The Dangers Behind Image Resizing

Source: The dangers behind image resizing (zuru.tech)

Here, I just want to test with my own version.

Backgrounds

Image resizing is a very common geometrical transformation of an image, and it simply consists of a scaling operation.

Since it is one of the most common image processing operations, you can find its implementation in all image processing libraries. Because it is so common, you can expect that the behavior is **well defined** and **will be the same** among the libraries.

Unfortunately, this is **not true** because some little implementation details differ from library to library. If you are not aware of it, this could create a lot of trouble for your applications.

Why the behaviour of resizing is different?

The definition of scaling function is mathematical and should never be a function of the library being used. Unfortunately, implementations differ across **commonly-used** libraries and mainly come from how it is done the **interpolation**.

Usually, when you compute source coordinates, you get floating-point numbers, so you need to decide how to choose which source pixel to copy into the destination.

The naive approach is to round the coordinates to the nearest integers (nearest-neighbor interpolation). However, better results can be achieved by using more sophisticated interpolation methods, where a polynomial function is fit into some neighborhood of the computed pixel ($f_x(x,y)$, $f_x(x,y)$), and then the value of the polynomial at ($f_y(x,y)$, $f_y(x,y)$) is taken as the interpolated pixel value.

The problem is that different library could have some little differences in how they implement the interpolation filters but above all, if they introduce the **anti-aliasing filter**. In fact, if we interpret the image scaling as a form of image resampling from the view of the Nyquist sampling theorem, downsampling to a smaller image from a higher-resolution original can only be carried out after applying a suitable 2D anti-aliasing filter to prevent aliasing artifacts.

OpenCV that could be considered the standard de-facto in image processing does not use an anti-aliasing filter. On the contrary, **Pillow**, probably the most known and used image processing library in Python, introduces the anti-aliasing filter.

Comparison of Libraries

The tested libraries:

1. OpenCV v4.5.3: `cv2.resize()`
2. Pillow Image Library (PIL) v8.2.0: `Image.resize()`
3. Tensorflow v2.5.0: `tf.image.resize()`

   This method has a flag anti-alias to enable the anti-aliasing filter (default is false). We tested the method in both cases, either flag disabled and enabled.

4. PyTorch v1.9.0: `torch.nn.functional.interpolate()`

   PyTorch has another method for resizing, that is torchvision.transforms.Resize. We decided not to use it for the tests because it is a wrapper around the **PIL library**, so that the results will be the same compared to Pillow.

Each method supports a set of filters. Therefore, we chose a common subset present in almost all libraries and tested the functions with them. In particular, we chose:

- **nearest**: the most naive approach that often leads to aliasing. The high-frequency information in the original image becomes misrepresented in the downsampled image.

- **bilinear**: a linear filter that works by interpolating pixel values, introducing a continuous transition into the output even where the original image has discrete transitions. It is an extension of linear interpolation to a rectangular grid.

- **bicubic**: similar to bilinear filter but in contrast to it, which only takes 4 pixels (2×2) into account, bicubic interpolation considers 16 pixels (4×4). Images resampled with bicubic interpolation should be smoother and have fewer interpolation artifacts.

- **lanczos**: calculate the output pixel value using a high-quality Lanczos filter (a truncated sinc) on all pixels that may contribute to the output value. Typically it is used on an 8x8 neighborhood.

- **box** (aka area): is the simplest linear filter; while better than naive subsampling above, it is typically not used. Each pixel of the source image contributes to one pixel of the destination image with identical weights.

Qualitative Results

To better see the different results obtained with the presented functions, we follow the test made in 3. We create a synthetic 128 x 128 image with a circle of thickness 1 px, and we resized it with a downscaling factor of 4 (destination image will be 32 x 32).

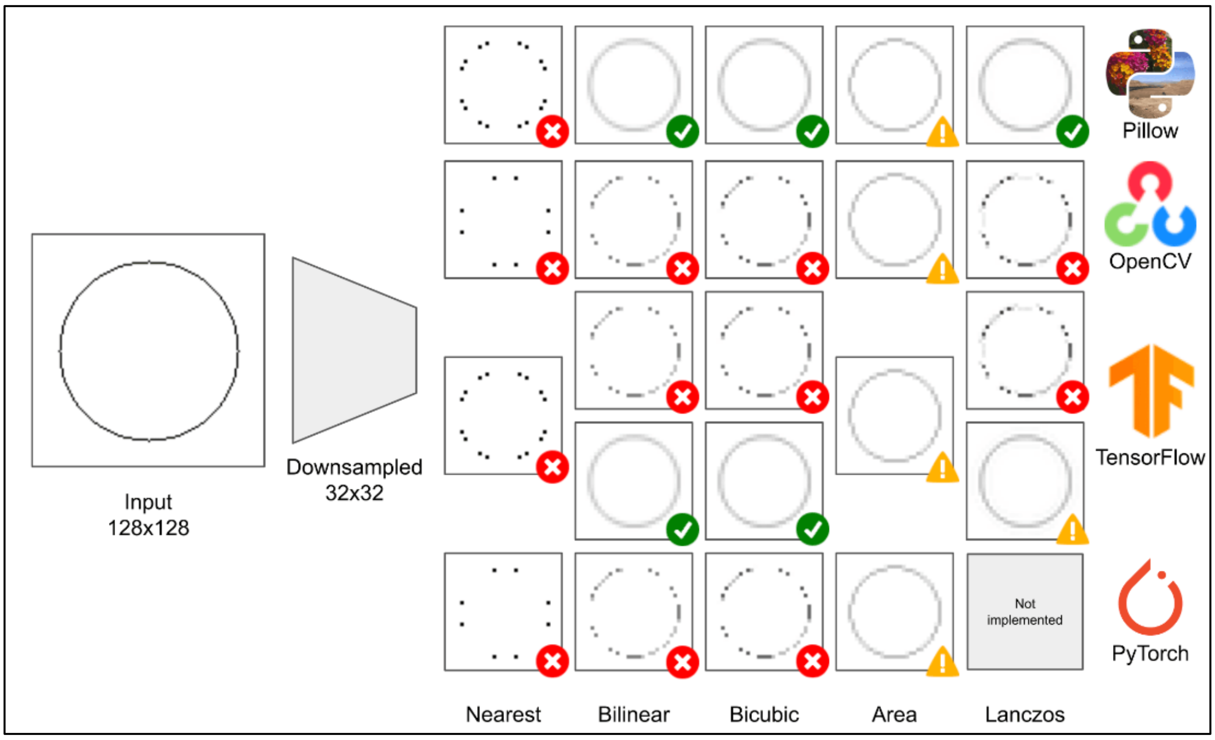


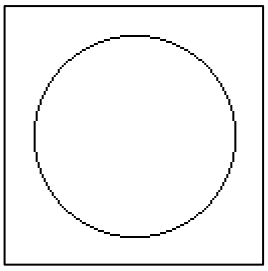
Figure 1 - Qualitative Results by the Article's Author

Real Experiments

So here I am testing the downscaling method for each library, here are my version:

- Pillow v9.2.0: `Image.resize()`
- OpenCV v4.6.0: `cv2.imresize()`
- Tensorflow v2.3.0: `tf.image.resize()`

Here the input image:



Here the results:

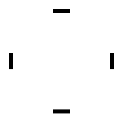
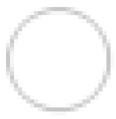
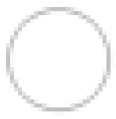
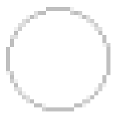
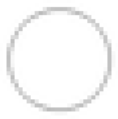
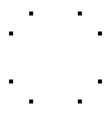
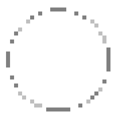
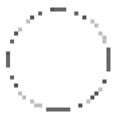
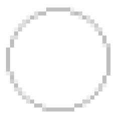
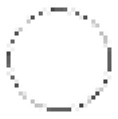
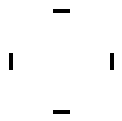
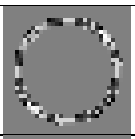
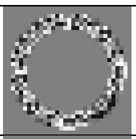
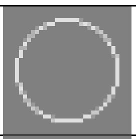
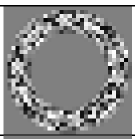
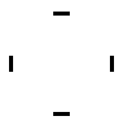
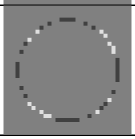
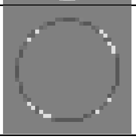
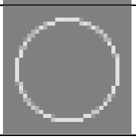
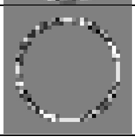
Nearest	Bilinear	Bicubic	Area	Lanczos	Library
					Pillow
					OpenCV
					Tensorflow Anti-alias: True
					Tensorflow Anti-alias: False

Image Resizing in ML Models

Imagine you want to use a model from an ML third-party library. For example, you want to solve a problem of instance segmentation from images. You find an incredible library from Facebook Research, called Detectron2, which contains the state-of-the-art for this class of problems. The authors provide all kinds of functionality, including scripts from training and evaluation on your dataset and even the script for exporting your new trained model. It could be challenging and time-consuming to reproduce the same features from scratch, so you can use the library in your application.

Suppose you train the network on your data. Then you want to put the model into production. Your deployment environment is written in C++, so you export the model as TorchScript, and you use the PyTorch C++ library to load the module and run the experiments. You know that Detectron needs images with a fixed size as input, then you use OpenCV to rescale your input. If you feed the network with the same images, you will expect the same results, but unfortunately, this is not true. You check everything, and it seems correct. If you dig inside the code, you will eventually find Detectron uses Pillow to resize the input image, and, as you could see in the section before, the results could be very different.

Conclusions

This article demonstrated that even if image resizing is one of the most used image processing operations, it could hide some traps. It is essential to carefully choose the library you want to use to perform resize operations, particularly if you're going to deploy your ML solution. Since we noticed that the most correct behavior is given by the Pillow resize and we are interested in deploying our applications in C++, it could be useful to use it in C++. The Pillow image processing algorithms are almost all written in C, but they cannot be directly used because they are designed to be part of the Python wrapper. We, therefore, released a porting of the resize method in a new standalone library that works on cv::Mat so it would be compatible with all OpenCV algorithms. You can find the library here: pillow-resize. We want to remark again that the right way to use the resize operation in an ML application is to export the algorithm's pre-processing step. In this way, you are sure that your Python model works similarly to your deployed model.