MIT CSAIL

6.819/6.869 Advances in Computer Vision

Spring 2022

Problem Set 7

Jiahui Tang
jiahuita@mit.edu

Problem 1 *:Texture Synthesis*

Code Snippets:

Find Best Matches:

```
def Find_matches(template, sample, G):

    epsilon = 0.1
    delta = 0.3

    #### TODO:
    _, w, nChannels = template.shape
    validMask = ~np.isnan(template)
    # multiply by G
    msk = validMask * G
    # normalize
    msk = msk/msk.sum()
    # unfold to column vector
    msk = msk.ravel(order = 'F')

    #### TODO:
    feature_matrix = np.concatenate(
        [im2col_sliding(sample[:, :, i], template.shape[:-1])
                for i in range(nChannels)],
        axis = 0
    )
    tmp = template.ravel(order = 'F')

    #### TODO:
    tmp[np.isnan(tmp)] = 0
    dist_with_msk = (feature_matrix.T - tmp)*msk
    ssd = (dist_with_msk**2).sum(axis = 1)
    # ssd error less than min ssd times (1+epi), and below threshold delta
    idx = np.argwhere( ssd <= min(ssd.min()* (1+epsilon), delta))

    best_matches = feature_matrix[(w*w)//2::(w*w), idx]
```

```
errors = ssd[idx]
return best_matches, errors
```

Gaussian Kernel:

```
G = np.zeros((w, w))
G[w//2, w//2] = 1
G = ndimage.gaussian_filter(G, sigma = w/6.4, mode = 'wrap')
G = np.repeat(G[:, :, np.newaxis], nChannels, axis=2)
```

Sample from Best Match:

```
### TODO:
### Sample from best matches and update synthIm
synthIm[ii[i], jj[i],:] = np.squeeze(best_matches[:,
                np.random.choice(np.arange(best_matches.shape[1]))])
```

Output Image:

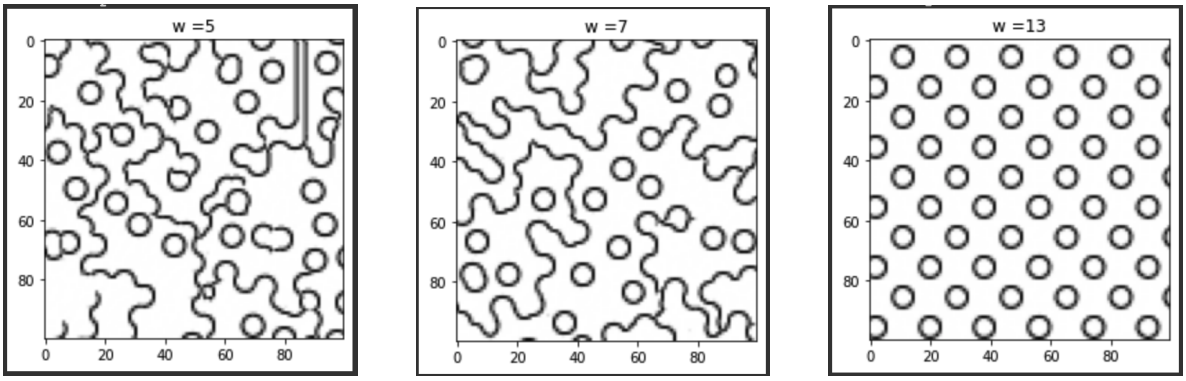

Figure 1: window widths of w = 5,7,13 and initial starting seed of (x,y) = (3,31)

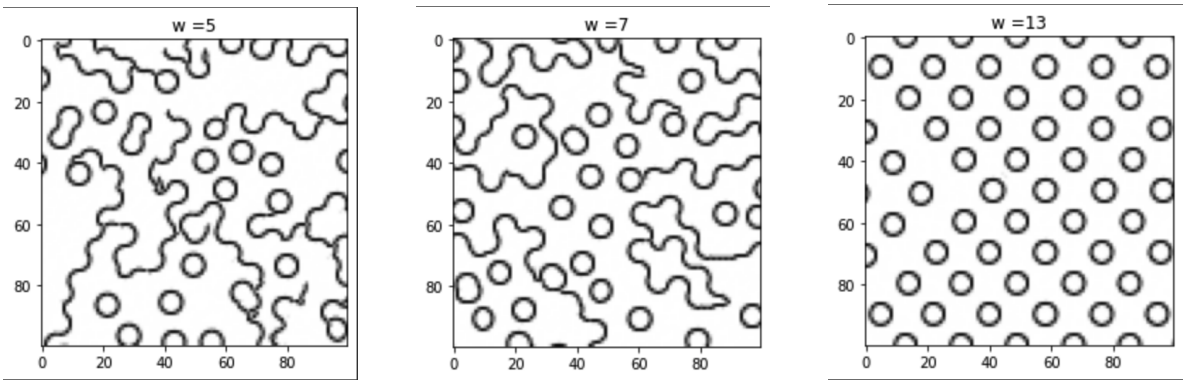

Figure 2: rerun the algorithm - 1 with the same starting seed

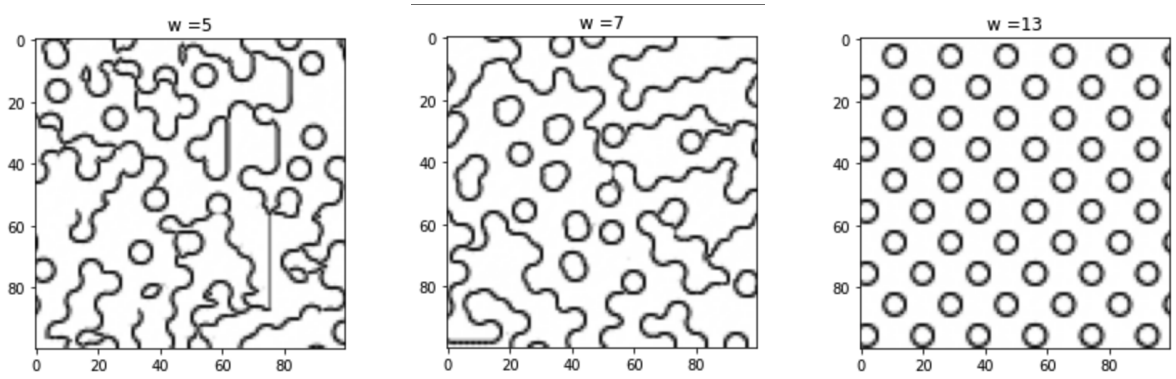

Figure 3: rerun the algorithm - 2 with the same starting seed

Analysis:

1. Algorithm's Performance with respect to window size:

Improving the window size significantly improve the quality of the output texture.

The algorithm could be more determined, and find more accurate match in source pattern when window size gets larger. It is because the algorithm are considering a bigger window, and a single pixel value in source image will be more likely to make sense.

When window size = 5 and 7, there are less ring patterns; however, when window size = 13, the pattern are ring patterns, with its original spatial information between rings. Thus, the synthesized image with larger window size looks closer to the source pattern.

2. Rerun:

For a given window size, if I re-run the algorithm with the same starting seed, I will get different result. It is because we are doing sampling and random choice from the best match from source image. Thus, different rerun will give slightly different results. When window size is smaller, the difference is more obvious. However, this observation holds for all window size.

It is especially clear when we look at the window size = 5,7, where it has different texture in each run.

At a first look, it seems to be getting same result when window size = 13, but when we look closer, the permutation of the texture are still slightly different, though the general texture of window 13 is quite uniformed, making it hard to distinguish the subtle differences.

Problem 2 *:Neural Style Transfer*

(1)

Code Snippets:

```
def gram_matrix(input):
    a, b, c, d = input.size()  # a=batch size(=1)
    # b=number of feature maps
    # (c,d)=dimensions of a f. map (N=c*d)

    features = input.view(a * b, c * d)  # resise F_XL into \hat F_XL

    # inner product between the vectorised feature maps i and j in layer l
    G =  features@features.T

    # we 'normalize' the values of the gram matrix
    # by dividing by the number of element in each feature maps.
    return G.div(a * b * c * d)
```

And the output images are attached below

Figure 4: Left: Conv1; Right: Conv5

(2)

(b), (c)

(3)

(b), (c)

(4)

Summary of *Arbitrary Style Transfer in Real-time with Adaptive Instance Normalization, Huang and Belongie*:

On top of the style transfer algorithm, this paper introduces an algorithm that improves the original style transfer algorithm's slow iterative optimization process, by speeding up the style

transfer process using fast approximations with a feed forward neural network (FFNN), but also at the same time enable real time arbitrary styles transfer.

It resolves fundamental flexibility-speed dilemma. The architecture is improved by adding a layer called novel adaptive instance normalization (AdaIN) layer, which aligns the mean and variance of content features from any new arbitrary style template. It combines flexibility of optimization based algorithm with speed of FFNN. This layer could do style normalization by normalizing and transferring feature statistics (especially the channel wise mean and variance), and later a decoder network will synthesize stylized image by inverting output from AdaIn back to image space. The decoder mostly mirros the encoder, with pooling layers replaced by nearest up-sampling to reduce checkerboard effects. Instead of training to modify pixel values to directly match feature statistics, this network architecture directly align statistics in the feature space in one shot, and inverts feature back to pixel space. This method also enables abundant user control, color and spatial control.