

# Problem Set 3 Solutions

CMSC 426

Thursday, October 13

## Background Subtraction 15 points

For this problem, you are given a set of 100 images of background,  $I_1, I_2, \dots, I_{100}$ , 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:

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

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.

We can do this with the assistance of the following two functions. First, we really need a function to read in 100 images; doing this by hand would be too tedious.

```
function WT = read_waving_trees(st, en)
% There were really more than 100 images I was playing with. So I made
% this function general enough to read in any subset of them. st and
% en should be integers that indicate the first and last images that we
% want to use. These could just be set to 0 and 99.
WT = [];
% We will store the images in a 3D matrix. WT(:,:,i) will contain
% image i.
for i = st:en
    is = int2str(i);
    zeros = [];
    for j = 1:(5-length(is))
        zeros = [zeros, '0'];
    end
    % The file name will have either 2, 3 or 4 zeros depending on the
    % number of characters needed to represent the current image
    % number.
    fn = ['b', zeros, is, '.bmp'];
```

```

    % Notice we can concatenate strings together just as we do vectors,
    % since a string is a vector of characters.
    WT = cat(3, WT, rgb2gray(imread(fn)));
    % We convert everything to gray scale, since we don't use color.
    % Notice that cat can be used to concatenate matrices in the 3rd
    % dimension.
end

```

Now we do the real work. B will contain the background images, in the form produced by read\_waving\_trees. I contains the foreground image (we'll read this in and convert it to gray with interactive commands).

```

function P = background_probability(B, I, sigma)
% B will be a 3D matrix in which B(:,:,i) is an image.
P = zeros(size(I));
% We will sum the probabilities into this matrix.
ID = double(I);
for i = 1:size(B,3)
    % I'm going to iterate to avoid using large amounts of memory.
    D = ID - double(B(:,:,i));
    % D now contains the difference between the foreground image and background image
    % i.
    P = P + (1/(100*sigma*sqrt(2*pi))) * exp( -(D.^2)/(2*sigma^2));
% This implements the equation given in the problem.
end

```

We can turn this result into a binary array in which foreground is 1, with a command like:

```
P < SomeThreshold.
```

A sigma that is a small integer makes sense to use in computing this, but to find a good value for SomeThreshold one really just has to use trial and error.

### 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,  $I(x+1,y)=k$ . 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. I will give you some partial code, with some functions you 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`. You are given code in `GrowImageShell.m` that has some parts missing, which you'll need to fill in.

1. **SSD 25 points:**  $S$  will contain a sample of the texture we want to generate.  $T$  contains a small  $(2n+1) \times (2n+1)$  neighborhood of pixels. Not all the pixels in this neighborhood have been filled in with valid values however. So  $M$  (the *mask*) is a  $(2n+1) \times (2n+1)$  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:

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

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

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

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:

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

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 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. `GetNeighborhoodWindow()` 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.

---

## Solution

I'll just put the whole solution here at the bottom, with comments.

```

function I = GrowImage(S, n, w, h)
if n/2 == round(n/2)
    error('n must be odd');
end
nh = (n-1)/2;
Ipad = init(S,n,w,h,nh);
S = double(S);
PixelList = GetUnfilledNeighbors(Ipad);
count = 0;
while ~isempty(PixelList)
    count=count+1
    % I just print out count so I can see how far along we are.
    for i = 1:size(PixelList,2)
        Pixel = PixelList(:,i);
        Template = GetWindow(Pixel, Ipad, nh);
        % Template may contain -1 or -2; we don't use those pixels.
        BMs = FindMatches(Template, S);
        BM = RandomPick(BMs);
        Ipad(Pixel(1),Pixel(2)) = BM;
    end
    PixelList = GetUnfilledNeighbors(Ipad);
end
I = Ipad((nh+1):(h+nh), (nh+1):(w+nh));

```

```

function BM = RandomPick(BMs)
% Pick a column at random
ind = 1+floor(rand*length(BMs));
% This gives us a random number between 1 and length(BMs).
BM = BMs(ind);

function BMs = FindMatches(Tem, S)
% This function will compute the SSD between the template and every
% equal-sized window in S. If the Template contains a -1, we want to
% ignore that pixel in computing the SSD. Then we want to return the
% pixel values in S at the center of every region that has a distance
% to the template that is no more than 1.1 times the distance of the
% nearest region.
THRESHOLD = 1.1;
ValidMask = Tem>=0;
% We only use pixels in the template that are non-negative.
D = SSD(S, Tem, ValidMask);
Best = max(0,min(D(:)));
% With round-off error, we can get a best distance that is negative,
% and this causes trouble.
BMs = S(D <= Best*THRESHOLD);

function D = SSD(S, Tem, M)
ker = Tem.*M;
% This is the part of the template, Tem, that has valid values, as
% indicated by the mask, M. Anything that is 0 in M is 0 in ker too.
Ssq = S.^2;
S2 = imfilter(Ssq, double(M));
% S2 will contain, for each pixel, the sum of squares of values in S
% that are in the neighborhood of that pixel, and that play a part in
% comparing the region around that pixel to the template in Tem.
S_Tem = imfilter(S, ker);
% This is the result of correlating S and Tem, again only using the
% valid parts of Tem.
Tem2 = sum(sum(ker.^2));
% Sum of squares of valid entries in Tem. Note that this is a scalar,
% while S2 and S_Tem are matrices. This is because the value computed
% in Tem2 doesn't depend on pixel position. But we can still combine
% these with no problem.
D = S2 + Tem2 - 2*S_Tem;

function Tem = GetWindow(P, I, nh);
% Extract an 2*nh+1x2*nh+1 window from I that is centered on pixel P.
Tem = I((P(1)-nh):(P(1)+nh), (P(2)-nh):(P(2)+nh));

function PL = GetUnfilledNeighbors(I)
% We want to find every -1 that is next to something that's not -1.
% For speed, it is important to do this without looping, using
% imfilter and find. These are returned as a 2xn matrix, where every
% column gives the position of a pixel that is -1, but next to
% something not -1. The first row of PL gives the row position of the
% pixels, the second row gives the column position.
Id = imfilter(I>=0, [0 1 0; 1 0 1; 0 1 0]) > 0;
% Id will indicate whether the number of neighboring pixels that are
% non-negative is bigger than 0. Note that I only used four neighbors.
% One could probably use a neighborhood of eight pixels equally well.

```

```

[rows, cols] = find(Id & (I==-1));
% This will find rows and columns for values that are -1 and that have
% a neighbor that is non-negative.
PL = [rows'; cols'];

function Ipad = init(S,n,w,h,nh)
% We will pad the image with -2s so that we don't have to check
% boundary conditions when we index into it.
% -1 will be a pixel we still need to fill in.
% We will start out Ipad with a 3x3 section in the center that we take
% randomly from the sample, S.
Ipad = [-2*ones(h+2*nh,nh), [-2*ones(nh,w+nh); ...
    [-1*ones(h,w), -2*ones(h,nh)]; -2*ones(nh,w+nh)]];
[Sh,Sw] = size(S);
r = 1+floor((Sh-2)*rand);
c = 1+floor((Sw-2)*rand);
hc = round(nh+h/2);
wc = round(nh+w/2);
Ipad(hc:(hc+2), wc:(wc+2)) = S(r:(r+2), c:(c+2));

```