1.1

IP&PR: Guided super-resolution

Braulio Sespede, Sara Naghedi

November 4, 2019

1 Introduction

In this report we will explain our implementation of the guided filter [1] in the context of guided super-resolution. The guided filter, just like the bilateral filter, is a filter capable of smoothing an image while preserving it's edges. In contrast to the bilateral filter, a guidance image is used to restrict the filter's smoothing capabilities. While discussing the bilateral filter, it is relevant to mention that the guided filter's complexity is independent of the chosen filter size, making it specially interesting for high-resolution images where bigger filter sizes might be needed. Even though it is frequent to use the same image to guide the filter, other images can be used. Such is the case of guided super-resolution, where a higher resolution image can be used to recover the lost edge information during downsampling.

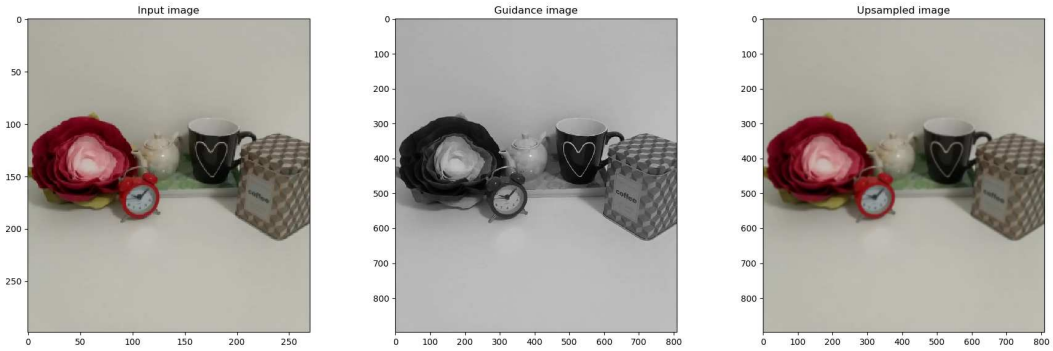

Figure 1: Left: Downsampled image taken with a cellphone, Center: Guidance high-res image, Right: Upsampled with the use guided filter

In the first assignment we were given the task of transforming a 3-channel image I into a higher-resolution version of said image. To achieve said task a 1-channel guidance image G was used. G has the necessary edges to recover the lost structure during downsampling. Furthermore, the guided filter allows us to spread the intensity values of the RGB channels between said edges. The goal is to obtain an image as similar as possible to the original reference image. We are particularly interested in taking a look at why and how certain parameters of the guided filter affect the given task.

As a last introductory note, we will briefly mention the two different approaches that will be discussed at a further point of the report: (i) The first method requires resizing the input image I to an equal size as the guidance image G, then it's 3-channels are filtered individually using the guided filter with the guidance image G. Finally, the 3 filtered channels are merged into one image to obtain the super-resolution image. (ii) In the second approach, the guidance image G is downsampled to the same resolution as the input image I. Then, the image is filtered in a similar manner to the first approach. Since this would result in a low resolution image, the linear coefficients a and b are resized to the reference image size and used to compute a filtered image q. We are particularly interested in analyzing the effects of the loss of information in the guidance image.

2 Method

The guided filter relies heavily on the computation of efficient *box filters* and *variance* images, thus special importance was placed on using efficient implementations. Even though there are variations of the guided filter that can reduce the temporal complexity even further (e.g. using local histograms), we decided to use the "vanilla" version of the guided filter. In particular, since python is a high-level interpreted language, it is important to rely on functions that wrap efficient lower-level language functions. In our case, we used the sliding window technique explained during the practical in order to avoid nesting for loops unnecessarily. Since the use of *view_as_windows* implied adding a new import, we decided to build the windows manually (thus only traversing through the filtering window).

Another aspect that we consider relevant is the fact that our implementation of the guided filter is only able to filter images using grayscale guidance images. Even though it would be trivial to extend, we considered that it was sufficient for the expected images. This implies that our self-taken guidance image will have to be converted to grayscale as well.

Before describing the evaluation procedure and the corresponding results we will mentioned the supporting hardware and software used to develop and evaluate our implementation:

- **CPU:** Intel i7 8550U @ 1.8 GHz x8
- **OS:** Ubuntu 19.10
- **Language:** Python 3.7 (Conda)

Beyond specific implementation details, we will now mentioned the upcoming evaluation procedure. First, we will evaluate how the different guided filter parameters (i.e. smoothing factor and filter size) affect the end-result qualitatively. This procedure will be performed for both methods mentioned in the introduction. Furthermore, we will compare the effect of several downsampling rates when using either of the approaches. Second, we will perform a quantitative analysis by taking into account the *peak signal to noise ration* (PSNR) for each of these parameters (also for both methods). Finally, we will take a look at the runtime performance for each of these parameters in order to see how they affect performance (if they do so at all).

3 Results

As previously mentioned, we will begin by taking a look at the effect the hyper parameters have on the resulting super-resolution image. To do so, we first evaluate our implementation on the *monarch.png* image. We plan to use the following parameters for both approaches:

- **Epsilon:** [0.001, 1, 1000]
- **Filter sizes:** [3, 7, 15]

Figure 2 shows the results using the first approach. As we can see the images get progressively blurry as we move from low filter sizes to bigger ones. This makes sense if we considering that the mean of our linear factors increases its effects to more pixels. Something interesting we notice is that when we use really low epsilons, the images become overexposed and display a bloom effect (see top right corner of Figure 2). Other than these observations, we notice that when proper parameters are chosen, the super-resolution works properly and the effects of the filter allow input image to preserve the edges of the guidance image.

In the case of the second approach, we first notice that the same parameters cause different results. This is not surprising if we consider the fact that the guidance image was downsampled, and the linear factors a and b upsampled. In contrast to the previous experiment, we don't notice the bloom effect in the lower epsilon and big filter size case. Moreover, we notice that as images get blurry, they still manage to preserve edges partially (specially noticeable in the image with $eps = 1$ and $fs = 15$). This shows that as the regularization parameter eps increases the images starts progressively losing its original structure.

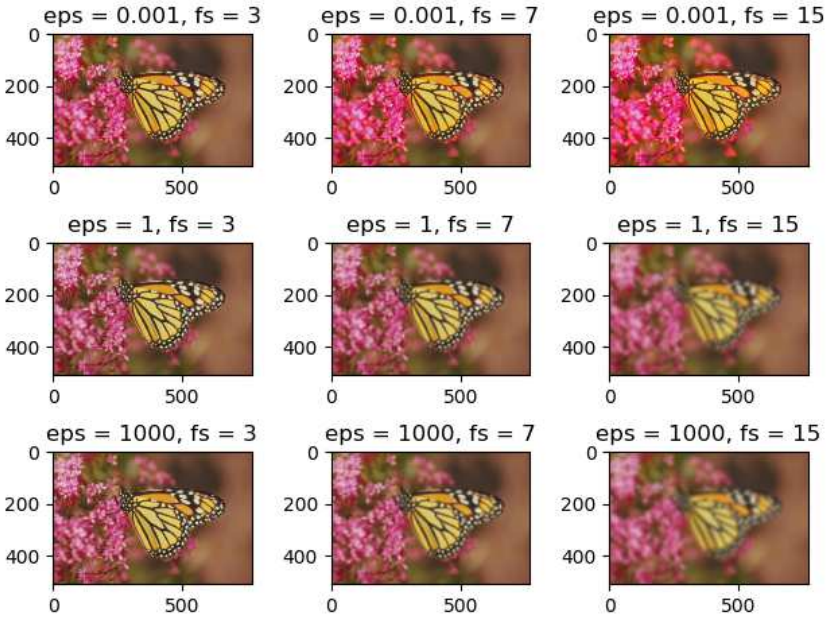

Figure 2: First approach: Comparison of different epsilon's and filter sizes while maintaining the downsampling rate at 4. The images shown are the end-result of the super-resolution algorithm.

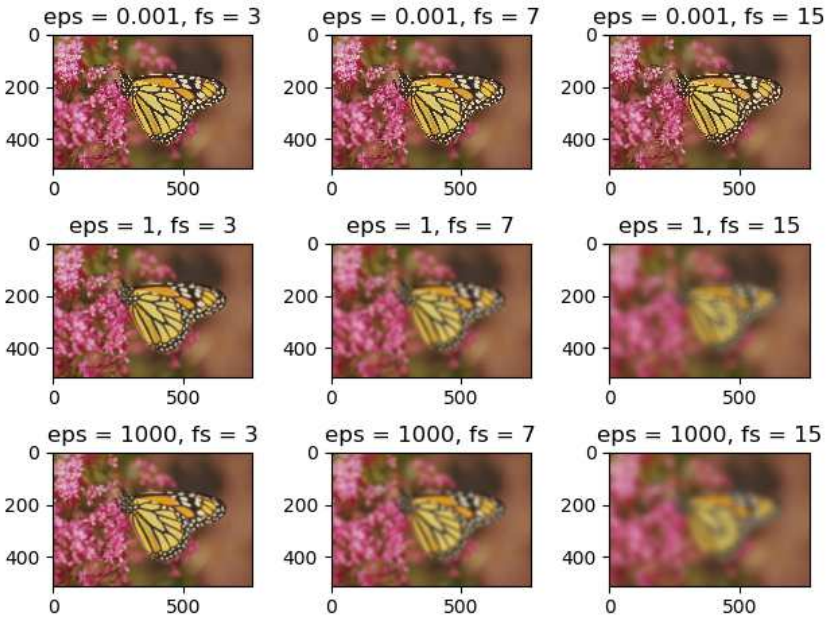

Figure 3: Same as Figure 2, but now using the second approach.

Up ahead we will visualize different levels of downsampling rates for each of the 2 approaches (shown in Figure 4). In particular, we used 2, 8, and 16 as downsampling rates. We can qualitatively observe that while the first approach looks blockier, the second one behaves as if no guidance image was present as we increase the downsampling rate. This might an advantage to the first approach if we consider edge preservation. If we consider the fact that we downscaled the guidance image this makes sense, as edges were lost during such procedure.

As we can observe in the Appendix table, the PSNR is quite similar for all the configurations. Even

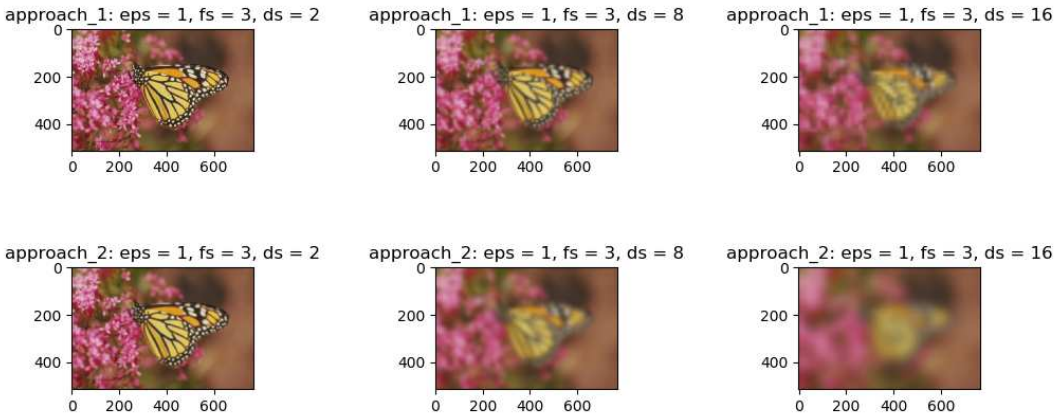

Figure 4: First row shows the first approach with progressively higher downsampling rates. The second row follows the same idea for the second approach. The *epsilon* and $filtersize$ remain the same for both approaches.

though this might seems counter-intuitive, it's clear that PSNR is not meant to faithfully represent the human perception of images. This makes sense if we consider that the PSNR merely compares color difference between pixels. Furthermore, it has no sense of locality, simply summing over such differences. This applies to PSNR resulting from the comparison of the upsampled images and the reference images, as well as the resized input image with the reference image (for both approaches). Observing the runtime performance we notice that the only parameter that seems to significantly increase runtime is the filter size on the first approach (this might be related to the fact that we operate at a higher resolution). Other than that, we notice that the use of the sliding window technique makes the overall performance significantly better.

4 Conclusion

To conclude this report, we will mentioned a few takeaways from this experience:

- The first approach does a better job at preserving edges when heavy downsampling is applied to the input image.
- The first approach might be slower due to the fact that it filters at a higher resolution than the second approach.
- Since the different approaches work at different resolutions it might be an unfair comparison to use the same parameters when evaluating them.
- It is necessary to avoid nesting loops in *Python* and use functions that were implemented using lower-level languages.
- PSNR is a really bad metric for comparing images at this particular task. (vi) The guided filter is really good for this particular task and is perceptively equal to a reference image (when using the correct parameters).

References

[1] Kaiming He, Jian Sun, and Xiaoou Tang. Guided image filtering. *IEEE transactions on pattern analysis and machine intelligence*, 35(6):1397–1409, 2012.

5 Appendix: PSNR & runtime

parameters			PSNR				time (sec)	
eps	fs	ds	filtered_1	filtered_2	upsample_1	upsample_2	method_1	method_2
0.001	3	2	22.6703	22.6700	22.6692	22.6692	0.2876	0.1987
0.001	3	8	22.6648	22.6688	22.6646	22.6646	0.3534	0.1067
0.001	3	16	22.6619	22.6672	22.6618	22.6618	0.2610	0.1134
0.001	7	2	22.6717	22.6698	22.6692	22.6692	1.4754	0.4438
0.001	7	8	22.6650	22.6684	22.6646	22.6646	1.5494	0.1263
0.001	7	16	22.6620	22.6673	22.6618	22.6618	1.5344	0.1313
0.001	15	2	22.6733	22.6693	22.6692	22.6692	6.0607	1.5688
0.001	15	8	22.6656	22.6678	22.6646	22.6646	6.3702	0.1731
0.001	15	16	22.6622	22.6670	22.6618	22.6618	5.7773	0.1196
1	3	2	22.6684	22.6671	22.6692	22.6692	0.2593	0.1420
1	3	8	22.6645	22.6617	22.6646	22.6646	0.2657	0.1415
1	3	16	22.6618	22.6597	22.6618	22.6618	0.3446	0.1588
1	7	2	22.6665	22.6640	22.6692	22.6692	1.4839	0.3456
1	7	8	22.6640	22.6595	22.6646	22.6646	1.4685	0.1421
1	7	16	22.6617	22.6576	22.6618	22.6618	1.3569	0.1164
1	15	2	22.6637	22.6614	22.6692	22.6692	5.6861	1.4521
1	15	8	22.6627	22.6575	22.6646	22.6646	6.0613	0.1554
1	15	16	22.6613	22.6558	22.6618	22.6618	6.0883	0.1262
1000	3	2	22.6684	22.6670	22.6692	22.6692	0.2577	0.1419
1000	3	8	22.6645	22.6616	22.6646	22.6646	0.2484	0.1129
1000	3	16	22.6618	22.6596	22.6618	22.6618	0.2647	0.1484
1000	7	2	22.6664	22.6637	22.6692	22.6692	1.5392	0.3858
1000	7	8	22.6640	22.6593	22.6646	22.6646	1.5170	0.1246
1000	7	16	22.6617	22.6575	22.6618	22.6618	1.5671	0.1613
1000	15	2	22.6634	22.6611	22.6692	22.6692	5.8524	1.4994
1000	15	8	22.6626	22.6573	22.6646	22.6646	5.2413	0.1548
1000	15	16	22.6613	22.6557	22.6618	22.6618	5.7366	0.1330

5.1

Index of comments

1.1 Your implementation of compute_psnr is incorrect:
1. You should use log to the base 10
2. You call the method with floating point values already normalized between 0 and 1, and additionally divide them again through 255 in compute_psnr(...)

5.1 The reported PSNR values are incorrect