CAP 4410: Computer Vision Project 1: Transformations

Vivekkumar Patel

1. Introduction

With evolving technology of Artificial Intelligence and Machine Learning visioning the images plays important role in making AI and ML more usable into the real world. For example face recognition application matches the images from its database to confirm the identity and Transformations of the images make it possible. To match the image with face, those applications need to change the size or orientation of the images.

In this project we have been given the skeleton code from instructor and there are different types of transformation function that we needed to complete then apply on the different types of images. Further in this report all of those functions are explained briefly.

2. Transformations

There are total of 12 types of transformations functions and 5 types of helper functions, which helped the those main functions to work properly.

2.1. Translation

Translation of the image means shifting the image vertically and horizontally. Now that can be done in two ways, one which will move the image and keep the the image size same and keep the portion of the output image black with top left corner remaining (0 0) (helper function:- inverse_warp_b), second when we move the image we map the image to the portion of the output image so it does not look like that image is been move or not (helper function:- inverse_warp_a).

When we translate the image, it preserves orientation, angles and length of the images. In this function image can be shift up or down and right or left based on the input value. It takes in two value as input, x and y. If input value is positive, image will shift right or down, if the input value is negative, image will shift left or up. In the following example both, in x direction and in y directions, the value is positive so image will move to the right and down.

Mathematical presentation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{1}$$

The following code is been use to generate the image in the figure 1.

Listing 1. Translation function

```
def translation(self, shift_x, shift_y):
    def t(x, y):
        return(x+shift_x, y+shift_y)
    def t_inv(x, y):
        return(x-shift_x, y-shift_y)

    img_new = self.inverse_warp_a(t, t_inv)

    return img_new.astype(np.uint8)
```

Since the image move vertically and horizontally, there two functions have been use which helps to determine the final coordinates of the image after putting the input value. At the end it will return the image in unit8 format.

I used inverse_warp_a function to plot the image because inverse_warp_a function maps the image after shifting. Here the code which was already given, but I have to add one condition to check h_inv function returns a value out of the input plane's range.

Listing 2. inverse_warp_a function

```
def inverse_warp_a(self, h, h_inv):
   # Get 4 corner points
   cx, cy = [], []
   for fx in [0, self.width - 1]:
       for fy in [0, self.height - 1]:
           x, y = h(fx, fy)
```

```
        x, y = int(x), int(y)
        cx.append(x)
        cy.append(y)

    # Get min and max, then new width and height
    min_x, max_x=int(min(cx)), int(max(cx))
    min_y, max_y = int(min(cy)), int(max(cy))
    width_g = max_x - min_x + 1
    height_g = max_y - min_y + 1

    # Creates empty new image
    img_new = np.zeros((height_g, width_g, self.channels)) \newline

    # Find pixel values and map to new image
    for gy in range(min_y, max_y + 1):
        for gx in range(min_x, max_x + 1):
            fx, fy = h_inv(gx, gy)
            #need to have the condition to check h_inv
            #returns a value out of the input plane's range.
            if (fx >self.shape[1] - 1
                or fx < 0 or fy < 0 or fy > self.shape[0] - 1):
                continue
            fx, fy = int(fx), int(fy)
            img_new[gy - min_y, gx - min_x] = self.img[fy, fx]

    # Returns new image
    return img_new
```

In the above code, it map the the corners of the input image to decide the size of the new image after modification. Then it finds the pixel values using inverse map which will also check the condition of out of plane's range and finally return the new image.

Last, Final output of translation function will be look like the image shown in the figure 1.

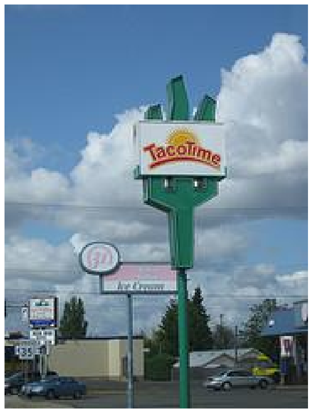

Figure 1. Translation of the image

Figure 2. Translation of the image

2.2. Rotation

Rotation of the image means rotate the image clockwise or counter-clockwise according to the angle. If the angle is positive, image will rotate in counter-clockwise and if the angle is negative, it will rotate clockwise.

In this function, I used inverse_warp_b helper function which was also given but added the condition to check h_inv function returns a value out of the input plane's range. By using this function, it will blackout the rest of the space and cut the image depending the rotation.

Listing 3. inverse_warp_b function

```
def inverse_warp_b(self, h_inv, output_shape):
    # Create empty new image
    if len(output_shape) < 3:
        output_shape = output_shape + (self.channels,)
    img_new = np.zeros(output_shape)

    # Find pixel values and map to new image
    for gy in range(output_shape[0]):
        for gx in range(output_shape[1]):
            fx, fy = h_inv(gx, gy)
            if (fx >self.shape[1] - 1
                or fx < 0 or fy < 0 or fy > self.shape[0] - 1):
                continue
            fx, fy = int(fx), int(fy)
            img_new[max(0, gy), max(0, gx)] = self.img[fy, fx]

    # Returns new image
    return img_new
```

Rotation preserves the angles and length of the image. The mathematical representation of the rotation include the translation variable as well but we can keep it 0 to get only rotation.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & t_x \\ \sin(\theta) & \cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2)$$

The following code is been used to rotate the image shown in the figure 2.

Listing 4. Rotation function

```
def rotate(self, theta):
    # You can delete this, this just makes all black image.
    #img_new = np.zeros(self.shape)

    t = np.array([0, 0])  # we keep it zero

    def h(x, y):
        r = np.array([[np.cos(theta), -np.sin(theta), t[0]],
                      [np.sin(theta), np.cos(theta), t[1]],
                      [0, 0, 1]])
        img_new = r @ np.array([[x], [y], [1]])
        return(img_new[0]/img_new[2], img_new[1]/img_new[2])

    def h_inv(x, y):
        r_inv = np.array([[np.cos(theta), np.sin(theta), -t[0]],
                          [-np.sin(theta), np.cos(theta), -t[1]],
                          [0, 0, 1]])
        img_new = r_inv @ np.array([[x], [y], [1]])
        return(img_new[0]/img_new[2], img_new[1]/img_new[2])

    img_new = self.inverse_warp_b(h_inv, self.shape)
    # Only return new image as uint8
    return img_new.astype(np.uint8)
```

The above code is simply the implementation of the mathematical representation of the rotation. The following image is final output image of the rotation function.

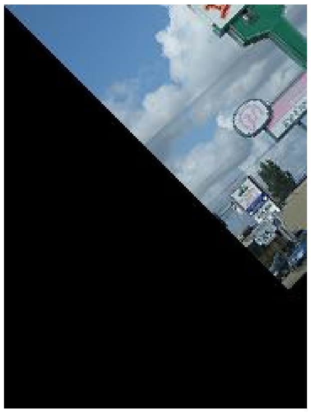

Figure 3. Rotation of the image

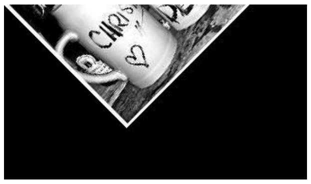

Figure 4. Rotation of the image

2.3. Scaling the Image

Scaling the image means resizing the digital image. It preserve angles. This function will take in an input number and of the number is greater than 100%, the image will expand and if the input is less than 100%, then image will contract. In this function, the image is return using inverse_warp_b helper function, which will give the output image sown in figure 3.

Mathematical representation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s\cos(\theta) & -s\sin(\theta) & t_x \\ s\sin(\theta) & s\cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (3)$$

The mathematical representation of scalling include angel and translation but since we are only scaling the image, I putted the angle and translation to 0. Following code is been use to create the output image.

Listing 5. Scale function

```
def scale(self, scale_percent):
    t = np.array([0, 0])
    s = scale_percent
    theta = 0*np.pi/180
    def h(x, y):
        r = np.array([[s*np.cos(theta), -s*np.sin(theta), t[0]],
                      [s*np.sin(theta), s*np.cos(theta), t[1]],
                      [0, 0, 1]])
        img_new = r @ np.array([[x], [y], [1]])
        return(img_new[0]/img_new[2], img_new[1]/img_new[2])

    def h_inv(x, y):
        r_inv = np.array([[np.cos(theta)/s, np.sin(theta)/s, -t[0]],
                          [-np.sin(theta)/s, np.cos(theta)/s, -t[1]],
                          [0, 0, 1]])
        img_new = r_inv @ np.array([[x], [y], [1]])
        return(img_new[0]/img_new[2], img_new[1]/img_new[2])

    img_new = self.inverse_warp_b(h_inv, self.shape)

    # Only return new image as uint8
    return img_new.astype(np.uint8)
```

Finally the output image after scaling the image shown below.

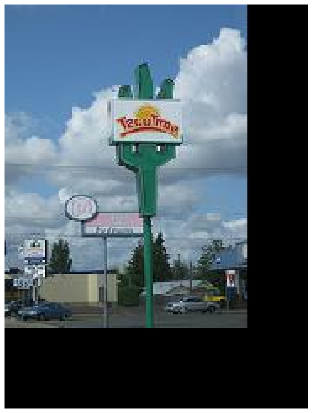

Figure 5. Scale image

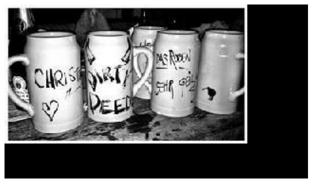

Figure 6. Scale image

2.4. Affine Transformation

Affine transformation preserve points, lines and planes means only parallelism but chnages orientation, angle and lenghts. The affine transformation has vector of 6 parameters and keeps the parallel lines of the image parallel.

Mathematical representation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & t_x \\ a_{10} & a_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (4)$$

To apply the affine transformation, simply implement the above representation into code shown below.

Listing 6. Affine function

```
def affine(self, A):
    # You can delete this, this just makes all black image.

    a_mat = [[A[0][0], A[0][1]],
             [A[1][0], A[1][1]]]
    A_mat = np.array(a_mat)

    t = [A[0][2], A[1][2]]

    def affine(x, y):
        a = np.array([[A_mat[0][0], A_mat[0][1], t[0]],
                      [A_mat[1][0], A_mat[1][1], t[1]],
                      [0, 0, 1]])
        img_new = a @ np.array([[x], [y], [1]])
        return (img_new[0]/img_new[2], img_new[1]/img_new[2])

    def affine_inv(x, y):
        B = np.linalg.inv(A_mat)
        a_inv = np.array([[B[0][0], B[0][1], -t[0]],
                          [B[1][0], B[1][1], -t[1]],
                          [0, 0, 1]])

        img_new = a_inv @ np.array([[x], [y], [1]])
        return (img_new[0]/img_new[2], img_new[1]/img_new[2])

    img_new = self.inverse_warp_b(affine_inv, self.shape)


    # Only return new image as uint8
    return img_new.astype(np.uint8)
```

As we can see, the above code takes in parameter A, and then I have devided into two matrix, because the A is list of the parameters because images wont be saved if they are not in array. As asked, the inverse_warp_b helper function is been use to get final image, shown below.

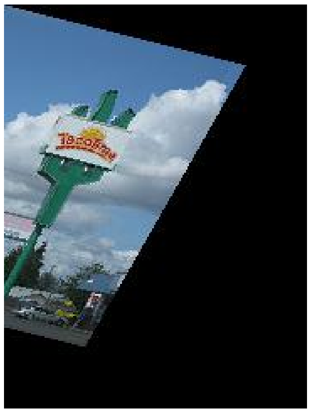
Figure 7. Affine image

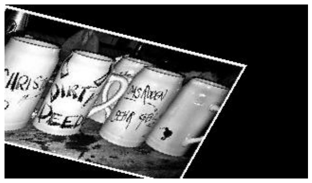
Figure 8. Affine image

2.5. Projective Transformation

Projective transforamation only preserves the straight lines. It means that when the view of the observer chnages, perceived object changes. It creates perspective distortion. The projective transformation consist vector of 8 parameters shown in the mathematical representation below.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (5)$$

Same as Affine transformation, simply implement the mathematical representation to apply the projective transformation.

Listing 7. Projective function

```
def projective(self, H):

    H_mat = np.array(H)

    def proj(x, y):
        img_new = H_mat @ np.array([[x], [y], [1]])
        return (img_new[0]/img_new[2], img_new[1]/img_new[2])

    def proj_inv(x, y):
        p_inv = np.linalg.inv(H_mat)
        img_new = p_inv @ np.array([[x], [y], [1]])
        return (img_new[0]/img_new[2], img_new[1]/img_new[2])

    img_new = self.inverse_warp_b(proj_inv, self.shape)

    # Only return new image as uint8
    return img_new.astype(np.uint8)
```

Above function takes in list as the input parameter and the image will not be saved if it is not in array format, I converted the list H to H_mat, then implemented the mathematical representation. Finally, the output image came, by using inverse_warp_b helper function describe earlier, as shown in the figure below.

2.6. Contrast and Brightness

One of the main transformation of the images is contrast and brightness. This type of transformations help to see image clear or make it darker

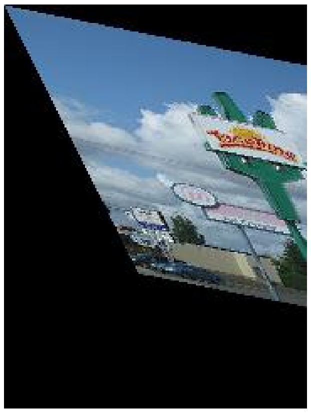
Figure 9. Projective image

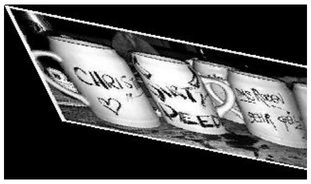
Figure 10. Projective image

that no one can see. We can change the image brightness and conrtast by changing the value of pixel. In this function, I was asked to apply the contrast and brightness at same time. So, I used the code and formula from the lecture notes of Dr. Sarkar to create the helper functions for brightness and contrast.

Here are the helper function

Listing 8. Contrast function

```
def change_constrast(self, image, a):
    img_new = a*image
    img_new = np.where(img_new > 1, 1, img_new)
    img_new = np.where(img_new<0, 0, img_new)
    return img_new
```

Listing 9. Brightness function

```
def change_bright(self, image, b):
    img_new = image + b
    img_new = np.where(img_new>1, 1, img_new)
    img_new = np.where(img_new<0, 0, img_new)
    return img_new
```

The above functions, only works when input a and input b are constant or they are an array with spatially variant values over the image locations. These function will do the element wise multiplication and summation then clip the intensity to max 1 and max 0 and return the image.

Now we have been asked to modify only the L-channel of the image, where L is lightness. To modify the L- channel, we have to convert the image to Lab format form RGB as shown the code below.

Listing 10. Contrast-Brightness function

```
def brightness_contrast(self, a, b):
    img_g = self.img
    img_new = rgb2lab(img_g)
    img_new[:,:,0] = self.change_constrast(
                    self.change_bright(img_new[:,:,0]/100, a), b) *100.0
    img_new = lab2rgb(img_new)*255

    # Only return new image as uint8
    return img_new.astype(np.uint8)
```

Then we apply the helper functions to change the L-channel, then as require we have to save the image in RGB format, so converted the image to RGB and return the image. See the difference in the below image from other images.

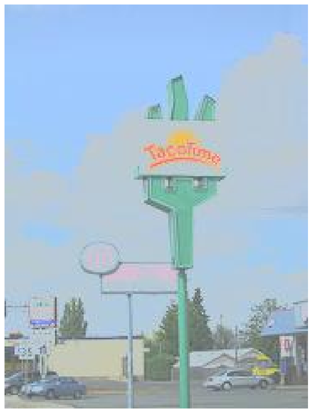
Figure 11. Contrast-Brightness image

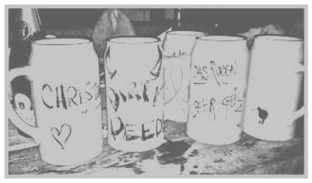
Figure 12. Contrast-Brightness image

2.7. Gamma correction of the L-channel

Gamma corrections is used to remove the non-linear mapping between input radiance and quantized pixel value. It controls the overall brightness of the image. Sometimes images looks too dark or bleached out if gamma corrections are not set accurately, like in games. The mathematical representation shown below.

$$g(i,j) = [f(i,j)]^{\frac{1}{\gamma}} \tag{6}$$

The mathematical representation is a simply the power of gamma value fraction of the image array. After implementing the function it will look like below.

Listing 11. Gamma function

```
def gama_correct(self, image, g):
    return (np.power(image, 1/g))
```

Then just call this function into gamma_correction function after changing the image to Lab as shown below and return the final image in RGB format as shown in the next figure.

Listing 12. Contrast-Brightness of the image

```
def gamma_correction(self, a, b):
    img_g = self.img
    img_new = rgb2lab(img_g)
    img_new[:, :, 0] = self.gama_correct(img_new[:,:,0]/100, a) * 100.0
    img_new = lab2rgb(img_new)*255
    # Only return new image as uint8
    return img_new.astype(np.uint8)
```

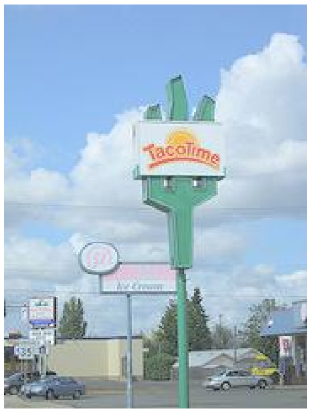

Figure 13. Gamma Correction image

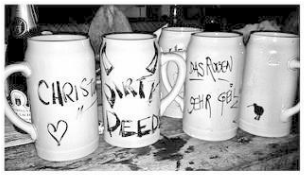

Figure 14. Gamma Correction image

2.8. Histogram equalization of the L-channel

The goal of histogram equalization is to get the images in more contrast form. We can do that by changing the pixel value using a function so the pixel value can be uniform. Form the cumulative histogram, which is sum of the probability distribution. Since we need required only for L-channel it can be changed as shown in the code below.

Listing 13. Histogram equalization of L-channel function

```
img_g = self.img
    img_new = rgb2lab(img_g)
    histogram, bin_edge = np.histogram(img_new[:,:,0], bins=256)
    histogram = histogram/np.sum(histogram)
    cummul_histo = np.cumsum(histogram)

    img_new[:,:,0] = cummul_histo[img_g[:,:,0]]*100
    img_new = lab2rgb(img_new)*255

    # Only return new image as uint8
    return img_new.astype(np.uint8)
```

Same as before, we have to change the image to Lab and then we have to calculate the cummulative histogram value of apixel then apply that cummulative value on L-channel and cahnge the image in RGB format then return image, shown below.

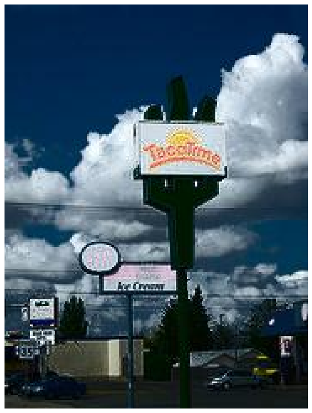

Figure 15. Histogram equalization image

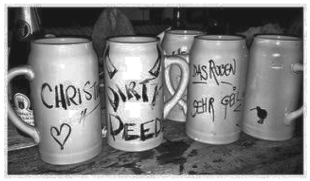

Figure 16. Histogram equalization image

2.9. Mean image and the Standard Deviation image

Mean of the image means calculate the mean of all pixels of the images then return the image. Similarly calculate the Standard deviation of all pixels of the images then return the image. see the images below which will help more to understand.

As we can see in the images, it is combination of different images. Basically we combine the images by calculating the mean and standard deviation of

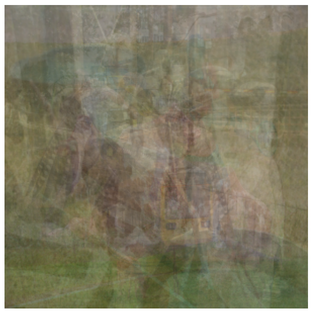

Figure 17. Mean of the images

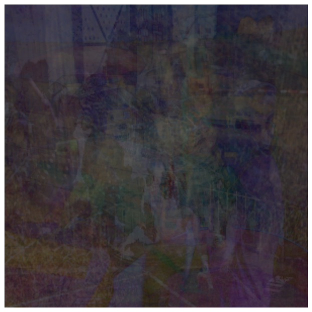

Figure 18. Standard deviation of the images

the pixel values then we put them in one list of arrays and return those arrays in image form. As shown in the code below.

Listing 14. Mean and Standard Deviation function
```
def mean_sd(self, resize_shape):
    # You can delete this, just copies images from batch
    batch = self.batch
    new_imageList = []   #create new list of images
    for im in batch:   # go through each images and
                        #resize them since all images
                        #are different sizes
        resize_image = resize(im, resize_shape)
        new_imageList.append(resize_image)
#add the images to the new list

    img_g = np.array(new_imageList)
# save as numpy array since they wont save as image
    #now get the mean and std of those images
    _mean = np.mean(img_g, axis=(0,))
# uses axis= to make sure images are in 2D
    _sd = np.std(img_g, axis=(0, ))

    img_mean = _mean*255
    img_sd = _sd*255

    # Return only mean and sd images as uint8
    return (
        img_mean.astype(np.uint8),
        img_sd.astype(np.uint8)
    )
```

2.10. Batch normalize an image

To perform normalization operation, we have to devide the pixel values of the main image by the maximum possible value, which is here 255, so we can get the pixel value between 0 and 1 and call it normalize_img. Then we have to calculate the mean and standard deviation of the pixels of normalize_img. shown the code below.

Listing 15. Batch normalization function
```
def batch_norm(self, resize_shape):
    # You can delete this, just copies images from batch
    batch = self.batch
    batch_new = []
    for img in tqdm(batch):
        # scale the pixel value between 0 ad 1
        img = img.astype(np.float)
        normalize_img = img/np.max(img)
        _mean = np.mean(normalize_img, axis=(0,1))
        _sd = np.std(normalize_img, axis=(0,1))

        new_image = (normalize_img - _mean[None, None, :])/_sd[None, None, :]
        # rescale value
        image_clipp = (new_image + 3)/6
        image_clipp = np.where(image_clipp >1, 1, image_clipp)
        image_clipp = np.where(image_clipp <0, 0, image_clipp)

        new_image = image_clipp*255

        batch_new.append(new_image)

    # Return batch normalized images as uint8
    batch_new = [img.astype(np.uint8) for img in batch_new]
    return batch_new
```

The subtract the mean from each pixel of and divide by the standard deviation. Now the image we got so far has negative value so added 3 then divide by 6 which will rescale the pixel value between 0 and 1. Now clipp the value beyond 0 and 1. Then return the image.

Here, we are dealing with whole batch of 10 images, so went through each images in the batch by using for loop, save the changed images into batch list and then return whole list.

3. References

Lecture Notes, Computer Vision, Sudeep Sarkar, University of South Florida, Tampa