CONTENTS

ABSTRACT

Transposed convolutional layers (TCL) were extensively utilized in lots of deep learning models such as encoding-decoding network for semantic image segmentation and deep learning models for unsupervised learning for up-sampling. Checkerboard problem is one of the major boundaries of TCL operations. This is due to the truth that no direct relationship exists amongst adjoining pixels at the output featured map.

To deal with this problem, we suggest the pixel transposed convolutional layer (Pixel TCL) to set up direct relationships amongst adjoining pixels at the up-sampled featured map. Our technique is primarily based totally on a sparkling interpretation of the regular transposed convolutional operation. The ensuing Pixel TCL may be used to update any kind of TCL in a way in which we do not have to configure each and every time, hence there is no need of actually compromising the completely trainable abilities of the actual models.

The Pixel TCL that is introduced in this study, and the regular TCL are then applied. The take a look at effects on semantic segmentation demonstrates which of the above applied convolutional layer can recollect spatial features which includes edges, shapes and yield segmentation outputs with greater accuracy.

Keywords: Deep learning, TCL, up-sampling, PixelTcl

1. INTRODUCTION

Deep learning is very vast field, in which numerous researches are being held. There are various methods in deep learning that have shown an eye-catching amount of results in a variety of fields in artificial intelligence such as image classification, semantic segmentation and natural image generation. Today, each and every task we perform is somehow dependent upon these so-called deep learning methodologies. Few of the important layers that involve pooling layers, convolution layers, and transposed convolutional layers (TCLs), were often used to generate deep learning models for various number of responsibilities. Transposed convolutional layers were broadly speaking utilized in deep learning models in which up-sampling of featured maps is required, which includes generative models and also encoder-decoder architectures. Although TCLs are able to generate featured maps from smaller ones to the larger, they be afflicted by the checkerboard problem. This substantially makes the deep learning model to limit its competencies in producing photographically-realistic images and generating a smooth output on semantic segmentation. Till now, there have been little or no efforts were dedicated in enhancing the transposed convolutional operation.

Deep learning has been very successful while experimenting with images as statistics and is presently at a level in which this is really working better than people on more than one use-case. Maximum essential issues that human beings had been very keen about, by fixing with computer vision are classification of images, object detection and image segmentation withinside the increasing order in their difficulty.

In the obvious older challenges of image classification we were simply interested by getting the labels of all of the items which can be found in a particular image. In object detection we move a step forward and attempt to understand along side what all objects which might be found in an image, the place at which the objects are present with the assistance of bounding boxes. Image segmentation takes it to a brand new stage through attempting to find out accurately the precise boundary of the objects withinside the image.
In this study, we have proposed a simple, but an effective technique, called PixelTCL, to cope with the checkerboard artifacts. This technique is stimulated as a sparkling

interpretation of TCL functions that simply points out the main cause of the problem of checkerboard. The featured up-sampled map is generated with the aid of using transposed convolutional layer may be taken into consideration because the end result of periodic shuffling of a couple of intermediate featured maps obtained from an input featured map with the aid of using independent convolutions. Adjoining pixels at the output featured map aren't related directly, hence results in the checkerboard problem. In order to solve this problem, we endorse the pixel tcn functions for using it in PixelTCL. In the latest layer, the intermediate featured maps are about to be generated in a sequential manner in order to let the featured maps generated in a later level are required to rely on the previously generated ones. In this manner, the direct relationships amongst adjoining pixels at the output featured map were established. Sequentially generating intermediate featured maps in PixelTCL can also additionally bring about mild lower in computational efficiency, however we display that this may be in a large part triumph over with the aid of using an implementational technique. Experimental effects on semantic image segmentation, tasks based upon image generation show that the proposed Pixel TCL is able to successfully solve the checkerboard artifacts, further enhance the performance by effective prediction and generation.

2. LITERATURE SURVEY

Our work is based upon the IEEE paper published “Pixel Transposed convolutional Network”. This study deals with improving the deep learning models previously introduced in the existing system in order to solve the checker-board problem. This study also provides us the knowledge of the main idea and is based on establishing the relationship amongst the units on the same feature map which, which is related to the PixelRNNs, PixelCNNs, that are basically generative models that are considered to have a certain relationship amongst the units on a particular featured map. They belong to a general class of autoregressive methods used for estimating the probability density. With the help of masked-CNN during the training, the training time of PixelRNNs and PixelCNNs is comparable to that of other generative models such as generative adversarial networks (GANs) and variational autoencoders (VAEs). However, the time taken to predict PixelRNNs or PixelCNNs is very slow when compared, it has to generate images pixel by pixel. In contrast, our PixelTCL can be used to replace any transposed convolutional layer in a way in which we do not have to configure each and every time, and the slight decrease in efficiency can be largely overcome by an implementation trick.

3. PROBLEM ANALYSIS

3.1. PROBLEM STATEMENT

Our main intention is to solve the checkerboard problem, which is actually caused due to the absence of any direct relationship amongst the adjacent pixels on the resulting featured map. It can be solved by using Pixel Transposed Convolutional Network.

3.2. EXISTING SYSTEM

Transposed convolutional layers had been usually utilized in the deep learning models which has the requirement of up-sampling of featured maps, consisting of generative models and encoder-decoder architectures. Even though tcls are much capable of generating large featured maps from the one that are small, they be afflicted by the checkerboard problem. This will significantly limit the competencies of the deep leaning model's in producing photographically-realistic images and generate smoother outputs on semantic image segmentation. Till today, there is either little or no efforts that have been dedicated in the direction of enhancing the transposed convolutional operations.

3.3. PROPOSED SYSTEM

We are going to introduce a simple, and an effective way with better efficiency, to solve the checkerboard artifacts that are being suffered by tcl operations. The up-sampled featured map which is obtained by TCL can be considered to be as the outcome of periodic shuffling of more than one intermediate featured map obtained from the input featured map by performing independent convolution. As a result, the adjoining pixels on the resultant featured map have no direct relation, and hence leads to checkerboard problem. To solve this, we introduce the operations for pixel transposed convolution to be used in PixelTCL.

4. IMPLEMENTATION

4.1. SYSTEM REQUIREMENTS

- Python 3.8 C
- Anaconda
- Jupyter Notebook (or)
- Google colab
- numpy==1.19.2
- tensorflow==1.5.0
- h5py==2.10.0
- progressbar==3.37.1
- PIL==8.2.0
- argparse==1.1
- scipy==1.6.2
- imageio==2.9.0

4.2. DATASET

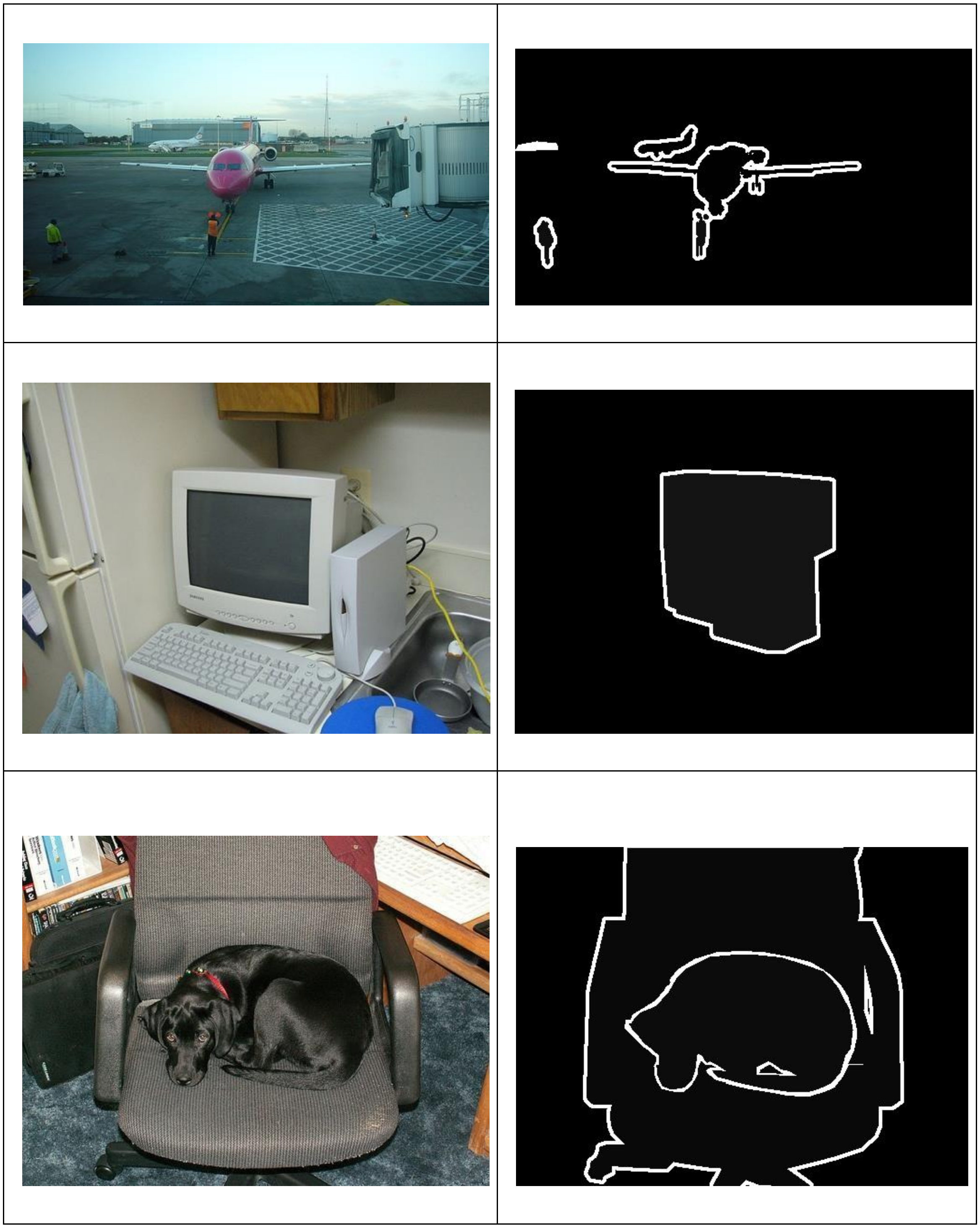
In this study, we are using the PASCAL 2012 segmentation dataset for the purpose of evaluating the pixel transposed convolutional techniques that are proposed, in semantic image segmentation tasks. For each datasets, the images are then resized for the purpose of batch training.

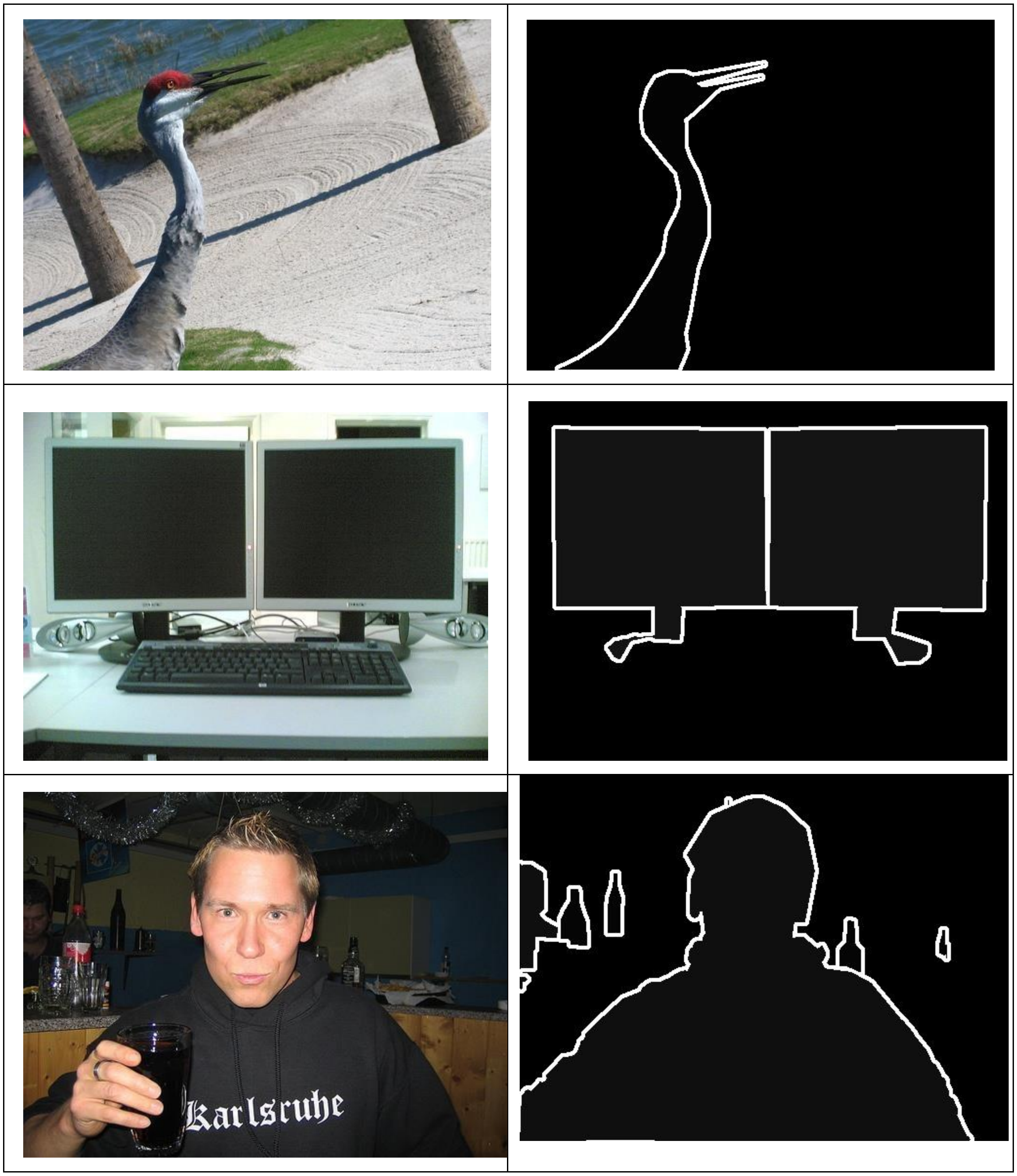
PASCAL 2012 segmentation dataset:

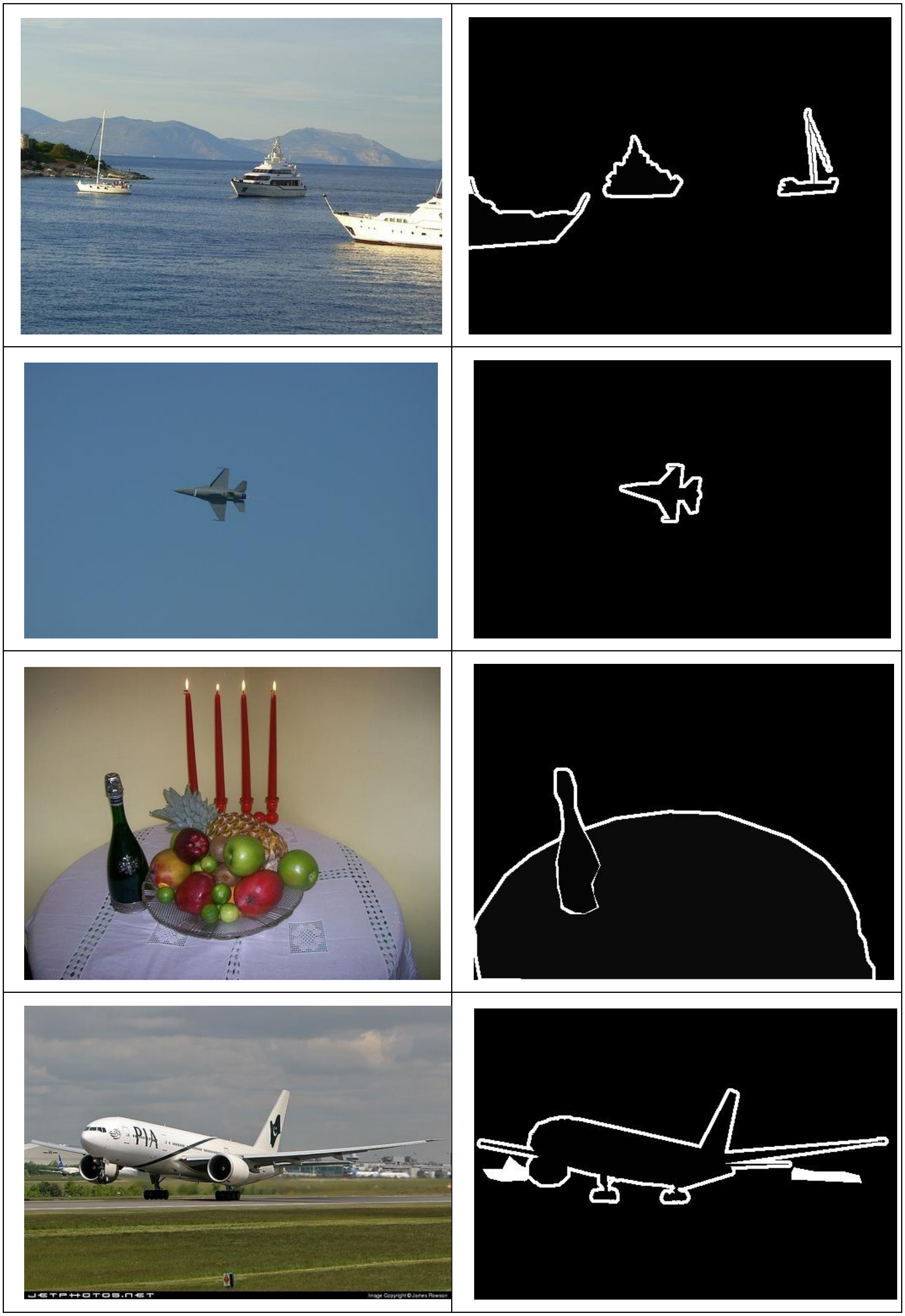
- **SEGMENTATION:** The 2012 dataset consists of images from 2008-2011 for which additional segmentations had been prepared. As in previous years the task to training/check units has been maintained. The overall variety of images with segmentation has been extended from 7,062 to 9,993.

Sample images from the dataset is provided below:

- Original picture(left)
- Object detection(right)




Karlsruhe


PIA
JETPHOTOS.NET

4.3. TRANSPOSED COVOLUTIONAL LAYER

TCLs have been extensively utilized in various deep learning models for a lot of applications which includes semantic image segmentation and generative models. A wide number of encoding-decoding architectures use TCLs in decoding for the purpose of up sampling. A way to understand TCL functions is that the output of the up-sampled featured map is generated by periodically shuffling two or more intermediate featured maps which are produced by applying multiple convolutional operations on the input featured maps.

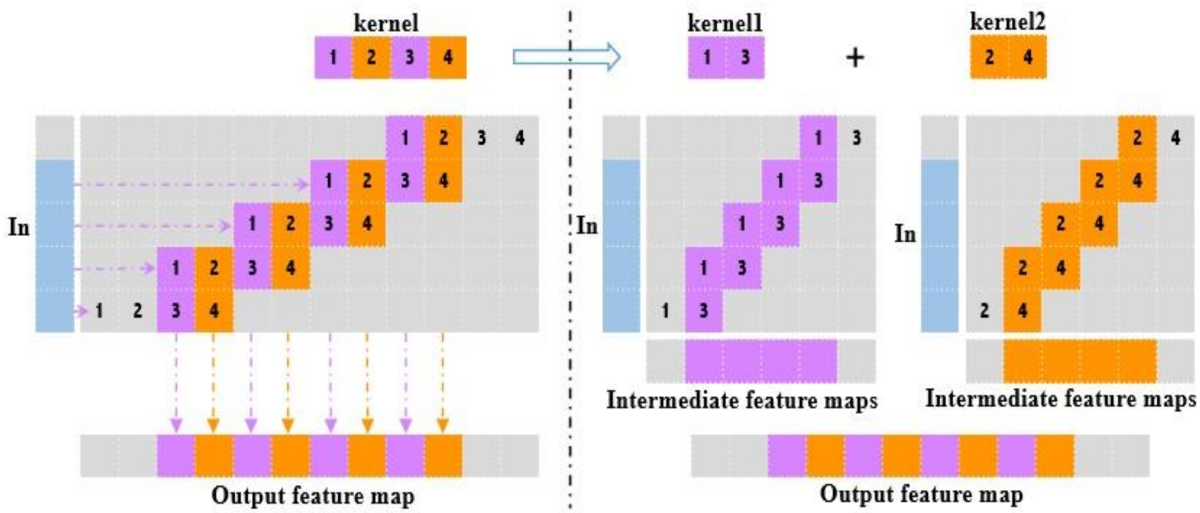


Fig. This figure shows an illustration of TCL operation in 1-Dimension. Here, we have a 4x1 feature map being up-sampled into a 8x1 feature map. Therefore, sum of all the values of each column result as an output featured map.

$$
\begin{aligned}
F_1 &= F_{in} \circledast k_1, \\
F_2 &= F_{in} \circledast k_2, \\
F_3 &= F_{in} \circledast k_3, \\
F_4 &= F_{in} \circledast k_4, \\
F_{out} &= F_1 \oplus F_2 \oplus F_3 \oplus F_4,
\end{aligned}
$$

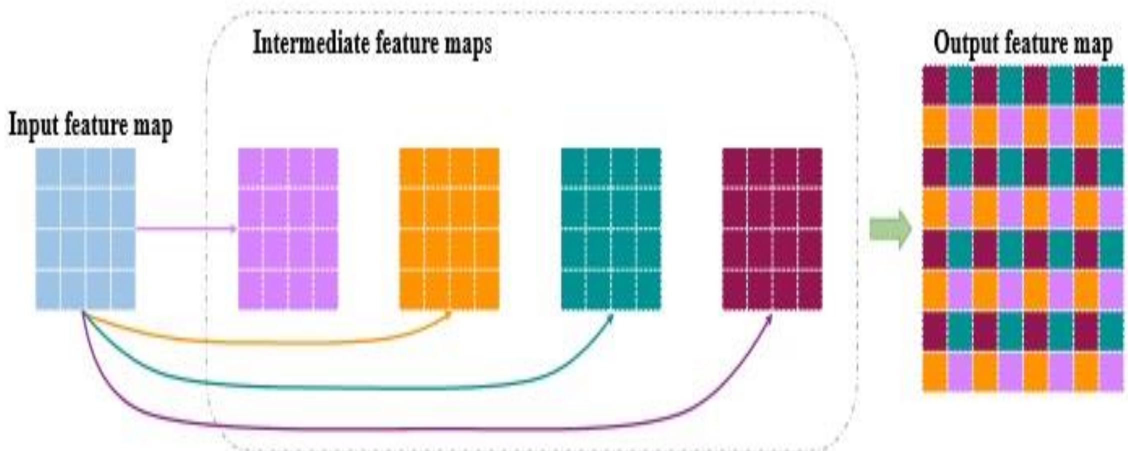


Fig. In the above TCL, 4x4 feature map that is upsampled into a 8x8 feature map. The 4 intermediate feature maps, i.e. orange, purple, red and blue are generated with the help of 4 different convoalutional kernels. These 4 featured maps (which dependent upon the input feature map, having no direct relationship among them) are later shuffled and combined to generate the final output feature map of 8x8.

```
def ipixel_cl(inputs, out_num, kernel_size, scope, activation_fn=tf.nn.relu,
              d_format='NHWC'):
    """
    inputs: input tensor
    out_num: output channel number
    kernel_size: convolutional kernel size
    scope: operation scope
    activation_fn: activation function, could be None if needed
    """
    axis = (d_format.index('H'), d_format.index('W'))
    channel_axis = d_format.index('C')
    conv1 = conv2d(inputs, out_num, kernel_size, scope+'/conv1',
                   stride=2, d_format=d_format)
    dialte1 = dilate_tensor(conv1, axis, (0, 0), scope+'/dialte1')
    shifted_inputs = shift_tensor(inputs, axis, (1, 1), scope+'/shift1')
    conv1_concat = tf.concat(
        [shifted_inputs, dialte1], channel_axis, name=scope+'/concat1')
    conv2 = conv2d(inputs, out_num, kernel_size, scope+'/conv2',
                   stride=2, d_format=d_format)
    dialte2 = dilate_tensor(conv2, axis, (1, 1), scope+'/dialte2')
    conv3 = tf.add_n([dialte1, dialte2], scope+'/add')
    shifted_inputs = shift_tensor(inputs, axis, 1, 0, scope+'/shift2')
    conv2_concat = tf.concat(
        [shifted_inputs, conv3], channel_axis, name=scope+'/concat2')
    conv4 = conv2d(inputs, out_num, kernel_size, scope+'/conv4',
                   stride=2, d_format=d_format)
    dialte3 = dilate_tensor(conv4, axis, (1, 0), scope+'/dialte3')
    shifted_inputs = shift_tensor(inputs, axis, 0, 1, scope+'/shift3')
    conv2_concat = tf.concat(
        [shifted_inputs, conv3], channel_axis, name=scope+'/concat3')
    conv5 = conv2d(inputs, out_num, kernel_size, scope+'/conv5',
                   stride=2, d_format=d_format)
    dialte4 = dilate_tensor(conv5, axis, (0, 1), scope+'/dialte4')
    outputs = tf.add_n([dialte1, dialte2, dialte3, dialte4], scope+'/add')
    if activation_fn:
        outputs = activation_fn(outputs)
    return outputs
```

4.4. iPIXEL-TRANSPOSED CONVOLUTIONAL LAYER

In iPixelTCL, we upload dependencies amongst intermediate featured maps that have been generated, making adjoining pixels on the final resultant featured maps being related to one another directly. In this process, the statistics of the existing input featured map is again and again used whilst producing intermediate featured maps. When producing the intermediate featured maps, in formation from each the input featured map and previous intermediate featured maps is used. As the preceding intermediate featured maps are already included in the statistics of the input featured map, the dependencies on the input featured map can be removed. By removing such type of dependencies for a few number of intermediate featured maps can't only enhance the efficiency of computing however additionally reducing the variety of training parameters in deep learning models.

$$
\begin{aligned}
F_1 &= F_{in} \circledast k_1, \\
F_2 &= [F_{in}, F_1] \circledast k_2, \\
F_3 &= [F_{in}, F_1, F_2] \circledast k_3, \\
F_4 &= [F_{in}, F_1, F_2, F_3] \circledast k_4, \\
F_{out} &= F_1 \oplus F_2 \oplus F_3 \oplus F_4,
\end{aligned}
$$

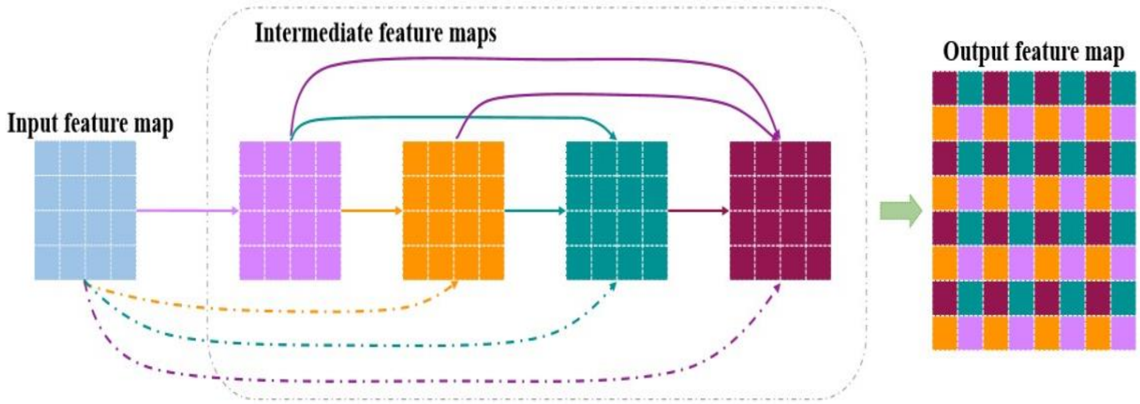


Fig. The above figure shows an illustration of 2-Dimension TCL opertion. Here, a 4x4 featured map is upsampled into a 8x8 feature map. These 4 intermediate feature maps, i.e. orange, purple, red and blue are generated with the help of 4 different convoalutional kernels. These 4 featured maps (which are dependent upon the input feature map, having no direct relationship among them) are later shuffled and combined to generate the final output feature map of 8x8. (iPxel TCL)

```
def ipixel_dcl(inputs, out_num, kernel_size, scope, activation_fn=tf.nn.relu,
               d_format='NHWC'):
    """
    inputs: input tensor
    out_num: output channel number
    kernel_size: convolutional kernel size
    scope: operation scope
    activation_fn: activation function, could be None if needed
    """
    axis = (d_format.index('H'), d_format.index('W'))
    channel_axis = d_format.index('C')
    conv1 = conv2d(inputs, out_num, kernel_size,
                   scope+'/conv1', d_format=d_format)
    conv1_concat = tf.concat(
        [inputs, conv1], channel_axis, name=scope+'/concat1')
    conv2 = conv2d(conv1_concat, out_num, kernel_size,
                   scope+'/conv2', d_format=d_format)
    conv2_concat = tf.concat(
        [conv1_concat, conv2], channel_axis, name=scope+'/concat2')
    conv3 = conv2d(conv2_concat, 2*out_num, kernel_size,
                   scope+'/conv3', d_format=d_format)
    conv4, conv5 = tf.split(conv3, 2, channel_axis, name=scope+'/split')
    dialte1 = dilate_tensor(conv1, axis, (0, 0), scope+'/dialte1')
    dialte2 = dilate_tensor(conv2, axis, (1, 1), scope+'/dialte2')
    dialte3 = dilate_tensor(conv4, axis, (1, 0), scope+'/dialte3')
    dialte4 = dilate_tensor(conv5, axis, (0, 1), scope+'/dialte4')
    outputs = tf.add_n([dialte1, dialte2, dialte3, dialte4], scope+'/add')
    if activation_fn:
        outputs = activation_fn(outputs)
    return outputs
```

4.5. PIXEL-TRANSPOSED CONVOLUTIONAL LAYER

Pixel TCLs can be implemented to update any TCLs in diverse fashions concerning up-sampling operations consisting of U-Net, VAEs, GANs. By changing transposed convolutional layers with Pixel TCL, transposed convolutional networks come to be pixel transposed convolutional networks (PixelTCN). In U-Net, for semanticimage segmentation, PixelTCLs may be used for the purpose of up-sampling into a higer resolution ones from lower resolution featured maps.

$$\begin{aligned} F_1 &= F_{in} \circledast k_1, \\ F_2 &= F_1 \circledast k_2, \\ F_3 &= [F_1, F_2] \circledast k_3, \\ F_4 &= [F_1, F_2, F_3] \circledast k_4, \\ F_{out} &= F_1 \oplus F_2 \oplus F_3 \oplus F_4. \end{aligned}$$

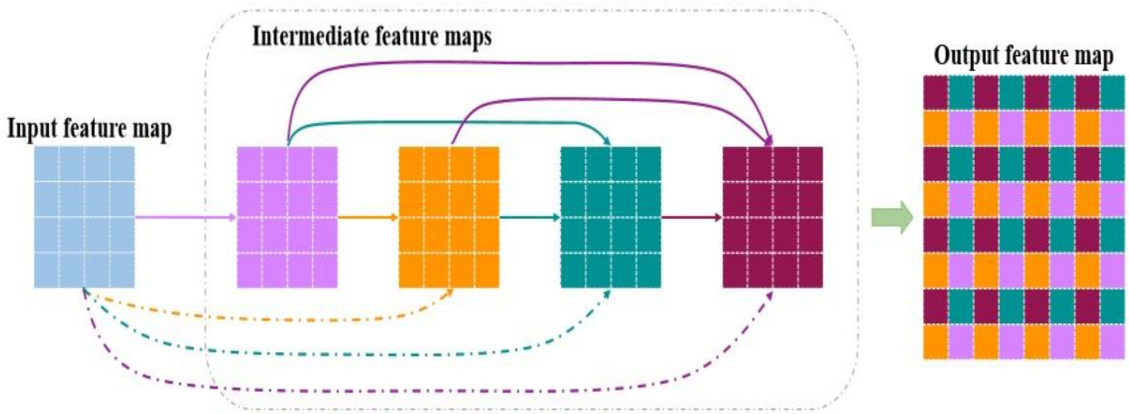


Fig. The above figure shows an illustration of Pixel Transposed convolutional Layer. The 4 intermediate feature maps, i.e. orange, purple, red and blue are generated with the help of 4 different convoalutional kernels. These 4 featured maps (which dependant upon the input feature map, having no direct relationship among them) are later shuffled and combined to generate the final output feature map of 8x8. There were additional dependancies among intermediate feature maps. We transport a step that is similar and allowsnly 1 initial intermediate feature map to depend on the input feature map. This helps us in increasing the Pixel TCL's efficiency. The dashes represent the connections which have been removed to avoid repeatition of the effect of the input feature map. Finally, we have only one input feature map that is prodoced from the actual input and the other maps are not directly dependant on the input.

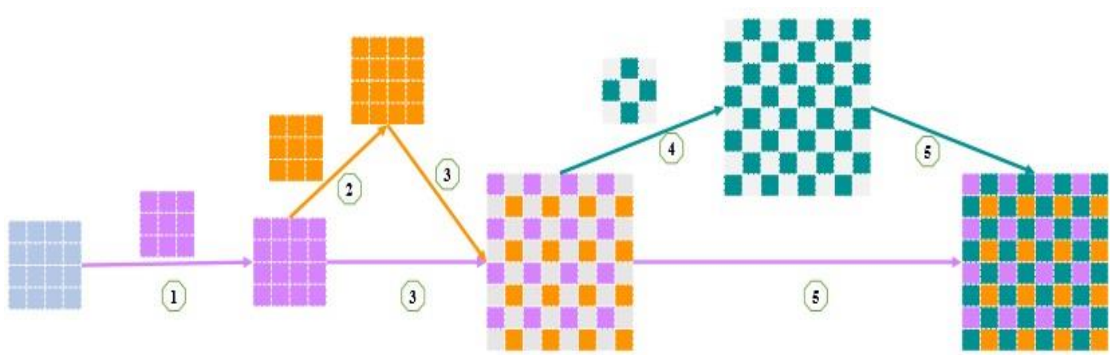


Fig. The above figure shows an illustration of a very effective implementation of Pixel

TCL. Here, a 4x4 featured map is upsampled into a 8x8 feature map. These 4 intermediate feature maps, i.e. orange, purple, red and blue are generated with the help of 4 different convoalutional kernels. 3x3 convolution operation is used to generate the purple featured map from its input featured map. To produce an orange featured map, 3x3 cnvolution operation is applied on the purple featured map. These two maps are further combined together in order to produce a large featured map. A mask of 3x3 can be applied among the remaining intermediate featured maps instead of separate 3x3 convolution operations as there is no direct relationship amongst them. At last, these two large featured maps produced are added together in order to obtain the final output featured map.

```
def pixel_dcl(inputs, out_num, kernel_size, scope, activation_fn=tf.nn.relu,
              d_format='NHWC'):
    """
    inputs: input tensor
    out_num: output channel number
    kernel_size: convolutional kernel size
    scope: operation scope
    activation_fn: activation function, could be None if needed
    """
    axis = (d_format.index('H'), d_format.index('W'))
    conv0 = conv2d(inputs, out_num, kernel_size,
                   scope+'/conv0', d_format=d_format)
    conv1 = conv2d(conv0, out_num, kernel_size,
                   scope+'/conv1', d_format=d_format)
    dilated_conv0 = dilate_tensor(conv0, axis, (0, 0), scope+'/dialte_conv0')
    dilated_conv1 = dilate_tensor(conv1, axis, (1, 1), scope+'/dialte_conv1')
    conv1 = tf.add(dilated_conv0, dilated_conv1, scope+'/add1')
    with tf.variable_scope(scope+'/conv2'):
        shape = list(kernel_size) + [out_num, out_num]
        weights = tf.get_variable(
            'weights', shape, initializer=tf.truncated_normal_initializer())
        weights = tf.multiply(weights, get_mask(shape, scope))
        strides = [1, 1, 1, 1]
        conv2 = tf.nn.conv2d(conv1, weights, strides, padding='SAME',
                             data_format=d_format)
    outputs = tf.add(conv1, conv2, name=scope+'/add2')
    if activation_fn:
        outputs = activation_fn(outputs)
    return outputs
```

4.6. U-NET ARCHITECTURE

The main intention behind semantic image segmentation is to label all the pixels of an existing image which has a corresponding class that defines what's represented. Prediction for each and every pixel in the image, also called as dense prediction. The output which is predicted in semantic image segmentation labels, bounding field parameters. The output is an image with higher resolution (that is same size/shape as the input image) where each and every pixel is classified into a specific class.

Applications of u-net:

- Autonomous vehicles
- Bio clinical image diagnosis
- Geo Sensing

Transposed convolution is likewise referred to as as deconvolution in a few cases. It is a method to carry out up-sampling of an image with learning parameters. Transposed convolutional layer is precisely the alternative technique of that of a regular convolution i.e., the volume of the input is an image with low resolution and the volume of the output is an image with high resolution for transposed convolution, while the volume of the input is an image with high resolution and the volume of the output is an image with a low resolution for a regular convolution.

A regular convolutional layer may be expressed as a matrix multiplication of input image and a filter in order to produce the output image. Transpose of the filtered matrix, can ensure us to reverse the convolutional technique and obtain a transposed convolution network.

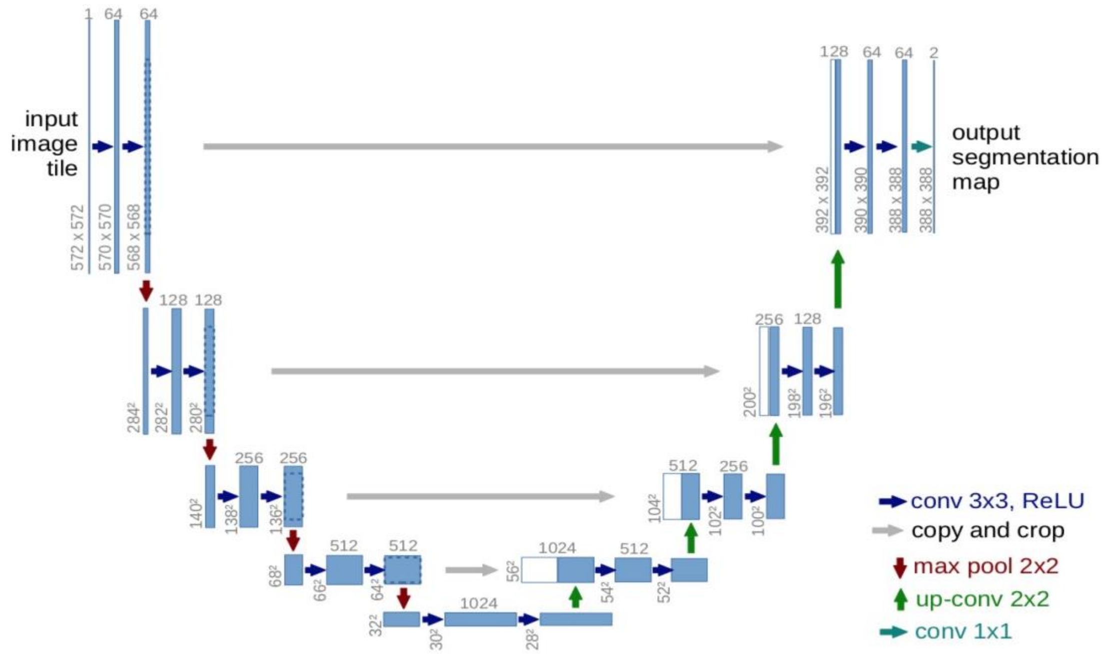


Fig. The above figure is an illustration of the U-NET architecture.

In order to recollect the necessary point:

- Receptive field or context
- Convolutional, pooling operations down pattern the photograph, i.e. convert a better decision photograph to a decrease decision photograph
- Max Pooling operation typically facilitates in understanding "WHAT" is there in an image via way of means of increasing the receptive field. Still, it seems to lose the information of "WHERE" the objects are.
- In semantic image segmentation, we should know "WHAT" is present in the image as well as "WHERE" is it present. So, we need to come up with a way in order to upsample an image from lower resolution to higher resolution that facilitates us to recollect the "WHERE" records.
- Transposed Convolution is one of the most desired and taken into consideration to carry out an up-sampling, which essentially tries to learn the parameters via the backpropagation in order to transform from a lower resolution image to a higher resolution image.

4.7. CODE IMPLEMENTATION

- **h5_util:** This module is used to convert data files to h5 database to facilitate training
- **img_util:**
- **data_reader:** This module provides three data readers:
- **Directly from file**
- **From h5 database**
- **Use channel**
- h5 database is recommended since it could enable data feeding speed
- **pixel_dcn:** This module releases all the three models
- **pixel_dcl:** realizes Pixel Deconvolutional Layer
- **ipixel_dcl:** releazes Input Pixel Deconvolutional Layer
- **ipixel_cl:** realizes Input Pixel Convolutional Layer
- **ops:** This module provides some short functions to reduce the volume of the code.

Hence, increases the ease of reusability.

- **network:** This module build a standard U-NET for semantic segmentation.
- **main:** This file provides configuration to build U-NET for semantic segmentation.

4.7.1. h5_util.py

This module is used to convert data files to h5 database to facilitate training.

```python
import numpy as np
import h5py
from progressbar import ProgressBar
from PIL import Image

# IMG_MEAN for Pascal dataset
IMG_MEAN = np.array(
    (122.67891434, 116.66876762, 104.00698793), dtype=np.float32)  # RGB

def read_images(data_list):
    with open(data_list, 'r') as f:
        data = [line.strip("\n").split(' ') for line in f]
    return data
def process_image(image, shape, resize_mode=Image.BILINEAR):
    img = Image.open(image)
    img = img.resize(shape, resize_mode)
    img.load()
    img = np.asarray(img, dtype="float32")
    if len(img.shape) < 3:
        return img.T
    else:
        return np.transpose(img, (1,0,2))
def build_h5_dataset(data_dir, list_path, out_dir, shape, name, norm=False):
    images = read_images(list_path)
    images_size = len(images)
    dataset = h5py.File(out_dir+name+'.h5', 'w')
    dataset.create_dataset('X', (images_size, *shape, 3), dtype='f')
    dataset.create_dataset('Y', (images_size, *shape), dtype='f')
    pbar = ProgressBar()
    for index, (image, label) in pbar(enumerate(images)):
        image = process_image(data_dir+image, shape)
        label = process_image(data_dir+label, shape, Image.NEAREST)
        image -= IMG_MEAN
        image = image / 255. if norm else image
        dataset['X'][index], dataset['Y'][index] = image, label
    dataset.close()
if __name__ == '__main__':
    shape = (256, 256)
    data_dir = './dataset'
    list_dir = './dataset/'
    output_dir = './dataset/'

    data_files = {
        'training': 'train.txt',
        'validation': 'val.txt',
        'testing': 'test.txt'
    }
    for name, list_path in data_files.items():
        build_h5_dataset(data_dir, list_dir+list_path, output_dir, shape, name)
```

4.7.2. img_util.py

```
import os
import scipy
import scipy.misc
import h5py
import numpy as np


def center_crop(image, pre_height, pre_width, height, width):
    h, w = image.shape[:2]
    j, i = int((h - pre_height)/2.), int((w - pre_width)/2.)
    return scipy.misc.imresize(
        image[j:j+pre_height, i:i+pre_width], [height, width])


def transform(image, pre_height, pre_width, height, width, is_crop):
    if is_crop:
        new_image = center_crop(image, pre_height, pre_width, height, width)
    else:
        new_image = scipy.misc.imresize(image, [height, width])
    return np.array(new_image)/127.5 - 1.


def imread(path, is_grayscale=False):
    if is_grayscale:
        return scipy.misc.imread(path, flatten=True).astype(np.float)
    return scipy.misc.imread(path).astype(np.float)


def imsave(image, path):
    label_colours = [
        (0,0,0),
        # 0=background
        (128,0,0),(0,128,0),(128,128,0),(0,0,128),(128,0,128),
        # 1=aeroplane, 2=bicycle, 3=bird, 4=boat, 5=bottle
        (0,128,128),(128,128,128),(64,0,0),(192,0,0),(64,128,0),
        # 6=bus, 7=car, 8=cat, 9=chair, 10=cow
        (192,128,0),(64,0,128),(192,0,128),(64,128,128),(192,128,128),
        # 11=diningtable, 12=dog, 13=horse, 14=motorbike, 15=person
        (0,64,0),(128,64,0),(0,192,0),(128,192,0),(0,64,128)]
        # 16=potted plant, 17=sheep, 18=sofa, 19=train, 20=tv/monitor
    images = np.ones(list(image.shape)+[3])
    for j_, j in enumerate(image):
        for k_, k in enumerate(j):
            if k < 21:
                images[j_, k_] = label_colours[int(k)]
    scipy.misc.imsave(path, images)


def get_images(paths, pre_height, pre_width, height, width,
               is_crop=False, is_grayscale=False):
    images = []
    for path in paths:
        image = imread(path, is_grayscale)
        new_image = transform(
            image, pre_height, pre_width, height, width, is_crop)
        images.append(new_image)
    return np.array(images).astype(np.float32)

def save_data(path, image_folder='./images/', label_folder='./labels/'):
    if not os.path.exists(image_folder):
        os.makedirs(image_folder)
    if not os.path.exists(label_folder):
        os.makedirs(label_folder)
    data_file = h5py.File(path, 'r')
    for index in range(data_file['X'].shape[0]):
        scipy.misc.imsave(image_folder+str(index)+'.png', data_file['X'][index])
        imsave(data_file['Y'][index], label_folder+str(index)+'.png')
```

```
def compose_images(ids, wides, folders, name):
    result_folder = './results/'
    if not os.path.exists(result_folder):
        os.makedirs(result_folder)
    id_imgs = []
    for i, index in enumerate(ids):
        imgs = []
        for folder in folders:
            path = folder + str(index) +'.png'
            cur_img = scipy.misc.imread(path).astype(np.float)
            cur_img = scipy.misc.imresize(cur_img, [256, int(256*wides[i])])
            imgs.append(cur_img)
            imgs.append(np.ones([3]+list(cur_img.shape)[1:])*255)
        img = np.concatenate(imgs[:-1], axis=0)
        id_imgs.append(img)
        id_imgs.append(np.ones((img.shape[0], 2, img.shape[2]))*255)
    id_img = np.concatenate(id_imgs[:-1], axis=1)
    scipy.misc.imsave(result_folder+name+'.png', id_img)

if __name__ == '__main__':
    folders = ['./images/', './labels/', './samples3/', './samples1/', './samples2/']
    pre_folders = ['./images/', './labels/', './samples3/', './samples2/']
    # folders = ['./images/', './labels/', './samples/']
    ids = [214, 238, 720, 256, 276,277,298,480,571,920,1017,1422]
    wides = [1]*len(ids)
    ids_pre = [15,153,160,534,906]
    pre_wides = [1.3, 1.2, 1.8, 1.1, 1.1]
    compose_images(ids_pre, pre_wides, pre_folders, 'pre_result')
    compose_images(ids, wides, folders, 'result')
```

4.7.3. data_reader.py

This module provides three data readers

```
import glob
import h5py
import random
import tensorflow as tf
import numpy as np
from .img_utils import get_images

class FileDataReader(object):

    def __init__(self, data_dir, input_height, input_width, height, width,
                 batch_size):
        self.data_dir = data_dir
        self.input_height, self.input_width = input_height, input_width
        self.height, self.width = height, width
        self.batch_size = batch_size
        self.image_files = glob.glob(data_dir+'*')

    def next_batch(self, batch_size):
        sample_files = np.random.choice(self.image_files, batch_size)
        images = get_images(
            sample_files, self.input_height, self.input_width,
            self.height, self.width)
        return images
```

```
class H5DataLoader(object):

    def __init__(self, data_path, is_train=True):
        self.is_train = is_train
        data_file = h5py.File(data_path, 'r')
        self.images, self.labels = data_file['X'], data_file['Y']
        self.gen_indexes()

    def gen_indexes(self):
        if self.is_train:
            self.indexes = np.random.permutation(range(self.images.shape[0]))
        else:
            self.indexes = np.array(range(self.images.shape[0]))
        self.cur_index = 0

    def next_batch(self, batch_size):
        next_index = self.cur_index+batch_size
        cur_indexes = list(self.indexes[self.cur_index:next_index])
        self.cur_index = next_index
        if len(cur_indexes) < batch_size and self.is_train:
            self.gen_indexes()
            return self.next_batch(batch_size)
        cur_indexes.sort()
        return self.images[cur_indexes], self.labels[cur_indexes]
```

4.7.4. pixel_dcn.py

This module releases all the three models

```
import tensorflow as tf
import numpy as np

def pixel_dcl(inputs, out_num, kernel_size, scope, activation_fn=tf.nn.relu,
              d_format='NHWC'):
    """
    inputs: input tensor
    out_num: output channel number
    kernel_size: convolutional kernel size
    scope: operation scope
    activation_fn: activation function, could be None if needed
    """
    axis = (d_format.index('H'), d_format.index('W'))
    conv0 = conv2d(inputs, out_num, kernel_size,
                   scope+'/conv0', d_format=d_format)
    conv1 = conv2d(conv0, out_num, kernel_size,
                   scope+'/conv1', d_format=d_format)
    dilated_conv0 = dilate_tensor(conv0, axis, (0, 0), scope+'/dialte_conv0')
    dilated_conv1 = dilate_tensor(conv1, axis, (1, 1), scope+'/dialte_conv1')
    conv1 = tf.add(dilated_conv0, dilated_conv1, scope+'/add1')
    with tf.variable_scope(scope+'/conv2'):
        shape = list(kernel_size) + [out_num, out_num]
        weights = tf.get_variable(
            'weights', shape, initializer=tf.truncated_normal_initializer())
        weights = tf.multiply(weights, get_mask(shape, scope))
        strides = [1, 1, 1, 1]
        conv2 = tf.nn.conv2d(conv1, weights, strides, padding='SAME',
                             data_format=d_format)
    outputs = tf.add(conv1, conv2, name=scope+'/add2')
    if activation_fn:
        outputs = activation_fn(outputs)
    return outputs
```

```
def ipixel_dcl3d(inputs, out_num, kernel_size, scope, action='concat', activation_fn=tf.nn.relu):
    """
    inputs: input tensor
    out_num: output channel number
    kernel_size: convolutional kernel size
    scope: operation scope
    activation_fn: activation function, could be None if needed
    """
    axis, c_axis = (1, 2, 3), 4  # only support format 'NDHWC'
    conv0 = conv3d(inputs, out_num, kernel_size, scope+'/conv0')
    combine1 = combine([inputs, conv0], action, c_axis, scope+'/combine1')
    conv1 = conv3d(combine1, out_num, kernel_size, scope+'/conv1')
    combine2 = combine([combine1, conv1], action, c_axis, scope+'/combine2')
    conv2 = conv3d(combine2, 3*out_num, kernel_size, scope+'/conv2')
    conv2_list = tf.split(conv2, 3, c_axis, name=scope+'/split1')
    combine3 = combine(conv2_list+[combine2], action, c_axis, scope+'/combine3')
    conv3 = conv3d(combine3, 3*out_num, kernel_size, scope+'/conv3')
    conv3_list = tf.split(conv3, 3, c_axis, name=scope+'/split2')
    dilated_conv0 = dilate_tensor(
        conv0, axis, (0, 0, 0), scope+'/dialte_conv0')
    dilated_conv1 = dilate_tensor(
        conv1, axis, (1, 1, 1), scope+'/dialte_conv1')
    dilated_list = [dilated_conv0, dilated_conv1]
    for index, shifts in enumerate([(1, 1, 0), (1, 0, 1), (0, 1, 1)]):
        dilated_list.append(dilate_tensor(
            conv2_list[index], axis, shifts, scope+'/dialte_conv2_%s' % index))
    for index, shifts in enumerate([(1, 0, 0), (0, 0, 1), (0, 1, 0)]):
        dilated_list.append(dilate_tensor(
            conv3_list[index], axis, shifts, scope+'/dialte_conv3_%s' % index))
    outputs = tf.add_n(dilated_list, name=scope+'/add')
    if activation_fn:
        outputs = activation_fn(outputs)
    return outputs


def pixel_dcl3d(inputs, out_num, kernel_size, scope, action='concat', activation_fn=tf.nn.relu):
    """
    inputs: input tensor
    out_num: output channel number
    kernel_size: convolutional kernel size
    scope: operation scope
    activation_fn: activation function, could be None if needed
    """
    axis, c_axis = (1, 2, 3), 4  # only support format 'NDHWC'
    conv0 = conv3d(inputs, out_num, kernel_size, scope+'/conv0')
    conv1 = conv3d(conv0, out_num, kernel_size, scope+'/conv1')
    combine1 = combine([conv0, conv1], action, c_axis, scope+'/combine1')
    conv2 = conv3d(combine1, 3*out_num, kernel_size, scope+'/conv2')
    conv2_list = tf.split(conv2, 3, c_axis, name=scope+'/split1')
    combine2 = combine([conv0]+conv2_list, action, c_axis, scope+'/combine2')
    conv3 = conv3d(combine2, 3*out_num, kernel_size, scope+'/conv3')
    conv3_list = tf.split(conv3, 3, c_axis, name=scope+'/split2')
    dilated_conv0 = dilate_tensor(
        conv0, axis, (0, 0, 0), scope+'/dialte_conv0')
    dilated_conv1 = dilate_tensor(
        conv1, axis, (1, 1, 1), scope+'/dialte_conv1')
    dilated_list = [dilated_conv0, dilated_conv1]
    for index, shifts in enumerate([(1, 1, 0), (1, 0, 1), (0, 1, 1)]):
        dilated_list.append(dilate_tensor(
            conv2_list[index], axis, shifts, scope+'/dialte_conv2_%s' % index))
    for index, shifts in enumerate([(1, 0, 0), (0, 0, 1), (0, 1, 0)]):
        dilated_list.append(dilate_tensor(
            conv3_list[index], axis, shifts, scope+'/dialte_conv3_%s' % index))
    outputs = tf.add_n(dilated_list, name=scope+'/add')
    if activation_fn:
        outputs = activation_fn(outputs)
    return outputs


def combine(tensors, action, axis, name):
    if action == 'concat':
        return tf.concat(tensors, axis, name=name)
    else:
        return tf.add_n(tensors, name=name)
```

```python
def ipixel_dcl(inputs, out_num, kernel_size, scope, activation_fn=tf.nn.relu,
               d_format='NHWC'):
    """
    inputs: input tensor
    out_num: output channel number
    kernel_size: convolutional kernel size
    scope: operation scope
    activation_fn: activation function, could be None if needed
    """
    axis = (d_format.index('H'), d_format.index('W'))
    channel_axis = d_format.index('C')
    conv1 = conv2d(inputs, out_num, kernel_size,
                   scope+'/conv1', d_format=d_format)
    conv1_concat = tf.concat(
        [inputs, conv1], channel_axis, name=scope+'/concat1')
    conv2 = conv2d(conv1_concat, out_num, kernel_size,
                   scope+'/conv2', d_format=d_format)
    conv2_concat = tf.concat(
        [conv1_concat, conv2], channel_axis, name=scope+'/concat2')
    conv3 = conv2d(conv2_concat, 2*out_num, kernel_size,
                   scope+'/conv3', d_format=d_format)
    conv4, conv5 = tf.split(conv3, 2, channel_axis, name=scope+'/split')
    dialte1 = dilate_tensor(conv1, axis, (0, 0), scope+'/dialte1')
    dialte2 = dilate_tensor(conv2, axis, (1, 1), scope+'/dialte2')
    dialte3 = dilate_tensor(conv4, axis, (1, 0), scope+'/dialte3')
    dialte4 = dilate_tensor(conv5, axis, (0, 1), scope+'/dialte4')
    outputs = tf.add_n([dialte1, dialte2, dialte3, dialte4], scope+'/add')
    if activation_fn:
        outputs = activation_fn(outputs)
    return outputs


def ipixel_cl(inputs, out_num, kernel_size, scope, activation_fn=tf.nn.relu,
              d_format='NHWC'):
    """
    inputs: input tensor
    out_num: output channel number
    kernel_size: convolutional kernel size
    scope: operation scope
    activation_fn: activation function, could be None if needed
    """
    axis = (d_format.index('H'), d_format.index('W'))
    channel_axis = d_format.index('C')
    conv1 = conv2d(inputs, out_num, kernel_size, scope+'/conv1',
                   stride=2, d_format=d_format)
    dialte1 = dilate_tensor(conv1, axis, (0, 0), scope+'/dialte1')
    shifted_inputs = shift_tensor(inputs, axis, (1, 1), scope+'/shift1')
    conv1_concat = tf.concat(
        [shifted_inputs, dialte1], channel_axis, name=scope+'/concat1')
    conv2 = conv2d(inputs, out_num, kernel_size, scope+'/conv2',
                   stride=2, d_format=d_format)
    dialte2 = dilate_tensor(conv2, axis, (1, 1), scope+'/dialte2')
    conv3 = tf.add_n([dialte1, dialte2], scope+'/add')
    shifted_inputs = shift_tensor(inputs, axis, 1, 0, scope+'/shift2')
    conv2_concat = tf.concat(
        [shifted_inputs, conv3], channel_axis, name=scope+'/concat2')
    conv4 = conv2d(inputs, out_num, kernel_size, scope+'/conv4',
                   stride=2, d_format=d_format)
    dialte3 = dilate_tensor(conv4, axis, (1, 0), scope+'/dialte3')
    shifted_inputs = shift_tensor(inputs, axis, 0, 1, scope+'/shift3')
    conv2_concat = tf.concat(
        [shifted_inputs, conv3], channel_axis, name=scope+'/concat3')
    conv5 = conv2d(inputs, out_num, kernel_size, scope+'/conv5',
                   stride=2, d_format=d_format)
    dialte4 = dilate_tensor(conv5, axis, (0, 1), scope+'/dialte4')
    outputs = tf.add_n([dialte1, dialte2, dialte3, dialte4], scope+'/add')
    if activation_fn:
        outputs = activation_fn(outputs)
    return outputs
```

```python
def conv2d(inputs, out_num, kernel_size, scope, stride=1, d_format='NHWC'):
    outputs = tf.contrib.layers.conv2d(
        inputs, out_num, kernel_size, scope=scope, stride=stride,
        data_format=d_format, activation_fn=None, biases_initializer=None)
    return outputs


def conv3d(inputs, out_num, kernel_size, scope):
    shape = list(kernel_size) + [inputs.shape[-1].value, out_num]
    weights = tf.get_variable(
        scope+'/conv/weights', shape, initializer=tf.truncated_normal_initializer())
    outputs = tf.nn.conv3d(
        inputs, weights, (1, 1, 1, 1, 1), padding='SAME', name=scope+'/conv')
    return outputs


def get_mask(shape, scope):
    new_shape = (np.prod(shape[:-2]), shape[-2], shape[-1])
    mask = np.ones(new_shape, dtype=np.float32)
    for i in range(0, new_shape[0], 2):
        mask[i, :, :] = 0
    mask = np.reshape(mask, shape, 'F')
    return tf.constant(mask, dtype=tf.float32, name=scope+'/mask')


def dilate_tensor(inputs, axes, shifts, scope):
    for index, axis in enumerate(axes):
        eles = tf.unstack(inputs, axis=axis, name=scope+'/unstack%s' % index)
        zeros = tf.zeros_like(
            eles[0], dtype=tf.float32, name=scope+'/zeros%s' % index)
        for ele_index in range(len(eles), 0, -1):
            eles.insert(ele_index-shifts[index], zeros)
        inputs = tf.stack(eles, axis=axis, name=scope+'/stack%s' % index)
    return inputs


def shift_tensor(inputs, axes, row_shift, column_shift, scope):
    if row_shift:
        rows = tf.unstack(inputs, axis=axes[0], name=scope+'/rowsunstack')
        row_zeros = tf.zeros_like(
            rows[0], dtype=tf.float32, name=scope+'/rowzeros')
        rows = rows[row_shift:] + [row_zeros]*row_shift
        inputs = tf.stack(rows, axis=axes[0], name=scope+'/rowsstack')
    if column_shift:
        columns = tf.unstack(
            inputs, axis=axes[1], name=scope+'/columnsunstack')
        columns_zeros = tf.zeros_like(
            columns[0], dtype=tf.float32, name=scope+'/columnzeros')
        columns = columns[column_shift:] + [columns_zeros]*column_shift
        inputs = tf.stack(columns, axis=axes[1], name=scope+'/columnsstack')
    return inputs
```

4.7.5. ops.py

This module provides some functions to reduce the volume of the code.

```python
import tensorflow as tf
import pixel_dcn

def pixel_dcl(inputs, out_num, kernel_size, scope, data_type='2D', action='add'):
    if data_type == '2D':
        outs = pixel_dcn.pixel_dcl(inputs, out_num, kernel_size, scope, None)
    else:
        outs = pixel_dcn.pixel_dcl3d(inputs, out_num, kernel_size, scope, action, None)
    return tf.contrib.layers.batch_norm(
        outs, decay=0.9, epsilon=1e-5, activation_fn=tf.nn.relu,
        updates_collections=None, scope=scope+'/batch_norm')

def ipixel_cl(inputs, out_num, kernel_size, scope, data_type='2D'):
    # only support 2d
    outputs = pixel_dcn.ipixel_cl(inputs, out_num, kernel_size, scope, None)
    return tf.contrib.layers.batch_norm(
        outputs, decay=0.9, epsilon=1e-5, activation_fn=tf.nn.relu,
        updates_collections=None, scope=scope+'/batch_norm')
```

```
def ipixel_dcl(inputs, out_num, kernel_size, scope, data_type='2D', action='add'):
    if data_type == '2D':
        outs = pixel_dcn.ipixel_dcl(inputs, out_num, kernel_size, scope, None)
    else:
        outs = pixel_dcn.ipixel_dcl3d(
            inputs, out_num, kernel_size, scope, action, None)
    return tf.contrib.layers.batch_norm(
        outs, decay=0.9, epsilon=1e-5, activation_fn=tf.nn.relu,
        updates_collections=None, scope=scope+'/batch_norm')

def conv(inputs, out_num, kernel_size, scope, data_type='2D', norm=True):
    if data_type == '2D':
        outs = tf.layers.conv2d(
            inputs, out_num, kernel_size, padding='same', name=scope+'/conv',
            kernel_initializer=tf.truncated_normal_initializer)
    else:
        shape = list(kernel_size) + [inputs.shape[-1].value, out_num]
        weights = tf.get_variable(
            scope+'/conv/weights', shape,
            initializer=tf.truncated_normal_initializer())
        outs = tf.nn.conv3d(
            inputs, weights, (1, 1, 1, 1, 1), padding='SAME',
            name=scope+'/conv')
    if norm:
        return tf.contrib.layers.batch_norm(
            outs, decay=0.9, epsilon=1e-5, activation_fn=tf.nn.relu,
            updates_collections=None, scope=scope+'/batch_norm')
    else:
        return tf.contrib.layers.batch_norm(
            outs, decay=0.9, epsilon=1e-5, activation_fn=None,
            updates_collections=None, scope=scope+'/batch_norm')

def deconv(inputs, out_num, kernel_size, scope, data_type='2D', **kws):
    if data_type == '2D':
        outs = tf.layers.conv2d_transpose(
            inputs, out_num, kernel_size, (2, 2), padding='same', name=scope,
            kernel_initializer=tf.truncated_normal_initializer)
    else:
        shape = list(kernel_size) + [out_num, out_num]
        input_shape = inputs.shape.as_list()
        out_shape = [input_shape[0]] + \
            list(map(lambda x: x*2, input_shape[1:-1])) + [out_num]
        weights = tf.get_variable(
            scope+'/deconv/weights', shape,
            initializer=tf.truncated_normal_initializer())
        outs = tf.nn.conv3d_transpose(
            inputs, weights, out_shape, (1, 2, 2, 2, 1), name=scope+'/deconv')
    return tf.contrib.layers.batch_norm(
        outs, decay=0.9, epsilon=1e-5, activation_fn=tf.nn.relu,
        updates_collections=None, scope=scope+'/batch_norm')

def pool(inputs, kernel_size, scope, data_type='2D'):
    if data_type == '2D':
        return tf.layers.max_pooling2d(inputs, kernel_size, (2, 2), name=scope)
    return tf.layers.max_pooling3d(inputs, kernel_size, (2, 2, 2), name=scope)
```

4.7.6. network.py

This module build a standard U-NET for semantic segmentation.

```
import os
import numpy as np
import tensorflow as tf
import ops

class PixelDCN(object):

    def __init__(self, sess, conf):
        self.sess = sess
        self.conf = conf
        self.def_params()
        if not os.path.exists(conf.modeldir):
            os.makedirs(conf.modeldir)
        if not os.path.exists(conf.logdir):
            os.makedirs(conf.logdir)
        if not os.path.exists(conf.sampledir):
            os.makedirs(conf.sampledir)
        self.configure_networks()
        self.train_summary = self.config_summary('train')
        self.valid_summary = self.config_summary('valid')
```

```python
def def_params(self):
    self.data_format = 'NHWC'
    if self.conf.data_type == '3D':
        self.conv_size = (3, 3, 3)
        self.pool_size = (2, 2, 2)
        self.axis, self.channel_axis = (1, 2, 3), 4
        self.input_shape = [
            self.conf.batch, self.conf.depth, self.conf.height,
            self.conf.width, self.conf.channel]
        self.output_shape = [
            self.conf.batch, self.conf.depth, self.conf.height,
            self.conf.width]
    else:
        self.conv_size = (3, 3)
        self.pool_size = (2, 2)
        self.axis, self.channel_axis = (1, 2), 3
        self.input_shape = [
            self.conf.batch, self.conf.height, self.conf.width,
            self.conf.channel]
        self.output_shape = [
            self.conf.batch, self.conf.height, self.conf.width]

def configure_networks(self):
    self.build_network()
    optimizer = tf.train.AdamOptimizer(self.conf.learning_rate)
    self.train_op = optimizer.minimize(self.loss_op, name='train_op')
    tf.set_random_seed(self.conf.random_seed)
    self.sess.run(tf.global_variables_initializer())
    trainable_vars = tf.trainable_variables()
    self.saver = tf.train.Saver(var_list=trainable_vars, max_to_keep=0)
    self.writer = tf.summary.FileWriter(self.conf.logdir, self.sess.graph)

def build_network(self):
    self.inputs = tf.placeholder(
        tf.float32, self.input_shape, name='inputs')
    self.labels = tf.placeholder(
        tf.int64, self.output_shape, name='labels')
    self.predictions = self.inference(self.inputs)
    self.cal_loss()

def cal_loss(self):
    one_hot_labels = tf.one_hot(
        self.labels, depth=self.conf.class_num,
        axis=self.channel_axis, name='labels/one_hot')
    losses = tf.losses.softmax_cross_entropy(
        one_hot_labels, self.predictions, scope='loss/losses')
    self.loss_op = tf.reduce_mean(losses, name='loss/loss_op')
    self.decoded_preds = tf.argmax(
        self.predictions, self.channel_axis, name='accuracy/decode_pred')
    correct_prediction = tf.equal(
        self.labels, self.decoded_preds,
        name='accuracy/correct_pred')
    self.accuracy_op = tf.reduce_mean(
        tf.cast(correct_prediction, tf.float32, name='accuracy/cast'),
        name='accuracy/accuracy_op')
    # weights = tf.cast(
    #     tf.greater(self.decoded_preds, 0, name='m_iou/greater'),
    #     tf.int32, name='m_iou/weights')
    weights = tf.cast(
        tf.less(self.labels, self.conf.channel, name='m_iou/greater'),
        tf.int64, name='m_iou/weights')
    labels = tf.multiply(self.labels, weights, name='m_iou/mul')
    self.m_iou, self.miou_op = tf.metrics.mean_iou(
        self.labels, self.decoded_preds, self.conf.class_num,
        weights, name='m_iou/m_ious')
```

```
def config_summary(self, name):
    summarys = []
    summarys.append(tf.summary.scalar(name+'/loss', self.loss_op))
    summarys.append(tf.summary.scalar(name+'/accuracy', self.accuracy_op))
    if name == 'valid' and self.conf.data_type == '2D':
        summarys.append(
            tf.summary.image(name+'/input', self.inputs, max_outputs=100))
        summarys.append(
            tf.summary.image(
                name+'/annotation',
                tf.cast(tf.expand_dims(self.labels, -1),
                        tf.float32), max_outputs=100))
        summarys.append(
            tf.summary.image(
                name+'/prediction',
                tf.cast(tf.expand_dims(self.decoded_preds, -1),
                        tf.float32), max_outputs=100))
    summary = tf.summary.merge(summarys)
    return summary

def inference(self, inputs):
    outputs = inputs
    down_outputs = []
    for layer_index in range(self.conf.network_depth-1):
        is_first = True if not layer_index else False
        name = 'down%s' % layer_index
        outputs = self.build_down_block(
            outputs, name, down_outputs, is_first)
    outputs = self.build_bottom_block(outputs, 'bottom')
    for layer_index in range(self.conf.network_depth-2, -1, -1):
        is_final = True if layer_index == 0 else False
        name = 'up%s' % layer_index
        down_inputs = down_outputs[layer_index]
        outputs = self.build_up_block(
            outputs, down_inputs, name, is_final)
    return outputs
def build_down_block(self, inputs, name, down_outputs, first=False):
    out_num = self.conf.start_channel_num if first else 2 * \
        inputs.shape[self.channel_axis].value
    conv1 = ops.conv(inputs, out_num, self.conv_size,
                     name+'/conv1', self.conf.data_type)
    conv2 = ops.conv(conv1, out_num, self.conv_size,
                     name+'/conv2', self.conf.data_type)
    down_outputs.append(conv2)
    pool = ops.pool(conv2, self.pool_size, name +
                    '/pool', self.conf.data_type)
    return pool
def build_bottom_block(self, inputs, name):
    out_num = inputs.shape[self.channel_axis].value
    conv1 = ops.conv(
        inputs, 2*out_num, self.conv_size, name+'/conv1',
        self.conf.data_type)
    conv2 = ops.conv(
        conv1, out_num, self.conv_size, name+'/conv2', self.conf.data_type)
    return conv2

def build_up_block(self, inputs, down_inputs, name, final=False):
    out_num = inputs.shape[self.channel_axis].value
    conv1 = self.deconv_func()(
        inputs, out_num, self.conv_size, name+'/conv1',
        self.conf.data_type, action=self.conf.action)
    conv1 = tf.concat(
        [conv1, down_inputs], self.channel_axis, name=name+'/concat')
    conv2 = self.conv_func()(
        conv1, out_num, self.conv_size, name+'/conv2', self.conf.data_type)
    out_num = self.conf.class_num if final else out_num/2
    conv3 = ops.conv(
        conv2, out_num, self.conv_size, name+'/conv3', self.conf.data_type,
        not final)
    return conv3
```

```
def deconv_func(self):
    return getattr(ops, self.conf.deconv_name)

def conv_func(self):
    return getattr(ops, self.conf.conv_name)

def save_summary(self, summary, step):
    print('---->summarizing', step)
    self.writer.add_summary(summary, step)

def train(self):
    if self.conf.reload_step > 0:
        self.reload(self.conf.reload_step)
    if self.conf.data_type == '2D':
        train_reader = H5DataLoader(
            self.conf.data_dir+self.conf.train_data)
        valid_reader = H5DataLoader(
            self.conf.data_dir+self.conf.valid_data)
    else:
        train_reader = H53DDataLoader(
            self.conf.data_dir+self.conf.train_data, self.input_shape)
        valid_reader = H53DDataLoader(
            self.conf.data_dir+self.conf.valid_data, self.input_shape)
    for epoch_num in range(self.conf.max_step+1):
        if epoch_num and epoch_num % self.conf.test_interval == 0:
            inputs, labels = valid_reader.next_batch(self.conf.batch)
            feed_dict = {self.inputs: inputs,
                         self.labels: labels}
            loss, summary = self.sess.run(
                [self.loss_op, self.valid_summary], feed_dict=feed_dict)
            self.save_summary(summary, epoch_num+self.conf.reload_step)
            print('----testing loss', loss)
        if epoch_num and epoch_num % self.conf.summary_interval == 0:
            inputs, labels = train_reader.next_batch(self.conf.batch)
            feed_dict = {self.inputs: inputs,
                         self.labels: labels}
            loss, _, summary = self.sess.run(
                [self.loss_op, self.train_op, self.train_summary],
                feed_dict=feed_dict)
            self.save_summary(summary, epoch_num+self.conf.reload_step)
        else:
            inputs, labels = train_reader.next_batch(self.conf.batch)
            feed_dict = {self.inputs: inputs,
                         self.labels: labels}
            loss, _ = self.sess.run(
                [self.loss_op, self.train_op], feed_dict=feed_dict)
            print('----training loss', loss)
        if epoch_num and epoch_num % self.conf.save_interval == 0:
            self.save(epoch_num+self.conf.reload_step)


def test(self):
    print('---->testing ', self.conf.test_step)
    if self.conf.test_step > 0:
        self.reload(self.conf.test_step)
    else:
        print("please set a reasonable test_step")
        return
    if self.conf.data_type == '2D':
        test_reader = H5DataLoader(
            self.conf.data_dir+self.conf.test_data, False)
    else:
        test_reader = H53DDataLoader(
            self.conf.data_dir+self.conf.test_data, self.input_shape)
    self.sess.run(tf.local_variables_initializer())
    count = 0
    losses = []
    accuracies = []
    m_ious = []
    while True:
        inputs, labels = test_reader.next_batch(self.conf.batch)
        if inputs.shape[0] < self.conf.batch:
            break
        feed_dict = {self.inputs: inputs, self.labels: labels}
        loss, accuracy, m_iou, _ = self.sess.run(
            [self.loss_op, self.accuracy_op, self.m_iou, self.miou_op],
            feed_dict=feed_dict)
        print('values----->', loss, accuracy, m_iou)
        count += 1
        losses.append(loss)
        accuracies.append(accuracy)
        m_ious.append(m_iou)
    print('Loss: ', np.mean(losses))
    print('Accuracy: ', np.mean(accuracies))
    print('M_iou: ', m_ious[-1])
```

```
def predict(self):
    print('---->predicting ', self.conf.test_step)
    if self.conf.test_step > 0:
        self.reload(self.conf.test_step)
    else:
        print("please set a reasonable test_step")
        return
    if self.conf.data_type == '2D':
        test_reader = H5DataLoader(
            self.conf.data_dir+self.conf.test_data, False)
    else:
        test_reader = H53DDataLoader(
            self.conf.data_dir+self.conf.test_data, self.input_shape)
    predictions = []
    while True:
        inputs, labels = test_reader.next_batch(self.conf.batch)
        if inputs.shape[0] < self.conf.batch:
            break
        feed_dict = {self.inputs: inputs, self.labels: labels}
        predictions.append(self.sess.run(
            self.decoded_preds, feed_dict=feed_dict))
    print('----->saving predictions')
    for index, prediction in enumerate(predictions):
        for i in range(prediction.shape[0]):
            imsave(prediction[i], self.conf.sampledir +
                   str(index*prediction.shape[0]+i)+'.png')

def save(self, step):
    print('---->saving', step)
    checkpoint_path = os.path.join(
        self.conf.modeldir, self.conf.model_name)
    self.saver.save(self.sess, checkpoint_path, global_step=step)

def reload(self, step):
    checkpoint_path = os.path.join(
        self.conf.modeldir, self.conf.model_name)
    model_path = checkpoint_path+'-'+str(step)
    if not os.path.exists(model_path+'.meta'):
        print('------- no such checkpoint', model_path)
        return
    self.saver.restore(self.sess, model_path)
```

4.7.7. main.py

All network hyper parameters have been configured in here.

Training:

max_step: number of iterations or steps to train test_step: number of steps to perform a mini test or validation save_step: number of steps to save the model summary_step: number of steps to save the summary

Data:

data_dir: represents the data directory train_data: creates h5 file for training valid_data: creates h5 file for validation test_data: creates h5 file for testing batch: represents batch size channel:

represents input image channel number height,
width: represents height and width of input
image

Debug:

logdir: defines where to store log modeldir: defines where to store saved models sampledir: defines where to store predicted samples, please add a / at the end for convinience model_name: defines the name prefix of saved models reload_step: defines where to return training test_step: defines which step to test or predict random_seed: defines random seed for tensorflow

Network Architecture:

network_depth: defines how deep of the U-Net including the bottom layer class_num: defines how many classes. Usually number of classes plus one for background start_channel_num: represents the number of channel for the first conv layer conv_name: to use which convolutional layer in decoder. We have conv2d for standard convolutional layer, and ipixel_cl for input pixel convolutional layer proposed in our paper.

deconv_name: to use which upsampling layer in decoder. We have deconv for standard transposed convolutional layer, ipixel_dcl for input pixel transposed convolutional layer, and pixel_dcl for pixel transposed convolutional layer proposed in our paper.

```
import os
import time
import argparse
import tensorflow as tf
from network import PixelDCN

def configure():
    # training
    flags = tf.app.flags
    flags.DEFINE_integer('max_step', 6, '# of step for training')
    flags.DEFINE_integer('test_interval', 100, '# of interval to test a model')
    flags.DEFINE_integer('save_interval', 2, '# of interval to save  model')
    flags.DEFINE_integer('summary_interval', 100, '# of step to save summary')
    flags.DEFINE_float('learning_rate', 1e-3, 'learning rate')
    # data
    flags.DEFINE_string('data_dir', './dataset/', 'Name of data directory')
    flags.DEFINE_string('train_data', 'training3d.h5', 'Training data')
    flags.DEFINE_string('valid_data', 'validation3d.h5', 'Validation data')
    flags.DEFINE_string('test_data', 'testing3d.h5', 'Testing data')
    flags.DEFINE_string('data_type', '3D', '2D data or 3D data')
    flags.DEFINE_integer('batch', 2, 'batch size')
    flags.DEFINE_integer('channel', 1, 'channel size')
    flags.DEFINE_integer('depth', 16, 'depth size')
    flags.DEFINE_integer('height', 256, 'height size')
    flags.DEFINE_integer('width', 256, 'width size')
    # Debug
    flags.DEFINE_string('logdir', './logdir', 'Log dir')
    flags.DEFINE_string('modeldir', './modeldir', 'Model dir')
    flags.DEFINE_string('sampledir', './samples/', 'Sample directory')
    flags.DEFINE_string('model_name', 'model', 'Model file name')
    flags.DEFINE_integer('reload_step', 0, 'Reload step to continue training')
    flags.DEFINE_integer('test_step', 0, 'Test or predict model at this step')
    flags.DEFINE_integer('random_seed', int(time.time()), 'random seed')
    # network architecture
    flags.DEFINE_integer('network_depth', 5, 'network depth for U-Net')
    flags.DEFINE_integer('class_num', 2, 'output class number')
    flags.DEFINE_integer('start_channel_num', 16,
                         'start number of outputs for the first conv layer')
    flags.DEFINE_string(
        'conv_name', 'conv',
        'Use which conv op in decoder: conv or ipixel_cl')
    flags.DEFINE_string(
        'deconv_name', 'ipixel_dcl',
        'Use which deconv op in decoder: deconv, pixel_dcl, ipixel_dcl')
    flags.DEFINE_string(
        'action', 'concat',
        'Use how to combine feature maps in pixel_dcl and ipixel_dcl: concat or add')
    # fix bug of flags
    flags.FLAGS.__dict__['__parsed'] = False
    return flags.FLAGS
```

5. TESTING AND VALIDATION

This study attempts to develop a system for the semantic segmentation for solving the checkerboard artifacts using pixel transposed convolutional network.

The dataset used for the study contains images of both natural JPEG images and also its corresponding segmentation class augmented images.

Result:

- **Input image:**

- **Obtained image:**

6. RESULT

The effect of sample segmentation of U-Net the use of TCL, PixelTCL at the PASCAL 2012 segmentation dataset. We can have a look at that model using PixelTCL can be better in capturing the nearby facts of the images than the same base model using normal TCLs. By using Pixel TCLs, greater spacial-functions along with edges, shapes are taken into consideration while predicting the labels of adjoining pixels. Moreover, the semantic image segmentation results show that the proposed models have a tendency to provide smooth outputs when compared to the model using transposed convolution layer. We additionally examine that, while the training epoch is small, the model that employs Pixel TCL has higher segmentation outputs than the model using iPixel TCL. When the training epoch is massive enough (e.g., one hundred epochs), they've comparable performance, even though Pixel TCL nevertheless outperforms iPixel TCL in maximum cases. This shows that Pixel TCL is greater efficient and effective, because it has plenty fewer parameters to learn.

7. CONCLUSION

In this study, we proposed the Pixel TCL which can overcome the checkerboard problem raised in DCLs. The problem of checkerboard is occurred because no direct-relationship amongst the intermediate featured maps being generated in DCLs. Pixel TCL introduced right here attempts to add direct dependencies amongst those generated intermediate featured maps. Pixel TCL generates intermediate featured maps in a sequential manner in order that the intermediate featured maps are generated in a later level, required to rely upon the one generated previously. The status quo of dependencies in Pixel TCL is capable to make certain adjoining pixels on output featured maps are directly related. Experimental results on semantic segmentation and the tasks of image generation displays that the Pixel TCL is powerful in order to overcome the problem of checkerboard. Outcome of the semantic image segmentation additionally displays that Pixel TCL is capable of recalling nearby spatial functions along with edges, shapes, which lead to more precise segmented outcomes. In future, the Pixel TCL can be made more effective in a broader class of models, along with other effective models such as generative antagonistic networks (GANs).

8. REFERENCES

- https://ieeexplore.ieee.org/document/8618415
- https://en.wikipedia.org/wiki/Image_segmentation
- https://nanonets.com/blog/semantic-image-segmentation-2020/
- https://towardsdatascience.com/semantic-segmentation-with-deep-learning
- https://towardsdatascience.com/
- https://medium.com/
- https://towardsdatascience.com/what-is-transposed-convolutional-layer-