Assignment 2

Yifan Luo, u7351505

Task 1: Harris Corner Detection

1. Complete the missing sections in "*harris.py*".

The full implementation can be found in the uploaded file. The screenshots in the next task also provide the implementation of ***harris.py***.

2. Add a comment on line #53 and solutions after line #60.

The following 7 screenshots include the full implementation of ***harris.py*** from top to bottom:

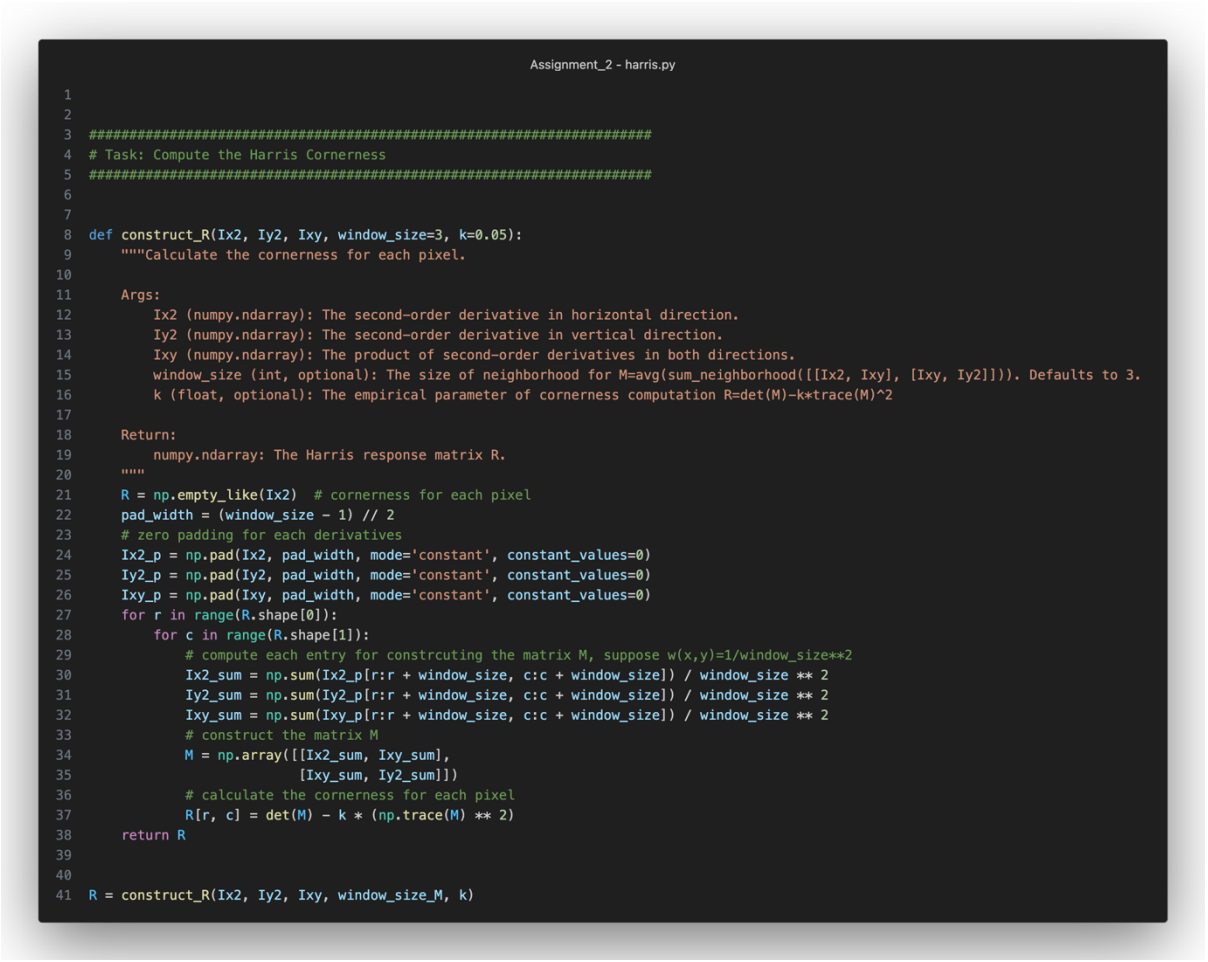
Assignment_2 - harris.py

```python
"""
CLAB Task-1: Harris Corner Detector
Yifan Luo (u7351505)
"""

import numpy as np
import matplotlib.pyplot as plt
import cv2
from scipy.linalg import det, inv


def conv2(img, conv_filter):
    """Convolution operation.

    Args:
        img (numpy.ndarray): The input image.
        conv_filter (numpy.ndarray): The input convolution filter.

    Returns:
        numpy.ndarray: The convolution operation result.
    """
    # flip the filter
    f_size_1, f_size_2 = conv_filter.shape
    conv_filter = conv_filter[range(f_size_1 - 1, -1, -1), :][:, range(f_size_2 - 1, -1, -1)]
    pad = (conv_filter.shape[0] - 1) // 2
    result = np.zeros((img.shape))
    img = np.pad(img, ((pad, pad), (pad, pad)), 'constant', constant_values=(0, 0))
    filter_size = conv_filter.shape[0]
    for r in np.arange(img.shape[0] - filter_size + 1):
        for c in np.arange(img.shape[1] - filter_size + 1):
            curr_region = img[r:r + filter_size, c:c + filter_size]
            curr_result = curr_region * conv_filter
            conv_sum = np.sum(curr_result)  # Summing the result of multiplication.
            result[r, c] = conv_sum  # Saving the summation in the convolution layer feature map.

    return result
```

Assignment_2 - harris.py

```
1
2
3  def fspecial(shape=(3, 3), sigma=0.5):
4      """Generate a Gaussian filter given its shape and standard deviation.
5
6      Args:
7          shape (tuple, optional): The shape of the Gaussian filter. Defaults to (3, 3).
8          sigma (float, optional): The standard deviation of the Gaussian distribution. Defaults to 0.5.
9
10     Returns:
11         h (numpy.ndarray): The generated Gaussian kernel.
12     """
13     m, n = [(ss - 1.) / 2. for ss in shape]
14     y, x = np.ogrid[-m:m + 1, -n:n + 1]  # coordinates of the filter
15     h = np.exp(-(x * x + y * y) / (2. * sigma * sigma))  # Gaussian kernel 13 * 13
16     h[h < np.finfo(h.dtype).eps * h.max()] = 0
17     # normalise the sum of h to 1
18     sumh = h.sum()
19     if sumh != 0:
20         h /= sumh
21     return h
22
23
24 # Parameters, add more if needed
25 sigma = 2  # parameter for the Gaussian filter in fspecial
26 img_path = 'Task1/Harris-2.jpg'
27 window_size_M = 3  # window size for the calculation of M
28 window_size_R = 3  # neighbourhood size for local maxima of R
29 k = 0.05         # empirically determined constant ranged from 0.04 to 0,06
30 thresh = 1e-8    # threshold in thresholding process on R
31
32 # Derivative masks
33 dx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])  # vertical Sobel filter
34 dy = dx.transpose()                                   # horizontal Sobel filter
35
36 # Load pictures
37 bw = plt.imread(img_path)   # (row, col, rgb) for Harris-[1345].jpg; (row, col) for Harris-2.jpg
38 bw = np.array(bw * 255, dtype='uint8')
39 if bw.ndim == 3:    # convert RGB images (Harris-[1345].jpg) to grayscale images
40     bw = cv2.cvtColor(bw, cv2.COLOR_RGB2GRAY)
41
42 # Computer x and y derivatives of image
43 Ix = conv2(bw, dx)  # horizontal derivative
44 Iy = conv2(bw, dy)  # vertical derivative
45
46 # Generate a Gaussian filter for smoothing the second-order derivatives
47 g = fspecial(shape=(max(1, np.floor(3 * sigma) * 2 + 1), max(1, np.floor(3 * sigma) * 2 + 1)), sigma=sigma)
48 Iy2 = conv2(np.power(Iy, 2), g)  # smoothed second-order vertical derivative (Iy)
49 Ix2 = conv2(np.power(Ix, 2), g)  # smoothed second-order horizontal derivative (Ix)
50 Ixy = conv2(Ix * Iy, g)          # smoothed production of Iy and Ix
```

The screenshot above shows the relevant comments of *g=fspecial()* at line #46.



The screenshot above shows the implementation of the calculation of Harris Cornerness with documents and necessary comments.

Assignment_2 - harris.py

```python

#####################################################################
# Task: Perform non-maximum suppression and
#       thresholding, return the N corner points
#       as an Nx2 matrix of x and y coordinates
#####################################################################


def thresh_non_max_sup(R, thresh, window_size):
    """Conduct the thresholding and non-maximum suppression on the input cornerness R.

    Args:
        R (numpy.ndarray): The input cornerness for each pixel.
        thresh (float): The threshold which only keeps R>thresh
        window_size (int): The neighborhood size for the local maxima.

    Returns:
        (numpy.ndarray, numpy.ndarray): Return the index after thresholding and non-max suppression.
    """
    # thresholding: select index where the cornerness is larger than threshold
    thresh_idx_x, thresh_idx_y = np.where(R > thresh)
    thresh_idx = np.array([(x, y) for (x, y) in zip(thresh_idx_x, thresh_idx_y)])
    # non-max suppression: select index where is the local maxima cornerness
    non_max_idx = []
    R_p = np.pad(R, pad_width=(window_size - 1) // 2, mode='reflect')
    for r, c in thresh_idx:
        # for each pixel, find its local maxima of cornerness
        local_max = R_p[r:r + window_size, c:c + window_size].max()
        # print(R[r, c], R_p[r:r + window_size, c:c + window_size])
        # only keep index of local maxima
        if R[r, c] == local_max:
            non_max_idx.append([r, c])
    return thresh_idx, np.asarray(non_max_idx)


# R results after thresholding and non-max suppression
corner_thresh, corner_thresh_non_max = thresh_non_max_sup(R, thresh*R.max(), window_size_R)
# R results using the built-in function cv2.cornerHarris
R_cv = cv2.cornerHarris(bw, blockSize=window_size_M, ksize=3, k=k)
corner_thresh_cv, corner_thresh_non_max_cv = thresh_non_max_sup(R_cv, thresh*R_cv.max(), window_size_R)
```

The non-maximum suppression and thresholding implementation for Harris Corner detection.

Assignment_2 - harris.py

```
1
2
3  def plot_R(img_path, R, corner_thresh, corner_thresh_non_max):
4      # results comparison between:
5      img = plt.imread(img_path)
6      fig, ax = plt.subplots(1, 4, figsize=(8, 4))
7      fig.suptitle(img_path.split('/')[1] + '\n' +
8                   "threshold:" + str(thresh) + " k:" + str(k))
9      # 1. original image
10     ax[0].imshow(img, cmap='gray' if np.ndim(img) == 2 else 'viridis')
11     ax[0].axis('off')
12     ax[0].set_title("Original")
13     # 2. cornerness R
14     ax[1].imshow(R, cmap='gray')
15     ax[1].axis('off')
16     ax[1].set_title("Cornerness")
17     # 3. corners after threholding with non-max suppression
18     ax[2].imshow(img, cmap='gray' if np.ndim(img) == 2 else 'viridis')
19     ax[2].scatter(corner_thresh[:, 1], corner_thresh[:, 0], color='green', s=0.8, marker='o')
20     ax[2].axis('off')
21     ax[2].set_title("Thresh")
22     # 4. corners after threholding with non-max suppression
23     ax[3].imshow(img, cmap='gray' if np.ndim(img) == 2 else 'viridis')
24     ax[3].scatter(corner_thresh_non_max[:, 1], corner_thresh_non_max[:, 0], color='green', s=0.8, marker='o')
25     ax[3].axis('off')
26     ax[3].set_title("Thresh + Non-Max")
27     plt.show()
28
29
30 plot_R(img_path, R, corner_thresh, corner_thresh_non_max)
31 plot_R(img_path, R_cv, corner_thresh_cv, corner_thresh_non_max_cv)
```

The function used to plot all images and corners.

Assignment_2 - harris.py

```python

def trans_mat_out_shape(img, theta):
    """Return the transformation matrix and the corresponding output shape.

    Args:
        img (numpy.ndarray):
        theta (float): The rotation degree. The value should in the range of [0, 360].

    Returns:
        (np.ndarray, (int, int)): The generated transformation matrix, and the shape of output image (n_row, n_col).
    """
    radians = theta * np.pi / 180
    trans_mat = np.array([[np.cos(radians), -np.sin(radians)],
                          [np.sin(radians), np.cos(radians)]])
    corner_top_left = np.array([0, 0]).T
    corner_top_right = np.array([0, img.shape[1] - 1]).T
    corner_bot_left = np.array([img.shape[0] - 1, 0]).T
    corner_bot_right = np.array([img.shape[0] - 1, img.shape[1] - 1]).T

    if 0 <= theta < 90 or 180 <= theta < 270:
        n_r = np.abs((trans_mat @ corner_top_right)[0] - (trans_mat @ corner_bot_left)[0])
        n_c = np.abs((trans_mat @ corner_bot_right)[1] - (trans_mat @ corner_top_left)[1])
    if 90 <= theta < 180 or 270 <= theta <= 360:
        n_r = np.abs((trans_mat @ corner_bot_right)[0] - (trans_mat @ corner_top_left)[0])
        n_c = np.abs((trans_mat @ corner_bot_left)[1] - (trans_mat @ corner_top_right)[1])
    output_shape = (int(np.ceil(n_r)), int(np.ceil(n_c))) if img.ndim == 2 else (int(np.ceil(n_r)), int(np.ceil(n_c)), 3)
    return trans_mat, output_shape


def inv_warp(img, inv_trans_mat, out_size):
    """Inverse image warpping.

    Args:
        img (numpy.ndarray): The image to be rotated.
        inv_trans_mat (numpy.ndarray): The inverse of transformation matrix used to rotate the image.
        out_size (Tuple(int, int)): The size of output rotated image.

    Returns:
        numpy.ndarray: The rotated image.
    """
    n_row, n_col = out_size[0], out_size[1]
    cent_x, cent_y = img.shape[0] / 2, img.shape[1] / 2
    rot_u, rot_v = inv(inv_trans_mat) @ np.array([cent_x, cent_y]).T
    cent_u, cent_v = n_row / 2, n_col / 2
    shift_u, shift_v = rot_u - cent_u, rot_v - cent_v
    res = np.zeros(out_size)
    cond = lambda x, y: (0 <= x < img.shape[0]) and (0 <= y < img.shape[1])
    for u in range(n_row):
        for v in range(n_col):
            x, y = inv_trans_mat @ np.array([u + shift_u, v + shift_v]).T
            if cond(x, y):
                if int(x) == x and int(y) == y:
                    res[u, v] = img[int(x), int(y)]
                else:
                    # bilinear interpolation
                    y1, y2 = int(np.floor(y)), int(np.ceil(y))
                    x1, x2 = int(np.floor(x)), int(np.ceil(x))
                    if cond(x1, y2) and cond(x2, y2) and cond(x1, y1) and cond(x2, y1):
                        if x1 == x2:
                            res[u, v] = img[x1, y2] * (y - y1) / (y2 - y1) + img[x1, x1] * (y2 - y) / (y2 - y1)
                        elif y1 == y2:
                            res[u, v] = img[x2, y2] * (x2 - x) / (x2 - x1) + img[x1, y2] * (x - x1) / (x2 - x1)
                        else:
                            R1 = img[x1, y2] * ((x2 - x) / (x2 - x1)) + img[x2, y2] * ((x - x1) / (x2 - x1))
                            R2 = img[x1, y1] * ((x2 - x) / (x2 - x1)) + img[x2, y1] * ((x - x1) / (x2 - x1))
                            res[u, v] = R1 * (y - y1) / (y2 - y1) + R2 * (y2 - y) / (y2 - y1)

    return res.astype('uint8')

```

The first function is used to calculate the transformation matrix and its corresponding output shape; The second function is the implementation of the inverse wrapping with bilinear interpolation.

Assignment_2 - harris.py

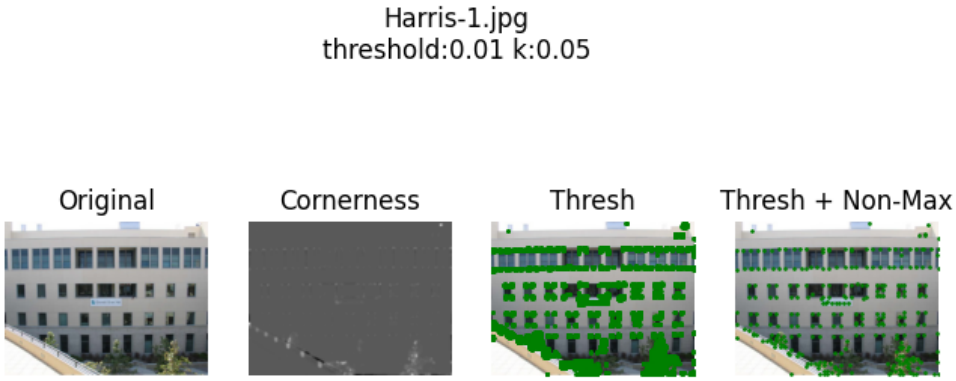
```
1
2
3  img = plt.imread(img_path)
4  fig, ax = plt.subplots(2, 4)
5  retate_imgs = []
6  for i, theta in enumerate([0, 90, 180, 270]):
7      trans_mat, output_shape = trans_mat_out_shape(img, theta=theta)
8      rotate_img = inv_warp(img, inv(trans_mat), output_shape)
9      retate_imgs.append(rotate_img)
10     ax[0, i].imshow(rotate_img, cmap='gray' if rotate_img.ndim == 2 else 'viridis')
11     ax[0, i].axis('off')
12     ax[0, i].set_title(str(theta) + " degrees")
13     bw = cv2.cvtColor(rotate_img, cv2.COLOR_RGB2GRAY) if rotate_img.ndim == 3 else rotate_img
14     Ix = conv2(bw, dx)  # horizontal derivative
15     Iy = conv2(bw, dy)  # vertical derivative
16     Iy2 = conv2(np.power(Iy, 2), g)  # smoothed second-order vertical derivative (Iy)
17     Ix2 = conv2(np.power(Ix, 2), g)  # smoothed second-order horizontal derivative (Ix)
18     Ixy = conv2(Ix * Iy, g)          # smoothed production of Iy and Ix
19     R = construct_R(Ix2, Iy2, Ixy, window_size_M, k)
20     corner_thresh, corner_thresh_non_max = thresh_non_max_sup(R, thresh*R.max(), window_size_R)
21     ax[1, i].imshow(rotate_img, cmap='gray' if np.ndim(rotate_img) == 2 else 'viridis')
22     ax[1, i].scatter(corner_thresh_non_max[:, 1], corner_thresh_non_max[:, 0], color='green', s=0.8, marker='o')
23     ax[1, i].axis('off')
24 fig.suptitle(img_path.split('/')[1])
25 plt.show()
```

The process of Harris corner detection.

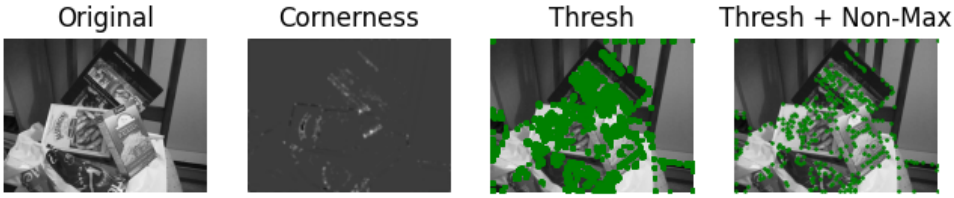
3. Tests on the first four images

The following 4 pictures show the results of:

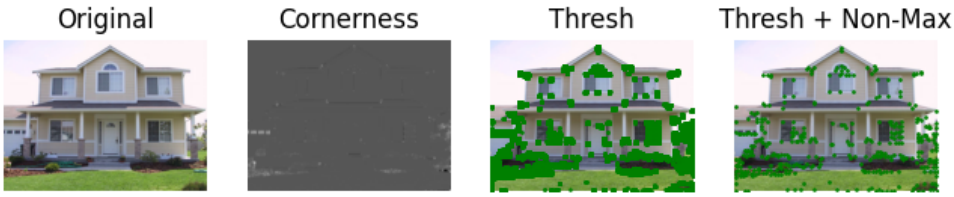
- original image
- cornerness
- original corners with thresholding
- corners with thresholding + non-maximum suppression



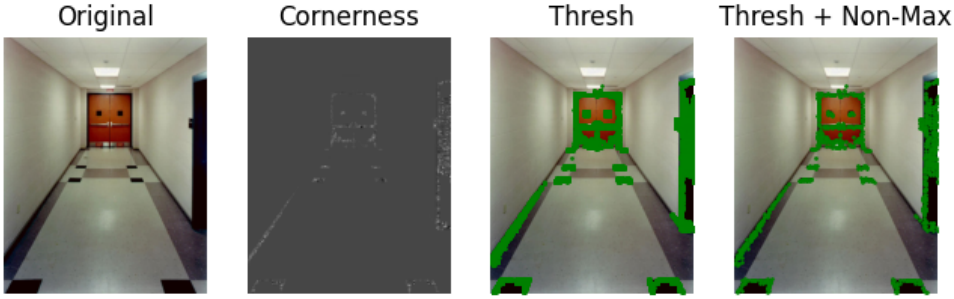
Harris-2.jpg
threshold:0.01 k:0.05



Harris-3.jpg
threshold:0.01 k:0.05



Harris-4.jpg
threshold:0.01 k:0.05

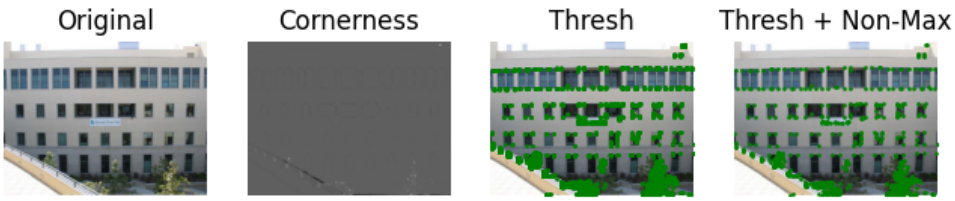


4. The built-in function results from cv2 library

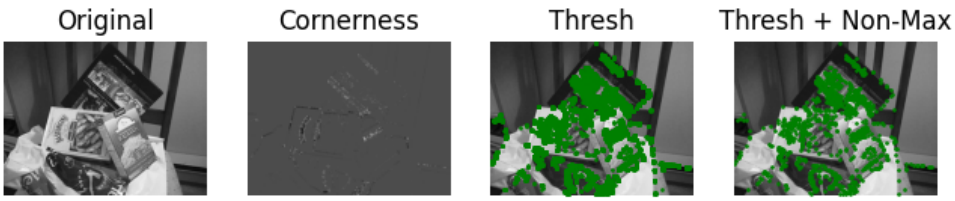
The following pictures show the results from "***cv2.corerHarris***" for calculating the cornerness while using the same threshold and k parameter as task 1.3.

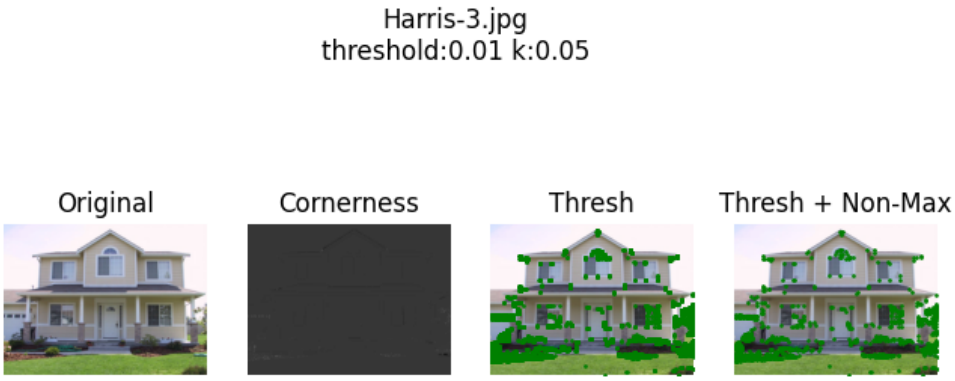
In general, the cornerness from the built-in function is larger than the self-implemented function. The corners in the pictures also suggest more corners under the same threshold and k value. By tuning the threshold and k value, both functions can generate similar results.

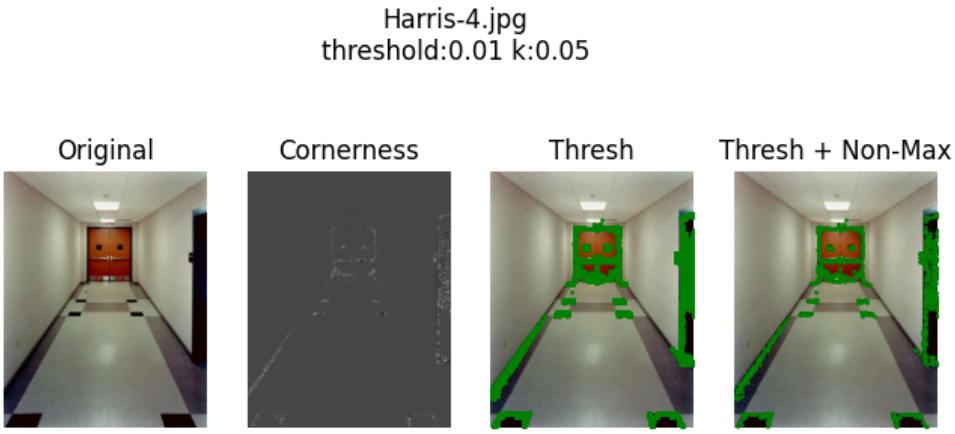
Harris-1.jpg
threshold:0.01 k:0.05



Harris-2.jpg
threshold:0.01 k:0.05







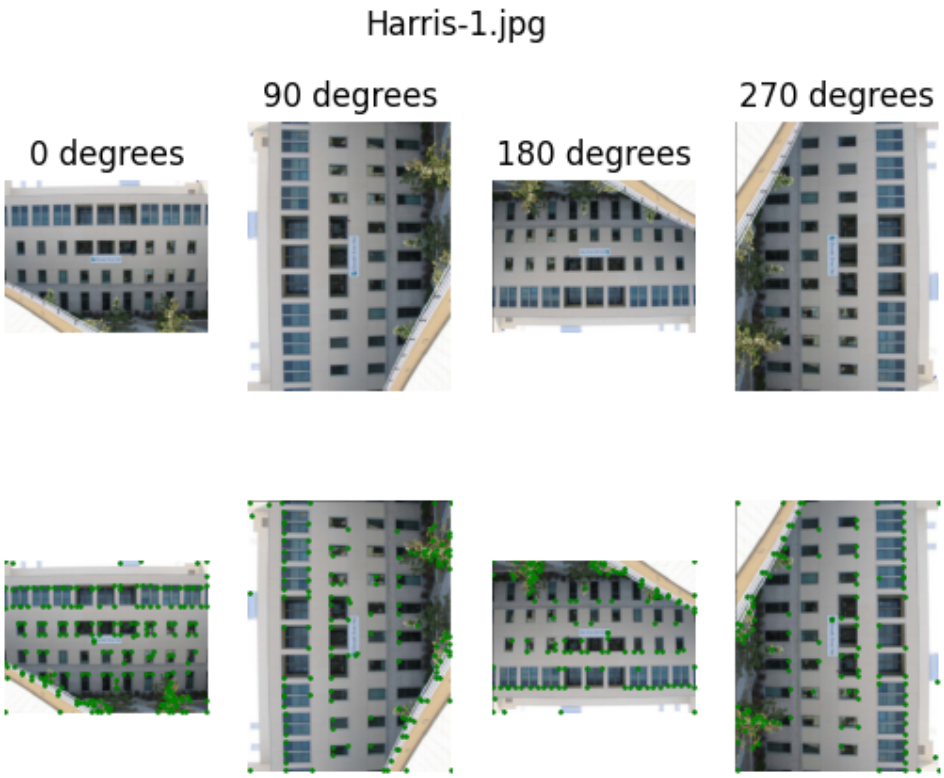
5. Inverse warping function

The implementation of inverse warping is shown in the task 1.2.

The function ***inv_warp*** takes an image, a transformation matrix and the output shape, and returns the inverse warping result. The function also implements the **bilinear interpolation** techniques to fill the potential "black holes" in the reversed image.

6. Rotation and corner detection of Harris-1.jpg

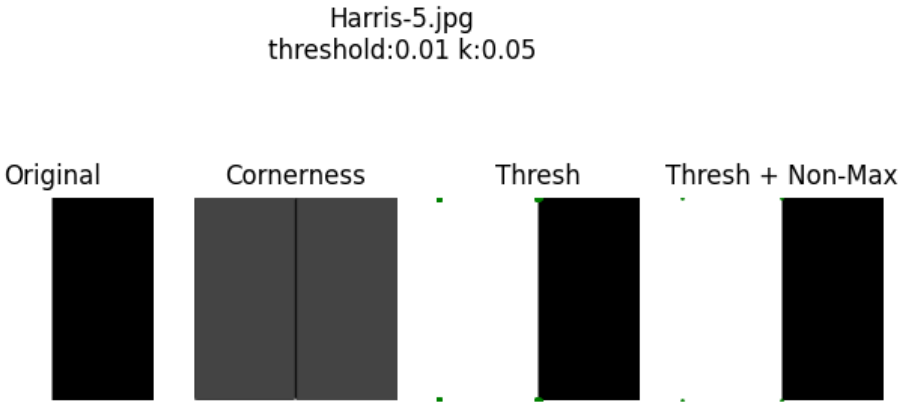
All the rotated images are processed to the Harris corner detection functions with the same threshold and k value. The rotation results and the corresponding corners of Harris-1.jpg are shown below, the hyperparameters are the same as task 1.3:



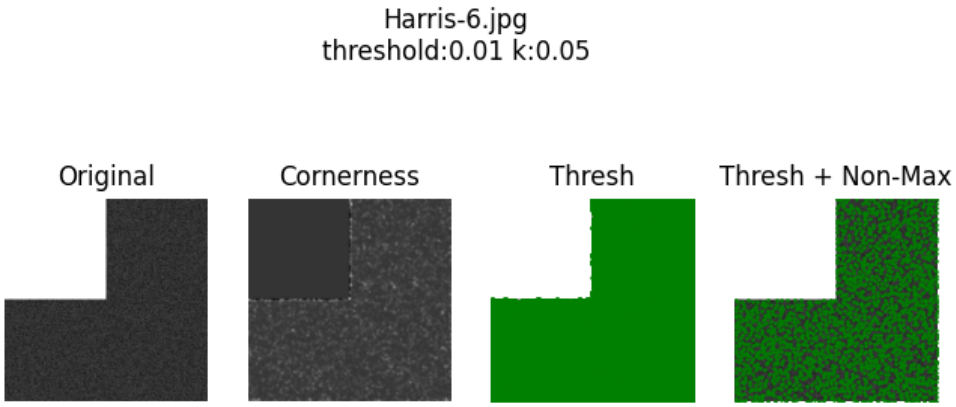
Theoretically, Harris corner detection is inherently invariant to rotation, since after rotations, the image may have different spatial relationships which can affect the detection of corners. Especially, the cornerness is based on local intensity variations in the image. Rotation can alter the intensity gradients of pixels, potentially affecting the response values computed by the detector.

In our results, Harris-1.jpg shows no visual difference after being rotated. The reason may be the gradients in the image are more robust under rotation. All the windows and buildings are vertically or horizontally aligned. Due to the same reason, the derivates and corners are less sensitive to rotation since after being rotated, most of the lines are still vertical or horizontal in the rotated images.

7. Analysis of Harris-5.jpg and Harris-6.jpg



During the pre-processing of the image, we use zero-padding in the convolution operation, and the derivatives around the right side of the black bar are all zeros. The black bar does not show any corners on its top right and bottom right areas regardless of any rotation.



The Harris-6.jpg has salt-and-pepper noise on its black region, i.e., randomly occurring black and white pixels in the black region. We can see after thresholding and non-maximum suppression, that there are still a lot of false corners due to the noise.

The full source code:

Assignment_2 - harris.py

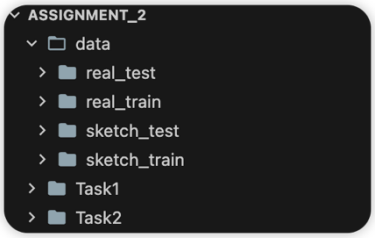
```python
"""
CLAB Task-1: Harris Corner Detector
Yifan Luo (u7351505)
"""

import numpy as np
import matplotlib.pyplot as plt
import cv2
from scipy.linalg import det, inv


def conv2(img, conv_filter):
    """Convolution operation.

    Args:
        img (numpy.ndarray): The input image.
        conv_filter (numpy.ndarray): The input convolution filter.

    Returns:
        numpy.ndarray: The convolution operation result.
    """
    # flip the filter
    f_size_1, f_size_2 = conv_filter.shape
    conv_filter = conv_filter[range(f_size_1 - 1, -1, -1), :][:, range(f_size_2 - 1, -1, -1)]
    pad = (conv_filter.shape[0] - 1) // 2
    result = np.zeros((img.shape))
    img = np.pad(img, ((pad, pad), (pad, pad)), 'constant', constant_values=(0, 0))
    filter_size = conv_filter.shape[0]
    for r in np.arange(img.shape[0] - filter_size + 1):
        for c in np.arange(img.shape[1] - filter_size + 1):
            curr_region = img[r:r + filter_size, c:c + filter_size]
            curr_result = curr_region * conv_filter
            conv_sum = np.sum(curr_result)  # Summing the result of multiplication.
            result[r, c] = conv_sum  # Saving the summation in the convolution layer feature map.

    return result


def fspecial(shape=(3, 3), sigma=0.5):
    """Generate a Gaussian filter given its shape and standard deviation.

    Args:
        shape (tuple, optional): The shape of the Gaussian filter. Defaults to (3, 3).
        sigma (float, optional): The standard deviation of the Gaussian distribution. Defaults to 0.5.

    Returns:
        h (numpy.ndarray): The generated Gaussian kernel.
    """
    m, n = [(ss - 1.) / 2. for ss in shape]
    y, x = np.ogrid[-m:m + 1, -n:n + 1]  # coordinates of the filter
    h = np.exp(-(x * x + y * y) / (2. * sigma * sigma))  # Gaussian kernel 13 * 13
    h[h < np.finfo(h.dtype).eps * h.max()] = 0
    # normalise the sum of h to 1
    sumh = h.sum()
    if sumh != 0:
        h /= sumh
    return h


# Parameters, add more if needed
sigma = 2  # parameter for the Gaussian filter in fspecial
# img_path = '/Users/yifan/Documents/ANU/2451/COMP6528_Computer_Vision/wk5/Assignment_2/Task1/Harris-1.jpg'
# img_path = '/Users/yifan/Documents/ANU/2451/COMP6528_Computer_Vision/wk5/Assignment_2/Task1/Harris-2.jpg'
# img_path = '/Users/yifan/Documents/ANU/2451/COMP6528_Computer_Vision/wk5/Assignment_2/Task1/Harris-3.jpg'
# img_path = '/Users/yifan/Documents/ANU/2451/COMP6528_Computer_Vision/wk5/Assignment_2/Task1/Harris-4.jpg'
# img_path = '/Users/yifan/Documents/ANU/2451/COMP6528_Computer_Vision/wk5/Assignment_2/Task1/Harris-5.jpg'
img_path = '/Users/yifan/Documents/ANU/2451/COMP6528_Computer_Vision/wk5/Assignment_2/Task1/Harris-6.jpg'
window_size_M = 3  # window size for the calculation of M
window_size_R = 3  # neighbourhood size for local maxima of R
k = 0.05          # empirically determined constant ranged from 0.04 to 0.06
thresh = 1e-2     # threshold in thresholding process on R

# Derivative masks
dx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])  # vertical Sobel filter
dy = dx.transpose()                                 # horizontal Sobel filter

# Load pictures
bw = plt.imread(img_path)   # (row, col, rgb) for Harris-[1345].jpg; (row, col) for Harris-2.jpg
bw = np.array(bw * 255, dtype='uint8')
if bw.ndim == 3:    # convert RGB images (Harris-[1345].jpg) to grayscale images
    bw = cv2.cvtColor(bw, cv2.COLOR_RGB2GRAY)

# Computer x and y derivatives of image
Ix = conv2(bw, dx)  # horizontal derivative
Iy = conv2(bw, dy)  # vertical derivative

# Generate a Gaussian filter for smoothing the second-order derivatives
g = fspecial(shape=(max(1, np.floor(3 * sigma) * 2 + 1), max(1, np.floor(3 * sigma) * 2 + 1)), sigma=sigma)
Iy2 = conv2(np.power(Iy, 2), g)  # smoothed second-order vertical derivative (Iy)
Ix2 = conv2(np.power(Ix, 2), g)  # smoothed second-order horizontal derivative (Ix)
Ixy = conv2(Ix * Iy, g)          # smoothed production of Iy and Ix


#####################################################################
# Task: Compute the Harris Cornerness
#####################################################################


def construct_R(Ix2, Iy2, Ixy, window_size=3, k=0.05):
    """Calculate the cornerness for each pixel.

    Args:
        Ix2 (numpy.ndarray): The second-order derivative in horizontal direction.
        Iy2 (numpy.ndarray): The second-order derivative in vertical direction.
        Ixy (numpy.ndarray): The product of second-order derivatives in both directions.
        window_size (int, optional): The size of neighborhood for M=avg(sum_neighborhood([[Ix2, Ixy], [Ixy, Iy2]])). Defaults to 3.
        k (float, optional): The empirical parameter of cornerness computation R=det(M)-k*trace(M)^2

    Return:
        numpy.ndarray: The Harris response matrix R.
    """
    R = np.empty_like(Ix2)  # cornerness for each pixel
    pad_width = (window_size - 1) // 2
    # zero padding for each derivatives
    Ix2_p = np.pad(Ix2, pad_width, mode='constant', constant_values=0)
    Iy2_p = np.pad(Iy2, pad_width, mode='constant', constant_values=0)
    Ixy_p = np.pad(Ixy, pad_width, mode='constant', constant_values=0)
    for r in range(R.shape[0]):
        for c in range(R.shape[1]):
            # compute each entry for constructing the matrix M, suppose w(x,y)=1/window_size**2
            Ix2_sum = np.sum(Ix2_p[r:r + window_size, c:c + window_size]) / window_size ** 2
            Iy2_sum = np.sum(Iy2_p[r:r + window_size, c:c + window_size]) / window_size ** 2
            Ixy_sum = np.sum(Ixy_p[r:r + window_size, c:c + window_size]) / window_size ** 2
            # construct the matrix M
            M = np.array([[Ix2_sum, Ixy_sum],
                          [Ixy_sum, Iy2_sum]])
            # calculate the cornerness for each pixel
            R[r, c] = det(M) - k * (np.trace(M) ** 2)
    return R


R = construct_R(Ix2, Iy2, Ixy, window_size_M, k)


#####################################################################
# Task: Perform non-maximum suppression and
#       thresholding, return the N corner points
#       as an Nx2 matrix of x and y coordinates
#####################################################################


def thresh_non_max_sup(R, thresh, window_size):
    """Conduct the thresholding and non-maximum suppression on the input cornerness R.

    Args:
        R (numpy.ndarray): The input cornerness for each pixel.
        thresh (float): The threshold which only keeps R>thresh
        window_size (int): The neighborhood size for the local maxima.

    Returns:
        (numpy.ndarray, numpy.ndarray): Return the index after thresholding and non-max suppression.
    """
    # thresholding: select index where the cornerness is larger than threshold
    thresh_idx_x, thresh_idx_y = np.where(R > thresh)
    thresh_idx = np.array([(x, y) for (x, y) in zip(thresh_idx_x, thresh_idx_y)])
    # non-max suppression: select index where is the local maxima cornerness
    non_max_idx = []
    R_p = np.pad(R, pad_width=(window_size - 1) // 2, mode='reflect')
    for r, c in thresh_idx:
        # for each pixel, find its local maxima of cornerness
        local_max = R_p[r:r + window_size, c:c + window_size].max()
        # print(R[r, c], R_p[r:r + window_size, c:c + window_size])
        # only keep index of local maxima
        if R[r, c] == local_max:
            non_max_idx.append([r, c])
    return thresh_idx, np.asarray(non_max_idx)


# R results after thresholding and non-max suppression
corner_thresh, corner_thresh_non_max = thresh_non_max_sup(R, thresh*R.max(), window_size_R)
# R results using the built-in function cv2.cornerHarris
# R_cv = cv2.cornerHarris(bw, blockSize=window_size_M, ksize=3, k=k)
# corner_thresh_cv, corner_thresh_non_max_cv = thresh_non_max_sup(R_cv, thresh*R_cv.max(), window_size_R)

# print("self-implemented cornerness:\n", corner_thresh_non_max)
# print("built-in cornerness:\n", corner_thresh_non_max_cv)


def plot_R(img_path, R, corner_thresh, corner_thresh_non_max):
    # results comparison between:
    img = plt.imread(img_path)
    fig, ax = plt.subplots(1, 4, figsize=(8, 4))
    fig.suptitle(img_path.split('/')[-1] + "\n" +
                 "threshold:" + str(thresh) + " k:" + str(k))
    # 1. original image
    ax[0].imshow(img, cmap='gray' if np.ndim(img) == 2 else 'viridis')
    ax[0].axis('off')
    ax[0].set_title("Original")
    # 2. cornerness R
    ax[1].imshow(R, cmap='gray')
    ax[1].axis('off')
    ax[1].set_title("Cornerness")
    # 3. corners after threholding with non-max suppression
    ax[2].imshow(img, cmap='gray' if np.ndim(img) == 2 else 'viridis')
    ax[2].scatter(corner_thresh[:, 1], corner_thresh[:, 0], color='green', s=0.8, marker='o')
    ax[2].axis('off')
    ax[2].set_title("Thresh")
    # 4. corners after threholding with non-max suppression
    ax[3].imshow(img, cmap='gray' if np.ndim(img) == 2 else 'viridis')
    ax[3].scatter(corner_thresh_non_max[:, 1], corner_thresh_non_max[:, 0], color='green', s=0.8, marker='o')
    ax[3].axis('off')
    ax[3].set_title("Thresh + Non-Max")
    plt.show()


plot_R(img_path, R, corner_thresh, corner_thresh_non_max)
# plot_R(img_path, R_cv, corner_thresh_cv, corner_thresh_non_max_cv)


def trans_mat_out_shape(img, theta):
    """Return the transformation matrix and the corresponding output shape.

    Args:
        img (numpy.ndarray):
        theta (float): The rotation degree. The value should in the range of [0, 360].

    Returns:
        (np.ndarray, (int, int)): The generated transformation matrix, and the shape of output image (n_row, n_col).
    """
    radians = theta * np.pi / 180
    trans_mat = np.array([[np.cos(radians), -np.sin(radians)],
                          [np.sin(radians), np.cos(radians)]])
    corner_top_left = np.array([0, 0]).T
    corner_top_right = np.array([0, img.shape[1] - 1]).T
    corner_bot_left = np.array([img.shape[0] - 1, 0]).T
    corner_bot_right = np.array([img.shape[0] - 1, img.shape[1] - 1]).T

    if 0 <= theta < 90 or 180 <= theta < 270:
        n_r = np.abs((trans_mat @ corner_top_right)[0] - (trans_mat @ corner_bot_left)[0])
        n_c = np.abs((trans_mat @ corner_bot_right)[1] - (trans_mat @ corner_top_left)[1])
    if 90 <= theta < 180 or 270 <= theta <= 360:
        n_r = np.abs((trans_mat @ corner_bot_right)[0] - (trans_mat @ corner_top_left)[0])
        n_c = np.abs((trans_mat @ corner_bot_left)[1] - (trans_mat @ corner_top_right)[1])
    output_shape = (int(np.ceil(n_r)), int(np.ceil(n_c))) if img.ndim == 2 else (int(np.ceil(n_r)), int(np.ceil(n_c)), 3)
    return trans_mat, output_shape


def inv_warp(img, inv_trans_mat, out_size):
    """Inverse image warpping.

    Args:
        img (numpy.ndarray): The image to be rotated.
        inv_trans_mat (numpy.ndarray): The inverse of transformation matrix used to rotate the image.
        out_size (Tuple(int, int)): The size of output rotated image.

    Returns:
        numpy.ndarray: The rotated image.
    """
    n_row, n_col = out_size[0], out_size[1]
    cent_x, cent_y = img.shape[0] / 2, img.shape[1] / 2
    rot_u, rot_v = inv(inv_trans_mat) @ np.array([cent_x, cent_y]).T
    cent_u, cent_v = n_row / 2, n_col / 2
    shift_u, shift_v = rot_u - cent_u, rot_v - cent_v
    res = np.zeros(out_size)
    cond = lambda x, y: (0 <= x < img.shape[0]) and (0 <= y < img.shape[1])
    for u in range(n_row):
        for v in range(n_col):
            x, y = inv_trans_mat @ np.array([u + shift_u, v + shift_v]).T
            if cond(x, y):
                if int(x) == x and int(y) == y:
                    res[u, v] = img[int(x), int(y)]
                else:
                    # bilinear interpolation
                    y1, y2 = int(np.floor(y)), int(np.ceil(y))
                    x1, x2 = int(np.floor(x)), int(np.ceil(x))
                    if cond(x1, y2) and cond(x2, y2) and cond(x1, y1) and cond(x2, y1):
                        if x1 == x2:
                            res[u, v] = img[x1, y2] * (y - y1) / (y2 - y1) + img[x1, x1] * (y2 - y) / (y2 - y1)
                        elif y1 == y2:
                            res[u, v] = img[x2, y2] * (x2 - x) / (x2 - x1) + img[x1, y2] * (x - x1) / (x2 - x1)
                        else:
                            R1 = img[x1, y2] * ((x2 - x) / (x2 - x1)) + img[x2, y2] * ((x - x1) / (x2 - x1))
                            R2 = img[x1, y1] * ((x2 - x) / (x2 - x1)) + img[x2, y1] * ((x - x1) / (x2 - x1))
                            res[u, v] = R1 * (y - y1) / (y2 - y1) + R2 * (y2 - y) / (y2 - y1)

    return res.astype('uint8')


img = plt.imread(img_path)
fig, ax = plt.subplots(2, 4)
rotate_imgs = []
for i, theta in enumerate([0, 90, 180, 270]):
    trans_mat, output_shape = trans_mat_out_shape(img, theta=theta)
    rotate_img = inv_warp(img, inv(trans_mat), output_shape)
    rotate_imgs.append(rotate_img)
    ax[0, i].imshow(rotate_img, cmap='gray' if rotate_img.ndim == 2 else 'viridis')
    ax[0, i].axis('off')
    ax[0, i].set_title(str(theta) + " degrees")
    bw = cv2.cvtColor(rotate_img, cv2.COLOR_RGB2GRAY) if rotate_img.ndim == 3 else rotate_img
    Ix = conv2(bw, dx)  # horizontal derivative
    Iy = conv2(bw, dy)  # vertical derivative
    Iy2 = conv2(np.power(Iy, 2), g)  # smoothed second-order vertical derivative (Iy)
    Ix2 = conv2(np.power(Ix, 2), g)  # smoothed second-order horizontal derivative (Ix)
    Ixy = conv2(Ix * Iy, g)          # smoothed production of Iy and Ix
    R = construct_R(Ix2, Iy2, Ixy, window_size_M, k)
    corner_thresh, corner_thresh_non_max = thresh_non_max_sup(R, thresh*R.max(), window_size_R)
    ax[1, i].imshow(rotate_img, cmap='gray' if np.ndim(rotate_img) == 2 else 'viridis')
    ax[1, i].scatter(corner_thresh_non_max[:, 1], corner_thresh_non_max[:, 0], color='green', s=0.8, marker='o')
    ax[1, i].axis('off')
fig.suptitle(img_path.split('/')[-1])
plt.show()
```

Task 2: Domain Adaption

1. Complete the following preparation steps

(a) Download the dataset.

Download the dataset from the script *dataset_downloader.py*. Each subfolder has 10 categories of data from DomainNet dataset, where ***real_test*** and ***real_trains*** are from the **Real domain**, and ***sketch_test*** and ***sketch_train*** are from the **Sketch domain**.



Assignment_2 - utils.py

```python
# dataset info
REAL_TRAIN_PATH = './data/real_train'
REAL_TEST_PATH = './data/real_test'
SKETCH_TRAIN_PATH = './data/sketch_train'
SKETCH_TEST_PATH = './data/sketch_test'
```

(b) Implement a Dataset class and load the data with proper transformations.

Assignment_2 - utils.py

```python
class DomainNetDataset(Dataset):
    """The customised DomainNet dataset for Task 2."""
    def __init__(self, root, transform: lambda x: x):
        # return all image info in form of [path, category]
        self.image_info = self.get_image_paths(root)
        # resize + normalise
        self.transform = transform
    def __len__(self):
        # return total number of images
        return len(self.image_info)
    def __getitem__(self, index):
        """
        For each data (X, y):
            X: (H, W, C) np.ndarray for each RGB image
            y: an int from range [0, ..., 9] for each class
        """
        if isinstance(index, slice):  # slicing index
            res = []
            for path, label in self.image_info[index]:
                res.append([self.transform(plt.imread(path)), label_encoding(label)])
        else:  # single index
            path, label = self.image_info[index]
            res = [self.transform(plt.imread(path)), label_encoding(label)]
        return res
    def get_image_paths(self, root):
        """For each image, retrieve its path and category info."""
        res = []
        categories = os.listdir(root)
        for category in categories:
            category_path = os.path.join(root, category)
            image_names = os.listdir(category_path)
            image_paths = [os.path.join(category_path, image_name) for image_name in image_names]
            for image_path in image_paths:
                res.append([image_path, category])
        return res
```

The above image shows the implementation of a PyTorch Dataset class for the downloaded DomainNet dataset.

- When an instance is initialised from DomainNetDataset, ***self.image_info*** will only retrieve the **i**mage **paths** and **categories** to save more storage during usage.
- The attribute ***self.transform*** receives a pre-processing function ***transform***, that reshapes the image into 224 * 224, and normalises it with means (0.485, 0.456, 0.406) and standard deviations (0.229, 0.224, 0.225).

Assignment_2 - utils.py

```python
# preprocessing params
IMAGE_SHAPE = (224, 224)
NORM_MEAN = (0.485, 0.456, 0.406)
NORM_STD = (0.229, 0.224, 0.225)
```

Assignment_2 - utils.py

```python
def transform(image):
    # resizing
    image = cv2.resize(image, IMAGE_SHAPE)
    # normalisation
    image = (image - NORM_MEAN) / NORM_STD
    # set to np.uint8
    image = image.astype('uint8')
    # TODO: Augmentation
    return image
```

- The function ***get_image_paths*** is used to iterate through all the image locations and return their path and category.
- When one wants to get a specific image by indexing, the function ***__getitem__*** can return the image itself in the form of *np.ndarray* and its integer label from range 0-9. Each *np.ndarray* has the shape (H:224, W:224, C:3) after using the function ***transform***. Each label is processed by ***label_encoding*** and has an integer in the range [0, 9] to represent its category. Specifically, we have

Assignment_2 - utils.py

```python
NUM_CLASSES = 10
CLASSES = ['backpack', 'book', 'car', 'pizza', 'sandwich',
           'snake', 'sock', 'tiger', 'tree', 'watermelon']
```

Assignment_2 - utils.py

```python
def label_encoding(category):
    res = np.where(np.array(CLASSES) == category)[0][0]  # e.g. (array[8],)->8
    return res
```

(c) Load the pre-trained ResNet-34 and modify its final layer.

According to the PyTorch documents, when the parameter ***weights*** is set to **ResNet34_Weights.DEFAULT**, it automatically downloads the **latest weights** from the internet. The following function returns a modified pre-trained ResNet-34 model from PyTorch.

Assignment_2 - utils.py

```python
def ResNet34DomainNet(weights=ResNet34_Weights.DEFAULT):
    # load the latest ResNet-34 pretrained model
    model = resnet34(weights=weights)
    # modify the last FC layer to fit the 10 classes
    model.fc = Linear(model.fc.in_features, NUM_CLASSES)
    return model
```

The **final FC layer** is modified with the same input shape but a new output shape to fit the dataset. The parameter ***NUM_CLASSES*** represents the 10 categories and is shown in the last step.

2. Implement the code for fine-tuning a model

(a) Implement the train test functions for fine-tuning.

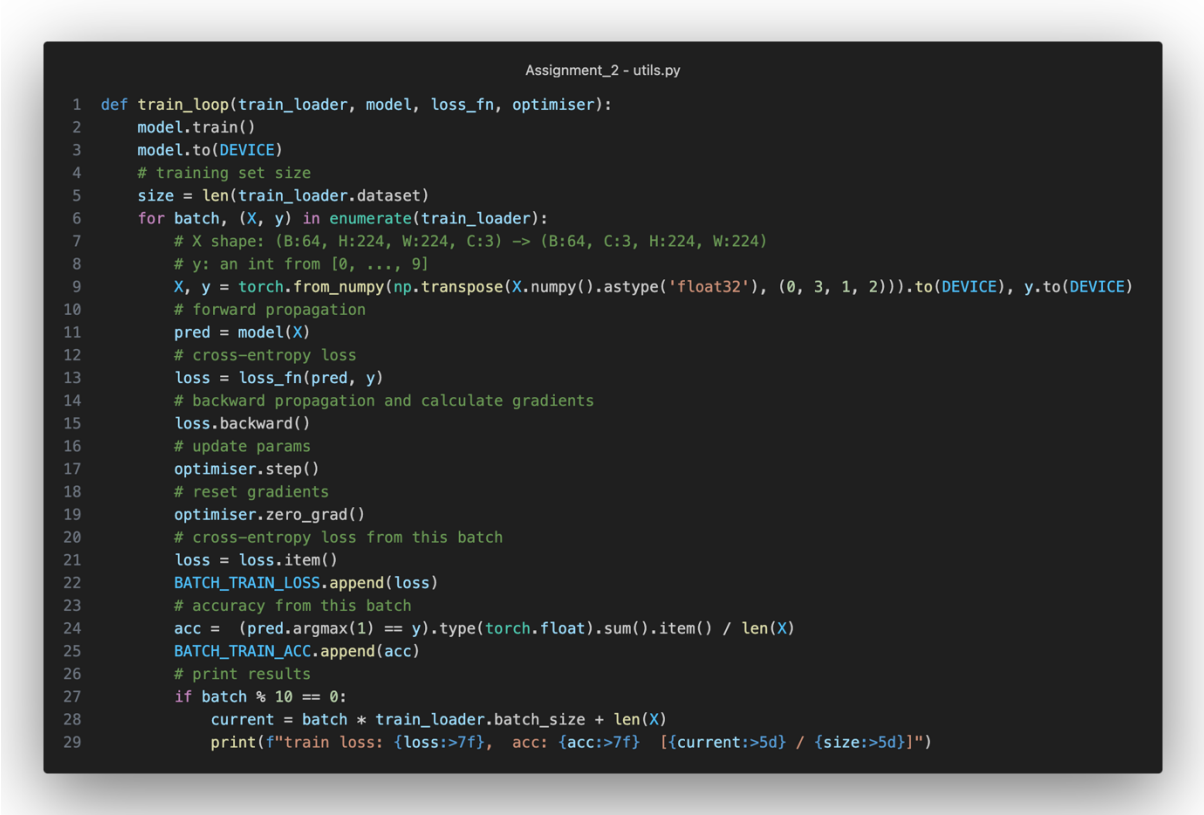
Define a function for creating a data loader in this task:

Assignment_2 - utils.py

```python
def get_dataloader(path, batch_size, transform=transform):
    dataset = DomainNetDataset(path, transform)
    dataloader = DataLoader(dataset, batch_size, shuffle=True)
    return dataloader
```

The above function will load the transformed data from ***DomainNetDataset***, and return a PyTorch DataLoader with the input batch size.

We define a function ***train_loop*** for training the dataset:



For one epoch, the ***train_loop*** will

1. Read the data from ***train_loader***, which is a variable returned from ***get_dataloader***.
2. Conduct a forward propagation.
3. Evaluate with a loss function ***loss_fn*** (e.g., cross-entropy loss from PyTorch or a modified joint loss) and report the results (e.g., loss and accuracy for each batch).
4. Conduct a backward propagation.
5. Update the model parameters with an input optimising function ***optimiser*** (e.g., Adam, SGD from PyTorch).

Assignment_2 - utils.py

```python
def test_loop(test_loader, model, loss_fn):
    model.eval()
    model.to(DEVICE)
    # test set size
    size = len(test_loader.dataset)
    with torch.no_grad():
        for batch, (X, y) in enumerate(test_loader):
            # X shape: (B:64, H:224, W:224, C:3) -> (B:64, C:3, H:224, W:224)
            # y: an int from [0, ..., 9]
            X, y = torch.from_numpy(np.transpose(X.numpy().astype('float32'), (0, 3, 1, 2))).to(DEVICE), y.to(DEVICE)
            pred = model(X)
            # cross-entropy loss from this batch
            loss = loss_fn(pred, y)
            loss = loss.item()
            BATCH_TEST_LOSS.append(loss)
            # accuracy from this batch
            acc =  (pred.argmax(1) == y).type(torch.float).sum().item() / len(X)
            BATCH_TEST_ACC.append(acc)
            # print results
            if batch % 10 == 0:
                current = batch * test_loader.batch_size + len(X)
                print(f"test loss: {loss:>7f}, acc: {acc:>7f}    [{current:>5d} / {size:>5d}]")
```

Similarly, for one epoch, the function ***test_loop*** will:

1. Read the data from ***test_loade****r*, which is a variable returned from ***get_dataloader.***
2. Conduct a forward propagation.
3. Evaluate with a loss function ***loss_fn*** (e.g., cross-entropy loss from PyTorch or a modified joint loss) and report the results (e.g., loss and accuracy for each batch).

(b) Train on *sketch_train* dataset and evaluate on *sketch_test* dataset.

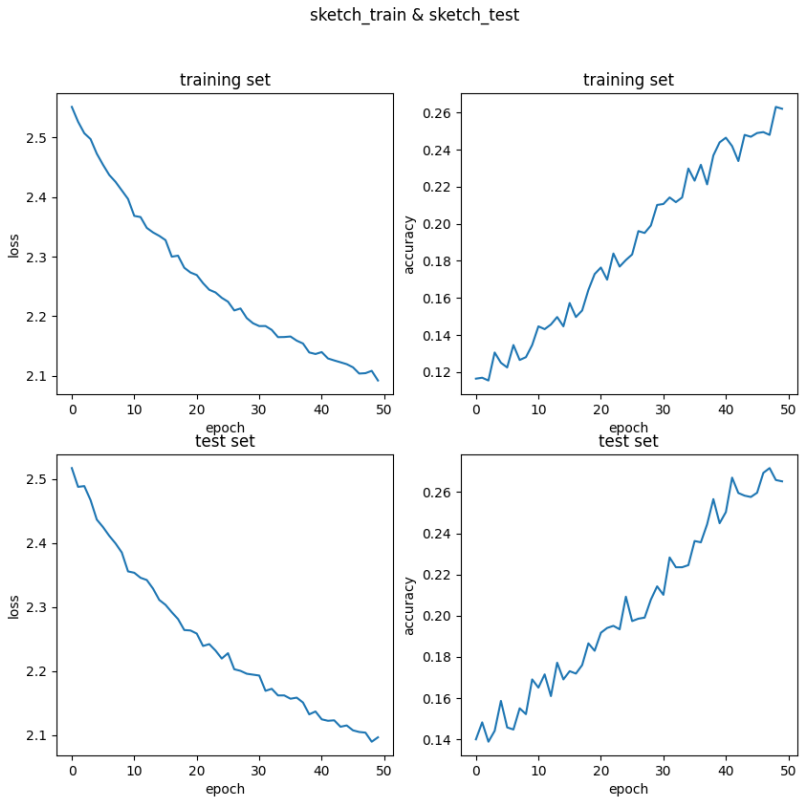
The following script ***train_sketch_test_sketch.py*** shows the training and testing process, where:

- Training set: *sketch_train*
- Test set*:* sketch_test
- Epochs: 50
- Batch size: 32
- Learning rate: 1e-5
- Model: modified pre-trained ResNet-34 from PyTorch
- Optimiser: SGD from PyTorch
- Loss function: Cross-entropy loss from PyTorch

Assignment_2 - train_sketch_eval_sketch.py

```python
from utils import *
from torch.optim import Adam, SGD
from torch.nn import CrossEntropyLoss


if __name__ == '__main__':
    EPOCHS = 50
    BATCH_SIZE = 32
    LEARNING_RATE = 1e-5
    MODEL = ResNet34DomainNet()
    OPTIMISER = SGD(MODEL.parameters(), LEARNING_RATE)
    # OPTIMISER = Adam(MODEL.parameters(), LEARNING_RATE)
    LOSS_FN = CrossEntropyLoss()

    sketch_train_loader = get_dataloader(SKETCH_TRAIN_PATH, BATCH_SIZE)
    sketch_test_loader = get_dataloader(SKETCH_TEST_PATH, BATCH_SIZE)

    for epoch in range(EPOCHS):
        print(f"--------------- epoch {epoch+1} ---------------")
        train_loop(sketch_train_loader, MODEL, LOSS_FN, OPTIMISER)
        test_loop(sketch_test_loader, MODEL, LOSS_FN)
        avg_train_loss = np.mean(BATCH_TRAIN_LOSS)
        avg_train_acc = np.mean(BATCH_TRAIN_ACC)
        avg_test_loss = np.mean(BATCH_TEST_LOSS)
        avg_test_acc = np.mean(BATCH_TEST_ACC)
        print(f"avg train loss: {avg_train_loss:>7f},  avg train acc: {avg_train_acc:>7f}")
        print(f"avg test loss: {avg_test_loss:>7f},  avg train acc: {avg_test_acc:>7f}")

    plot_metric(BATCH_TRAIN_LOSS, BATCH_TRAIN_LOSS,
            BATCH_TEST_LOSS, BATCH_TEST_LOSS,
            title="sketch_train & sketch_test")
```

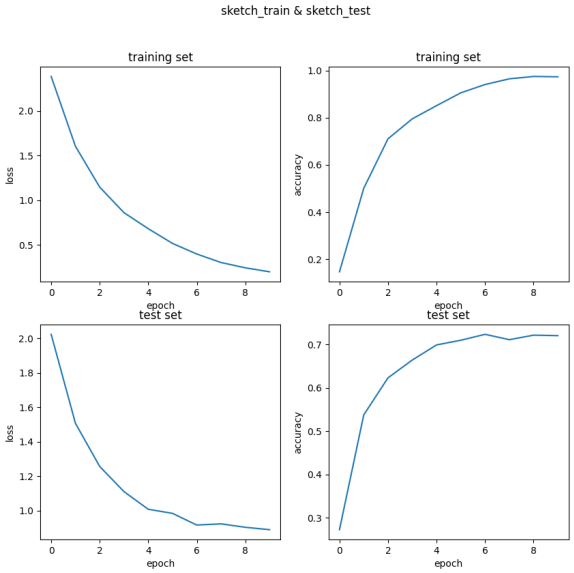
The training loss, training accuracy, test loss and test accuracy are shown below:



We can see:

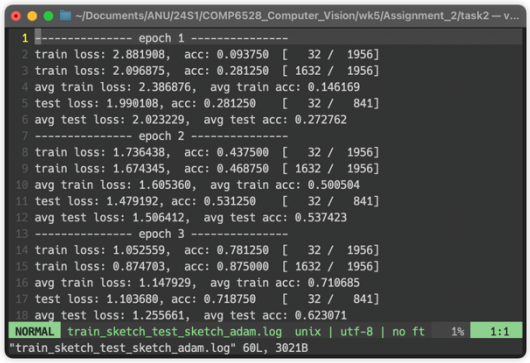
- During the training on *sketch_train*, the cross-entropy loss is continuously decreasing along each epoch. The accuracy on *sketch_train* is increasing in general but still not performing well. The highest accuracy is around 26.5%.
- During the testing on sketch_test, the cross-entropy loss is decreasing in general. The loss from both the training set and the test set ended up around the same level. The accuracy on *sketch_test* increases gradually through the whole process. Notice that the highest accuracy is still around 26.5%, which is the same level as on the training set.

In the previous experiment, when the optimiser is set to **Adam** from PyTorch and all other parameters remain the same, the highest accuracy on *sketch_train* can reach 95% under 7 epochs while the final accuracy on *sketch_test* is around 70%:



Due to the lack of computation resources, the SDG optimiser could perform better on *sketch_train* and *sketch_test* with more epochs. We will keep using the SGD optimiser for compartments with later tasks.

The logs are saved as "***train_sketch_test_sketch_sgd.log***" (right) and "**train_sketch_test_sketch_adam.log**" (left) separately:



(c) Train on *real_train* dataset and evaluate on *sketch_test* dataset.

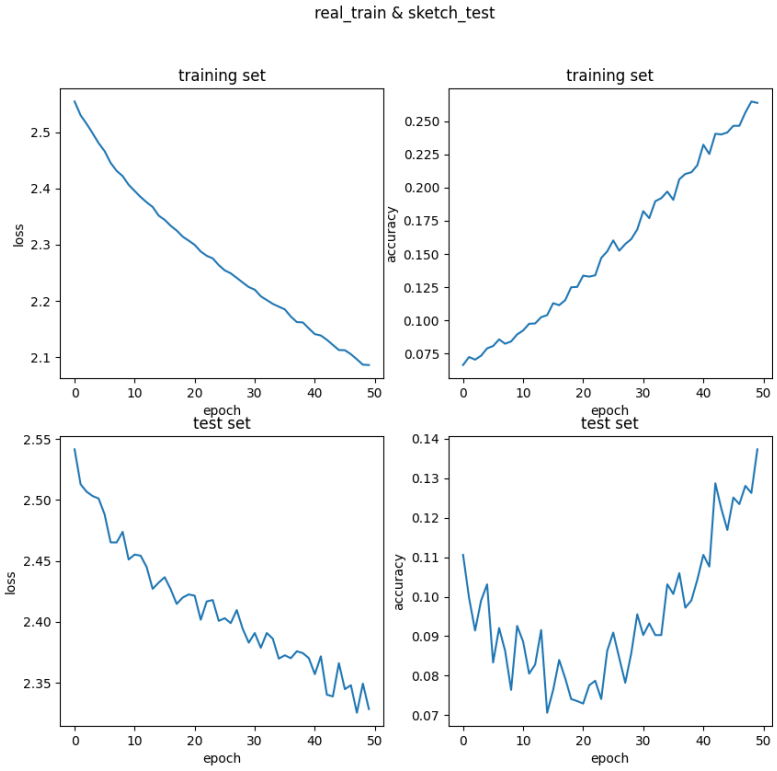
The following script ***train_real_test_sketch.py*** shows the training and testing process, where:

- Training set: *real_train*
- Test set: sketch_test
- Epochs: 50
- Batch size: 32
- Learning rate: 1e-5
- Model: modified pre-trained ResNet-34 from PyTorch
- Optimiser: SGD from PyTorch
- Loss function: Cross-entropy loss from PyTorch

Assignment_2 - train_real_eval_sketch.py

```python
from utils import *
import numpy as np
from torch.optim import Adam, SGD
from torch.nn import CrossEntropyLoss


if __name__ == '__main__':
    EPOCHS = 50
    BATCH_SIZE = 32
    LEARNING_RATE = 1e-5
    MODEL = ResNet34DomainNet()
    OPTIMISER = SGD(MODEL.parameters(), LEARNING_RATE)
    # OPTIMISER = Adam(MODEL.parameters(), LEARNING_RATE)
    LOSS_FN = CrossEntropyLoss()

    real_train_loader = get_dataloader(REAL_TRAIN_PATH, BATCH_SIZE)
    sketch_test_loader = get_dataloader(SKETCH_TEST_PATH, BATCH_SIZE)

    for epoch in range(EPOCHS):
        print(f"---------- epoch {epoch+1} ----------")
        train_loop(real_train_loader, MODEL, LOSS_FN, OPTIMISER)
        test_loop(sketch_test_loader, MODEL, LOSS_FN)
        avg_train_loss = np.mean(BATCH_TRAIN_LOSS)
        avg_train_acc = np.mean(BATCH_TRAIN_ACC)
        avg_test_loss = np.mean(BATCH_TEST_LOSS)
        avg_test_acc = np.mean(BATCH_TEST_ACC)
        print(f"avg train loss: {avg_train_loss:>7f},  avg train acc: {avg_train_acc:>7f}")
        print(f"avg test loss: {avg_test_loss:>7f},  avg train acc: {avg_test_acc:>7f}")


    plot_metric(BATCH_TRAIN_LOSS, BATCH_TRAIN_ACC,
                BATCH_TEST_LOSS, BATCH_TEST_ACC,
                title="real_train & sketch_test")
```

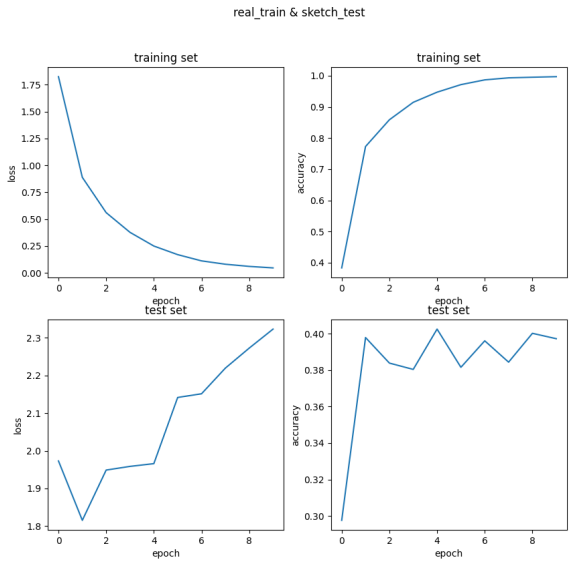
The training loss, training accuracy, test loss and test accuracy are shown below:



We can see:

- During the training on *real_train*, the cross-entropy loss is continuously decreasing along each epoch. The accuracy on *real_train* is increasing in general but still not performing well. The highest accuracy is around 26.5%, which is about the same level shown in the last task.
- During the testing on sketch_test, the cross-entropy loss is decreasing in general but still has a large gap with the loss from *real_train*. The accuracy on *sketch_test* decreases during the first 20 epochs but increases gradually afterwards. Notice that the highest accuracy is less than 14%.

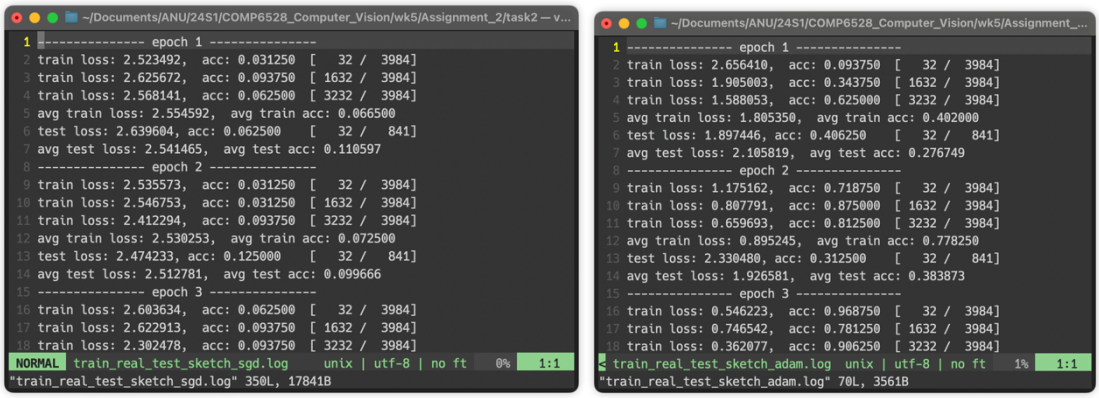
In the previous experiment, when the optimiser is set to **Adam** from PyTorch and all other parameters remain the same,, the highest accuracy on *real_train* can reach 95% under 5 epochs while the final accuracy on *real_test* is less than 40%:



In both cases, the accuracies on *sketch_test* when the model is trained with *real_train* have a huge gap from the results in the last task (train on *sketch_train* and test on *sketch_test*): 14% v.s. 27% with SGD, and 40% v.s. 70% with Adam.

It suggests that the domain of the training set and test set are significantly different, which will be fixed through domain adaption techniques in the later tasks.

The logs are saved as "***train_real_test_sketch_sgd.log***" (left) and "***train_real_test_sketch_adam.log***" (right) separately:



(d) Report the accuracy of the above tasks. Compare and provide the explanations.

When the optimiser is **SGD (learning rate** 1e-5**),** the **epoch** number is 50, and the **batch size** is 32, the **final accuracy** on the **test set:**

- *sketch_train* & *sketch_test*: **26.5%**
- *real_train & sketch_test:* **13.5%**

When the optimiser is **Adam (learning rate** 1e-5**),** the **epoch** number is 10, and the **batch size** is 32, the **final accuracy** on the **test set**:

- *sketch_train* & *sketch_test*: **70%**
- *real_train & sketch_test:* **40%**

We see with both optimisers, that the accuracy under the training set ***real_train*** always has a huge gap with the accuracy under the training set ***sketch_train*** (13.5% v.s. 26.5% and 40% v.s. 70%).

It suggests that the model trained on a specific domain ***(real_train***) does not generalise well to the new domain (***sketch_test***). Specifically, **domain shift** problems occur when the distributions of the source domain (***real_train***) and the target domain (***sketch_test***) are different. These differences can include changes in data statistics such as mean, variance, or other higher-order moments, as well as changes in the underlying data structures or relationships.

More specifically, in this task, the input feature distributions are different in *real_train* and *sketch_test*. The images in *real_train* are more colourful compared with the sketched images in *sketch_test*, which are more monochrome in most cases. The images in *real_train* are more like "photos" compared with "comic sketches" or "edited photos" in *sketch_test*. It suggests both **covariate shift** and **concept shift** problems occur in this task.

3. Implement the code for domain adaption

(a) Implementation of MMD (Maximum Mean Discrepancy) loss.

Assignment_2 - utils.py

```
class MaximumMeanDiscrepancyLoss(Module):
    def __init__(self):
        super(MaximumMeanDiscrepancyLoss, self).__init__()

    def forward(self, outputs, targets):
        # mean of source logits
        mu_s = torch.as_tensor(outputs, dtype=torch.float32).mean(axis=0)
        # mean of target logits
        mu_t = torch.as_tensor(targets, dtype=torch.float32).mean(axis=0)
        # return the L2 norm of the difference between mu_s and mu_t
        return torch.norm(mu_s - mu_t)
```

We choose MMD in the following domain adaption tasks. The implementation is a subclass of "*torch.nn.Module*" from PyTorch.

To customise a loss function and use it in the later model training and testing process, the initialisation method **"*__init__*"** is crucial as it registers all the components of "*Module*" with this subclass.

The "***forward***" method implements the calculation of MMD loss. Specifically, it returns the L2 difference between the mean of source feature vectors and the mean of target feature vectors.

(b) Domain adaptation and comparison

The following two screenshots include the main training (top) and testing (bottom) process:

Assignment_2 - dom_adapt.py

```
for batch, ((X_real, y_real), (X_sketch, y_sketch)) in enumerate(zip(real_train_loader, sketch_train_loader)):
            # import data from both datasets
            X_real = torch.from_numpy(np.transpose(X_real.numpy().astype('float32'), (0, 3, 1, 2))).to(DEVICE)
            y_real = y_real.to(DEVICE)
            X_sketch = torch.from_numpy(np.transpose(X_sketch.numpy().astype('float32'), (0, 3, 1, 2))).to(DEVICE)
            # logits to be calculated in MMD loss function (BATCH_SIZE, NUM_CLASSES)
            pred_real = MODEL(X_real)
            pred_sketch = MODEL(X_sketch)
            # MMD loss: L2 dist between mean of pred_real and mean of pred_sketch
            mmd_loss = MMD_LOSS_FN(pred_real, pred_sketch)
            # classification loss: only source logits, pred_real and y_real
            cls_loss = CLS_LOSS_FN(pred_real, y_real)
            # joint loss
            joint_loss = cls_loss + MMD_LOSS_WEIGHT * mmd_loss
            # bp
            joint_loss.backward()
            # update parameters
            OPTIMISER.step()
            OPTIMISER.zero_grad()
            # loss
            loss_mdd = mmd_loss.item()
            DA_BATCH_MMD_LOSS.append(loss_mdd)
            loss_cls = cls_loss.item()
            DA_BATCH_CLS_LOSS.append(loss_cls)
            loss_joint = joint_loss.item()
            DA_BATCH_JON_LOSS.append(loss_joint)
            # acc for real datasets
            acc = (pred_real.argmax(1) == y_real).type(torch.float).sum().item() / len(X_real)
            DA_BATCH_TRAIN_ACC.append(acc)
            # print results
            if batch % 50 == 0:
                current_real = batch * real_train_loader.batch_size + len(X_real)
                current_sketch = batch * sketch_train_loader.batch_size + len(X_sketch)
                print(f"train loss {loss_joint:>7f} ({loss_cls:>3f}+{MMD_LOSS_WEIGHT}*{loss_mdd:>3f}), acc: {acc:>7f}")
        avg_train_loss = np.mean(DA_BATCH_JON_LOSS)
        avg_train_acc = np.mean(DA_BATCH_TRAIN_ACC)
        print(f"avg train loss: {avg_train_loss:>7f},  avg train acc: {avg_train_acc:>7f}")
        DA_EPOCH_TRAIN_LOSS.append(avg_train_loss)
        DA_EPOCH_TRAIN_ACC.append(avg_train_acc)
```

In each forward propagation, the model is fed with training data from both *real_train* and *sketch_train*. The MMD loss function will take the logit vectors from the final FC layer and calculate the L2 difference between the two means of feature vectors. The classification loss will calculate the performance on *real_test*. The joint loss will calculate the combination of the two losses with a hyperparameter weight (0.01 for MMD loss). The backward propagation will calculate the gradients from the joint loss function and update the model by the SGD optimiser.

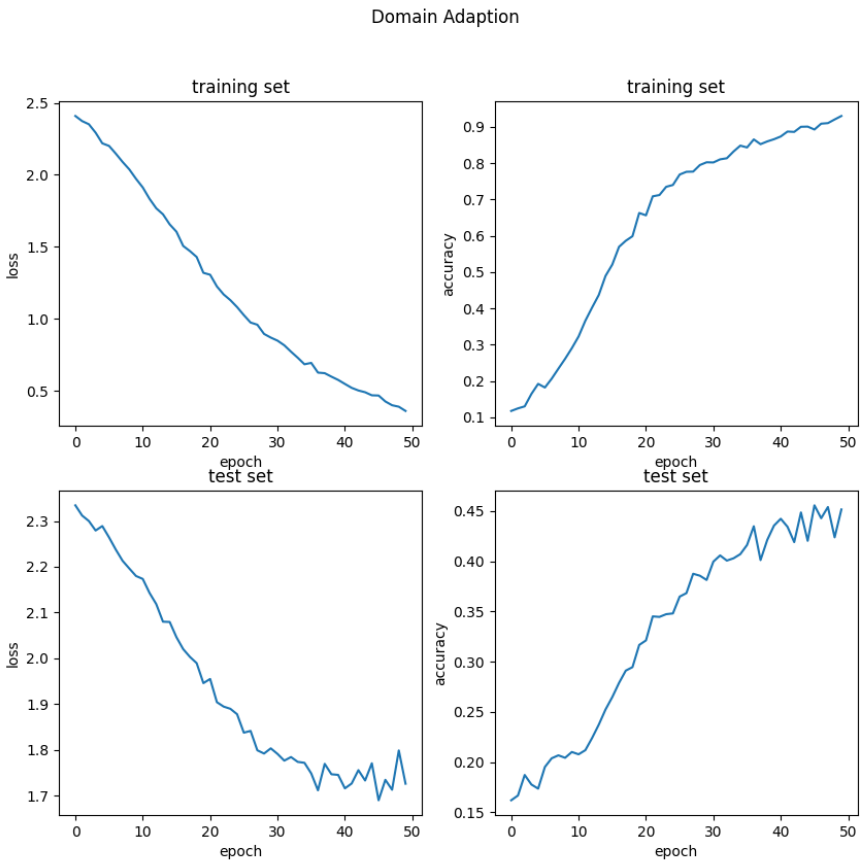
Assignment_2 - dom_adapt.py

```
""" test """
        MODEL.eval()
        MODEL.to(DEVICE)
        size = len(sketch_test_loader.dataset)
        with torch.no_grad():
            for batch, (X, y) in enumerate(sketch_test_loader):
                # X shape: (B:64, H:224, W:224, C:3) -> (B:64, C:3, H:224, W:224)
                # y: an int from [0, ..., 9]
                X, y = torch.from_numpy(np.transpose(X.numpy().astype('float32'), (0, 3, 1, 2))).to(DEVICE), y.to(DEVICE)
                pred = MODEL(X)
                # cross-entropy loss from this batch
                loss = CLS_LOSS_FN(pred, y)
                loss = loss.item()
                DA_BATCH_TEST_LOSS.append(loss)
                # accuracy from this batch
                acc =  (pred.argmax(1) == y).type(torch.float).sum().item() / len(X)
                DA_BATCH_TEST_ACC.append(acc)
                # print results
                if batch % 50 == 0:
                    current = batch * sketch_test_loader.batch_size + len(X)
                    print(f"test loss: {loss:>7f}, acc: {acc:>7f}    [{current:>5d} / {size:>5d}]")
            avg_test_loss = np.mean(DA_BATCH_TEST_LOSS)
            avg_test_acc = np.mean(DA_BATCH_TEST_ACC)
            print(f"avg test loss: {avg_test_loss:>7f},  avg test acc: {avg_test_acc:>7f}")
            DA_EPOCH_TEST_LOSS.append(avg_test_loss)
            DA_EPOCH_TEST_ACC.append(avg_test_acc)
```

The testing process only calculates the cross-entropy loss and accuracy on *sketch_test*.

All the other hyperparameters are the same as the ones in the last task.

The performance from the both training and testing processes:



The final accuracy on *sketch_test* is around **45%**, compared with

- *real_train* & *sketch_test* in Task 2.2 (b)*:***14%**
- *sketch_train* & *sketch*_test in Task 2.2 (c): **26.5%**

We can conclude that there is a significant improvement due to our domain adaptation techniques.

- Compared with only training on *real_train* and testing on *sketch_test*, the domain adaptation improves generalization by effectively transferring knowledge from the source domain (real) to the target domain (sketch), reducing the impact of distribution shift, and adapting the model to better match the characteristics of the target domain data.
- Compared with training on *sketch_train* and *sketch_test*, the model gained more knowledge from real domain, which shares underlying structures or relationships with sketch domain, even though they may have different data distributions. Domain adaptation leverages this shared information to improve model performance on the target domain.

The whole domain adaptation script is named "***dom_adapt.py***" and is shown below:

Assignment_2 - dom_adapt.py

```python
from utils import *
from torch.optim import SGD, Adam
from torch.nn import CrossEntropyLoss


if __name__ == '__main__':
    EPOCHS = 50
    BATCH_SIZE = 32
    LEARNING_RATE = 1e-5
    MODEL = ResNet34DomainNet()
    # OPTIMISER = SGD(MODEL.parameters(), LEARNING_RATE)
    OPTIMISER = Adam(MODEL.parameters(), LEARNING_RATE)
    CLS_LOSS_FN = CrossEntropyLoss()
    MMD_LOSS_FN = MaximumMeanDiscrepancyLoss()
    MMD_LOSS_WEIGHT = 0.01

    real_train_loader = get_dataloader(REAL_TRAIN_PATH, BATCH_SIZE)
    real_test_loader = get_dataloader(REAL_TEST_PATH, BATCH_SIZE)
    sketch_train_loader = get_dataloader(SKETCH_TRAIN_PATH, BATCH_SIZE)
    sketch_test_loader = get_dataloader(SKETCH_TEST_PATH, BATCH_SIZE)

    # train loss and acc
    DA_EPOCH_TRAIN_LOSS = []
    DA_EPOCH_TRAIN_ACC = []
    DA_EPOCH_TEST_LOSS = []
    DA_EPOCH_TEST_ACC = []

    for epoch in range(EPOCHS):
        """ train """
        MODEL.train()
        MODEL.to(DEVICE)
        DA_BATCH_CLS_LOSS = []
        DA_BATCH_MMD_LOSS = []
        DA_BATCH_JON_LOSS = []
        DA_BATCH_TEST_LOSS = []
        DA_BATCH_TRAIN_ACC = []
        DA_BATCH_TEST_ACC = []
        print(f"---------------- epoch {epoch+1} ----------------")
        for batch, ((X_real, y_real), (X_sketch, y_sketch)) in enumerate(zip(real_train_loader, sketch_train_loader)):
            # import data from both datasets
            X_real = torch.from_numpy(np.transpose(X_real.numpy().astype('float32'), (0, 3, 1, 2))).to(DEVICE)
            y_real = y_real.to(DEVICE)
            X_sketch = torch.from_numpy(np.transpose(X_sketch.numpy().astype('float32'), (0, 3, 1, 2))).to(DEVICE)
            # logits to be calculated in MMD loss function (BATCH_SIZE, NUM_CLASSES)
            pred_real = MODEL(X_real)
            pred_sketch = MODEL(X_sketch)
            # MMD loss: L2 dist between mean of pred_real and mean of pred_sketch
            mmd_loss = MMD_LOSS_FN(pred_real, pred_sketch)
            # classification loss: only source logits, pred_real and y_real
            cls_loss = CLS_LOSS_FN(pred_real, y_real)
            # joint loss
            joint_loss = cls_loss + MMD_LOSS_WEIGHT * mmd_loss
            # bp
            joint_loss.backward()
            # update parameters
            OPTIMISER.step()
            OPTIMISER.zero_grad()
            # loss
            loss_mdd = mmd_loss.item()
            DA_BATCH_MMD_LOSS.append(loss_mdd)
            loss_cls = cls_loss.item()
            DA_BATCH_CLS_LOSS.append(loss_cls)
            loss_joint = joint_loss.item()
            DA_BATCH_JON_LOSS.append(loss_joint)
            # acc for real datasets
            acc = (pred_real.argmax(1) == y_real).type(torch.float).sum().item() / len(X_real)
            DA_BATCH_TRAIN_ACC.append(acc)
            # print results
            if batch % 50 == 0:
                current_real = batch * real_train_loader.batch_size + len(X_real)
                current_sketch = batch * sketch_train_loader.batch_size + len(X_sketch)
                print(f"train loss {loss_joint:>7f} ({loss_cls:>3f}+{MMD_LOSS_WEIGHT}*{loss_mdd:>3f}), acc: {acc:>7f}")
        avg_train_loss = np.mean(DA_BATCH_JON_LOSS)
        avg_train_acc = np.mean(DA_BATCH_TRAIN_ACC)
        print(f"avg train loss: {avg_train_loss:>7f},  avg train acc: {avg_train_acc:>7f}")
        DA_EPOCH_TRAIN_LOSS.append(avg_train_loss)
        DA_EPOCH_TRAIN_ACC.append(avg_train_acc)

        """ test """
        MODEL.eval()
        MODEL.to(DEVICE)
        size = len(sketch_test_loader.dataset)
        with torch.no_grad():
            for batch, (X, y) in enumerate(sketch_test_loader):
                # X shape: (B:64, H:224, W:224, C:3) -> (B:64, C:3, H:224, W:224)
                # y: an int from [0, ..., 9]
                X, y = torch.from_numpy(np.transpose(X.numpy().astype('float32'), (0, 3, 1, 2))).to(DEVICE), y.to(DEVICE)
                pred = MODEL(X)
                # cross-entropy loss from this batch
                loss = CLS_LOSS_FN(pred, y)
                loss = loss.item()
                DA_BATCH_TEST_LOSS.append(loss)
                # accuracy from this batch
                acc =  (pred.argmax(1) == y).type(torch.float).sum().item() / len(X)
                DA_BATCH_TEST_ACC.append(acc)
                # print results
                if batch % 50 == 0:
                    current = batch * sketch_test_loader.batch_size + len(X)
                    print(f"test loss: {loss:>7f}, acc: {acc:>7f}    [{current:>5d} / {size:>5d}]")
            avg_test_loss = np.mean(DA_BATCH_TEST_LOSS)
            avg_test_acc = np.mean(DA_BATCH_TEST_ACC)
            print(f"avg test loss: {avg_test_loss:>7f},  avg test acc: {avg_test_acc:>7f}")
            DA_EPOCH_TEST_LOSS.append(avg_test_loss)
            DA_EPOCH_TEST_ACC.append(avg_test_acc)


    plot_metric(DA_EPOCH_TRAIN_LOSS, DA_EPOCH_TRAIN_ACC,
                DA_EPOCH_TEST_LOSS, DA_EPOCH_TEST_ACC,
                title="Domain Adaption")
```

(c) Discussion of the weight of domain adaptation loss and hyperparameter selections.

The recommended weights do not always work in practice. Consider another different target dataset, which includes 5 categories of animal pictures, while keeping the source dataset the same (real domain). It generates new issues such as:

- New label shift: The 10 categories in the sketch domain vs. the 5 categories of animal pictures. The label distribution or conditional probability distribution changes variously depending on different target datasets.
- New covariate shift: The new animal domain has different input features from the sketch domain. Since the animal pictures are more colourful in most cases while the sketch images are more monochrome.
- New concept shift: The pictures in the new animal domain include 5 categories of animals, while the sketch domain includes 10 categories of different objects. The underlying concepts or relationships have been changed in the new target domain.

Due to the above new challenges, the new domain adaptation loss with be different from the loss with the sketch domain. The joint loss will need to find a new balance through a different weight setting to achieve a better performance.

The issues from the textbook's way of selecting hyperparameters:

The assumption ignores the domain shift, which is more common and happens often in real life. When using a labelled validation set for hyperparameter tuning, the tunned model will be more fitted with the source domain data but will not generalise well for the target domain. Without considering the differences between the source and target domains, the model may fail to capture the relevant patterns and relationships in the target domain data.

Notice that there is a risk of bias towards the source domain and a lack of consideration for the unique characteristics of the target domain. This can result in suboptimal model performance and potentially biased decision-making when the model is applied to the target domain.