```
import tensorflow as tf

def calc_acc(logits, labels, scope_name, depth, object_class):

    tf.variable_scope(scope_name)

    logits_flat = tf.reshape(logits, [-1, 3])
    labels_flat = tf.reshape(labels, [-1, 3])
    correct_prediction=tf.equal(tf.reduce_max(logits,axis=1),tf.reduce_max(tf.multiply(labels,logits),axis=1))
    accuracy=tf.reduce_mean(tf.cast(correct_prediction,dtype=tf.float32))


    #l=tf.unstack(tf.nn.relu(tf.reshape(logits,[-1,3])))
    #for x in l :
        #y=tf.cond(tf.equal(tf.reduce_max(x),tf.constant(0,dtype=tf.float32)),lambda: tf.cast(tf.stack([0,0,0]),dtype=tf.float32),lambda: tf.cast(tf.stack(tf.floor_div(x,tf.reduce_max(x))),dtype=tf.float
        #logits_flat=tf.concat([logits_flat,y],axis=0)

    #tf.cond(tf.equal(a,a),lambda:np.mean(np.equal([list(list(sess.run(logits_flat))[i]).index(np.max(list(sess.run(logits_flat))[i]))
      #for i in range(len(list(sess.run(logits_flat))))],[list(list(sess.run(labels_flat))[i]).index(np.max(list(sess.run(labels_flat))[i])) for i in range(len(list(sess.run(label_flat))))])),lambda : np

    #acc,acc_op=tf.metrics.accuracy(labels=tf.argmax(labels, 1),predictions=tf.argmax(logits,1))
    #from tensorflow.keras.metrics import Accuracy as k_acc
    #m=k_acc(name='accuracy')  m.reset_states()m.update_state(labels, logits, sample_weight=None)  acc=m.result().numpy()

    """
    mapWeight=[]
    if (object_class=="IndustrialBuilding"):
        mapWeight=[0.1,0.8,1] #background; Industrial Buildings; Other Buildings


    acc_weighted= tf.reduce_mean(tf.multiply(accuracy,mapWeight))


    """
    return accuracy
def iou():
    probability=tf.softmax(logits)

def calc_loss(logits, labels, scope_name, depth, object_class):

    tf.variable_scope(scope_name)

                #flatten logits and labels
    logits_flat = tf.reshape(logits, [-1, depth])
    labels_flat = tf.reshape(labels, [-1, depth])
                #print ("logit "+str(logits_flat.get_shape()))
    mse=tf.losses.mean_squared_error(tf.reduce_max(labels,axis=1),tf.reduce_max(tf.multiply(labels,labels),axis=1))
    cross_entropy_loss_bg = calc_cross_ent_loss_by_class(logits_flat, labels_flat, scope_name,depth)
    iou_loss_bg = calc_iou_loss_by_class(logits_flat, labels_flat, scope_name, depth)

    mapWeight=[]
    if depth > 2:
        print("depth >2")
        mapWeight=[0.89,0.99,1] # background ; interieur ; contour
    elif depth == 2:
        print("depth = 2")
        mapWeight=[0.1,1] #background class

    cross_entropy_loss_weight= tf.reduce_mean(tf.multiply(cross_entropy_loss_bg,mapWeight))

    iou_loss_loss_weight= tf.reduce_sum(tf.multiply(iou_loss_bg,mapWeight))

    loss=cross_entropy_loss_weight
    return loss


def calc_cross_ent_loss_by_class(logits_flat, labels_flat, scope_name,depth):
    """
    if it is a binary classification, calculate sigmoid cross entropy loss given the logits and the ground-truth
    otherwise, calculate softmax cross entropy loss given the logits and iou_loss_loss_weight=tf.reduce_meanthe ground-truth

    Args:
        logits (matrix [float])      : flattened version of the unscaled output generated by the network
        labels (matrix [float])      : flattened verison of the ground-truth
        depth (int) : depth of the label layer (1 for binary classification

    Returns:
        loss (float) : scalar loss
    """
    with tf.variable_scope(scope_name):
        #if depth == 1:
        #    loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels = labels_flat, logits = logits_flat),0)
        #else:
        loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(labels = labels_flat, logits = logits_flat))

    return loss




def calc_iou_loss_by_class(logits_flat, labels_flat,scope_name, depth):
    """
    calculate intersection over union loss
    unscaled scores need to be converted to probability distribution using sigmoid or softmax depending on
    number of classes. If it a binary classification, use sigmoid. Use softmax if it is multi-class classification

    Args:
        logits (matrix [float]) : flattened version of the unscaled output generated by the network
        labels (matrix [float]) : flattened verison of the ground-truth
        depth (int) : depth of the label layer (1 for binary classification, num_of_classes for multi-label classification)

    Returns:
        loss (float) : scalar loss
    """
    #convert unscaled output generated by the network to probs
    #if depth == 1:
    #    probs_flat = tf.nn.sigmoid(logits_flat)

    #     #probs and labels for both foreground and background classes
    #     # probs_flat = tf.concat([probs_flat, tf.subtract(tf.constant(1.0), probs_flat)], axis = 1)
    #     # labels_flat = tf.concat([labels_flat, tf.subtract(tf.constant(1.0), labels_flat)], axis = 1)
    #else:
    probs_flat = tf.nn.softmax(logits_flat)

    #calculate intersection over union loss
    #calculate intersection over union loss
    with tf.variable_scope(scope_name):

        #calculate intersection of probs_flat and labels_flat (pixelwise multiplication)
        inter = tf.multiply(probs_flat, labels_flat)

        #calculate union of probs_flat and labels_flat
        union = tf.subtract(tf.add(probs_flat, labels_flat), inter)

        #sum each column of inter and union
        inter_sum = tf.reduce_sum(inter,0)
        union_sum = tf.reduce_sum(union,0)
        inter_sum += 1e-16
        union_sum += 1e-16
        loss = tf.multiply(tf.constant(-1.0), tf.log(tf.divide(inter_sum, union_sum)))

    return loss








def output_layer(inputs, depth):
    """
    convert the unscaled inputs to a probability map

    Args:
        inputs (4d tensor [float]) : unscaled inputs

    Returns:
        output (4d tensor [int]) : scaled outputs
    """
    with tf.variable_scope('output_layer'):

        #scale the unscaled inputs to 0 - 1
        probs = tf.nn.softmax(inputs)

    return probs

def calc_loss_save(logits, labels, scope_name, depth):
    """
    flatten logits and labels to 2D tensors, where dimensions are [batch_size x height x width, depth]
    calculate the cross entropy loss

    Args:
        logits (matrix [float]) : unscaled output generated by the network, dims: [batch_size, height, width, depth]
        labels (matrix [float]) : groun-truth, dims: [batch_size, height, width, depth]
        scope_name (str)        : name of the scope
        depth (int)             : depth of the classification layer

    Returns:
        loss (float) : loss
    """
    with tf.variable_scope(scope_name):

        #flatten logits and labels
        logits_flat = tf.reshape(logits, [-1, depth])
        labels_flat = tf.reshape(labels, [-1, depth])

        loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(labels = labels_flat, logits = logits_flat))
    return loss





def upsample_concat(inputs1, inputs2, num_of_channels_reduce_factor, training, scope_name):
    """
    double height and width, reduce number of channels
    concatenate inputs1 and upsampled version of inputs2

    Args:
        inputs1 (4d tensor [float])         : input that would be concatenated with inputs2
        inputs2 (4d tensor [float])         : input that would be upsampled and concatenated with inputs1
        training (1d tensor [bool])         : True = training, False = test
        num_of_channels_reduce_factor (int) : 2 = # of channels is halved
                                              4 = # of channels is divided by 4
        scope_name (str)                    : name of the upsampling layer

    Returns:
        output (4d tensor [float])  : output
    """

    num_of_filters2 = inputs2.get_shape().as_list()[1]

    with tf.variable_scope(scope_name):
        logits = tf.layers.conv2d_transpose(inputs = inputs2, filters = num_of_filters2 // num_of_channels_reduce_factor,
                                            kernel_size = 2, strides = (2, 2),
                                            data_format = 'channels_first', padding = 'same', name = 'deconv')

        logits_bn = tf.layers.batch_normalization(logits,fused=True,axis =1, training = training)
        inputs2_upsampled = tf.nn.relu(logits_bn)

        #concat along the first dimension
        output = tf.concat([inputs2_upsampled, inputs1], axis = 1)

    return output
def conv_block(inputs, filters, kernel_size, strides, training, scope_name):
    """
    2d convolution block

    Args:
        inputs (4d tensor [float]) : input 4d tensor
        filters (int)              : number of output filters
        kernel_size (int)          : size of the kernel for the convolution
        strides (int)              : strides for the convolution
        training (bool)            : True = training, False = test
        scope_name (str)           : name of the block

    Returns:
        output (4d tensor [float]) : output
    """
    with tf.variable_scope(scope_name):
        logits =tf.layers.conv2d(inputs = inputs, filters = filters, kernel_size = kernel_size, strides = strides,data_format = 'channels_first', padding = 'same', name = 'conv2d',kernel_initializer='he
        logits = tf.layers.conv2d(inputs = inputs, filters = filters, kernel_size = kernel_size, strides = strides,data_format = 'channels_first', padding = 'same', name = 'conv2d',kernel_initializer='he_

        logits_bn = tf.layers.batch_normalization(logits,fused=True,axis =1, training = training)
        output = tf.nn.relu(logits_bn)
    return output

def conv2d_softmax(inputs, filters, kernel_size, strides, conv_name):
    """
    2d convolution

    Args:
        inputs (4d tensor [float]) : input 4d tensor
        filters (int)              : number of output filters
        kernel_size (int)          : soze of the kernel for the convolution
        strides (int)              : strides for the convolution
        conv_name (str)            : name of the convolution operation
    """
    logits = tf.layers.conv2d(inputs = inputs, filters = filters, kernel_size = kernel_size, strides = strides,
                              data_format = 'channels_first', padding = 'same', name = conv_name,activation='softmax',kernel_initializer='glorot_normal', kernel_regularizer='l2')
    return logits

def conv2d(inputs, filters, kernel_size, strides, conv_name):
    """
    2d convolution

    Args:
        inputs (4d tensor [float]) : input 4d tensor
        filters (int)              : number of output filters
        kernel_size (int)          : soze of the kernel for the convolution
        strides (int)              : strides for the convolution
        conv_name (str)            : name of the convolution operation
    """
    logits = tf.layers.conv2d(inputs = inputs, filters = filters, kernel_size = kernel_size, strides = strides,data_format = 'channels_first', padding = 'same', name = conv_name, kernel_regularizer='l2')
    return logits
def conv_block_sequence(inputs, filters, num_of_conv_blocks, training, scope_name):
    """
    consecutive convolutions in the Unet model

    Args:
        inputs (4d tensor [float]) : input 4d tensor
        filters (int)              : number of output filters used in the convolutions
        num_of_conv_blocks (int)   : number of convolutional blocks in a row
        training (bool)            : True = training
                                     False = test
        scope_name (str)           : name of the sequence

    Returns:
        layer2_output (4d tensor [float]) : output
    """
    kernel_size = 3
    strides = (1, 1)

    outputs = inputs

    with tf.variable_scope(scope_name):
        #apply convolution blocks in a row
        for conv_block_no in range(1, num_of_conv_blocks+1 ):
            outputs = conv_block(outputs, filters, kernel_size, strides, training,'conv_' + str(conv_block_no))

    return outputs



def max_pool(inputs, scope_name):
    """
    Pooling operation that reduces width and height of the input layer to half

    Args:
        inputs (4d tensor [float]) : input 4d tensor
        scope_name (str)           : name of the pooling layer

    Returns:
        output (4d tensor [float]) : output
    """

    with tf.variable_scope(scope_name):
        output = tf.layers.max_pooling2d(inputs = inputs, pool_size = (2, 2), strides = (2, 2), data_format='channels_first')

    return output




import tensorflow as tf

import numpy as np
from os.path import join
import gdal
import math
from unet_model_helpers import *
from data_processor import Data_processor
import os.path
import time
import os

class Unet_model:

    """
    the model for joint or multiple learning aproach (the same model can be used for both approaches)


    Attributes:
        sess (session) : Tensorflow session
        data_processor (Data_processor) : an instance of Data_processor class
        num_of_channels (int)   : # of channels
        depth (int)             : depth of the classification layer
        patch_size (float)      : Training : size of each training patch
                                  Test     : size of the patch that would be read from the big test image
        batch_size (int)        : # of patches in a batch
        learning rate(float)    : Used only in training phase. Learning rate for the adam optimizer
        num_of_epochs (int)     : Used only in training phase. # of epochs used during optimization
        num_of_iterations (int) : Used only in training phase. # of iterations in each epoch
        decay_epoch (int)       : Used only in training phase. Parameter, determining when the learning rate would be decreased
        decay_rate (float)      : Used only in training phase. Parameter, determining how much the learning rate would be decreased
        padding (int)           : Used only in test phase. Padding for the patches
    """
```

```
def __init__(self,
                   images_dir,
                   labels_dir,
                   images_dir_val,
                   labels_dir_val,
                   images_dir_test,
                   labels_dir_test,
                   gt_folder_name,
                   patch_size,
                   padding,
                   num_of_classes,
                   mean_list,
                   batch_size,
                   learning_rate,
                   num_of_epochs,
                   num_of_iterations,
                   decay_epoch,
                   decay_rate,
                   is_training,
                   method_name,
                   object_class,
                   by_folder,
                   patch_size_val,
                   num_epoch_test_pred,
                   data_type,
                   mini_batch,
                   hors_ville_image_dir,
                   hors_ville_label_dir

                   ):

    print ("method_name "+str(method_name))


    self.data_processor = Data_processor(images_dir = images_dir,
                                         labels_dir= labels_dir,
                                         images_dir_val = images_dir_val,
                                         labels_dir_val = labels_dir_val,
                                         images_dir_test =images_dir_test,
                                         labels_dir_test= labels_dir_test,
                                         gt_folder_name = gt_folder_name,
                                         patch_size = patch_size,
                                         padding = padding,
                                         num_of_classes = num_of_classes,
                                         mean_list = mean_list,
                                         batch_size = batch_size,
                                         is_training = is_training,
                                         method_name = method_name,
                                         object_class= object_class,
                                         by_folder= by_folder,
                                         patch_size_val = patch_size_val,
                                         data_type = data_type,
                                         hors_ville_image_dir=hors_ville_image_dir,
                                         hors_ville_label_dir=hors_ville_label_dir


                                        )

    self.num_of_channels = self.data_processor.num_of_channels
    self.num_epoch_test_pred = num_epoch_test_pred
    self.data_type = data_type
    self.by_folder = by_folder

    #set parameters for the training phase
    if is_training:
        self.mini_batch=mini_batch
        self.patch_size = self.data_processor.patch_size
        self.learning_rate = learning_rate
        self.num_of_epochs = num_of_epochs
        self.num_of_iterations = num_of_iterations
        self.decay_epoch = decay_epoch
        self.decay_rate = decay_rate
        self.batch_size = batch_size
        self.depth = num_of_classes - 1
        self.object_class =object_class
        self.images_dir_val = images_dir_val
        self.labels_dir_val = labels_dir_val
        self.padding = padding
        self.patch_size_val = patch_size_val
        self.images_dir_test = images_dir_test
        self.labels_dir_test = labels_dir_test
        self.best_val_loss= math.inf
    #set parameters for the test phase
    else:
        self.patch_size = patch_size
        self.batch_size = 1
        self.padding = padding
        self.depth = num_of_classes - 1
        self.object_class =object_class



def build_model(self, input_patches, scope_name, is_training, depth, start_filter_num = 64, reuse = False):
    """
    build the Unet model described in the paper
    this function heavily uses the helper functions defined in <unet_model_helpers.py>
    it is recommended to check that python script

    Args:
        input_patches (4d tensor [float]) : inputs image patches
        scope_name (str)                  : name of the scope
        is_training (bool)                : True  : training
                                            False : test
        depth (int)                       : number of filters in the last layer
        start_filter_num (int)            : number of output filters for the first convolution. Optional (64 by default)
        reuse (bool)                      : False : initialize the variables
                                            True  : reuse the values that have already been initialized

    Returns:
        pred (4d tensor [float]) : unscaled predictions
    """
    is4layer=False
    with tf.variable_scope(scope_name, reuse = reuse):

        #contraction part
        #convolution sequence 1
```

```
            conv_seq1 = conv_block_sequence(inputs = input_patches, filters = start_filter_num, num_of_conv_blocks = 2,
                                            training = is_training, scope_name = 'seq1')
            pool1 = max_pool(conv_seq1, 'pool1')



#             #*****************************************************************************************************************************************************************************
#             if (is_training):
#                 conv_seq11=conv_seq1*tf.cast(tf.random.uniform(shape=[1,conv_seq1.shape[1],conv_seq1.shape[2],conv_seq1.shape[3]],minval=0,maxval=1)>0.0001,tf.float32)
#                 pool1 = max_pool(conv_seq11, 'pool1')
#             else:
#                 pool1 = max_pool(conv_seq1, 'pool1')



            #convolution sequence 2
            conv_seq2 = conv_block_sequence(inputs = pool1, filters = start_filter_num * 2, num_of_conv_blocks = 2,
                                            training = is_training, scope_name = 'seq2')
            pool2 = max_pool(conv_seq2, 'pool2')
# #             #*****************************************************************************************************************************************************************************
#             if (is_training):
#                 conv_seq22=conv_seq2*tf.cast(tf.random.uniform(shape=[1,conv_seq2.shape[1],conv_seq2.shape[2],conv_seq2.shape[3]],minval=0,maxval=1)>0.0001,tf.float32)
#                 pool2 = max_pool(conv_seq22, 'pool2')
#             else:
#                 pool2 = max_pool(conv_seq2, 'pool2')




            #convolution sequence 3
            conv_seq3 = conv_block_sequence(inputs = pool2, filters = start_filter_num * 4, num_of_conv_blocks = 3,
                                            training = is_training, scope_name = 'seq3')
            #*****************************************************************************************************************************************************************************
#             if (is_training):
#                 conv_seq33=conv_seq3*tf.cast(tf.random.uniform(shape=[1,conv_seq3.shape[1],conv_seq3.shape[2],conv_seq3.shape[3]],minval=0,maxval=1)>0.0001,tf.float32)
#                 pool3 = max_pool(conv_seq33, 'pool3')
#             else:
#                 pool3 = max_pool(conv_seq3, 'pool3')


            pool3 = max_pool(conv_seq3, 'pool3')


            #convolution sequence 4
            conv_seq4 = conv_block_sequence(inputs = pool3, filters = start_filter_num * 8, num_of_conv_blocks = 3,
                                            training = is_training, scope_name = 'seq4')
            pool4 = max_pool(conv_seq4, 'pool4')

            if not is4layer:
                #convolution sequence 5
                conv_seq5 = conv_block_sequence(inputs = pool4, filters = start_filter_num * 8, num_of_conv_blocks = 3,
                                            training = is_training, scope_name = 'seq5')

                #center
                pool5 = max_pool(conv_seq5, 'pool5')
                center = conv2d(inputs = pool5, filters = start_filter_num * 8, kernel_size = (3, 3), strides = (1, 1), conv_name = 'center')


                #expansion part
                #upsample - concatenation - convolution 1

                up1 = upsample_concat(inputs1 =conv_seq5, inputs2 = center, num_of_channels_reduce_factor = 2,
                                  training = is_training, scope_name = 'up1')
                up1_conv_seq = conv_block_sequence(inputs = up1, filters = conv_seq5.get_shape().as_list()[1], num_of_conv_blocks = 3,
                                               training = is_training, scope_name = 'up1_seq')

                #upsample - concatenation - convolution 2
                up2 = upsample_concat(inputs1 = conv_seq4, inputs2 = up1_conv_seq, num_of_channels_reduce_factor = 2,
                                  training = is_training, scope_name = 'up2')
                up2_conv_seq = conv_block_sequence(inputs = up2, filters = conv_seq4.get_shape().as_list()[1], num_of_conv_blocks = 3,
                                               training = is_training, scope_name = 'up2_seq')

            if  is4layer:
                center = conv2d(inputs = pool4, filters = start_filter_num * 8, kernel_size = (3, 3), strides = (1, 1), conv_name = 'center')

                #upsample - concatenation - convolution 2
                up2 = upsample_concat(inputs1 = conv_seq4, inputs2 = center, num_of_channels_reduce_factor = 2,
                                      training = is_training, scope_name = 'up2')
                up2_conv_seq = conv_block_sequence(inputs = up2, filters = conv_seq4.get_shape().as_list()[1], num_of_conv_blocks = 3,
                                                   training = is_training, scope_name = 'up2_seq')

            #upsample - concatenation - convolution 3
            up3 = upsample_concat(inputs1 = conv_seq3, inputs2 = up2_conv_seq, num_of_channels_reduce_factor = 4,
                                  training = is_training, scope_name = 'up3')
            up3_conv_seq = conv_block_sequence(inputs = up3, filters = conv_seq3.get_shape().as_list()[1], num_of_conv_blocks = 3,
                                               training = is_training, scope_name = 'up3_seq')

            #upsample - concatenation - convolution 4
            up4 = upsample_concat(inputs1 = conv_seq2, inputs2 = up3_conv_seq, num_of_channels_reduce_factor = 4,
                                  training = is_training, scope_name = 'up4')
            up4_conv_seq = conv_block_sequence(inputs = up4, filters = conv_seq2.get_shape().as_list()[1], num_of_conv_blocks = 2,
                                               training = is_training, scope_name = 'up4_seq')

            #upsample - concatenation - convolution 5
            up5 = upsample_concat(inputs1 = conv_seq1, inputs2 = up4_conv_seq, num_of_channels_reduce_factor = 4,
                                  training = is_training, scope_name = 'up5')
            up5_conv_seq = conv_block_sequence(inputs = up5, filters = conv_seq2.get_shape().as_list()[1], num_of_conv_blocks = 1,
                                               training = is_training, scope_name = 'up5_seq')

            #final convolution layer
#             if(is_training):
#                 final_conv = conv2d_softmax(up5_conv_seq, filters = depth, kernel_size = (1, 1), strides = (1, 1), conv_name = 'final_conv')
#             else:
            final_conv = conv2d(up5_conv_seq, filters = depth, kernel_size = (1, 1), strides = (1, 1), conv_name = 'final_conv')
            pred = tf.transpose(final_conv, [0, 2, 3, 1])

        return pred

    def train_model(self, snap_dir, snap_freq, log_dir, fine_tuning):
        """
        train the neural network and save weights of the trained network to the disk

        args:
            snap_dir (str)  : directory, where the trained network would be saved
            snap_freq (int) : parameter determining how often the trained model would be saved
            log_dir (str)   : directory, where the loss over the time would be saved
```

```
"""

self.sess = tf.Session()

#create an iterator for training data
if(self.mini_batch):
    training_generator = self.data_processor.mini_batch_generator(is_training = True)
else:
    training_generator = self.data_processor.batch_generator(is_training = True)


training_batch = training_generator.get_next()
training_image_batch = training_batch[0]
training_label_batch = training_batch[1]
training_image_batch = tf.transpose(training_image_batch, [0, 3, 1, 2])


#gen=np.array(self.sess.run(training_generator.get_next()))
#image= tf.transpose(gen[0], [0, 3, 1, 2])
#label=gen[1]

if (self.images_dir_val):
    validation_generator = self.data_processor.batch_generator_val()
    validation_batch = validation_generator.get_next()
    validation_image_batch = validation_batch[0]
    validation_label_batch = validation_batch[1]
    validation_image_batch = tf.transpose(validation_image_batch, [0, 3, 1, 2])


#------------------------------------------------------------------------------------------
"""
saturated=tf.transpose(tf.clip_by_value(tf.image.random_saturation(training_image_batch, 0.75, 1.25, seed=None), clip_value_min = 0, clip_value_max = 32600),[0,3,1,2])
contrasted=tf.clip_by_value(tf.image.random_contrast(training_image_batch, lower = 0.75, upper = 1.25), clip_value_min = 0, clip_value_max = 32600)
def flipud(img,l): return tf.image.flip_up_down(img),tf.image.flip_up_down(l)
def fliplr(img,l): return tf.image.flip_left_right(img),tf.image.flip_left_right(l)
def noflip(img,l): return img,l

udimg,udlabel=tf.cond((tf.random_uniform(shape=[1],minval=0,maxval=1)>0.5)[0],lambda: flipud(contrasted,training_label_batch),lambda: noflip(contrasted,training_label_batch))
image,label=tf.cond((tf.random_uniform(shape=[1],minval=0,maxval=1)>0.5)[0],lambda: fliplr(udimg,udlabel),lambda: noflip(udimg,udlabel))
"""

def contrasted(): return tf.clip_by_value(tf.image.random_contrast(training_image_batch, lower = 0.75, upper = 1.25), clip_value_min = 0, clip_value_max = 32600)
def saturated(img): return tf.transpose(tf.clip_by_value(tf.image.random_saturation(img, 0.9, 1.1, seed=None), clip_value_min = 0, clip_value_max = 32600),[0,3,1,2])
def same(img): return img
image=tf.cond((tf.random_uniform(shape=[1],minval=0,maxval=1)>0.5)[0],lambda: contrasted(),lambda: same(training_image_batch))
#image=tf.cond((tf.random_uniform(shape=[1],minval=0,maxval=1)>0.5)[0],lambda: saturated(image1),lambda: same(image1))

training_pred = self.build_model(input_patches = image, scope_name = 'model', is_training = True, depth = self.depth)
training_loss = calc_loss(training_pred, training_label_batch, 'training_loss', self.depth, self.object_class)
horsvilles_loss = calc_loss(training_pred[-2], training_label_batch[-2], 'horsvilles_loss', self.depth, self.object_class)
acc=calc_acc(training_pred, training_label_batch, 'training_acc', self.depth, self.object_class)
horsvilles_acc = calc_acc(tf.reshape(training_pred[-1], [-1, 3]),tf.reshape(training_label_batch[-1], [-1, 3]), 'horsvilles_acc', self.depth, self.object_class)

training_loss +=tf.contrib.layers.apply_regularization(tf.contrib.layers.l2_regularizer(scale=0.0001), tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES))

varUPDATE_OPS = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.control_dependencies(varUPDATE_OPS):
    training_step = tf.train.AdamOptimizer(self.learning_rate).minimize(training_loss)

#------------------------------------------------------------------------------------------

#initialize all the variables
self.sess.run(tf.global_variables_initializer())

if (self.images_dir_val):
    validation_pred = self.build_model(input_patches = validation_image_batch, scope_name = 'model', is_training = False, depth = self.depth, reuse = True)
    validation_loss = calc_loss(validation_pred, validation_label_batch, 'validation_loss', self.depth, self.object_class)
    validation_acc = calc_acc(validation_pred, validation_label_batch, 'validation_acc', self.depth, self.object_class)


#if fine_tuning mode is on, the pretrained model is restored
if fine_tuning:
    curr_epoch = 1
    if os.path.isdir(snap_dir):
        curr_epoch=self.find_model_epoch(snap_dir) + 1
        curr_epoch_finetuning=self.find_model_epoch(snap_dir) + 1
    new_model_saver = tf.train.Saver(max_to_keep = (self.num_of_epochs - curr_epoch + 1) // snap_freq + 1,name = 'new_model_saver')
    if os.path.isdir(snap_dir):
        self.restore_model(new_model_saver,snap_dir)
    else:
        self.restore_model_fromFile(new_model_saver,snap_dir)
#if fine_tuning mode is off, the model is trained from scratch
else:
    curr_epoch = 1
    curr_epoch_finetuning = 0
    #saver for the new model
    new_model_saver = tf.train.Saver(max_to_keep = (self.num_of_epochs - curr_epoch + 1) // snap_freq + 1,name = 'new_model_saver')



tf.summary.scalar('loss_train', training_loss)
tf.summary.scalar('acc_train',  acc)
NoBuilding_acc=tf.summary.scalar('acc_NoBuilding_train',horsvilles_acc)
NoBuilding_train=tf.summary.scalar('loss_NoBuilding_train',horsvilles_loss)
if (self.images_dir_val):
    tf.summary.scalar('loss_validation', validation_loss)
    tf.summary.scalar('acc_validation', validation_acc)
merged = tf.summary.merge_all()
#save the graph under <log_dir>
train_writer = tf.summary.FileWriter(log_dir,self.sess.graph)



#save initial state of the model before starting the training
new_model_saver.save(self.sess, snap_dir + 'model', global_step = (curr_epoch - 1), write_meta_graph = False)



#each training epoch
while curr_epoch <= int(self.num_epoch_test_pred+curr_epoch_finetuning):

    if curr_epoch == self.decay_epoch:
        self.learning_rate = self.learning_rate * self.decay_rate
    print ("learning rate "+str(self.learning_rate))

    #each iteration in an epoch
```

```
        for curr_iter in range(1, self.num_of_iterations ):


            time_start = time.time()
            #self.sess.run([training_step0,training_step1])

            if (self.images_dir_val):
                if(((curr_iter - 1) % 100) != 0):
                    training_accuracy,validation_accuracy, training_lossd,validation_lossd,_,NoBuild,NoBuilding_accuracy = self.sess.run([acc, validation_acc,training_loss, validation_loss,training_s
                    train_writer.add_summary(NoBuilding_accuracy, (curr_epoch - 1) * self.num_of_iterations + curr_iter)
                    train_writer.add_summary(NoBuild, (curr_epoch - 1) * self.num_of_iterations + curr_iter)
                else:
                    training_accuracy,validation_accuracy, training_lossd,validation_lossd,_,summary,NoBuild,NoBuilding_accuracy = self.sess.run([acc, validation_acc,training_loss, validation_loss,tr
                    train_writer.add_summary(NoBuilding_accuracy, (curr_epoch - 1) * self.num_of_iterations + curr_iter)
                    train_writer.add_summary(summary, (curr_epoch - 1) * self.num_of_iterations + curr_iter)
                    train_writer.add_summary(NoBuild, (curr_epoch - 1) * self.num_of_iterations + curr_iter)

                print (" training_loss_summary "+str(training_lossd))
                print (" validation_loss_summary "+str(validation_lossd))
                print (" training_accuracy "+str(training_accuracy))
                print (" validation_accuracy "+str(validation_accuracy))
                train_writer.flush()


            else :
                acc,training_loss_summary, training_lossd,_ = self.sess.run([accuracy,training_loss_summary_op,training_loss,training_step])
                print (" training_loss_summary "+str(training_lossd))
                print (" training_accuracy "+str(training_accuracy))

            elapsed_time = time.time() - time_start
            print('epoch: %d / %d, iter : %d / %d, elapsed_time : %.4f secs' % (curr_epoch, self.num_of_epochs, curr_iter, self.num_of_iterations, elapsed_time))

        if curr_epoch % snap_freq == 0:
                new_model_saver.save(self.sess, snap_dir + 'model', global_step = curr_epoch, write_meta_graph = False)

        curr_epoch += 1
    self.sess.close()


def find_model_epoch(self, snap_dir):
    """
    find out for how many epochs the previous model has been trained
    Note: when calculating the number of epochs, only the model indicated by the latest checkpoint is considered
    the others are ignored

    Args:
        snap_dir (str)    : directory, where parameters for the trained network are located

    Returns:
        model_epoch (int) : number of epochs have been used in the training phase for the model that would be restored
    """

    f = open(join(snap_dir, 'checkpoint'),'r')
    lines = f.readlines()
    model_id = lines[0].split('"')

    model_epoch = np.int(model_id[1].split('-')[-1])
    f.close()

    return model_epoch

def restore_model(self, saver, snap_dir):
    """
    restore parameters of the pretrained network using the last checkpoint in the snapshot directory

    Args:
        saver (tf saver) : tensorflow saver
        snap_dir (str)   : directory, where the trained network is located
    """

    latest_check_point = tf.train.latest_checkpoint(snap_dir)
    saver.restore(self.sess, latest_check_point)

def restore_model2(self, snap_dir):
    """
    restore parameters of the pretrained network using the last checkpoint in the snapshot directory

    Args:
        saver (tf saver) : tensorflow saver
        snap_dir (str)   : directory, where the trained network is located
    """
    saver = tf.train.Saver(var_list = tf.global_variables())
    print (str(snap_dir))
    latest_check_point = tf.train.latest_checkpoint(snap_dir)
    print ("latest check point", str(latest_check_point))
    saver.restore(self.sess, latest_check_point)

def restore_model_fromFile2(self, snap_file):
    """
    restore parameters of the pretrained network using the last checkpoint in the snapshot file

    Args:
        snap_dir (str) : directory, where the trained network is located
    """

    saver = tf.train.Saver(var_list = tf.global_variables())
    print ("snap_file="+str(snap_file))
    saver.restore(self.sess, snap_file)


def restore_model_fromFile(self, saver,snap_file):
    """snap_dir
    restore parameters of the pretrained network using the last checkpoint in the snapshot directory

    Args:
        snap_dir (str) : directory, where the trained network is located
    """
    print ("snap_file="+str(snap_file))
    saver.restore(self.sess, snap_file)

def classify(self, snap_dir):
    """
    load the learned parameters to classify each test image
    since the test images might be big, perform classification patch by patch
    in order to get rid of border effects, pad each patch
```

```
        Args:
            snap_dir (str) : directory, where parameters for the trained network are located
                             the learned parameters are loaded from the latest checkpoint under this directory
        """
        self.sess = tf.Session()


        #create an iterator
        generator = self.data_processor.test_patch_generator()

        #get a patch, its top-left x and y coordinate location in the actual image
        #and its actual height and width
        next_element = generator.get_next()
        patch = next_element[0]
        y_top_left_tensor = next_element[1]
        x_top_left_tensor = next_element[2]
        patch_height_tensor = next_element[3]
        patch_width_tensor = next_element[4]




        pred = self.build_model(input_patches = patch, is_training = False, depth = self.depth, scope_name = 'model')


        #get variables of the model
        model_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope = 'model')
        #saver for the model
        model_saver = tf.train.Saver({v.op.name: v for v in model_vars})

        #restore the model
        import os.path

        if os.path.isdir(snap_dir):
            self.restore_model2(snap_dir)
        else:
            self.restore_model_fromFile2(snap_dir)

        probs = output_layer(pred, self.depth)

        while True:
            try:

                #get the patch produced by the generator
                patch_probs, y_top_left, x_top_left, patch_height, patch_width = self.sess.run([probs,
                                                                                                y_top_left_tensor,
                                                                                                x_top_left_tensor,
                                                                                                patch_height_tensor,
                                                                                                patch_width_tensor])

                #threshold the probability map at 0.5 to convert it to a classification map
                #patch_pred = np.array(patch_probs >= 0.5).astype(np.uint8)
                #print (str("patch_probs ")+str(patch_probs))
                pred_for_actual_patch = patch_probs[0,
                                                   self.padding:(self.padding + patch_height),
                                                   self.padding:(self.padding + patch_width), :]

                pred_for_actual_patchcpy=pred_for_actual_patch.copy()

                pred_for_actual_patch= np.argmax(pred_for_actual_patch, axis = -1).astype(np.uint8)
                print ("shape "+str(pred_for_actual_patch.shape))
                print ("start "+str(int(x_top_left))+" "+str(int(y_top_left)))
                #write the patch to a file
                #for i in range(self.depth):

                sizex=pred_for_actual_patch.shape[0]
                sizey=pred_for_actual_patch.shape[1]
                if(sizex+int(x_top_left)>self.data_processor.geo_image.RasterXSize):
                    sizex=self.data_processor.geo_image.RasterXSize-int(x_top_left)
                if(sizey+int(y_top_left)>self.data_processor.geo_image.RasterYSize):
                    sizey=self.data_processor.geo_image.RasterYSize-int(y_top_left)

                maskNoData=np.zeros((sizex,sizey))
                print ("so shape "+str(sizex)+" "+str(sizey))
                for c in range(1,int(self.num_of_channels)+1):

                    maskNoDatatmp = self.data_processor.geo_image.GetRasterBand(c).ReadAsArray(int(x_top_left), int(y_top_left), sizex,sizey)
                    maskNoData+=maskNoDatatmp

                ones = np.ones(pred_for_actual_patch.shape, dtype=np.uint8)
                ones[:maskNoData.shape[0], :maskNoData.shape[1]]=maskNoData

                pred_for_actual_patch=pred_for_actual_patch*np.where(ones> 0, 1, 0).astype(np.uint8)
                self.data_processor.geo_label_map.GetRasterBand(1).WriteArray(pred_for_actual_patch, int(x_top_left), int(y_top_left))
                #print("shape",str(pred_for_actual_patchcpy.shape))
#                self.data_processor.geo_label_proba.GetRasterBand(1).WriteArray(pred_for_actual_patchcpy[:,:,0], int(x_top_left), int(y_top_left))
#                self.data_processor.geo_label_proba.GetRasterBand(2).WriteArray(pred_for_actual_patchcpy[:,:,1], int(x_top_left), int(y_top_left))


            #when all the patches are read, exit
            except tf.errors.OutOfRangeError:
                break
        self.sess.close()

    def classify_test(self, snap_dir, epoch):
        """
        load the learned parameters to classify each test image during the training
        since the test images might be big, perform classification patch by patch
        in order to get rid of border effects, pad each patch

        Args:
            snap_dir (str) : directory, where parameters for the trained network are located
                             the learned parameters are loaded from the latest checkpoint under this directory
        """

        self.sess = tf.Session()
        #create an iterator
        generator = self.data_processor.test_during_training_patch_generator(epoch)

        #get a patch, its top-left x and y coordinate location in the actual image
        #and its actual height and width
        next_element = generator.get_next()
        patch = next_element[0]
        y_top_left_tensor = next_element[1]
        x_top_left_tensor = next_element[2]
        patch_height_tensor = next_element[3]
        patch_width_tensor = next_element[4]

        pred = self.build_model(input_patches = patch, is_training = False, depth = self.depth, scope_name = 'model')

        #get variables of the model
        model_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES, scope = 'model')

        #saver for the model
        model_saver = tf.train.Saver({v.op.name: v for v in model_vars})


        if os.path.isdir(snap_dir):

            self.restore_model2(snap_dir)
        else:
            self.restore_model_fromFile2(snap_dir)

        probs = output_layer(pred, self.depth)

        while True:

            try:

                #get the patch produced by the generator
                patch_probs, y_top_left, x_top_left, patch_height, patch_width = self.sess.run([probs,
                                                                                                y_top_left_tensor,
                                                                                                x_top_left_tensor,
                                                                                                patch_height_tensor,
                                                                                                patch_width_tensor])


                #threshold the probability map at 0.5 to convert it to a classification map
                #patch_pred = np.array(patch_probs >= 0.5).astype(np.uint8)
                #print (str("patch_probs ")+str(patch_probs))
                pred_for_actual_patch = patch_probs[0, self.padding:(patch_height+self.padding), self.padding:(patch_width+self.padding), :]



                pred_for_actual_patch= np.argmax(pred_for_actual_patch, axis = -1).astype(np.uint8)
                print ("shape "+str(pred_for_actual_patch.shape))
                print ("start "+str(int(x_top_left))+" "+str(int(y_top_left)))
                #write the patch to a file
                #for i in range(self.depth):

                sizex=pred_for_actual_patch.shape[0]
                sizey=pred_for_actual_patch.shape[1]
                if(sizex+int(x_top_left)>self.data_processor.geo_image.RasterXSize):
                    sizex=self.data_processor.geo_image.RasterXSize-int(x_top_left)
                if(sizey+int(y_top_left)>self.data_processor.geo_image.RasterYSize):
                    sizey=self.data_processor.geo_image.RasterYSize-int(y_top_left)

                maskNoData=np.zeros((sizex,sizey))
                print ("so shape "+str(sizex)+" "+str(sizey))
                for c in range(1,int(self.num_of_channels)+1):

                    maskNoDatatmp = self.data_processor.geo_image.GetRasterBand(c).ReadAsArray(int(x_top_left), int(y_top_left), sizex,sizey)
                    maskNoData+=maskNoDatatmp

                ones = np.ones(pred_for_actual_patch.shape, dtype=np.uint8)
                ones[:maskNoData.shape[0], :maskNoData.shape[1]]=maskNoData

                pred_for_actual_patch=pred_for_actual_patch*np.where(ones> 0, 1, 0).astype(np.uint8)
                self.data_processor.geo_label_map.GetRasterBand(1).WriteArray(pred_for_actual_patch, int(x_top_left), int(y_top_left))

            #when all the patches are read, exit
            except tf.errors.OutOfRangeError:
                break
        self.sess.close()

import tensorflow as tf

import gdal
import os

from os import listdir, rename, remove
from os.path import join, isdir, dirname, abspath

import functools
import numpy as np
import random
import math
import cv2
import time

from subprocess import call



def adjust_gamma(image, data_type, gamma=1.0):
    if (data_type=="8bits"):
        table = np.array([((i / 255.0) ** invGamma) * 255
            for i in np.arange(0, 256)]).astype("uint8")

    elif (data_type=="16bits"):
        table = np.array([((i / 10000.0) ** invGamma) * 10000
            for i in np.arange(0, 10000)]).astype(np.uint16)

    return cv2.LUT(image, table)

def adjust_alpha_beta(image, data_type, alpha=1.0,beta=1.0):

    if (data_type=="8bits"):
        table = np.array([min(max(0,((i *alpha)+beta)),255)
            for i in np.arange(0, 256)]).astype("uint8")

    elif (data_type=="16bits"):
        table = np.array([min(max(0,((i *alpha)+beta)),10000)
            for i in np.arange(0, 10000)]).astype(np.uint16)

    # apply gamma correction using the lookup table
    return cv2.LUT(image, table)

def add_backgound(image, data_type):
    table = np.array([i+1 for i in np.arange(0, 256)]).astype("uint8")
    return cv2.LUT(image, table)
```

```
def random_etal(image, data_type):

    if (data_type=="8bits"):
        minr=np.random.uniform(0,10)
        maxr=np.random.uniform(245,255)
        table = np.array([min(255,max(0,(i-minr)/float(maxr)*255.0))
            for i in np.arange(0, 256)]).astype("uint8")

    elif (data_type=="16bits"):
        minr=np.random.uniform(0,10)
        maxr=np.random.uniform(9990,10000)
        table = np.array([min(10000,max(0,(i-minr)/float(maxr)*10000.0))
            for i in np.arange(0, 10000)]).astype(np.uint16)

    return cv2.LUT(image, table)

def delete_contour(image, data_type):
    if (data_type=="8bits"):
        table = np.array([min(1,i) for i in np.arange(0, 256)]).astype("uint8")
    elif (data_type=="16bits"):
        table = np.array([min(1,i) for i in np.arange(0, 10000)]).astype(np.uint16)

    return cv2.LUT(image, table)





class Data_processor:
    """
    Data_processor class handles data related operations such as
    retrieving a batch of patches, augmenting patches, etc.


    Attributes:
        is_training (bool) : True : Training
                             False: Test
        mean_list (list [float]) : mean value of each channel. Mean values are subtracted from all the pixels
        patch_size (float)       : Training : size of each training patch
                                   Test     : size of the patch that would be read from the big test image
        num_of_channels (int)    : # of channels
        num_of_classes           : # of classes including the background class. For instance, if the classes are building and road
                                   this parameter has to be 3. Additional 1 is for the background
        padding (int)            : padding for the patches
        batch_size (int)         : # of patches in a batch
        image_paths (list[list[list[list(str)]]]) : full paths of the images
        label_paths (list[list[list[list(str)]]]) : full paths of the label maps -> training
                                                                      predicted maps -> test
        geo_image (geo object)   : Used only in test phase. Geo object, which points the current test image
        geo_label_map (geo object) : Used only in test phase. Geo object, which points the predicted map for the current test image
    """

    def __init__(self,
                 images_dir,
                 labels_dir,
                 images_dir_val,
                 labels_dir_val,
                 images_dir_test,
                 labels_dir_test,
                 gt_folder_name,
                 patch_size,
                 padding,
                 num_of_classes,
                 mean_list,
                 batch_size,
                 is_training,
                 method_name,
                 object_class,
                 by_folder,
                 patch_size_val,
                 data_type,
                 hors_ville_image_dir,
                 hors_ville_label_dir
              ):
        print (str(method_name))

        self.is_training = is_training
        self.mean_list = mean_list
        self.method_name = method_name
        self.by_folder = by_folder
        self.object_class = object_class
        self.num = 0
        self.data_type = data_type
        self.images_dir_val =images_dir_val
        self.labels_dir_val =labels_dir_val

        if self.is_training:
            self.images_dir_test =images_dir_test
            self.labels_dir_test =labels_dir_test
            self.patch_size = patch_size
            self.patch_size_val = patch_size_val

            #self.image_paths, self.label_paths = self.create_image_label_paths(db_main_dir, 'train', gt_folder_name = gt_folder_name)

            if(self.by_folder):
                self.image_paths, self.label_paths = self.create_image_label_paths_by_folder(images_dir, labels_dir)
                self.hors_ville_image_paths,self.hors_ville_label_paths=self.create_image_label_paths_by_folder(hors_ville_image_dir, hors_ville_label_dir)


                if (self.images_dir_test and self.labels_dir_test):
                    self.image_paths_test, self.label_paths_test = self.create_image_label_paths_OLD(images_dir_test,labels_dir_test, 'test',method_name = self.method_name)

            else:
                self.image_paths, self.label_paths = self.create_image_label_paths_OLD(images_dir,labels_dir,'train',  gt_folder_name = gt_folder_name)

        else:
            print (str("test"))
            self.patch_size = patch_size
            self.image_paths, self.label_paths = self.create_image_label_paths_OLD(images_dir,labels_dir, 'test',method_name = method_name)


        self.num_of_channels = self.find_num_of_channels()

        self.num_of_classes = num_of_classes
        self.padding = padding
        self.batch_size = batch_size
        print(str(self.num_of_channels )+" "+str(self.num_of_classes) +" "+str(self.patch_size)   )

    def mini_batch_generator(self, is_training):
        """
        create a generator, which retrieves a batch of image patches and their corresponding label maps

        Returns:
            iterator (tensorflow iterator object): iterator, which generates batches
        """

        #shapes of the outputs that generator produces
        output_shapes = (tf.TensorShape([self.patch_size, self.patch_size, self.num_of_channels]),
                         tf.TensorShape([self.patch_size, self.patch_size]))
        #data types of the outputs that generator produces
        if (self.data_type =="8bits"):
            data_types = (tf.uint8, tf.uint8)
        elif (self.data_type =="16bits"):
            data_types = (tf.int32,tf.int32)  #np.uint16,np.uint16
        #create a dataset object
        if(self.by_folder):
            generator_by_folder = functools.partial(self.patch_generator_by_folder_train_new, is_training = is_training)
            dataset = tf.data.Dataset.from_generator(generator_by_folder, output_types = (tf.int32,tf.int32),output_shapes = output_shapes)
        else :
            dataset = tf.data.Dataset.from_generator(self.patch_generator_OLD,output_types = (tf.int32,tf.int32),output_shapes = output_shapes)

        dataset = dataset.map(lambda image, label_map: self.process_training_patches(image, label_map, is_training), num_parallel_calls = self.batch_size)
        dataset = dataset.batch(self.batch_size)
        dataset = dataset.prefetch(1)
        iterator = dataset.make_one_shot_iterator()
        return iterator

    def patch_generator_by_folder_train_new(self,is_training):

        local_image_paths = self.image_paths
        image_paths_horsvilles = self.hors_ville_image_paths
        nb_iter=functools.reduce(lambda x,y:x+y,[len(local_image_paths[i]) for i in range(len(local_image_paths))])
        remain_index=[]
        remain_index_horsvilles=[]
        cnt=1

        mean=int(np.mean([len(local_image_paths[i]) for i in range (len(local_image_paths))]))
        for i in range (int(len(local_image_paths))):
            length=len(local_image_paths[i])
            remain_index.append([i,list(np.arange(length))])
            if(mean>length):
                for j in range(mean-length):
                    (remain_index[i][1]).append(np.random.randint(length))

        for i in range (int(len(image_paths_horsvilles))):
            remain_index_horsvilles.append([i,list(np.arange(len(image_paths_horsvilles[i])))])



        while True:
            if ((cnt%6)!=0):
                cnt+=1
                city=np.random.randint(0, len(remain_index))
                city_index=(remain_index[city])[0]
                img= np.random.randint(0, len((remain_index[city])[1]))
                image_index=((remain_index[city])[1])[img]
                ((remain_index[city])[1]).pop(img)
                if (len((remain_index[city])[1])==0):
                    remain_index.pop(city)
                if (len(remain_index)==0):
                    for i in range (int(len(local_image_paths))):
                        length=len(local_image_paths[i])
                        remain_index.append([i,list(np.arange(length))])
                        if(mean>length):
                            for j in range(mean-length):
                                (remain_index[i][1]).append(np.random.randint(length))

                image_patch, label_patch  = self.read_training_patch_by_folder_new(city_index, image_index, is_training,False)
            else:
                cnt=1
                locus=np.random.randint(0, len(remain_index_horsvilles))
                locus_index=(remain_index_horsvilles[locus])[0]
                position= np.random.randint(0, len((remain_index_horsvilles[locus])[1]))
                position_index=((remain_index_horsvilles[locus])[1])[position]
                ((remain_index_horsvilles[locus])[1]).pop(position)
                if (len((remain_index_horsvilles[locus])[1])==0):
                    remain_index_horsvilles.pop(locus)
                if (len(remain_index_horsvilles)==0):
                    for i in range (int(len(image_paths_horsvilles))):
                        remain_index_horsvilles.append([i,list(np.arange(len(image_paths_horsvilles[i])))])
                image_patch, label_patch  = self.read_training_patch_by_folder_new(locus_index, position_index, is_training,True)
            label_patch=add_backgound(label_patch, self.data_type)
            yield image_patch, label_patch

    def read_training_patch_by_folder_new(self, folder_index, image_index, is_training,hors_ville):
        """
        read an image patch and its label map

        Args:
            image_path (str) : full path of the image patch
            label_path (str) : full path of the label map

        Returns:
            image (tensor) : image patch
            label (tensor) : label patch
        """
        if (is_training and not(hors_ville)) :
            image_paths = self.image_paths
            label_paths = self.label_paths
        elif (is_training and hors_ville) :
            image_paths = self.hors_ville_image_paths
            label_paths = self.hors_ville_label_paths

        else :
            image_paths = self.image_paths_test
            label_paths = self.label_paths_test

        #read an image patch
        #convert chw to hwc
        geo_image = gdal.Open(image_paths[folder_index][image_index])
        image = np.transpose(geo_image.ReadAsArray(), [1, 2, 0])
        image=image[0:self.patch_size,0:self.patch_size,0:self.patch_size]
        image=np.where(image<0, 0, image)
        geo_label = gdal.Open(label_paths[folder_index][image_index])
        label = geo_label.ReadAsArray()
        label=np.where(label<0 , 0, label)
        label = label.astype(np.uint8)
        label=label[0:self.patch_size,0:self.patch_size]
        return image, label

    def read_training_patch_by_folder(self, folder_index, image_index, is_training):
        """
        read an image patch and its label map

        Args:
            image_path (str) : full path of the image patch
            label_path (str) : full path of the label map

        Returns:
            image (tensor) : image patch
            label (tensor) : label patch
        """
        if (is_training) :
            image_paths = self.image_paths
            label_paths = self.label_paths


        else :
            image_paths = self.image_paths_test
            label_paths = self.label_paths_test

        #read an image patch
        #convert chw to hwc
        geo_image = gdal.Open(image_paths[folder_index][image_index])
        image = np.transpose(geo_image.ReadAsArray(), [1, 2, 0])
        image=image[0:self.patch_size,0:self.patch_size,0:self.patch_size]

        image=np.where(image<0, 0, image)
        geo_label = gdal.Open(label_paths[folder_index][image_index])
        label = geo_label.ReadAsArray()
        label=np.where(label<0 , 0, label)
        label = label.astype(np.uint8)
        label=label[0:self.patch_size,0:self.patch_size]
        return image, label




    def batch_generator(self, is_training):
        """
        create a generator, which retrieves a batch of image patches and their corresponding label maps

        Returns:
            iterator (tensorflow iterator object): iterator, which generates batches
        """

        #shapes of the outputs that generator produces
        output_shapes = (tf.TensorShape([self.patch_size, self.patch_size, self.num_of_channels]),
                         tf.TensorShape([self.patch_size, self.patch_size]))

        #data types of the outputs that generator produces
        if (self.data_type =="8bits"):
            data_types = (tf.uint8, tf.uint8)


        elif (self.data_type =="16bits"):
            data_types = (tf.int32,tf.int32)  #np.uint16,np.uint16


        #create a dataset object
        if(self.by_folder):
            generator_by_folder = functools.partial(self.patch_generator_by_folder_train, is_training = is_training)
            dataset = tf.data.Dataset.from_generator(generator_by_folder, output_types = (tf.int32,tf.int32),output_shapes = output_shapes)

        else :
            dataset = tf.data.Dataset.from_generator(self.patch_generator_OLD,output_types = (tf.int32,tf.int32),output_shapes = output_shapes)



        #augment the data in parallel

        dataset = dataset.map(lambda image, label_map: self.process_training_patches(image, label_map, is_training), num_parallel_calls = self.batch_size)


        #get a batch
        dataset = dataset.batch(self.batch_size)

        #prefetch is used to increase the training speed
        #while the data in Nth iteration is being processed, the data for (N + 1)th iteration is getting prepared
        dataset = dataset.prefetch(1)

        iterator = dataset.make_one_shot_iterator()

        return iterator

    def batch_generator_val(self):
        """
        create a generator, which retrieves a batch of image patches and their corresponding label maps

        Returns:
            iterator (tensorflow iterator object): iterator, which generates batches
        """

        #shapes of the outputs that generator produces
        output_shapes = (tf.TensorShape([self.patch_size, self.patch_size, self.num_of_channels]),
                         tf.TensorShape([self.patch_size, self.patch_size]))

        #data types of the outputs that generator produces
        if (self.data_type =="8bits"):
            data_types = (tf.uint8, tf.uint8)

        elif (self.data_type =="16bits"):
            data_types = (tf.int32,tf.int32)


        #create a dataset object
        if(self.by_folder):
            self.image_paths_val, self.label_paths_val = self.create_image_label_paths_by_folder(self.images_dir_val,self.labels_dir_val)

            generator_by_folder = functools.partial(self.patch_generator_by_folder_val, is_training = False)
            dataset = tf.data.Dataset.from_generator(generator_by_folder,
                                                     output_types = data_types,
                                                     output_shapes = output_shapes)
        else :
            dataset = tf.data.Dataset.from_generator(self.patch_generator_OLD,
                                                     output_types = data_types,
                                                     output_shapes = output_shapes)
        #augment the data in parallel
        dataset = dataset.map(lambda image, label_map: self.process_training_patches(image, label_map, False), num_parallel_calls = self.batch_size)

        #get a batch
        dataset = dataset.batch(self.batch_size)

        #prefetch is used to increase the training speed
        #while the data in Nth iteration is being processed, the data for (N + 1)th iteration is getting prepared
        dataset = dataset.prefetch(1)

        iterator = dataset.make_one_shot_iterator()

        return iterator

    def batch_generator_val_epoch(self):
        """
        create a generator, which retrieves a batch of image patches and their corresponding label maps

        Returns:
            iterator (tensorflow iterator object): iterator, which generates batches
        """

        #create a dataset object
        if(self.by_folder):
             #shapes of the outputs that generator produces
            output_shapes = (tf.TensorShape([self.patch_size, self.patch_size, self.num_of_channels]),
                             tf.TensorShape([self.patch_size, self.patch_size]),
                             tf.TensorShape(None),
                             tf.TensorShape(None)
                             )

            #data types of the outputs that generator produces
            if (self.data_type =="8bits"):
                data_types = (tf.uint8, tf.uint8, tf.string, tf.string)

            elif (self.data_type =="16bits"):
                data_types = (tf.int32, tf.int32, tf.string, tf.string)

            generator_by_folder = functools.partial(self.patch_generator_by_folder_val_epoch)
            dataset = tf.data.Dataset.from_generator(generator_by_folder,
                                                     output_types = data_types,
                                                     output_shapes = output_shapes)
            #augment the data in parallel
            dataset = dataset.map(lambda image, label_map, city, name: self.process_training_patches_epoch(image, label_map, city, name, False), num_parallel_calls = 1)

        else :
            output_shapes = (tf.TensorShape([self.patch_size, self.patch_size, self.num_of_channels]),
                             tf.TensorShape([self.patch_size, self.patch_size])
                             )

            #data types of the outputs that generator produces
            if (self.data_type =="8bits"):
                data_types = (tf.uint8, tf.uint8)

            elif (self.data_type =="16bits"):
                data_types = (tf.int32, tf.int32)

            dataset = tf.data.Dataset.from_generator(self.patch_generator_OLD,
                                                     output_types = data_types,
                                                     output_shapes = output_shapes)

            dataset = dataset.map(lambda image, label_map: self.process_training_patches_epoch(image, label_map, False), num_parallel_calls = 1)


        #get a batch
        dataset = dataset.batch(1)

        #prefetch is used to increase the training speed
        #while the data in Nth iteration is being processed, the data for (N + 1)th iteration is getting prepared
        dataset = dataset.prefetch(1)

        iterator = dataset.make_one_shot_iterator()

        return iterator


    def patch_generator_fun(self):
        """
        generator function that yields an image patch and a label map
        the patch is sampled with the following algorithm

        1 - select a random continent
        2 - select a random country from the chosen continent
        3 - select a random city from the chosen country
        4 - select a random patch from the chosen city

        Yields:
            image (matrix): an image patch: [patch_size,
                                             patch_size,
                                             # of channels]
            label (matrix): label map     : [patch_size,
                                             patch_size]
        """

        local_image_paths = self.image_paths
        local_label_paths = self.label_paths

        while True:
            continent_index = random.randint(0, int(len(local_image_paths)) - 1)
            country_index = random.randint(0, int(len(local_image_paths[continent_index])) - 1)
            city_index = random.randint(0, int(len(local_image_paths[continent_index][country_index])) - 1)
            image_index = random.randint(0, int(len(local_image_paths[continent_index][country_index][city_index])) - 1)
            image, label = self.read_training_patch(local_image_paths[continent_index][country_index][city_index][image_index],
                                                    local_label_paths[continent_index][country_index][city_index][image_index])

            yield image, label
```

```
def test_patch_generator(self):
    """
    create a generator, which retrieves a patch from the big test image

    Returns:
        iterator (tensorflow iterator object): iterator
    """
    #shapes of the outputs that generator produces
    output_shapes = (tf.TensorShape([1,
                                     self.num_of_channels,
                                     self.patch_size + 2 * self.padding,
                                     self.patch_size + 2 * self.padding]),
                     tf.TensorShape([]),
                     tf.TensorShape([]),
                     tf.TensorShape([]),
                     tf.TensorShape([]))

    #data types of the outputs that generator produces
    data_types = (tf.float32, tf.int64, tf.int64, tf.int64, tf.int64)

    #create a dataset object
    dataset = tf.data.Dataset.from_generator(self.test_patch_generator_fun,
                                             output_types = data_types,
                                             output_shapes = output_shapes)

    iterator = dataset.make_one_shot_iterator()

    return iterator

def test_during_training_patch_generator(self, epoch):
    """
    create a generator, which retrieves a patch from the big val image

    Returns:
        iterator (tensorflow iterator object): iterator
    """
    #shapes of the outputs that generator produces
    self.num = epoch
    print("num of channels ", self.num_of_channels)
    print("padding", self.padding)
    print("patch_size", self.patch_size)

    output_shapes = (tf.TensorShape([1,
                                     self.num_of_channels,
                                     self.patch_size + 2 * self.padding,
                                     self.patch_size + 2 * self.padding]),
                     tf.TensorShape([]),
                     tf.TensorShape([]),
                     tf.TensorShape([]),
                     tf.TensorShape([]))

    #data types of the outputs that generator produces
    data_types = (tf.float32, tf.int64, tf.int64, tf.int64, tf.int64)

    #create a dataset object

    dataset = tf.data.Dataset.from_generator(self.test_during_training_patch_generator_fun,
                                             output_types = data_types,
                                             output_shapes = output_shapes)

    iterator = dataset.make_one_shot_iterator()

    return iterator


def test_patch_generator_funSaved(self):
    """
    generator function that yields patches from the test images
    the function also yields top-left x and y coordinate location of the patches in big the images
    and their actual size (height and width of rightmost and bottommost patches migh be lower than <self.patch_size>)

    assume that there is a big tif file consisting of 20 patches.
    this function yields the patches in this order:
     0 -  1 -  2 -  3 -  4
     5 -  6 -  7 -  8 -  9
    10 - 11 - 12 - 13 - 14
    15 - 16 - 17 - 18 - 19

    Yields:
        patch_4d (matrix): a normalized image patch:  [1,
                                                       # of channels,
                                                       patch_size,
                                                       patch_size]
        y_top_left (int) : y coordinate of top-left location of the patch in the image
        x_top_left (int) : x coordinate of top-left location of the patch in the image
        actual_patch_height (int) : height of the patch
        actual_patch_width (int)  : width of the patch
    """
    print ("test_patch_generator_fun "+str(continent_image_path )+" "+str(continent_pred_path ) )
    #iterate over each continent
    for continent_image_path, continent_pred_path in zip(self.image_paths, self.label_paths):

        #iterate over each country
        for country_image_path, country_pred_path in zip(continent_image_path, continent_pred_path):

            #iterate over each city
            for city_image_path, city_pred_path in zip(country_image_path, country_pred_path):

                #iterate over each image
                for image_path, pred_path in zip(city_image_path, city_pred_path):
                    print ("Read img "+str(image_path) +"pred_path "+str(pred_path))
                    image_info = image_path.split('/')

                    image_name = image_info[-1].split('.')[0]
                    city_name = image_info[-3]
                    country_name = image_info[-4]
                    continent_name = image_info[-5]
                    print ("Read img "+str(image_path))
                    #create a tif file for the predicted map
                    pred_path=pred_path.split(".tif")[0]+"_"+self.method_name+".tif"
                    self.open_test_image_label(image_path, pred_path)
                    #height and width of the image
                    orig_img_h = self.geo_image.RasterYSize
                    orig_img_w = self.geo_image.RasterXSize

                    #number of patches horizontally and vertically
                    n_patch_horiz = int(math.ceil(orig_img_w / self.patch_size))
                    n_patch_vert = int(math.ceil(orig_img_h / self.patch_size))

                    total_num_of_patches = n_patch_horiz * n_patch_vert

                    #iterate over each patch in the big image
                    for i in range(n_patch_vert):
                        for j in range(n_patch_horiz):

                            #top - left location of the patch
                            y_top_left = i * self.patch_size
                            x_top_left = j * self.patch_size

                            #actual height and width of each patch
                            #sicv2.imwrite("label_"+struuid+"_DataAugmentation.png",label_patch)ze of rightmost and bottommost patches might be lower than <self.patch_size>
                            actual_patch_height = min(self.patch_size, (orig_img_h - y_top_left))
                            actual_patch_width = min(self.patch_size, (orig_img_w - x_top_left))

                            #read a patch
                            patch = self.read_test_patch(x_top_left, y_top_left, orig_img_w, orig_img_h).astype(np.float32)

                            #normalize the patch
                            #print ("normalize data")
                            patch_normalized = self.normalize_data(patch)

                            #convert from hwc to chw
                            patch_normalized = np.transpose(patch_normalized, [2, 0, 1])

                            #convert <patch_normalized> to 4d matrix
                            patch_4d = np.expand_dims(patch_normalized, axis = 0)

                            time_start = time.time()

                            #generate a patch as well as its location and actual dimensions
                            #location and dimensions are needed to determine where to put the predicted label map
                            yield patch_4d, y_top_left, x_top_left, actual_patch_height, actual_patch_width
                            time_elapsed = time.time() - time_start

                            print('%s -> %s -> %s -> %s, patch %d / %d has been classified, elapsed time: %.4f secs' %
                                  (continent_name, country_name, city_name, image_name,
                                  i * n_patch_horiz + j + 1, total_num_of_patches,
                                  time_elapsed))

                    #close the current image and its predicted map
                    self.close_test_image_label()

                    #compress the predicted label map to save space
                    self.compress_label_map(pred_path)


def test_patch_generator_fun(self):
    """
    generator function that yields patches from the test images
    the function also yields top-left x and y coordinate location of the patches in big the images
    and their actual size (height and width of rightmost and bottommost patches migh be lower than <self.patch_size>)

    assume that there is a big tif file consisting of 20 patches.
    this function yields the patches in this order:
     0 -  1 -  2 -  3 -  4
     5 -  6 -  7 -  8 -  9
    10 - 11 - 12 - 13 - 14
    15 - 16 - 17 - 18 - 19

    Yields:
        patch_4d (matrix): a normalized image patch:  [1,
                                                       # of channels,
                                                       patch_size,
                                                       patch_size]
        y_top_left (int) : y coordinate of top-left location of the patch in the image
        x_top_left (int) : x coordinate of top-left location of the patch in the image
        actual_patch_height (int) : height of the patch
        actual_patch_width (int)  : width of the patch
    """


    #iterate over each city
    print ("self.image_paths"+str(self.image_paths))
    print ("self.label_paths"+str(self.label_paths))
    for image_path , label_path in zip(self.image_paths,self.label_paths) :

        print ("Read img "+str(image_path))
        print ("Read Path "+str(label_path))
        print ("Read Path "+str(label_path.split(".tif")[0]))
        print ("Read Path "+str(self.method_name))
        label_path=str(label_path)
        #create a tif file for the predicted map
        label_path=str(label_path.split(".tif")[0])+"_"+str(self.method_name)+".tif"
        print ("Export path prediction "+str(label_path))
        self.open_test_image_label(image_path,label_path)

        #height and width of the image
        orig_img_h = self.geo_image.RasterYSize
        orig_img_w = self.geo_image.RasterXSize

        #number of patches horizontally and vertically
        n_patch_horiz = int(math.ceil(orig_img_w / self.patch_size))
        n_patch_vert = int(math.ceil(orig_img_h / self.patch_size))

        total_num_of_patches = n_patch_horiz * n_patch_vert

        #iterate over each patch in the big image
        for i in range(n_patch_vert):
            for j in range(n_patch_horiz):

                #top - left location of the patch
                y_top_left = i * self.patch_size
                x_top_left = j * self.patch_size

                #actual height and width of each patch
                #size of rightmost and bottommost patches might be lower than <self.patch_size>
                actual_patch_height = min(self.patch_size, (orig_img_h - y_top_left))
                actual_patch_height = min(self.patch_size, (orig_img_h - y_top_left))
                actual_patch_width = min(self.patch_size, (orig_img_w - x_top_left))

                #read a patch
                patch = self.read_test_patch(x_top_left, y_top_left, orig_img_w, orig_img_h).astype(np.float32)
                print ("min patch"+str(patch[..., 0].min()), "max patch",str(patch[..., 0].max()))
                #normalize the patch
                #print ("normalize data")
                patch_normalized = self.normalize_data_val(patch)
                print ("min patch norm "+str(patch_normalized[..., 0].min()), "max patch norm",str(patch_normalized[..., 0].max()))
                #convert from hwc to chw
                patch_normalized = np.transpose(patch_normalized, [2, 0, 1])

                #convert <patch_normalized> to 4d matrix
                patch_4d = np.expand_dims(patch_normalized, axis = 0)

                time_start = time.time()

                #generate a patch as well as its location and actual dimensions
                #location and dimensions are needed to determine where to put the predicted label map
                yield patch_4d, y_top_left, x_top_left, actual_patch_height, actual_patch_width
                time_elapsed = time.time() - time_start

                print('patch %d / %d has been classified, elapsed time: %.4f secs' %
                      (
                      i * n_patch_horiz + j + 1, total_num_of_patches,
                      time_elapsed))

        #close the current image and its predicted map
        self.close_test_image_label()

        #compress the predicted label map to save space
        self.compress_label_map(image_path+"pred.tif")

def test_during_training_patch_generator_fun(self):
    """
    generator function that yields patches from the validation images
    the function also yields top-left x and y coordinate location of the patches in big the images
    and their actual size (height and width of rightmost and bottommost patches migh be lower than <self.patch_size>)

    Yields:
        patch_4d (matrix): a normalized image patch:  [1,
                                                       # of channels,
                                                       patch_size,
                                                       patch_size]
        y_top_left (int) : y coordinate of top-left location of the patch in the image
        x_top_left (int) : x coordinate of top-left location of the patch in the image
        actual_patch_height (int) : height of the patch
        actual_patch_width (int)  : width of the patch

    """

    print("Prediction on the test images")
    #self.image_paths_val, self.label_paths_val = self.create_image_label_paths_OLD(self.images_dir_val,self.labels_dir_val, 'test',method_name = 'pred')

    #iterate over each city
    for image_path , label_path in zip(self.image_paths_test,self.label_paths_test) :

        print ("Read img "+str(image_path))
        print ("Read Path "+str(label_path))
        print ("Read Path "+str(label_path.split(".tif")[0]))
        self.method_name = 'pred'


        print ("Read Path "+str(self.method_name))
        label_path=str(label_path)
        #create a tif file for the predicted map
        label_path=str(label_path.split(".tif")[0])+"_"+str(self.method_name)+"_epoch_"+str(self.num)+".tif"

        print ("Export path prediction "+str(label_path))
        self.open_test_image_label(image_path,label_path)


        #height and width of the image
        orig_img_h = self.geo_image.RasterYSize
        orig_img_w = self.geo_image.RasterXSize

        #number of patches horizontally and vertically
        n_patch_horiz = int(math.ceil(orig_img_w / self.patch_size))
        n_patch_vert = int(math.ceil(orig_img_h / self.patch_size))

        total_num_of_patches = n_patch_horiz * n_patch_vert

        #iterate over each patch in the big image
        for i in range(n_patch_vert):   #n_patch_vert
            for j in range(n_patch_horiz): #n_patch_horiz


                #top - left location of the patch
                y_top_left = i * self.patch_size
                x_top_left = j * self.patch_size

                #actual height and width of each patch
                #size of rightmost and bottommost patches might be lower than <self.patch_size>
                actual_patch_height = min(self.patch_size, (orig_img_h - y_top_left))
                actual_patch_width = min(self.patch_size, (orig_img_w - x_top_left))

                #read a patch
                patch = self.read_test_patch(x_top_left, y_top_left, orig_img_w, orig_img_h).astype(np.float32)

                #normalize the patch
                #print ("normalize data")
                patch_normalized = self.normalize_data_val(patch)

                #convert from hwc to chw
                patch_normalized = np.transpose(patch_normalized, [2, 0, 1])

                #convert <patch_normalized> to 4d matrix
                patch_4d = np.expand_dims(patch_normalized, axis = 0)

                time_start = time.time()
                #generate a patch as well as its location and actual dimensions
                #location and dimensions are needed to determine where to put the predicted label map
                yield patch_4d, y_top_left, x_top_left, actual_patch_height, actual_patch_width


                time_elapsed = time.time() - time_start
                time_elapsed = time.time() - time_start

                print('patch %d / %d has been classified, elapsed time: %.4f secs' %
                      (
                      i * n_patch_horiz + j + 1, total_num_of_patches,
                      time_elapsed))

        #close the current image and its predicted map
        self.close_test_image_label()

        #compress the predicted label map to save space
        self.compress_label_map(image_path+"pred.tif")

def read_training_patch(self, image_path, label_path):
    """
    read an image patch and its label map

    Args:
        image_path (str) : full path of the image patch
        label_path (str) : full path of the label map

    Returns:
        image (tensor) : image patch
        label (tensor) : label patch
    """


    #read an image patch
    #convert chw to hwc
    geo_image = gdal.Open(image_path)
    image = np.transpose(geo_image.ReadAsArray(), [1, 2, 0])
    image=image[0:self.patch_size,0:self.patch_size,0:self.patch_size]

    #read a label map
    geo_label = gdal.Open(label_path)
    label = geo_label.ReadAsArray().astype(np.uint8)
    label=label[0:self.patch_size,0:self.patch_size]


    return image, label




def read_val_patch_by_folder(self, folder_index, image_index):
    """
    read an image patch and its label map

    Args:
        image_path (str) : full path of the image patch
        label_path (str) : full path of the label map

    Returns:
        image (tensor) : image patch
        label (tensor) : label patch
    """

    image_paths = self.image_paths_val
    label_paths = self.label_paths_val

    #read an image patch
    #convert chw to hwc
    geo_image = gdal.Open(image_paths[folder_index][image_index])
    image = np.transpose(geo_image.ReadAsArray(), [1, 2, 0])
    image=image[0:self.patch_size,0:self.patch_size,0:self.patch_size]


    image=np.where(image<0, 0, image)
    #read a label map
    geo_label = gdal.Open(label_paths[folder_index][image_index])

    label = geo_label.ReadAsArray().astype(np.uint8)

    label=label[0:self.patch_size,0:self.patch_size]

    return image, label

def read_val_patch_by_folder_epoch(self, image_path, label_path):
    """
    read an image patch and its label map

    Args:
        image_path (str) : full path of the image patch
        label_path (str) : full path of the label map

    Returns:
        image (tensor) : image patch
        label (tensor) : label patch
    """

    #read an image patch
    #convert chw to hwc
    geo_image = gdal.Open(str(image_path))
    image = np.transpose(geo_image.ReadAsArray(), [1, 2, 0])
    image=image[0:self.patch_size,0:self.patch_size,0:self.patch_size]

    #read a label map
    geo_label = gdal.Open(str(label_path))

    label = geo_label.ReadAsArray().astype(np.uint8)

    label=label[0:self.patch_size,0:self.patch_size]

    return image, label


def process_training_patches(self, image, label, is_training):
    """
    - normalize images
    - one hot encode label maps
    - augment data

    Args:
        image (matrix) : image [patch_size, patch_size, # of channels]
        label (matrix) : label [patch_size, patch_size]

    Returns:
    Returns:
        image (tensor) : modified image [patch_size, patch_size, # of channels]
        label (tensor) : modified label [patch_size, patch_size, 1]
    """

    #outputs of any tf.py_func have no shape
    #reshape image and label.


    image = tf.reshape(image, [self.patch_size, self.patch_size, self.num_of_channels])
    label = tf.reshape(label, [self.patch_size, self.patch_size])


    #if there are more than one class, one hot encode the label map
    if self.num_of_classes > 1:
        label = tf.one_hot(label, depth = self.num_of_classes)

        #ignore the first class, we assume that the first class is background
        label = label[:, :, 1:]

    #if there is only one class in the ground-truth, the label map is already one hot encoded
    #just expand dimension
    else:
        label = tf.expand_dims(label, -1)

    #cast both input image and label map to float32
    image = tf.cast(image, tf.float32)
    label = tf.cast(label, tf.float32)

    if is_training:
        #randomly rotate
        rotate_flag = tf.random_uniform(shape = [], minval = 0, maxval = 4, dtype = tf.int32)
        #rotate_flag:
        #0 - no rotation
        #1 - rotate 90 degrees
        #2 - rotate 180 degrees
        #3 - rotate 270 degrees
        image = tf.image.rot90(image, k = rotate_flag)
        label = tf.image.rot90(label, k = rotate_flag)

    #normalize the patch
    image = self.normalize_data(image)

    return image, label

def process_training_patches_epoch(self, image, label, city, name, is_training):
    """
    - normalize images
    - one hot encode label maps
    - augment data

    Args:
        image (matrix) : image [patch_size, patch_size, # of channels]
        label (matrix) : label [patch_size, patch_size]

    Returns:
        image (tensor) : modified image [patch_size, patch_size, # of channels]
        label (tensor) : modified label [patch_size, patch_size, 1]
    """

    #outputs of any tf.py_func have no shape
    #reshape image and label

    image = tf.reshape(image, [self.patch_size, self.patch_size, self.num_of_channels])
    label = tf.reshape(label, [self.patch_size, self.patch_size])

    #if there are more than one class, one hot encode the label map
    if self.num_of_classes > 1:
        label = tf.one_hot(label, depth = self.num_of_classes)

        #ignore the first class, we assume that the first class is background
        label = label[:, :, 1:]

    #if there is only one class in the ground-truth, the label map is already one hot encoded
    #just expand dimension
    else:
        label = tf.expand_dims(label, -1)

    #cast both input image and label map to float32
    image = tf.cast(image, tf.float32)
    label = tf.cast(label, tf.float32)

    #normalize the patch
    image = self.normalize_data(image)

    return image, label, city, name



def flip_up_down(self, image, label):
    """
    up-down flip

    Args:
        image (matrix) : input image
        label (matrix) : input label map

    Returns:
        image (matrix) : flipped image
        label (matrix) : flipped label map
    """
    image = tf.image.flip_up_down(image)
    label = tf.image.flip_up_down(label)

    return image, label


def flip_left_right(self, image, label):
    """
    left-right flip

    Args:
        image (matrix) : input image
        label (matrix) : input label map

    Returns:
        image (matrix) : flipped image
        image (matrix) : flipped image
        label (matrix) : flipped label map
    """

    image = tf.image.flip_left_right(image)
    label = tf.image.flip_left_right(label)

    return image, label

def add_gaussian_noise(self, image, std = 1.0):
    """
    add a noise to the input image using the gaussian distribution,
    where mean is 0.0 and standard deviation is <std>

    Args:
        image (matrix) : input image
        str (float)    : standard deviation for the gaussian distribution (optional, default: 1.0)

    Returns:
        image (matrix) : modified image
    """

    noise = tf.random_normal(shape = tf.shape(image), mean = 0.0, stddev = std, dtype = tf.float32)
    image = tf.add(image, noise)

    #constrain value of each pixel in the image between 0 and 255
    #we assume that the image is 8 bit
    image = tf.clip_by_value(image, clip_value_min = 0.0, clip_value_max = 255.0)

    return image

def random_contrast(self, image, min_val = 0.75, max_val = 1.25):
    """
    randomly change contrast of the image

    Args:
        image (matrix)  : input image
        min_val (float) : minimum value for the contrast change (optional, default: 0.75)
        max_val (float) : maximum value for the contrast change (optional, default: 1.25)

    Returns:
        image (matrix)  : modified image
    """

    image = tf.image.random_contrast(image, lower = min_val, upper = max_val)

    #constrain value of each pixel in the image between 0 and 255
    #we assume that the image is 8 bit
    image = tf.clip_by_value(image, clip_value_min = 0.0, clip_value_max = 255.0)

    return image

def translate_patch(self, image, label_map):
    """
    translate the image as well as its label map to left, right, top, and bottom directions
    magnitude of the translation for each direction is selected randomly
    after the image patch and label map are translated, their background pixels are cropped out
    then their remaining parts are resized back to their original sizes

    Args:
        image (matrix)     : image patch
        label_map (matrix) : label map

    Returns:
        image (matrix)     : translated image patch
        label_map (matrix) : translated label map
    """

    #min and max value for the shift
    shift_min = -int(self.patch_size / 5)
    shift_max = int(self.patch_size / 5)

    #generate random values for the horizontal and vertical shifts
    vert_shift = tf.random_uniform(shape = [], minval = shift_min, maxval = shift_max, dtype = tf.int32)
    horiz_shift = tf.random_uniform(shape = [], minval = shift_min, maxval = shift_max, dtype = tf.int32)

    top_left_x = tf.maximum(horiz_shift, tf.constant(0))
    top_left_y = tf.maximum(vert_shift, tf.constant(0))
    width = tf.subtract(self.patch_size, tf.abs(horiz_shift))
    height = tf.subtract(self.patch_size, tf.abs(vert_shift))

    #crop image according to the randomy generated values
    cropped_image_patch = tf.image.crop_to_bounding_box(image, top_left_y, top_left_x, height, width)
    cropped_label_patch = tf.image.crop_to_bounding_box(label_map, top_left_y, top_left_x, height, width)

    #resize both image and label patches to their original sizes
    resized_image_patch = tf.image.resize_images(images = cropped_image_patch, size = (self.patch_size, self.patch_size))
    resized_label_patch = tf.image.resize_images(images = cropped_label_patch, size = (self.patch_size, self.patch_size))

    #convert label map to binary matrix again
    resized_label_patch = tf.cast(resized_label_patch >= 0.5, tf.float32)

    return resized_image_patch, resized_label_patch

def gamma_correction(self, image, gamma):
    """
    gamma correction decribed in:
    https://en.wikipedia.org/wiki/Gamma_correction
    A is assumed to be 1, it has not been implemented
    we also assume that the input image is 8 bit

    Args:
        image (tensor) : input image
        gamma (float)  : gamma value for the correction

    Returns:
        image_gamma_corrected (tensor) : gamma corrected image
    """

    image_norm = tf.div(image, 255)
    image_gamma_corrected = tf.multiply(tf.pow(image_norm, gamma), 255)

    return image_gamma_corrected

def alpha_beta_correction(image, alpha=1.0,beta=1.0):
    alpha_beta_corrected=tf.min(tf.add(tf.max(0,tf.multiply(image, alpha)), beta),255)
    return alpha_beta_corrected
```

```
        return alpha_beta_corrected

    def normalize_data(self, image_patch):
        """
        normalize the data with the following formula
        x_normalized = (x - mean)

        Args:
            image_patch: image patch, whose shape is [patch_size,
                                                      patch_size,
                                                      # of channels>]

        Returns:
            image_patch_normalized: normalized patch with the same shape

        type of image_patch and image_patch_normalized is <tensor> during training phase,
                                                          <numpy array> during test phase
        """
        #if self.is_training:
        #    image_patch_normalized = tf.subtract(image_patch, self.mean_list)
        #else:
        #    image_patch_normalized = (image_patch.astype(np.float) - self.mean_list)

        if (self.data_type=="8bits"):
            if self.is_training:
                image_patch_normalized = tf.subtract(tf.div(image_patch,255.0), 0.5)
            else:
                image_patch_normalized = (image_patch.astype(np.float)/255.0 - 0.5)

        elif (self.data_type=="16bits"):
            if self.is_training:
                image_patch_normalized = tf.subtract(tf.div(image_patch,10000.0), 0.5)
            else:
                image_patch_normalized = (image_patch.astype(np.float)/10000.0 - 0.5)

        return image_patch_normalized

    def normalize_data_val(self, image_patch):
        """
        normalize the data with the following formula
        x_normalized = (x - mean)

        Args:
            image_patch: image patch, whose shape is [patch_size,
                                                      patch_size,
                                                      # of channels>]

        Returns:
            image_patch_normalized: normalized patch with the same shape

        type of image_patch and image_patch_normalized is <tensor> during training phase,
                                                          <numpy array> during test phase
        """


        if (self.data_type=="8bits"):
            image_patch_normalized = (image_patch.astype(np.float)/255.0 - 0.5)


        elif (self.data_type=="16bits"):
            image_patch_normalized = (image_patch.astype(np.float)/10000.0 - 0.5)


        return image_patch_normalized

    def compress_label_map(self, label_path):
        return
        """
        reduce the space that the label map located at <label_path> occupies
        using LZW compression algorithm implemented in GDAL

        Args:
            label_path (str) : full path of the label map
        """

        #create a temporary full path for the compressed image
        compressed_label_path = label_path[:-4] + '_c.tif'

        #compress the image using LZW compression algorithm
        call(['gdal_translate', '-co', 'COMPRESS=LZW', label_path, compressed_label_path])
        #if the label map is very big, comment out the line above and use the line below instead, otherwise the code might give an error
        #call(['gdal_translate', '-co', 'COMPRESS=LZW', '-co', 'BIGTIFF=YES', label_path, compressed_label_path])

        #remove the original image
        remove(label_path)

        #rename the compressed image as <label_path>
        rename(compressed_label_path, label_path)

    def read_test_patch(self, x_top_left, y_top_left, width, height):
        """
        read a patch from the data pointed by <self.geo_image>
        the patch is padded if it is needed

        Args:
            x_top_left (int) : top left location (x coordinate) of the patch in the image
            y_top_left (int) : top left location (y coordinate) of the patch in the image
            width (int)      : width of the image, from which the patch would be read
            height (int)     : height of the image, from which the patch would be read

        Returns:
            patch : shape : [# of channels, height, width]
        """

        #left padding
        pad_x_before = abs(min((x_top_left - self.padding), 0))

        #right padding
        pad_x_after = abs(min(width - (x_top_left + self.patch_size + self.padding), 0))

        #top padding
        pad_y_before = abs(min(y_top_left - self.padding, 0))

        #bottom padding
        pad_y_after = abs(min(height - (y_top_left + self.patch_size + self.padding), 0))

        #read a patch from the data pointed by <self.geo_image>
        #read a patch from the data pointed by <self.geo_image>
        patch = self.geo_image.ReadAsArray(int(x_top_left - self.padding + pad_x_before),
                                           int(y_top_left - self.padding + pad_y_before),
                                           int(self.patch_size + 2 * self.padding - pad_x_before - pad_x_after),
                                           int(self.patch_size + 2 * self.padding - pad_y_before - pad_y_after))

        num_of_channels = self.geo_image.RasterCount

        #pad the patch if it is needed
        if num_of_channels == 1:
            patch_padded = np.pad(patch, ((pad_y_before, pad_y_after), (pad_x_before, pad_x_after)), mode = 'symmetric')
            #transform patch_padded from [height, width] to [1, height, width]
            patch_padded = np.expand_dims(patch_padded, axis = 0)
        else:
            patch_padded = np.pad(patch, ((0, 0), (pad_y_before, pad_y_after), (pad_x_before, pad_x_after)), mode = 'symmetric')

        #convert chw to hwc
        patch_padded = np.transpose(patch_padded, [1, 2, 0])

        return patch_padded



    def get_name_of_tifs_in_dir(self, main_dir):
        """
        get name of the files ending with 'tif' under <main_dir>

        Args:
            main_dir (str) : main directories, where tif files are located

        Returns:
            file_names (list [str]) : list, which keeps names of each tif file
        """

        all_file_names = listdir(main_dir)

        file_names = []

        for file_name in all_file_names:

            #filter out all the files, which do not end with '.tif'
            if file_name.endswith('.tif') or file_name.endswith('.vrt'):
                file_names.append(file_name)

        return file_names

    def get_folder_full_paths(self, main_dir):
        """
        get full path of each folder located under <main_dir>

        Args:
            main_dir (str) : main directory

        Returns:
            folder_paths (list [str]) : list that keeps full path of each folder located under <main_dir>
        """
        folder_paths = [join(main_dir, file_name) for file_name in listdir(main_dir) if isdir(join(main_dir, file_name))]
        return folder_paths

    def create_list_of_empty_lists(self, num_of_elements):
        """
        create a list, which keeps multiple empty lists

        Args:
            num_of_elements (int) : parameters determining how many empty elements would be in the list

        Returns:
            output_list (list []) : list, which keeps <num_of_elements> times empty lists
        """

        output_list = []

        for _ in range(num_of_elements):
            output_list.append([])

        return output_list

    def create_image_label_paths_OLD(self, images_dir,label_dir, train_or_test, gt_folder_name = None, method_name = None):
        """
        get full paths of the images and their corresponding label/predicted maps in a directory

        Args:
            images_dir (str) : directory in which images are located
            labels_dir (str) : directory in which label/predicted maps are/would be located

        Returns:
            image_paths (list [str]): list that keeps full paths of all the images in a directory
                images_paths[0]     : full path of the image<1>
                images_paths[1]     : full path of the image<2>
                    ...
                images_paths[n - 1] : full path of the image<n>

            label_paths (list [str]): list that keeps full paths of all the label/pred maps in a directory
                label_paths[0]      : full path of the label/pred map<1>
                label_paths[1]      : full path of the label/pred map<2>
                    ...
                label_paths[n - 1]  : full path of the label/pred map<n>
        """

        #get names of all the files under the given directory
        from os import listdir
        from os.path import isfile, join

        onlyfilesImages = [f for f in listdir(images_dir) if isfile(f)]
        image_paths = []
        label_paths = []

        #there can be redundant files (applications like QGIS usually create an ".xml" file when an image is displayed)
        #all the files except ".tif" need to be filtered out

        for image_name in listdir(images_dir):
            if image_name.endswith('.tif') or image_name.endswith('.vrt'):
                image_paths.append(join(images_dir, image_name))
                label_paths.append(join(label_dir, image_name))

        return image_paths, label_paths
```

```
def create_image_label_paths(self, db_main_dir, train_or_test, gt_folder_name = None, method_name = None):
    """
    create full paths for the inputs images as well as label maps

    Args:
        db_main_dir (str)    : main directory, where the database is located
        train_or_test (str)  : 'train' for training data
                               'test' for test data
        method_name (str)    : name of the method. Used only in test phase. It is appended to name of the output file
        gt_folder_name (str) : parameter determining which ground-truth to be used. Used only in training phase
    Returns:
        image_full_paths (list[list[list[list(str)]]]) : full paths of the training/test images
        label_full_paths (list[list[list[list(str)]]]) : full paths of the label maps for training
                                                                           predicted maps for test

        both image_full_paths and label_full_paths are in this format:
        data[i] => list, which keeps continents
        data[i][j] => list, which keeps countries
        data[i][j][k] => list, which keeps cities
        data[i][j][k][l] => list, which keep full path of the images
    """

    #get full paths of the continents
    continent_paths = self.get_folder_full_paths(join(db_main_dir, train_or_test))
    num_of_continents = len(continent_paths)

    image_full_paths = self.create_list_of_empty_lists(num_of_continents)
    label_full_paths = self.create_list_of_empty_lists(num_of_continents)

    #traverse over each continent
    for i, continent_path in enumerate(continent_paths):

        #get full paths of the countries
        country_paths = self.get_folder_full_paths(continent_path)
        num_of_countries = len(country_paths)

        image_full_paths[i] = self.create_list_of_empty_lists(num_of_countries)
        label_full_paths[i] = self.create_list_of_empty_lists(num_of_countries)

        #traverse over each country
        for j, country_path in enumerate(country_paths):

            #get full paths of the cities
            city_paths = self.get_folder_full_paths(country_path)
            num_of_cities = len(city_paths)

            image_full_paths[i][j] = self.create_list_of_empty_lists(num_of_cities)
            label_full_paths[i][j] = self.create_list_of_empty_lists(num_of_cities)

            #traverse over each city
            for k, city_path in enumerate(city_paths):

                file_names = self.get_name_of_tifs_in_dir(join(city_path, 'image'))

                #add all the files to <image_full_paths> and <label_full_paths>
                for file_name in file_names:

                    image_full_paths[i][j][k].append(join(city_path, 'image', file_name))

                    if self.is_training:
                        label_full_paths[i][j][k].append(join(city_path, gt_folder_name, file_name))
                    else:
                        file_name = file_name.split('.')[0] + '_' + method_name + '.tif'
                        label_full_paths[i][j][k].append(join(city_path, 'pred', file_name))

    return image_full_paths, label_full_paths

def create_image_label_paths_by_folder(self,images_dir,label_dir):
    """
    create full paths for the inputs images as well as label maps

    Args:
        images_dir (str) : directory in which images are located
        labels_dir (str) : directory in which label/predicted maps are/would be located
        method_name (str)    : name of the method. Used only in test phase. It is appended to name of the output file
        gt_folder_name (str) : parameter determining which ground-truth to be used. Used only in training phase
    Returns:
        image_full_paths (list(str)) : full paths of the training/test images
        label_full_paths (list(str)) : full paths of the label maps for training


        both image_full_paths and label_full_paths are in this format:
        data[k] => list, which keeps cities

    """

    #get full paths of the cities
    city_paths = self.get_folder_full_paths(images_dir)

    num_of_cities = len(city_paths)

    print("num " ,num_of_cities )
    label_paths = self.get_folder_full_paths(label_dir)

    #print("number of cites", num_of_cities, city_paths)
    image_full_paths = self.create_list_of_empty_lists(num_of_cities)
    label_full_paths = self.create_list_of_empty_lists(num_of_cities)


    #traverse over each continent
    for k, city_path in enumerate(city_paths):
        file_names = self.get_name_of_tifs_in_dir(city_path)

        for file_name in file_names:

            image_full_paths[k].append(join(city_path, file_name))

            label_full_paths[k].append(join(label_paths[k], file_name))


    return image_full_paths, label_full_paths

def patch_generator_by_folder_train(self, is_training):

    """
    generator function that yields an image patch and a label map
```

```
        the patch is sampled with the following algorithm

        1 - select a folder (a random city from all folders)
        2 - select a random patch from the chosen city

        Yields:
            image (matrix): an image patch: [patch_size,
                                             patch_size,
                                             # of channels]
            label (matrix): label map     : [patch_size,
                                             patch_size]
        """

        local_image_paths = self.image_paths

        while True:
           # good=False
            folder_index = random.randint(0, int(len(local_image_paths)) - 1)
            image_index  = random.randint(0, int(len(local_image_paths[folder_index])) - 1)
            image_patch, label_patch  = self.read_training_patch_by_folder(folder_index, image_index, is_training)
#            while  not good :
#                print ("redo")
#                folder_index = random.randint(0, int(len(local_image_paths)) - 1)
#                image_index  = random.randint(0, int(len(local_image_paths[folder_index])) - 1)
#                good,image_patch, label_patch  = self.read_training_patch_by_folder(folder_index, image_index, is_training)


            if self.object_class=="building":
                k1 = cv2.getStructuringElement(cv2.MORPH_CROSS,
                                               (3,3))
                label_patch = cv2.dilate(label_patch, k1, iterations=1)


            if self.object_class=="buildingNoBorder":
                label_patch = delete_contour(label_patch, self.data_type)


            if self.object_class=="road":

                label_patch = delete_contour(label_patch, self.data_type)
                label_patch_dilate = label_patch

                k1 = cv2.getStructuringElement(cv2.MORPH_CROSS, (5,5))
                label_patch_dilate =cv2.dilate(label_patch_dilate, k1, iterations=3)

                mask = (label_patch_dilate-label_patch)*2

                label_patch = label_patch +mask


            if (self.data_type=="8bits"):

                if  random.randint(0, 2)==1  :

                    alpha= np.random.uniform(0.8, 1.2)#0.5,1.5 #https://docs.opencv.org/3.0-beta/doc/tutorials/core/basic_linear_transform/basic_linear_transform.html
                    beta=  np.random.uniform(-10, 10)#-40,40
                    gamma= np.random.uniform(0.67,1.5)#https://docs.opencv.org/3.3.0/d3/dc1/tutorial_basic_linear_transform.html


                    image_patch[:,:,0]=random_etal(image_patch[:,:,0], self.data_type)
                    image_patch[:,:,1]=random_etal(image_patch[:,:,1], self.data_type)
                    image_patch[:,:,2]=random_etal(image_patch[:,:,2], self.data_type)

                    image_patch=adjust_alpha_beta(image_patch, self.data_type, alpha,beta)
                    image_patch=adjust_gamma(image_patch,self.data_type, gamma)

                #add_backgound, just +1
#                image_patch[0]=fix_etalIr(image_patch[:,:,0])
#                image_patch[1]=fix_etalR(image_patch[:,:,1])
#                image_patch[2]=fix_etalG(image_patch[:,:,2])
#
            label_patch=add_backgound(label_patch, self.data_type)
            yield image_patch, label_patch



    def patch_generator_by_folder_val(self, is_training):

        """
        generator function that yields an image patch and a label map
        the patch is sampled with the following algorithm

        1 - select a folder (a random city from all folders)
        2 - select a random patch from the chosen city

        Yields:
            image (matrix): an image patch: [patch_size,
                                             patch_size,
                                             # of channels]
            label (matrix): label map     : [patch_size,
                                             patch_size]
        """

        local_image_paths = self.image_paths_val

        while True:
            folder_index = random.randint(0, int(len(local_image_paths)) - 1)
            image_index  = random.randint(0, int(len(local_image_paths[folder_index])) - 1)
            image_patch, label_patch  = self.read_val_patch_by_folder(folder_index, image_index)

            if self.object_class=="building":
                    k1 = cv2.getStructuringElement(cv2.MORPH_CROSS, (3,3))
                    label_patch = cv2.dilate(label_patch, k1, iterations=1)


            if self.object_class=="buildingNoBorder":
                    label_patch = delete_contour(label_patch, self.data_type)


            if self.object_class=="road":

                    label_patch = delete_contour(label_patch, self.data_type)
                    label_patch_dilate = label_patch

                    k1 = cv2.getStructuringElement(cv2.MORPH_CROSS, (5,5))
                    label_patch_dilate =cv2.dilate(label_patch_dilate, k1, iterations=3)
                    mask = (label_patch_dilate-label_patch)*2

                    label_patch = label_patch +mask


            if (self.data_type=="8bits"):

                    if  random.randint(0, 2)==1  :

                        alpha= np.random.uniform(0.8, 1.2)#0.5,1.5 #https://docs.opencv.org/3.0-beta/doc/tutorials/core/basic_linear_transform/basic_linear_transform.html
                        beta=  np.random.uniform(-10, 10)#-40,40
                        gamma= np.random.uniform(0.67,1.5)#https://docs.opencv.org/3.3.0/d3/dc1/tutorial_basic_linear_transform.html


                        image_patch[:,:,0]=random_etal(image_patch[:,:,0], self.data_type)
                        image_patch[:,:,1]=random_etal(image_patch[:,:,1], self.data_type)
                        image_patch[:,:,2]=random_etal(image_patch[:,:,2], self.data_type)

                        image_patch=adjust_alpha_beta(image_patch, self.data_type, alpha,beta)
                        image_patch=adjust_gamma(image_patch,self.data_type, gamma)

                    #add_backgound, just +1
            label_patch=add_backgound(label_patch, self.data_type)

            yield image_patch, label_patch



    def patch_generator_by_folder_val_epoch(self):

        """
        generator function that yields an image patch and a label map
        the patch is sampled with the following algorithm

        1 - select a folder (a random city from all folders)
        2 - select a random patch from the chosen city

        Yields:
            image (matrix): an image patch: [patch_size,
                                             patch_size,
                                             # of channels]
            label (matrix): label map     : [patch_size,
                                             patch_size]
            city: path of the city corresponding to the image: string
            image: path of the image: string
        """


        self.image_paths_val, self.label_paths_val = self.create_image_label_paths_by_folder(self.images_dir_val,self.labels_dir_val)

        for city_index, list_images in enumerate(self.image_paths_val):
            for image_index, image in enumerate(self.image_paths_val[city_index]):

                city = os.path.dirname(image)
                image_patch, label_patch  = self.read_val_patch_by_folder_epoch(image, self.label_paths_val[city_index][image_index])

                if self.object_class=="building":
                            k1 = cv2.getStructuringElement(cv2.MORPH_CROSS, (3,3))
                            label_patch = cv2.dilate(label_patch, k1, iterations=1)


                if self.object_class=="buildingNoBorder":
                            label_patch = delete_contour(label_patch, self.data_type)


                if self.object_class=="road":
                            label_patch = delete_contour(label_patch, self.data_type)
                            label_patch_dilate = label_patch

                            k1 = cv2.getStructuringElement(cv2.MORPH_CROSS, (5,5))
                            label_patch_dilate =cv2.dilate(label_patch_dilate, k1, iterations=3)

                            mask = (label_patch_dilate-label_patch)*2

                            label_patch = label_patch +mask


                if (self.data_type=="8bits"):

                            if  random.randint(0, 2)==1  :

                                alpha= np.random.uniform(0.8, 1.2)#0.5,1.5 #https://docs.opencv.org/3.0-beta/doc/tutorials/core/basic_linear_transform/basic_linear_transform.html
                                beta=  np.random.uniform(-10, 10)#-40,40
                                gamma= np.random.uniform(0.67,1.5)#https://docs.opencv.org/3.3.0/d3/dc1/tutorial_basic_linear_transform.html

                                image_patch[:,:,0]=random_etal(image_patch[:,:,0], self.data_type)
                                image_patch[:,:,1]=random_etal(image_patch[:,:,1], self.data_type)
                                image_patch[:,:,2]=random_etal(image_patch[:,:,2], self.data_type)

                                image_patch=adjust_alpha_beta(image_patch, self.data_type, alpha,beta)
                                image_patch=adjust_gamma(image_patch,self.data_type, gamma)

                            #add_backgound, just +1
                label_patch=add_backgound(label_patch, self.data_type)

                yield image_patch, label_patch, city, image


    def patch_generator_OLD(self):
        """
        generator function that yields an image patch and a label map
        the patch is sampled with the following algorithm


        Yields:
            image (matrix): an image patch: [patch_size,
                                             patch_size,
                                             # of channels]
            label (matrix): label map     : [patch_size,
                                             patch_size]
        """

        local_image_paths = self.image_paths
        local_label_paths = self.label_paths
```

```
        while True:
            import cv2

            index = random.randint(0, int(len(local_image_paths)) - 1)
            image_patch, label_patch = self.read_training_patch(local_image_paths[index],
                                                    local_label_paths[index])


            if self.object_class=="building":
                k1 = cv2.getStructuringElement(cv2.MORPH_CROSS, (3,3))
                label_patch = cv2.dilate(label_patch, k1, iterations=1)

            if self.object_class=="buildingNoBorder":
                label_patch=delete_contour(label_patch, self.data_type)

            if  random.randint(0, 2)==1  :#if bati
                #dilate roof
                #k1 = cv2.getStructuringElement(cv2.MORPH_CROSS, (3,3))

                #label_patch[:, :, 2]=mask1
                #label_patch[:, :, 0]=label_patch[:, :, 0]-label_patch[:, :, 2]
                #label_patch[:, :, 1]=label_patch[:, :, 1]-label_patch[:, :, 2]


                alpha= np.random.uniform(0.8, 1.2)#0.5,1.5 #https://docs.opencv.org/3.0-beta/doc/tutorials/core/basic_linear_transform/basic_linear_transform.html
                beta=  np.random.uniform(-10, 10)#-40,40
                gamma= np.random.uniform(0.67,1.5)#https://docs.opencv.org/3.3.0/d3/dc1/tutorial_basic_linear_transform.html


                image_patch[:,:,0]=random_etal(image_patch[:,:,0], self.data_type)
                image_patch[:,:,1]=random_etal(image_patch[:,:,1], self.data_type)
                image_patch[:,:,2]=random_etal(image_patch[:,:,2], self.data_type)



                image_patch=adjust_alpha_beta(image_patch, self.data_type, alpha, beta)
                image_patch=adjust_gamma(image_patch,self.data_type, gamma)

            #add_backgound, just +1
            label_patch=add_backgound(label_patch,self.data_type)
            yield image_patch, label_patch



    def open_test_image_label(self, image_path, pred_path):
        """
        create a tif file for the output classification map
        georeference the classification map using the input image

        Args:
            image_path (str) : full path for the input image
            pred_path (str)  : full path for the predicted label map
        """

        #open current image
        self.geo_image = gdal.Open(image_path)
        prj = self.geo_image.GetProjection()
        geotransform = self.geo_image.GetGeoTransform()

        height = self.geo_image.RasterYSize
        width = self.geo_image.RasterXSize

        driver = gdal.GetDriverByName("GTiff")

        #create a tif file for the predicted maps
        self.geo_label_map = driver.Create(pred_path,
                                           width,
                                           height,
                                           1,gdal.GDT_Byte)

#        self.geo_label_proba = driver.Create(pred_path+"_prob.tif",
#                                            width,
#                                            height,
#                                            2,gdal.GDT_Float32)
#                                            #gdal.GDT_Byte, ['NBITS=2'])

        #if the input image is georeferenced
        #georeference the label map using georeference information of the input image
        self.geo_label_map.SetGeoTransform(geotransform)
        if len(prj) > 0:
            self.geo_label_map.SetProjection(prj)

#        self.geo_label_proba.SetGeoTransform(geotransform)
#        if len(prj) > 0:
#            self.geo_label_proba.SetProjection(prj)

    def close_test_image_label(self):
        """
        close the created label map and the opened image
        """
        self.geo_label_map = None
        self.geo_image = None

    def find_num_of_channels(self):
        """
        find # of channels of the patches using the first patch
        we assume that all of the patches have the same number of channels

        Returns:
            num_of_channels (int) : number of channels in each patch
        """

        #four zeros correspond to the first patch of the first city of the first country of the first continent
        #geo = gdal.Open(self.image_paths[0][0][0][0])

        if (self.by_folder):

            geo = gdal.Open(self.image_paths[0][0])

        else:

            geo = gdal.Open(self.image_paths[0])
        num_of_channels = geo.RasterCount
        del geo
        return num_of_channels
    def find_patch_size(self):
        """
        find patch size of the patches using the first patch
        we assume that height and width of all the patches are the same and equal to patch size
        """

        #four zeros correspond to the first patch of the first city of the first country of the first continent
        #geo = gdal.Open(self.image_paths[0][0][0][0])
        print (str(self.image_paths[0]))
        geo = gdal.Open(self.image_paths[0])
        patch_size = geo.RasterYSize
        del geo
        return patch_size


from subprocess import call
import glob

batch_size=6

call(["python", "main.py",
      "--is_training=True",
      "--fine_tuning=True",
      "--by_folder=True",
      "--mini_batch=True",

      "--hors_ville_image_dir=/home/iheb/externe/buildingTraining_hors_ville/images",
      "--hors_ville_label_dir=/home/iheb/externe/buildingTraining_hors_ville/gt",
      "--images_dir=/home/iheb/externe/buildingTraining/data_sample/images",
      "--labels_dir=/home/iheb/externe/buildingTraining/data_sample/gt",
      "--images_dir_val=/home/iheb/externe/buildingValid/data_sample/images",
      "--labels_dir_val=/home/iheb/externe/buildingValid/data_sample/gt",
      "--images_dir_test=/home/iheb/interne/buildingTest/strict_villes",
      "--labels_dir_test=/home/iheb/interne/buildingTest/predduringtrain",
      "--num_epoch_test_pred=100",
      "--mean_list=119.78,82.72,81.52",
      "--mean_list_val=0,0,0",
      "--num_of_classes=4",
      "--patch_size=384",
      "--padding=92",
      "--batch_size="+str(batch_size),
      "--learning_rate=0.0002",
      "--num_of_epochs=12000",
      "--num_of_iterations=10000",#+str(len(glob.glob("/home/iheb/externe/buildingTraining/data_sample/images/*/*"))//batch_size),
      "--decay_epoch=500",
      "--padding_val=64",
      "--patch_size_val=2240",
      "--decay_rate=0.0001",
      "--data_type=16bits",
      "--snap_dir=//home/iheb/externe/snap0/",
      "--snap_freq=2",
      "--log_dir=/home/iheb/externe/log0",
      "--object_class=IndustrialBuilding"
])


from subprocess import call

call(["python", "main.py",
      "--is_training=False",
      "--fine_tuning=False",
      "--images_dir=/home/iheb/interne/buildingTest/strict_villes/japon",
      "--labels_dir=/home/iheb/interne/buildingTest/1",
      "--mean_list=0,0,0",
      "--std_list=0,0,0",
      "--num_of_classes=4",
      "--patch_size=2240",
      "--padding=64",
      "--data_type=16bits",
      "--snap_dir=/home/iheb/externe/snap0/",
      "--object_class=IndustrialBuilding",

])

import sys
import numpy as np
import tensorflow as tf
from unet_model import Unet_model


def del_all_flags(FLAGS):
    flags_dict = FLAGS._flags()
    keys_list = [keys for keys in flags_dict]
    for keys in keys_list:
        FLAGS.__delattr__(keys)

#del_all_flags(tf.flags.FLAGS)

flags = tf.app.flags
#parameters for both phases
flags.DEFINE_string("is_training", None, "True: training phase, False: test phase")

#flags.DEFINE_string("db_main_dir", None, "main directory, where database is located")
flags.DEFINE_string("by_folder", None, "True : the vrt are classify in folders in images/ and /gt/ (if you used lxDatasetGenerationDL with the parameter -by_folder y)"+
                                    "False: the vrt are directly in images/ and gt/")
flags.DEFINE_string("images_dir", None, "images directory for train or test")
flags.DEFINE_string("labels_dir", None, "labels directory for train or test")

flags.DEFINE_string("hors_ville_image_dir", None, "images directory for out of city images ")
flags.DEFINE_string("hors_ville_label_dir", None, "labels directory for out of city images")

flags.DEFINE_string("snap_dir", None, "snapshot directory, where weights are saved regularly as the training continues. " +
                                      "during the test phase, weights are restored from the last checkpoint under this directory")
flags.DEFINE_string("mean_list", None, "mean value for each channel")
flags.DEFINE_string("data_type", None, "data type : uint8 or uint16")



#parameters for the training phase only
flags.DEFINE_boolean ("mini_batch", None, "True:mini_batchGD, False:SGD")

    #use a validation dataset to evaluate the model for each epoch
flags.DEFINE_string("images_dir_val", None, "images directory for validation")
flags.DEFINE_string("labels_dir_val", None, "labels directory fro validation")

    #use a test dataset (need to be small) to do a prediction on these images every num_epoch_test_pred epochs
flags.DEFINE_string("images_dir_test", None, "images directory for testing the model every num_epoch_test_pred epochs")
flags.DEFINE_string("labels_dir_test", None, "output predictions directory")

flags.DEFINE_integer("batch_size", None, "number of patches in a batch during the training phase")
flags.DEFINE_integer("num_of_classes", None, "number of classes")
flags.DEFINE_integer("num_epoch_test_pred", None, "number of epochs at the end of which we will make a prediction on validation images")
flags.DEFINE_string("gt_folder_name", None, "parameter determining which ground-truth to use")
flags.DEFINE_float("learning_rate", None, "learning rate for the adam optimizer")
flags.DEFINE_integer("num_of_epochs", None, "number of epochs")
flags.DEFINE_integer("num_of_iterations", None, "number of iterations in each epoch")
flags.DEFINE_integer("decay_epoch", None, "the parameter to determine in which epoch the learning rate for the adam optimizer would be decreased")
flags.DEFINE_float("decay_rate", None, "the parameter to determine how much the learning rate would be decreased")
#flags.DEFINE_string("log_dir", None, "log directory, where logs are saved")
flags.DEFINE_string("fine_tuning", None, "True  : fine tuning mode on. Pretrained model is restored and continued training." +
                                          "False : fine tuning model off. The model is trained from scratch")
flags.DEFINE_integer("snap_freq", None, "parameter detemining how often the trained model would be saved")
flags.DEFINE_string("object_class", None, "the kind of object to classify : building, tree, road or buildingNoBorder ")
flags.DEFINE_integer("patch_size_val", None, "since the val image might be very big, it is segmented patch by patch." +
                                         "This parameter sets height and width of each patch")
flags.DEFINE_integer("padding_val", None, "padding is used to get rid of border effect during the test phase. " +
                                      "This parameter determines overlapping amount between the patches that are read from the big test image")
flags.DEFINE_string("mean_list_val", None, "mean value for each channel for the validation")


#parameters for the test phase only
flags.DEFINE_integer("patch_size", None, "since the test image might be very big, it is segmented patch by patch." +
                                         "This parameter sets height and width of each patch")
flags.DEFINE_integer("padding", None, "padding is used to get rid of border effect during the test phase. " +
                                      "This parameter determines overlapping amount between the patches that are read from the big test image")
flags.DEFINE_string("model_name", None, "name of the method. It is appended to the output file name")
flags.DEFINE_string("idGPU", "0","id of the GPU device")


FLAGS = flags.FLAGS

def check_parameters():
    """
    Check of all the required parameters are set
    """

    FLAGS.is_training="True"== FLAGS.is_training
    FLAGS.fine_tuning="True" ==FLAGS.fine_tuning
    FLAGS.by_folder="True" ==FLAGS.by_folder


    #check the common parameters
    if FLAGS.is_training == None:
        sys.exit('--is_training parameter has to be set!')
    if FLAGS.snap_dir == None:
        sys.exit('--snap_dir parameter has to be set!')
    if FLAGS.mean_list == None:
        sys.exit('--mean_list parameter has to be set!')
    if FLAGS.images_dir == None:
        sys.exit('--images_dir parameter has to be set!')
    if FLAGS.labels_dir == None:
        sys.exit('--labels_dir parameter has to be set!')


    if FLAGS.num_of_classes == None:
        sys.exit('--num_of_classes parameter has to be set!')
    if FLAGS.padding == None:
            sys.exit('--padding parameter has to be set!')
    if FLAGS.data_type == None:
            sys.exit('--data_type parameter has to be set!')

    #check the parameters for training
    if FLAGS.is_training:
        if FLAGS.hors_ville_image_dir == None:
            sys.exit('--hors_ville_image_dir parameter has to be set!')
        if FLAGS.hors_ville_label_dir == None:
            sys.exit('--hors_ville_label_dir parameter has to be set!')
        if FLAGS.mini_batch == None:
            sys.exit('--mini_batch parameter has to be set!')
        if FLAGS.batch_size == None:
            sys.exit('--batch_size parameter has to be set!')
        if FLAGS.learning_rate == None:
            sys.exit('--learning_rate parameter has to be set!')
        if FLAGS.num_of_epochs == None:
            sys.exit('--num_of_epochs parameter has to be set!')
        if FLAGS.decay_epoch == None:
            sys.exit('--decay_epoch parameter has to be set!')
        if FLAGS.num_of_iterations == None:
            sys.exit('--num_of_iterations parameter has to be set!')
        if FLAGS.decay_rate == None:
            sys.exit('--decay_rate parameter has to be set!')
        if FLAGS.log_dir == None:
            sys.exit('--log_dir parameter has to be set!')
        if FLAGS.fine_tuning == None:
            sys.exit('--fine_tuning parameter has to be set!')
        if FLAGS.snap_freq == None:
            FLAGS.snap_freq=50
        if FLAGS.object_class == None:
            sys.exit('--object_class parameter has to be set!')
        if FLAGS.by_folder == None:
            sys.exit('--by_folder parameter has to be set!')
        if FLAGS.num_epoch_test_pred == None:
            sys.exit('--num_epoch_test_pred parameter has to be set!')
        if FLAGS.images_dir_val == None:
            sys.exit('--images_dir_val parameter has to be set!')
        if FLAGS.labels_dir_val == None:
            sys.exit('--labels_dir_val parameter has to be set!')
        if FLAGS.images_dir_test == None:
            sys.exit('--images_dir_test parameter has to be set!')
        if FLAGS.labels_dir_test == None:
            sys.exit('--labels_dir_test parameter has to be set!')

    #check the parameters for inference
    else:
        if FLAGS.patch_size == None:
            sys.exit('--patch_size parameter has to be set!')


def parse_mean_list(mean_list):
    """
    parse FLAGS.mean_list according to comma

    Returns:
        mean_list (list [float])   : list containing mean value for each channel
    """
    mean_list = np.array(FLAGS.mean_list.split(','), np.float32)
    return mean_list

def main(_):
    import os
    os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"   # see issue #152
    os.environ["CUDA_VISIBLE_DEVICES"]=str(FLAGS.idGPU)
    print (str(FLAGS.is_training))
    print (str(FLAGS.fine_tuning))

    check_parameters()
    mean_list = parse_mean_list(FLAGS.mean_list)
    print (str(FLAGS.model_name))

    unet_model = Unet_model(
                           images_dir = FLAGS.images_dir,
                           labels_dir = FLAGS.labels_dir,
                           images_dir_val = FLAGS.images_dir_val,
                           labels_dir_val = FLAGS.labels_dir_val,
                           images_dir_test = FLAGS.images_dir_test,
                           labels_dir_test = FLAGS.labels_dir_test,
                           gt_folder_name = FLAGS.gt_folder_name,
                           patch_size = FLAGS.patch_size,
                           padding = FLAGS.padding,
                           num_of_classes = FLAGS.num_of_classes,
                           mean_list = mean_list,
                           batch_size = FLAGS.batch_size,
                           learning_rate = FLAGS.learning_rate,
                           num_of_epochs = FLAGS.num_of_epochs,
                           num_of_iterations = FLAGS.num_of_iterations,
                           decay_epoch = FLAGS.decay_epoch,
                           decay_rate = FLAGS.decay_rate,
                           is_training = FLAGS.is_training,
                           method_name = FLAGS.model_name,
                           object_class = FLAGS.object_class,
                           by_folder = FLAGS.by_folder,
                           patch_size_val = FLAGS.patch_size,
                           num_epoch_test_pred = FLAGS.num_epoch_test_pred,
                           data_type = FLAGS.data_type,
                           mini_batch=FLAGS.mini_batch,
                           hors_ville_image_dir=FLAGS.hors_ville_image_dir,
                           hors_ville_label_dir=FLAGS.hors_ville_label_dir
                           )
    if FLAGS.is_training:

            unet_model.train_model(FLAGS.snap_dir, FLAGS.snap_freq, FLAGS.log_dir, FLAGS.fine_tuning)


            num = int(int(FLAGS.num_of_epochs)/int(FLAGS.num_epoch_test_pred))
            for epoch in range(num):
                if (FLAGS.images_dir_val and FLAGS.labels_dir_val):
                    if (FLAGS.images_dir_test and FLAGS.labels_dir_test):

                        tf.reset_default_graph()

                        mean_list_val = parse_mean_list(FLAGS.mean_list_val)
                        unet_model = Unet_model(
                                        images_dir = FLAGS.images_dir,
                                        labels_dir = FLAGS.labels_dir,
                                        images_dir_val = FLAGS.images_dir_val,
                                        labels_dir_val = FLAGS.labels_dir_val,
                                        images_dir_test = FLAGS.images_dir_test,
                                        labels_dir_test = FLAGS.labels_dir_test,
                                        gt_folder_name = FLAGS.gt_folder_name,
                                        patch_size = FLAGS.patch_size_val,
                                        padding = FLAGS.padding_val,
                                        num_of_classes = FLAGS.num_of_classes,
                                        mean_list = mean_list_val,
                                        batch_size = FLAGS.batch_size,
                                        learning_rate = FLAGS.learning_rate,
                                        num_of_epochs = FLAGS.num_of_epochs,
                                        num_of_iterations = FLAGS.num_of_iterations,
                                        decay_epoch = FLAGS.decay_epoch,
                                        decay_rate = FLAGS.decay_rate,
                                        is_training = FLAGS.is_training,
                                        method_name = "pred",
                                        object_class = FLAGS.object_class,
                                        by_folder = FLAGS.by_folder,
                                        patch_size_val = FLAGS.patch_size,
                                        num_epoch_test_pred = FLAGS.num_epoch_test_pred,
                                        data_type = FLAGS.data_type,
                                        mini_batch=FLAGS.mini_batch,
                                        hors_ville_image_dir=FLAGS.hors_ville_image_dir,
                                        hors_ville_label_dir=FLAGS.hors_ville_label_dir


                                        )


                        unet_model.classify_test(FLAGS.snap_dir, (epoch+1)*int(FLAGS.num_epoch_test_pred))

                print("Train again", epoch )
                tf.reset_default_graph()

                unet_model = Unet_model(
                        images_dir = FLAGS.images_dir,
                        labels_dir = FLAGS.labels_dir,
                        images_dir_val = FLAGS.images_dir_val,
                        labels_dir_val = FLAGS.labels_dir_val,
                        images_dir_test = FLAGS.images_dir_test,
                        labels_dir_test = FLAGS.labels_dir_test,
                        gt_folder_name = FLAGS.gt_folder_name,
                        patch_size = FLAGS.patch_size,
                        padding = FLAGS.padding,
                        num_of_classes = FLAGS.num_of_classes,
                        mean_list = mean_list,
                        batch_size = FLAGS.batch_size,
                        learning_rate = FLAGS.learning_rate,
                        num_of_epochs = FLAGS.num_of_epochs,
                        num_of_iterations = FLAGS.num_of_iterations,
                        decay_epoch = FLAGS.decay_epoch,
                        decay_rate = FLAGS.decay_rate,
                        is_training = FLAGS.is_training,
                        method_name = FLAGS.model_name,
                        object_class = FLAGS.object_class,
                        by_folder = FLAGS.by_folder,
                        patch_size_val = FLAGS.patch_size,
                        num_epoch_test_pred = FLAGS.num_epoch_test_pred,
                        data_type = FLAGS.data_type,
                        mini_batch=FLAGS.mini_batch,
                        hors_ville_image_dir=FLAGS.hors_ville_image_dir,
                        hors_ville_label_dir=FLAGS.hors_ville_label_dir
                        )

                unet_model.train_model(FLAGS.snap_dir, FLAGS.snap_freq, FLAGS.log_dir, True)

    else:
            print(" model" , FLAGS.snap_dir)
            unet_model.classify(FLAGS.snap_dir)

if __name__ == '__main__':
    tf.app.run()
```