```
#Data Pre-Processing

#ROI – Preprocessing Generate_ROI.py
import os
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] = "0,1"
import cv2
import numpy as np
import scipy.io as sio
import scipy.misc
from keras.preprocessing import image
from skimage.transform import rotate, resize
from skimage.measure import label, regionprops
from time import time
from mnet_utils import pro_process, BW_img, disc_crop
import matplotlib.pyplot as plt
from skimage.io import imsave




import Model_DiscSeg as DiscModel

#ROI_size_list = [400, 500, 600, 700, 800]
ROI_size_list = [700]
DiscROI_size = 700
DiscSeg_size = 640 # input size to the disc detection model


train_data_type = '.png'
mask_data_type = '.png'

Original_vali_img_path =
'/DATA/charlie/AWC/data_original/drishti/test/image/'
Original_Mask_img_path = '/DATA/charlie/AWC/data_original/drishti/test/mask/'

Image_save_path = '/DATA/charlie/AWC/CADA_Tutorial_Image/Target_Test/image/'
MaskImage_save_path =
'/DATA/charlie/AWC/CADA_Tutorial_Image/Target_Test/mask/'

if not os.path.exists(Image_save_path):
    os.makedirs(Image_save_path)

if not os.path.exists(MaskImage_save_path):
    os.makedirs(MaskImage_save_path)

is_polar_coordinate = False  # in MICCAI version, this is false.
is_only_image = False  #for target domain training images we do not have
masks, then it is True

file_train_list = [file for file in os.listdir(Original_vali_img_path) if
file.lower().endswith(train_data_type)]
print(str(len(file_train_list)))


```

```
DiscSeg_model = DiscModel.DeepModel(size_set=DiscSeg_size)
DiscSeg_model.load_weights('Model_DiscSeg_pretrain.h5')


for lineIdx in range(0, len(file_train_list)):


    ########################Generate mask ROIs
############################


    temp_txt = [elt.strip() for elt in file_train_list[lineIdx].split(',')]
    nameLen = len(temp_txt[0])
    # print(' Processing Img: ' + temp_txt[0])
    # load image
    org_img = np.asarray(image.load_img(Original_vali_img_path +
temp_txt[0]))
    # plt.imshow(org_img)
    # plt.title('org_img')
    # plt.show()

    # Disc region detection by U-Net
    temp_org_img = resize(org_img, (DiscSeg_size, DiscSeg_size, 3))
    # plt.imshow(temp_org_img)
    # plt.title('temp_org_img')
    # plt.show()



    if is_only_image == False:
        org_mask = np.asarray(image.load_img(Original_Mask_img_path +
                                             temp_txt[0][:nameLen - 4] +
mask_data_type))[:, :, 0]
        # plt.imshow(org_mask)
        # plt.title('org_mask')
        # plt.show()

        org_disc = org_mask < 255
        # plt.imshow(org_disc)
        # plt.title('org_disc')
        # plt.show()

        org_cup = org_mask == 0
        # plt.imshow(org_cup)
        # plt.title('org_cup')
        # plt.show()



        temp_org_mask = resize(org_mask, (DiscSeg_size, DiscSeg_size))
        # plt.imshow(temp_org_mask)
        # plt.title('temp_org_mask')
        # plt.show()

        temp_org_disc = resize(org_disc, (DiscSeg_size, DiscSeg_size))
        # plt.imshow(temp_org_disc)
        # plt.title('temp_org_disc')
```

```
            # plt.show()

            temp_org_cup = resize(org_cup, (DiscSeg_size, DiscSeg_size))
            # plt.imshow(temp_org_cup)
            # plt.title('temp_org_cup')
            # plt.show()
            org_disc_bw = BW_img(np.reshape(temp_org_disc, (DiscSeg_size,
DiscSeg_size)), 0.5)
            org_cup_bw = BW_img(np.reshape(temp_org_cup, (DiscSeg_size,
DiscSeg_size)), 0.5)


        temp_org_img = np.reshape(temp_org_img, (1,) + temp_org_img.shape) * 255

        prob_10 = DiscSeg_model.predict([temp_org_img])

        # plt.imshow(np.squeeze(np.clip(prob_10*255,0,255).astype('uint8')))
        # plt.title('temp_img')
        # plt.show()

        org_img_disc_map = BW_img(np.reshape(prob_10, (DiscSeg_size,
DiscSeg_size)), 0.5)
        regions = regionprops(label(org_img_disc_map))

        C_x = int(regions[0].centroid[0] * org_img.shape[0] / DiscSeg_size)
        C_y = int(regions[0].centroid[1] * org_img.shape[1] / DiscSeg_size)

        for disc_idx, DiscROI_size in enumerate(ROI_size_list):

            org_img_disc_region, err_coord, crop_coord = disc_crop(org_img,
DiscROI_size, C_x, C_y)
            # plt.imshow(org_img_disc_region)
            # plt.title('org_img_disc_region')
            # plt.show()

            if is_only_image == False:
                org_mask_region, err_coord, crop_coord = disc_crop(org_mask,
DiscROI_size, C_x, C_y)
                org_disc_region, err_coord_disc, crop_coord_disc =
disc_crop(org_disc, DiscROI_size, C_x, C_y)
                # plt.imshow(org_disc_region)
                # plt.title('org_disc_region')
                # plt.show()

                org_cup_region, err_coord_cup, crop_coord_cup =
disc_crop(org_cup, DiscROI_size, C_x, C_y)
                # plt.imshow(org_cup_region)
                # plt.title('org_cup_region')
                # plt.show()

                ROI_mask_result = np.array(BW_img(org_disc_region, 0.5),
dtype=int) + np.array(BW_img(org_cup_region, 0.5),

dtype=int)
                # plt.imshow(ROI_mask_result)
                # plt.title('ROI_mask_result')
                # plt.show()
```

```
169
170             ROI_mask_result = (255.0 / ROI_mask_result.max() *
171 (ROI_mask_result - ROI_mask_result.min())).astype(np.uint8)
172             ROI_mask_result[ROI_mask_result == 255] = 200
173             ROI_mask_result[ROI_mask_result == 0] = 255
174             ROI_mask_result[ROI_mask_result == 200] = 0
175             ROI_mask_result[(ROI_mask_result > 0) & (ROI_mask_result < 255)]
176 = 128
177             # plt.imshow(ROI_mask_result)
178             # plt.title('ROI_mask_result')
179             # plt.show()
180
181         if is_polar_coordinate:
182
183             if is_only_image == False:
184                 ROI_mask_result = rotate(cv2.linearPolar(ROI_mask_result,
185 (DiscROI_size / 2, DiscROI_size / 2), DiscROI_size / 2,
186                                          cv2.INTER_NEAREST +
187 cv2.WARP_FILL_OUTLIERS), -90)
188                 # plt.imshow(ROI_mask_result)
189                 # plt.title('ROI_mask_result')
190                 # plt.show()
191
192
193             ROI_img_result = rotate(cv2.linearPolar(org_img_disc_region,
194 (DiscROI_size / 2, DiscROI_size / 2), DiscROI_size / 2,
195                                          cv2.INTER_NEAREST +
196 cv2.WARP_FILL_OUTLIERS), -90)
197         else:
198             ROI_img_result = org_img_disc_region
199
200
201         # filename_ROI_Img = '{}_{}{}'.format(temp_txt[0][:nameLen - 4],
202 DiscROI_size, train_data_type)
203         filename_ROI_Img = '{}{}'.format(temp_txt[0][:nameLen - 4], '.jpg')
204         imsave(Image_save_path + filename_ROI_Img, ROI_img_result)
205         if is_only_image == False:
206             filename_ROI_Mask = '{}{}'.format(temp_txt[0][:nameLen - 4],
207 '.bmp')
208             imsave(MaskImage_save_path + filename_ROI_Mask, ROI_mask_result)
209
210     ##########################Generate mask ROI done
211 ############################
212
```

#M-NET Utils – mnet_utils.py

```
213 #The code is modified from https://github.com/HzFu/MNet_DeepCDR
214 # -*- coding: utf-8 -*-
215
216 from __future__ import print_function
217
218 import os
219
220 import numpy as np
221 from PIL import Image
222 from scipy.ndimage import binary_fill_holes
223
```

```
from skimage.measure import label, regionprops
from tensorflow.python.keras import backend as K
from tensorflow.python.keras.preprocessing import image


def pro_process(temp_img, input_size):
    img = np.asarray(temp_img*255).astype('uint8')
    img = np.array(Image.fromarray(img).resize((input_size, input_size)))
    return img


def train_loader(data_list, data_path, mask_path, input_size):
    while 1:
        for lineIdx, temp_txt in enumerate(data_list):
            train_img = np.asarray(image.load_img(os.path.join(data_path,
temp_txt),
                                                  target_size=(input_size,
input_size, 3))
                                   ).astype('float32')
            img_mask = np.asarray(
                image.load_img(os.path.join(mask_path, temp_txt),
                               target_size=(input_size, input_size, 3))
            ) / 255.0

            train_img = np.reshape(train_img, (1,) + train_img.shape)
            img_mask = np.reshape(img_mask, (1,) + img_mask.shape)
            yield ([train_img], [img_mask, img_mask, img_mask, img_mask,
img_mask])


def BW_img(input, thresholding):
    if input.max() > thresholding:
        binary = input > thresholding
    else:
        binary = input > input.max() / 2.0

    label_image = label(binary)
    regions = regionprops(label_image)
    area_list = [region.area for region in regions]
    if area_list:
        idx_max = np.argmax(area_list)
        binary[label_image != idx_max + 1] = 0
    return binary_fill_holes(np.asarray(binary).astype(int))


def dice_coef(y_true, y_pred):
    smooth = 1.
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return 1- (2. * intersection + smooth) / (K.sum(y_true_f) +
K.sum(y_pred_f) + smooth)


def dice_coef2(y_true, y_pred):
    score0 = dice_coef(y_true[:, :, :, 0], y_pred[:, :, :, 0])
    score1 = dice_coef(y_true[:, :, :, 1], y_pred[:, :, :, 1])
```

```
    score = 0.5 * score0 + 0.5 * score1

    return score


def dice_coef_loss(y_true, y_pred):
    return dice_coef2(y_true, y_pred)



def disc_crop(org_img, DiscROI_size, C_x, C_y):
    tmp_size = int(DiscROI_size / 2);
    if len(org_img.shape) == 2:
        disc_region = np.zeros((DiscROI_size, DiscROI_size),
dtype=org_img.dtype)
    else:
        disc_region = np.zeros((DiscROI_size, DiscROI_size, 3),
dtype=org_img.dtype)

    crop_coord = np.array([C_x - tmp_size, C_x + tmp_size, C_y - tmp_size,
C_y + tmp_size], dtype=int)
    err_coord = [0, DiscROI_size, 0, DiscROI_size]

    if crop_coord[0] < 0:
        err_coord[0] = abs(crop_coord[0])
        crop_coord[0] = 0

    if crop_coord[2] < 0:
        err_coord[2] = abs(crop_coord[2])
        crop_coord[2] = 0

    if crop_coord[1] > org_img.shape[0]:
        err_coord[1] = err_coord[1] - (crop_coord[1] - org_img.shape[0])
        crop_coord[1] = org_img.shape[0]

    if crop_coord[3] > org_img.shape[1]:
        err_coord[3] = err_coord[3] - (crop_coord[3] - org_img.shape[1])
        crop_coord[3] = org_img.shape[1]
    if len(org_img.shape) == 2:
        disc_region[err_coord[0]:err_coord[1], err_coord[2]:err_coord[3]] =
org_img[crop_coord[0]:crop_coord[1],

crop_coord[2]:crop_coord[3]]
    else:
        disc_region[err_coord[0]:err_coord[1], err_coord[2]:err_coord[3], ] =
org_img[crop_coord[0]:crop_coord[1],

crop_coord[2]:crop_coord[3], ]

    return disc_region, err_coord, crop_coord


def mk_dir(dir_path):
    if not os.path.exists(dir_path):
        os.makedirs(dir_path)
    return dir_path

```

```
338
339 def files_with_ext(data_path, data_type):
340     file_list = [file for file in os.listdir(data_path) if
341 file.lower().endswith(data_type)]
342     print(len(file_list))
343     return file_list
344
```

#Disc Seg Model – Model_DiscSeg.py

```
345 #The code is modified from https://github.com/HzFu/MNet_DeepCDR
346 # -*- coding: utf-8 -*-
347
348 from __future__ import absolute_import
349 from __future__ import print_function
350
351 from tensorflow.python.keras.layers import (Input, concatenate, Conv2D,
352 MaxPooling2D,
353                                             Conv2DTranspose, UpSampling2D,
354 average)
355 from tensorflow.python.keras.models import Model
356
357
358 def DeepModel(size_set=640):
359     img_input = Input(shape=(size_set, size_set, 3))
360
361     conv1 = Conv2D(32, (3, 3), activation='relu', padding='same',
362 name='block1_conv1')(img_input)
363     conv1 = Conv2D(32, (3, 3), activation='relu', padding='same',
364 name='block1_conv2')(conv1)
365     pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
366
367     conv2 = Conv2D(64, (3, 3), activation='relu', padding='same',
368 name='block2_conv1')(pool1)
369     conv2 = Conv2D(64, (3, 3), activation='relu', padding='same',
370 name='block2_conv2')(conv2)
371     pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
372
373     conv3 = Conv2D(128, (3, 3), activation='relu', padding='same',
374 name='block3_conv1')(pool2)
375     conv3 = Conv2D(128, (3, 3), activation='relu', padding='same',
376 name='block3_conv2')(conv3)
377     pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
378
379     conv4 = Conv2D(256, (3, 3), activation='relu', padding='same',
380 name='block4_conv1')(pool3)
381     conv4 = Conv2D(256, (3, 3), activation='relu', padding='same',
382 name='block4_conv2')(conv4)
383     pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)
384
385     conv5 = Conv2D(512, (3, 3), activation='relu', padding='same',
386 name='block5_conv1')(pool4)
387     conv5 = Conv2D(512, (3, 3), activation='relu', padding='same',
388 name='block5_conv2')(conv5)
389
390     up6 = concatenate(
391         [Conv2DTranspose(256, (2, 2), strides=(2, 2), padding='same',
392
```

```
name='block6_dconv')(conv5), conv4],
        axis=3)
    conv6 = Conv2D(256, (3, 3), activation='relu', padding='same',
name='block6_conv1')(up6)
    conv6 = Conv2D(256, (3, 3), activation='relu', padding='same',
name='block6_conv2')(conv6)

    up7 = concatenate(
        [Conv2DTranspose(128, (2, 2), strides=(2, 2), padding='same',
name='block7_dconv')(conv6), conv3],
        axis=3)
    conv7 = Conv2D(128, (3, 3), activation='relu', padding='same',
name='block7_conv1')(up7)
    conv7 = Conv2D(128, (3, 3), activation='relu', padding='same',
name='block7_conv2')(conv7)

    up8 = concatenate([Conv2DTranspose(64, (2, 2), strides=(2, 2),
padding='same', name='block8_dconv')(conv7), conv2],
                      axis=3)
    conv8 = Conv2D(64, (3, 3), activation='relu', padding='same',
name='block8_conv1')(up8)
    conv8 = Conv2D(64, (3, 3), activation='relu', padding='same',
name='block8_conv2')(conv8)

    up9 = concatenate([Conv2DTranspose(32, (2, 2), strides=(2, 2),
padding='same', name='block9_dconv')(conv8), conv1],
                      axis=3)
    conv9 = Conv2D(32, (3, 3), activation='relu', padding='same',
name='block9_conv1')(up9)
    conv9 = Conv2D(32, (3, 3), activation='relu', padding='same',
name='block9_conv2')(conv9)

    side6 = UpSampling2D(size=(8, 8))(conv6)
    side7 = UpSampling2D(size=(4, 4))(conv7)
    side8 = UpSampling2D(size=(2, 2))(conv8)
    out6 = Conv2D(1, (1, 1), activation='sigmoid', name='side_6')(side6)
    out7 = Conv2D(1, (1, 1), activation='sigmoid', name='side_7')(side7)
    out8 = Conv2D(1, (1, 1), activation='sigmoid', name='side_8')(side8)
    out9 = Conv2D(1, (1, 1), activation='sigmoid', name='side_9')(conv9)

    out10 = average([out6, out7, out8, out9])
    # out10 = Conv2D(1, (1, 1), activation='sigmoid', name='side_10')(out10)

    return Model(inputs=[img_input], outputs=[out10])


#Data Loader

#Data Util – refuge.py
"""Dataset setting and data loader for USPS.

Modified from
https://github.com/mingyuliutw/CoGAN/blob/master/cogan_pytorch/src/dataset_us
ps.py.

use the test data as target data
```

```
    added by PengLiu 12/08/2018
    """

    import os
    import numpy as np
    import scipy.misc as m
    from matplotlib.pyplot import imread
    import cv2
    import glob
    import torch
    import torch.utils.data as data


    src_image_dir = '/DATA/charlie/AWC/CADA_Tutorial_Image/Source/image/'
    src_mask_dir = '/DATA/charlie/AWC/CADA_Tutorial_Image/Source/mask/'

    tgt_image_dir = '/DATA/charlie/AWC/CADA_Tutorial_Image/Target_Train/image/'

    test_image_dir = '/DATA/charlie/AWC/CADA_Tutorial_Image/Target_Test/image/'
    test_mask_dir = '/DATA/charlie/AWC/CADA_Tutorial_Image/Target_Test/mask/'


    class REFUGE(data.Dataset):
        """REFUGE Dataset.
        Args:
            root (string): Root directory of dataset where dataset file exist.
            train (bool, optional): If True, resample from dataset randomly.
            download (bool, optional): If true, downloads the dataset
                from the internet and puts it in root directory.
                If dataset is already downloaded, it is not downloaded again.
            transform (callable, optional): A function/transform that takes in
                an PIL image and returns a transformed version.
                E.g, ``transforms.RandomCrop``
        """

        def __init__(
            self,
            train=True,
            domain='REFUGE_SRC',
            is_transform=False,
            augmentations=None,
            aug_for_target=None,
            img_size=(400, 400),
            max_iters=None
        ):
            self.train = train
            self.is_transform = is_transform
            self.augmentations = augmentations
            self.aug_for_target = aug_for_target
            self.dataset_size = None

            self.domain = domain
            if domain == 'REFUGE_SRC':
                self.img_dir = src_image_dir
                self.mask_dir = src_mask_dir
            if domain == 'REFUGE_DST':
                self.img_dir = tgt_image_dir
```

```
504             if domain == 'REFUGE_TEST':
505                 self.img_dir = test_image_dir
506                 self.mask_dir = test_mask_dir
507             self.class_map = {0: 2, 128: 1, 255: 0}
508             self.img_size = (
509                 img_size if isinstance(img_size, tuple) else (img_size, img_size)
510             )
511 
512             self._glob_img_files()
513             if not max_iters == None:
514                 self.image_files = self.image_files * int(
515                     np.ceil(float(max_iters) / len(self.image_files)))
516 
517         def __getitem__(self, index):
518             """Get images and target for data loader.
519             Args:
520                 index (int): Index
521             Returns:
522                 tuple: (image, target) where target is index of the target class.
523             """
524             image_file = self.image_files[index]
525             img = imread(image_file)
526             img = np.array(img, dtype=np.uint8)
527 
528             if self.domain != 'REFUGE_DST':
529                 label_file = os.path.join(self.mask_dir,
530                                           os.path.basename(image_file))[:-3] +
531     'bmp'
532                 lbl_ori = imread(label_file)
533                 lbl = lbl_ori.copy()
534                 lbl = cv2.resize(lbl,(self.img_size[0],
535     self.img_size[1]),interpolation=cv2.INTER_NEAREST)
536                 lbl = self.encode_segmap(np.array(lbl, dtype=np.uint8))
537 
538             else:
539                 lbl = np.zeros((self.img_size[0], self.img_size[1]),
540     dtype=np.uint8)
541 
542             if self.augmentations is not None:
543 
544                 # data augmentation for student
545                 aug = self.augmentations(image=img, mask=lbl)
546                 img, lbl = aug['image'], aug['mask']
547 
548                 # data augmentation for teacher
549                 aug = self.aug_for_target(image=img, mask=lbl)
550                 img0, lbl0 = aug['image'], aug['mask']
551 
552             else:
553                 img0, lbl0 = img.copy(), lbl.copy()
554 
555 
556             if self.is_transform:
557                 img, lbl = self.transform(img,lbl)
558                 img0, lbl0 = self.transform(img0, lbl0)
559 
560             return img, lbl, img0, lbl0, os.path.basename(image_file)[:-4]
```

```
561
562         def __len__(self):
563             """Return size of dataset."""
564             return len(self.image_files)
565
566         def _glob_img_files(self):
567             """Check if dataset is download and in right place."""
568             if self.domain == 'REFUGE_SRC':
569                 self.image_files = glob.glob(os.path.join(self.img_dir, '*.jpg'))
570             if self.domain == 'REFUGE_DST':
571                 self.image_files = glob.glob(os.path.join(self.img_dir, '*.jpg'))
572             if self.domain == 'REFUGE_TEST':
573                 self.image_files = glob.glob(os.path.join(self.img_dir, '*.jpg'))
574
575
576         def transform(self, img, lbl):
577             """transform
578             :param img:
579             :param lbl:
580             """
581             img = m.imresize(
582                 img, (self.img_size[0], self.img_size[1])
583             )  # uint8 with RGB mode
584             img = img[:, :, ::-1]  # RGB -> BGR
585             img = img.astype(np.float64)
586
587             img = img.transpose(2, 0, 1)
588
589             classes = np.unique(lbl)
590             lbl = lbl.astype(float)
591             lbl = cv2.resize(lbl, (self.img_size[0], self.img_size[1]),
592     interpolation=cv2.INTER_NEAREST)
593             lbl = lbl.astype(int)
594
595             if not np.all(classes == np.unique(lbl)):
596                 print("WARN: resizing labels yielded fewer classes")
597
598             img = torch.from_numpy(img).float()
599             lbl = torch.from_numpy(lbl).long()
600
601             return img, lbl
602
603
604         def encode_segmap(self, mask):
605             # Put all void classes to zero
606             classes = np.unique(mask)
607             for each_class in classes:
608                 assert each_class in self.class_map.keys()
609
610             for _validc in self.class_map.keys():
611                 mask[mask == _validc] = self.class_map[_validc]
612             return mask
```

```
#Evaluation

#Evaluation Metric Calculation – evaluation_segmentation.py
import numpy as np
import matplotlib.pyplot as plt

from scipy import misc
from os import path, makedirs
import glob
from evaluation.file_management import
save_csv_mean_segmentation_performance, save_csv_segmentation_table
#from file_management import save_csv_mean_segmentation_performance,
save_csv_segmentation_table

EPS = 1e-7


def dice_coefficient(binary_segmentation, binary_gt_label):
    """
    Compute the Dice coefficient between two binary segmentation.
    Dice coefficient is defined as here:
    https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient
    Input:
        binary_segmentation: binary 2D numpy array representing the region of
        interest as segmented by the algorithm
        binary_gt_label: binary 2D numpy array representing the region of
        interest as provided in the database
    Output:
        dice_value: Dice coefficient between the segmentation and the ground
        truth
    """

    # turn all variables to booleans, just in case
    binary_segmentation = np.asarray(binary_segmentation, dtype=np.bool)
    binary_gt_label = np.asarray(binary_gt_label, dtype=np.bool)

    # compute the intersection
    intersection = np.logical_and(binary_segmentation, binary_gt_label)

    # count the number of True pixels in the binary segmentation
    segmentation_pixels = float(np.sum(binary_segmentation.flatten()))
    # same for the ground truth
    gt_label_pixels = float(np.sum(binary_gt_label.flatten()))
    # same for the intersection
    intersection = float(np.sum(intersection.flatten()))

    # compute the Dice coefficient
    dice_value = 2 * intersection / (segmentation_pixels + gt_label_pixels)

    return dice_value


def vertical_diameter(binary_segmentation):
    """
    Get the vertical diameter from a binary segmentation.
    The vertical diameter is defined as the "fattest" area of the
```

```
    binary_segmentation parameter.
        Input:
            binary_segmentation: a boolean 2D numpy array representing a region
    of interest.
        Output:
            diameter: the vertical diameter of the structure, defined as the
    largest diameter between the upper and the lower interfaces
        """

        # turn the variable to boolean, just in case
        binary_segmentation = np.asarray(binary_segmentation, dtype=np.bool)

        # get the sum of the pixels in the vertical axis
        vertical_axis_diameter = np.sum(binary_segmentation, axis=0)

        # pick the maximum value
        diameter = np.max(vertical_axis_diameter)

        # return it
        return float(diameter)


    def vertical_cup_to_disc_ratio(segmentation):
        """
        Compute the vertical cup-to-disc ratio from a given labelling map.
        The vertical cup to disc ratio is defined as here:
    https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1722393/pdf/v082p01118.pdf
        Input:
            segmentation: binary 2D numpy array representing a segmentation, with
    0: optic cup, 128: optic disc, 255: elsewhere.
        Output:
            cdr: vertical cup to disc ratio
        """

        # compute the cup diameter
        cup_diameter = vertical_diameter(segmentation == 0)
        # compute the disc diameter
        disc_diameter = vertical_diameter(segmentation < 255)

        return cup_diameter / (disc_diameter + EPS)


    def absolute_error(predicted, reference):
        """
        Compute the absolute error between a predicted and a reference outcomes.
        Input:
            predicted: a float value representing a predicted outcome
            reference: a float value representing the reference outcome
        Output:
            abs_err: the absolute difference between predicted and reference
        """

        return abs(predicted - reference)


    def evaluate_binary_segmentation(segmentation, gt_label):
        """
```

```
    Compute the evaluation metrics of the REFUGE challenge by comparing the
segmentation with the ground truth
    Input:
        segmentation: binary 2D numpy array representing the segmentation,
with 0: optic cup, 128: optic disc, 255: elsewhere.
        gt_label: binary 2D numpy array representing the ground truth
annotation, with the same format
    Output:
        cup_dice: Dice coefficient for the optic cup
        disc_dice: Dice coefficient for the optic disc
        cdr: absolute error between the vertical cup to disc ratio as
estimated from the segmentation vs. the gt_label, in pixels
    """

    # compute the Dice coefficient for the optic cup
    # plt.imshow(segmentation==0)
    # plt.title('cup binary_segmentation')
    # plt.show()
    # plt.imshow(gt_label==0)
    # plt.title('cup binary_gt_label')
    # plt.show()
    cup_dice = dice_coefficient(segmentation == 0, gt_label == 0)
    # compute the Dice coefficient for the optic disc
    # plt.imshow(segmentation<255)
    # plt.title('disc binary_segmentation')
    # plt.show()
    # plt.imshow(gt_label<255)
    # plt.title('disc binary_gt_label')
    # plt.show()
    disc_dice = dice_coefficient(segmentation < 255, gt_label < 255)
    # compute the absolute error between the cup to disc ratio estimated from
the segmentation vs. the gt label
    cdr = absolute_error(vertical_cup_to_disc_ratio(segmentation),
vertical_cup_to_disc_ratio(gt_label))

    return cup_dice, disc_dice, cdr


def generate_table_of_results(image_filenames, gt_folder, is_training=False):
    """
    Generates a table with image_filename, cup_dice, disc_dice and cdr values
    Input:
        image_filenames: a list of strings with the names of the images.
        segmentation_folder: a string representing the full path to the
folder where the segmentation files are
        gt_folder: a string representing the full path to the folder where
the ground truth annotation files are
        is_training: a boolean value indicating if the evaluation is
performed on training data or not
    Output:
        image_filenames: same as the input parameter
        cup_dices: a numpy array with the same length than the
image_filenames list, with the Dice coefficient for each optic cup
        disc_dices: a numpy array with the same length than the
image_filenames list, with the Dice coefficient for each optic disc
        ae_cdrs: a numpy array with the same length than the image_filenames
list, with the absolute error of the vertical cup to disc ratio
```

```
    """

    # initialize an array for the Dice coefficients of the optic cups
    cup_dices = np.zeros(len(image_filenames), dtype=np.float)
    # initialize an array for the Dice coefficients of the optic discs
    disc_dices = np.zeros(len(image_filenames), dtype=np.float)
    # initialize an array for the absolute errors of the vertical cup to disc
ratios
    ae_cdrs = np.zeros(len(image_filenames), dtype=np.float)

    # iterate for each image filename
    for i in range(len(image_filenames)):

        # read the segmentation
        segmentation = misc.imread(image_filenames[i])
        if False:
            plt.imshow(segmentation)
            plt.title('segmentation')
            plt.show()
        if len(segmentation.shape) > 2:
            segmentation = segmentation[:, :, 0]
        # read the gt
        if is_training:
            gt_filename = path.join(gt_folder, 'Glaucoma',
path.basename(image_filenames[i]))
            if path.exists(gt_filename):
                gt_label = misc.imread(gt_filename)
            else:
                gt_filename = path.join(gt_folder, 'Non-Glaucoma',
path.basename(image_filenames[i]))
                if path.exists(gt_filename):
                    gt_label = misc.imread(gt_filename)
                else:
                    raise ValueError(
                        'Unable to find {} in your training folder. Make sure
that you have the folder organized as provided in our website.'.format(
                            image_filenames[i]))
        else:
            gt_filename = path.join(gt_folder,
path.basename(image_filenames[i]))
            if path.exists(gt_filename):
                gt_label = misc.imread(gt_filename)

                if False:
                    plt.title('gt_label')
                    plt.imshow(gt_label)
                    plt.show()
            else:
                raise ValueError(
                    'Unable to find {} in your ground truth folder. If you
are using training data, make sure to use the parameter is_training in
True.'.format(
                        image_filenames[i]))

        # evaluate the results and assign to the corresponding row in the
table
        cup_dices[i], disc_dices[i], ae_cdrs[i] =
```

```
evaluate_binary_segmentation(segmentation, gt_label)
        print('cup_dices:', cup_dices[i], '---disc_dices:', disc_dices[i], '-
-ae_cdrs:', ae_cdrs[i])

    # return the colums of the table
    return image_filenames, cup_dices, disc_dices, ae_cdrs


def get_mean_values_from_table(cup_dices, disc_dices, ae_cdrs):
    """
    Compute the mean evaluation metrics for the segmentation task.
    Input:
        cup_dices: a numpy array with the same length than the
image_filenames list, with the Dice coefficient for each optic cup
        disc_dices: a numpy array with the same length than the
image_filenames list, with the Dice coefficient for each optic disc
        ae_cdrs: a numpy array with the same length than the image_filenames
list, with the absolute error of the vertical cup to disc ratio
    Output:
        mean_cup_dice: the mean Dice coefficient for the optic cups
        mean_disc_dice: the mean Dice coefficient for the optic disc
        mae_cdr: the mean absolute error for the vertical cup to disc ratio
    """

    # compute the mean values of each column
    mean_cup_dice = np.mean(cup_dices)
    mean_disc_dice = np.mean(disc_dices)
    mae_cdr = np.mean(ae_cdrs)

    return mean_cup_dice, mean_disc_dice, mae_cdr


def evaluate_segmentation_results(segmentation_folder, gt_folder,
output_path=None, export_table=False,
                                  is_training=False):
    """
    Evaluate the segmentation results of a single submission
    Input:
        segmentation_folder: full path to the segmentation files
        gt_folder: full path to the ground truth files
        [output_path]: a folder where the results will be saved. If not
provided, the results are not saved
        [export_table]: a boolean value indicating if the table will be
exported or not
        [is_training]: a boolean value indicating if the evaluation is
performed on training data or not
    Output:
        mean_cup_dice: the mean Dice coefficient for the optic cups
        mean_disc_dice: the mean Dice coefficient for the optic disc
        mae_cdr: the mean absolute error for the vertical cup to disc ratio
    """

    # get all the image filenames
    image_filenames = glob.glob(path.join(segmentation_folder, '*.bmp'))  #
get_filenames(segmentation_folder, 'bmp')
    if len(image_filenames) == 0:
        print(
```

```
            '** The segmentation folder does not include any bmp file. Check
the files extension and resubmit your results.')
        raise ValueError()
    # create output path if it does not exist
    if not (output_path is None) and not (path.exists(output_path)):
        makedirs(output_path)

    # generate a table of results
    _, cup_dices, disc_dices, ae_cdrs =
generate_table_of_results(image_filenames, gt_folder, is_training)
    # if we need to save the table
    if not (output_path is None) and (export_table):
        # initialize the table filename
        table_filename = path.join(output_path,
'evaluation_table_segmentation.csv')
        # save the table
        save_csv_segmentation_table(table_filename, image_filenames,
cup_dices, disc_dices, ae_cdrs)

    # compute the mean values
    mean_cup_dice, mean_disc_dice, mae_cdr =
get_mean_values_from_table(cup_dices, disc_dices, ae_cdrs)
    # print the results on screen
    print('Dice Optic Cup = {}\nDice Optic Disc = {}\nMAE CDR =
{}'.format(str(mean_cup_dice), str(mean_disc_dice),

str(mae_cdr)))
    # save the mean values in the output path
    if not (output_path is None):
        # initialize the output filename
        output_filename = path.join(output_path,
'evaluation_segmentation.csv')
        # save the results
        save_csv_mean_segmentation_performance(output_filename,
mean_cup_dice, mean_disc_dice, mae_cdr)

    # return the average performance
    return mean_cup_dice, mean_disc_dice, mae_cdr


if __name__ == '__main__':
    results_folder = '../../data/result_UNet120000_v22_refineNet/'
    gt_folder = '../../data/valiMaskImage_save_path_460/'
    output_path = results_folder
    export_table = True
    evaluate_segmentation_results(results_folder, gt_folder, output_path,
export_table)
```

#File-Management Utilities – file_management.py

```
import csv
import numpy as np

from scipy.io import savemat
```

```
950 from os import listdir, path, makedirs
951
952
953 def parse_boolean(input_string):
954     '''
955     Parse a string as a boolean
956     '''
957     return input_string.upper() == 'TRUE'
958
959
960 def get_filenames(path_to_files, extension):
961     '''
962     Get all the files on a given folder with the given extension
963
964     Input:
965         path_to_files: string to a path where the files are
966         [extension]: string representing the extension of the files
967     Output:
968         image_filenames: a list of strings with the filenames in the folder
969     '''
970
971     # initialize a list of image filenames
972     image_filenames = []
973     # add to this list only those filenames with the corresponding extension
974     for file in listdir(path_to_files):
975         if file.endswith('.' + extension):
976             image_filenames = image_filenames + [file]
977
978     return image_filenames
979
980
981 def read_csv_classification_results(csv_filename):
982     '''
983     Read a two-column CSV file that has the classification results inside.
984
985     Input:
986         csv_filename: full path and filename to a two column CSV file with
987 the classification results (image filename, score)
988     Output:
989         image_filenames: list of image filenames, as retrieved from the first
990 column of the CSV file
991         scores: numpy array of floats, as retrieved from the second column of
992 the CSV file
993     '''
994
995     # initialize the output variables
996     image_filenames = []
997     scores = []
998
999     # open the file
1000     with open(csv_filename, 'r') as csv_file:
1001         # initialize a reader
1002         csv_reader = csv.reader(csv_file)
1003         # ignore the first row, that only has the header
1004         next(csv_reader)
1005         # and now, iterate and fill the arrays
1006         for row in csv_reader:
```

```
            image_filenames = image_filenames + [row[0]]
            scores = scores + [float(row[1])]

    # turn the list of scores into a numpy array
    scores = np.asarray(scores, dtype=np.float)

    # return the image filenames and the scores
    return image_filenames, scores


def sort_scores_by_filename(target_names, names_to_sort, values_to_sort):
    '''
    This function is intended to correct the ordering in the outputs, just in
case...

    Input:
        target_names: a list of names sorted in the order that we want
        names_to_sort: a list of names to sort
        values_to_sort: a numpy array of values to sort
    Output:
        sorted_values: same array than values_to_sort, but this time sorted
:)
    '''

    names_to_sort = [x.upper() for x in names_to_sort]

    # initialize the array of sorted values
    sorted_values = np.zeros(values_to_sort.shape)

    # iterate for each filename in the target names
    for i in range(len(target_names)):
        # assign the value to the correct position in the array
        sorted_values[i] =
values_to_sort[names_to_sort.index(target_names[i].upper())]

    # return the sorted values
    return sorted_values


def sort_coordinates_by_filename(target_names, names_to_sort,
values_to_sort):
    '''
    This function is intended to correct the ordering in the outputs, just in
case...

    Input:
        target_names: a list of names sorted in the order that we want
        names_to_sort: a list of names to sort
        values_to_sort: a numpy array of values to sort
    Output:
        sorted_values: same array than values_to_sort, but this time sorted
:)
    '''

    # initialize the array of sorted values
    sorted_values = np.zeros(values_to_sort.shape)

```

```
    # iterate for each filename in the target names
    for i in range(len(target_names)):
        # assign the value to the correct position in the array
        sorted_values[i, :] =
values_to_sort[names_to_sort.index(target_names[i])]

    # return the sorted values
    return sorted_values


def get_labels_from_training_data(gt_folder):
    '''
    Since the training data has two folder, "Glaucoma" and "Non-Glaucoma", we
can use
    this function to generate an array of labels automatically, according to
the image
    filenames

    Input:
        gt_folder: path to the training folder, with "Glaucoma" and "Non-
Glaucoma" folder inside
    Output:
        image_filenames: filenames in the gt folders
        labels: binary labels (0: healthy, 1:glaucomatous)
    '''

    # prepare the folders to read
    glaucoma_folder = path.join(gt_folder, 'Glaucoma')
    non_glaucoma_folder = path.join(gt_folder, 'Non-Glaucoma')

    # get all the filenames inside each folder
    glaucoma_filenames = get_filenames(glaucoma_folder, 'bmp')
    non_glaucoma_filenames = get_filenames(non_glaucoma_folder, 'bmp')

    # concatenate them to generate the array of image filenames
    image_filenames = glaucoma_filenames + non_glaucoma_filenames

    # generate the array of labels
    labels = np.zeros(len(image_filenames), dtype=np.bool)
    labels[0:len(glaucoma_filenames)] = True

    return image_filenames, labels


def save_roc_curve(filename, tpr, fpr, auc):
    '''
    Save the ROC curve values on a .mat file

    Input:
        filename: output filename
        tpr: true positive rate
        fpr: false positive rate
        auc: area under the ROC curve
    '''

    # save the current ROC curve as a .mat file for MATLAB
    savemat(filename, {'tpr': tpr, 'fpr': fpr, 'auc': auc})
```

```
def save_csv_classification_performance(output_filename, auc,
reference_sensitivity):
    '''
    Save the AUC and the reference sensitivity values in a CSV file

    Input:
        output_filename: a string with the full path and the output file name
(with .csv extension)
        auc: area under the ROC curve
        reference_sensitivity: sensitivity value for a given specificity
    '''

    # open the file
    with open(output_filename, 'w') as csv_file:
        # initialize the writer
        my_writer = csv.writer(csv_file)
        # write the column names
        my_writer.writerow(['AUC', 'Sensitivity'])
        # write the values
        my_writer.writerow([str(auc), str(reference_sensitivity)])


def save_csv_fovea_location_performance(output_filename, distance):
    '''
    Save the mean Euclidean distance on a CSV file

    Input:
        output_filename: a string with the full path and the output file name
(with .csv extension)
        distance: mean Euclidean distance
    '''

    # open the file
    with open(output_filename, 'w') as csv_file:
        # initialize the writer
        my_writer = csv.writer(csv_file)
        # write the column names
        my_writer.writerow(['Mean Euclidean distance'])
        # write the values
        my_writer.writerow([str(distance)])


def save_csv_segmentation_table(table_filename, image_filenames, cup_dices,
disc_dices, ae_cdrs):
    '''
    Save the table of segmentation results as a CSV file.

    Input:
        table_filename: a string with the full path and the table filename
(with .csv extension)
        image_filenames: a list of strings with the names of the images
        cup_dices: a numpy array with the same length than the
image_filenames list, with the Dice coefficient for each optic cup
        disc_dices: a numpy array with the same length than the
image_filenames list, with the Dice coefficient for each optic disc
```

```
        ae_cdrs: a numpy array with the same length than the image_filenames
list, with the absolute error of the vertical cup to disc ratio
    '''

    # write the data
    with open(table_filename, 'w') as csv_file:
        # initialize the writer
        table_writer = csv.writer(csv_file)
        # write the column names
        table_writer.writerow(['Filename', 'Cup-Dice', 'Disc-Dice', 'AE-
CDR'])
        # write each row
        for i in range(len(image_filenames)):
            table_writer.writerow([image_filenames[i], str(cup_dices[i]),
str(disc_dices[i]), str(ae_cdrs[i])])


def save_csv_fovea_location_table(table_filename, image_filenames,
distances):
    '''
    Save the table of Euclidean distances results as a CSV file.

    Input:
        table_filename: a string with the full path and the table filename
(with .csv extension)
        image_filenames: a list of strings with the names of the images
        distances: a 1D numpy array with the Euclidean distances of the
prediction, for each image
    '''

    # write the data
    with open(table_filename, 'w') as csv_file:
        # initialize the writer
        table_writer = csv.writer(csv_file)
        # write the column names
        table_writer.writerow(['Filename', 'Euclidean distance'])
        # write each row
        for i in range(len(image_filenames)):
            table_writer.writerow([image_filenames[i], str(distances[i])])


def save_csv_mean_segmentation_performance(output_filename, mean_cup_dice,
mean_disc_dice, mae_cdrs):
    '''
    Save a CSV file with the mean performance

    Input:
        output_filename: a string with the full path and the table filename
(with .csv extension)
        mean_cup_dice: average Dice coefficient for the optic cups
        mean_disc_dice: average Dice coefficient for the optic discs
        mae_cdrs: mean absolute error of the vertical cup to disc ratios
    '''

    # write the data
    with open(output_filename, 'w') as csv_file:
        # initialize the writer
```

```
        table_writer = csv.writer(csv_file)
        # write the column names
        table_writer.writerow(['Cup-Dice', 'Disc-Dice', 'AE-CDR'])
        # write each row
        table_writer.writerow([str(mean_cup_dice), str(mean_disc_dice),
str(mae_cdrs)])


def read_fovea_location_results(csv_filename):
    '''
    Read a CSV file with 3 columns: the first contains the filenames, and the
second/third have
    the (x,y) coordinates, respectively.

    Input:
        csv_filename: full path and filename to a three columns CSV file with
the fovea location results (image filename, x, y)
    Output:
        image_filenames: list of image filenames, as retrieved from the first
column of the CSV file
        coordinates: a 2D numpy array of coordinates
    '''

    # initialize the output variables
    image_filenames = []
    coordinates = None

    # open the file
    with open(csv_filename, 'r') as csv_file:
        # initialize a reader
        csv_reader = csv.reader(csv_file)
        # ignore the first row, that only has the header
        next(csv_reader)
        # and now, iterate and fill the arrays
        for row in csv_reader:
            # append the filename
            image_filenames = image_filenames + [row[0]]
            # append the coordinates
            current_coordinates = np.asarray(row[1:], dtype=np.float)
            if coordinates is None:
                coordinates = current_coordinates
            else:
                coordinates = np.vstack((coordinates, current_coordinates))

    return image_filenames, coordinates


import openpyxl


def read_gt_fovea_location(xlsx_filename, is_training=False):
    '''
    Read a XLSX file with 3 columns: the first contains the filenames, and
the second/third have
    the (x,y) coordinates, respectively.

    Input:
```

```
        xlsx_filename: full path and filename to a three columns XLSX file
with the fovea location results (image filename, x, y)
        [is_training]: boolean indicating if we are using training data or no
    Output:
        image_filenames: list of image filenames, as retrieved from the first
column of the CSV file
        coordinates: a 2D numpy array of coordinates
    '''

    # initialize the output variables
    image_filenames = []
    coordinates = None

    # read the xlsx file
    book = openpyxl.load_workbook(xlsx_filename)
    current_sheet = book.active

    # iterate for each row
    for row in current_sheet.iter_rows(min_row=2, min_col=1):
        # append the filename
        image_filenames = image_filenames + [row[1].value]
        # append the coordinates
        if is_training:
            current_coordinates = np.asarray([float(row[2].value),
float(row[3].value)], dtype=np.float)
        else:
            current_coordinates = np.asarray([float(row[3].value),
float(row[4].value)], dtype=np.float)
        if coordinates is None:
            coordinates = current_coordinates
        else:
            coordinates = np.vstack((coordinates, current_coordinates))

    return image_filenames, coordinates


import openpyxl


def read_gt_labels(xlsx_filename):
    '''
    Read a XLSX file with 2 columns: the first contains the filenames, and
the second/third have
    the binary label for glaucoma (1) / healthy (0).

    Input:
        xlsx_filename: full path and filename to a three columns XLSX file
with the fovea location results (image filename, x, y)
    Output:
        image_filenames: list of image filenames, as retrieved from the first
column of the CSV file
        labels: a 2D numpy array of coordinates
    '''

    # initialize the output variables
    image_filenames = []
    labels = None
```

```
    # read the xlsx file
    book = openpyxl.load_workbook(xlsx_filename)
    current_sheet = book.active

    # iterate for each row
    for row in current_sheet.iter_rows(min_row=2, min_col=1):
        # append the filename
        current_name = row[0].value[:-3] + 'jpg'
        image_filenames = image_filenames + [current_name]
        # append the coordinates
        current_label = row[1].value > 0
        if labels is None:
            labels = current_label
        else:
            labels = np.vstack((labels, current_label))

    return image_filenames, labels


import zipfile


def unzip_submission(submission_file, output_folder):
    '''
    Unzip a .ZIP file with a submission to REFUGE from a team

    Input:
        submission_file: full path and filename of the .zip file
        output_folder: folder where the output will be saved
    '''

    # initialize the output folder
    if not path.exists(output_folder):
        makedirs(output_folder)

    # open the zip file
    zip_ref = zipfile.ZipFile(submission_file, 'r')
    zip_ref.extractall(output_folder)
    zip_ref.close()


def export_table_of_results(table_filename, team_names, segmentation_results,
classification_results,
                            fovea_detection_results):
    '''
    Export a table of results (unsorted) as a CSV

    Input:
        table_filename: filename of the CSV file with the table of results
        team_names: names of the teams evaluated
        segmentation_results: list of segmentation results
        classification_results: list of classification results
        fovea_detection_results: list of fovea detection results
    '''

    # write the data
```

```
    with open(table_filename, 'w') as csv_file:
        # initialize the writer
        table_writer = csv.writer(csv_file)
        # write the column names
        table_writer.writerow(
            ['Team name', 'Mean optic cup Dice', 'Mean optic disc Dice', 'MAE
cup to disc ratio', 'AUC',
             'Reference Sensitivity', 'Mean Euclidean distance'])
        # write each row
        for i in range(len(team_names)):
            # retrieve current results
            current_segmentation_results = segmentation_results[i]
            current_classification_results = classification_results[i]
            current_fovea_detection_results = fovea_detection_results[i]
            # write a row of results
            table_writer.writerow(
                [team_names[i], str(current_segmentation_results[0]),
str(current_segmentation_results[1]),
                 str(current_segmentation_results[2]),
                 str(current_classification_results[0]),
str(current_classification_results[1]),
                 str(current_fovea_detection_results)])


def export_ranking(table_filename, header, team_names, scores):
    '''
    Export the ranking

    Input:
        table_filename: filename of the CSV file with the table of results
        header: list of strings with the header for the output file
        team_names: names of the teams evaluated
        scores: a numpy array with ranking information
    '''

    scores = np.asarray(scores)

    # write the data
    with open(table_filename, 'w') as csv_file:
        # initialize the writer
        table_writer = csv.writer(csv_file)
        # write the column names
        table_writer.writerow(header)
        # write each row
        for i in range(len(team_names)):
            # write a row of results
            if len(scores.shape) > 1:
                table_writer.writerow([team_names[i]] + scores[i,
:].tolist())
            else:
                table_writer.writerow([team_names[i]] + [scores[i]])


def read_table_of_results(table_filename):
    '''
    Read the table of results (unsorted) as a CSV

```

```
        Input:
            table_filename: filename of the CSV file with the table of results
        Output:
            header: a list of strings with the name of the evaluation metrics
            teams: a list of strings with the name of the teams
            results: a numpy matrix of evaluation metrics
        '''

        # open the file
        with open(table_filename, 'r') as csv_file:
            # initialize the reader
            csv_reader = csv.reader(csv_file)
            # get the first row
            header = next(csv_reader)[1:]

            # initialize the list of teams
            teams = []
            # initialize a numpy matrix with all the other results
            results = None
            # and now, iterate and fill the arrays
            for row in csv_reader:
                # append the team name
                teams = teams + [row[0]]
                # append the results
                current_results = np.asarray(row[1:], dtype=np.float)
                if results is None:
                    results = current_results
                else:
                    results = np.vstack((results, current_results))

        return header, teams, results
```

#Models

#Discriminator Model – Discriminator.py

```
import torch.nn as nn
import torch
from models.spectral import SpectralNorm
import numpy as np
from models.self_attn import Self_Attn

class FCDiscriminator(nn.Module):

    def __init__(self, num_classes, ndf = 64):
        super(FCDiscriminator, self).__init__()

        self.conv1 = nn.Conv2d(num_classes, ndf, kernel_size=4, stride=2,
padding=1)
        self.conv2 = nn.Conv2d(ndf, ndf*2, kernel_size=4, stride=2,
padding=1)
        self.conv3 = nn.Conv2d(ndf*2, ndf*4, kernel_size=4, stride=2,
padding=1)
        self.conv4 = nn.Conv2d(ndf*4, ndf*8, kernel_size=4, stride=2,
```

```
padding=1)
        self.classifier = nn.Conv2d(ndf*8, 1, kernel_size=4, stride=2,
padding=1)

        self.leaky_relu = nn.LeakyReLU(negative_slope=0.2, inplace=True)
        #self.up_sample = nn.Upsample(scale_factor=32, mode='bilinear')
        #self.sigmoid = nn.Sigmoid()
        # self.attn1 = Self_Attn(256, 'relu')
        # self.attn2 = Self_Attn(512, 'relu')

    def forward(self, x):
        x = self.conv1(x)
        x = self.leaky_relu(x)
        x = self.conv2(x)
        x = self.leaky_relu(x)
        x = self.conv3(x)
        x = self.leaky_relu(x)
        # x, p1 = self.attn1(x)
        x = self.conv4(x)
        x = self.leaky_relu(x)
        # x, p2 = self.attn2(x)
        x = self.classifier(x)
        #x = self.up_sample(x)
        #x = self.sigmoid(x)

        return x

class Generator(nn.Module):
    """Generator."""

    def __init__(self, batch_size, image_size=64, z_dim=100, conv_dim=64):
        super(Generator, self).__init__()
        self.imsize = image_size
        layer1 = []
        layer2 = []
        layer3 = []
        last = []

        repeat_num = int(np.log2(self.imsize)) - 3
        mult = 2 ** repeat_num  # 8
        layer1.append(SpectralNorm(nn.ConvTranspose2d(z_dim, conv_dim * mult,
4)))
        layer1.append(nn.BatchNorm2d(conv_dim * mult))
        layer1.append(nn.ReLU())

        curr_dim = conv_dim * mult

        layer2.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim
/ 2), 4, 2, 1)))
        layer2.append(nn.BatchNorm2d(int(curr_dim / 2)))
        layer2.append(nn.ReLU())

        curr_dim = int(curr_dim / 2)

        layer3.append(SpectralNorm(nn.ConvTranspose2d(curr_dim, int(curr_dim
/ 2), 4, 2, 1)))
        layer3.append(nn.BatchNorm2d(int(curr_dim / 2)))
```

```
            layer3.append(nn.ReLU())

            if self.imsize == 64:
                layer4 = []
                curr_dim = int(curr_dim / 2)
                layer4.append(SpectralNorm(nn.ConvTranspose2d(curr_dim,
int(curr_dim / 2), 4, 2, 1)))
                layer4.append(nn.BatchNorm2d(int(curr_dim / 2)))
                layer4.append(nn.ReLU())
                self.l4 = nn.Sequential(*layer4)
                curr_dim = int(curr_dim / 2)

            self.l1 = nn.Sequential(*layer1)
            self.l2 = nn.Sequential(*layer2)
            self.l3 = nn.Sequential(*layer3)

            last.append(nn.ConvTranspose2d(curr_dim, 3, 4, 2, 1))
            last.append(nn.Tanh())
            self.last = nn.Sequential(*last)

            self.attn1 = Self_Attn(128, 'relu')
            self.attn2 = Self_Attn(64, 'relu')

        def forward(self, z):
            z = z.view(z.size(0), z.size(1), 1, 1)
            out = self.l1(z)
            out = self.l2(out)
            out = self.l3(out)
            out, p1 = self.attn1(out)
            out = self.l4(out)
            out, p2 = self.attn2(out)
            out = self.last(out)

            return out, p1, p2


    class Discriminator(nn.Module):
        """Discriminator, Auxiliary Classifier."""

        def __init__(self, image_size=64, conv_dim=64):
            super(Discriminator, self).__init__()
            self.imsize = image_size
            layer1 = []
            layer2 = []
            layer3 = []
            last = []

            layer1.append(SpectralNorm(nn.Conv2d(3, conv_dim, 4, 2, 1)))
            layer1.append(nn.LeakyReLU(0.1))

            curr_dim = conv_dim

            layer2.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2,
1)))
            layer2.append(nn.LeakyReLU(0.1))
            curr_dim = curr_dim * 2
```

```
        layer3.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4, 2,
1)))
        layer3.append(nn.LeakyReLU(0.1))
        curr_dim = curr_dim * 2

        if self.imsize == 64:
            layer4 = []
            layer4.append(SpectralNorm(nn.Conv2d(curr_dim, curr_dim * 2, 4,
2, 1)))
            layer4.append(nn.LeakyReLU(0.1))
            self.l4 = nn.Sequential(*layer4)
            curr_dim = curr_dim * 2
        self.l1 = nn.Sequential(*layer1)
        self.l2 = nn.Sequential(*layer2)
        self.l3 = nn.Sequential(*layer3)

        last.append(nn.Conv2d(curr_dim, 1, 4))
        self.last = nn.Sequential(*last)

        self.attn1 = Self_Attn(256, 'relu')
        self.attn2 = Self_Attn(512, 'relu')

    def forward(self, x):
        out = self.l1(x)
        out = self.l2(out)
        out = self.l3(out)
        out, p1 = self.attn1(out)
        out = self.l4(out)
        out, p2 = self.attn2(out)
        out = self.last(out)

        return out.squeeze(), p1, p2
```

#Exponential Moving Average – optim_weight_ema.py

```
class WeightEMA (object):
    def __init__(self, params, src_params, alpha=0.999):
        self.params = list(params)
        self.src_params = list(src_params)
        self.alpha = alpha

        for p, src_p in zip(self.params, self.src_params):
            p.data[:] = src_p.data[:]

    def step(self):
        one_minus_alpha = 1.0 - self.alpha
        for p, src_p in zip(self.params, self.src_params):
            p.data.mul_(self.alpha)
            p.data.add_(src_p.data * one_minus_alpha)
```

#Self-Attention Utility – self_attn.py

```
import torch.nn as nn
import torch
from models.spectral import SpectralNorm
import numpy as np
```

```
class Self_Attn(nn.Module):
    """ Self attention Layer"""

    def __init__(self, in_dim, activation):
        super(Self_Attn, self).__init__()
        self.chanel_in = in_dim
        self.activation = activation

        self.query_conv = nn.Conv2d(in_channels=in_dim, out_channels=in_dim
// 8, kernel_size=1)
        self.key_conv = nn.Conv2d(in_channels=in_dim, out_channels=in_dim //
8, kernel_size=1)
        self.value_conv = nn.Conv2d(in_channels=in_dim, out_channels=in_dim,
kernel_size=1)
        self.gamma = nn.Parameter(torch.zeros(1))

        self.softmax = nn.Softmax(dim=-1)  #

    def forward(self, x):
        """
            inputs :
                x : input feature maps( B X C X W X H)
            returns :
                out : self attention value + input feature
                attention: B X N X N (N is Width*Height)
        """
        m_batchsize, C, width, height = x.size()
        proj_query = self.query_conv(x).view(m_batchsize, -1, width *
height).permute(0, 2, 1)  # B X CX(N)
        proj_key = self.key_conv(x).view(m_batchsize, -1, width * height)  #
B X C x (*W*H)
        energy = torch.bmm(proj_query, proj_key)  # transpose check
        attention = self.softmax(energy)  # BX (N) X (N)
        proj_value = self.value_conv(x).view(m_batchsize, -1, width * height)
# B X C X N

        out = torch.bmm(proj_value, attention.permute(0, 2, 1))
        out = out.view(m_batchsize, C, width, height)

        out = self.gamma * out + x
        return out, attention
```

#Spectral Utilities – spectral.py

```
import torch
from torch.optim.optimizer import Optimizer, required

from torch.autograd import Variable
import torch.nn.functional as F
from torch import nn
from torch import Tensor
from torch.nn import Parameter

def l2normalize(v, eps=1e-12):
    return v / (v.norm() + eps)
```

```
class SpectralNorm(nn.Module):
    def __init__(self, module, name='weight', power_iterations=1):
        super(SpectralNorm, self).__init__()
        self.module = module
        self.name = name
        self.power_iterations = power_iterations
        if not self._made_params():
            self._make_params()

    def _update_u_v(self):
        u = getattr(self.module, self.name + "_u")
        v = getattr(self.module, self.name + "_v")
        w = getattr(self.module, self.name + "_bar")

        height = w.data.shape[0]
        for _ in range(self.power_iterations):
            v.data = l2normalize(torch.mv(torch.t(w.view(height,-1).data),
u.data))
            u.data = l2normalize(torch.mv(w.view(height,-1).data, v.data))

        # sigma = torch.dot(u.data, torch.mv(w.view(height,-1).data, v.data))
        sigma = u.dot(w.view(height, -1).mv(v))
        setattr(self.module, self.name, w / sigma.expand_as(w))

    def _made_params(self):
        try:
            u = getattr(self.module, self.name + "_u")
            v = getattr(self.module, self.name + "_v")
            w = getattr(self.module, self.name + "_bar")
            return True
        except AttributeError:
            return False


    def _make_params(self):
        w = getattr(self.module, self.name)

        height = w.data.shape[0]
        width = w.view(height, -1).data.shape[1]

        u = Parameter(w.data.new(height).normal_(0, 1), requires_grad=False)
        v = Parameter(w.data.new(width).normal_(0, 1), requires_grad=False)
        u.data = l2normalize(u.data)
        v.data = l2normalize(v.data)
        w_bar = Parameter(w.data)

        del self.module._parameters[self.name]

        self.module.register_parameter(self.name + "_u", u)
        self.module.register_parameter(self.name + "_v", v)
        self.module.register_parameter(self.name + "_bar", w_bar)


    def forward(self, *args):
        self._update_u_v()
        return self.module.forward(*args)
```

#U-NET Model – unet.py

```
# full assembly of the sub-parts to form the complete net

# sub-parts of the U-Net model

import torch
import torch.nn as nn
import torch.nn.functional as F


class UNet(nn.Module):
    def __init__(self, n_channels, n_classes):
        super(UNet, self).__init__()
        self.down1 = down(n_channels, 32)
        self.AdditionalInput1 = AdditionalInput(2, 3, 64)
        # self.single_conv1 = single_conv(3, 64)

        self.x1_out = outconv(32, n_classes)

        self.down2 = down(96, 64)
        self.AdditionalInput2 = AdditionalInput(4, 3, 128)
        # self.single_conv2 = single_conv(3, 128)

        self.x2_out = outconv(64, n_classes)

        self.down3 = down(192, 128)
        self.AdditionalInput3 = AdditionalInput(8, 3, 256)
        # self.single_conv3 = single_conv(3, 256)

        self.x3_out = outconv(128, n_classes)

        self.down4 = down(384, 256)

        self.x4_out = outconv(256, n_classes)

        self.down5 = down(256, 256)

        self.conv5 = double_conv(256, 512)

        self.up1 = up(512, 256)
        self.up2 = up(256, 128)
        self.up3 = up(128, 64)
        self.up4 = up(64, 32)

        self.out6 = outconv(256, n_classes)
        self.out7 = outconv(128, n_classes)
        self.out8 = outconv(64, n_classes)
        self.out9 = outconv(32, n_classes)

    def forward(self, x):
        # input1 = x   # 400x400
        input2 = self.AdditionalInput1(x)  # 200x200
        input3 = self.AdditionalInput2(x)  # 100x100
        input4 = self.AdditionalInput3(x)  # 50x50

        ###first level
        x1_conv, x1_downsample = self.down1(x)  # 32x200x200
```

```
        # x1_out = self.x1_out(x1_conv)

        ####second level
        input2 = torch.cat([input2, x1_downsample], dim=1)
        x2_conv, x2_downsample = self.down2(input2)  # 64x100x100

        # x2_out = self.x2_out(x2_conv)

        ####3rd level
        input3 = torch.cat([input3, x2_downsample], dim=1)
        x3_conv, x3_downsample = self.down3(input3)  # 128x50x50

        # x3_out = self.x3_out(x3_conv)

        ### 4th level
        input4 = torch.cat([input4, x3_downsample], dim=1)
        x4_conv, x4_downsample = self.down4(input4)  # 256x25x25

        # x4_out = self.x4_out(x4_conv)

        ### 5th level
        x5 = self.conv5(x4_downsample)  # 512x25x25

        ### -4th level
        x6 = self.up1(x5, x4_conv)  # 256x50x50
        side6 = nn.Upsample(scale_factor=8, mode='bilinear',
align_corners=True)(x6)

        ### -3th level
        x7 = self.up2(x6, x3_conv)  # 128x100x100
        side7 = nn.Upsample(scale_factor=4, mode='bilinear',
align_corners=True)(x7)

        x8 = self.up3(x7, x2_conv)
        side8 = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)(x8)

        x9 = self.up4(x8, x1_conv)

        out6 = self.out6(side6)
        out7 = self.out7(side7)
        out8 = self.out8(side8)
        out9 = self.out9(x9)

        my_list = [out6, out7, out8, out9]
        out10 = torch.mean(torch.stack(my_list), dim=0)

        return out6, out7, out8, out9, out10


class double_conv(nn.Module):
    '''(conv => BN => ReLU) * 2'''

    def __init__(self, in_ch, out_ch):
        super(double_conv, self).__init__()
        self.conv = nn.Sequential(
```

```
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        x = self.conv(x)
        return x


class single_conv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(single_conv, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        x = self.conv(x)
        return x


class AdditionalInput(nn.Module):
    def __init__(self, poolsize, in_ch, out_ch):
        super(AdditionalInput, self).__init__()
        self.AddiInput = nn.Sequential(
            nn.AvgPool2d(poolsize),
            nn.Conv2d(in_ch, out_ch, 3, padding=1)
        )

    def forward(self, x):
        x = self.AddiInput(x)
        return x


class inconv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(inconv, self).__init__()
        self.conv = double_conv(in_ch, out_ch)

    def forward(self, x):
        x = self.conv(x)
        return x


class down(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(down, self).__init__()
        self.mpconv = double_conv(in_ch, out_ch)
        self.downsample = nn.MaxPool2d(2)

    def forward(self, x):
        x_conv = self.mpconv(x)
```

```
        x_downsample = self.downsample(x_conv)
        return x_conv, x_downsample


class up(nn.Module):
    def __init__(self, in_ch, out_ch, bilinear=False):
        super(up, self).__init__()

        #  would be a nice idea if the upsampling could be learned too,
        #  but my machine do not have enough memory to handle all those
weights
        self.conv = double_conv(in_ch, out_ch)

        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)
        else:
            # self.up = nn.ConvTranspose2d(in_ch//2, in_ch//2, 2, stride=2)
            self.up = nn.ConvTranspose2d(in_ch, in_ch // 2, 2, stride=2)

    def forward(self, input_1, input_2):
        input_1 = self.up(input_1)

        # input is CHW
        diffY = input_2.size()[2] - input_1.size()[2]
        diffX = input_2.size()[3] - input_1.size()[3]

        input_1 = F.pad(input_1, (diffX // 2, diffX - diffX // 2,
                                  diffY // 2, diffY - diffY // 2))

        # for padding issues, see
        # https://github.com/HaiyongJiang/U-Net-Pytorch-Unstructured-
Buggy/commit/0e854509c2cea854e247a9c615f175f76fbb2e3a
        # https://github.com/xiaopeng-liao/Pytorch-
UNet/commit/8ebac70e633bac59fc22bb5195e513d5832fb3bd

        x = torch.cat([input_2, input_1], dim=1)
        x = self.conv(x)
        return x


class up5(nn.Module):
    def __init__(self, in_ch, out_ch, bilinear=True):
        super(up5, self).__init__()

        #  would be a nice idea if the upsampling could be learned too,
        #  but my machine do not have enough memory to handle all those
weights
        self.conv = double_conv(in_ch, out_ch)

        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)
        else:
            self.up = nn.ConvTranspose2d(in_ch // 2, in_ch // 2, 2, stride=2)

    def forward(self, x):
```

```
        x = self.up5(x)
        return x


class outconv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(outconv, self).__init__()
        self.conv = nn.Sequential(nn.Conv2d(in_ch, out_ch, 1),
                                  nn.Sigmoid()
                                  )

    def forward(self, x):
        x = self.conv(x)
        return x


def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)


if __name__ == '__main__':
    model = UNet(3, 3).cuda()
    input = torch.zeros((1, 3, 400, 400)).cuda()
   # m = nn.AvgPool2d(2).cuda()
   # input2 = m(input).cuda()
   # n = nn.Conv2d(3,256,3, padding=1).cuda()
   # input3 = n(input2)
   # print(input3.shape)
    x7, out6, _, _, _, output9 = model(input)
    hello = nn.Upsample(scale_factor=4, mode='bilinear',
align_corners=True)(x7)
    convolution = nn.Conv2d(128, 3,1).cuda()
    hello2 = convolution(hello).cuda()
    print(x7.shape)
    print(hello.shape)
    print(out6.shape)
  #  print(side7.shape)
```

#U-NET Single Scale (CFEA) – unet_SingleScale.py

```
1# full assembly of the sub-parts to form the complete net

# sub-parts of the U-Net model

import torch
import torch.nn as nn
import torch.nn.functional as F




class UNet(nn.Module):
    def __init__(self, n_channels, n_classes):

        super(UNet, self).__init__()
        self.down1 = down(n_channels, 32)
```

```
        # self.AdditionalInput1 = AdditionalInput(2, 3, 64)
        # self.single_conv1 = single_conv(3, 64)

        # self.x1_out = outconv(32, n_classes)

        self.down2 = down(32, 64)
        # self.AdditionalInput2 = AdditionalInput(4, 3, 128)
        # self.single_conv2 = single_conv(3, 128)

        # self.x2_out = outconv(64, n_classes)

        self.down3 = down(64, 128)
        # self.AdditionalInput3 = AdditionalInput(8, 3, 256)
        # self.single_conv3 = single_conv(3, 256)

        # self.x3_out = outconv(128, n_classes)

        self.down4 = down(128, 256)

        # self.x4_out = outconv(256, n_classes)

        # self.down5 = down(256, 512)

        self.conv5 = double_conv(256,512)

        self.up1 = up(512, 256)
        self.up2 = up(256, 128)
        self.up3 = up(128, 64)
        self.up4 = up(64, 32)

        self.out6 = outconv(256, n_classes)
        self.out7 = outconv(128, n_classes)
        self.out8 = outconv(64, n_classes)
        self.out9 = outconv(32, n_classes)

    def forward(self, x):


        ###first level
        x1_conv, x1_downsample = self.down1(x)  #32x200x200


        ####second level

        x2_conv, x2_downsample = self.down2(x1_downsample) #64x100x100


        ####3rd level

        x3_conv, x3_downsample = self.down3(x2_downsample)  #128x50x50


        ### 4th level

        x4_conv, x4_downsample = self.down4(x3_downsample) #256x25x25
```

```
        ### 5th level
        x5 = self.conv5(x4_downsample) #512x25x25

        ### -4th level
        x6 = self.up1(x5, x4_conv) #256x50x50
        side6 = nn.Upsample(scale_factor=8, mode='bilinear',
align_corners=True)(x6) #512x400x400

        ### -3th level
        x7 = self.up2(x6, x3_conv) #256x100x100
        side7 = nn.Upsample(scale_factor=4, mode='bilinear',
align_corners=True)(x7)

        x8 = self.up3(x7, x2_conv) #128x100x100
        side8 = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)(x8)

        x9 = self.up4(x8, x1_conv) #64x400x400

        out6 = self.out6(side6)
        out7 = self.out7(side7)
        out8 = self.out8(side8)
        out9 = self.out9(x9)

        #my_list=[out6, out7, out8, out9]
        my_list = [out6, out7, out8, out9]
        out10 = torch.mean(torch.stack(my_list), dim=0)

        return out6, out7, out8, out9, out10

class double_conv(nn.Module):
    '''(conv => BN => ReLU) * 2'''
    def __init__(self, in_ch, out_ch):
        super(double_conv, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        x = self.conv(x)
        return x


class single_conv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(single_conv, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
```

```
        x = self.conv(x)
        return x


class AdditionalInput(nn.Module):
    def __init__(self,poolsize, in_ch, out_ch):
        super(AdditionalInput, self).__init__()
        self.AddiInput =nn.Sequential(
             nn.AvgPool2d(poolsize),
             nn.Conv2d(in_ch, out_ch, 3, padding=1)
        )

    def forward(self, x):
        x = self.AddiInput(x)
        return x


class inconv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(inconv, self).__init__()
        self.conv = double_conv(in_ch, out_ch)

    def forward(self, x):
        x = self.conv(x)
        return x


class down(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(down, self).__init__()
        self.mpconv = double_conv(in_ch, out_ch)
        self.downsample = nn.MaxPool2d(2)

    def forward(self, x):
        x_conv = self.mpconv(x)
        x_downsample = self.downsample(x_conv)
        return x_conv, x_downsample


class up(nn.Module):
    def __init__(self, in_ch, out_ch, bilinear=False):
        super(up, self).__init__()

        #  would be a nice idea if the upsampling could be learned too,
        #  but my machine do not have enough memory to handle all those
weights
        self.conv = double_conv(in_ch, out_ch)

        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)
        else:
            # self.up = nn.ConvTranspose2d(in_ch//2, in_ch//2, 2, stride=2)
            self.up = nn.ConvTranspose2d(in_ch, in_ch // 2, 2, stride=2)

    def forward(self, input_1, input_2):
        input_1 = self.up(input_1)
```

```
        # input is CHW
        diffY = input_2.size()[2] - input_1.size()[2]
        diffX = input_2.size()[3] - input_1.size()[3]

        input_1 = F.pad(input_1, (diffX // 2, diffX - diffX//2,
                        diffY // 2, diffY - diffY//2))


        x = torch.cat([input_2, input_1], dim=1)
        x = self.conv(x)
        return x


class up5(nn.Module):
    def __init__(self, in_ch, out_ch, bilinear=True):
        super(up5, self).__init__()

        #  would be a nice idea if the upsampling could be learned too,
        #  but my machine do not have enough memory to handle all those
weights
        self.conv = double_conv(in_ch, out_ch)

        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode='bilinear',
align_corners=True)
        else:
            self.up = nn.ConvTranspose2d(in_ch // 2, in_ch // 2, 2, stride=2)

    def forward(self, x):
        x = self.up5(x)
        return x


class outconv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(outconv, self).__init__()
        self.conv = nn.Sequential(nn.Conv2d(in_ch, out_ch, 1),
                                  nn.Sigmoid()
                                  )

    def forward(self, x):
        x = self.conv(x)
        return x


def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)


if __name__ == '__main__':
    model = UNet(3, 3).cuda()
    input = torch.zeros((1, 3, 512, 512)).cuda()
    output = model(input)
    print(output.shape)
    print(count_parameters(model))
```

```
#Training

#Arguments – arguments.py
import argparse

BATCH_SIZE = 1
ITER_SIZE = 1
NUM_WORKERS = 0
INPUT_SIZE = '600, 600'
TGT_SIZE = '500,500'
LEARNING_RATE = 1e-4
MOMENTUM = 0.9
NUM_CLASSES = 3
NUM_STEPS = 250000
NUM_STEPS_STOP = 200000  # early stopping
POWER = 0.9
RESTORE_FROM = ''
SAVE_NUM_IMAGES = 2
SAVE_PRED_EVERY = 1000
SNAPSHOT_DIR = '/home/charlietran/CADA_Tutorial/Model_Weights/Trial1/'
TENSORBOARD_DIR =
'/home/charlietran/CADA_Tutorial/tensorboard_directory/Trial1/'
WEIGHT_DECAY = 0.0005
TEACHER_ALPHA = 0.99

LEARNING_RATE_D = 2.5e-5

class_weights=[0.4,0.4,0.2]


def get_arguments():
    """Parse all the arguments provided from the CLI.
    Returns:
      A list of parsed arguments.
    """
    parser = argparse.ArgumentParser(description="Domain Adaptation ")
    parser.add_argument("--batch-size", type=int, default=BATCH_SIZE,
                        help="Number of images sent to the network in one
step.")
    parser.add_argument("--iter-size", type=int, default=ITER_SIZE,
                        help="Accumulate gradients for ITER_SIZE
iterations.")
    parser.add_argument("--num-workers", type=int, default=NUM_WORKERS,
                        help="number of workers for multithread
dataloading.")
    parser.add_argument("--input-size", type=str, default=INPUT_SIZE,
                        help="Comma-separated string with height and width of
source images.")
    parser.add_argument("--tgt-size", type=str, default=TGT_SIZE,
                        help="Comma-separ   ated string with height and width
of target images.")
    parser.add_argument("--is-training", action="store_true",
                        help="Whether to updates the running means and
variances during the training.")
    parser.add_argument("--learning-rate", type=float, default=LEARNING_RATE,
                        help="Base learning rate for training with polynomial
```

```
decay.")
    parser.add_argument("--learning-rate-D", type=float,
default=LEARNING_RATE_D,
                        help="Base learning rate for discriminator.")
    parser.add_argument("--momentum", type=float, default=MOMENTUM,
                        help="Momentum component of the optimiser.")
    parser.add_argument("--num-classes", type=int, default=NUM_CLASSES,
                        help="Number of classes to predict (including
background).")
    parser.add_argument("--num-steps", type=int, default=NUM_STEPS,
                        help="Number of training steps.")
    parser.add_argument("--num-steps-stop", type=int, default=NUM_STEPS_STOP,
                        help="Number of training steps for early stopping.")
    parser.add_argument("--power", type=float, default=POWER,
                        help="Decay parameter to compute the learning rate.")
    parser.add_argument("--random-mirror", action="store_true",
                        help="Whether to randomly mirror the inputs during
the training.")
    parser.add_argument("--random-scale", action="store_true",
                        help="Whether to randomly scale the inputs during the
training.")
    parser.add_argument("--restore-from", type=str, default=RESTORE_FROM,
                        help="Where restore model parameters from.")
    parser.add_argument("--save-num-images", type=int,
default=SAVE_NUM_IMAGES,
                        help="How many images to save.")
    parser.add_argument("--save-pred-every", type=int,
default=SAVE_PRED_EVERY,
                        help="Save summaries and checkpoint every often.")
    parser.add_argument("--snapshot-dir", type=str, default=SNAPSHOT_DIR,
                        help="Where to save snapshots of the model.")
    parser.add_argument("--tensorboard-dir", type =str, default=
TENSORBOARD_DIR, help="WHERE TO TENSORBOARD")
    parser.add_argument("--weight-decay", type=float, default=WEIGHT_DECAY,
                        help="Regularisation parameter for L2-loss.")
    parser.add_argument("--gpu", type=int, default=0,
                        help="choose gpu device.")
    parser.add_argument("--teacher_alpha", type=float, default=TEACHER_ALPHA,
                        help="Teacher EMA alpha (decay)")
    parser.add_argument('--unsup_weight6', type=float, default= 1,
        help='unsupervised loss weight')
    parser.add_argument('--unsup_weight7', type=float, default= 1,
        help='unsupervised loss weight')
    parser.add_argument('--unsup_weight8', type=float, default= 1,
        help='unsupervised loss weight')
    parser.add_argument('--unsup_weight9', type=float, default= 1,
        help='unsupervised loss weight')
    parser.add_argument('--unsup_weight10', type=float, default=1,
                        help='unsupervised loss weight')
    parser.add_argument("--lambda-adv-tgt6", type=float, default= 1,
                        help="lambda_adv for adversarial training.")
    parser.add_argument("--lambda-adv-tgt7", type=float, default= 1,
                        help="lambda_adv for adversarial training.")
    parser.add_argument("--lambda-adv-tgt8", type=float, default= 1,
                        help="lambda_adv for adversarial training.")
    parser.add_argument("--lambda-adv-tgt9", type=float, default= 1,
                        help="lambda_adv for adversarial training.")
```

```
    parser.add_argument("--lambda-adv-tgt10", type=float, default= 1,
                        help="lambda_adv for adversarial training.")
    parser.add_argument('--mse-weight6', type=float, default=1,
                        help='mse weight for discriminative training')
    parser.add_argument('--mse-weight7', type=float, default=1,
                        help='mse weight for discriminative training')
    parser.add_argument('--mse-weight8', type=float, default=1,
                        help='mse weight for discriminative training')
    parser.add_argument('--mse-weight9', type=float, default=1,
                        help='mse weight for discriminative training')
    parser.add_argument('--mse-weight10', type=float, default=1,
                        help='mse weight for discriminative training')



    parser.add_argument("--class_weights", type=float, default=[0.4, 0.4,
0.2],
                        help="segmentation pixel-wise class weights.")

    parser.add_argument('--t', type=int, default=3, help='t for Recurrent
step of R2U_Net or R2AttU_Net')

    return parser.parse_args()
```

#Training Utilities – pytorch_utils.py

```
import torch.nn as nn
import torch
import torch.nn.functional as F
from torch.autograd import Variable

def lr_poly(base_lr, iter, max_iter, power):
    return base_lr * ((1 - float(iter) / max_iter) ** (power))


def adjust_learning_rate(optimizer, i_iter, args):
    lr = lr_poly(args.learning_rate, i_iter, args.num_steps, args.power)
    optimizer.param_groups[0]['lr'] = lr
    if len(optimizer.param_groups) > 1:
        optimizer.param_groups[1]['lr'] = lr * 10


def adjust_learning_rate_D(optimizer, i_iter, args):
    lr = lr_poly(args.learning_rate_D, i_iter, args.num_steps, args.power)
    optimizer.param_groups[0]['lr'] = lr
    if len(optimizer.param_groups) > 1:
        optimizer.param_groups[1]['lr'] = lr * 10


def calc_mse_loss(item1, item2, batch_size):
    criterion = nn.MSELoss(reduce=False)
    return criterion(item1, item2).sum() / batch_size

def calc_l1_loss(item1, item2, batch_size, gpu):
    item2 = Variable(item2.float()).cuda(gpu)
```

```
    criterion = nn.L1Loss()
    return criterion(item1, item2).sum() / batch_size


class LossMulti(nn.Module):
    def __init__(self, jaccard_weight=0, class_weights=None, num_classes=1):
        if class_weights is not None:
            self.nll_weight =
class_weights#Variable(class_weights.float()).cuda()
        else:
            self.nll_weight = None

        self.jaccard_weight = jaccard_weight
        self.num_classes = num_classes

    def __call__(self, outputs, targets):

        loss = (1 - self.jaccard_weight) * F.cross_entropy(outputs, targets,
weight=self.nll_weight)

        if self.jaccard_weight:
            eps = 1e-15
            outputs = F.softmax(outputs)
            for cls in range(self.num_classes):
                jaccard_target = (targets == cls).float()
                jaccard_output = outputs[:, cls]#.exp()
                intersection = (jaccard_output * jaccard_target).sum()

                union = jaccard_output.sum() + jaccard_target.sum()
                loss -= torch.log((intersection + eps) / (union -
intersection + eps)) * self.jaccard_weight
        return loss



def Weighted_Jaccard_loss (label, pred, class_weights=None, gpu=0):
    """
    This function returns cross entropy loss for semantic segmentation
    """
    # out shape batch_size x channels x h x w -> batch_size x channels x h x
w
    # label shape h x w x 1 x batch_size  -> batch_size x 1 x h x w
    label = Variable(label.long()).cuda(gpu)
    if class_weights is not None  and class_weights != 0:
        class_weights = torch.Tensor(class_weights)
        class_weights = Variable(class_weights).cuda(gpu)
        criterion = LossMulti(jaccard_weight=0.5,
class_weights=class_weights,num_classes=3)#.cuda(gpu)
    else:
        criterion = LossMulti(jaccard_weight=0.5, num_classes=3)   #
.cuda(gpu)
    return criterion(pred, label)



def dice_loss(true, logits, eps=1e-7):
    """Computes the Sørensen-Dice loss.
```

```
    Note that PyTorch optimizers minimize a loss. In this
    case, we would like to maximize the dice loss so we
    return the negated dice loss.
    Args:
        true: a tensor of shape [B, 1, H, W].
        logits: a tensor of shape [B, C, H, W]. Corresponds to
            the raw output or logits of the model.
        eps: added to the denominator for numerical stability.
    Returns:
        dice_loss: the Sørensen-Dice loss.
        https://github.com/kevinzakka/pytorch-goodies/blob/master/losses.py
    """
    num_classes = logits.shape[1]
    if num_classes == 1:
        true_1_hot = torch.eye(num_classes + 1)[true.squeeze(1)]
        true_1_hot = true_1_hot.permute(0, 3, 1, 2).float()
        true_1_hot_f = true_1_hot[:, 0:1, :, :]
        true_1_hot_s = true_1_hot[:, 1:2, :, :]
        true_1_hot = torch.cat([true_1_hot_s, true_1_hot_f], dim=1)
        pos_prob = torch.sigmoid(logits)
        neg_prob = 1 - pos_prob
        probas = torch.cat([pos_prob, neg_prob], dim=1)
    else:
        true_1_hot = torch.eye(num_classes)[true.squeeze(1)]
        true_1_hot = true_1_hot.permute(0, 3, 1, 2).float()
        probas = F.softmax(logits, dim=1)
    true_1_hot = true_1_hot.type(logits.type())
    dims = (0,) + tuple(range(2, true.ndimension()))
    intersection = torch.sum(probas * true_1_hot, dims)
    cardinality = torch.sum(probas + true_1_hot, dims)
    dice_loss = (2. * intersection / (cardinality + eps)).mean()
    return (1 - dice_loss)
```

#CADA Training Code – CADA.py

```
import os
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] = "3,4"
import torch
from torch.utils import data
from torch.autograd import Variable
import torch.optim as optim
import torch.backends.cudnn as cudnn
import torch.nn.functional as F
from albumentations import (
    HorizontalFlip,
    VerticalFlip,
    Compose,
    Transpose,
    RandomRotate90,
    OneOf,
    CLAHE,
    RandomGamma,
    HueSaturationValue,
    IAAAdditiveGaussianNoise, GaussNoise,
    RandomBrightnessContrast,
```

```
    IAASharpen, IAAEmboss
)

from models.unet import UNet
from models.discriminator import FCDiscriminator
from dataset.refuge import REFUGE
from pytorch_utils import (adjust_learning_rate, adjust_learning_rate_D,
                           calc_mse_loss, Weighted_Jaccard_loss, dice_loss)
from models import optim_weight_ema
from arguments import get_arguments

import tensorboard_logger as tb_logger
import numpy as np

aug_student = Compose([
    OneOf([
        Transpose(p=0.5),
        HorizontalFlip(p=0.5),
        VerticalFlip(p=0.5),
        RandomRotate90(p=0.5)], p=0.2),

    OneOf([
        IAAAdditiveGaussianNoise(p=0.5),
        GaussNoise(p=0.5),
    ], p=0.2),

    OneOf([
        CLAHE(clip_limit=2),
        IAASharpen(p=0.5),
        IAAEmboss(p=0.5),
        RandomBrightnessContrast(p=0.5),
    ], p=0.2),
    HueSaturationValue(p=0.2),
    RandomGamma(p=0.2)])



aug_teacher = Compose([

    OneOf([
        IAAAdditiveGaussianNoise(p=0.5),
        GaussNoise(p=0.5),
    ], p=0.2),

    OneOf([
        CLAHE(clip_limit=2),
        IAASharpen(p=0.5),
        IAAEmboss(p=0.5),
        RandomBrightnessContrast(p=0.5),
    ], p=0.2),
    HueSaturationValue(p=0.2),
    RandomGamma(p=0.2)])

def main():
    """Create the model and start the training."""
    args = get_arguments()

```

```
    cudnn.enabled = True
    n_discriminators = 5
    logger = tb_logger.Logger(logdir= args.tensorboard_dir, flush_secs=2)
    # create teacher & student
    student_net = UNet(3, n_classes=args.num_classes)
   # saved_state_dict = torch.load(args.restore_from)
   # print('The pretrained weights have been loaded', args.restore_from)
   # student_net.load_state_dict(saved_state_dict)
    teacher_net = UNet(3, n_classes=args.num_classes)
  #  saved_state_dict = torch.load(args.restore_from)
  #  teacher_net.load_state_dict(saved_state_dict)
    student_params = list(student_net.parameters())


    # teacher doesn't need gradient as it's just a EMA of the student
    teacher_params = list(teacher_net.parameters())
    for param in teacher_params:
        param.requires_grad = False

    student_net.train()
    student_net.cuda(args.gpu)
    teacher_net.train()
    teacher_net.cuda(args.gpu)

    cudnn.benchmark = True
    unsup_weights = [args.unsup_weight6, args.unsup_weight7,
args.unsup_weight8,
                     args.unsup_weight9, args.unsup_weight10]
    lambda_adv_tgts = [args.lambda_adv_tgt6, args.lambda_adv_tgt7,
                       args.lambda_adv_tgt8, args.lambda_adv_tgt9,
                       args.lambda_adv_tgt10]
    mse_weights =  [args.mse_weight6, args.mse_weight7, args.mse_weight8,
args.mse_weight9,
                    args.mse_weight10]

    # create a list of discriminators
    discriminators = []
    for dis_idx in range(n_discriminators):
        discriminators.append(FCDiscriminator(num_classes=args.num_classes))
        discriminators[dis_idx].train()
        discriminators[dis_idx].cuda(args.gpu)

    if not os.path.exists(args.snapshot_dir):
        os.makedirs(args.snapshot_dir)

    max_iters = args.num_steps * args.iter_size * args.batch_size
    src_set = REFUGE(True, domain='REFUGE_SRC', is_transform=True,
                     augmentations=aug_student, aug_for_target=aug_teacher,
max_iters=max_iters)
    src_loader = data.DataLoader(src_set,
                                 batch_size=args.batch_size,
                                 shuffle=True,
                                 num_workers=args.num_workers,
                                 pin_memory=True)

    src_loader_iter = enumerate(src_loader)
    tgt_set = REFUGE(True, domain='REFUGE_DST', is_transform=True,
```

```
                        augmentations=aug_student, aug_for_target=aug_teacher,
                        max_iters=max_iters)
    tgt_loader = data.DataLoader(tgt_set,
                                 batch_size=args.batch_size,
                                 shuffle=True,
                                 num_workers=args.num_workers,
                                 pin_memory=True)

    tgt_loader_iter = enumerate(tgt_loader)
    student_optimizer = optim.SGD(student_params,
                                  lr=args.learning_rate,
                                  momentum=args.momentum,
                                  weight_decay=args.weight_decay)
    teacher_optimizer = optim_weight_ema.WeightEMA(
        teacher_params, student_params, alpha=args.teacher_alpha)

    d_optimizers = []
    for idx in range(n_discriminators):
        optimizer = optim.Adam(discriminators[idx].parameters(),
                               lr=args.learning_rate_D,
                               betas=(0.9, 0.99))
        d_optimizers.append(optimizer)

    calc_bce_loss = torch.nn.BCEWithLogitsLoss()

    # labels for adversarial training
    source_label, tgt_label = 0, 1
    for i_iter in range(args.num_steps):

        total_seg_loss = 0
        seg_loss_vals = [0] * n_discriminators
        adv_tgt_loss_vals = [0] * n_discriminators
        d_loss_vals = [0] * n_discriminators
        unsup_loss_vals = [0] * n_discriminators

        for d_optimizer in d_optimizers:
            d_optimizer.zero_grad()
            adjust_learning_rate_D(d_optimizer, i_iter, args)

        student_optimizer.zero_grad()
        adjust_learning_rate(student_optimizer, i_iter, args)

        for sub_i in range(args.iter_size):

            # ******** Optimize source network with segmentation loss
********
            # As we don't change the discriminators, their parameters are
fixed
            for discriminator in discriminators:
                for param in discriminator.parameters():
                    param.requires_grad = False

            _, src_batch = src_loader_iter.__next__()
            _, _, src_images, src_labels, _ = src_batch
            src_images = Variable(src_images).cuda(args.gpu)

            # calculate the segmentation losses
```

```
            sup_preds = list(student_net(src_images))
            seg_losses, total_seg_loss = [], 0
            for idx, sup_pred in enumerate(sup_preds):
                sup_interp_pred = (sup_pred)
                # you also can use dice loss like: dice_loss(src_labels,
sup_interp_pred)
                seg_loss = Weighted_Jaccard_loss(src_labels, sup_interp_pred,
args.class_weights, args.gpu)
                seg_losses.append(seg_loss)
                total_seg_loss += seg_loss * unsup_weights[idx]
                seg_loss_vals[idx] += seg_loss.item() / args.iter_size

            _, tgt_batch = tgt_loader_iter.__next__()
            tgt_images0, tgt_lbl0, tgt_images1, tgt_lbl1, _ = tgt_batch
            tgt_images0 = Variable(tgt_images0).cuda(args.gpu)
            tgt_images1 = Variable(tgt_images1).cuda(args.gpu)

            # calculate ensemble losses
            stu_unsup_preds = list(student_net(tgt_images1))
            tea_unsup_preds = teacher_net(tgt_images0)
            total_mse_loss = 0


            for idx in range(n_discriminators):
                stu_unsup_probs = F.softmax(stu_unsup_preds[idx], dim=-1)
                tea_unsup_probs = F.softmax(tea_unsup_preds[idx], dim=-1)

                unsup_loss = calc_mse_loss(stu_unsup_probs, tea_unsup_probs,
args.batch_size)
                unsup_loss_vals[idx] += unsup_loss.item() / args.iter_size
                total_mse_loss += unsup_loss * mse_weights[idx]


            total_mse_loss = total_mse_loss / args.iter_size


            # As the requires_grad is set to False in the discriminator, the
            # gradients are only accumulated in the generator, the target
            # student network is optimized to make the outputs of target
domain
            # images close to the outputs of source domain images
            stu_unsup_preds = list(student_net(tgt_images0))
            d_outs, total_adv_loss = [], 0
            for idx in range(n_discriminators):
                stu_unsup_interp_pred = (stu_unsup_preds[idx])
                d_outs.append(discriminators[idx](stu_unsup_interp_pred))
                label_size = d_outs[idx].data.size()
                labels = torch.FloatTensor(label_size).fill_(source_label)
                labels = Variable(labels).cuda(args.gpu)
                adv_tgt_loss = calc_bce_loss(d_outs[idx], labels)
                total_adv_loss += lambda_adv_tgts[idx] * adv_tgt_loss
                adv_tgt_loss_vals[idx] += adv_tgt_loss.item() /
args.iter_size

            total_adv_loss = total_adv_loss / args.iter_size
```

```
            # requires_grad is set to True in the discriminator,  we only
            # accumulate gradients in the discriminators, the discriminators
are
            # optimized to make true predictions
            d_losses = []
            for idx in range(n_discriminators):
                discriminator = discriminators[idx]
                for param in discriminator.parameters():
                    param.requires_grad = True

                sup_preds[idx] = sup_preds[idx].detach()
                d_outs[idx] = discriminators[idx](sup_preds[idx])

                label_size = d_outs[idx].data.size()
                labels = torch.FloatTensor(label_size).fill_(source_label)
                labels = Variable(labels).cuda(args.gpu)

                d_losses.append(calc_bce_loss(d_outs[idx], labels))
                d_losses[idx] = d_losses[idx] / args.iter_size / 2
                d_losses[idx].backward()
                d_loss_vals[idx] += d_losses[idx].item()

            for idx in range(n_discriminators):
                stu_unsup_preds[idx] = stu_unsup_preds[idx].detach()
                d_outs[idx] = discriminators[idx](stu_unsup_preds[idx])

                label_size = d_outs[idx].data.size()
                labels = torch.FloatTensor(label_size).fill_(tgt_label)
                labels = Variable(labels).cuda(args.gpu)

                d_losses[idx] = calc_bce_loss(d_outs[idx], labels)
                d_losses[idx] = d_losses[idx] / args.iter_size / 2
                d_losses[idx].backward()
                d_loss_vals[idx] += d_losses[idx].item()

        for d_optimizer in d_optimizers:
            d_optimizer.step()


        total_loss = total_seg_loss + total_adv_loss + total_mse_loss

        logger.log_value('total_seg_loss', total_seg_loss, i_iter)
        logger.log_value('adv_loss', total_adv_loss, i_iter)
        logger.log_value('mse_loss', total_mse_loss, i_iter)
        logger.log_value('target_loss', total_adv_loss + total_mse_loss,
i_iter)
        logger.log_value('d_loss_0,', d_loss_vals[0], i_iter)
        logger.log_value('d_loss_1,', d_loss_vals[1], i_iter)
        logger.log_value('d_loss_2,', d_loss_vals[2], i_iter)
        logger.log_value('d_loss_3,', d_loss_vals[3], i_iter)
        logger.log_value('d_loss_4,', d_loss_vals[4], i_iter)
        logger.log_value('d_loss', np.mean(d_loss_vals[0] + d_loss_vals[1] +
d_loss_vals[2] + d_loss_vals[3] + d_loss_vals[4]), i_iter)

        total_loss.backward()
        student_optimizer.step()
        teacher_optimizer.step()
```

```
            log_str = 'iter = {0:7d}/{1:7d}'.format(i_iter, args.num_steps)
            log_str += ', total_seg_loss = {0:.3f} '.format(total_seg_loss)
            templ = 'seg_losses = [' + ', '.join(['%.2f'] * len(seg_loss_vals))
            log_str += templ % tuple(seg_loss_vals) + '] '
            templ = 'ens_losses = [' + ', '.join(['%.5f'] * len(unsup_loss_vals))
            log_str += templ % tuple(unsup_loss_vals) + '] '
            templ = 'adv_losses = [' + ', '.join(['%.2f'] *
len(adv_tgt_loss_vals))
            log_str += templ % tuple(adv_tgt_loss_vals) + '] '
            templ = 'd_losses = [' + ', '.join(['%.2f'] * len(d_loss_vals))
            log_str += templ % tuple(d_loss_vals) + '] '

            print(log_str)
            if i_iter >= args.num_steps_stop - 1:
                print('save model ...')
                filename = 'UNet' + str(args.num_steps_stop) +
'_v18_weightedclass.pth'
                torch.save(teacher_net.cpu().state_dict(),
                           os.path.join(args.snapshot_dir, filename))
                break

            if i_iter % args.save_pred_every == 0 and i_iter != 0:
                print('taking snapshot ...')
                filename = 'UNet' + str(i_iter) + '_v18_weightedclass.pth'
                torch.save(teacher_net.cpu().state_dict(),
                           os.path.join(args.snapshot_dir, filename))
                teacher_net.cuda(args.gpu)


if __name__ == '__main__':
    main()
```

#Testing Code – predict.py

```
import argparse

import numpy as np

from packaging import version

import os
os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"]="2,3"
from PIL import Image
import matplotlib.pyplot as plt
import cv2
from skimage.transform import rotate
import torch
from torch.autograd import Variable

import torch.nn as nn
from torch.utils import data

from models.unet import UNet
from dataset.refuge import REFUGE
```

```
NUM_CLASSES = 3
NUM_STEPS = 512 # Number of images in the validation set.
RESTORE_FROM =
'/home/charlietran/CADA_Tutorial/Model_Weights/Trial1/UNet1000_v18_weightedcl
ass.pth'
SAVE_PATH = '/home/charlietran/CADA_Tutorial/result/Trial1/'
MODEL = 'Unet'
BATCH_SIZE = 1
is_polar = False  #If need to transfer the image and labels to polar
coordinates: MICCAI version is False
ROI_size = 700  #ROI size
from evaluation.evaluation_segmentation import *


print(RESTORE_FROM)

palette=[
    255, 255, 255, # black background
    128, 128, 128, # index 1 is red
    0, 0, 0, # index 2 is yellow
    0, 0 , 0 # index 3 is orange
]

zero_pad = 256 * 3 - len(palette)
for i in range(zero_pad):
    palette.append(0)


def colorize_mask(mask):
    # mask: numpy array of the mask
    new_mask = Image.fromarray(mask.astype(np.uint8)).convert('P')
    new_mask.putpalette(palette)

    return new_mask

def get_arguments():
    """Parse all the arguments provided from the CLI.
    Returns:
      A list of parsed arguments.
    """
    parser = argparse.ArgumentParser(description="Unet Network")
    parser.add_argument("--model", type=str, default=MODEL,
                        help="Model Choice Unet.")
    parser.add_argument("--num-classes", type=int, default=NUM_CLASSES,
                        help="Number of classes to predict (including
background).")
    parser.add_argument("--restore-from", type=str, default=RESTORE_FROM,
                        help="Where restore model parameters from.")
    parser.add_argument("--batch-size", type=int, default=BATCH_SIZE,
                        help="Number of images sent to the network in one
step.")
    parser.add_argument("--gpu", type=int, default=0,
                        help="choose gpu device.")
    parser.add_argument("--save", type=str, default=SAVE_PATH,
                        help="Path to save result.")
    parser.add_argument("--is_polar", type=bool, default=False,
```

```
                        help="If proceed images in polar coordinate. MICCAI
version is false")
    parser.add_argument("--ROI_size", type=int, default=460,
                        help="Size of ROI.")

    parser.add_argument('--t', type=int, default=3, help='t for Recurrent
step of R2U_Net or R2AttU_Net')

    return parser.parse_args()


def main():
    """Create the model and start the evaluation process."""

    args = get_arguments()

    gpu0 = args.gpu

    if not os.path.exists(args.save):
        os.makedirs(args.save)

    model = UNet(3, n_classes=args.num_classes)

    saved_state_dict = torch.load(args.restore_from)
    model.load_state_dict(saved_state_dict)

    model.cuda(gpu0)
    model.train()

    testloader = data.DataLoader(REFUGE(False, domain='REFUGE_TEST',
is_transform=True),
                                 batch_size=args.batch_size,
shuffle=False, pin_memory=True)


    if version.parse(torch.__version__) >= version.parse('0.4.0'):
        interp = nn.Upsample(size=(ROI_size, ROI_size), mode='bilinear',
align_corners=True)
    else:
        interp = nn.Upsample(size=(ROI_size, ROI_size), mode='bilinear')

    for index, batch in enumerate(testloader):
        if index % 100 == 0:
            print('%d processd' % index)
        image, label, _, _, name = batch
        if args.model == 'Unet':
            _,_,_,_, output2  = model(Variable(image,
volatile=True).cuda(gpu0))

            output = interp(output2).cpu().data.numpy()


        for idx, one_name in enumerate(name):
            pred = output[idx]
            pred = pred.transpose(1,2,0)
            pred = np.asarray(np.argmax(pred, axis=2), dtype=np.uint8)
            output_col = colorize_mask(pred)
```

```
            print(output_col.size)
            one_name = one_name.split('/')[-1]
            output_col = output_col.convert('L')
            output_col.save('%s/%s.bmp' % (args.save, one_name))


if __name__ == '__main__':
    main()
    results_folder = SAVE_PATH
    gt_folder = '/DATA/charlie/AWC/CADA_Tutorial_Image/Target_Test/mask/'
    output_path = results_folder
    export_table = True
    evaluate_segmentation_results(results_folder, gt_folder, output_path,
export_table)
```

#YAML Packages – packages.yml

```
name: cada_tutorial
channels:
 - pytorch
 - anaconda
 - conda-forge
 - defaults
dependencies:
 - _libgcc_mutex=0.1=main
 - albumentations=0.5.2=pyhd8ed1ab_0
 - attrs=20.3.0=pyhd3deb0d_0
 - backports=1.0=py_2
 - backports.functools_lru_cache=1.6.3=pyhd8ed1ab_0
 - blas=1.0=mkl
 - bleach=3.3.0=pyh44b312d_0
 - bzip2=1.0.8=h7b6447c_0
 - ca-certificates=2020.12.5=ha878542_0
 - cairo=1.16.0=hf32fb01_1
 - certifi=2020.12.5=py36h5fab9bb_1
 - cffi=1.14.5=py36h261ae71_0
 - cloudpickle=1.6.0=py_0
 - cudatoolkit=9.0=h13b8566_0
```

```
  - cudnn=7.1.2=cuda9.0_0
  - cycler=0.10.0=py36_0
  - cytoolz=0.11.0=py36h7b6447c_0
  - dask-core=2.30.0=py_0
  - decorator=4.4.2=py_0
  - defusedxml=0.7.1=pyhd8ed1ab_0
  - entrypoints=0.3=pyhd8ed1ab_1003
  - ffmpeg=4.0=hcdf2ecd_0
  - fontconfig=2.13.1=h6c09931_0
  - freeglut=3.0.0=hf484d3e_5
  - freetype=2.10.4=h5ab3b9f_0
  - geos=3.8.1=he1b5a44_0
  - glib=2.68.0=h36276a3_0
  - graphite2=1.3.14=h23475e2_0
  - harfbuzz=1.8.8=hffaf4a1_0
  - hdf5=1.10.2=hba1933b_1
  - icu=58.2=he6710b0_3
  - imageio=2.9.0=py_0
  - imgaug=0.4.0=py_1
  - importlib-metadata=3.8.0=py36h5fab9bb_0
  - intel-openmp=2020.2=254
  - ipykernel=5.5.0=py36he448a4c_1
  - ipython=5.8.0=py36_1
  - ipython_genutils=0.2.0=py_1
  - jasper=2.0.14=h07fcdf6_1
  - jinja2=2.11.3=pyh44b312d_0
  - jpeg=9b=habf39ab_1
  - jsonschema=3.2.0=pyhd8ed1ab_3
  - jupyter_client=6.1.12=pyhd8ed1ab_0
  - jupyter_core=4.7.1=py36h5fab9bb_0
  - kiwisolver=1.2.0=py36hfd86e86_0
```

- lcms2=2.11=h396b838_0
- ld_impl_linux-64=2.33.1=h53a641e_7
- libffi=3.3=he6710b0_2
- libgcc-ng=9.1.0=hdf63c60_0
- libgfortran-ng=7.3.0=hdf63c60_0
- libglu=9.0.0=hf484d3e_1
- libopencv=3.4.2=hb342d67_1
- libopus=1.3.1=h7b6447c_0
- libpng=1.6.37=hbc83047_0
- libsodium=1.0.18=h36c2ea0_1
- libstdcxx-ng=9.1.0=hdf63c60_0
- libtiff=4.1.0=h2733197_1
- libuuid=1.0.3=h1bed415_2
- libvpx=1.7.0=h439df22_0
- libxcb=1.14=h7b6447c_0
- libxml2=2.9.10=hb55368b_3
- lz4-c=1.9.2=heb0550a_3
- markupsafe=1.1.1=py36he6145b8_2
- matplotlib-base=3.3.1=py36h817c723_0
- mistune=0.8.4=py36h1d69622_1002
- mkl=2020.2=256
- mkl-service=2.3.0=py36he8ac12f_0
- mkl_fft=1.3.0=py36h54f3939_0
- mkl_random=1.1.1=py36h0573a6f_0
- nbconvert=5.6.1=py36h9f0ad1d_1
- nbformat=5.1.2=pyhd8ed1ab_1
- ncurses=6.2=he6710b0_1
- networkx=2.5=py_0
- ninja=1.10.2=py36hff7bd54_0
- notebook=5.7.10=py36h9f0ad1d_0
- numpy-base=1.19.2=py36hfa32c7d_0

```
  - olefile=0.46=py36_0
  - opencv=3.4.2=py36h6fd60c2_1
  - openssl=1.1.1k=h27cfd23_0
  - packaging=20.9=pyh44b312d_0
  - pandoc=2.12=h7f98852_0
  - pandocfilters=1.4.2=py_1
  - pcre=8.44=he6710b0_0
  - pexpect=4.8.0=pyh9f0ad1d_2
  - pickleshare=0.7.5=py_1003
  - pillow=8.0.0=py36h9a89aac_0
  - pip=21.0.1=py36h06a4308_0
  - pixman=0.40.0=h7b6447c_0
  - prometheus_client=0.9.0=pyhd3deb0d_0
  - prompt_toolkit=1.0.15=py_1
  - ptyprocess=0.7.0=pyhd3deb0d_0
  - py-opencv=3.4.2=py36hb342d67_1
  - pycparser=2.20=py_2
  - pygments=2.8.1=pyhd8ed1ab_0
  - pyparsing=2.4.7=py_0
  - pyrsistent=0.17.3=py36h1d69622_1
  - python=3.6.13=hdb3f193_0
  - python-dateutil=2.8.1=py_0
  - python_abi=3.6=1_cp36m
  - pytorch=1.0.0=py3.6_cuda9.0.176_cudnn7.4.1_1
  - pywavelets=1.1.1=py36h7b6447c_2
  - pyzmq=19.0.2=py36h9947dbf_2
  - readline=8.1=h27cfd23_0
  - scikit-image=0.17.2=py36hdf5156a_0
  - send2trash=1.5.0=py_0
  - setuptools=52.0.0=py36h06a4308_0
  - shapely=1.7.1=py36h5d6357d_1
```

```
  - simplegeneric=0.8.1=py_1
  - six=1.15.0=py_0
  - sqlite=3.35.2=hdfb4753_0
  - terminado=0.9.3=py36h5fab9bb_0
  - testpath=0.4.4=py_0
  - tifffile=2020.10.1=py36hdd07704_2
  - tk=8.6.10=hbc83047_0
  - toolz=0.11.1=py_0
  - torchvision=0.2.1=py_2
  - tornado=6.0.4=py36h7b6447c_1
  - tqdm=4.59.0=pyhd3eb1b0_1
  - traitlets=4.3.3=py36h9f0ad1d_1
  - typing_extensions=3.7.4.3=py_0
  - wcwidth=0.2.5=pyh9f0ad1d_2
  - webencodings=0.5.1=py_1
  - wheel=0.36.2=pyhd3eb1b0_0
  - xz=5.2.5=h7b6447c_0
  - yaml=0.2.5=h7b6447c_0
  - zeromq=4.3.4=h2531618_0
  - zipp=3.4.1=pyhd8ed1ab_0
  - zlib=1.2.11=h7b6447c_3
  - zstd=1.4.4=h0b5b093_3
  - pip:
    - absl-py==0.12.0
    - astor==0.8.1
    - cached-property==1.5.2
    - chardet==4.0.0
    - config==0.5.0.post0
    - docopt==0.6.2
    - et-xmlfile==1.0.1
    - gast==0.4.0
```

```
    - google-pasta==0.2.0
    - grpcio==1.36.1
    - h5py==2.10.0
    - idna==2.10
    - keras==2.1.6
    - keras-applications==1.0.8
    - keras-preprocessing==1.1.2
    - markdown==3.3.4
    - numpy==1.19.5
    - openpyxl==3.0.7
    - pipreqs==0.4.10
    - protobuf==3.15.6
    - requests==2.25.1
    - scipy==1.2.2
    - tensorboard==1.12.2
    - tensorboard-logger==0.1.0
    - tensorflow-estimator==1.14.0
    - tensorflow-gpu==1.12.0
    - termcolor==1.1.0
    - urllib3==1.26.6
    - werkzeug==1.0.1
    - wrapt==1.12.1
    - yarg==0.1.9
prefix: /home/charlietran/anaconda3/envs/da
```