Generation of an Image's Color Palette using a Kohonen Self-Organizing Map

CS51 Final Project Writeup
Spyridon Ampanavos (GSD), Diana Chen (College),
Lezhi Li (GSD), Ting Zhou (College)

Overview

Kohonen Self-Organizing Maps (SOMs) are used to represent high-dimensional data in a two dimensional form. Specifically, we are using an SOM for color quantization, which is the idea of reducing the number of distinct colors in an image. Color quantization has two main applications: displaying image with many colors on devices that can only display a limited number of colors, and image compression. Our general objective was to use python to code an algorithm that would break down the color specifics of an image and generate a discrete color palette from it.

Evaluation

Our core software does what it is supposed to, quantizing and mapping an image's colors, but it has its limitations. First, if the SOM size is chosen to be small, then the SOM may show only dominant colors and is likely to miss colors that are present in small quantities, regardless of how visible the colors are. On the other hand, if the SOM is chosen to be large, then the algorithm takes an exponentially longer time to run. Regarding an extension to our core project, the algorithm that determines which image in the database is most similar to the input image can be improved. Our current algorithm calculates the Euclidean distance between the average R, G, and B values of the input image's SOM and all of the database SOMs. It then returns the image whose SOM is the minimum distance to that of the input image. While this does return the image with the most similar R, G, and B values, it does not account for the positions of colors, so the most similar image determined by this method may look quite different from the input image.

Instruction: See code/README
Below is how the functions listed in the projects specs are achieved:

1. Take an image (jpeg) and generate a numerical color palette based on the colors in the image: achieved
2. Produce a dialog box showing the outputted color palette in non-numerical form (actually show color): achieved
3. Implement a user interface to allow easier image input: not implemented, users can specify input image in "run_som.py".
4. Reproduce the inputted image using the outputted color palette à la Remapped Image: achieved
5. Create a database of images and modify the algorithm to find color similarities between the input image and those in the database and then

output that group of images à la "PicSOM - A Framework for Content-Based Image Database Retrieval using Self-Organizing Maps" : achieved, similarity if determined by the average R, G, Bs of colors in output SOM.

We implemented all of the above functionality except 3 (we edited that out in our final spec)

How good was your original planning?

Our original planning was very high-level - we knew the algorithm and language we wanted to use, but many of the specifics were not lined out until we delved further into the process. However, we believed our goals were realistic and pushed our exploration of the SOM algorithm.

How did your milestones go?

There are roughly 3 milestones that we set for different stages of the project: mathematical functions, core structure, and additional functionalities. After getting a relatively clear picture of how the algorithm is implemented, we started from lower-level functions where most of the mathematical calculations are, as well as basic data structures. After we accomplished that, we went on to structure the code by calling of those functions in appropriate loops. And after finishing the core of our code, we added more functionality such as image recreation and database searching.

What was your experience with design, interfaces, languages, systems, testing, etc.?

Python and algorithm development were both new for all group members. There was a steep learning curve, and as we became more comfortable with the language and clarified the product of our algorithm, the ease of testing and design came with.

What surprises, pleasant or otherwise, did you encounter on the way?

The main unpleasant surprise was how long it took the code to run. A reasonable number of iterations and SOM size took several minutes to finish, creating a lot of waiting time during testing. To work around this without compromising the quality of the SOMs, we compressed the input images to be fewer pixels.

What choices did you make that worked out well or badly?

We chose to reproduce the input image with the resulting SOM (of reduced colours) and that was a success. The reproduced image emphasized the colours observed and dropped by our algorithm and added a great visual to our project.

We also chose to tackle our reach goal, the extension to our SOM algorithm that would return an image from our database which is most similar to the inputted image. However, the algorithm that we used to implement this part has a lot of room for improvement. Based on the input image and it's specific SOM colour scheme, our extension is a hit-or-miss, as it is highly dependent on the images already in the database. Despite its shortcomings, we enjoyed the extension and thought it a well choice because it gave us more information on our algorithm and how the input image and it's respective SOM compares to those already in the database.

What would you like to do if there were more time?

One thing we consider most important for this algorithm is to cluster the output colors from SOM so that the number of colors in the palette can be limited to even fewer colors, maybe around 8-10.

Another thing that could be added to our extension that returns the most similar image would be a better algorithm to measure “similarity”. Ideally it wouldn’t just take the average RGB values of the entire image (which may or may not result in a similar image being returned) but would also be computationally light and fast.

Additionally, it would be interesting to test out different mathematical functions for both our core and extension functions to improve run time and product, respectively.

How would you do things differently next time?

In general, the process of our project went quite smoothly. The most significant trouble we had was at the beginning when we had very little knowledge of both SOM and python. If we were to do similar projects next time, we think it would be better to do more research into the actual implementation of that algorithm and try to get a clear idea at an early stage.

What was each group member's contribution to the project?

Spyridon and Lezhi worked more on side of the classes and core algorithm, while Diana and Ting worked more on the side of the extensions. That being said the four of us often worked together, contributing bits and pieces to core, extension, and testing.

What is the most important thing you learned from the project?

Beyond learning python, the most valuable thing we learned about was the process of coding. Before we had begun coding, the project seemed pretty intimidating: we didn’t know any python and we weren’t even confident that we completely understood the algorithm. Ultimately we didn’t really know where to start, but we had to start somewhere. As a group, we coded the classes and necessary equations, all the while the project slowly became clearer. As we added the algorithm itself, our core project was near completion at the same time the algorithm and extensions became clear to us beyond the conceptual level. Then, adding the extensions and perfecting our algorithm was the fun part. Recognizing that we started at a place where we had an idea but were not sure how to implement it to actually bringing that idea to a working program was the rewarding part. The idea of putting pen to paper and getting started is definitely an important take-away that can be applied not only to further programming but also to attacking and solving problems in general.

Link to video

https://vimeo.com/126572569

----------Draft and Final Specification below----------

Overview

Kohonen Self-Organizing Maps (SOMs) are used to represent high-dimensional data in a two dimensional form. Our general objective is to use python and its machine learning libraries to code an algorithm that will break down the color specifics of an image and generate a color palette from it. Additional objectives include enhancements to ease-of-use and increased functionality.

More specifically, our core goal for the project is to use a Kohonen self-organizing map to analyze an image composed of 3-dimensional vectors (R,G,B) and map these to 2-dimensional space. This is our minimal-complete project, one that takes in images and outputs a color palette through the terminal. Immediate further goals include a developed front-end for inputting images and displaying the output color palette, which would enhance the experience and ease-of-use of the program. Our first 'reach' goal would be the reproduction of the image using the 2-dimensional palette generated by the algorithm, an interesting way to show the accuracy of the algorithm. A second 'reach' goal would involve pre-loading a database of images and outputting images that have similar color signatures to that of the inputted image.

Feature List

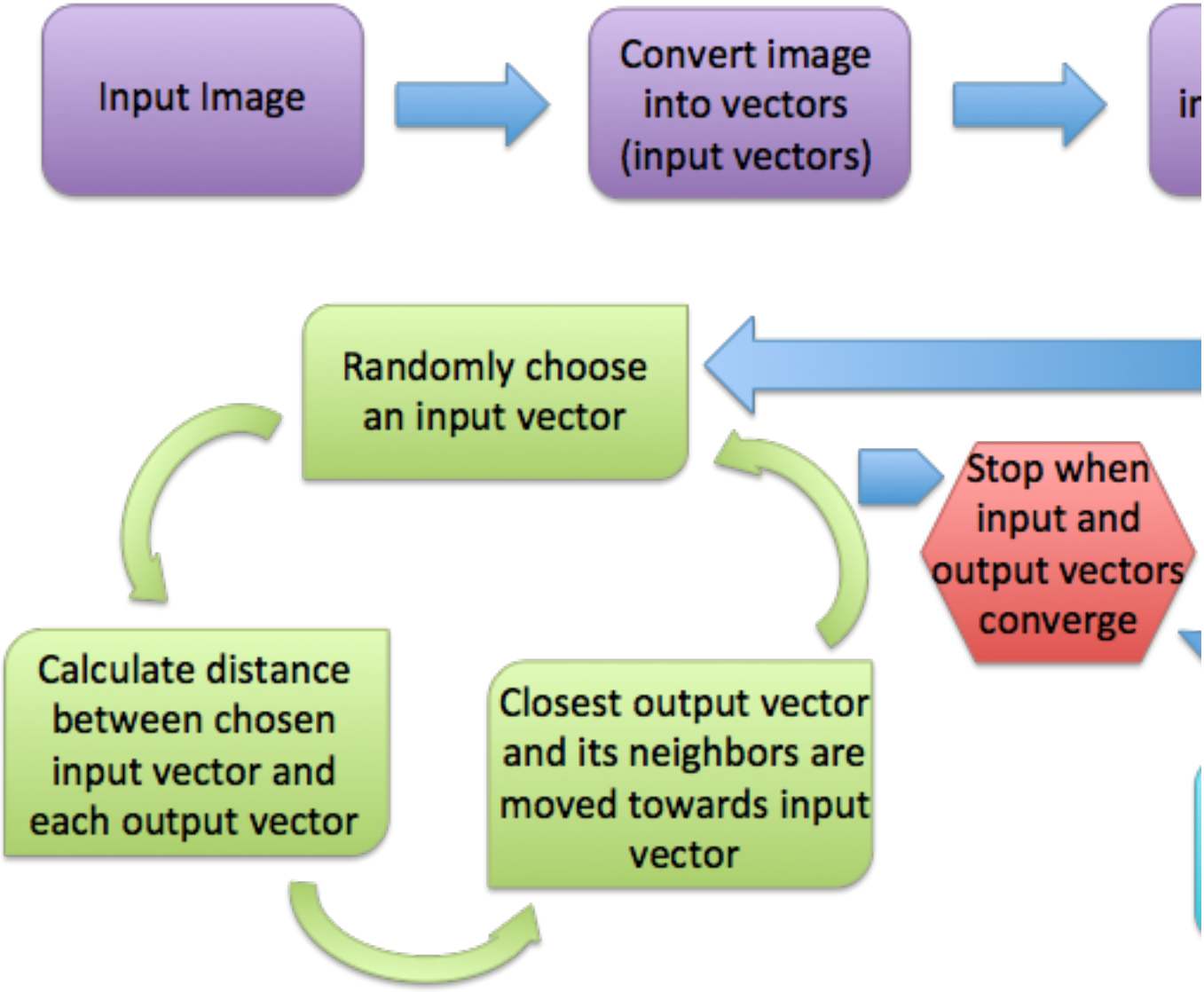
The following list is prioritized from most vital to the project's success to non-critical for core functionality but important nonetheless.

1. Take an image (jpeg) and generate a numerical color palette based on the colors in the image.
    a. This our minimal-complete version of the project.
2. Produce a dialog box showing the outputted color palette in non-numerical form (actually show color).
3. Implement a user interface to allow easier image input.
4. Reproduce the inputted image using the outputted color palette à la Remapped Image.
5. Create a database of images and modify the algorithm to find color similarities between the input image and those in the database and then output that group of images à la "PicSOM - A Framework for Content-Based Image Database Retrieval using Self-Organizing Maps"

The core algorithm was implemented basically as follows: First, we saved the RGB vector of every pixel in the image. At the same time, we initialized an m-by-n array of random RGB values, which we will call the SOM. The 'm' and 'n' are determined by the user and resulting in m-by-n distinct colors. Next, we calculate the euclidian distance between the first pixel's RGB vector and each of the RGB vectors in the randomly initialized SOM, saving that with the minimum distance, which we will call the "winner". This corresponds to being the closest

color-wise. Next, we apply a neighborhood function which updates all of the RGB vectors in the vicinity of the SOM's winner vector. We do this over every pixel N times and produce the final SOM. What was the random SOM is now the final SOM, and its RGB data contains the discrete color palette.
Our two core structures are our Dataset class and Node class, which contains information on our input image and SOM, respectively. The Dataset class contains size of the image and the position and RGB vector of each pixel, while the Node class contains just the position and RGB vector of each pixel. Beyond the classes, our algorithm used mainly arrays, lists, and tuples.

Technical Specification
Language: Python
Libraries: scikit learn …



Picture 1: High Level Flow Diagram of Project

Here is an overview of the SOM algorithm [SOM Image Analysis Tutorial]

1. Convert the images **I** into vectors **X**
2. Initialize the **m** vectors associated with each output node to random values.
3. Present one input vector.
4. Compute the distance d between the input vector **m**(t) and each output node j using: **d = || X(t) - m(t) ||** , where m(t) is the code vector associated to node j, and **X(t)** is the input vector.
5. Select the output node as the node associated with the vector **m**(t) that minimizes **d**.
6. Update code vectors in node j and neighborhood. **m** are updated for node j and all nodes in a neighborhood N whose radius is defined by some decreasing function **h(n)**

   m(t+1) = m(t) + a(t)*h(t,r) (X(t) - m(t))

   where a(t) is a decreasing function that controls the magnitude of the changes with the time (0<a(t)<1).

   The neighborhood **h** is defined as:

   h(n) = exp((-d d) / (2 * r * r)).
7. Repeat from step 2 until process converge.

Modules

In order to implement this algorithm, we will need the following modules:

Input module. The general module that associate a pixel (in cases of color quantization) or an image (in cases of image content recognition) with a vector.

```
module type INPUT =
sig
 type input
 type vector
 val input_vector : input -> vector
end
```

Pixel module. Inside of this module we calculate R, G, B of a pixel. It also returns a certain pixel if x and y coordinates (as related to an image) are given.

```
module type PIXEL =
sig
 type pixel
 val x : int
 var y : int
 val r : pixel -> int
 val g : pixel -> int
 val b : pixel -> int
 val vector_from_pixel : pixel -> (int * int * int * int * int)
end
```

The five “int”s in “vector_from_pixel” function are x, y, r, g, b. Because pixels with similar x and y coordinates tend to fall in the same color category, and those with similar r,g,b tend to fit in the same category (not sure).

Image module. The purpose of this module is to transform an image into a vector that describes the characteristics of this image so as to be inputted into the SOM algorithm.

```
module type IMAGE =
sig
 type pix
 type img
 val find : (int * int) -> pix
 val vector_from_img : img -> (int * int * int * int * ...)
end
```

(Not quite sure about “vector_from_img”. From my understanding, if we want to categorize colors, we will need a vector from each pixel; only if we want to categorize images will we need a vector from each image. If we want to calculate vector_from_image, we will need vetor_from_pixel.)

```
module PxImage (P : PIXEL) : (IMAGE with type elt = P.pixel)
struct
  type pix = P.pixel
  type img = “array of elt”
end
```

Node module. Inside of this module we calculate output vector of a node. It also returns a certain node if x and y coordinates (as related to a map) are given.

```
module type NODE =
sig
 module I : INPUT
 type input = I.vector
 type node
 val x : int
 val y : int
 val randomize : unit -> (int * int * int)
 val compute_distance : input -> node -> float
 val update : int -> node -> input -> (int * int * int)
 val output_vector : node -> (int * int * int)
end
```

The “int” in the “update” function corresponds to “t” in the algorithm description above (not sure).

Map module. The output map that contains all the vector information in its nodes.

```
module type MAP =
sig
 module N : NODE
 type n = N.node
 type map
 val width : int
 val height : int
 val find : (int * int) -> node
 val neighborhood : int -> map -> node -> ((int * int) * (int * int))
 val update_map : int -> map -> map
end
```

The “neighborhood” function calculates the neighborhood of a node in a map. The first “int” also corresponds to “t” in the algorithm description above. So does the “int” in the “update_map” function (not sure).

Next Steps

To do before the final technical specification is submitted:

1. Everyone should have python installed.
2. All necessary modules, whether math-, image-, or machine-learning-related, should be determined and downloaded.
3. Familiarize ourselves with the language and libraries and begin our work on the algorithm.

Detailed Description

Modules/Signatures/Actual Code

From classes.py

```
from PIL import Image

class Dataset:
  def __init__ (self, filename):
     im = Image.open(filename,'r') #Can be many different formats.
     pix = im.load()
     im.show()

     self.size = im.size
     self._vectors = []
     self._labels = []

     for j in xrange(im.size[1]):
        for i in xrange(im.size[0]):
           self._vectors.append(pix[i,j])
           self._labels.append((i,j))

  def __len__ (self):
     return len (self._vectors)
```

```
    def __str__ (self):
        return str (self._vectors)

    def __getitem__ (self, key):
        return (self._vectors[key], self._labels[key])

    def size (self):
        return self.size

    def vectors (self):
        return self._vectors

    def labels (self):
        return self._labels

    def get_vector (self, point):
        return self._vectors[point]

    def get_label (self, point):
        return self._labels[point]

class Node:
    def __init__ (self, _v, _x, _y):
        self.v = _v
        self.x = _x
        self.y = _y

    def __str__ (self):
        return str (self.v)

    def set_v (self, v):
        new_v = ()
        for i in xrange(len(v)):
            new_dim = int(round(v[i]))
            new_v = new_v + (new_dim,)
        self.v = new_v

    def get_v (self):
        return self.v

    def set_pos (self, pos):
        self.x = pos[0]
        self.y = pos[1]

    def get_pos (self):
        return (self.x, self.y)
```

From functions.py

```
# set max possible distance between two colors
max_dist = 3 * 255 * 255

# set parameters
N = 5;
```

```
neuron_w = 4
neuron_h = 4 # (neuron_w + neuron_h) must be greater than or equal to 8
som_dim = 300

# initial learning rate
a0 = 0.1

# initialize randomized SOM
def initialize_som():
  out_map = []
  for i in range(neuron_h):
     out_map.append([])
     for j in range(neuron_w):
        temp_col =classes.Node((random.randint(0,255), random.randint(0,255),
random.randint(0,255)), j, i)
        out_map[i].append(temp_col)
  return out_map

# define neighborhood distance
def neighborhood(d2, r2):
  return math.exp(-d2 / (2 * r2))

# calculate euclidean distance between RGB triples
def vec_distance(pv, n):    #pv: pixel vector (to be called with get_vector) , n: node
  return math.sqrt( (pv[0]-n.v[0])*(pv[0]-n.v[0]) + (pv[1]-n.v[1])*(pv[1]-n.v[1]) + (pv[2] -
n.v[2])*(pv[2] - n.v[2]))

# determine SOM node closest in color to image pixel vector
def winner_node(pv, som):
  d_min = math.sqrt(max_dist)
  for column in som:
     for n in column:
        if vec_distance(pv, n) < d_min:
           winner = n
           d_min = vec_distance(pv, n)
  return winner

# calculate distance between SOM nodes
def sq_node_distance(n0, n1):
  return (n0.x - n1.x) * (n0.x - n1.x) + (n0.y - n1.y) * (n0.y - n1.y)

# update SOM node's RGB values
def update_node(pv,wn,cn,t):
  _r = radius(t)
  _r2 = _r * _r
  _d2 = sq_node_distance(wn,cn)
  if (_d2 < _r2):
     _a = a(t)
     _n = neighborhood(_d2,_r2)
     v = ()
     for i in xrange(len(pv)):
        new_dim = cn.get_v()[i] + _a * _n * (pv[i]-cn.get_v()[i])
        v = v + (new_dim,)
     cn.set_v(v)
```

```
# update all of SOM's RGB values
def update_som(som,pv,wn,t):
  for column in som:
     for cn in column:
        update_node(pv, wn,cn,t)

# calculate radius
def radius(t):
  return (neuron_w + neuron_h)/4 * math.exp(-t / (N / math.log( (neuron_w + neuron_h)/4)
))

# time-varying learning rate
def a(t):
  return a0 * math.exp(-t / N)

# train the SOM
def train (data,som):
  for t in xrange(N):
     draw(som)
     for i in xrange(len(data)):
        print "t: "+str(t)+", "+"i: "+str(i)
        pixel_v = data.get_vector(i)
        win_n = winner_node(pixel_v, som)
        update_som(som, pixel_v, win_n, t)
  print "training finished"

# display the SOM
def draw (som):
  for i in range(neuron_w):
     for j in range(neuron_h):
        print som[j][i]
  som_im = Image.new('RGB', (neuron_w,neuron_h), None)
  som_val = som_im.load()
  for i in range(neuron_w):
     for j in range(neuron_h):
        som_val[i,j] = som[j][i].get_v()
  som_im2 = som_im.resize((som_dim,som_dim))
  som_im2.show()
```

Timeline

Sunday April 19:

- Fully complete final spec to be submitted to TF

Wednesday April 22:

- Have written working python code for input, pixel, image, and node modules
- Begin putting the modules together towards a working SOM

Friday April 24:

- Complete base project (1) or be very near completion of base project (within the weekend) and begin adding features

Wednesday April 29:

- Produce dialog box showing the outputted color palette in non-numerical form (2)
- Create/Import database of images (3)
- Modify algorithm to find color similarities between the input image and those in the database and then output that group of images (3)
- Reproduce inputted image using the outputted color palette (4)

Friday May 1:

- Test functionality and corner cases comprehensively with iterations through each potential path
- Format code/Make sure style is clean
- Submit!

Progress Report

TF added to Github to see files. Have core functionality done.

Ting: (765)432-6444
Spyros: 8579991541
Diana: 9788772830
Lezhi: 617 852 4626

Useful resource:

SOM image analysis tutorial
: http://xmipp.cnb.uam.es/Xmipp/chapter7/index.html

SOM tutorial :
http://www.ai-junkie.com/ann/som/som1.html

mnist handwritten data-set