Problem Set 3

CMSC 426

Assigned Tuesday, Sept. 27, Due Tuesday, October 11

Background Subtraction 15 points

For this problem, you are given a set of 100 images of background, I_1, I_2, … I_{100}. and one test image, *J*. These are on the class web page in BackgroundImages.zip and foreground_image.jpg. Your task is to classify each pixel in the test image as either foreground or background. Suppose pixel *J(x,y)* has intensity *k*. To classify it, you should compute:

$$P(J(x,y)=k)=\sum_{i=1}^{100}\frac{1}{100\sigma\sqrt{2\pi}}\exp\left(-(k-I_i(x,y))^2/2\sigma^2\right)$$

Once you've computed this for each pixel, you'll need to choose a threshold, *T*, so that you classify all pixels as foreground when $P(J(x,y)=k)<T$. Choose a value of *T* by hand that seems to produce pleasing results. Our results are in BackgroundSubtractionResults.jpg. Turn in your code, a picture of your result, and indicate which threshold you used. Our results are included on the class web site for comparison.

Hint: As usual, you can achieve better performance by minimizing the amount of looping you do. However, when dealing with 100 images, you must be a little careful to avoid using up too much memory; in my implementation I do loop through the images rather than trying to perform some operations on all of them at once.

Challenge Problem: Up to 20 points

See if you can improve on this performance by using a statistical model that does not assume that every pixel is independent. For example, you could try to model the distribution of pairs of pixels, or make use of the fact that if $I(x,y) = k$, it is somewhat likely that in the next image, $I(x+1,y)=k$, (to model, for example, the way the trees move around in the wind). Describe what you did and whether it worked.

Note that this is a vague problem, and I'm not sure whether really good results can be obtained. But any good ideas that you try will get some credit, whether they work or not.

Texture Synthesis

The goal of this problem is to implement the texture synthesis method of Efros and Leung. This is described in the paper: ``Texture Synthesis by Non-parametric Sampling'', by Efros and Leung, in the International Conference on Computer Vision, 1999. There is a link to their web site on the class web site, which includes links to their paper and pseudocode for their algorithm. In this assignment, I have simplified their

algorithm a little bit, to remove some details such as Gaussian weighting, which don't seem to be necessary to achieve good performance.

This algorithm takes a *sample* of some texture and generates a new *image* containing a similar texture. The strategy of the algorithm is to generate each new pixel in the image using a *neighborhood* of already generated pixels. One looks in the sample for similar neighborhoods, selects one of these similar neighborhoods at random, and copies the corresponding pixel into the new image. You are only required to implement this program for black and white images. You are given code in GrowImageShell.m that has some parts missing, which you'll need to fill in

As always in Matlab, it is important to avoid loops to get good performance. It is quite possible for you to complete this assignment without using any additional loops beyond the ones I use in GrowImage.

1. **SSD 25 points:** S will contain a sample of the texture we want to generate. T contains a small (2n+1)x(2n+1) neighborhood of pixels. Not all the pixels in this neighborhood have been filled in with valid values however. So M (the *mask*) is a (2n+1)x(2n+1) matrix that contains a 1 for each position in which T contains a valid pixel, and a 0 whenever the corresponding pixel in T should be ignored. Computing the SSD is like correlation in that we shift the template over every position in the sample, and compute a separate result for each position. Thus, the output D is the same size as S. To compute D(i,j) we shift T so that its center is right on top of S(i,j). Then we take the difference between each valid pixel in T and the corresponding pixel in S, square the result, and then add all these together. This computation is described on page 147 of Trucco and Verri, and by Efros and Leung.

   To begin, you should write a function called SSD which performs the central step of the algorithm. This will compute the sum of squared difference between a little portion of the new image you are making and every portion of the sample. It should have the form:

function D = SSD(S, T, M)

There are two things that make SSD a bit tricky. The first is that we want to do it with correlation, rather than with looping. This is partly because this will be more efficient in Matlab, but also because we want to make concrete for you the relationship between correlation and SSD as a way of matching a template to an image. The second thing that's tricky is that we need to use a mask, *M*, which tells us to only perform the computation on some pixels. To separate these issues, let's first look at SSD without a mask. We can write SSD mathematically as:

$$D(x,y) = \sum_{i=-n}^{n} \sum_{j=-n}^{n} \left(S(x+i, y+j) - T(i,j)\right)^2$$

To see how this is related to correlation, let's open up the expression:

$$
\begin{aligned}
D(x,y) &= \sum_{i=-n}^{n}\sum_{j=-n}^{n}\left(S(x+i,y+j)-T(i,j)\right)^2 \\
&= \sum_{i=-n}^{n}\sum_{j=-n}^{n}\left(S(x+i,y+j)^2 - 2T(i,j)S(x+i,y+j)+T(i,j)^2\right) \\
&= \sum_{i=-n}^{n}\sum_{j=-n}^{n}S(x+i,y+j)^2 - \sum_{i=-n}^{n}\sum_{j=-n}^{n}2T(i,j)S(x+i,y+j)+\sum_{i=-n}^{n}\sum_{j=-n}^{n}T(i,j)^2
\end{aligned}
$$

The bottom line shows that to compute SSD we can combine 3 separate summations. You should notice that the middle term is just -2 times the result of correlating the template, *T*, with the sample image, *S*. We know how to compute this, for example, by using *imfilter*. This also gives us a sense of the relationship between SSD and correlation; in general, the higher the correlation between the template and the sample at some location, the lower the SSD will be. This is why correlation alone is sometimes used as a way of finding locations where the template and sample are similar.

The third term does not depend on *S, x,* or *y* at all. It only depends on the template, *T*. So it doesn't have to be computed at every point in *S*, it just has to be computed once (you should be able to compute it without looping).

So the only problem we have left is to compute the first term:

$$\sum_{i=-n}^{n}\sum_{j=-n}^{n}S(x+i,y+j)^2$$

This only depends on the squared value of the sample image. Given this squared sample image, we need to add up all the values in a square region of S. You should try to figure out how to do this using *imfilter* (and no loops).

Things get a little more complicated when we have a mask. We can write this as:

$$D(x,y) = \sum_{i=-n}^{n}\sum_{j=-n}^{n}\left(S(x+i,y+j)-T(i,j)\right)^2 M(i,j)$$

In this case, for every summation, we only want to use the pixels in the template that are part of the mask. For example, if we make some of the pixels in the template 0, these will not have any effect when we correlate the template with *S*, or when we compute the sum of squared values in the template. When we compute the first of the three terms in SSD, we must filter the squared sample values so that we will only count the valid values in the template.

You can test your function using the my_checkerboard function. Execute the call:

SSD(my_checkerboard(3,3),[0 1 1; 1 0 -1; 1 0 0], [1 1 1; 1 1 0; 1 1 1])

and my program produces:

```
SSD(my_checkerboard(3,3),[0 1 1; 1 0 -1; 1 0 0], [1 1 1; 1 1 0; 1 1 1])
ans =
   4   4   5   7   5   4   2   4   4
   4   4   4   5   4   4   3   4   4
   6   5   3   2   3   5   6   5   4
   7   5   3   0   3   5   8   5   4
   5   4   4   3   4   4   5   4   4
   3   3   5   6   5   3   2   3   4
   2   3   5   8   5   3   0   3   4
   4   4   4   5   4   4   3   4   4
   4   4   3   3   3   4   4   4   4
```

Turn in a printout of the results, along with your code.

You may be worried about how to compute SSD when the template goes outside the boundary. Don't worry about this. You can do anything that produces reasonable results. For example, you can just use the default values of imfilter, which treats pixels outside S as if they were 0. This is what I did in the example above. This will work fine, since these areas then will generally not match your template very well.

2. **15 points:** Using SSD and the pseudocode below, implement the function FindMatches. This should find all candidate pixels in the sample that have a neighborhood that is sufficiently similar to the template. In finding the best matches it is also important that you not loop through the entire image. Here is one useful Matlab trick for avoiding such loops. You can index into a matrix using a binary array that is the same size as the matrix. You will then get back a vector with all the matrix elements that are at locations where the index was 1. For example:

```
a = rand(3,3)
a =
   0.5515   0.1027   0.4576
   0.1866   0.7407   0.9439
   0.2879   0.3420   0.9477
>> a(a>.5)
ans =
   0.5515
   0.7407
   0.9439
   0.9477
```

You might want to write FindMatches so that it takes three inputs: the sample image, the template, and the mask. Test your program on the same example that you used in problem 1. Hand in printouts of the results, and of your program.

3. **35 points:** Now complete your program, following the skeleton of the code that we provide. I am also including below the pseudocode given by Efros and Leung, which is similar to the skeleton in GrowImageShell.m. Test your program using the checkerboard function to generate a sample checkerboard pattern, and using the bricks image in brickbw.jpg. Hand in printouts of the results. Be warned that with this algorithm it can be a bit slow to generate a large texture, even if you have implemented SSD efficiently. My implementation takes about 45 seconds to generate a 100x100 image on my laptop, when I use a value of n=7. Run time grows with n. Your implementation might be slower if you use WAM computers, but if it's much, much slower, you might reconsider your implementation.
4. **10 points:** Use two more images of your own choosing as sample textures and generate new textures using these samples. Turn in printouts of the original images and of the textures you generate. Also, include these images in your electronic submission. For **10 points** extra credit, modify your program so that it will work with color images, and generate color textures.

Appendix: Below is pseudocode provided by Efros and Leung. Please note that when you code this project in Matlab you won't want to follow this pseudocode exactly. For example, some operations that they perform with loops, you will want to perform with single Matlab routines.

Algorithm details

Let `SampleImage` contain the image we are sampling from and let `Image` be the mostly empty image that we want to fill in (if synthesizing from scratch, it should contain a 3-by-3 seed in the center randomly taken from `SampleImage`, for constrained synthesis it should contain all the known pixels). `WindowSize`, the size of the neighborhood window, is the only user-settable parameter. The main portion of the algorithm is presented below. I have removed the part of the pseudocode that deals with Gaussian filtering, for simplicity.

```
function GrowImage(SampleImage,Image,WindowSize)
  while Image not filled do
    progress = 0
    PixelList = GetUnfilledNeighbors(Image)
    foreach Pixel in PixelList do
      Template = GetNeighborhoodWindow(Pixel)
      BestMatches = FindMatches(Template, SampleImage)
      BestMatch = RandomPick(BestMatches)
      Pixel.value = BestMatch.value
  end
  return Image
end
```

Function `GetUnfilledNeighbors()` returns a list of all unfilled pixels that have filled pixels as their neighbors (the image is subtracted from its morphological dilation). The list is randomly permuted and then sorted by decreasing number of filled neighbor pixels. `GetNeigborhoodWindow()` returns a window of size `WindowSize` around a given pixel. `RandomPick()` picks an element randomly from the list. `FindMatches()` is as follows:

```
function FindMatches(Template,SampleImage)
  ValidMask = 1s where Template is filled, 0s otherwise
  TotWeight = sum i,j ValidMask(i,j)
  for i,j do
    for ii,jj do
      dist = (Template(ii,jj)-SampleImage(i-ii,j-jj))^2
      SSD(i,j) = SSD(i,j) + dist*ValidMask(ii,jj)*GaussMask(ii,jj)
    end
    SSD(i,j) = SSD(i,j) / TotWeight
  end
  PixelList = all pixels (i,j) where SSD(i,j) <=
min(SSD)*(1+ErrThreshold)
  return PixelList
end
```

In our implementation the constant were set as follows: `ErrThreshold = 0.1`. Pixel values are in the range of 0 to 1.
