Title: 이미지의 Subsampling과 Color Segmentation을 활용한 틀린 그림 찾기

Author: 중앙대학교 예술공학대학 컴퓨터예술학부 20194004 양소영

Teaser Figure (Overall concept, target problem, etc.)

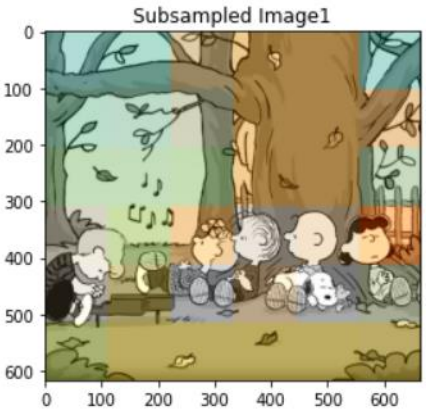
[Original Image]

[Change Image]

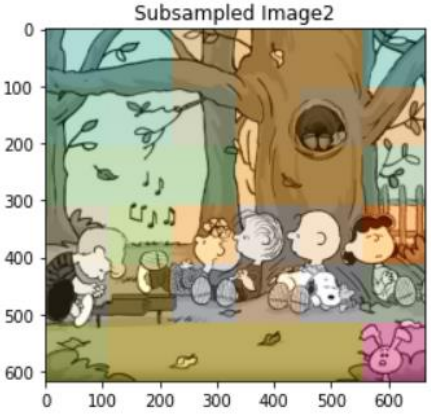
원본 이미지와 비교 이미지의 다른 부분을 찾아 표시해주는 알고리즘을 개발하는 것이 목표이다. 두 이미지의 다른 부분은 나무 가운데의 구멍과 왼편의 울타리, 그리고 우측 하단의 분홍색 토끼 세 곳이다.

Framework Figure

1. 두 이미지 Subsampling



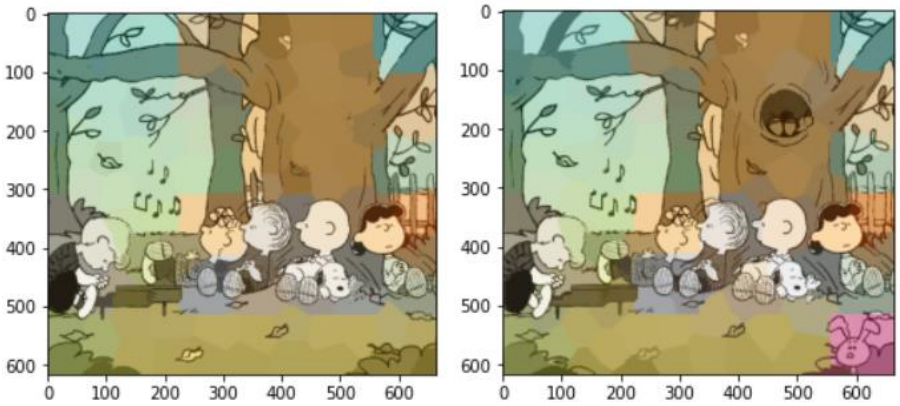
[Original Image]



[Change Image]

이미지를 clustering하기 전에, 비교할 때 아주 작고 세밀한 부분까지 다룰 필요가 없기 때문에 해상도를 낮추기 위해서 적절한 만큼 subsampling 과정을 진행한다.

2. 두 이미지 Clustering-based Color Segmentation

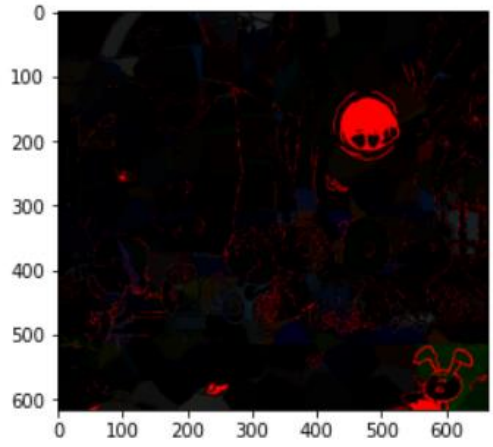


[Original Image] **[Change Image]**

두 이미지의 다른 부분이 색상적으로 아주 큰 차이가 없기 때문에, 근접한 픽셀들의 위치를 고려하며 보간해서 대표적으로 하나의 색상만을 나타내도록 clustering을 진행한다.

Experiment Figure (qualitative results, ablation tests, analysis, etc.)

3. Threshold Result



두 이미지의 차이를 구하는 함수를 사용하여 차이가 있는 곳, 즉 이미지 간의 다른 부분은 한 눈에 알아볼 수 있도록 빨간색으로 표시한다.

General description of your target problem

서로 다른 두 장의 이미지를 비교하여 색상 차이를 기반으로 다른 부분을 찾으려고 함.

The limitation of the target problem that you want to solve

이미지의 segmentation을 기반으로 둘 사이의 색상을 비교하여 차이를 얻고 싶은데 이를 구현할 수 있는 방법을 찾지 못하여 처음 구상 방향과 다른, 이미지의 canny edge를 비교하는 방향으로 바꿔야 할 지 고민이 되었다. 많은 시행착오 끝에 OpenCV의 함수를 찾는 도중 'subtract'라는 함수를 발견하여 문제를 해결할 수 있었다.

The solution you've found to solve the limitation

처음 목적은 두 이미지의 해상도를 낮추고 segmentation마다 색상 차이를 구한 후, 서로 다른 부분은 한 눈에 알아볼 수 있도록 해당 영역에 빨간색으로 표시하고자 하였다. 이미지의 clustering-based color segmentation을 진행하고 각 segmentation을 기반으로 색상을 비교하여 차이를 얻기 위해 subtract 함수를 이용하였는데, 이미지를 배열화 하여 두 이미지 간의 색상 차이를 구한 다음 threshold를 지정하여 차이가 나는 부분을 추출할 수 있도록 하였다.

The detailed description of your framework

제일 먼저 기본이 되는 original image를 불러와 1/150으로 CrCb 채널을 subsampling하도록 한다. RGB를 YCrCb로 변환해주기 위해 cvtColor 함수를 이용하였으며, 각 채널별로 matrix를 분리하여 해상도를 줄이고 본인과 가까운 픽셀들과 색을 합쳐 주기 위해 근처 픽셀끼리 보간하는 resize 함수의 INTER_NEAREST 파라미터를 이용한다. 각 채널들을 취합하고 BGR 이미지로 변환하면 해상도가 낮은 이미지를 확인할 수 있다. 비교 대상이 되는 change image도 같은 방법으로 진행해준다.

다음으로는 각 이미지마다 위치 정보를 포함한 Clustering-based Color Segmentation을 추출하도록 한다. 군집화 클래스를 만들 수 있는 MiniBatchKMeans 라이브러리를 불러오고, 클러스터의 개수는 150으로 지정해준다. 각각의 h와 w값은 이미지의 height, width를 가져오게 하고 reshape 함수를 사용하여 이미지를 배열 형식으로 바꿔준다. 각 픽셀의 색상이 비슷한 것들끼리 위치 기반으로 묶어주려면 x, y 위치 값을 추가해주어야 한다. 이미지의 width, height의 개수만큼 모든 픽셀의 위치를 처음부터 끝까지 추가해주어야 하기 때문에 arrange 함수로 최솟값과 최댓값을 이미지의 height, width 순으로 설정해준다. 그 후에는 x, y값이 포함되도록 픽셀의 위치를 정의해야 하므로 meshgrid 함수로 이미지 픽셀의 위치 값을 직사각형 격자 모양의 행렬로 생성한다. 그러면 새로운 vector 행렬로 만들어진 것을 hstack 함수로 행렬의 요소들을 더하면 각 픽셀의 색상 정

보에 위치 좌표까지 더해진다. 미리 지정해 둔 클러스터의 개수만큼 MiniBatchKMeans 알고리즘에 입력시키면 각 cluster의 가운데 값을 대응시켜 세로 줄 형식의 matrix를 다시 reshape 함수에 삽입하여 이미지 형태의 배열로 바꿔준다. 이로써 위치와 색상 둘을 고려한 clustering이 실행되었다.

마지막으로 subtract을 이용하여 이미지 연산을 진행한다. 이미지 연산은 하나 또는 둘 이상의 이미지에 대해 수학적인 연산을 수행한다. 이 뺄셈 함수는 배열과 배열 또는 배열과 스칼라의 요소별 차를 계산해주는 역할을 하기 때문에 원본 이미지에서 비교 이미지를 빼어준다.[1] 차이 값을 gray 이미지로 변환시킨 후 threshold 함수에 넣어주는데, 이 때 thresholding은 바이너리 이미지를 만드는 가장 대표적인 방법이다. 바이너리 이미지(binary image)란 검은색과 흰색으로 표현한 이미지를 의미하고, 이미지를 어떤 임계점을 기준으로 두 가지 부류로 나누도록 한다.[2] Gray 이미지를 파라미터에 넣고 threshold 임계 값을 0, 임계 값 기준에 만족하는 픽셀에 적용할 값을 255, 마지막 파라미터로 threshold 적용 방법을 THRESH_BINARY_INV(픽셀 값이 임계 값을 넘으면 0, 넘지 못하면 255로 지정) 및 오츠(OTSU)의 알고리즘으로 지정해준다. 오츠 알고리즘은 임계 값을 임의로 정해 픽셀을 두 부류로 나누고, 두 부류의 명암 분포를 구하는 작업을 반복한다. 모든 경우의 수 중에서 두 부류의 명암 분포가 가장 균일할 때의 임계 값을 사용하므로 어차피 threshold 함수에 앞서 전달했던 파라미터는 모두 무시된다. 첫번째 결과인 ret은 thresholding에 사용한 임계 값이고, 두번째 결과인 mask는 thresholding이 적용된 binary image이다. 결국 mask에서 차이 값이 없는 부분이 아니면 빨간색으로 표시하도록 이미지를 보여준다. 그리하여 최종적으로 원본 이미지와 다른 부분이 나타나게 된다.

처음에는 원본 이미지와 비교 이미지의 다른 빨간색 부분이 비교적 많이 나타나 몇몇 값들을 조절해야 할 필요가 있었다. 원래는 두 이미지의 해상도를 1/100로 낮추고 cluster의 개수를 100개로 지정하였는데, 여러 값들을 넣어보면서 해상도를 1/50으로 낮추기도 해보고 cluster의 개수를 200개로 늘려 보기도 하였다. 결국 해상도는 1/100으로, cluster의 개수는 150으로 지정하였을 때가 최선인 것 같아 이 값들로 진행하였다.

이미지 간의 차이를 계산하는 함수를 단번에 찾기도 쉽지 않았다. 원하는 방향대로 구현을 할 수 없을까 봐 plan B로 canny edge detection 방법을 생각해 두었다. 하지만 기존의 구상 방향에 계속 미련이 남아 구글에서 여러가지를 검색해 보다가 최적의 함수를 찾을 수 있었다.

코드는 Computer_Vision_HW4의 Subsampling 코드와 Clustering-based Segmentation 코드를 사용하였고, 마지막에는 외부 코드를 참고하였다.

---

[1] OpenCV 강좌-이미지 연산: https://076923.github.io/posts/C-opencv4-12/

[2] Open CV-Thresholding: https://bkshin.tistory.com/entry/OpenCV-8-%EC%8A%A4%EB%A0%88%EC%8B%9C%ED%99%80%EB%94%A9Thresholding

Experimental results

결과적으로 원본 이미지와 비교 이미지가 다른 부분은 오츠의 이진화 알고리즘(Otsu's binarization method)에 의해 표시될 수 있었다. 내가 알고리즘을 설계할 때 가장 고민이 되었던 부분이 '두 이미지의 차이를 어떻게 구할 것인지, 차이를 구해서 어떻게 나타낼 것인지' 였는데 thresholding에 대해 더욱 자세히 조사하는 도중에 오츠의 알고리즘을 발견하게 되었다.

서로 다른 부분을 포함하여, 기존의 비슷한 두 이미지의 차이는 0아니면 255이다. 바이너리 이미지를 만들 때 가장 중요한 점은 임계 값을 얼마로 정하느냐이다. 오츠의 알고리즘은 임계 값을 임의로 정해 명암 분포를 구하는 것을 반복하는데, 최종으로 명암 분포가 가장 균일할 때의 임계 값을 선택한다. 오츠의 알고리즘을 적용하기 위해서는 threshold 함수의 마지막 파라미터로 THRESH_OTSU를 전달하기만 하면 된다. 오츠의 알고리즘이 최적의 임계 값을 자동으로 찾아준다는 장점이 있지만, 모든 경우의 수에 대해 조사해야 하므로 속도는 조금 느리다는 단점이 있다.

내가 사용한 방법은 어떤 임계 값을 정한 뒤 픽셀 값이 임계 값을 넘으면 255, 넘지 않으면 0으로 지정하는 전역 thresholding이었다. 이 전역 thresholding이 매번 좋은 성능을 내는 것은 아니고, 원본 이미지에서 조명이 일정하지 않거나 배경 색이 여러 개인 경우에는 하나의 임계 값으로 선명한 바이너리 이미지를 만들어 내기 어려울 수도 있다. 이런 경우는 이미지를 여러 영역으로 나눈 뒤 그 주변 픽셀들을 활용하여 임계 값을 구해야 하는데, 이를 적응형 thresholding (Adaptive Thresholding)이라고 한다. 내가 가져온 이미지는 일러스트 그림이라 조명이 일정하지 않거나 배경 색이 단조롭기 때문에 이 방식을 사용하지는 않았다. 대부분의 이미지는 그림자가 있거나 조명 차이가 있기 때문에 전역 thresholding보다 적응형 thresholding을 많이 사용한다고 한다.

Conclusion & Future works

처음 구상했던 순서인 '두 이미지의 subsampling→ clustering→ 차이 계산→ 다른 부분 표시' 알고리즘 해결 방향에 따라서는 적절히 수행한 것 같다.

초반에 이미지를 subsampling할 때 해상도를 잘 맞추거나, clustering할 때 cluster의 개수를 잘 조절하면 노이즈가 덜 발생할 수도 있을 것 같다. 또한 두 이미지가 나란히 합쳐 있었던 원본 사진을 다운받아 직접 그림판으로 그림을 분리했기 때문에 픽셀의 위치에 약간의 오차가 발생했을 가능성도 있다고 본다. 울타리를 제외하고 원하는 큼지막한 부분들은 찾아냈지만 완벽하고 깔끔하게 나오지는 못해서 조금의 아쉬움이 남는다.

이를 개선하기 위해서는 완벽하게 같은 두 이미지에서 한 쪽에만 차이를 두기 위해 몇 가지를 합성하여 있는 그대로를 비교하는 방식이 나을 것 같기도 하다.