Name: Junchao Wu (junchao3)

CS 445 – Final Project Report

Implementation of Image Analogies

1. Motivation and Impact

Image analogies is a popular topic in computational photography. In recent years, I saw amazing deep learning algorithms and applications on tech news that could produce famous artists' style paintings from normal photos. It surprised me when I found this paper, a 20-year-old algorithm could do the similar job compared to these cutting-edge machine learning algorithms. Therefore, I decided to implement this paper, and see how it works. I also think this topic has more general importance. For example, it is unrealistic and time-consuming to find specific algorithm for each image transition, and some transition is also hard to describe by human, so using such an analogy algorithm is a good choice in this situation. Meanwhile, the public demand for automatic photo processing and creative filters has been high as the social media became an important part of daily life. Image analogies make it simple to create many stylized photo filters on photos.

2. Approach

Generally, I followed the pseudocode on paper. To begin with, I converted the RGB images to YUV color space. That's because only luminance channel is used in synthesis process. Then, the luminance channel is remapped if the image is used in artistic filter, as paper does. The formula is $Y(p) \leftarrow \frac{\sigma_B}{\sigma_A}(Y(p) - \mu_A) + \mu_B$, where μ_A and μ_B are mean luminance and σ_B and σ_A are standard deviations of luminance. $Y(p)$ is the luminance of a pixel. Then, I generated 2 level pyramids (3 layers) for image A, A', B, and copy B to B'. Then, for level 1 and 2 image of pyramid A, A', B, I calculated the feature vectors for each pixel, and stored them in variables "a/ap/b_features". The feature vector of a pixel is its 5x5 neighboring pixels' luminance channel and its corresponding lower-level pixel's 3x3 neighboring pixels' luminance channel. The images are padded by 1 or 2 pixels, so pixels on edge can have 3x3 and 5x5 neighbors as well. Then, I called ann function with A and A"s feature vectors to generate ANN structure. It also returns $F_l(p)$, the concatenation of all the feature vectors within p's neighborhood for both A and A' at both current fine level l and lower coarse level $l-1$.

With these preparations, now it's time to start the Bestmatch process. From coarsest to finest level of pyramid, for each pixel q in image B', I calculated its best approximate match and best coherence match and got two best match pixels $p_{app}\ and\ p_{coh}$. Best approximate match is calculated by calling related third-party package function, and for best coherence match, a formula is given by the paper. That is $s(r^*) + (q - r^*)$, where $r^* = arg\min_{e \in N(q)} \|F_l(s(r) + (q - r)) - F_l(q)\|\text{^}2$, and $N(q)$ is the neighborhood of already synthesized pixels neighboring to q in b'_l. $F_l(q)$ is also updated in each iteration after previous q is fixed. After checking some edge cases, I used these two pixels to calculate d_{app} and d_{coh}, the norm $\|F_l(p) - F_l(q)\|\text{^}2$ as a weighted distance over the feature vectors $F(p)$ and $F(q)$. In order to keep the coherence term consistent at different scale, a factor of $k * 2\text{^}(l - L)$ is applied on d_{coh}, L is the number of layers. According to the paper, the larger the value of k, the more coherence is favored over accuracy in the synthesized image.

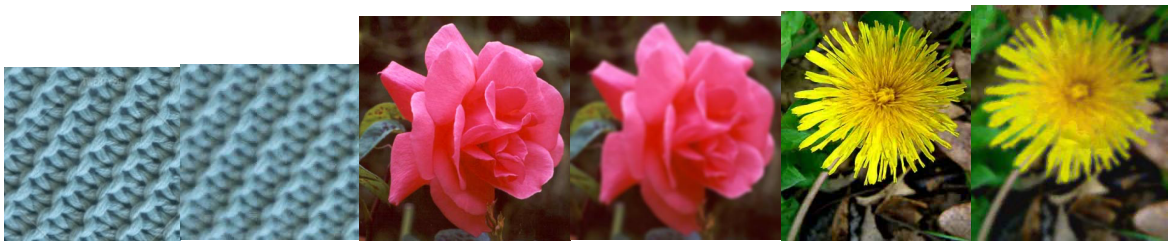
By comparing d_{app} and $d_{coh} * k * 2\text{^}(l - L)$, I can decide which p to use. The location of p is

stored in $s_l(q)$, and $s_l(q)$ is used when calculating best approximate match and best coherence match. p's Y channel is copied to q. For color channels U and V, in most cases, we use the pixel at same location in B to get the color information. However, for re-color filter and texture-by-numbers filter, since we use color from image A and A', we copy all three channels from p. After all loops finished, the finest level of pyramid B' is the output image. It also needs to be converted back to RGB color space.

3. Results

(Image from left to right in each line is A, A', B, B' if not specifically mentioned)

1. Identity filter and blur filter



These results are used to check if the algorithm works properly. The left most image is used as A, A', B in identity filter case. The results are as expected, although A' in Identity filter case is a bit blurred. K for Identity filter is 0 and second one is 0.5.

2. Texture Transfer

The results are surprisingly good. The right most three images are generated with different parameter k=0.5, 1, 2. The higher k corresponding to more obvious texture from A.

3. Super-resolution

The results are bad. In first fluff case, the result is even blurrier than the original B. In flower case, the leaves on left-up corner are upscaled successfully, but for the yellow flower, although high frequency details are added to it, it is not correct. K=2 in both cases.

4. Re-color gray image

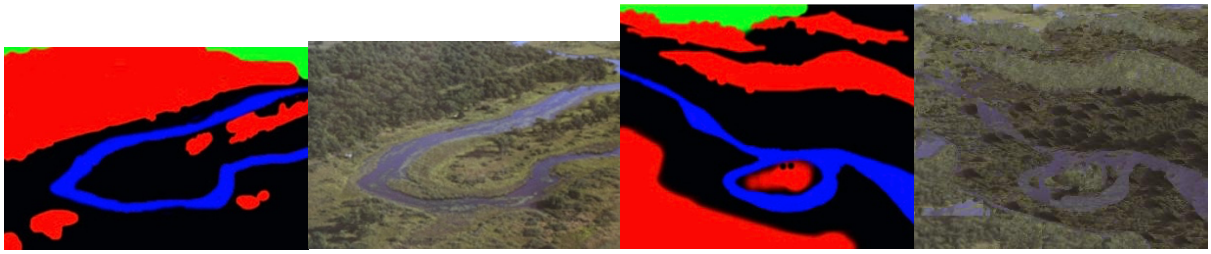
The result meets my expectation. Color is successfully mapped by luminance.

5. Artistic filters

The results are good. I also used this case to discover the influence of different k, and how luminance remapping influenced the result. Third and fourth images have k=0.5, but third one is disabled luminance remapping, and I found there are unnatural white pixels in the third image's sky area. Fourth image also has smoother oil painting texture compares to image 3. For image 4,5,6, the corresponding k parameters are k=0.5, 5, 20. As k goes higher, the intensity from A' has more obvious impact on B'. A good range for k in artistic filters should be 0.5<=k<=2.

This is a failure case. The intensity from A' influenced B' completely. This image is generated with k=2. I also tried k=0.5, but the first level output shows a similar result.

6. Texture-by-numbers



The result is great with k=1.5. Similar river and land with expected shape B'.

4. Implementation details

The functional codes are written in Python, and using Jupyter notebook to generate images.
To increase the speed of generating images, I divided the Jupyter notebook into 4 pieces and run them on multiple computers. The detail about each Jupyter notebook contains can be found in README.md.
Third-party packages:
"matplotlib" for image display. "cv2" for reading and converting images between different color spaces. "numpy" for computation. "imageio" for write image onto disks. "pyflann" for ANN search algorithm.
External resources:

1. This article explains the paper in a more readable way. It also includes some examples, which gives me a general concept of how the results could look like. I also used some images from this repository.
   Link: http://jmecom.github.io/projects/computational-photography/image-analogies/
2. Original paper: Hertzmann et al., "Image analogies," SIGGRAPH 2001.
   I also cut some images from the paper.
3. Photoshop is used to generate some images A'.

5. Challenge / Innovation

I would give my projects 20/20 points for this section. There are many challenges to implement it, especially for a junior student. Firstly, in the paper, the authors didn't talk much about the details of two matching algorithms, but when I tried to implement these algorithms, I found it is not easy. For ANN algorithm, I need to find an efficient third-party package and understand how it works, and for best coherence matching, I need to implement it based on the one-line formula in the paper. Besides, it is hard to locate bugs happened in two matching algorithms, as I have no way to tell whether the returned pixel locations are correct except getting the output image, and then infer the reasons to solve bugs. Secondly, the paper didn't talk much about the selection of k parameter, so I need to explore it by myself.

Both challenges require many attempts to run the codes. However, here is the biggest challenge in this project: the extremely long running time of the algorithm. Although I was aware about it when I read the paper and articles, but when I started writing the code, I found it really increased the difficulty of debugging codes and tune the parameters. On my laptop, it takes about 2.5 hours to generating 90000 pixels' image and cannot take advantage from multi-core CPU. To ease the problem, I print the level 1 image from (B prime)'s image pyramid, and it can provide a rough assumption about the actual output image. Therefore, I can decide whether to terminate the calculation based on that, and it saved me much time. Moreover, I asked my friends to build remote virtual machine for me so I can use there computing resource to run multiple images at same time, although it added more difficulty on file management and syncing codes.

6. Extra figures

Three more successful cases. Sunset one with k=0.5. Flower one with k=1. Leaves one with k=1.3.