```
# Meant to run on Python 2.7.11, 64-bit

from math import sqrt

from numpy import (array, unravel_index, nditer, linalg, random, subtract,
                   power, exp, pi, zeros, arange, outer, meshgrid, dot)from
collections import defaultdict
from warnings import warn



def fast_norm(x):
    """Returns norm-2 of a 1-D numpy array.

    * faster than linalg.norm in case of 1-D arrays (numpy 1.9.2rc1).
    """
    return sqrt(dot(x, x.T))



class MiniSom(object):
    def __init__(self, x, y, input_len, sigma=1.0, learning_rate=0.5,
decay_function=None, random_seed=3000):
        """
            Initializes a Self Organizing Map.
            x,y - dimensions of the SOM
            input_len - number of the elements of the vectors in input
            sigma - spread of the neighborhood function (Gaussian), needs to be
adequate to the dimensions of the map.
            (at the iteration t we have sigma(t) = sigma / (1 + t/T) where T is
#num_iteration/2)
            learning_rate - initial learning rate
            (at the iteration t we have learning_rate(t) = learning_rate / (1 +
t/T) where T is #num_iteration/2)
            decay_function, function that reduces learning_rate and sigma at
each iteration
                            default function: lambda
x,current_iteration,max_iter: x/(1+current_iteration/max_iter)
            random_seed, random seed to use.
        """
        if sigma >= x/2.0 or sigma >= y/2.0:
            warn('Warning: sigma is too high for the dimension of the map.')
        if random_seed:
            self.random_generator = random.RandomState(random_seed)
        else:
            self.random_generator = random.RandomState(random_seed)
        if decay_function:
            self._decay_function = decay_function
        else:
            self._decay_function = lambda x, t, max_iter: x/(1+t/max_iter)
        self.learning_rate = learning_rate
        self.sigma = sigma
        self.weights = self.random_generator.rand(x,y,input_len)*2-1 # random
initialization
```

```
        for i in range(x):
            for j in range(y):
                self.weights[i,j] = self.weights[i,j] / 
fast_norm(self.weights[i,j]) # normalization
        self.activation_map = zeros((x,y))
        self.neigx = arange(x)
        self.neigy = arange(y) # used to evaluate the neighborhood function
        self.neighborhood = self.gaussian

    def _activate(self, x):
        #Updates matrix activation_map, in this matrix the element i,j is the 
response of the neuron i,j to x
        s = subtract(x, self.weights) # x - w
        it = nditer(self.activation_map, flags=['multi_index'])        while not 
it.finished:
            self.activation_map[it.multi_index] = fast_norm(s[it.multi_index]) 
# || x - w ||
            it.iternext()


    def activate(self, x):
        #Returns the activation map to x
        self._activate(x)
        return self.activation_map

    def gaussian(self, c, sigma):
        """ Returns a Gaussian centered in c """
        d = 2*pi*sigma*sigma
        ax = exp(-power(self.neigx-c[0], 2)/d)
        ay = exp(-power(self.neigy-c[1], 2)/d)
        return outer(ax, ay)  # the external product gives a matrix

    def winner(self, x):
        """ Computes the coordinates of the winning neuron for the sample x """
        self._activate(x)
        return unravel_index(self.activation_map.argmin(), 
self.activation_map.shape)

    def update(self, x, win, t):
        """
            Updates the weights of the neurons.
            x - current pattern to learn
            win - position of the winning neuron for x (array or tuple).
            t - iteration index
        """
        eta = self._decay_function(self.learning_rate, t, self.T)
        sig = self._decay_function(self.sigma, t, self.T) # sigma and learning 
rate decrease with the same rule
        g = self.neighborhood(win, sig)*eta # improves the performances
        it = nditer(g, flags=['multi_index'])
        while not it.finished:
            # eta * neighborhood_function * (x-w)
```

```
            self.weights[it.multi_index] +=
g[it.multi_index]*(x-self.weights[it.multi_index])
            # normalization
            self.weights[it.multi_index] = self.weights[it.multi_index] /
fast_norm(self.weights[it.multi_index])
            it.iternext()

    def random_weights_init(self, data):
        """ Initializes the weights of the SOM picking random samples from data
"""
        it = nditer(self.activation_map, flags=['multi_index'])        while not
it.finished:
            self.weights[it.multi_index] =
data[self.random_generator.randint(len(data))]
            self.weights[it.multi_index] =
self.weights[it.multi_index]/fast_norm(self.weights[it.multi_index])
            it.iternext()

    def train_random(self, data, num_iteration):
        """ Trains the SOM picking samples at random from data """
        self._init_T(num_iteration)
        for iteration in range(num_iteration):
            rand_i = self.random_generator.randint(len(data)) # pick a random
sample
            self.update(data[rand_i], self.winner(data[rand_i]), iteration)

    def _init_T(self, num_iteration):
        """ Initializes the parameter T needed to adjust the learning rate """
        self.T = num_iteration/2  # keeps the learning rate nearly constant for
the last half of the iterations

    def quantization_error(self, data):
        """
            Returns the quantization error computed as the average distance
between
            each input sample and its best matching unit.
        """
        error = 0
        for x in data:
            error += fast_norm(x-self.weights[self.winner(x)])
        if (len(data)>0):
            return error/len(data),len(data)

from pylab import imread,imshow,figure,show,subplot,title,cm
from numpy import linalg,reshape,flipud,unravel_index,zeros,genfromtxt, empty,
uint16

import os

PathDiv = "/home/ndetos/Documents/2017/unistra/copernicus/vegas/divisions/" #the
path where the images are stored.
```

```
lstFilesDiv = []  # create an empty list to store the files for 6 images

import glob

#put the path of each image in PathDiv, and confirm
for infile in glob.glob( os.path.join(PathDiv, '*-3-3.png') ):#in this instance
image division 3-3 is used to illustrate. The process was repeated for each of
the 16 divisions
    print "current file is: " + infile
    lstFilesDiv.append(infile)


lstFilesDiv.sort()      #to have the images in the order they were taken, i.e.
1984 to 2009
Qerror = []     #an empty list to which the QE values and samples will be
populated into
count = 1
# read an image from the folder created
for filenameDiv in lstFilesDiv:
    img = imread(filenameDiv)
    # reshaping the pixels matrix, to fit SOM requirement
    pixels = reshape(img,(img.shape[0]*img.shape[1],3))

    # SOM initialization and training
    print
    print('training...image number ',count, 'of ',len(lstFilesDiv))

    som = MiniSom(4,4,3,sigma=1.2,learning_rate=0.2) # create a 4x4 = 16 SOM
with 3 features
    som.random_weights_init(pixels) #initialise the SOM, picking values randomly
from the image

    som.train_random(pixels,10000)  #train the SOM, with 10,000 iterations

    #calculate QE
    qe=som.quantization_error(pixels)   # find the image QE value
    print filenameDiv, qe, 'is image QE, total samples'
    Qerror.append(qe)
    count = count + 1

import csv

#reduce Qerror to 4 decimal point, and to float data type
QE, Samples = zip(*Qerror) #to unpack QE,Samples from pairs into lists
err= zip([round(float(i), 4) for i in QE], Samples) # extract the QE values into
err, ignore the samples for now.

print
print('write to a csv file...')

with open('data/Err_div_3-3_testing.csv', 'w') as f:    #QE values for the
divisions 3-3 are written to file named Err_div_3-3. Repeated for each of the 16
```

```
divisions.
    writer = csv.writer(f, delimiter='\t')
    writer.writerows(err)
print('Done')
```