COMP6223 Coursework 1
Hybrid Images

Rhys Thomas, rt8g15@ecs.soton.ac.uk

8 Nov. 2018

1 Template Convolution

Template convolution is a group operator used to modify pixels in an image. Methods of performing the convolution include either an iterative approach, walking through the image and multiplying/summing neighbouring pixels, or using the Fourier transform.

1.1 Iterative Method

The iterative method works by multiplying every image pixel within a kernel and then summing them together; the result is then written to the centre pixel of the kernel. This is quite computationally expensive since the number of iterations over the image is given by equation 1.

$$\#\text{ of Iterations} = \prod_{i \in \{h,w\}} \left(\text{Image}_i - \left\lfloor \frac{\text{Kernel}_i}{2} \right\rfloor \right) \times \text{Colours} \tag{1}$$

So for a 300×300 tricolour image with a kernel size of 3×3, this would result in 268,203 iterations! The source code for this method is given below.

```
34 def convolution(img, kernel):
35     """ This function executes the convolution between `img` and `kernel`.
36     """
37     print("[" + img + "]\tRunning convolution...\n")
38     # Load the image.
39     image = cv2.imread(img)
40     # Flip template before convolution.
41     kernel = cv2.flip(kernel, -1)
42     # Get size of image and kernel. 3rd value of shape is colour channel.
43     (image_h, image_w) = image.shape[:2]
44     (kernel_h, kernel_w) = kernel.shape[:2]
45     (pad_h, pad_w) = (kernel_h // 2, kernel_w // 2)
46     # Create image to write to.
47     output = np.zeros(image.shape)
48     # Slide kernel across every pixel.
49     for y in range(pad_h, image_h-pad_h):
50         for x in range(pad_w, image_w-pad_w):
51             # If coloured, loop for colours.
52             for colour in range(image.shape[2]):
53                 # Get center pixel.
54                 center = image[y - pad_h:y + pad_h + 1,
55                                x - pad_w:x + pad_w + 1,
56                                colour]
57                 # Perform convolution and map value to [0, 255].
58                 # Write back value to output image.
59                 output[y, x, colour] = (center * kernel).sum() / 255
60
61     # Return the result of the convolution.
62     return output
```

You will notice that on line 42 the kernel is inverted on both axis in the spacial domain. This is to insure that we are implementing *convolution* rather than *correlation*, which does not require spacial domain inversion.

Figure 1: Template convolution without cropping.

An artefact intrinsic to the iterative method is that the resulting image needs to be cropped. This is because the kernel has a border which is not included in the calculation – remember, the convolution result during each iteration is only written to the centre pixel – thus reducing the size of the image; the effect of which is shown in figure 1.

Both this and the computation expense of the algorithm gives rise to the need for alternative methods. Performing convolution via the Fast Fourier Transform removes the padding border around the image – since the kernel is padded to the image size – and is much faster than iterative calculations.

1.2 Fourier Method

The Fourier convolution gives an equivalent result to template convolution, except it is more computationally efficient by working in the frequency domain, rather than the spacial domain. The Fourier convolution is calculated for each colour channel in the image using equation 2.

$$\text{Convolution} = \mathcal{F}^{-1}\left(\mathcal{F}(\text{image}) \cdot \times \mathcal{F}(\text{kernel})\right) \tag{2}$$

In the Fourier function shown below, the image and kernel are first loaded and their dimensions examined. With this information we are able to calculate the padding required around the kernel, in order to make it the same dimensions as the (larger) image.

When applying convolution using Fourier, the kernel and image must be the same size. This was not a requirement for the iterative method, since it scanned across the image applying calculations on groups of pixels. The padding can be easily calculated by subtracting the kernel dimensions from that of the image, and floor-dividing (`//`) by 2.

The FFT of each image is then calculated, multiplied together and then the FFT is inverted. You will notice, unlike in the template method, the kernel does not require inversion. This is because the inversion of the kernel is intrinsic to the Fourier transform.

```
65  def fourier(img, kernel):
66      """ Compute convolution between `img` and `kernel` using numpy's FFT.
67      """
68      # Load the image.
69      image = cv2.imread(img)
70      # Get size of image and kernel.
71      (image_h, image_w) = image.shape[:2]
72      (kernel_h, kernel_w) = kernel.shape[:2]
73      # Apply padding to the kernel.
74      padded_kernel = np.zeros(image.shape[:2])
75      start_h = (image_h - kernel_h) // 2
76      start_w = (image_w - kernel_w) // 2
77      padded_kernel[start_h:start_h + kernel_h,
78                    start_w:start_w + kernel_w] = kernel
79      # Create image to write to.
80      output = np.zeros(image.shape)
81      # Run FFT on all 3 channels.
82      for colour in range(3):
83          Fi = np.fft.fft2(image[:, :, colour])
84          Fk = np.fft.fft2(padded_kernel)
85          # Inverse fourier.
86          output[:, :, colour] = np.fft.fftshift(np.fft.ifft2(Fi * Fk)) / 255
```

```
87
88      # Return the result of convolution.
89      return output
```

It can be seen on line 87 that after computing the inverse FFT, `fftshift()` is called. This is due to how numpy calculates the two dimensional FFT; it assumes the origin (DC components) are at the top left of the image whereas we define them in the centre. To compensate for this, we manually shift the FFT so that the origin is in the centre of the image, which gives us the desired result. The effect of this shifting can be seen in figure 2 where pixels that exceed the image dimensions wrap around to the opposite side.

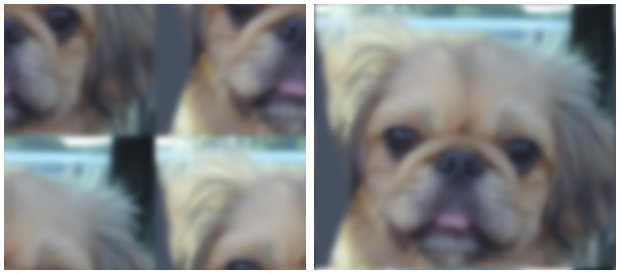

Figure 2: Effect on the image when not shifting (left) and shifting (right) the convolved images.

The speed difference between the iterative and Fourier methods is noticeable when you run the script. The recorded execution times for each are as follows,

```
time ./hybrid.py -i data/dog.bmp data/cat.bmp -c 4 4 -o hybrid.jpg -v visual.jpg
    6.68s user 0.15s system 80% cpu 8.496 total

time ./hybrid.py -i data/dog.bmp data/cat.bmp -c 4 4 -o hybrid.jpg -v visual.jpg -f
    0.67s user 0.13s system 156% cpu 0.510 total
```

This emphasises how much more practical/efficient the Fourier method is. Iterative template convolution is useful to *understand* convolution, however it serves little purpose beyond that.

2 Gaussian Filters

To low-pass filter an image, convolve a Gaussian filter – of particular cutoff frequency, σ – with the original image. One can simply subtract the low-pass filtered result from the original image in order to produce the high-pass filter equivalent.

The equation for a 2-dimensional Gaussian filter is shown in equation 3, however the $1/2\pi\sigma^2$ term is usually approximated to the sum of the kernel over its window size.

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(\frac{-(x^2 + y^2)}{2\sigma^2}\right) \tag{3}$$

The below function creates a Gaussian kernel, based on the value of σ passed to the function, and then runs either Fourier or template convolution based on the global flag `use_f`.

```
92  def gaussian_blur(image, sigma):
93      """ Builds a Gaussian kernel used to perform the LPF on an image.
94      """
95      print("[" + image + "]\tCalculating Gaussian kernel...")
96      # Calculate size of filter.
97      size = 8 * sigma + 1
98      if not size % 2:
99          size = size + 1
100
101     center = size // 2
102     kernel = np.zeros((size, size))
103
```

```
    # Generate Gaussian blur.
    for y in range(size):
        for x in range(size):
            diff = (y - center) ** 2 + (x - center) ** 2
            kernel[y, x] = np.exp(-diff / (2 * sigma ** 2))

    kernel = kernel / np.sum(kernel)

    if use_f:
        return fourier(image, kernel)
    else:
        return convolution(image, kernel)
```

Figure 5 shows the effect of applying a Gaussian blur to an image in order to produce a low-pass filtered version (left). As previously mentioned, a low-pass filtered image can be subtracted from the original image in order to produce a high-pass filtered equivalent (right). Note that each pixel of the high-pass image is increased by `0.5` to increase visibility.

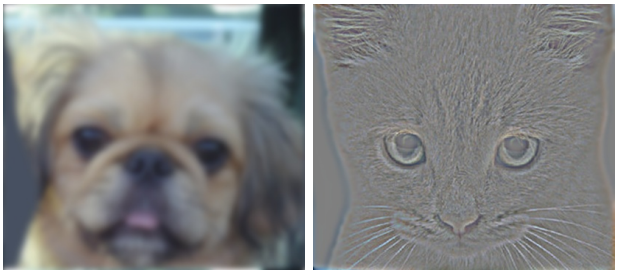

Figure 3: Low and high-pass filtered images with $\sigma = 4$.

3 Hybrid Images

Hybrid images are formed by summing together two *different* low-pass and high-pass filtered images. The higher frequencies dominate when the viewing distance is short, however at a longer distance the lower frequencies dominate. Figure 4 is a hybrid image composed from the two images in the previous section. The result is downsampled in order to demonstrate the aforementioned effect.

```
> ./hybrid.py -i data/dog.bmp data/cat.bmp -c 4 4 -o hybrid.jpg -v visual.jpg -f
```

Figure 4: Visualising the effect of hybrid images filtered with $\sigma = 4$.

4 Other Kernels

To demonstrate the convolution further, the script offers support for additional kernels. You will note that these other convolutions only act on the first image entered with the `-i` flag. This is to reduce script complexity yet demonstrating the working concept.

4.1 Sobel Edge Detection

Sobel edge detection applies the two dimensional kernels shown in the code below. One detects horizontal edges while the other detects vertical edges. The result of these two convolutions can be summed in order to display edges in both directions as shown in figure 5.

```
215         sobel_x = fourier(images[0], 255 * np.array([[1, 0, -1],
216                                                      [2, 0, -2],
217                                                      [1, 0, -1]]))
218         sobel_y = fourier(images[0], 255 * np.array([[1, 2, 1],
219                                                      [0, 0, 0],
220                                                      [-1, -2, -1]]))
```

This kernel can be applied to the first image in the two element list using the `-s` flag, as shown below.

```
> ./hybrid.py -i Data/plane.bmp Data/bird.bmp -s
```

Figure 5: Sobel edge detection applied in both horizontal (left) and vertical (middle) directions. These are then summed to produce a combined edge detection (right).

4.2 Arbitrary Averaging

The `-k` flag allows you to set the dimensions of a simple averaging kernel. The kernel is simply an array of ones, divided by the product of the dimensions specified – with an additional scaling of 255 for increased display visibility.

```
196         kernel = np.ones(kSize, dtype="float") * (255.0 / (kSize[0] * kSize[1]))
```

Within the `main` function, the template convolution is used for arbitrary kernels in order to demonstrate the effect of asymmetric dimensions on the image border. Notice how the top/bottom borders in figure 6 are larger than the left/right.

```
> ./hybrid.py -k 31 3 -i Data/bird.bmp Data/plane.bmp -o arb.jpg
```

5 Usage

The `argparse` python module was used to generate arguments for the script. It allows the user to define kernel sizes, cutoff frequencies and input images etc.

```
usage: hybrid.py [-h] -i IMAGE IMAGE [-k KERNEL KERNEL] [-c CUTOFF CUTOFF]
                 [-o OUTPUT] [-v VISUAL] [-f] [-s]
```

Figure 6: Simple averaging with a kernel size of 31×3.

```
optional arguments:
  -h, --help            show this help message and exit
  -i IMAGE IMAGE, --image IMAGE IMAGE
                        Path to input images.
  -k KERNEL KERNEL, --kernel KERNEL KERNEL
                        Kernal size, e.g. 5 7. Note: first image in list will
                        be used.
  -c CUTOFF CUTOFF, --cutoff CUTOFF CUTOFF
                        Gaussian cutoff frequencies, e.g. 5 5.
  -o OUTPUT, --output OUTPUT
                        Path to output image file.
  -v VISUAL, --visual VISUAL
                        Path to output visualisation file.
  -f, --fourier         Use Fourier convolution.
  -s, --sobel           Run Sobel edge detection on the first image.
```

Example usage is shown below, with the convolution taking place between "dog.bmp" and "cat.bmp" with the cutoff frequency of $\sigma = 4$ for each. The output hybrid image is saved to "hybrid.jpg" and the visualisation is saved to "visual.jpg". The `-f` flag indicates that the Fourier method of convolution will be used.

```
> ./hybrid.py -i data/dog.bmp data/cat.bmp -c 4 4 -o hybrid.jpg -v visual.jpg -f
[Data/dog.bmp]    Generating low pass image...
[Data/dog.bmp]    Calculating Gaussian kernel...
./hybrid.py:85: ComplexWarning: Casting complex values to real discards the imaginary part
  output[:, :, colour] = np.fft.fftshift(np.fft.ifft2(Fi * Fk)) / 255
[Data/cat.bmp]    Generating high pass image...
[Data/cat.bmp]    Calculating Gaussian kernel...
Creating hybrid image...
Creating visualisation...
Done.
```

The full code can be found in the archive submitted along with the report.