Resizing, Kernel Inversion, and Restoration of Native Resolutions

kageru

kageru.moe
kageru@encode.moe

June 1, 2017

Abstract

For many years, publishers and digital media distributors have used various upscaling techniques to digitally enhance older content or content that was produced at resolutions lower than the current hardware standards. Prominent examples of this are Blu-ray re-releases of old standard definition anime productions and more recently UHD Blu-ray releases of upscaled animation and live-action movies alike.
The ability to accurately restore the original image can be useful for a number of reasons. Firstly, an upscaled 1080p video file will be significantly larger than the original 720p version of the same video, despite not containing any additional information. This wastes bandwidth during the distribution process and hard drive space for long-term storage or archiving. Due to the higher number of pixels in each frame, many computations, such as video filters and various compression techniques, will also be more time-consuming. It can therefore be beneficial to filter, process, compress, and distribute visual media in its "native" resolution, especially when operating in environments where adherence to standards is not necessary.
This paper will be divided into three sections. The first section will be dedicated to exlaining the theory behind image resizing which will be necessary to understand the later chapters. Section two will explore different ways to find the original resolution of any given upscale, and section three will cover ways to restore the original image data.

I. Image Resizing

When referring to digital image processing, the terms "resizing" and "scaling" are used to describe an action that alters the size of a given image without affecting its overall structure and contents. There are different algorithms for different types of content, some very specific and others broadly applicable.
Disregarding neural networks and other evolutionary algorithms, the basic principle is the same for all of these. A number of reference pixels n is selected and used to calculate the value of a new pixel as a weighed average of all surrounding pixels with $n \in \mathbb{N}\backslash\{0\}$. The case $n = 1$ describes a point or nearest neighbor resizer. Other commonly used kernels can be found in Table 1.

Table 1: *Common resizers*

Kernel	n	Comments
Linear	2	Simple linear interpolation
Cubic	4	Curves (e.g. Lagrange)
Bilinear	4	Linear in x and y direction.
Bicubic	16	4 separate cubic curves

In the case of neural networks, a traditional resizer may be used to initially resize the image to its target resolution. Then, a number of convolutions or similar filters is applied to each pixel to achieve the desired result, usually super-res with a high sharpness and level of detail. Prominent examples of neural network based image upscaling include waifu2x,[1] NNEDI3,[2] and Google's RAISR[3].

[1] github.com/nagadomi/waifu2x
[2] avisynth.nl/index.php/Nnedi3
[3] I'd really recommend reading this article to get a general idea of the technology behind neural resizers: research.googleblog.com/2016/11/enhance-raisr-sharp-images-with-machine.html. For a more technical and detailed description, read the corresponding paper here: arxiv.org/abs/1606.01299

While the latter examples will produce seemingly arbitrary outputs depending on the data used to train the models, the former are deterministic.[4] This means that such a resize operation will always yield the same result and can be inverted as long as no information was lost.[5,6]

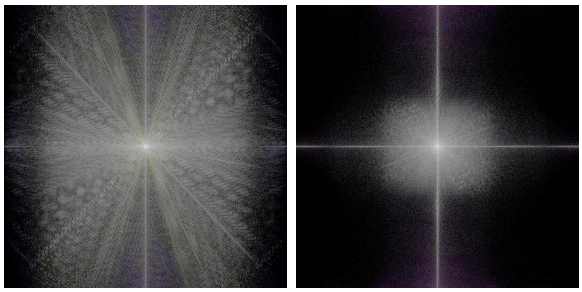
II. Native Resolutions

In the context of this paper, the term "native resolution" will be used to describe the resolution that any given material was filmed, scanned, drawn, animated, or generally produced at. A video file originally filmed in 720p that was later upscaled to 1080p for the Blu-ray release will therefore have a resolution of 1920x1080, but its native resolution will be 1280x720. Restoring this original state can be beneficial in a number of scenarios, such as processing, compression, distribution, and storage. Even if the result has to be consistent with the original release—possibly having to adhere to the same hardware standards—reversing a "cheap" upscale[7] and using a more sophisticated resize algorithm to create a subjectively better high resolution image is also possible.[8]
There are multiple ways to find the native resolutions[9] of an upscale image. Two of them will be described in this chapter.

i. Fourier Transforms

The first method is fairly well-known in certain online communities and has been used for over a decade to identify upscales of Japanese animation.[10] After transforming the image into the frequency domain, the frequency of the image components can be used to draw conclusions regarding the material's native resolution. Very small image components and fine details are represented high frequencies, whereas the basic image structure will be represented by mid to low frequencies. The absence of high frequencies is likely caused by an upscale, as an upscaled image does not contain high-frequency information. Figure 1 illustrates this.

Figure 1: *Fourier analyses. Left: native 1080p video, right: SD upscale. ©Anibin*



ii. Error measurements

Another way to find native resolutions is simple trial and error. This is advantageous as it is simple to implement and can identify multiple resolutions within one image, however, the process is slower than Fourier transforms.[11]
An inverse resizer[12] is used to downscale to all possible resolution or all resolutions within a sane range (e.g. 400p-1079p). The resulting image is then upscaled back to its source resolution and the two images are compared. At the correct resolution, the error will be significantly smaller than for all other resolutions. An example implementation of this using the frame server software Vapoursynth has been released by the author. The relevant part is shown in figure 2.[13]

[4] Hypothetically, if the exact training model is known, the filtering can be reversed by combining deconvolutions with a reversal of the "cheap" resizer that was used during the first resize step.
[5] Which usually translates to "if the image was upscaled".
[6] NNEDI3 is a special case, as it only interpolates every other pixel in an image without touching the reference pixels. Thus, removing every other row of pixels in an image will reverse the effects of NNEDI upsampling. Only magnification by a factor 2^n for $n \in \mathbb{N}$ is supported.
[7] such as bilinear resizing
[8] Companies like the German streaming service "Anime on Demand" do this to resize native 720p material to higher resolutions to conform to their standards.
[9] When dealing with synthetic material, is technically possible to have multiple native resolutions in one frame.
[10] Please refer to anibin.blogspot.de for examples.
[11] A few seconds per frame without multithreading is still sufficiently fast. After all, only a few frames per video have to be analyzed.
[12] More details in chapter 3
[13] The full script is available here: gist.github.com/kageru/549e059335d6efbae709e567ed081799

Figure 2: *Measuring the resize error*

```
def get_error(source, width, height):
    down = inverse_bilinear(source, width, height)
    up = down.resize.Bilinear(source.width, source.height)
    error = core.std.Expr([source, up], 'x y - abs') # absolute difference
    error = core.std.PlaneStats(error)
    luma_error = mask.get_frame(0).props.PlaneStatsAverage
    return luma_error # this is a number between 0 and 1
```

III. Inverse Resizing

As mentioned in chapter 1, "traditional" resizers are deterministic mathematical formulas which can be inverted if all necessary parameters are known. The process can be described as $A * x = b$, where A is a matrix that depends on the resize kernel and parameters, x is a vector of input pixels, and b is the vector of output pixels. If the originally used resize kernel is known, A can be calculated. b is known as it is part of the final image. With this knowledge, the linear equation can be solved to restore the original pixels in x.[14]

[14] For a more detailed explanation, read the description of Frechdachs' Vapoursynth-Descale plugin here: github.com/Frechdachs/vapoursynth-descale