BMI 826/CS 838 Homework Assignment 1

Chun-Min Jimmy Chang (cchang253@wisc.edu)

October 2, 2019

---

1. Overview

The goal of this assignment is to implement basic image processing functions and assemble them into a data augmentation pipeline. These functions are often used as the "front-end" for machine learning models. The assignment will cover image processing techniques, including resizing, cropping, color manipulation and rotation and basic data input pipeline.

1.1 Image Resizing in class Scale

To match the shortest side to a pre-specified size, the resizing ratio is calculated by that size over the length of the shortest side. That is, $\text{ratio} = \text{size}/\min(\text{height}, \text{width})$. Then, resize to a specific size by using resize_image in utils.py.

```
def __call__(self, img):
    """
    Args:
        img (numpy array): Image to be scaled.

    Returns:
        numpy array: Rescaled image
    """
    # sample interpolation method
    interpolation = random.sample(self.interpolations, 1)[0]

    # scale the image
    if isinstance(self.size, int):
      h, w, c = img.shape
      ratio = self.size/min(h,w)
      img = resize_image(img, (int(w*ratio),int(h*ratio)), interpolation)
      return img
    else:
      img = resize_image(img, self.size, interpolation)
      return img
```

1.2 Image Cropping in class RandomSizedCrop

The implementation of image cropping is summarized as below.

(1) Given a random area and aspect ratio, we can calculate the height and width by

$e1.\, Area = height \times width$, and

$e2.\, height = aspect\_ratio \times width$ or $e3.\, width = aspect\_ratio \times height$.

One possibility is solving $e1.$ and $e2.$

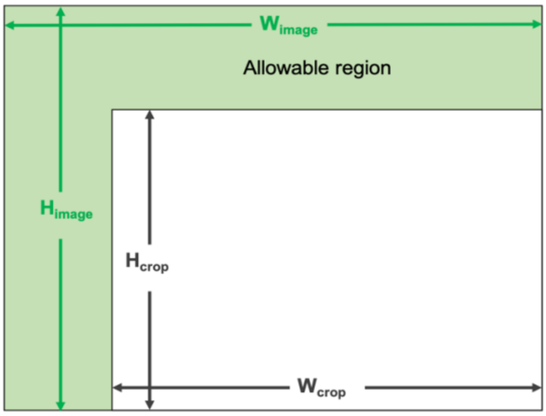
$width = \sqrt{Area/aspect\_ratio}$ and $height = \sqrt{Are \times aspect\_ratio}$

Another possibility is solving $e1.$ and $e3.$

$width = \sqrt{Are \times aspect\_ratio}$ and $height = \sqrt{Area/aspect\_ratio}$

(2) Check whether any of the above two possibilities can have a valid crop.

(3) If there is a valid crop, we randomly sample a point from the *allowable region* as the top-left corner of the cropped image. The *allowable region* is defined as below.



The top-left corner of the cropped image should be located in the green region.

Let the top-left corner be at (x, y), and thus:

$x \in [0, \mathrm{H}_{\mathrm{image}} - \mathrm{H}_{\mathrm{crop}}]$, and

$y \in [0, \mathrm{W}_{\mathrm{image}} - \mathrm{W}_{\mathrm{crop}}]$.

(4) If there is no valid crop but want to have a square crop, we crop the patch in the center.

(5) If there is no valid crop and a specific size is given, we resize the whole image directly.

```
def __call__(self, img):
  # sample interpolation method
  interpolation = random.sample(self.interpolations, 1)[0]

  for attempt in range(self.num_trials):

    # sample target area / aspect ratio from area range and ratio range
    area = img.shape[0] * img.shape[1]
    target_area = random.uniform(self.area_range[0], self.area_range[1]) * area
    aspect_ratio = random.uniform(self.ratio_range[0], self.ratio_range[1])

    #################################################################################
    # Fill in the code here
    #################################################################################
    # compute the width and height
    # note that there are two possibilities
    # crop the image and resize to output size

    # h*w = (w*aspect_ratio)*w = target_area
    hw_list = []
    width = (target_area/aspect_ratio)**0.5
    height = width*aspect_ratio
    height, width = int(height), int(width)

    # two possibilities:
    hw_list = [(height, width), (width,height)]
    for h, w in hw_list:
      # find a suitable crop area and aspect_ratio
      if h < img.shape[0] and w < img.shape[1]:
        print("found suitable crop.")
        randX = random.sample(range(0, img.shape[0]-h), 1)[0]
        randY = random.sample(range(0, img.shape[1]-w), 1)[0]
        if isinstance(self.size, int):
          img = resize_image(img[randX:(randX+h), randY:(randY+w)],
                             (self.size, self.size),
                             interpolation)
        else:
          img = resize_image(img[randX:(randX+h), randY:(randY+w)],
                             self.size,
                             interpolation)
        return img
```

```
  # Fall back
  if isinstance(self.size, int):
    im_scale = Scale(self.size, interpolations=self.interpolations)
    img = im_scale(img)
    #################################################################################
    # Fill in the code here
    #################################################################################
    # with a square sized output, the default is to crop the patch in the center
    # (after all trials fail)
    imgH, imgW = img.shape[0], img.shape[1]
    if imgH > imgW:
      offset = imgH - imgW
      img = img[offset//2:-(offset-offset//2),:]
    else:
      offset = imgW - imgH
      img = img[:, offset//2:-(offset-offset//2)]
    return img
  else:
    # with a pre-specified output size, the default crop is the image itself
    im_scale = Scale(self.size, interpolations=self.interpolations)
    img = im_scale(img)
    return img
```

1.3 Color Jitter in class RandomColor

For every pixel value, we perform the following procedures by channels:

(1) Convert the 8-bit unsigned integer to a floating float representation.

(2) Multiply a randomly sampled constant, (1+alpha).

(3) Clip the value greater than 255 and less than 0 to avoid

(4) Convert back to the 8-bit unsigned integer format.

However, this algorithm takes 28.035 seconds for a 512-by-512 RGB image.

```
def __call__(self, img):
  ################################################################################
  # Fill in the code here
  ################################################################################
  for c in range(img.shape[2]):
    alpha = random.uniform(-self.color_range, self.color_range)
    for i in range(img.shape[0]):
      for j in range(img.shape[1]):
        img[i,j,c]= np.clip(img[i,j,c]*(1+alpha), 0, 255).astype(np.uint8)
  return img
```

Following the hint, I propose a fast implementation, class FastRandomColor. The idea is that since there are only 256 values in an image (integers from 0 to 255), we can calculate the transformation result of these 256 values in advance and create a lookup table to do the color jitter effect. For example, 5 will be mapped to 6 if alpha = 0.2. In this mean, we only do 256 multiplications, 256 clip operations, and n^2 re-assignments. However, the former approach takes n^2 multiplications, n^2 clip operations, and n^2 re-assignments. As a result, this fast algorithm takes only 1.985 seconds for a 512-by-512 RGB image.

```
class FastRandomColor(RandomColor):
  """
  A fast implementation of RandomColor using pre-calculated lookup table.
  """
  def __init__(self, color_range):
    super().__init__(color_range)

  def __call__(self, img):
    for c in range(img.shape[2]):
      alpha = random.uniform(-self.color_range, self.color_range)
      color_dict = {}
      for i in range(256):
        color_dict[i] = np.clip(i*(1+alpha), 0, 255).astype(np.uint8)
      for i in range(img.shape[0]):
        for j in range(img.shape[1]):
          img[i,j,c]=color_dict[img[i,j,c]]
    return img
```

Knowing that NumPy uses BLAS to accelerate matrix multiplication, I wrote a faster implementation that outperforms the above two implementations. The algorithm directly does a channel-wise transformation, instead of three for-loops to loop through every pixel in every channel. This algorithm takes 0.00718 seconds for a 512-by-512 RGB image. Note that my computer uses Intel MKL as the BLAS library. The complete comparison among the three implementation is summarized in the Result section.

```python
class FasterRandomColor(RandomColor):
  """
  A faster implementation of RandomColor using matrix calculations.
  """
  def __init__(self, color_range):
    super().__init__(color_range)

  def __call__(self, img):
    for c in range(img.shape[2]):
      alpha = random.uniform(-self.color_range, self.color_range)
      img[:,:,c] = np.clip(img[:,:,c].astype(float)*(1+alpha), 0, 255).astype(np.uint8)
    return img
```

1.4 Rotation in class RandomRotate

(1) Create the rotation matrix by cv2.getRotationMatrix2D using the image center as the rotation center and without scaling.

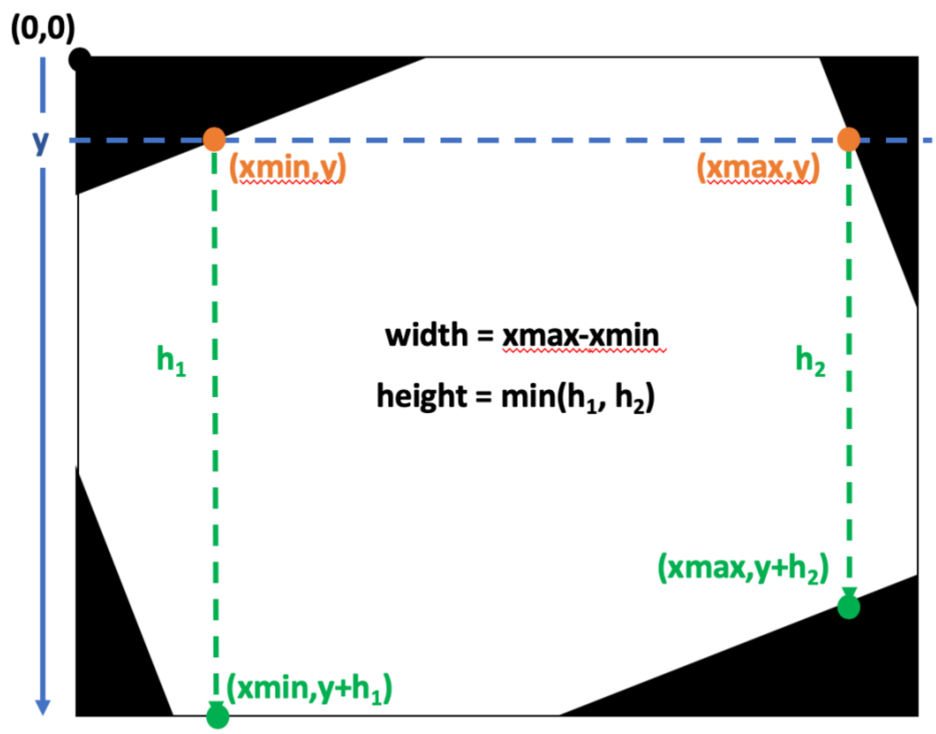
(2) Apply the cv2.warpAffine method with the rotation matrix of (1) on the image.

(3) Find the rectangular region with the maximum area by the below algorithm.
Loop through the rows of image. For each row, y

a. Find the minimum and maximum indices of the nonzero elements at the row y, denoted as xmin and xmax, respectively. Then, the width of the maximum rectangular region is xmax – xmin.

b. Find the maximum index of the nonzero element at the column xmin and xmax respectively, denoted as y1 and y2. The possible height found at these two points is either h1 = y1 – y or h2 = y2 – y.

c. The minimum of h1 and h2 is the height of the maximum rectangular region. The following figure visualizes the procedures.



(4) Crop the maximum rectangular region.

```python
def __call__(self, img):
  # sample interpolation method
  interpolation = random.sample(self.interpolations, 1)[0]
  # sample rotation
  degree = random.uniform(-self.degree_range, self.degree_range)
  # ignore small rotations
  if np.abs(degree) <= 1.0:
    return img

  ##################################################################################
  # Fill in the code here (Done)
  ##################################################################################
  # get the max rectangular within the rotated image

  # 2D rotation matrix
  M = cv2.getRotationMatrix2D((img.shape[1]//2, img.shape[0]//2), degree,1)
  img = cv2.warpAffine(img, M, (img.shape[1], img.shape[0]))

  # to find the max rectangular
  opt = 0
  # to work in grayscale
  arr = np.mean(img,axis=2)

  # loop over each row
  for y in range(arr.shape[0]):
    # find the nonzero elements with min-index and max-index
    index = np.where(arr[y,:]>0)[0]
    xmin, xmax = min(index), max(index)
    width = xmax - xmin

    # find heights
    height = min(max(np.where(arr[y:, xmin]>0)[0]), max(np.where(arr[y:, xmax]>0)[0]))
    area = height*width
    if area > opt:
      opt = area
      r, c, h, w = y, xmin, height, width
  return img[r:(r+h), c:(c+w)]
```

2. Result

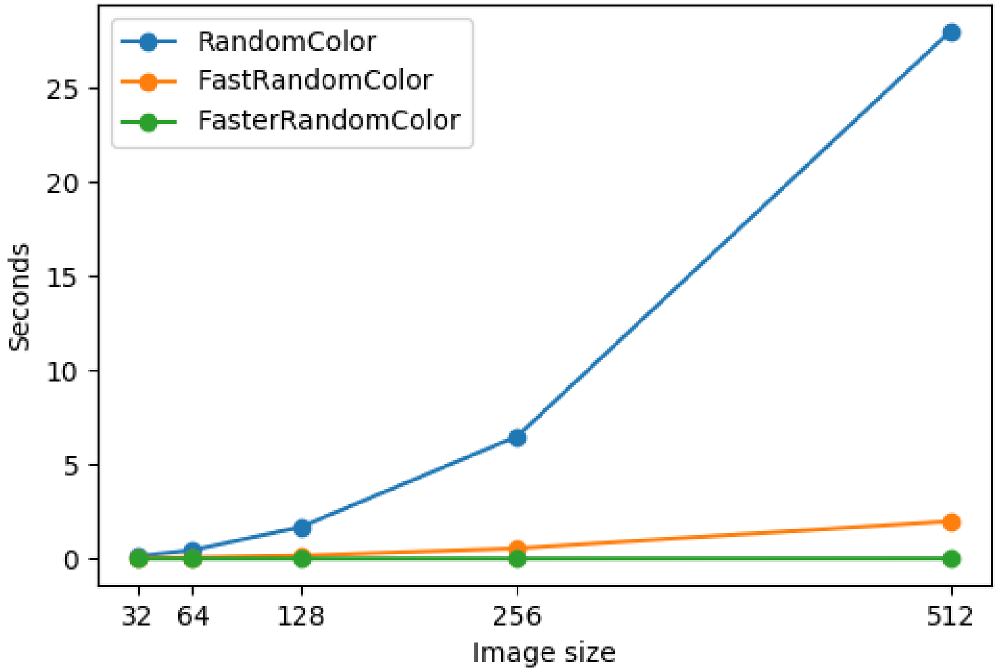
2.1 Image Outputs

Combining multiple transformations together can generate a wide variety of augmented data. Thanks to applying random sampling on the top-left corner of the cropped image, we still can generate diverse images although there are some image orientations are similar.

The transformed result of Image1 and Image2.

2.2 Comparison among Different Implementations of Color Jitter

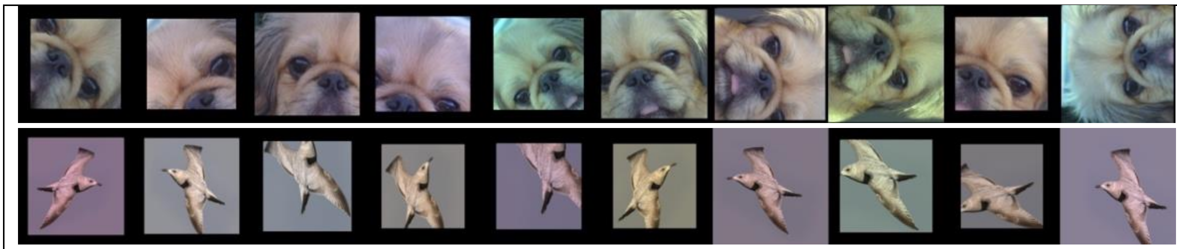
Here I evaluate those three implementations using different image sizes. It is shown that the execution time of RandomColor is much slower than the other two and grows exponentially along the increase of image size. However, FastRandomColor creates a lookup table to record the transformation result of 256 pixel values, making the time complexity in terms of multiplication being a constant, and avoids repetitive multiplications. Furthermore, FasterRandomColor leverages the fast matrix multiplication of BLAS to achieve the best performance, down to 0.00718 seconds.



3. Data Augmentation and Input Pipeline

3.1 Composition of Image Transforms

The order of some transformations would be crucial. Taking RandomSizedCrop as an example. The provided composition uses RandomSizedCrop(224) as the final step to make sure every output image is a 224-by-224 square image. If we interchange the order of RandomSizedCrop(224) and RandomRotate(90), then the output may a rectangular image with random width and height due to the step of finding the maximum rectangular region in RandomRotate, as shown below.



Interchanging the order of RandomSizedCrop and RandomRotate causes inconsistent image sizes after transformations. To make them have the same size, I put images in the center and fill the peripheral region with black pixels.

However, the order of some transformations is not important and can interchange, such as RandomColor and RandomRotate. Because RandomColor manipulates RGB three color channels and RandomRotate is a kind of spatial mapping that changes the position of every pixel, these two transformations operate independently and do not influence each other.

Interchanging the order of RandomColor and RandomRotate has no impact on outputs.

3.2 Data Input

PyTorch provides an easy-to-use data loader and it has some characteristics. First, DataLoader always loops through the entire dataset. If there are 5 images in total and batch size is set at 2, then the output series will be 2 images, 2 images, and 1 image. In this way, PyTorch's DataLoader can guarantee every image will be seen once in one epoch. In addition, if you would like to make every batch have the same number of samples in PyTorch's DataLoader, we can set "drop_last" to "True" to drop the last incomplete batch.

Secondly, when setting "shuffle" argument to be "True", the order of data will be changed randomly at every epoch. Shuffling data is a commonly used way to increase variation among data batches and reduce data dependency. In particular, when using batch normalization in the network, we must shuffle data at every epoch or every batch statistic will not change and probably ruin model generalization.

Third, DataLoader has an argument "num_workers", which means how many subprocesses to use for data loading. However, when I set num_workers greater than zero, it shows an error message "RuntimeError: DataLoader worker (pid(s) 10941) exited unexpectedly". I check some relevant issues in the GitHub and it seems like the problem may result from using cv2 in DataLoader. (see SsnL's reply in https://github.com/pytorch/pytorch/issues/4969)

Last but not least, I evaluate the time to loop through the example dataset using different setting in DataLoader. Note that the default setting is batch_size=5, shuffle=True, and uses comp_transforms provided in the example.

- The effect of batch_size:

	batch_size = 1	batch_size =5
Time (second)	0.258469	0.262865

- The effect of shuffle:

	shuffle=False	shuffle=True
Time (second)	0.261656	0.262865

- The effect of transform complexity:

	simple_transforms	comp_transforms
Time (second)	0.0129431	0.262865

where the details of these two transforms are listed in below

simple_transforms	comp_transforms
tfs = [] tfs.append(RandomSizedCrop(224)) tfs.append(ToTensor()) simple_transforms = Compose(tfs)	tfs = [] tfs.append(Scale(320)) tfs.append(RandomHorizontalFlip()) tfs.append(RandomColor(0.15)) tfs.append(RandomRotate(30)) tfs.append(RandomSizedCrop(224)) tfs.append(ToTensor()) comp_transforms = Compose(tfs)

From the above evaluation, we found that the bottleneck of data loading here is the transform complexity. When using a simple transform, loading data for one epoch only takes 0.013 seconds, but it takes 0.26 seconds while using the provided complex transform.

Environment

OS: MacOS Mojave Version 10.14.6
Processor: Intel Core i5 2.6GHz
Memory: 8GB 1600 MHz DDR3

Python 3.6.3
torch==1.2.0
torchvision==0.4.0
Pillow==4.3.0
opencv-python==4.0.0.21
numpy==1.17.1
matplotlib==2.2.0