, 19)
Input: Brazil [BRA]      Output: Brazil [BRA]  Coors: (19, 17)
Input: Australia [AUS]      Output: Australia [AUS]  Coors: (3, 24)
Input: Argentina [ARG]      Output: Argentina [ARG]  Coors: (15, 20)
Input: Afghanistan [AFG]      Output: Afghanistan [AFG]  Coors: (23, 23)
Input: Ireland [IRL]      Output: Afghanistan [AFG]  Coors: (15, 3)
Input: Poland [POL]      Output: Poland [POL]  Coors: (10, 13)
Input: Romania [ROU]      Output: Romania [ROU]  Coors: (14, 16)
Número de aciertos: 37 de 41
```

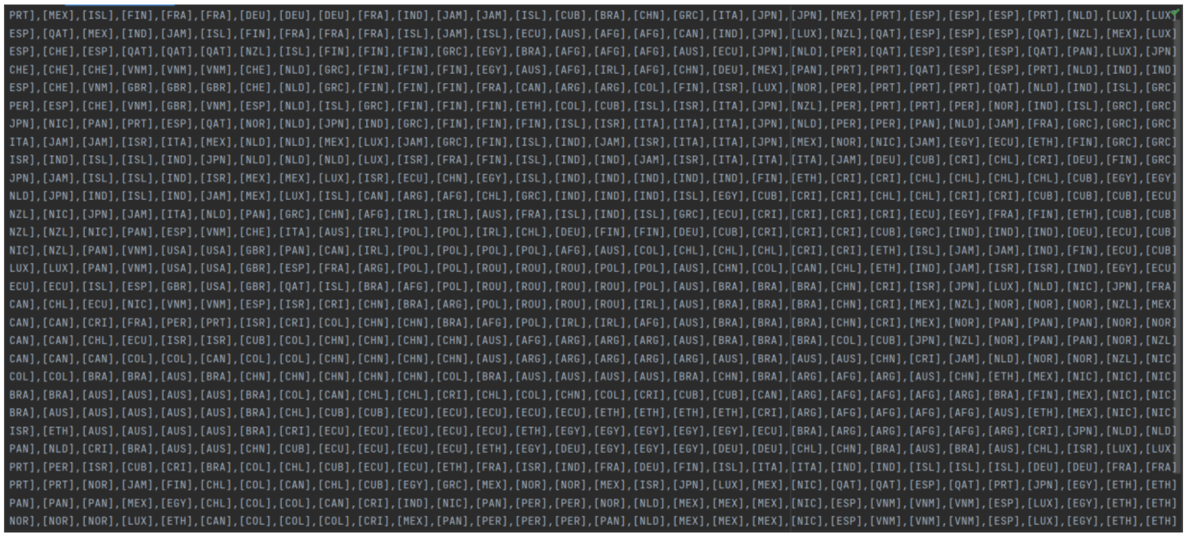
Fig. 2, 3 and 4. BMU of the 41 countries

One interesting thing about this is that, even with 2000 iterations, it fails with 4 countries, even, talking about the fact that it receives 10 * 41 numbers is kind of unfair just counting its errors because it gets rights with 37 countries.

Map

```python
grid = []

for col in range(HEIGHT):
    row = [round(kohonen_network.nodes[i * WIDTH +
col].prediction_vector[0]) for i in range(WIDTH)]
    grid.append(row)

grid_frame = pd.DataFrame(grid)
grid_frame.to_csv('map.csv', header=False, index=False)
print(grid_frame)

countries_frame = pd.DataFrame(countries)
countries_frame.to_csv('countries.csv', header=False)

for i in range(len(countries)):
    country = countries[i]
    grid_frame = grid_frame.replace(i, country[-5:])

grid_frame.to_csv('map_names.csv', header=False, index=False)
```

Code 9. Printing the map



```
PRT],[MEX],[ISL],[FIN],[FRA],[FRA],[DEU],[DEU],[DEU],[FRA],[IND],[JAM],[JAM],[ISL],[CUB],[BRA],[CHN],[GRC],[ITA],[JPN],[JPN],[MEX],[PRT],[ESP],[ESP],[ESP],[PRT],[NLD],[LUX],[LUX]
ESP],[QAT],[MEX],[IND],[JAM],[ISL],[FIN],[FRA],[FRA],[FRA],[ISL],[JAM],[ISL],[ECU],[AUS],[AFG],[AFG],[CAN],[IND],[JPN],[LUX],[NZL],[QAT],[ESP],[ESP],[ESP],[QAT],[NZL],[MEX],[LUX]
ESP],[CHE],[ESP],[QAT],[QAT],[QAT],[NZL],[ISL],[FIN],[FIN],[FIN],[GRC],[EGY],[BRA],[AFG],[AFG],[AFG],[AUS],[ECU],[JPN],[NLD],[PER],[QAT],[ESP],[ESP],[ESP],[QAT],[PAN],[LUX],[JPN]
CHE],[CHE],[CHE],[VNM],[VNM],[VNM],[CHE],[NLD],[GRC],[FIN],[FIN],[FIN],[EGY],[AUS],[AFG],[IRL],[AFG],[CHN],[DEU],[MEX],[PAN],[PRT],[PRT],[QAT],[ESP],[ESP],[PRT],[NLD],[IND],[IND]
ESP],[CHE],[VNM],[GBR],[GBR],[GBR],[CHE],[NLD],[GRC],[FIN],[FIN],[FIN],[FRA],[CAN],[ARG],[ARG],[COL],[FIN],[ISR],[LUX],[NOR],[PER],[PRT],[PRT],[PRT],[QAT],[NLD],[IND],[ISL],[GRC]
PER],[ESP],[CHE],[VNM],[GBR],[VNM],[ESP],[NLD],[ISL],[GRC],[FIN],[FIN],[FIN],[ETH],[COL],[CUB],[ISL],[ISR],[ITA],[JPN],[NZL],[PER],[PRT],[PRT],[PER],[NOR],[IND],[ISL],[GRC],[GRC]
JPN],[NIC],[PAN],[PRT],[ESP],[QAT],[NOR],[NLD],[JPN],[IND],[GRC],[FIN],[FIN],[FIN],[ISL],[ISR],[ITA],[ITA],[ITA],[JPN],[NLD],[PER],[PER],[PAN],[NLD],[JAM],[FRA],[GRC],[GRC],[GRC]
ITA],[JAM],[JAM],[ISR],[ITA],[MEX],[NLD],[NLD],[MEX],[LUX],[JAM],[GRC],[FIN],[ISL],[IND],[JAM],[ISR],[ITA],[ITA],[JPN],[MEX],[NOR],[NIC],[JAM],[EGY],[ECU],[ETH],[FIN],[GRC],[GRC]
ISR],[IND],[ISL],[ISL],[IND],[JPN],[NLD],[NLD],[NLD],[LUX],[ISR],[FRA],[FIN],[ISL],[IND],[IND],[JAM],[ISR],[ITA],[ITA],[ITA],[JAM],[DEU],[CUB],[CRI],[CHL],[CRI],[DEU],[FIN],[GRC]
JPN],[JAM],[ISL],[ISL],[IND],[ISR],[MEX],[MEX],[LUX],[ISR],[ECU],[CHN],[EGY],[ISL],[IND],[IND],[IND],[IND],[IND],[FIN],[ETH],[CRI],[CRI],[CHL],[CHL],[CHL],[CHL],[CUB],[EGY],[EGY]
NLD],[JPN],[IND],[ISL],[IND],[JAM],[MEX],[LUX],[ISL],[CAN],[ARG],[AFG],[CHL],[GRC],[IND],[IND],[IND],[ISL],[EGY],[CUB],[CRI],[CRI],[CHL],[CHL],[CRI],[CRI],[CUB],[CUB],[CUB],[ECU]
NZL],[NIC],[JPN],[JAM],[ITA],[NLD],[PAN],[GRC],[CHN],[AFG],[IRL],[IRL],[AUS],[FRA],[ISL],[IND],[ISL],[GRC],[ECU],[CRI],[CRI],[CRI],[CRI],[ECU],[EGY],[FRA],[FIN],[ETH],[CUB],[CUB]
NZL],[NZL],[NIC],[PAN],[ESP],[VNM],[CHE],[ITA],[AUS],[IRL],[POL],[POL],[IRL],[CHL],[DEU],[FIN],[FIN],[DEU],[CUB],[CRI],[CRI],[CRI],[CUB],[GRC],[IND],[IND],[IND],[DEU],[ECU],[CUB]
NIC],[NZL],[PAN],[VNM],[USA],[USA],[GBR],[PAN],[CAN],[IRL],[POL],[POL],[POL],[POL],[AFG],[AUS],[COL],[CHL],[CHL],[CHL],[CRI],[CRI],[ETH],[ISL],[JAM],[JAM],[IND],[FIN],[ECU],[CUB]
LUX],[LUX],[PAN],[VNM],[USA],[USA],[GBR],[ESP],[FRA],[ARG],[POL],[POL],[ROU],[ROU],[ROU],[POL],[POL],[AUS],[CHN],[COL],[CAN],[CHL],[ETH],[IND],[JAM],[ISR],[ISR],[IND],[EGY],[ECU]
ECU],[ECU],[ISL],[ESP],[GBR],[USA],[GBR],[QAT],[ISL],[BRA],[AFG],[POL],[ROU],[ROU],[ROU],[ROU],[POL],[AUS],[BRA],[BRA],[BRA],[CHN],[CRI],[ISR],[JPN],[LUX],[NLD],[NIC],[JPN],[FRA]
CAN],[CHL],[ECU],[NIC],[VNM],[VNM],[ESP],[ISR],[CRI],[CHN],[BRA],[ARG],[POL],[ROU],[ROU],[ROU],[IRL],[AUS],[BRA],[BRA],[BRA],[CHN],[CRI],[MEX],[NZL],[NOR],[NOR],[NOR],[NZL],[MEX]
CAN],[CAN],[CRI],[FRA],[PER],[PRT],[ISR],[CRI],[COL],[CHN],[CHN],[BRA],[AFG],[POL],[IRL],[IRL],[AFG],[AUS],[BRA],[BRA],[BRA],[CHN],[CRI],[MEX],[NOR],[PAN],[PAN],[PAN],[NOR],[NOR]
CAN],[CAN],[CHL],[ECU],[ISR],[ISR],[CUB],[COL],[CHN],[CHN],[CHN],[CHN],[AUS],[AFG],[ARG],[ARG],[ARG],[AUS],[BRA],[BRA],[BRA],[COL],[CUB],[JPN],[NZL],[NOR],[PAN],[PAN],[NOR],[NZL]
CAN],[CAN],[CAN],[COL],[COL],[CAN],[COL],[COL],[CHN],[CHN],[CHN],[CHN],[AUS],[ARG],[ARG],[ARG],[ARG],[ARG],[AUS],[BRA],[AUS],[AUS],[CHN],[CRI],[JAM],[NLD],[NOR],[NOR],[NZL],[NIC]
COL],[COL],[BRA],[BRA],[AUS],[BRA],[CHN],[CHN],[CHN],[CHN],[CHN],[COL],[BRA],[AUS],[AUS],[AUS],[AUS],[BRA],[CHN],[BRA],[ARG],[AFG],[ARG],[AUS],[CHN],[ETH],[MEX],[NIC],[NIC],[NIC]
BRA],[BRA],[AUS],[AUS],[AUS],[AUS],[BRA],[COL],[CAN],[CHL],[CHL],[CRI],[CHL],[COL],[CHN],[COL],[CRI],[CUB],[CUB],[CAN],[ARG],[AFG],[AFG],[AFG],[ARG],[BRA],[FIN],[MEX],[NIC],[NIC]
BRA],[AUS],[AUS],[AUS],[AUS],[AUS],[BRA],[CHL],[CUB],[CUB],[ECU],[ECU],[ECU],[ECU],[ECU],[ETH],[ETH],[ETH],[ETH],[CRI],[ARG],[AFG],[AFG],[AFG],[AFG],[AUS],[ETH],[MEX],[NIC],[NIC]
ISR],[ETH],[AUS],[AUS],[AUS],[AUS],[BRA],[CRI],[ECU],[ECU],[ECU],[ECU],[ECU],[ETH],[EGY],[EGY],[EGY],[EGY],[EGY],[ECU],[BRA],[ARG],[ARG],[AFG],[AFG],[ARG],[CRI],[JPN],[NLD],[NLD]
PAN],[NLD],[CRI],[BRA],[AUS],[AUS],[CHN],[CUB],[ECU],[ECU],[ECU],[ECU],[ETH],[EGY],[DEU],[EGY],[EGY],[EGY],[DEU],[DEU],[CHL],[CHN],[BRA],[AUS],[BRA],[AUS],[CHL],[ISR],[LUX],[LUX]
PRT],[PER],[ISR],[CUB],[CRI],[BRA],[COL],[CHL],[CUB],[ECU],[ECU],[ETH],[FRA],[ISR],[IND],[FRA],[DEU],[FIN],[ISL],[ITA],[ITA],[IND],[IND],[ISL],[ISL],[ISL],[DEU],[DEU],[FRA],[FRA]
PRT],[PRT],[NOR],[JAM],[FIN],[CHL],[COL],[CAN],[CHL],[CUB],[EGY],[GRC],[MEX],[NOR],[NOR],[MEX],[ISR],[JPN],[LUX],[MEX],[NIC],[QAT],[QAT],[ESP],[QAT],[PRT],[JPN],[EGY],[ETH],[ETH]
PAN],[PAN],[PAN],[MEX],[EGY],[CHL],[COL],[COL],[CAN],[CRI],[IND],[NIC],[PAN],[PER],[PER],[NOR],[NLD],[MEX],[MEX],[MEX],[NIC],[ESP],[VNM],[VNM],[VNM],[ESP],[LUX],[EGY],[ETH],[ETH]
NOR],[NOR],[NOR],[LUX],[ETH],[CAN],[COL],[COL],[COL],[CRI],[MEX],[PAN],[PER],[PER],[PER],[PAN],[NLD],[MEX],[MEX],[MEX],[NIC],[ESP],[VNM],[VNM],[VNM],[ESP],[LUX],[EGY],[ETH],[ETH]
```

Fig. 5 Self-Organizing Map

So, this is the result from the SOM training, it took around 30 minutes, which is not a big deal considering neural networks usually take a lot of time in training, so it was not too bad to run. And even, its accuracy is quite high.

Conclusions and recommendations

When I took neural networks, it was a basic introduction about them, but I tried to go deeper, especially in its history, I got amazed by all the things that I read, but some networks I just couldn't understand how its algorithm worked, even my professor told us that there exists some competitive technique, I really didn't get it until now. I already knew there was a Kohonen (which I though was Japanese) network, but just didn't investigate about it (the name sounds just as scary as Boltzmann Machine).

I am really amused how Kohonen get to that idea, recalling that all networks used Backpropagation and the Descent gradient, I can't imagine thinking about scouts and physics to have this idea. I'm looking forward to learning more about, well, everything, all topics have been pretty different, so I fell open to learn about something.

References

[1] A. Khazri. Self-Organizing Maps. 2019. Towards Data Science. Accessed on: Nov. 12, 2021. [Online] Available: towardsdatascience.com/self-organizing-maps-1b7d2a84e065

[2] T. Kohonen. The Self-Organizing Map. 1990. IEEE. Department of Computer Science, Helsinki University of Technology, Finland. Accessed on: Nov. 12, 2021. [Online] Available: https://sci2s.ugr.es/keel/pdf/algorithm/articulo/1990-Kohonen-PIEEE.pdf

[3] IEEE CIS History. Teuvo Kohonen April 2015. 2015. YouTube. Accessed on: Nov. 12, 2021. [Online] Available: https://www.youtube.com/watch?v=07wU9t-UcNA&ab_channel=IEEECISHistory