IMVFX hw1

資工科碩 312551080 紀軒宇

1. Code

For the coding part, we have separated to few sections, "Find KNN for the given image", "Compute the affinity matrix A and all other stuff needed", "Solve for the linear system" and "Composition".

1.1 Find KNN for the given image

For this part, we have define the feature vector first before finding KNN, the code below have been seperated into two main function `get_features()` and `get_knn()`.

```
X = get_features(image_RGB, mode=features_mode)

n_jobs = 8
indices = get_knn(X, X, n_neighbors=n_neighbors, n_jobs=n_jobs)
```

First look in the function `get_features()` , this function takes two parameters

- `image_RGB` : An normalized RGB image
- `mode` : to specify feature vector representation
  - `HSV` for $X(i) = (cos(h), sin(h), s, v, x, y)_i$
  - `RGB` for $X(i) = (r, g, b, x, y)_i$
    Here `h`, `s`, `v` is pixel's value from an HSV color space image and `r`, `g`, `b` from RGB color space image, and `x`, `y` for each pixel's coordinates in the image.

In the function, we also normalized each feature to reduce the impact of different scales from different feature.

```
def get_features(image_RGB, mode='HSV'):
    # assume image is in RGB mode
    [h, w, c] = image_RGB.shape
    # get x, y coordinates and normalize
    x, y = np.unravel_index(np.arange(h * w), (h, w))
    x = x.reshape(-1, 1)
    y = y.reshape(-1, 1)
    x = x / w
    y = y / h
    feature_vec = np.concatenate(([x, y]), axis=1)
    if mode == 'RGB':
        feature_vec = np.concatenate((feature_vec, image_RGB.reshape(-1,
3)), axis=1)
    elif mode == 'HSV':
        # convert RGB to HSV
        image_HSV = cv2.cvtColor(image_RGB.astype(np.float32),
cv2.COLOR_RGB2HSV)
        H, S, V = cv2.split(image_HSV)
        H_cos = np.cos(H * 2 * np.pi).reshape(-1, 1)
        H_cos = (H_cos-(-1))/(1-(-1))
```

```
        H_sin = np.sin(H * 2 * np.pi).reshape(-1, 1)
        H_sin = (H_sin-(-1))/(1-(-1))
        S, V = S.reshape(-1, 1), V.reshape(-1, 1)
        feature_vec = np.concatenate((feature_vec, H_cos, H_sin, S, V),
axis=1)
    else:
        raise NotImplementedError
    return feature_vec
```

Then move to the second function `get_knn()`, this function takes four parameters

- `feature_vec` : the feature vectors for sklearn's NearestNeighbors to fit
- `obj_feature_vec` : the object feature vector to find
- `n_neighbors` : number of neighbors
- `n_jobs` : The number of parallel jobs to run for neighbors search

```
def get_knn(feature_vec, obj_feature_vec, n_neighbors=10, n_jobs=8):
    knn = sklearn.neighbors.NearestNeighbors(n_neighbors=n_neighbors,
n_jobs=n_jobs).fit(feature_vec)
    _, indices = knn.kneighbors(obj_feature_vec)
    return indices
```

1.2 Compute the affinity matrix A and all other stuff needed

For this part, we first compute affinity matrix $\mathcal{A}$, which is

$$\mathcal{A} = [k(i,j)]$$
$$k(i,j) = 1 - \frac{||X(i) - X(j)||}{C}$$

k is kernel function, for X is the feature vector we compute last step, i, j means the row and column respectively, and C is the number to normalize $||X(i) - X(j)||$ to $[0, 1]$, here we use `X.shape[1]` to get the number of feature types to normalize it.

```
i = np.repeat(np.arange(h * w), n_neighbors)
j = indices.reshape(-1)
k = 1-np.linalg.norm(X[i] - X[j], axis=1)/X.shape[1]
A = scipy.sparse.coo_matrix((k, (i, j)), shape=(h * w, h * w))
```

Next, we compute Laplacian L, which is deifne as

$$L = \mathcal{D} - \mathcal{A}$$

where $\mathcal{D} = \text{diag}(\mathcal{D}_i)$ and $\mathcal{D}_i = \sum_j k(i,j)$

```
D_non = scipy.sparse.diags(A.sum(axis=1).flatten().tolist()[0])
L = D_non - A
```

Finally, we need to compute for the optimal solution

$$H^{-1}c = (L + \lambda D)^{-1}(\lambda v)$$

where $D = \text{diag}(m)$ and m is a binary vector of indices of all the pixels marked by user, v is a binary vector of pixel indices marked as foreground by user.

```
# (L+λD)α = λv
# m is a binary vector of indices of all the marked-up pixels
m = (foreground+background).flatten()
# where D = diag(m)
D = scipy.sparse.diags(m)

# where v is a binary vector of pixel indices corresponding to user markups
for a given layer
v = foreground.flatten()
# Note that H = 2(L+λD) and c = 2λv
H = 2 * (L + my_lambda * D)
c = 2 * (my_lambda * np.transpose(v))
```

1.3 Solve for the linear system

For the last step of computing α, we solve the optimal solution $\alpha = H^{-1}c$ by using module `scipy`'s `scipy.sparse.linalg` since the H is sparse, for normal case we use `scipy.sparse.linalg.spsolve` to compute α, since it may have no exact solution exists, so for this situation, we use `scipy.sparse.linalg.lsqr` and take the first element of return value for the solution, finally, we clip the values into interval of $[0, 1]$ and reshape to `image_RGB`'s size.

```
warnings.filterwarnings('error')
alpha = []

try:
    alpha = scipy.sparse.linalg.spsolve(H, c)
    pass
except Warning:
    x = scipy.sparse.linalg.lsqr(H, c)
    alpha = x[0]
    pass
alpha = np.clip(alpha, 0, 1)
alpha = alpha.reshape(h, w)

```

1.4 Composition

The last step is to composite the foreground image into another background according to the alpha we compute before by the equation

$$I = \alpha F + (1 - \alpha)B$$

where F, B stands for foreground, background image respectly.

We get the alpha from previous step, then extend it to 3 channel array to compute to RGB image, for the new background image we want to compose, we resize it to the size of foreground to fit foreground.

```
alpha = knn_matting(image_RGB, trimap, features_mode=features,
n_neighbors=n_neighbors)
```

```
alpha = alpha[:, :, np.newaxis]
alpha3 = np.repeat(alpha, 3, axis=2)

bg_image = cv2.imread('./background/{}.png'.format(bg_image_name))
bg_image = cv2.resize(bg_image, (image.shape[1], image.shape[0]),
interpolation=cv2.INTER_CUBIC)
bg_image_RGB = cv2.cvtColor(bg_image, cv2.COLOR_BGR2RGB)

# I = αF + (1 − α)B
result = []
result = alpha3 * image_RGB + (1 - alpha3) * bg_image_RGB
result = result.astype(np.uint8)
result_BGR = cv2.cvtColor(result, cv2.COLOR_RGB2BGR)
cv2.imwrite('./result/{}_{}_k{}_{}.png'.format(image_name, bg_image_name,
n_neighbors, features), result_BGR)
cv2.imwrite('./result/{}_{}_k{}_{}_alpha.png'.format(image_name,
bg_image_name, n_neighbors, features), alpha * 255)
```

For the code, we have reference from [2], and the result of all images I work on when using `HSV` as feature vector representation and `K=10` is in `./result`.

2. Experiments

For the experiments, we have two experiments, first with the KNN matting with different parameter k, the second experiment is the different feature vector representation when finding KNN.

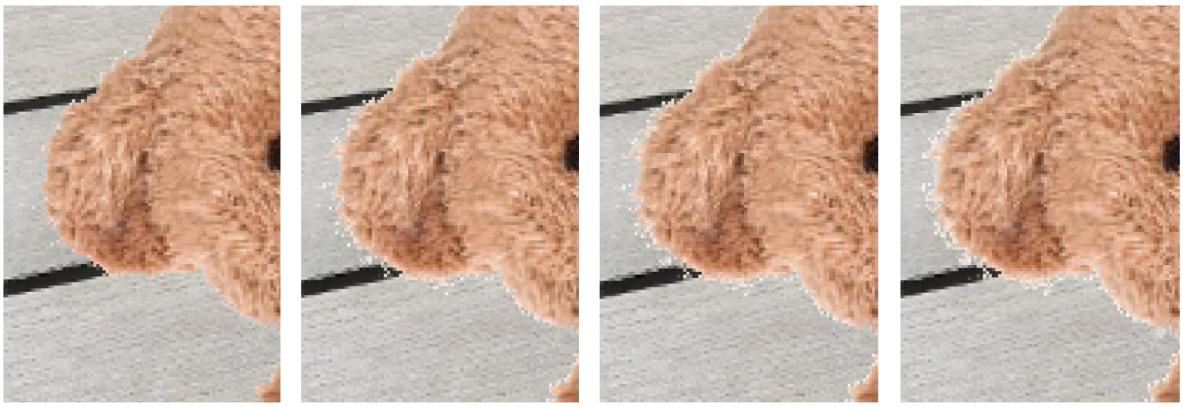
2.1 With different K

For the experiments, we have do two experiments, the first experiment is the KNN's parameter K, the following table is the is the different parameter K on each images, including the "troll" for bouns.

For all images, some pixel nearing edge are missing when K=5, shown in Figure 1 to 4, as K increase, they have something different, bear image start to have some noise around the edges, shown in Figure 1, gandalf image don't have too much different until k=50, the lower right corner of the image start to have some unknown part, for woman image and troll image, the bigger K will have better result, but for troll image, with bigger K, when the original image's background and

foreground we want to matting is too similar, some background will have higher probability to have higher alpha value.

For this experiment so we can conclude that when the image is not complex, K=10 is enough for matting, but when encountering some complex situation, such as foreground details mixed with background such as hair mixed with background, we have to increase k carefully to get better result while avoiding to let too many background have higher alpha value.

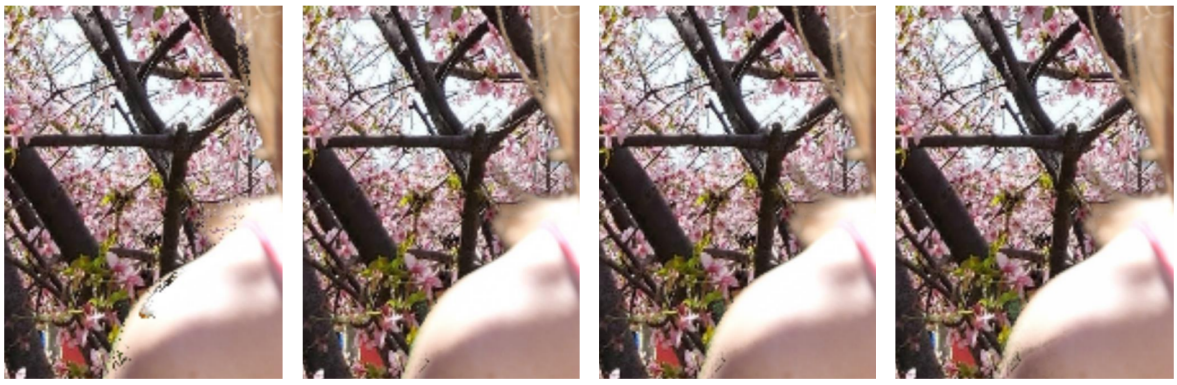


(a)K=5 (b)K=10 (c)K=15 (d)K=50

Figure 1: zooming in bear's composting result as K increasing

(a)K=5 (b)K=10 (c)K=15 (d)K=50

Figure 2: zooming in gandalf's composting result as K increasing



(a)K=5 (b)K=10 (c)K=15 (d)K=50

Figure 3: zooming in woman's composting result as K increasing

(a)K=5 (b)K=10 (c)K=15 (d)K=50

Figure 4: zooming in troll's composting result as K increasing

2.2 With different feature vector representation

The second experiment is change the feature represent, in the code part (see 1.1) , beside using HSV for $X(i) = (cos(h), sin(h), s, v, x, y)_i$ as feature vector, we also support user to choose $X(i) = (r, g, b, x, y)_i$ as feature vector.

For troll image in K=15, we can observe that the matting using HSV feature only have some edge from background while the matting using RGB feature have an whole part of castle from origin image being consider as foreground since the hair's color in RGB color space is too similar to the castle's , shown in Figure 5.

Another thing we observer is that when matting using RGB feature, we can get the whole hair part with K is only 10 while the matting using HSV feature can only get a few part of hair, shown in Figure 6.

For this experiment, we can conclude that, we can choose the proper feature repersentation according the image's we want to mat, for example, if the image's foregroud details mixed with background such as hair mixed with background, we can use HSV as feature with higher K for accuracy, otherwise we can use RGB as feature with lower K for faster computation.

(a)HSV as feature (b)RGB as feature

Figure 5: zooming in troll's composting result when using different feature representation (K=15)

(a)HSV as feature (b)RGB as feature

Figure 6: zooming in troll's composting result when using different feature representation (K=10)

3. Bonus

3.1 Apply to other images

For this bonus, we have use our method on other images on http://www.alphamatting.com/datasets.php [3], we choose the "troll" as the extra image, the result is shown in 2.1.

3.2 Different feature vectors representing

For this bonus, we try to use different ways of representing the feature vector, the detail and result is shown in 2.2.

3.3 KNN implement

For this bonus, we have try to implement KNN by ourselves, the code is as below, we take calculate the Euclidean distance of two feature_vec, sorted the result and take the first `n_neighbors` element's index as output, since the whole function is sequential, once the image's size or the K of KNN is too large, it will be time consuming, so we still use sklearn's NearestNeighbors for implement.

We have also compare the speed of using our own KNN and sklearn's, for an image of 128x160 and n_neighbors set to 10, for the part of getting neighbors, our implement takes 26.877 seconds while sklearn's only takes 0.0670 seconds.

```
def my_get_knn(feature_vec, n_neighbors):
    idx = []
    for i in range(feature_vec.shape[0]):
        dist = np.sum((feature_vec - feature_vec[i]) ** 2, axis=1)
        idx.append(np.argsort(dist)[:n_neighbors])
    return np.array(idx)
```

4. References

[1] Qifeng chen, Dingzeyu li, & Chi-keung tang. (n.d.). KNN Matting. https://dingzeyu.li/projects/knn/

[2] MarcoForte/knn-matting https://github.com/MarcoForte/knn-matting

[3] Alpha Matting Evaluation Website. http://www.alphamatting.com/index.html