Simple Transformations

This section includes simple but very common image transformations
like cropping, translation, scaling and rotation.

[res] image.crop([dst,] src, x1, y1, [x2, y2])

Crops image `src` at coordinate `(x1, y1)` up to coordinate
`(x2, y2)`. The coordinate indexing is zero-based and `(x2, y2)` is non-inclusive.
If `dst` is provided, it is used to store the output
image. Otherwise, returns a new `res` Tensor.

```
-- The indexing starts with 0 and 2 is non-inclusive coordinate.
> require('image')
> image.crop(torch.Tensor(3, 2, 2), 0, 0 , 2, 2) -- crop is a
correct crop and the result is 3x2x2 tensor.
(1,.,.) =
  0  0
  0  0

(2,.,.) =
  0  0
  0  0

(3,.,.) =
  0  0
  0  0
[torch.DoubleTensor of size 3x2x2]
```

[res] image.crop([dst,] src, format, width, height)

Crops a `width x height` section of source image `src`. The argument
`format` is a string specifying where to crop: it can be “c”, “tl”, “tr”,
“bl” or “br” for center, top left, top right, bottom left and bottom right,
respectively. If `dst` is provided, it is used to store the output
image. Otherwise, returns a new `res` Tensor.

[res] image.translate([dst,] src, x, y)

Translates image `src` by `x` pixels horizontally and `y` pixels vertically. If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

[res] image.scale(src, width, height, [mode])

Rescale the height and width of image `src` to have width `width` and height `height`. Variable `mode` specifies type of interpolation to be used. Valid values include bilinear (the default), bicubic, or *simple* interpolation. Returns a new `res` Tensor.

[res] image.scale(src, size, [mode])

Rescale the height and width of image `src`. Variable `size` is a number or a string specifying the size of the result image. When `size` is a number, it specifies the maximum height or width of the output. When it is a string like `WxH` or `MAX` or `^MIN`, `*SC` or `*SCn/SCd` it specifies the `height x width`, maximum height or width of the output, minimum height or width of the output, scaling factor (number), or fractional scaling factor (int/int), respectively.

[res] image.scale(dst, src, [mode])

Rescale the height and width of image `src` to fit the dimensions of Tensor `dst`.

[res] image.rotate([dst,], src, theta, [mode])

Rotates image `src` by `theta` radians.
If `dst` is specified it is used to store the results of the rotation.

Variable `mode` specifies type of interpolation to be used. Valid values include *simple* (the default) or *bilinear* interpolation.

[res] image.polar([dst,], src, [interpolation], [mode])

Converts image `src` to polar coordinates. In the polar image, angular information is in the vertical direction and radius information in the horizontal direction.
If `dst` is specified it is used to store the polar image. If `dst` is not specified, its size is automatically determined. Variable `interpolation` specifies type of interpolation to be used. Valid values include *simple* (the default) or *bilinear* interpolation. Variable `mode` determines whether the *full* image is converted to the polar space (implying empty regions in the polar image), or whether only the *valid* central part of the polar transform is returned (the default).

[res] image.logpolar([dst,], src, [interpolation], [mode])

Converts image `src` to log-polar coordinates. In the log-polar image, angular information is in the vertical direction and log-radius information in the horizontal direction.
If `dst` is specified it is used to store the polar image. If `dst` is not specified, its size is automatically determined. Variable `interpolation` specifies type of interpolation to be used. Valid values include *simple* (the default) or *bilinear* interpolation. Variable `mode` determines whether the *full* image is converted to the log-polar space (implying empty regions in the log-polar image), or whether only the *valid* central part of the log-polar transform is returned (the default).

[res] image.hflip([dst,] src)

Flips image `src` horizontally (left<->right). If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

[res] image.vflip([dst,], src)

Flips image `src` vertically (upsize<->down). If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

[res] image.flip([dst,] src, flip_dim)

Flips image `src` along the specified dimension. If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

[res] image.minmax{tensor, [min, max, ...]}

Compresses image `tensor` between `min` and `max`.
When omitted, `min` and `max` are infered from `tensor:min()` and `tensor:max()`, respectively.
The `tensor` is normalized using `min` and `max` by performing :

```
tensor:add(-min):div(max-min)
```

Other optional arguments (`...`) include `symm`, `inplace`, `saturate`, and `tensorOut`.
When `symm=true` and `min` and `max` are both omitted, `max = min*2` in the above equation. This results in a symmetric dynamic range that is particularly useful for drawing filters. The default is `false`.
When `inplace=true`, the result of the compression is stored in `tensor`.
The default is `false`.
When `saturate=true`, the result of the compression is passed through a function that clips the values between 0 and 1 (i.e. anything below 0 is set to 0, anything above 1 is set to 1).
When provided, Tensor `tensorOut` is used to store results.
Note that arguments should be provided as key-value pairs (in a table).

[res] image.gaussianpyramid([dst,] src, scales)

Constructs a Gaussian pyramid of scales `scales` from a 2D or 3D `src` image or size `[nChannel x] width x height`. Each Tensor at index `i` in the returned list of Tensors has size `[nChannel x] width*scales[i] x height*scales[i]`.

If list `dst` is provided, with or without Tensors, it is used to store the output images. Otherwise, returns a new `res` list of Tensors.

Internally, this function makes use of functions image.gaussian, image.scale and image.convolve.